Example: Enumerate Hidden Markov Model 459
Example: CJS Capture-Recapture Model for Ecological Data 467
Example: Nested Sampling for Gaussian Shells 475
Bayesian Imputation for Missing Values in Discrete Covariates 479
Time Series Forecasting 491
Ordinal Regression 499
Bayesian Imputation 505
Example: Gaussian Process 517
Example: Bayesian Neural Network 523
Example: AutoDAIS 527
Example: Sparse Regression 533
Example: Horseshoe Regression 541
Example: Proportion Test 545
Example: Generalized Linear Mixed Models 549
Example: Hidden Markov Model 553
Hilbert space approximation for Gaussian processes. 561
Hilbert space approximation for Gaussian processes (multidimensional) 573
Example: Predator-Prey Model 595
Solving differential equations (ODEs) for multiple initial conditions. 599
Example: Neural Transport 607
Example: Thompson sampling for Bayesian Optimization with GPs 613
Bayesian Hierarchical Stacking: Well Switching Case Study 621
Example: Sine-skewed sine (bivariate von Mises) mixture 635
Example: AR2 process 643
Example: Holt-Winters Exponential Smoothing 647
Example: Modelling mortality over space and time 653
Example: Zero-Inflated Poisson regression model 659
Example: Conditional Variational Autoencoder in Flax 663
Text-Based Ideal Points using NumPyro 665
Example: MCMC Methods for Tall Data 677
1.1 What is NumPyro?

NumPyro is a lightweight probabilistic programming library that provides a NumPy backend for Pyro. We rely on JAX for automatic differentiation and JIT compilation to GPU / CPU. NumPyro is under active development, so beware of brittleness, bugs, and changes to the API as the design evolves.

NumPyro is designed to be lightweight and focuses on providing a flexible substrate that users can build on:

- **Pyro Primitives**: NumPyro programs can contain regular Python and NumPy code, in addition to Pyro primitives like `sample` and `param`. The model code should look very similar to Pyro except for some minor differences between PyTorch and Numpy’s API. See the example below.

- **Inference algorithms**: NumPyro supports a number of inference algorithms, with a particular focus on MCMC algorithms like Hamiltonian Monte Carlo, including an implementation of the No U-Turn Sampler. Additional MCMC algorithms include MixedHMC (which can accommodate discrete latent variables) as well as HMCECS (which only computes the likelihood for subsets of the data in each iteration). One of the motivations for NumPyro was to speed up Hamiltonian Monte Carlo by JIT compiling the verlet integrator that includes multiple gradient computations. With JAX, we can compose `jit` and `grad` to compile the entire integration step into an XLA optimized kernel. We also eliminate Python overhead by JIT compiling the entire tree building stage in NUTS (this is possible using Iterative NUTS). There is also a basic Variational Inference implementation together with many flexible (auto)guides for Automatic Differentiation Variational Inference (ADVI). The variational inference implementation supports a number of features, including support for models with discrete latent variables (see TraceGraph_ELBO and TraceEnum_ELBO).

- **Distributions**: The `numpyro.distributions` module provides distribution classes, constraints and bijective transforms. The distribution classes wrap over samplers implemented to work with JAX’s functional pseudo-random number generator. The design of the distributions module largely follows from PyTorch. A major subset of the API is implemented, and it contains most of the common distributions that exist in PyTorch. As a result, Pyro and PyTorch users can rely on the same API and batching semantics as in `torch.distributions`. In addition to distributions, constraints and transforms are very useful when operating on distribution classes with bounded support. Finally, distributions from TensorFlow Probability (TFP) can directly be used in NumPyro models.

- **Effect handlers**: Like Pyro, primitives like `sample` and `param` can be provided nonstandard interpretations using effect-handlers from the `numpyro.handlers` module, and these can be easily extended to implement custom inference algorithms and inference utilities.
1.2 A Simple Example - 8 Schools

Let us explore NumPyro using a simple example. We will use the eight schools example from Gelman et al., Bayesian Data Analysis: Sec. 5.5, 2003, which studies the effect of coaching on SAT performance in eight schools.

The data is given by:

```python
>>> import numpy as np

>>> J = 8
>>> y = np.array([28.0, 8.0, -3.0, 7.0, -1.0, 1.0, 18.0, 12.0])
>>> sigma = np.array([15.0, 10.0, 16.0, 11.0, 9.0, 11.0, 10.0, 18.0])
```

where `y` are the treatment effects and `sigma` the standard error. We build a hierarchical model for the study where we assume that the group-level parameters `theta` for each school are sampled from a Normal distribution with unknown mean `mu` and standard deviation `tau`, while the observed data are in turn generated from a Normal distribution with mean and standard deviation given by `theta` (true effect) and `sigma`, respectively. This allows us to estimate the population-level parameters `mu` and `tau` by pooling from all the observations, while still allowing for individual variation amongst the schools using the group-level `theta` parameters.

```python
>>> import numpyro
>>> import numpyro.distributions as dist

>>> # Eight Schools example
... def eight_schools(J, sigma, y=None):
...     mu = numpyro.sample('mu', dist.Normal(0, 5))
...     tau = numpyro.sample('tau', dist.HalfCauchy(5))
...     with numpyro.plate('J', J):
...         theta = numpyro.sample('theta', dist.Normal(mu, tau))
...         numpyro.sample('obs', dist.Normal(theta, sigma), obs=y)
```

Let us infer the values of the unknown parameters in our model by running MCMC using the No-U-Turn Sampler (NUTS). Note the usage of the `extra_fields` argument in `MCMC.run`. By default, we only collect samples from the target (posterior) distribution when we run inference using `MCMC`. However, collecting additional fields like potential energy or the acceptance probability of a sample can be easily achieved by using the `extra_fields` argument. For a list of possible fields that can be collected, see the `HMCState` object. In this example, we will additionally collect the `potential_energy` for each sample.
```python
>>> nuts_kernel = NUTS(eight_schools)

>>> mcmc = MCMC(nuts_kernel, num_warmup=500, num_samples=1000)

>>> rng_key = random.PRNGKey(0)

>>> mcmc.run(rng_key, J, sigma, y=y, extra_fields=('potential_energy',))

We can print the summary of the MCMC run, and examine if we observed any divergences during inference. Additionally, since we collected the potential energy for each of the samples, we can easily compute the expected log joint density.

```python
>>> mcmc.print_summary()

```

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>mu</td>
<td>4.14</td>
<td>3.18</td>
<td>3.87</td>
<td>-0.76</td>
<td>9.50</td>
<td>115.42</td>
<td>1.01</td>
</tr>
<tr>
<td>tau</td>
<td>4.12</td>
<td>3.58</td>
<td>3.12</td>
<td>0.51</td>
<td>8.56</td>
<td>90.64</td>
<td>1.02</td>
</tr>
<tr>
<td>theta[0]</td>
<td>6.40</td>
<td>6.22</td>
<td>5.36</td>
<td>-2.54</td>
<td>15.27</td>
<td>176.75</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[1]</td>
<td>4.96</td>
<td>5.04</td>
<td>4.49</td>
<td>-1.98</td>
<td>14.22</td>
<td>217.12</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[2]</td>
<td>3.65</td>
<td>5.41</td>
<td>3.31</td>
<td>-3.47</td>
<td>13.77</td>
<td>247.64</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[3]</td>
<td>4.47</td>
<td>5.29</td>
<td>4.00</td>
<td>-3.22</td>
<td>12.92</td>
<td>213.36</td>
<td>1.01</td>
</tr>
<tr>
<td>theta[6]</td>
<td>6.55</td>
<td>5.72</td>
<td>5.66</td>
<td>-1.43</td>
<td>15.78</td>
<td>124.57</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 19

```python
>>> pe = mcmc.get_extra_fields()['potential_energy']

>>> print('Expected log joint density: {:.2f}'.format(np.mean(-pe)))

Expected log joint density: -54.55
The values above 1 for the split Gelman Rubin diagnostic (\(r_{\text{hat}}\)) indicates that the chain has not fully converged. The low value for the effective sample size (\(n_{\text{eff}}\)), particularly for \(\tau\), and the number of divergent transitions looks problematic. Fortunately, this is a common pathology that can be rectified by using a non-centered parameterization for \(\tau\) in our model. This is straightforward to do in NumPyro by using a TransformedDistribution instance together with a reparameterization effect handler. Let us rewrite the same model but instead of sampling \(\theta\) from a Normal(\(\mu, \tau\)), we will instead sample it from a base Normal(0, 1) distribution that is transformed using an AffineTransform. Note that by doing so, NumPyro runs HMC by generating samples \(\theta_{\text{base}}\) for the base Normal(0, 1) distribution instead. We see that the resulting chain does not suffer from the same pathology — the Gelman Rubin diagnostic is 1 for all the parameters and the effective sample size looks quite good!

```python
>>> from numpyro.infer.reparam import TransformReparam

>>> # Eight Schools example - Non-centered Reparametrization
... def eight_schools_noncentered(J, sigma, y=None):
...     mu = numpyro.sample('mu', dist.Normal(0, 5))
...     tau = numpyro.sample('tau', dist.HalfCauchy(5))
...     with numpyro.plate('J', J):
...         with numpyro.handlers.reparam(config={'theta': TransformReparam()}):
...             theta = numpyro.sample('theta',
...                 dist.TransformedDistribution(dist.Normal(0., 1.),
...                 dist.transforms.AffineTransform(mu, tau))
...             numpyro.sample('obs', dist.Normal(theta, sigma), obs=y)

>>> nuts_kernel = NUTS(eight_schools_noncentered)

>>> mcmc = MCMC(nuts_kernel, num_warmup=500, num_samples=1000)

>>> rng_key = random.PRNGKey(0)

>>> mcmc.run(rng_key, J, sigma, y=y, extra_fields=('potential_energy',))

>>> mcmc.print_summary(exclude_deterministic=False)
```

<table>
<thead>
<tr>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
</table>

(continues on next page)
NumPyro Documentation

(continued from previous page)

<table>
<thead>
<tr>
<th>mu</th>
<th>4.08</th>
<th>3.51</th>
<th>4.14</th>
<th>-1.69</th>
<th>9.71</th>
<th>720.43</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>tau</td>
<td>3.96</td>
<td>3.31</td>
<td>3.09</td>
<td>0.01</td>
<td>8.34</td>
<td>488.63</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[0]</td>
<td>6.48</td>
<td>5.72</td>
<td>6.08</td>
<td>-2.53</td>
<td>14.96</td>
<td>801.59</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[1]</td>
<td>4.95</td>
<td>5.10</td>
<td>4.91</td>
<td>-3.70</td>
<td>12.82</td>
<td>1183.06</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[2]</td>
<td>3.65</td>
<td>5.58</td>
<td>3.72</td>
<td>-5.71</td>
<td>12.13</td>
<td>581.31</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[3]</td>
<td>4.56</td>
<td>5.04</td>
<td>4.32</td>
<td>-3.14</td>
<td>12.92</td>
<td>1282.60</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[4]</td>
<td>3.41</td>
<td>4.79</td>
<td>3.47</td>
<td>-4.16</td>
<td>10.79</td>
<td>801.25</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[5]</td>
<td>3.58</td>
<td>4.80</td>
<td>3.78</td>
<td>-3.95</td>
<td>11.55</td>
<td>1101.33</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[6]</td>
<td>6.31</td>
<td>5.17</td>
<td>5.75</td>
<td>-2.93</td>
<td>13.87</td>
<td>1081.11</td>
<td>1.00</td>
</tr>
<tr>
<td>theta[7]</td>
<td>4.81</td>
<td>5.38</td>
<td>4.61</td>
<td>-3.29</td>
<td>14.05</td>
<td>954.14</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[0]</td>
<td>0.41</td>
<td>0.95</td>
<td>0.40</td>
<td>-1.09</td>
<td>1.95</td>
<td>851.45</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[1]</td>
<td>0.15</td>
<td>0.95</td>
<td>0.20</td>
<td>-1.42</td>
<td>1.66</td>
<td>1568.11</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[2]</td>
<td>-0.08</td>
<td>0.98</td>
<td>-0.10</td>
<td>-1.68</td>
<td>1.54</td>
<td>1037.16</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[3]</td>
<td>0.06</td>
<td>0.89</td>
<td>0.05</td>
<td>-1.42</td>
<td>1.47</td>
<td>1745.02</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[4]</td>
<td>-0.14</td>
<td>0.94</td>
<td>-0.16</td>
<td>-1.65</td>
<td>1.45</td>
<td>719.85</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[5]</td>
<td>-0.10</td>
<td>0.96</td>
<td>-0.14</td>
<td>-1.57</td>
<td>1.51</td>
<td>1128.45</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[6]</td>
<td>0.38</td>
<td>0.95</td>
<td>0.42</td>
<td>-1.32</td>
<td>1.82</td>
<td>1026.50</td>
<td>1.00</td>
</tr>
<tr>
<td>theta_base[7]</td>
<td>0.10</td>
<td>0.97</td>
<td>0.10</td>
<td>-1.51</td>
<td>1.65</td>
<td>1190.98</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 0

```python
>>> pe = mcmc.get_extra_fields()['potential_energy']

>>> # Compare with the earlier value

>>> print('Expected log joint density: {:.2f}'.format(np.mean(-pe)))

Expected log joint density: -46.09
```

Note that for the class of distributions with `loc`, `scale` parameters such as `Normal`, `Cauchy`, `StudentT`, we also provide a `LocScaleReparam` reparameterizer to achieve the same purpose. The corresponding code will be

1.2. A Simple Example - 8 Schools 5
with numpyro.handlers.reparam(config={"theta": LocScaleReparam(centered=0)}):

    theta = numpyro.sample('theta', dist.Normal(mu, tau))

Now, let us assume that we have a new school for which we have not observed any test scores, but we would like to generate predictions. NumPyro provides a `Predictive` class for such a purpose. Note that in the absence of any observed data, we simply use the population-level parameters to generate predictions. The `Predictive` utility conditions the unobserved `mu` and `tau` sites to values drawn from the posterior distribution from our last MCMC run, and runs the model forward to generate predictions.

```python
>>> from numpyro.infer import Predictive

>>> # New School
... def new_school():
...     return numpyro.sample('obs', dist.Normal(mu, tau))

>>> predict = Predictive(new_school, mcmc.get_samples())

>>> samples_predictive = predict(random.PRNGKey(1))

>>> print(np.mean(samples_predictive['obs']))
3.9886456
```

### 1.3 More Examples

For some more examples on specifying models and doing inference in NumPyro:

- **Bayesian Regression in NumPyro** - Start here to get acquainted with writing a simple model in NumPyro, MCMC inference API, effect handlers and writing custom inference utilities.
- **Time Series Forecasting** - Illustrates how to convert for loops in the model to JAX's `lax.scan` primitive for fast inference.
- **Annotation examples** - Illustrates how to utilize the enumeration mechanism to perform inference for models with discrete latent variables.
- **Baseball example** - Using NUTS for a simple hierarchical model. Compare this with the baseball example in Pyro.
- **Hidden Markov Model in NumPyro as compared to Stan.**
- **Variational Autoencoder** - As a simple example that uses Variational Inference with neural networks. Pyro implementation for comparison.
• **Gaussian Process** - Provides a simple example to use NUTS to sample from the posterior over the hyper-parameters of a Gaussian Process.

• **Horseshoe Regression** - Shows how to implement general linear models equipped with a Horseshoe prior for both binary-valued and real-valued outputs.

• **Statistical Rethinking with NumPyro** - Notebooks containing translation of the code in Richard McElreath’s *Statistical Rethinking* book second version, to NumPyro.

• Other model examples can be found in the examples site.

Pyro users will note that the API for model specification and inference is largely the same as Pyro, including the distributions API, by design. However, there are some important core differences (reflected in the internals) that users should be aware of. e.g. in NumPyro, there is no global parameter store or random state, to make it possible for us to leverage JAX’s JIT compilation. Also, users may need to write their models in a more functional style that works better with JAX. Refer to FAQs for a list of differences.

### 1.4 Overview of inference algorithms

We provide an overview of most of the inference algorithms supported by NumPyro and offer some guidelines about which inference algorithms may be appropriate for different classes of models.

#### 1.4.1 MCMC

• **NUTS**, which is an adaptive variant of HMC, is probably the most commonly used inference algorithm in NumPyro. Note that NUTS and HMC are not directly applicable to models with discrete latent variables, but in cases where the discrete variables have finite support and summing them out (i.e. enumeration) is tractable, NumPyro will automatically sum out discrete latent variables and perform NUTS/HMC on the remaining continuous latent variables.

As discussed above, model reparameterization may be important in some cases to get good performance. Note that, generally speaking, we expect inference to be harder as the dimension of the latent space increases. See the bad geometry tutorial for additional tips and tricks.

• **MixedHMC** can be an effective inference strategy for models that contain both continuous and discrete latent variables.

• **HMCECS** can be an effective inference strategy for models with a large number of data points. It is applicable to models with continuous latent variables. See here for an example.

• **BarkerMH** is a gradient-based MCMC method that may be competitive with HMC and NUTS for some models. It is applicable to models with continuous latent variables.

• **HMCGibbs** combines HMC/NUTS steps with custom Gibbs updates. Gibbs updates must be specified by the user.

• **DiscreteHMCGibbs** combines HMC/NUTS steps with Gibbs updates for discrete latent variables. The corresponding Gibbs updates are computed automatically.

• **SA** is the only MCMC method in NumPyro that does not leverage gradients. It is only applicable to models with continuous latent variables. It is expected to perform best for models whose latent dimension is low to moderate. It may be a good choice for models with non-differentiable log densities. Note that SA generally requires a very large number of samples, as mixing tends to be slow. On the plus side individual steps can be fast.

Like HMC/NUTS, all remaining MCMC algorithms support enumeration over discrete latent variables if possible (see restrictions). Enumerated sites need to be marked with `infer={'enumerate': 'parallel'}` like in the annotation example.
1.4.2 Nested Sampling

- NestedSampler offers a wrapper for jaxns. See JAXNS’s readthedocs for examples and Nested Sampling for Gaussian Shells example for how to apply the sampler on numpyro models. Can handle arbitrary models, including ones with discrete RVs, and non-invertible transformations.

1.4.3 Stochastic variational inference

- Variational objectives
  - Trace_ELBO is our basic ELBO implementation.
  - TraceMeanField_ELBO is like Trace_ELBO but computes part of the ELBO analytically if doing so is possible.
  - TraceGraph_ELBO offers variance reduction strategies for models with discrete latent variables. Generally speaking, this ELBO should always be used for models with discrete latent variables.
  - TraceEnum_ELBO offers variable enumeration strategies for models with discrete latent variables. Generally speaking, this ELBO should always be used for models with discrete latent variables when enumeration is possible.
- Automatic guides (appropriate for models with continuous latent variables)
  - AutoNormal and AutoDiagonalNormal are our basic mean-field guides. If the latent space is non-euclidean (due to e.g. a positivity constraint on one of the sample sites) an appropriate bijective transformation is automatically used under the hood to map between the unconstrained space (where the Normal variational distribution is defined) to the corresponding constrained space (note this is true for all automatic guides). These guides are a great place to start when trying to get variational inference to work on a model you are developing.
  - AutoMultivariateNormal and AutoLowRankMultivariateNormal also construct Normal variational distributions but offer more flexibility, as they can capture correlations in the posterior. Note that these guides may be difficult to fit in the high-dimensional setting.
  - AutoDelta is used for computing point estimates via MAP (maximum a posteriori estimation). See here for example usage.
  - AutoBNAFNormal and AutoIAFNormal offer flexible variational distributions parameterized by normalizing flows.
  - AutoDAIS is a powerful variational inference algorithm that leverages HMC. It can be a good choice for dealing with highly correlated posteriors but may be computationally expensive depending on the nature of the model.
  - AutoSurrogateLikelihoodDAIS is a powerful variational inference algorithm that leverages HMC and that supports data subsampling.
  - AutoSemiDAIS constructs a posterior approximation like AutoDAIS for local latent variables but provides support for data subsampling during ELBO training by utilizing a parametric guide for global latent variables.
  - AutoLaplaceApproximation can be used to compute a Laplace approximation.
1.4.4 Stein Variational Inference

See the docs for more details.

1.5 Installation

**Limited Windows Support:** Note that NumPyro is untested on Windows, and might require building jaxlib from source. See this JAX issue for more details. Alternatively, you can install Windows Subsystem for Linux and use NumPyro on it as on a Linux system. See also CUDA on Windows Subsystem for Linux and this forum post if you want to use GPUs on Windows.

To install NumPyro with the latest CPU version of JAX, you can use pip:

```
pip install numpyro
```

In case of compatibility issues arise during execution of the above command, you can instead force the installation of a known compatible CPU version of JAX with

```
pip install numpyro[cpu]
```

To use NumPyro on the GPU, you need to install CUDA first and then use the following pip command:

```
```

If you need further guidance, please have a look at the JAX GPU installation instructions.

To run NumPyro on Cloud TPU, you can look at some JAX on Cloud TPU examples.

For Cloud TPU VM, you need to setup the TPU backend as detailed in the Cloud TPU VM JAX Quickstart Guide. After you have verified that the TPU backend is properly set up, you can install NumPyro using the pip install numpyro command.

**Default Platform:** JAX will use GPU by default if CUDA-supported jaxlib package is installed. You can use set_platform utility `numpyro.set_platform("cpu")` to switch to CPU at the beginning of your program.

You can also install NumPyro from source:

```
git clone https://github.com/pyro-ppl/numpyro.git

cd numpyro

# install jax/jaxlib first for CUDA support
pip install -e .[dev]  # contains additional dependencies for NumPyro development
```

You can also install NumPyro with conda:

```
conda install -c conda-forge numpyro
```
1.6 Frequently Asked Questions

1. Unlike in Pyro, `numpyro.sample('x', dist.Normal(0, 1))` does not work. Why?

   You are most likely using a `numpyro.sample` statement outside an inference context. JAX does not have a global random state, and as such, distribution samplers need an explicit random number generator key (PRNGKey) to generate samples from. NumPyro’s inference algorithms use the `seed` handler to thread in a random number generator key, behind the scenes.

   Your options are:
   - Call the distribution directly and provide a PRNGKey, e.g. `dist.Normal(0, 1).sample(PRNGKey(0))`.
   - Provide the `rng_key` argument to `numpyro.sample`. e.g. `numpyro.sample('x', dist.Normal(0, 1), rng_key=PRNGKey(0))`.
   - Wrap the code in a `seed` handler, used either as a context manager or as a function that wraps over the original callable. e.g.

     ```python
     with handlers.seed(rng_seed=0):  # random.PRNGKey(0) is used
         x = numpyro.sample('x', dist.Beta(1, 1))  # uses a PRNGKey split from random.PRNGKey(0)
         y = numpyro.sample('y', dist.Bernoulli(x))  # uses different PRNGKey split from the last one
     ```

2. Can I use the same Pyro model for doing inference in NumPyro?

   As you may have noticed from the examples, NumPyro supports all Pyro primitives like `sample, param, plate and module`, and effect handlers. Additionally, we have ensured that the `distributions` API is based on `torch.distributions`, and the inference classes like `SVI` and `MCMC` have the same interface. This along with the similarity in the API for NumPy and PyTorch operations ensures that models containing Pyro primitive statements can be used with either backend with some minor changes. Example of some differences along with the changes needed, are noted below:

   - Any `torch` operation in your model will need to be written in terms of the corresponding `jax.numpy` operation. Additionally, not all `torch` operations have a `numpy` counterpart (and vice-versa), and sometimes there are minor differences in the API.
   - `pyro.sample` statements outside an inference context will need to be wrapped in a `seed` handler, as mentioned above.
• There is no global parameter store, and as such using `numpyro.param` outside an inference context will have no effect. To retrieve the optimized parameter values from SVI, use the `SVI.get_params` method. Note that you can still use `param` statements inside a model and NumPyro will use the substitute effect handler internally to substitute values from the optimizer when running the model in SVI.

• PyTorch neural network modules will need to rewritten as stax, flax, or haiku neural networks. See the VAE and ProdLDA examples for differences in syntax between the two backends.

• JAX works best with functional code, particularly if we would like to leverage JIT compilation, which NumPyro does internally for many inference subroutines. As such, if your model has side-effects that are not visible to the JAX tracer, it may need to rewritten in a more functional style.

For most small models, changes required to run inference in NumPyro should be minor. Additionally, we are working on `pyro-api` which allows you to write the same code and dispatch it to multiple backends, including NumPyro. This will necessarily be more restrictive, but has the advantage of being backend agnostic. See the documentation for an example, and let us know your feedback.

3. How can I contribute to the project?

Thanks for your interest in the project! You can take a look at beginner friendly issues that are marked with the good first issue tag on Github. Also, please feel to reach out to us on the forum.

1.7 Future / Ongoing Work

In the near term, we plan to work on the following. Please open new issues for feature requests and enhancements:

• Improving robustness of inference on different models, profiling and performance tuning.

• Supporting more functionality as part of the `pyro-api` generic modeling interface.

• More inference algorithms, particularly those that require second order derivatives or use HMC.

• Integration with Funsor to support inference algorithms with delayed sampling.

• Other areas motivated by Pyro’s research goals and application focus, and interest from the community.

1.8 Citing NumPyro

The motivating ideas behind NumPyro and a description of Iterative NUTS can be found in this paper that appeared in NeurIPS 2019 Program Transformations for Machine Learning Workshop.

If you use NumPyro, please consider citing:

```bibtex
@article{phan2019composable,
  title={Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro},
  author={Phan, Du and Pradhan, Neeraj and Jankowiak, Martin},
  year={2019}
}
```
as well as

@article{bingham2019pyro,
  author = {Eli Bingham and
            Jonathan P. Chen and
            Martin Jankowiak and
            Fritz Obermeyer and
            Neeraj Pradhan and
            Theofanis Karaletsos and
            Rohit Singh and
            Paul A. Szerlip and
            Paul Horsfall and
            Noah D. Goodman},
  title = {Pyro: Deep Universal Probabilistic Programming},
  journal = {J. Mach. Learn. Res.},
  volume = {20},
  pages = {28:1--28:6},
  year = {2019},
  url = {http://jmlr.org/papers/v20/18-403.html}
}
2.1 param

```
param(name, init_value=None, **kwargs)
```

Annotate the given site as an optimizable parameter for use with `jax.example_libraries.optimizers`. For an example of how `param` statements can be used in inference algorithms, refer to SVI.

**Parameters**

- `name (str)` – name of site.
- `init_value (jnp.ndarray or callable)` – initial value specified by the user or a lazy callable that accepts a JAX random PRNGKey and returns an array. Note that the onus of using this to initialize the optimizer is on the user inference algorithm, since there is no global parameter store in NumPyro.
- `constraint (numpyro.distributions.constraints.Constraint)` – NumPyro constraint, defaults to `constraints.real`.
- `event_dim (int)` – (optional) number of rightmost dimensions unrelated to batching. Dimension to the left of this will be considered batch dimensions; if the param statement is inside a subsampled plate, then corresponding batch dimensions of the parameter will be correspondingly subsampled. If unspecified, all dimensions will be considered event dims and no subsampling will be performed.

**Returns**

value for the parameter. Unless wrapped inside a handler like `substitute`, this will simply return the initial value.

2.2 sample

```
sample(name, fn, obs=None, rng_key=None, sample_shape=(), infer=None, obs_mask=None)
```

Returns a random sample from the stochastic function `fn`. This can have additional side effects when wrapped inside effect handlers like `substitute`.

**Note**

By design, `sample` primitive is meant to be used inside a NumPyro model. Then `seed` handler is used to inject a random state to `fn`. In those situations, `rng_key` keyword will take no effect.

**Parameters**
• **name** *(str)* – name of the sample site.
• **fn** – a stochastic function that returns a sample.
• **obs** *(jnp.ndarray)* – observed value
• **rng_key** *(jax.random.PRNGKey)* – an optional random key for fn.
• **sample_shape** – Shape of samples to be drawn.
• **infer** *(dict)* – an optional dictionary containing additional information for inference algorithms. For example, if fn is a discrete distribution, setting infer={'enumerate': 'parallel'} to tell MCMC marginalize this discrete latent site.
• **obs_mask** *(jnp.ndarray)* – Optional boolean array mask of shape broadcastable with fn. batch_shape. If provided, events with mask=True will be conditioned on obs and remaining events will be imputed by sampling. This introduces a latent sample site named name + "_unobserved" which should be used by guides in SVI. Note that this argument is not intended to be used with MCMC.

**Returns**

sample from the stochastic fn.

### 2.3 plate

**class plate** *(name, size, subsample_size=None, dim=None)*

Construct for annotating conditionally independent variables. Within a plate context manager, sample sites will be automatically broadcasted to the size of the plate. Additionally, a scale factor might be applied by certain inference algorithms if subsample_size is specified.

#### Note

This can be used to subsample minibatches of data:

```python
with plate("data", len(data), subsample_size=100) as ind:
    batch = data[ind]
    assert len(batch) == 100
```

**Parameters**

• **name** *(str)* – Name of the plate.
• **size** *(int)* – Size of the plate.
• **subsample_size** *(int)* – Optional argument denoting the size of the mini-batch. This can be used to apply a scaling factor by inference algorithms. e.g. when computing ELBO using a mini-batch.
• **dim** *(int)* – Optional argument to specify which dimension in the tensor is used as the plate dim. If None (default), the rightmost available dim is allocated.
### 2.4 plate_stack

**plate_stack**(*prefix, sizes, rightmost_dim=−1*)

Create a contiguous stack of *plate* s with dimensions:

```
rightmost_dim - len(sizes), ..., rightmost_dim
```

**Parameters**

- **prefix** *(str)* – Name prefix for plates.
- **sizes** *(iterable)* – An iterable of plate sizes.
- **rightmost_dim** *(int)* – The rightmost dim, counting from the right.

### 2.5 subsample

**subsample**(*data, event_dim*)

EXPERIMENTAL Subsampling statement to subsample data based on enclosing *plate* s.

This is typically called on arguments to `model()` when subsampling is performed automatically by *plate* s by passing `subsample_size` kwarg. For example the following are equivalent:

```python
# Version 1. using indexing
def model(data):
    with numpyro.plate("data", len(data), subsample_size=10, dim=-data.dim()) as ind:
        data = data[ind]
        # ...

# Version 2. using numpyro.subsample()
def model(data):
    with numpyro.plate("data", len(data), subsample_size=10, dim=-data.dim()):
        data = numpyro.subsample(data, event_dim=0)
        # ...
```

**Parameters**

- **data** *(jnp.ndarray)* – A tensor of batched data.
- **event_dim** *(int)* – The event dimension of the data tensor. Dimensions to the left are considered batch dimensions.

**Returns**

A subsampled version of `data`

**Return type**

`ndarray`
2.6 deterministic

deterministic(name, value)

Used to designate deterministic sites in the model. Note that most effect handlers will not operate on deterministic sites (except trace()), so deterministic sites should be side-effect free. The use case for deterministic nodes is to record any values in the model execution trace.

Parameters

- name (str) – name of the deterministic site.
- value (jnp.ndarray) – deterministic value to record in the trace.

2.7 prng_key

prng_key()

A statement to draw a pseudo-random number generator key PRNGKey() under seed handler.

Returns

a PRNG key of shape (2,) and dtype unit32.

2.8 factor

factor(name, log_factor)

Factor statement to add arbitrary log probability factor to a probabilistic model.

Parameters

- name (str) – Name of the trivial sample.
- log_factor (jnp.ndarray) – A possibly batched log probability factor.

2.9 get_mask

get_mask()

Records the effects of enclosing handlers.mask handlers. This is useful for avoiding expensive numpyro.factor() computations during prediction, when the log density need not be computed, e.g.:

```python
def model():
    # ...
    if numpyro.get_mask() is not False:
        log_density = my_expensive_computation()
        numpyro.factor("foo", log_density)
    # ...
```

Returns

The mask.

Return type

None, bool, or jnp.ndarray
2.10 module

```
module(name, nn, input_shape=None)
```

Declare a `stax` style neural network inside a model so that its parameters are registered for optimization via `param()` statements.

Parameters

- **name** *(str)* – name of the module to be registered.
- **nn** *(tuple)* – a tuple of `(init_fn, apply_fn)` obtained by a `stax` constructor function.
- **input_shape** *(tuple)* – shape of the input taken by the neural network.

Returns

A `apply_fn` with bound parameters that takes an array as an input and returns the neural network transformed output array.

2.11 flax_module

```
flax_module(name, nn_module, *args, input_shape=None, apply_rng=None, mutable=None, **kwargs)
```

Declare a `flax` style neural network inside a model so that its parameters are registered for optimization via `param()` statements.

Given a `flax nn_module`, in flax to evaluate the module with a given set of parameters, we use: `nn_module.apply(params, x)`. In a NumPyro model, the pattern will be:

```python
net = flax_module("net", nn_module)
y = net(x)
```

or with dropout layers:

```python
net = flax_module("net", nn_module, apply_rng=["dropout"])
rng_key = numpyro.prng_key()
y = net(x, rngs={"dropout": rng_key})
```

Parameters

- **name** *(str)* – name of the module to be registered.
- **nn_module** *(flax.linen.Module)* – a `flax` Module which has `.init` and `.apply` methods
- **args** – optional arguments to initialize flax neural network as an alternative to `input_shape`
- **input_shape** *(tuple)* – shape of the input taken by the neural network.
- **apply_rng** *(list)* – A list to indicate which extra rng _kinds_ are needed for `nn_module`. For example, when `nn_module` includes dropout layers, we need to set `apply_rng=["dropout"]`. Defaults to None, which means no extra rng key is needed. Please see Flax Linen Intro for more information in how Flax deals with stochastic layers like dropout.
- **mutable** *(list)* – A list to indicate mutable states of `nn_module`. For example, if your module has BatchNorm layer, we will need to define `mutable=["batch_stats"]`. See the above Flax Linen Intro tutorial for more information.
- **kwargs** – optional keyword arguments to initialize flax neural network as an alternative to `input_shape`
Returns
a callable with bound parameters that takes an array as an input and returns the neural network transformed output array.

### 2.12 haiku_module

**haiku_module** *(name, nn_module, *args, input_shape=None, apply_rng=False, **kwargs)*

Declare a haiku style neural network inside a model so that its parameters are registered for optimization via `param()` statements.

Given a haiku `nn_module`, in haiku to evaluate the module with a given set of parameters, we use: `nn_module.apply(params, None, x)`. In a NumPyro model, the pattern will be:

```python
net = haiku_module("net", nn_module)
y = net(x)  # or y = net(rng_key, x)
```

or with dropout layers:

```python
net = haiku_module("net", nn_module, apply_rng=True)
rng_key = numpyro.prng_key()
y = net(rng_key, x)
```

**Parameters**
- **name** *(str)* – name of the module to be registered.
- **nn_module** *(haiku.Transformed or haiku.TransformedWithState)* – a haiku Module which has .init and .apply methods.
- **args** – optional arguments to initialize flax neural network as an alternative to `input_shape`
- **input_shape** *(tuple)* – shape of the input taken by the neural network.
- **apply_rng** *(bool)* – A flag to indicate if the returned callable requires an rng argument (e.g. when `nn_module` includes dropout layers). Defaults to False, which means no rng argument is needed. If this is True, the signature of the returned callable `nn = haiku_module(..., apply_rng=True)` will be `nn(rng_key, x)` (rather than `nn(x)`).
- **kwargs** – optional keyword arguments to initialize flax neural network as an alternative to `input_shape`

**Returns**
a callable with bound parameters that takes an array as an input and returns the neural network transformed output array.

### 2.13 random_flax_module

**random_flax_module** *(name, nn_module, prior, *args, input_shape=None, apply_rng=None, mutable=None, **kwargs)*

A primitive to place a prior over the parameters of the Flax module `nn_module`. 
Parameters of a Flax module are stored in a nested dict. For example, the module \(B\) defined as follows:

```python
class A(flax.linen.Module):
    @flax.linen.compact
    def __call__(self, x):
        return nn.Dense(1, use_bias=False, name='dense')(x)

class B(flax.linen.Module):
    @flax.linen.compact
    def __call__(self, x):
        return A(name='inner')(x)
```

has parameters `{'inner': {'dense': {'kernel': param_value}}}`. In the argument `prior`, to specify `kernel` parameter, we join the path to it using dots: `prior={"inner.dense.kernel": param_prior}`.

### Parameters

- **name** *(str)* – name of NumPyro module
- **flax.linen.Module** – the module to be registered with NumPyro
- **prior** *(dict, Distribution or callable)* – a NumPyro distribution or a Python dict with parameter names as keys and respective distributions as values. For example:

  ```python
  net = random_flax_module("net",
      flax.linen.Dense(features=1),
      prior={"bias": dist.Cauchy(), "kernel": dist.Normal()},
      input_shape=(4,))
  ```

  Alternatively, we can use a callable. For example the following are equivalent:

  ```python
  prior=(lambda name, shape: dist.Cauchy() if name == "bias" else dist.Normal())
  prior={"bias": dist.Cauchy(), "kernel": dist.Normal()}
  ```

- **args** – optional arguments to initialize flax neural network as an alternative to `input_shape`
- **input_shape** *(tuple)* – shape of the input taken by the neural network.
- **apply_rng** *(list)* – A list to indicate which extra rng _kinds_ are needed for `nn_module`. For example, when `nn_module` includes dropout layers, we need to set `apply_rng=["dropout"]`. Defaults to None, which means no extra rng key is needed. Please see Flax Linen Intro for more information in how Flax deals with stochastic layers like dropout.
- **mutable** *(list)* – A list to indicate mutable states of `nn_module`. For example, if your module has BatchNorm layer, we will need to define `mutable=["batch_stats"]`. See the above Flax Linen Intro tutorial for more information.
- **kwargs** – optional keyword arguments to initialize flax neural network as an alternative to `input_shape`

### Returns

a sampled module

### Example
```python
>>> import numpy as np
>>> import tqdm
>>> from flax import linen as nn
>>> from jax import jit, random
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.contrib.module import random_flax_module
>>> from numpyro.infer import Predictive, SVI, TraceMeanField_ELBO, autoguide, init_to_feasible

>>> class Net(nn.Module):
...     n_units: int
...     @nn.compact
...     def __call__(self, x):
...         x = nn.Dense(self.n_units)(x[..., None])
...         x = nn.relu(x)
...         x = nn.Dense(self.n_units)(x)
...         x = nn.relu(x)
...         mean = nn.Dense(1)(x)
...         rho = nn.Dense(1)(x)
...         return mean.squeeze(), rho.squeeze()

>>> def generate_data(n_samples):
...     x = np.random.normal(size=n_samples)
...     y = np.cos(x * 3) + np.random.normal(size=n_samples) * np.abs(x) / 2
...     return x, y

>>> def model(x, y=None, batch_size=None):
...     module = Net(n_units=32)
...     net = random_flax_module("nn", module, dist.Normal(0, 0.1), input_shape=())
...     with numpyro.plate("batch", x.shape[0], subsample_size=batch_size):
...         batch_x = numpyro.subsample(x, event_dim=0) if y is not None else None
...         batch_y = numpyro.subsample(y, event_dim=0)
...         mean, rho = net(batch_x)
...         sigma = nn.softplus(rho)
...         numpyro.sample("obs", dist.Normal(mean, sigma), obs=batch_y)

>>> n_train_data = 5000
>>> x_train, y_train = generate_data(n_train_data)
>>> guide = autoguide.AutoNormal(model, init_loc_fn=init_to_feasible)
>>> svi = SVI(model, guide, numpyro.optim.Adam(5e-3), TraceMeanField_ELBO())
>>> n_iterations = 3000
>>> svi_result = svi.run(random.PRNGKey(0), n_iterations, x_train, y_train, batch_size=256)
>>> params, losses = svi_result.params, svi_result.losses
>>> n_test_data = 100
>>> x_test, y_test = generate_data(n_test_data)
>>> predictive = Predictive(model, guide=guide, params=params, num_samples=1000)
>>> y_pred = predictive(random.PRNGKey(1), x_test[:100])['obs'].copy()
>>> assert losses[-1] < 3000
```

(continues on next page)
>>> assert np.sqrt(np.mean(np.square(y_test - y_pred))) < 1

2.14 random_haiku_module

random_haiku_module(name, nn_module, prior, *args, input_shape=None, apply_rng=False, **kwargs)

A primitive to place a prior over the parameters of the Haiku module nn_module.

Parameters

- **name** *(str)* – name of NumPyro module
- **nn_module** *(haiku.Transformed or haiku.TransformedWithState)* – the module to be registered with NumPyro
- **prior** *(dict, Distribution or callable)* – a NumPyro distribution or a Python dict with parameter names as keys and respective distributions as values. For example:

  ```python
  net = random_haiku_module("net",
    haiku.transform(lambda x: hk.Linear(1)(x)),
    prior={"linear.b": dist.Cauchy(), "linear.w": dist.Normal()},
    input_shape=(4,))
  ```

  Alternatively, we can use a callable. For example the following are equivalent:

  ```python
  prior=(lambda name, shape: dist.Cauchy() if name.startswith("b") else dist.Normal())
  prior={"bias": dist.Cauchy(), "kernel": dist.Normal()}
  ```

- **args** – optional arguments to initialize flax neural network as an alternative to input_shape
- **input_shape** *(tuple)* – shape of the input taken by the neural network.
- **apply_rng** *(bool)* – A flag to indicate if the returned callable requires an rng argument (e.g. when nn_module includes dropout layers). Defaults to False, which means no rng argument is needed. If this is True, the signature of the returned callable `nn = haiku_module(..., apply_rng=True)` will be `nn(rng_key, x)` (rather than `nn(x)`).
- **kwargs** – optional keyword arguments to initialize flax neural network as an alternative to input_shape

Returns

- a sampled module

2.15 scan

scan(f, init, xs, length=None, reverse=False, history=1)

This primitive scans a function over the leading array axes of xs while carrying along state. See `jax.lax.scan()` for more information.

Usage:
NumPyro Documentation

```python
>>> import numpy as np
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.contrib.control_flow import scan

>>> def gaussian_hmm(y=None, T=10):
...     def transition(x_prev, y_curr):
...         x_curr = numpyro.sample('x', dist.Normal(x_prev, 1))
...         y_curr = numpyro.sample('y', dist.Normal(x_curr, 1), obs=y_curr)
...         return x_curr, (x_curr, y_curr)
...     
...     x0 = numpyro.sample('x_0', dist.Normal(0, 1))
...     _, (x, y) = scan(transition, x0, y, length=T)
...     return (x, y)

>>> # here we do some quick tests
>>> with numpyro.handlers.seed(rng_seed=0):
...     x, y = gaussian_hmm(np.arange(10.))
...     assert x.shape == (10,) and y.shape == (10,)
...     assert np.all(y == np.arange(10))

>>> with numpyro.handlers.seed(rng_seed=0):
...     x, y = gaussian_hmm()
...     assert x.shape == (10,) and y.shape == (10,)
```

⚠️ Warning
This is an experimental utility function that allows users to use JAX control flow with NumPyro’s effect handlers. Currently, `sample` and `deterministic` sites within the scan body `f` are supported. If you notice that any effect handlers or distributions are unsupported, please file an issue.

⚠️ Note
It is ambiguous to align `scan` dimension inside a `plate` context. So the following pattern won’t be supported

```python
with numpyro.plate('N', 10):
    last, ys = scan(f, init, xs)
```

All `plate` statements should be put inside `f`. For example, the corresponding working code is

```python
def g(*args, **kwargs):
    with numpyro.plate('N', 10):
        return f(*arg, **kwargs)

last, ys = scan(g, init, xs)
```

⚠️ Note
Nested scan is currently not supported.
We can scan over discrete latent variables in \( f \). The joint density is evaluated using parallel-scan (reference \cite{1}) over time dimension, which reduces parallel complexity to \( O(\log(\text{length})) \).

A trace of scan with discrete latent variables will contain the following sites:

- init sites: those sites belong to the first history traces of \( f \). Sites at the \( i \)-th trace will have name prefixed with \( '_{\text{PREV}}'_ * (2 * \text{history} - 1 - i) \).
- scanned sites: those sites collect the values of the remaining scan loop over \( f \). An addition time dimension \( _\text{time}_\text{foo} \) will be added to those sites, where \( \text{foo} \) is the name of the first site appeared in \( f \).

Not all transition functions \( f \) are supported. All of the restrictions from Pyro’s enumeration tutorial \cite{2} still apply here. In addition, there should not have any site outside of scan depend on the first output of scan (the last carry value).

References


Parameters

- \( f \) (callable) – a function to be scanned.
- \( \text{init} \) – the initial carrying state
- \( \text{xs} \) – the values over which we scan along the leading axis. This can be any JAX pytree (e.g. list/dict of arrays).
- \( \text{length} \) – optional value specifying the length of \( \text{xs} \) but can be used when \( \text{xs} \) is an empty pytree (e.g. None)
- \( \text{reverse} \) (bool) – optional boolean specifying whether to run the scan iteration forward (the default) or in reverse
- \( \text{history} \) (int) – The number of previous contexts visible from the current context. Defaults to 1. If zero, this is similar to \texttt{numpyro.plate}.

Returns

output of scan, quoted from \texttt{jax.lax.scan()} docs: “pair of type (c, [b]) where the first element represents the final loop carry value and the second element represents the stacked outputs of the second output of \( f \) when scanned over the leading axis of the inputs”.

2.15. \texttt{scan}
2.16 cond

\texttt{cond}(\textit{pred, true\_fun, false\_fun, operand})

This primitive conditionally applies \textit{true\_fun} or \textit{false\_fun}. See \texttt{jax.lax.cond()} for more information.

Usage:

```python
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from jax import random
>>> from numpyro.contrib.control_flow import cond
>>> from numpyro.infer import SVI, Trace_ELBO

>>> def model():
...     def true_fun(_):
...         return numpyro.sample("x", dist.Normal(20.0))
...     
...     def false_fun(_):
...         return numpyro.sample("x", dist.Normal(0.0))
...     
...     cluster = numpyro.sample("cluster", dist.Normal())
...     return cond(cluster > 0, true_fun, false_fun, None)

>>> def guide():
...     m1 = numpyro.param("m1", 10.0)
...     s1 = numpyro.param("s1", 0.1, constraint=dist.constraints.positive)
...     m2 = numpyro.param("m2", 10.0)
...     s2 = numpyro.param("s2", 0.1, constraint=dist.constraints.positive)
...     
...     def true_fun(_):
...         return numpyro.sample("x", dist.Normal(m1, s1))
...     
...     def false_fun(_):
...         return numpyro.sample("x", dist.Normal(m2, s2))
...     
...     cluster = numpyro.sample("cluster", dist.Normal())
...     return cond(cluster > 0, true_fun, false_fun, None)

>>> svi = SVI(model, guide, numpyro.optim.Adam(1e-2), Trace_ELBO(num_particles=100))
>>> svi_result = svi.run(random.PRNGKey(0), num_steps=2500)
```

⚠️ Warning

This is an experimental utility function that allows users to use JAX control flow with NumPyro’s effect handlers. Currently, \textit{sample} and \textit{deterministic} sites within \textit{true\_fun} and \textit{false\_fun} are supported. If you notice that any effect handlers or distributions are unsupported, please file an issue.

⚠️ Warning

The \textit{cond} primitive does not currently support enumeration and can not be used inside a \texttt{numpyro.plate} context.
All sample sites must belong to the same distribution class. For example the following is not supported

```python
cond(
    True,
    lambda _: numpyro.sample("x", dist.Normal()),
    lambda _: numpyro.sample("x", dist.Laplace()),
    None,
)
```

**Parameters**

- `pred (bool)` – Boolean scalar type indicating which branch function to apply
- `true_fun (callable)` – A function to be applied if `pred` is true.
- `false_fun (callable)` – A function to be applied if `pred` is false.
- `operand` – Operand input to either branch depending on `pred`. This can be any JAX PyTree (e.g. list / dict of arrays).

**Returns**

Output of the applied branch function.
CHAPTER
THREE

DISTRIBUTIONS

3.1 Base Distribution

3.1.1 Distribution

```python
class Distribution(batch_shape=(), event_shape=(), *, validate_args=None):
    Bases: object
    Base class for probability distributions in NumPyro. The design largely follows from torch.distributions.

    Parameters
    - batch_shape – The batch shape for the distribution. This designates independent (possibly non-identical) dimensions of a sample from the distribution. This is fixed for a distribution instance and is inferred from the shape of the distribution parameters.
    - event_shape – The event shape for the distribution. This designates the dependent dimensions of a sample from the distribution. These are collapsed when we evaluate the log probability density of a batch of samples using log_prob.
    - validate_args – Whether to enable validation of distribution parameters and arguments to log_prob method.

    As an example:

    >>> import jax.numpy as jnp
    >>> import numpyro.distributions as dist
    >>> d = dist.Dirichlet(jnp.ones((2, 3, 4)))
    >>> d.batch_shape
    (2, 3)
    >>> d.event_shape
    (4,)

    arg_constraints = {}
    support = None
    has.enumerate_support = False
    reparameterized_params = []
    pytree_data_fields = ()
    pytree_aux_fields = ('_batch_shape', '_event_shape')
```

27
classmethod gather_pytree_data_fields()
classmethod gather_pytree_aux_fields() \rightarrow \text{tuple}
tree_flatten()
classmethod tree_unflatten(aux_data, params)
static set_default_validate_args(value)

property batch_shape
    Returns the shape over which the distribution parameters are batched.
    
    Returns
    batch shape of the distribution.
    
    Return type
    \text{tuple}

property event_shape
    Returns the shape of a single sample from the distribution without batching.
    
    Returns
    event shape of the distribution.
    
    Return type
    \text{tuple}

property event_dim
    Returns
    Number of dimensions of individual events.
    
    Return type
    \text{int}

property has_rsample

rsample(key, sample_shape=())

shape(sample_shape=())
    The tensor shape of samples from this distribution.
    
    Samples are of shape:
    
    \begin{equation}
    \text{d}.\text{shape(sample_shape)} = \text{sample_shape} + \text{d}.\text{batch_shape} + \text{d}.\text{event_shape}
    \end{equation}
    
    Parameters
    sample_shape (tuple) – the size of the iid batch to be drawn from the distribution.
    
    Returns
    shape of samples.
    
    Return type
    \text{tuple}

sample(key, sample_shape=())
    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
    
    Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.
Parameters

- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

sample_with_intermediates(key, sample_shape=(),)

Same as sample except that any intermediate computations are returned (useful for TransformedDistribution).

Parameters

- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

log_prob(value)

Evaluates the log probability density for a batch of samples given by value.

Parameters

value – A batch of samples from the distribution.

Returns

an array with shape value.shape[:-self.event_shape]

Return type

numpy.ndarray

property mean

Mean of the distribution.

property variance

Variance of the distribution.

to_event(reinterpreted_batch_ndims=(),)

Interpret the rightmost reinterpreted_batch_ndims batch dimensions as dependent event dimensions.

Parameters

reinterpreted_batch_ndims – Number of rightmost batch dims to interpret as event dims.

Returns

An instance of Independent distribution.

Return type

numpyro.distributions.distribution.Independent

enumerate_support(expand=True)

Returns an array with shape len(support) x batch_shape containing all values in the support.

entropy()

Returns the entropy of the distribution.
**expand**(*batch_shape*)

Returns a new *ExpandedDistribution* instance with batch dimensions expanded to *batch_shape*.

**Parameters**

batch_shape *(tuple)* – batch shape to expand to.

**Returns**

an instance of *ExpandedDistribution*.

**Return type**

*ExpandedDistribution*

**expand_by**(*sample_shape*)

Expands a distribution by adding *sample_shape* to the left side of its *batch_shape*. To expand internal dims of *self.batch_shape* from 1 to something larger, use *expand()* instead.

**Parameters**

sample_shape *(tuple)* – The size of the iid batch to be drawn from the distribution.

**Returns**

An expanded version of this distribution.

**Return type**

*ExpandedDistribution*

**mask**(*mask*)

 Masks a distribution by a boolean or boolean-valued array that is broadcastable to the distributions *Distribution.batch_shape*.

**Parameters**

mask *(bool or jnp.ndarray)* – A boolean or boolean valued array (*True* includes a site, *False* excludes a site).

**Returns**

A masked copy of this distribution.

**Return type**

*MaskedDistribution*

**Example:**

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.distributions import constraints
>>> from numpyro.infer import SVI, Trace_ELBO

>>> def model(data, m):
...     f = numpyro.sample("latent_fairness", dist.Beta(1, 1))
...     with numpyro.plate("N", data.shape[0]):
...         # only take into account the values selected by the mask
...         masked_dist = dist.Bernoulli(f).mask(m)
...         numpyro.sample("obs", masked_dist, obs=data)

>>> def guide(data, m):
...     alpha_q = numpyro.param("alpha_q", 5., constraint=constraints.positive)
...     beta_q = numpyro.param("beta_q", 5., constraint=constraints.positive)
```

(continues on next page)
...  
numpyro.sample("latent_fairness", dist.Beta(alpha_q, beta_q))

```python
>>> data = jnp.concatenate([jnp.ones(5), jnp.zeros(5)])
```  
```python
>>> masked_array = jnp.where(data == 1, True, False)
```  
```python
>>> optimizer = numpyro.optim.Adam(step_size=0.05)
```  
```python
>>> svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
```  
```python
>>> svi_result = svi.run(random.PRNGKey(0), 300, data, masked_array)
```  
```python
>>> params = svi_result.params
```  
```python
>>> # inferred_mean is closer to 1
>>> inferred_mean = params["alpha_q"] / (params["alpha_q"] + params["beta_q"])
```

classmethod `infer_shapes`(*args, **kwargs)

Infers batch_shape and event_shape given shapes of args to __init__().

**Note**

This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

**Parameters**

- *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

**Returns**

A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

**Return type**

tuple

cdf(value)

The cumulative distribution function of this distribution.

**Parameters**

- value – samples from this distribution.

**Returns**

Output of the cumulative distribution function evaluated at `value`.

icdf(q)

The inverse cumulative distribution function of this distribution.

**Parameters**

- q – quantile values, should belong to [0, 1].

**Returns**

The samples whose cdf values equals to `q`.

**property** `is_discrete`

3.1. Base Distribution
3.1.2 ExpandedDistribution

class ExpandedDistribution(base_dist, batch_shape=())

Bases: Distribution

arg_constraints = {}

pytree_data_fields = ('base_dist',)

pytree_aux_fields = ('_expanded_sizes', '_interstitial_sizes')

property has_enumerate_support

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property has_rsample

rsample(key, sample_shape=())

property support

sample_with_intermediates(key, sample_shape=())

Same as sample except that any intermediate computations are returned (useful for TransformedDistribution).

Parameters

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

log_prob(value, intermediates=None)

Evaluates the log probability density for a batch of samples given by value.

Parameters

- **value** – A batch of samples from the distribution.
Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

numpy.ndarray

`enumerate_support` (**expand=True**)  
Returns an array with shape `len(support) x batch_shape` containing all values in the support.

property `mean`

Mean of the distribution.

property `variance`

Variance of the distribution.

`entropy()`  
Returns the entropy of the distribution.

### 3.1.3 FoldedDistribution

**class** FoldedDistribution(*base_dist, *, validate_args=None*)

Bases: TransformedDistribution

Equivalent to `TransformedDistribution(base_dist, AbsTransform())`, but additionally supports `log_prob()`.

Parameters

- **base_dist** (Distribution) – A univariate distribution to reflect.

**support** = Positive(lower_bound=0.0)

**log_prob**(*args, **kwargs*)  
Evaluates the log probability density for a batch of samples given by `value`.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

numpy.ndarray

### 3.1.4 ImproperUniform

**class** ImproperUniform(*support, batch_shape, event_shape, *, validate_args=None*)

Bases: Distribution

A helper distribution with zero `log_prob()` over the `support` domain.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>sample</em> method is not implemented for this distribution. In autoguide and mcmc, initial parameters for improper sites are derived from <code>init_to_uniform</code> or <code>init_to_value</code> strategies.</td>
</tr>
</tbody>
</table>

Usage:
>>> from numpyro import sample
>>> from numpyro.distributions import ImproperUniform, Normal, constraints

>>> def model():
...    # ordered vector with length 10
...    x = sample('x', ImproperUniform(constraints.ordered_vector, (), event_shape=(10,)))

...    # real matrix with shape (3, 4)
...    y = sample('y', ImproperUniform(constraints.real, (), event_shape=(3, 4)))

...    # a shape-(6, 8) batch of length-5 vectors greater than 3
...    z = sample('z', ImproperUniform(constraints.greater_than(3), (6, 8), event_shape=(5,)))

If you want to set improper prior over all values greater than \(a\), where \(a\) is another random variable, you might use

>>> def model():
...    a = sample('a', Normal(0, 1))
...    x = sample('x', ImproperUniform(constraints.greater_than(a), (), event_shape=()))

or if you want to reparameterize it

>>> from numpyro.distributions import TransformedDistribution, transforms
>>> from numpyro.handlers import reparam
>>> from numpyro.infer.reparam import TransformReparam

>>> def model():
...    a = sample('a', Normal(0, 1))
...    with reparam(config={'x': TransformReparam()}):
...       x = sample('x',
...                  TransformedDistribution(ImproperUniform(constraints.positive, ()
...                  transforms.AffineTransform(a, 1))))

Parameters

- **support** (*Constraint*) – the support of this distribution.
- **batch_shape** (*tuple*) – batch shape of this distribution. It is usually safe to set `batch_shape=()`.
- **event_shape** (*tuple*) – event shape of this distribution.

```python
arg_constraints = {}
pytree_data_fields = ('support',)
support = Dependent()
log_prob(*args, **kwargs)
```

Evaluates the log probability density for a batch of samples given by `value`.  

34 Chapter 3. Distributions
Parameters

- `value` – A batch of samples from the distribution.

Returns

- an array with shape `value.shape[:-self.event_shape]`

Return type

- `numpy.ndarray`

### 3.1.5 Independent

**class Independent**

```python
class Independent(base_dist, reinterpreted_batch_ndims, *, validate_args=None)
```

**Bases:** Distribution

Reinterprets batch dimensions of a distribution as event dims by shifting the batch-event dim boundary further to the left.

From a practical standpoint, this is useful when changing the result of `log_prob()`. For example, a univariate Normal distribution can be interpreted as a multivariate Normal with diagonal covariance:

```python
>>> import numpyro.distributions as dist
>>> normal = dist.Normal(jnp.zeros(3), jnp.ones(3))
>>> [normal.batch_shape, normal.event_shape]
[(3,), ()]
>>> diag_normal = dist.Independent(normal, 1)
>>> [diag_normal.batch_shape, diag_normal.event_shape]
[(), (3,)]
```

**Parameters**

- `reinterpreted_batch_ndims` *(int)* – the number of batch dims to reinterpret as event dims.

**arg_constraints** = `{}`

**pytree_data_fields** = (`'base_dist'`,)

**pytree_aux_fields** = (`'reinterpreted_batch_ndims'`,)

**property support**

**property has_enumerate_support**

```python
bool(x) -> bool
```

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**property reparametrized_params**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.
property mean
Mean of the distribution.

property variance
Variance of the distribution.

property has_rsample
rsample(key, sample_shape=())
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(value)
Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

expand(batch_shape)
Returns a new ExpandedDistribution instance with batch dimensions expanded to batch_shape.

Parameters
batch_shape (tuple) – batch shape to expand to.

Returns
an instance of ExpandedDistribution.

Return type
ExpandedDistribution

entropy()
Returns the entropy of the distribution.
3.1.6 MaskedDistribution

class MaskedDistribution(base_dist, mask)
    Bases: Distribution

Masks a distribution by a boolean array that is broadcastable to the distribution's Distribution.batch_shape. In the special case mask is False, computation of log_prob(), is skipped, and constant zero values are returned instead.

    Parameters
    mask (jnp.ndarray or bool) – A boolean or boolean-valued array.

    arg_constraints = {}

pytree_data_fields = ('base_dist', '_mask')
pytree_aux_fields = ('_mask',)

property has enumerate_support
    bool(x) -> bool

    Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property has rsample

rsample(key, sample_shape=())

property support

sample(key, sample_shape=())

    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

    Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
    • sample_shape (tuple) – the sample shape for the distribution.

    Returns
    an array of shape sample_shape + batch_shape + event_shape

    Return type
    numpy.ndarray

log_prob(value)

    Evaluates the log probability density for a batch of samples given by value.

    Parameters
    value – A batch of samples from the distribution.

    Returns
    an array with shape value.shape[:-self.event_shape]

    Return type
    numpy.ndarray

enumerate_support(expand=True)

    Returns an array with shape len(support) x batch_shape containing all values in the support.
property mean
    Mean of the distribution.

property variance
    Variance of the distribution.

tree_flatten()

classmethod tree_unflatten(aux_data, params)

3.1.7 TransformedDistribution

class TransformedDistribution(base_distribution, transforms, *, validate_args=None)
    Bases: Distribution

    Returns a distribution instance obtained as a result of applying a sequence of transforms to a base distribution. For an example, see LogNormal and HalfNormal.

    Parameters
    • base_distribution – the base distribution over which to apply transforms.
    • transforms – a single transform or a list of transforms.
    • validate_args – Whether to enable validation of distribution parameters and arguments to .log_prob method.

    arg_constraints = {}

    pytree_data_fields = ('base_dist', 'transforms')

    property has_rsample

    rsample(key, sample_shape=())

    property support

    sample(key, sample_shape=())

    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

    Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
    • sample_shape (tuple) – the sample shape for the distribution.

    Returns
    an array of shape sample_shape + batch_shape + event_shape

    Return type
    numpy.ndarray

    sample_with_intermediates(key, sample_shape=())

    Same as sample except that any intermediate computations are returned (useful for TransformedDistribution).

    Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• **sample_shape** (tuple) – the sample shape for the distribution.

**Returns**

an array of shape \(sample\_shape + batch\_shape + event\_shape\)

**Return type**

numpy.ndarray

**log_prob**(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by \(value\).

**Parameters**

\(value\) – A batch of samples from the distribution.

**Returns**

an array with shape \(value\_shape[:-self.event\_shape]\)

**Return type**

numpy.ndarray

**property mean**

Mean of the distribution.

**property variance**

Variance of the distribution.

### 3.1.8 Delta

**class Delta**(\(v=0.0\), \(log\_density=0.0\), \(event\_dim=0\), *, \(validate\_args=\text{None}\))

**Bases:** Distribution

**arg_constraints** = {'log_density': Real(), 'v': Dependent()}

**reparametrized_params** = ['v', 'log_density']

**property support**

**sample**(*key, sample_shape=())

Returns a sample from the distribution having shape given by \(sample\_shape + batch\_shape + event\_shape\). Note that when \(sample\_shape\) is non-empty, leading dimensions (of size \(sample\_shape\)) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

• **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

• **sample_shape** (tuple) – the sample shape for the distribution.

**Returns**

an array of shape \(sample\_shape + batch\_shape + event\_shape\)

**Return type**

numpy.ndarray

**log_prob**(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by \(value\).

**Parameters**

\(value\) – A batch of samples from the distribution.
Returns an array with shape `value.shape[:-self.event_shape]`

Return type
numpy.ndarray

property mean
Mean of the distribution.

property variance
Variance of the distribution.

entropy()
Returns the entropy of the distribution.

3.1.9 Unit

class Unit(log_factor, *, validate_args=None)
    Bases: Distribution

    Trivial nonnormalized distribution representing the unit type.
    The unit type has a single value with no data, i.e. value.size == 0.
    This is used for `numpyro.factor()` statements.

    arg_constraints = {'log_factor': Real()}

    support = Real()

    sample(key, sample_shape=())

        Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`.
        Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned
        sample will be filled with iid draws from the distribution instance.

        Parameters

            * key (`jax.random.PRNGKey`) – the rng_key key to be used for the distribution.

            * sample_shape (tuple) – the sample shape for the distribution.

        Returns

            an array of shape `sample_shape + batch_shape + event_shape`

        Return type

            numpy.ndarray

    log_prob(value)

        Evaluates the log probability density for a batch of samples given by `value`.

        Parameters

            * value – A batch of samples from the distribution.

        Returns

            an array with shape `value.shape[:-self.event_shape]`

        Return type

            numpy.ndarray
3.2 Continuous Distributions

3.2.1 AsymmetricLaplace

class AsymmetricLaplace(loc=0.0, scale=1.0, asymmetry=1.0, *, validate_args=None)
Bases: Distribution

arg_constraints = {'asymmetry': Positive(lower_bound=0.0), 'loc': Real(), 'scale': Positive(lower_bound=0.0)}

reparameterized_params = ['loc', 'scale', 'asymmetry']
support = Real()
left_scale()
right_scale()

log_prob(value)
   Evaluates the log probability density for a batch of samples given by value.

   Parameters
   value – A batch of samples from the distribution.

   Returns
   an array with shape value.shape[-self.event_shape]

   Return type
   numpy.ndarray

sample(key, sample_shape=())
   Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
   Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned
   sample will be filled with iid draws from the distribution instance.

   Parameters
   • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
   • sample_shape (tuple) – the sample shape for the distribution.

   Returns
   an array of shape sample_shape + batch_shape + event_shape

   Return type
   numpy.ndarray

property mean
   Mean of the distribution.

property variance
   Variance of the distribution.

cdf(value)
   The cumulative distribution function of this distribution.

   Parameters
   value – samples from this distribution.
Returns
output of the cumulative distribution function evaluated at value.

\texttt{icdf(value)}
The inverse cumulative distribution function of this distribution.

\textbf{Parameters}
\qquad q – quantile values, should belong to [0, 1].

\textbf{Returns}
the samples whose cdf values equals to \( q \).

### 3.2.2 AsymmetricLaplaceQuantile

**class AsymmetricLaplaceQuantile(loc=0.0, scale=1.0, quantile=0.5, *, validate_args=None)**

**Bases:** Distribution

An alternative parameterization of AsymmetricLaplace commonly applied in Bayesian quantile regression.

Instead of the \textit{asymmetry} parameter employed by AsymmetricLaplace, to define the balance between left-versus right-hand sides of the distribution, this class utilizes a \textit{quantile} parameter, which describes the proportion of probability density that falls to the left-hand side of the distribution.

The \textit{scale} parameter is also interpreted slightly differently than in AsymmetricLaplace. When \( \text{loc}=0 \) and \( \text{scale}=1 \), AsymmetricLaplace(0,1,1) is equivalent to Laplace(0,1), while AsymmetricLaplaceQuantile(0,1,0.5) is equivalent to Laplace(0,2).

**arg_constraints = \{'loc': Real(), 'quantile': OpenInterval(lower_bound=0.0, upper_bound=1.0), 'scale': Positive(lower_bound=0.0)\}**

**reparameterized_params = ['loc', 'scale', 'quantile']**

**support = Real()**

**pytree_data_fields = ('loc', 'scale', 'quantile', '_ald')**

\textbf{log_prob(value)}
Evaluates the log probability density for a batch of samples given by value.

\textbf{Parameters}
\qquad value – A batch of samples from the distribution.

\textbf{Returns}
an array with shape \( \text{value}.shape[:-self.event_shape] \)

\textbf{Return type}
numpy.ndarray

\textbf{sample(key, sample_shape=())}
Returns a sample from the distribution having shape given by \( \text{sample_shape} + \text{batch_shape} + \text{event_shape} \). Note that when \( \text{sample_shape} \) is non-empty, leading dimensions (of size \( \text{sample_shape} \)) of the returned sample will be filled with iid draws from the distribution instance.

\textbf{Parameters}
\quad \textbf{key (jax.random.PRNGKey)} – the rng_key key to be used for the distribution.

\quad \textbf{sample_shape (tuple)} – the sample shape for the distribution.

\textbf{Returns}
an array of shape \( \text{sample_shape} + \text{batch_shape} + \text{event_shape} \)
Return type

numpy.ndarray

property mean
Mean of the distribution.

property variance
Variance of the distribution.

cdf(value)
The cumulative distribution function of this distribution.

Parameters

value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.

icdf(value)
The inverse cumulative distribution function of this distribution.

Parameters

q – quantile values, should belong to [0, 1].

Returns
the samples whose cdf values equals to q.

3.2.3 Beta

class Beta(concentration1, concentration0, *, validate_args=None)
Bases: Distribution

arg_constraints = {'concentration0': Positive(lower_bound=0.0), 'concentration1': Positive(lower_bound=0.0)}

reparametrized_params = ['concentration1', 'concentration0']
support = UnitInterval(lower_bound=0.0, upper_bound=1.0)
pytree_data_fields = ('concentration0', 'concentration1', '_dirichlet')
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

• sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray
log_prob(*args, **kwargs)
   Evaluates the log probability density for a batch of samples given by value.

   Parameters
   value – A batch of samples from the distribution.

   Returns
   an array with shape value.shape[:-self.event_shape]

   Return type
   numpy.ndarray

property mean
   Mean of the distribution.

property variance
   Variance of the distribution.

cdf(value)
   The cumulative distribution function of this distribution.

   Parameters
   value – samples from this distribution.

   Returns
   output of the cumulative distribution function evaluated at value.

icdf(q)
   The inverse cumulative distribution function of this distribution.

   Parameters
   q – quantile values, should belong to [0, 1].

   Returns
   the samples whose cdf values equals to q.

entropy()
   Returns the entropy of the distribution.

3.2.4 BetaProportion

class BetaProportion(mean, concentration, *, validate_args=None)
   Bases: Beta

   The BetaProportion distribution is a reparameterization of the conventional Beta distribution in terms of a the
   variate mean and a precision parameter.

   Reference:

   Beta regression for modelling rates and proportion, Ferrari Silvia, and

   arg_constraints = {'concentration': Positive(lower_bound=0.0), 'mean':
                      OpenInterval(lower_bound=0.0, upper_bound=1.0)}

   reparametrized_params = ['mean', 'concentration']

   support = UnitInterval(lower_bound=0.0, upper_bound=1.0)

   pytree_data_fields = {'concentration',}
### 3.2.5 CAR

**class CAR(loc, correlation, conditional_precision, adj_matrix, *, is_sparse=False, validate_args=None)**

**Bases:** Distribution

The Conditional Autoregressive (CAR) distribution is a special case of the multivariate normal in which the precision matrix is structured according to the adjacency matrix of sites. The amount of autocorrelation between sites is controlled by `correlation`. The distribution is a popular prior for areal spatial data.

**Parameters**

- **loc** *(float or ndarray)* – mean of the multivariate normal
- **correlation** *(float)* – autoregression parameter. For most cases, the value should lie between 0 (sites are independent, collapses to an iid multivariate normal) and 1 (perfect autocorrelation between sites), but the specification allows for negative correlations.
- **conditional_precision** *(float)* – positive precision for the multivariate normal
- **adj_matrix** *(ndarray or scipy.sparse.csr_matrix)* – symmetric adjacency matrix where 1 indicates adjacency between sites and 0 otherwise. jax.numpy.ndarray adj_matrix is supported but is not recommended over numpy.ndarray or scipy.sparse.spmatrix.
- **is_sparse** *(bool)* – whether to use a sparse form of adj_matrix in calculations (must be True if adj_matrix is a scipy.sparse.spmatrix)

**arg_constraints** = {'adj_matrix': Dependent(), 'conditional_precision': Positive(lower_bound=0.0), 'correlation': OpenInterval(lower_bound=-1, upper_bound=1), 'loc': RealVector(Real(), 1)}

**support** = RealVector(Real(), 1)

**reparametrized_params** = ['loc', 'correlation', 'conditional_precision', 'adj_matrix']

**pytree_aux_fields** = ('is_sparse', 'adj_matrix')

**sample**(key, sample_shape=())

Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- **key** *(jax.random.PRNGKey)* – the rng_key key to be used for the distribution.
- **sample_shape** *(tuple)* – the sample shape for the distribution.

**Returns**

an array of shape `sample_shape + batch_shape + event_shape`

**Return type**

numpy.ndarray

**log_prob**(args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

**Parameters**

- **value** – A batch of samples from the distribution.
Returns
an array with shape `value.shape[:-self.event_shape]`

Return type
numpy.ndarray

property mean
Mean of the distribution.

precision_matrix()

static infer_shapes(loc, correlation, conditional_precision, adj_matrix)
Infers batch_shape and event_shape given shapes of args to `__init__()`.

Note
This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters
- *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns
A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type
tuple

tree_flatten()
classmethod tree_unflatten(aux_data, params)

3.2.6 Cauchy
class Cauchy(loc=0.0, scale=1.0, *, validate_args=None)
Bases: Distribution

arg_constraints = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}
support = Real()
reparameetrized_params = ['loc', 'scale']
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• **sample_shape** *(tuple)* – the sample shape for the distribution.

Returns
an array of shape *sample_shape + batch_shape + event_shape*

Return type
numpy.ndarray

**log_prob** (*args, **kwargs*)
Evaluates the log probability density for a batch of samples given by *value*.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape *value.shape[:-self.event_shape]*

Return type
numpy.ndarray

**property mean**
Mean of the distribution.

**property variance**
Variance of the distribution.

**cdf** (*value*)
The cumulative distribution function of this distribution.

Parameters
value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at *value*.

**icdf** (*q*)
The inverse cumulative distribution function of this distribution.

Parameters
q – quantile values, should belong to [0, 1].

Returns
the samples whose cdf values equals to *q*.

**entropy** ()
Returns the entropy of the distribution.

### 3.2.7 Chi2

**class Chi2** *(df, *, validate_args=None)*

Bases: Gamma

arg_constraints = {"df": Positive(lower_bound=0.0)}

reparametrized_params = ['df']
3.2.8 Dirichlet

class Dirichlet(concentration, *, validate_args=None)
Bases: Distribution

arg_constraints = {'concentration': IndependentConstraint(Positive(lower_bound=0.0), 1)}

reparametrized_params = ['concentration']
support = Simplex()
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters
- value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

property mean
Mean of the distribution.

property variance
Variance of the distribution.

static infer_shapes(concentration)

Infers batch_shape and event_shape given shapes of args to __init__().

Note
This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters
- *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
**kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns
A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type
tuple

entropy()
Returns the entropy of the distribution.

3.2.9 EulerMaruyama

class EulerMaruyama(t, sde_fn, init_dist, *, validate_args=None)
Bases: Distribution

Euler–Maruyama method is a method for the approximate numerical solution of a stochastic differential equation (SDE)

Parameters
• t (ndarray) – discretized time
• sde_fn (callable) – function returning the drift and diffusion coefficients of SDE
• init_dist (Distribution) – Distribution for initial values.

References

arg_constraints = {'t': OrderedVector()}
pytree_data_fields = ('t', 'init_dist')
pytree_aux_fields = ('sde_fn',)

property support

sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray
**log_prob(args, **kwargs)**

Evaluates the log probability density for a batch of samples given by *value*.

**Parameters**

- **value** – A batch of samples from the distribution.

**Returns**

- an array with shape *value.shape[:-self.event_shape]*

**Return type**

numpy.ndarray

### 3.2.10 Exponential

**class Exponential(rate=1.0, *, validate_args=None)**

**Bases:** Distribution

**reparametrized_params** = ['rate']

**arg_constraints** = {'rate': Positive(lower_bound=0.0)}

**support** = Positive(lower_bound=0.0)

**sample(key, sample_shape=())**

Returns a sample from the distribution having shape given by *sample_shape + batch_shape + event_shape*. Note that when *sample_shape* is non-empty, leading dimensions (of size *sample_shape*) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- **key** *(jax.random.PRNGKey)* – the rng_key key to be used for the distribution.

- **sample_shape** *(tuple)* – the sample shape for the distribution.

**Returns**

- an array of shape *sample_shape + batch_shape + event_shape*

**Return type**

numpy.ndarray

**log_prob(args, **kwargs)**

Evaluates the log probability density for a batch of samples given by *value*.

**Parameters**

- **value** – A batch of samples from the distribution.

**Returns**

- an array with shape *value.shape[:-self.event_shape]*

**Return type**

numpy.ndarray

**property mean**

Mean of the distribution.

**property variance**

Variance of the distribution.
**cdf(value)**

The cumulative distribution function of this distribution.

**Parameters**
- **value** – samples from this distribution.

**Returns**
- output of the cumulative distribution function evaluated at value.

**icdf(q)**

The inverse cumulative distribution function of this distribution.

**Parameters**
- **q** – quantile values, should belong to [0, 1].

**Returns**
- the samples whose cdf values equals to q.

**entropy()**

Returns the entropy of the distribution.

### 3.2.11 Gamma

**class Gamma(concentration, rate=1.0, *, validate_args=None)**

**Bases:** Distribution

**arg_constraints =**
- 'concentration': Positive(lower_bound=0.0),
- 'rate': Positive(lower_bound=0.0)

**support =** Positive(lower_bound=0.0)

**reparametrized_params =** ['concentration', 'rate']

**sample(key, sample_shape=())**

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**
- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

**Returns**
- an array of shape sample_shape + batch_shape + event_shape

**Return type**
- numpy.ndarray

**log_prob(*args, **kwargs)**

Evaluates the log probability density for a batch of samples given by value.

**Parameters**
- **value** – A batch of samples from the distribution.

**Returns**
- an array with shape value.shape[:-self.event_shape]
Return type
numpy.ndarray

property mean
Mean of the distribution.

property variance
Variance of the distribution.

cdf(x)
The cumulative distribution function of this distribution.

Parameters
value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.

icdf(q)
The inverse cumulative distribution function of this distribution.

Parameters
q – quantile values, should belong to [0, 1].

Returns
the samples whose cdf values equals to q.

entropy()
Returns the entropy of the distribution.

3.2.12 GaussianCopula

class GaussianCopula(marginal_dist, correlation_matrix=None, correlation_cholesky=None, *, validate_args=None)
Bases: Distribution
A distribution that links the batch_shape[-1] of marginal distribution marginal_dist with a multivariate Gaussian copula modelling the correlation between the axes.

Parameters
• marginal_dist (Distribution) – Distribution whose last batch axis is to be coupled.
• correlation_matrix (array_like) – Correlation matrix of coupling multivariate normal distribution.
• correlation_cholesky (array_like) – Correlation Cholesky factor of coupling multivariate normal distribution.

arg_constraints = {'correlation_cholesky': Chol(), 'correlation_matrix': Corr()}
reparameterized_params = ['correlation_matrix', 'correlation_cholesky']
pytree_data_fields = ('marginal_dist', 'base_dist')
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.
Parameters

- **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.
- **sample_shape** (*tuple*) – the sample shape for the distribution.

Returns

an array of shape `sample_shape + batch_shape + event_shape`

Return type

numpy.ndarray

**log_prob**(*args, **kwargs*)

Evaluates the log probability density for a batch of samples given by `value`.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

numpy.ndarray

**property mean**

Mean of the distribution.

**property variance**

Variance of the distribution.

**property support**

3.2.13 GaussianCopulaBeta

```python
class GaussianCopulaBeta(concentration1, concentration0, correlation_matrix=None, correlation_cholesky=None, *, validate_args=False)
```

Bases: GaussianCopula

arg_constraints = {'concentration0': Positive(lower_bound=0.0), 'concentration1': Positive(lower_bound=0.0), 'correlation_cholesky': CorrCholesky(), 'correlation_matrix': CorrMatrix()}

support = IndependentConstraint(UnitInterval(lower_bound=0.0, upper_bound=1.0), 1)

pytree_data_fields = ('concentration1', 'concentration0')

3.2. Continuous Distributions
3.2.14 GaussianRandomWalk

class GaussianRandomWalk(scale=1.0, num_steps=1, *, validate_args=None)
    Bases: Distribution
    arg_constraints = {'scale': Positive(lower_bound=0.0)}
    support = RealVector(Real(), 1)
    reparametrized_params = ['scale']
    pytree_aux_fields = ('num_steps',)
    sample(key, sample_shape=())
    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
    Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned
    sample will be filled with iid draws from the distribution instance.
    Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
    • sample_shape (tuple) – the sample shape for the distribution.
    Returns
    an array of shape sample_shape + batch_shape + event_shape
    Return type
    numpy.ndarray
    log_prob(*args, **kwargs)
    Evaluates the log probability density for a batch of samples given by value.
    Parameters
    value – A batch of samples from the distribution.
    Returns
    an array with shape value.shape[:-self.event_shape]
    Return type
    numpy.ndarray

property mean
    Mean of the distribution.

property variance
    Variance of the distribution.

3.2.15 Gompertz

class Gompertz(concentration, rate=1.0, *, validate_args=None)
    Bases: Distribution
    Gompertz Distribution.
    The Gompertz distribution is a distribution with support on the positive real line that is closely related to the
    Gumbel distribution. This implementation follows the notation used in the Wikipedia entry for the Gompertz
However, we call the parameter “eta” a concentration parameter and the parameter “b” a rate parameter (as opposed to scale parameter as in wikipedia description.)

The CDF, in terms of concentration (con) and rate, is

\[
F(x) = 1 - \exp \{-\text{con} \ast \exp\{x \ast \text{rate}\} - 1\}
\]

arg_constraints = {'concentration': Positive(lower_bound=0.0), 'rate': Positive(lower_bound=0.0)}
support = Positive(lower_bound=0.0)
reparametrized_params = ['concentration', 'rate']
sample(key, sample_shape=(i))
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray
cdf(value)
The cumulative distribution function of this distribution.

Parameters
value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.

icdf(q)
The inverse cumulative distribution function of this distribution.

Parameters
q – quantile values, should belong to \([0, 1]\).

Returns
the samples whose cdf values equals to q.
property mean
Mean of the distribution.

3.2.16 Gumbel

class Gumbel(loc=0.0, scale=1.0, *, validate_args=None)
Bases: Distribution
arg_constraints = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}
support = Real()
reparametrized_params = ['loc', 'scale']
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
- value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

property mean
Mean of the distribution.

property variance
Variance of the distribution.

cdf(value)
The cummulative distribution function of this distribution.

Parameters
- value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.
icdf(q)

The inverse cumulative distribution function of this distribution.

Parameters
q – quantile values, should belong to [0, 1].

Returns
the samples whose cdf values equals to q.

3.2.17 HalfCauchy

class HalfCauchy(scale=1.0, *, validate_args=None)

Bases: Distribution

reparametrized_params = ['scale']
support = Positive(lower_bound=0.0)
arg_constraints = {'scale': Positive(lower_bound=0.0)}
pytree_data_fields = ('_cauchy', 'scale')
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

cdf(value)

The cumulative distribution function of this distribution.

Parameters
value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.
icdf($q$)
The inverse cumulative distribution function of this distribution.

**Parameters**
- $q$ – quantile values, should belong to [0, 1].

**Returns**
the samples whose cdf values equals to $q$.

**property mean**
Mean of the distribution.

**property variance**
Variance of the distribution.

### 3.2.18 HalfNormal

class HalfNormal(scale=1.0, *, validate_args=None)

**Bases:** Distribution

**reparametrized_params** = ['scale']

**support** = Positive(lower_bound=0.0)

**arg_constraints** = {'scale': Positive(lower_bound=0.0)}

**pytree_data_fields** = ('_normal', 'scale')

**sample**(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**
- **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.
- **sample_shape** (*tuple*) – the sample shape for the distribution.

**Returns**
an array of shape sample_shape + batch_shape + event_shape

**Return type**
numpy.ndarray

**log_prob**(args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

**Parameters**
- **value** – A batch of samples from the distribution.

**Returns**
an array with shape value.shape[:-self.event_shape]

**Return type**
numpy.ndarray
**NumPyro Documentation**

- **cdf**(value)
  The cummulative distribution function of this distribution.

  **Parameters**
  value – samples from this distribution.

  **Returns**
  output of the cummulative distribution function evaluated at value.

- **icdf**(q)
  The inverse cumulative distribution function of this distribution.

  **Parameters**
  q – quantile values, should belong to [0, 1].

  **Returns**
  the samples whose cdf values equals to q.

- **property mean**
  Mean of the distribution.

- **property variance**
  Variance of the distribution.

### 3.2.19 InverseGamma

**class InverseGamma**(concentration, rate=1.0, *, validate_args=None)

**Bases:** TransformedDistribution

**Note**

We keep the same notation rate as in Pyro but it plays the role of scale parameter of InverseGamma in literatures (e.g. wikipedia: [https://en.wikipedia.org/wiki/Inverse-gamma_distribution](https://en.wikipedia.org/wiki/Inverse-gamma_distribution))

arg_constraints = {'concentration': Positive(lower_bound=0.0), 'rate': Positive(lower_bound=0.0)}

reparametrized_params = ['concentration', 'rate']

support = Positive(lower_bound=0.0)

**property mean**

Mean of the distribution.

**property variance**

Variance of the distribution.

- **cdf**(x)
  The cummulative distribution function of this distribution.

  **Parameters**
  value – samples from this distribution.

  **Returns**
  output of the cummulative distribution function evaluated at value.
entropy()

Returns the entropy of the distribution.

### 3.2.20 Kumaraswamy

class Kumaraswamy(concentration1, concentration0, *, validate_args=None)

**Bases:** Distribution

arg_constraints = {'concentration0': Positive(lower_bound=0.0), 'concentration1': Positive(lower_bound=0.0)}

reparametrized_params = ['concentration1', 'concentration0']

support = UnitInterval(lower_bound=0.0, upper_bound=1.0)

KL_KUMARASWAMY_BETA_TAYLOR_ORDER = 10

**sample(key, sample_shape=())**

Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

**Returns**

an array of shape `sample_shape + batch_shape + event_shape`

**Return type**

numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by `value`.

**Parameters**

- **value** – A batch of samples from the distribution.

**Returns**

an array with shape `value.shape[:-self.event_shape]`

**Return type**

numpy.ndarray

**property mean**

Mean of the distribution.

**property variance**

Variance of the distribution.
### 3.2.21 Laplace

```python
class Laplace(loc=0.0, scale=1.0, *, validate_args=None):
    Bases: Distribution

    arg_constraints = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}
    support = Real()
    reparametrized_params = ['loc', 'scale']

    sample(key, sample_shape=())
    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

    Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
    • sample_shape (tuple) – the sample shape for the distribution.

    Returns
    an array of shape sample_shape + batch_shape + event_shape

    Return type
numpy.ndarray

    log_prob(*args, **kwargs)
    Evaluates the log probability density for a batch of samples given by value.

    Parameters
    value – A batch of samples from the distribution.

    Returns
    an array with shape value.shape[:-self.event_shape]

    Return type
    numpy.ndarray

property mean
    Mean of the distribution.

property variance
    Variance of the distribution.

cdf(value)
    The cumulative distribution function of this distribution.

    Parameters
    value – samples from this distribution.

    Returns
    output of the cumulative distribution function evaluated at value.

icdf(q)
    The inverse cumulative distribution function of this distribution.

    Parameters
    q – quantile values, should belong to [0, 1].
```
Returns the samples whose cdf values equals to $q$.

entropy()

Returns the entropy of the distribution.

3.2.22 LKJ

class LKJ(dimension, concentration=1.0, sample_method='onion', *, validate_args=None)

Bases: TransformedDistribution

LKJ distribution for correlation matrices. The distribution is controlled by concentration parameter $\eta$ to make the probability of the correlation matrix $M$ proportional to $\det(M)^{\eta - 1}$. Because of that, when concentration == 1, we have a uniform distribution over correlation matrices.

When concentration > 1, the distribution favors samples with large determinant. This is useful when we know a priori that the underlying variables are not correlated.

When concentration < 1, the distribution favors samples with small determinant. This is useful when we know a priori that some underlying variables are correlated.

Sample code for using LKJ in the context of multivariate normal sample:

```python
def model(y):
    # y has dimension N x d
    d = y.shape[1]
    N = y.shape[0]
    # Vector of variances for each of the d variables
    theta = numpyro.sample("theta", dist.HalfCauchy(jnp.ones(d)))
    concentration = jnp.ones(1)  # Implies a uniform distribution over correlation matrices
    corr_mat = numpyro.sample("corr_mat", dist.LKJ(d, concentration))
    sigma = jnp.sqrt(theta)
    # we can also use a faster formula `cov_mat = jnp.outer(sigma, sigma) * corr_mat`
    cov_mat = jnp.matmul(jnp.matmul(jnp.diag(sigma), corr_mat), jnp.diag(sigma))
    # Vector of expectations
    mu = jnp.zeros(d)

    with numpyro.plate("observations", N):
        obs = numpyro.sample("obs", dist.MultivariateNormal(mu, covariance_matrix=cov_mat), obs=y)
    return obs
```

Parameters

- **dimension** *(int)* – dimension of the matrices
- **concentration** *(ndarray)* – concentration/shape parameter of the distribution (often referred to as eta)
- **sample_method** *(str)* – Either “cvine” or “onion”. Both methods are proposed in [1] and offer the same distribution over correlation matrices. But they are different in how to generate samples. Defaults to “onion”.

References
arg_constraints = {'concentration': Positive(lower_bound=0.0)}
reparametrized_params = ['concentration']
support = CorrMatrix()
pytree_aux_fields = ('dimension', 'sample_method')

property mean:
    Mean of the distribution.

3.2.23 LKJCholesky

class LKJCholesky(dimension, concentration=1.0, sample_method='onion', *, validate_args=None)

LKJ distribution for lower Cholesky factors of correlation matrices. The distribution is controlled by
concentration parameter $\eta$ to make the probability of the correlation matrix $M$ generated from a Cholesky
factor proportional to $\det(M)^{\eta-1}$. Because of that, when $\text{concentration} == 1$, we have a uniform distribution
over Cholesky factors of correlation matrices.

When $\text{concentration} > 1$, the distribution favors samples with large diagonal entries (hence large determin-
ent). This is useful when we know a priori that the underlying variables are not correlated.

When $\text{concentration} < 1$, the distribution favors samples with small diagonal entries (hence small determi-

ten). This is useful when we know a priori that some underlying variables are correlated.

Sample code for using LKJCholesky in the context of multivariate normal sample:

```python
def model(y):
    # y has dimension N x d
    d = y.shape[1]
    N = y.shape[0]
    # Vector of variances for each of the d variables
    theta = numpyro.sample("theta", dist.HalfCauchy(jnp.ones(d)))
    # Lower cholesky factor of a correlation matrix
    concentration = jnp.ones(1)  # Implies a uniform distribution over correlation matrices
    L_omega = numpyro.sample("L_omega", dist.LKJCholesky(d, concentration))
    # Lower cholesky factor of the covariance matrix
    sigma = jnp.sqrt(theta)
    # we can also use a faster formula `L_Omega = sigma[... , None] * L_omega`
    L_Omega = jnp.matmul(jnp.diag(sigma), L_omega)
    # Vector of expectations
    mu = jnp.zeros(d)

    with numpyro.plate("observations", N):
        obs = numpyro.sample("obs", dist.MultivariateNormal(mu, scale_tril=L_Omega),
        return obs
```

Parameters
• **dimension** (*int*) – dimension of the matrices

• **concentration** (*ndarray*) – concentration/shape parameter of the distribution (often referred to as eta)

• **sample method** (*str*) – Either “cvine” or “onion”. Both methods are proposed in [1] and offer the same distribution over correlation matrices. But they are different in how to generate samples. Defaults to “onion”.

**References**

[1] *Generating random correlation matrices based on vines and extended onion method*, Daniel Lewandowski, Dorota Kurowicka, Harry Joe

```python
arg_constraints = {'concentration': Positive(lower_bound=0.0)}
reparametrized_params = ['concentration']
support = CorrCholesky()
pytrees_data_fields = ('_beta', 'concentration')
pytrees_aux_fields = ('dimension', 'sample_method')

sample(key, sample_shape=())

Returns a sample from the distribution having shape given by \(\text{sample_shape} + \text{batch_shape} + \text{event_shape}\). Note that when \(\text{sample_shape}\) is non-empty, leading dimensions (of size \(\text{sample_shape}\)) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

• **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.

• **sample_shape** (*tuple*) – the sample shape for the distribution.

Returns

an array of shape \(\text{sample_shape} + \text{batch_shape} + \text{event_shape}\)

Return type

numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by \(value\).

Parameters

• **value** – A batch of samples from the distribution.

Returns

an array with shape \(value.shape[:-self.event_shape]\)

Return type

numpy.ndarray
```
### 3.2.24 LogNormal

```python
class LogNormal(loc=0.0, scale=1.0, *, validate_args=None):
    Bases: TransformDistribution

    arg_constraints = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}

    support = Positive(lower_bound=0.0)

    reparametrized_params = ['loc', 'scale']

    property mean
        Mean of the distribution.

    property variance
        Variance of the distribution.

    cdf(x)
        The cumulative distribution function of this distribution.

        Parameters
            value -- samples from this distribution.

        Returns
            output of the cumulative distribution function evaluated at value.

    entropy()
        Returns the entropy of the distribution.
```

### 3.2.25 LogUniform

```python
class LogUniform(low, high, *, validate_args=None):
    Bases: TransformDistribution

    arg_constraints = {'high': Positive(lower_bound=0.0), 'low': Positive(lower_bound=0.0)}

    reparametrized_params = ['low', 'high']

    pytree_data_fields = ('low', 'high', '_support')

    property support

    property mean
        Mean of the distribution.

    property variance
        Variance of the distribution.

    cdf(x)
        The cumulative distribution function of this distribution.

        Parameters
            value -- samples from this distribution.

        Returns
            output of the cumulative distribution function evaluated at value.
```
entropy()
  Returns the entropy of the distribution.

3.2.26 Logistic

class Logistic(loc=0.0, scale=1.0, *, validate_args=None)
  Bases: Distribution
  arg_constraints = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}
  support = Real()
  reparameterized_params = ['loc', 'scale']
  sample(key, sample_shape=())
    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
    Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.
    Parameters
    • key ([prng.PRNGKey]) – the rng_key key to be used for the distribution.
      • sample_shape (tuple) – the sample shape for the distribution.
    Returns
      an array of shape sample_shape + batch_shape + event_shape
    Return type
      numpy.ndarray
  log_prob(*args, **kwargs)
    Evaluates the log probability density for a batch of samples given by value.
    Parameters
    value – A batch of samples from the distribution.
    Returns
      an array with shape value.shape[:-self.event_shape]
    Return type
      numpy.ndarray
  property mean
    Mean of the distribution.
  property variance
    Variance of the distribution.
  cdf(value)
    The cumulative distribution function of this distribution.
    Parameters
    value – samples from this distribution.
    Returns
      output of the cumulative distribution function evaluated at value.
The inverse cumulative distribution function of this distribution.

Parameters

- **q** – quantile values, should belong to [0, 1].

Returns

the samples whose cdf values equals to \( q \).

entropy() Returns the entropy of the distribution.

### 3.2.27 LowRankMultivariateNormal

class **LowRankMultivariateNormal**(*loc*, *cov_factor*, *cov_diag*, *, validate_args=None*)

Bases: Distribution

arg_constraints = {'cov_diag': IndependentConstraint(Positive(lower_bound=0.0), 1), 'cov_factor': IndependentConstraint(Real(), 2), 'loc': RealVector(Real(), 1)}

support = RealVector(Real(), 1)

reparameterized_params = ['loc', 'cov_factor', 'cov_diag']

pytree_data_fields = ('loc', 'cov_factor', 'cov_diag', '_capacitance_tril')

property **mean**

Mean of the distribution.

**variance()**

Variance of the distribution.

**scale_tril()**

**covariance_matrix()**

**precision_matrix()**

**sample(key, sample_shape=())**

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions of size sample_shape of the returned sample will be filled with iid draws from the distribution instance.

Parameters

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

- **sample_shape** (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

**Return type**

numpy.ndarray

**log_prob(*args, **kwargs)**

Evaluates the log probability density for a batch of samples given by value.

Parameters

- **value** – A batch of samples from the distribution.
Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

entropy()
Returns the entropy of the distribution.

static infer_shapes(loc, cov_factor, cov_diag)
Infers batch_shape and event_shape given shapes of args to __init__().

### Note
This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters
- **args** – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- **kwargs** – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns
A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type
tuple

### 3.2.28 MatrixNormal
class MatrixNormal(loc, scale_tril_row, scale_tril_column, validate_args=None)
Bases: Distribution
Matrix variate normal distribution as described in [1] but with a lower_triangular parametrization, i.e. $U = \text{scale}_{\text{tril}, \text{row}} @ \text{scale}_{\text{tril}, \text{row}}^T$ and $V = \text{scale}_{\text{tril}, \text{column}} @ \text{scale}_{\text{tril}, \text{column}}^T$. The distribution is related to the multivariate normal distribution in the following way. If $X \sim MN(loc, U, V)$ then $\text{vec}(X) \sim MN(\text{vec}(loc), \text{kron}(V, U))$.

Parameters
- **loc** (array_like) – Location of the distribution.
- **scale_tril_row** (array_like) – Lower cholesky of rows correlation matrix.
- **scale_tril_column** (array_like) – Lower cholesky of columns correlation matrix.

References

arg_constraints = {'loc': RealVector(Real(), 1), 'scale_tril_column': LowerCholesky(), 'scale_tril_row': LowerCholesky()}
support = RealMatrix(Real(), 2)
reparametrized_params = ['loc', 'scale_tril_row', 'scale_tril_column']

property mean
Mean of the distribution.

sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

3.2.29 MultivariateNormal
class MultivariateNormal(loc=0.0, covariance_matrix=None, precision_matrix=None, scale_tril=None, validate_args=None, validate_args=None)

Bases: Distribution

arg_constraints = {'covariance_matrix': PositiveDefinite(), 'loc': RealVector(Real(), 1), 'precision_matrix': PositiveDefinite(), 'scale_tril': LowerCholesky()}
support = RealVector(Real(), 1)

reparametrized_params = ['loc', 'covariance_matrix', 'precision_matrix', 'scale_tril']
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• sample_shape (tuple) – the sample shape for the distribution.
Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

covariance_matrix()
covariance_matrix()

property mean
  Mean of the distribution.

property variance
  Variance of the distribution.

static infer_shapes(loc=(), covariance_matrix=None, precision_matrix=None, scale_tril=None)
  Infers batch_shape and event_shape given shapes of args to __init__().

  Parameters
  • *args – Positional args replacing each input arg with a tuple representing the sizes of each
tensor input.
  • **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each
tensor input.

  Returns
  A pair (batch_shape, event_shape) of the shapes of a distribution that would be created
  with input args of the given shapes.

  Return type
tuple

distance

  Returns the entropy of the distribution.
### 3.2.30 MultivariateStudentT

```python
class MultivariateStudentT(df, loc=0.0, scale_tril=None, validate_args=None)
```

#### Bases:
```
Distribution
```

#### arg_constraints
```
{'df': Positive(lower_bound=0.0), 'loc': RealVector(Real(), 1), 'scale_tril': LowerCholesky()}
```

#### support
```
RealVector(Real(), 1)
```

#### reparametrized_params
```
['df', 'loc', 'scale_tril']
```

#### pytree_data_fields
```
('df', 'loc', 'scale_tril', '_chi2')
```

#### sample(key, sample_shape=())

Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- `key` ([`jax.random.PRNGKey`]) – the rng_key key to be used for the distribution.
- `sample_shape` (`tuple`) – the sample shape for the distribution.

**Returns**

an array of shape `sample_shape + batch_shape + event_shape`

**Return type**
```
numpy.ndarray
```

#### log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by `value`.

**Parameters**

- `value` – A batch of samples from the distribution.

**Returns**

an array with shape `value.shape[:-self.event_shape]`

**Return type**
```
numpy.ndarray
```

#### covariance_matrix()

#### precision_matrix()

#### property mean

Mean of the distribution.

#### property variance

Variance of the distribution.

#### static infer_shapes(df, loc, scale_tril)

Infers `batch_shape` and `event_shape` given shapes of args to `__init__()`.

---

#### Note

This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.
Parameters

- **args** – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- **kwargs** – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns

A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type

tuple

### 3.2.31 Normal

**class Normal** (loc=0.0, scale=1.0, *, validate_args=None)

Bases: Distribution

**arg_constraints** = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}

**support** = Real()

reparametrized_params = ['loc', 'scale']

**sample**(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

- **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

**log_prob**(args, kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape value.shape[:-self.event_shape]

Return type

numpy.ndarray

**cdf**(value)

The cumulative distribution function of this distribution.

Parameters

- **value** – samples from this distribution.
Returns
output of the cumulative distribution function evaluated at value.

log_cdf(value)

icdf(q)
The inverse cumulative distribution function of this distribution.

Parameters
q – quantile values, should belong to [0, 1].

Returns
the samples whose cdf values equals to q.

property mean
Mean of the distribution.

property variance
Variance of the distribution.

entropy()
Returns the entropy of the distribution.

3.2.32 Pareto
class Pareto(scale, alpha, *, validate_args=None)
Bases: TransformedDistribution

arg_constraints = {'alpha': Positive(lower_bound=0.0), 'scale': Positive(lower_bound=0.0)}

reparametrized_params = ['scale', 'alpha']

property mean
Mean of the distribution.

property variance
Variance of the distribution.

property support
cdf(value)
The cumulative distribution function of this distribution.

Parameters
value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.

icdf(q)
The inverse cumulative distribution function of this distribution.

Parameters
q – quantile values, should belong to [0, 1].

Returns
the samples whose cdf values equals to q.
entropy()

Returns the entropy of the distribution.

### 3.2.33 RelaxedBernoulli

RelaxedBernoulli(temperature, probs=None, logits=None, *, validate_args=None)

### 3.2.34 RelaxedBernoulliLogits

class RelaxedBernoulliLogits(temperature, logits, *, validate_args=None)

Bases: TransformedDistribution

arg_constraints = {'logits': Real(), 'temperature': Positive(lower_bound=0.0)}
support = UnitInterval(lower_bound=0.0, upper_bound=1.0)

### 3.2.35 SoftLaplace

class SoftLaplace(loc, scale, *, validate_args=None)

Bases: Distribution

Smooth distribution with Laplace-like tail behavior.

This distribution corresponds to the log-convex density:

\[
\begin{align*}
    z &= (\text{value} - \text{loc}) / \text{scale} \\
    \log\text{prob} &= \log(2 / \pi) - \log(\text{scale}) - \logaddexp(z, -z)
\end{align*}
\]

Like the Laplace density, this density has the heaviest possible tails (asymptotically) while still being log-convex. Unlike the Laplace distribution, this distribution is infinitely differentiable everywhere, and is thus suitable for HMC and Laplace approximation.

**Parameters**

- `loc` – Location parameter.
- `scale` – Scale parameter.

arg_constraints = {'loc': Real(), 'scale': Positive(lower_bound=0.0)}
support = Real()
reparametrized_params = ['loc', 'scale']

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

**Parameters**

- `value` – A batch of samples from the distribution.

**Returns**

an array with shape `value.shape[:-self.event_shape]`

**Return type**

`numpy.ndarray`
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by \(\text{sample\_shape} + \text{batch\_shape} + \text{event\_shape}\). Note that when \(\text{sample\_shape}\) is non-empty, leading dimensions (of size \(\text{sample\_shape}\)) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns
an array of shape \(\text{sample\_shape} + \text{batch\_shape} + \text{event\_shape}\)

Return type
numpy.ndarray
cdf(value)

The cumulative distribution function of this distribution.

Parameters
- **value** – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at \(value\).
icdf(value)

The inverse cumulative distribution function of this distribution.

Parameters
- **q** – quantile values, should belong to \([0, 1]\).

Returns
the samples whose cdf values equals to \(q\).

property mean

Mean of the distribution.

property variance

Variance of the distribution.

3.2.36 StudentT
class StudentT(df, loc=0.0, scale=1.0, *, validate_args=None)

Bases: Distribution

arg_constraints = {'df': Positive(lower_bound=0.0), 'loc': Real(), 'scale': Positive(lower_bound=0.0)}
support = Real()
deparametrized_params = ['df', 'loc', 'scale']
pytree_data_fields = ('df', 'loc', 'scale', '_chi2')
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by \(\text{sample\_shape} + \text{batch\_shape} + \text{event\_shape}\). Note that when \(\text{sample\_shape}\) is non-empty, leading dimensions (of size \(\text{sample\_shape}\)) of the returned sample will be filled with iid draws from the distribution instance.
Parameters

- **key** *(jax.random.PRNGKey)* – the rng_key key to be used for the distribution.
- **sample_shape** *(tuple)* – the sample shape for the distribution.

Returns

an array of shape `sample_shape + batch_shape + event_shape`

Return type

`numpy.ndarray`

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by `value`.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

`numpy.ndarray`

property mean

Mean of the distribution.

property variance

Variance of the distribution.

cdf(value)

The cumulative distribution function of this distribution.

Parameters

- **value** – samples from this distribution.

Returns

output of the cumulative distribution function evaluated at `value`.

icdf(q)

The inverse cumulative distribution function of this distribution.

Parameters

- **q** – quantile values, should belong to [0, 1].

Returns

the samples whose cdf values equals to `q`.

entropy()

Returns the entropy of the distribution.
3.2.37 Uniform

class Uniform(low=0.0, high=1.0, *, validate_args=None)
    Bases: Distribution
    arg_constraints = {'high': Dependent(), 'low': Dependent()}
    reparametrized_params = ['low', 'high']
    pytree_data_fields = ('low', 'high', '_support')

property support

sample(key, sample_shape=())

    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
    • sample_shape (tuple) – the sample shape for the distribution.

Returns
    an array of shape sample_shape + batch_shape + event_shape

Return type
    numpy.ndarray

log_prob(*args, **kwargs)

    Evaluates the log probability density for a batch of samples given by value.

Parameters
    value – A batch of samples from the distribution.

Returns
    an array with shape value.shape[:-self.event_shape]

Return type
    numpy.ndarray

cdf(value)

    The cumulative distribution function of this distribution.

Parameters
    value – samples from this distribution.

Returns
    output of the cumulative distribution function evaluated at value.

icdf(value)

    The inverse cumulative distribution function of this distribution.

Parameters
    q – quantile values, should belong to [0, 1].

Returns
    the samples whose cdf values equals to q.

property mean

    Mean of the distribution.
property variance
Variance of the distribution.

static infer_shapes(low=(), high=())
Infers batch_shape and event_shape given shapes of args to __init__().

```
      1 Note
This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained
in those inputs.
```

Parameters
- **args** – Positional args replacing each input arg with a tuple representing the sizes of each
tensor input.
- **kwargs** – Keywords mapping name of input arg to tuple representing the sizes of each
tensor input.

Returns
A pair (batch_shape, event_shape) of the shapes of a distribution that would be created
with input args of the given shapes.

Return type
tuple

entropy()
Returns the entropy of the distribution.

3.2.38 Weibull
class Weibull(scale, concentration, *, validate_args=None)
Bases: Distribution

arg_constraints = {'concentration': Positive(lower_bound=0.0), 'scale':
Positive(lower_bound=0.0)}
support = Positive(lower_bound=0.0)
reparametrized_params = ['scale', 'concentration']
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape.
Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned
sample will be filled with iid draws from the distribution instance.

Parameters
- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray
log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by `value`.

**Parameters**

- **value** – A batch of samples from the distribution.

**Returns**

an array with shape `value.shape[:-self.event_shape]`

**Return type**

`numpy.ndarray`

cdf(value)

The cumulative distribution function of this distribution.

**Parameters**

- **value** – samples from this distribution.

**Returns**

output of the cumulative distribution function evaluated at `value`.

**property mean**

Mean of the distribution.

**property variance**

Variance of the distribution.

**entropy()**

Returns the entropy of the distribution.

### 3.2.39 Wishart

class Wishart(concentration, scale_matrix=None, rate_matrix=None, scale_tril=None, *, validate_args=None)

Bases: `TransformedDistribution`

Wishart distribution for covariance matrices.

**Parameters**

- **concentration** – Positive concentration parameter analogous to the concentration of a `Gamma` distribution. The concentration must be larger than the dimensionality of the scale matrix.
- **scale_matrix** – Scale matrix analogous to the inverse rate of a `Gamma` distribution.
- **rate_matrix** – Rate matrix analogous to the rate of a `Gamma` distribution.
- **scale_tril** – Cholesky decomposition of the `scale_matrix`.

**arg_constraints** = {'concentration': Dependent(), 'rate_matrix': PositiveDefinite(), 'scale_matrix': PositiveDefinite(), 'scale_tril': LowerCholesky()}

**support** = PositiveDefinite()

**reparameterized_params** = ['scale_matrix', 'rate_matrix', 'scale_tril']

concentration()

scale_matrix()
rate_matrix()

scale_tril()

mean()
    Mean of the distribution.

variance()
    Variance of the distribution.

static infer_shapes(concentration=(), scale_matrix=None, rate_matrix=None, scale_tril=None)
    Infers batch_shape and event_shape given shapes of args to __init__().

    Note
    This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters
    • *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
    • **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns
    A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type
tuple

entropy()
    Returns the entropy of the distribution.

3.2.40 WishartCholesky

class WishartCholesky(concentration, scale_matrix=None, rate_matrix=None, scale_tril=None, *, validate_args=None)
Bases: Distribution

Cholesky factor of a Wishart distribution for covariance matrices.

Parameters
    • concentration – Positive concentration parameter analogous to the concentration of a Gamma distribution. The concentration must be larger than the dimensionality of the scale matrix.
    • scale_matrix – Scale matrix analogous to the inverse rate of a Gamma distribution.
    • rate_matrix – Rate matrix analogous to the rate of a Gamma distribution.
    • scale_tril – Cholesky decomposition of the scale_matrix.
arg_constraints = {'concentration': Dependent(), 'rate_matrix': PositiveDefinite(), 'scale_matrix': PositiveDefinite(), 'scale_tril': LowerCholesky()}

support = LowerCholesky()

reparametrized_params = ['scale_matrix', 'rate_matrix', 'scale_tril']

log_prob(*args, **kwargs)
   Evaluates the log probability density for a batch of samples given by value.

   Parameters
   value – A batch of samples from the distribution.

   Returns
   an array with shape value.shape[:-self.event_shape]

   Return type
   numpy.ndarray

scale_matrix()
rate_matrix()

sample(key, sample_shape=())
   Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

   Parameters
   - key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
   - sample_shape (tuple) – the sample shape for the distribution.

   Returns
   an array of shape sample_shape + batch_shape + event_shape

   Return type
   numpy.ndarray

mean()
Mean of the distribution.

variance()
Variance of the distribution.

static infer_shapes(concentration=(), scale_matrix=None, rate_matrix=None, scale_tril=None)
   Infers batch_shape and event_shape given shapes of args to __init__().

   Parameters
   - *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.

---

3.2. Continuous Distributions
**kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

**Returns**
A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

**Return type**
tuple

## 3.2.41 ZeroSumNormal

class ZeroSumNormal(scale, event_shape, *, validate_args=None)

Bases: TransformedDistribution

Zero Sum Normal distribution adapted from PyMC [1] as described in [2,3]. This is a Normal distribution where one or more axes are constrained to sum to zero (the last axis by default).

\[
ZSN(\sigma) = N(0, \sigma^2(I - \frac{1}{n}J))
\]

where \( J_{ij} = 1 \) and \( n = \text{number of zero-sum axes} \)

**Parameters**

- **scale** (array_like) – Standard deviation of the underlying normal distribution before the zerosum constraint is enforced.
- **event_shape** (tuple) – The event shape of the distribution, the axes of which get constrained to sum to zero.

**Example:**

```python
>>> from numpy.testing import assert_allclose
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC, NUTS

>>> N = 1000
>>> n_categories = 20
>>> rng_key = random.PRNGKey(0)
>>> key1, key2, key3 = random.split(rng_key, 3)
>>> category_ind = random.choice(key1, jnp.arange(n_categories), shape=(N,))
>>> beta = random.normal(key2, shape=(n_categories,))
>>> beta -= beta.mean(-1)
>>> y = 5 + beta[category_ind] + random.normal(key3, shape=(N,))

>>> def model(category_ind, y):  # category_ind is an indexed categorical variable, with 20 categories
...     N = len(category_ind)
...     alpha = numpyro.sample("alpha", dist.Normal(0, 2.5))
...     beta = numpyro.sample("beta", dist.ZeroSumNormal(1, event_shape=(n_categories,)))
...     sigma = numpyro.sample("sigma", dist.Exponential(1))
```

(continues on next page)
... with numpyro.plate("observations", N):
...     mu = alpha + beta[category_ind]
...     obs = numpyro.sample("obs", dist.Normal(mu, sigma), obs=y)
...     return obs

>>> nuts_kernel = NUTS(model=model, target_accept_prob=0.9)
>>> mcmc = MCMC(
...     sampler=nuts_kernel,
...     num_samples=1_000, num_warmup=1_000, num_chains=4
... )
>>> mcmc.run(random.PRNGKey(0), category_ind=category_ind, y=y)
>>> posterior_samples = mcmc.get_samples()
>>> assert_allclose(posterior_samples['beta'].sum(-1), 0, atol=1e-3)

References

arg_constraints = {'scale': Positive(lower_bound=0.0)}

reparameterized_params = ['scale']

property support

property mean
    Mean of the distribution.

property variance
    Variance of the distribution.

3.3 Discrete Distributions

3.3.1 Bernoulli

Bernoulli(probs=None, logits=None, *, validate_args=None)

3.3.2 BernoulliLogits

class BernoulliLogits(logits=None, *, validate_args=None)
    Bases: Distribution

    arg_constraints = {'logits': Real()}

    support = Boolean()

    has.enumerate_support = True
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape value.shape[:-self.event_shape]

Return type

numpy.ndarray

probs()

property mean

Mean of the distribution.

property variance

Variance of the distribution.

enumerate_support(expand=True)

Returns an array with shape len(support) x batch_shape containing all values in the support.

entropy()

Returns the entropy of the distribution.

### 3.3.3 BernoulliProbs

class BernoulliProbs(probs, *, validate_args=None)

Bases: Distribution

arg_constraints = {‘probs’: UnitInterval(lower_bound=0.0, upper_bound=1.0)}

support = Boolean()

has_enumerate_support = True

sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.

• **sample_shape** (*tuple*) – the sample shape for the distribution.

Returns

an array of shape `sample_shape + batch_shape + event_shape`

Return type

`numpy.ndarray`

```python
log_prob(*args, **kwargs)
```

Evaluates the log probability density for a batch of samples given by `value`.

Parameters

value – A batch of samples from the distribution.

Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

`numpy.ndarray`

```python
logits()
```

```python
property mean
```

Mean of the distribution.

```python
property variance
```

Variance of the distribution.

```python
enumerate_support(expand=True)
```

Returns an array with shape `len(support) x batch_shape` containing all values in the support.

```python
entropy()
```

Returns the entropy of the distribution.

### 3.3.4 BetaBinomial

```python
class BetaBinomial(concentration1, concentration0, total_count=1, *, validate_args=None)
```

Bases: `Distribution`

Compound distribution comprising of a beta-binomial pair. The probability of success (`probs` for the `Binomial` distribution) is unknown and randomly drawn from a `Beta` distribution prior to a certain number of Bernoulli trials given by `total_count`.

Parameters

- **concentration1** (*numpy.ndarray*) – 1st concentration parameter (alpha) for the Beta distribution.
- **concentration0** (*numpy.ndarray*) – 2nd concentration parameter (beta) for the Beta distribution.
- **total_count** (*numpy.ndarray*) – number of Bernoulli trials.

```python
arg_constraints = {'concentration0': Positive(lower_bound=0.0), 'concentration1': Positive(lower_bound=0.0), 'total_count': IntegerNonnegative(lower_bound=0)}
```

```python
has.enumerate_support = True
```
enumerate_support(expand=True)

Returns an array with shape \( \text{len}(\text{support}) \times \text{batch_shape} \) containing all values in the support.

pytree_data_fields = ('concentration1', 'concentration0', 'total_count', '_beta')

sample(key, sample_shape=())

Returns a sample from the distribution having shape given by \( \text{sample_shape} + \text{batch_shape} + \text{event_shape} \). Note that when \( \text{sample_shape} \) is non-empty, leading dimensions (of size \( \text{sample_shape} \)) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** (tuple) – the sample shape for the distribution.

Returns

an array of shape \( \text{sample_shape} + \text{batch_shape} + \text{event_shape} \)

Return type

numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape value.shape[:-self.event_shape]

Return type

numpy.ndarray

property mean

Mean of the distribution.

property variance

Variance of the distribution.

property support

3.3.5 Binomial

Binomial(total_count=1, probs=None, logits=None, *, validate_args=None)

3.3.6 BinomialLogits

class BinomialLogits(logits, total_count=1, *, validate_args=None)

Bases: Distribution

arg_constraints = {'logits': Real(), 'total_count': IntegerNonnegative(lower_bound=0)}

has.enumerate_support = True
enumerate_support(expand=True)
Returns an array with shape \( \text{len(support)} \times \text{batch\_shape} \) containing all values in the support.

sample(key, sample\_shape=())
Returns a sample from the distribution having shape given by \( \text{sample\_shape} + \text{batch\_shape} + \text{event\_shape} \).
Note that when \( \text{sample\_shape} \) is non-empty, leading dimensions (of size \( \text{sample\_shape} \)) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample\_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape \( \text{sample\_shape} + \text{batch\_shape} + \text{event\_shape} \)

Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
- value – A batch of samples from the distribution.

Returns
an array with shape \( \text{value\_shape}[:-\text{self\_event\_shape}] \)

Return type
numpy.ndarray

probs()

property mean
Mean of the distribution.

property variance
Variance of the distribution.

property support

3.3.7 BinomialProbs
class BinomialProbs(probs, total\_count=1, *, validate\_args=None)
Bases: Distribution

arg_constraints = {'probs': UnitInterval(lower_bound=0.0, upper_bound=1.0),
'total\_count': IntegerNonnegative(lower_bound=0)}

has\_enumerate\_support = True

sample(key, sample\_shape=())
Returns a sample from the distribution having shape given by \( \text{sample\_shape} + \text{batch\_shape} + \text{event\_shape} \).
Note that when \( \text{sample\_shape} \) is non-empty, leading dimensions (of size \( \text{sample\_shape} \)) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• **sample_shape** *(tuple)* – the sample shape for the distribution.

Returns
an array of shape `sample_shape + batch_shape + event_shape`

Return type
`numpy.ndarray`

`log_prob(*args, **kwargs)`
Evaluates the log probability density for a batch of samples given by `value`.

Parameters
`value` – A batch of samples from the distribution.

Returns
an array with shape `value.shape[:-self.event_shape]`

Return type
`numpy.ndarray`

`logits()`

**property** *mean*
Mean of the distribution.

**property** *variance*
Variance of the distribution.

**property** *support*

`enumerate_support(expand=True)`
Returns an array with shape `len(support) x batch_shape` containing all values in the support.

### 3.3.8 Categorical

**Categorical**(probs=`None`, logits=`None`, *, validate_args=`None`)

### 3.3.9 CategoricalLogits

**class** *CategoricalLogits*(logits, *, validate_args=`None`)
Bases: *Distribution*

```
arg_constraints = {'logits': RealVector(Real(), 1)}

has_enumerate_support = True

sample(key, sample_shape=())
```
Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• **key** *(jax.random.PRNGKey)* – the rng_key key to be used for the distribution.

• **sample_shape** *(tuple)* – the sample shape for the distribution.

Returns
an array of shape `sample_shape + batch_shape + event_shape`
Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

probs()

property mean
Mean of the distribution.

property variance
Variance of the distribution.

property support
enumerate_support(expand=True)
Returns an array with shape len(support) x batch_shape containing all values in the support.

entropy()
Returns the entropy of the distribution.

3.3.10 CategoricalProbs

class CategoricalProbs(probs, *, validate_args=None)
Bases: Distribution

arg_constraints = {'probs': Simplex()}

has.enumerate_support = True

sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

• sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray
log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

logits()

property mean
Mean of the distribution.

property variance
Variance of the distribution.

property support

enumerate_support(expand=True)

Returns an array with shape len(support) x batch_shape containing all values in the support.

entropy()

Returns the entropy of the distribution.

3.3.11 DirichletMultinomial

class DirichletMultinomial(concentration, total_count=1, *, validate_args=None)

Bases: Distribution

Compound distribution comprising of a dirichlet-multinomial pair. The probability of classes (probs for the Multinomial distribution) is unknown and randomly drawn from a Dirichlet distribution prior to a certain number of Categorical trials given by total_count.

Parameters
• concentration (numpy.ndarray) – concentration parameter (alpha) for the Dirichlet distribution.
• total_count (numpy.ndarray) – number of Categorical trials.

arg_constraints = {
    'concentration': IndependentConstraint(Positive(lower_bound=0.0), 1), 
    'total_count': IntegerNonnegative(lower_bound=0)}

pytree_data_fields = ('concentration', '_dirichlet')

pytree_aux_fields = ('total_count',)

sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- **sample_shape** *(tuple)* – the sample shape for the distribution.

  **Returns**
  an array of shape `sample_shape + batch_shape + event_shape`

  **Return type**
  `numpy.ndarray`

**log_prob**(*args, **kwargs*)
Evaluates the log probability density for a batch of samples given by `value`.

  **Parameters**
  `value` – A batch of samples from the distribution.

  **Returns**
  an array with shape `value.shape[:-self.event_shape]`

  **Return type**
  `numpy.ndarray`

**property mean**
Mean of the distribution.

**property variance**
Variance of the distribution.

**property support**

**static infer_shapes**(concentration, total_count=())
Infers `batch_shape` and `event_shape` given shapes of args to `__init__()`.

```
: Note

This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.
```

**Parameters**

- *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

**Returns**
A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

**Return type**
`tuple`
### 3.3.12 DiscreteUniform

```python
class DiscreteUniform(low=0, high=1, *, validate_args=None)
    Bases: Distribution
    arg_constraints = {'high': Dependent(), 'low': Dependent()}
    has.enumerate.support = True
    pytree.data.fields = ('low', 'high', '_support')

    property support
    
    sample(key, sample_shape=())
        Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

        Parameters
        • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
        • sample_shape (tuple) – the sample shape for the distribution.

        Returns
        an array of shape sample_shape + batch_shape + event_shape

        Return type
        numpy.ndarray

    log_prob(*args, **kwargs)
        Evaluates the log probability density for a batch of samples given by value.

        Parameters
        value – A batch of samples from the distribution.

        Returns
        an array with shape value.shape[:-self.event_shape]

        Return type
        numpy.ndarray

    cdf(value)
        The cummulative distribution function of this distribution.

        Parameters
        value – samples from this distribution.

        Returns
        output of the cummulative distribution function evaluated at value.

    icdf(value)
        The inverse cumulative distribution function of this distribution.

        Parameters
        q – quantile values, should belong to [0, 1].

        Returns
        the samples whose cdf values equals to q.

    property mean
        Mean of the distribution.
```
property variance
Variance of the distribution.

def enumerate_support(expand=True)
Returns an array with shape len(support) x batch_shape containing all values in the support.

def entropy()
Returns the entropy of the distribution.

### 3.3.13 GammaPoisson

class GammaPoisson(concentration, rate=1.0, *, validate_args=None)
Bases: Distribution

Compound distribution comprising of a gamma-poisson pair, also referred to as a gamma-poisson mixture. The rate parameter for the Poisson distribution is unknown and randomly drawn from a Gamma distribution.

Parameters
- concentration (numpy.ndarray) – shape parameter (alpha) of the Gamma distribution.
- rate (numpy.ndarray) – rate parameter (beta) for the Gamma distribution.

arg_constraints = {'concentration': Positive(lower_bound=0.0), 'rate': Positive(lower_bound=0.0)}
support = IntegerNonnegative(lower_bound=0)
pytree_data_fields = ('concentration', 'rate', '_gamma')
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray
property mean
Mean of the distribution.

property variance
Variance of the distribution.

cdf(value)
The cumulative distribution function of this distribution.

Parameters
value – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at value.

3.3.14 Geometric

Geometric(probs=None, logits=None, *, validate_args=None)

3.3.15 GeometricLogits

class GeometricLogits(logits, *, validate_args=None)
Bases: Distribution

arg_constraints = {'logits': Real()}
support = IntegerNonnegative(lower_bound=0)

probs()
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
• sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray
property `mean`
Mean of the distribution.

property `variance`
Variance of the distribution.

`entropy()`
Returns the entropy of the distribution.

### 3.3.16 GeometricProbs

```python
class GeometricProbs(probs, *, validate_args=None):
    Bases: Distribution

    arg_constraints = {'probs': UnitInterval(lower_bound=0.0, upper_bound=1.0)}
    support = IntegerNonnegative(lower_bound=0)

    sample(key, sample_shape=())
    Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

    Parameters
    • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
    • sample_shape (tuple) – the sample shape for the distribution.

    Returns
    an array of shape `sample_shape + batch_shape + event_shape`

    Return type
    numpy.ndarray

    log_prob(*args, **kwargs)
    Evaluates the log probability density for a batch of samples given by `value`.

    Parameters
    • value – A batch of samples from the distribution.

    Returns
    an array with shape `value.shape[:-self.event_shape]`

    Return type
    numpy.ndarray
```

logits()

property `mean`
Mean of the distribution.

property `variance`
Variance of the distribution.

`entropy()`
Returns the entropy of the distribution.
3.3.17 Multinomial

**Multinomial** *(total_count=1, probs=None, logits=None, *, total_count_max=None, validate_args=None)*

Multinomial distribution.

**Parameters**

- **total_count** – number of trials. If this is a JAX array, it is required to specify `total_count_max`.
- **probs** – event probabilities
- **logits** – event log probabilities
- **total_count_max** *(int)* – the maximum number of trials, i.e. `max(total_count)`

3.3.18 MultinomialLogits

class **MultinomialLogits**(logits, total_count=1, *, total_count_max=None, validate_args=None)

**Bases:** Distribution

**arg_constraints** = {'logits': RealVector(Real(), 1), 'total_count': IntegerNonnegative(lower_bound=0)}

**pytree_data_fields** = ('logits',)

**pytree_aux_fields** = ('total_count', 'total_count_max')

**sample**(key, sample_shape=())

Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- **key** *(jax.random.PRNGKey)* – the rng_key key to be used for the distribution.
- **sample_shape** *(tuple)* – the sample shape for the distribution.

**Returns**

an array of shape `sample_shape + batch_shape + event_shape`

**Return type**

numpy.ndarray

**log_prob**(args, **kwargs)

Evaluates the log probability density for a batch of samples given by `value`.

**Parameters**

- **value** – A batch of samples from the distribution.

**Returns**

an array with shape `value.shape[:-self.event_shape]`

**Return type**

numpy.ndarray

**probs**()
**property mean**
Mean of the distribution.

**property variance**
Variance of the distribution.

**property support**

**static infer_shapes**(logits, total_count)
Infers batch_shape and event_shape given shapes of args to __init__().

### Note

This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

**Parameters**

- • *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- • **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

**Returns**

A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

**Return type**

tuple

### 3.3.19 MultinomialProbs

**class MultinomialProbs**(probs, total_count=1, *, total_count_max=None, validate_args=None)

**Bases:** Distribution

**arg_constraints** = {'probs': Simplex(), 'total_count': IntegerNonnegative(lower_bound=0)}

**pytree_data_fields** = ('probs',)

**pytree_aux_fields** = ('total_count', 'total_count_max')

**sample**(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- • sample_shape (tuple) – the sample shape for the distribution.

**Returns**

an array of shape sample_shape + batch_shape + event_shape
Return type

numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters

value – A batch of samples from the distribution.

Returns

an array with shape value.shape[:-self.event_shape]

Return type

numpy.ndarray

logits()

property mean

Mean of the distribution.

property variance

Variance of the distribution.

property support

static infer_shapes(probs, total_count)

Infers batch_shape and event_shape given shapes of args to __init__().

Note

This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters

• *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.

• **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns

A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type

tuple
3.3.20 OrderedLogistic

class OrderedLogistic(predictor, cutpoints, *, validate_args=None)

Bases: CategoricalProbs

A categorical distribution with ordered outcomes.

References:

1. Stan Functions Reference, v2.20 section 12.6, Stan Development Team

Parameters

- predictor (numpy.ndarray) – prediction in real domain; typically this is output of a linear model.
- cutpoints (numpy.ndarray) – positions in real domain to separate categories.

arg_constraints = {'cutpoints': OrderedVector(), 'predictor': Real()}

static infer_shapes(predictor, cutpoints)

Infers batch_shape and event_shape given shapes of args to __init__().

Note

This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters

- *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
- **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.

Returns

A pair (batch_shape, event_shape) of the shapes of a distribution that would be created with input args of the given shapes.

Return type
tuple

entropy()

Returns the entropy of the distribution.

3.3.21 NegativeBinomial

NegativeBinomial(total_count, probs=None, logits=None, *, validate_args=None)
3.3.22 NegativeBinomialLogits

class NegativeBinomialLogits(total_count, logits, *, validate_args=None)
    Bases: GammaPoisson
    arg_constraints = {'logits': Real(), 'total_count': Positive(lower_bound=0.0)}
    support = IntegerNonnegative(lower_bound=0)

    log_prob(*args, **kwargs)
    Evaluates the log probability density for a batch of samples given by value.
    Parameters
    value – A batch of samples from the distribution.
    Returns
    an array with shape value.shape[:-self.event_shape]
    Return type
    numpy.ndarray

3.3.23 NegativeBinomialProbs

class NegativeBinomialProbs(total_count, probs, *, validate_args=None)
    Bases: GammaPoisson
    arg_constraints = {'probs': UnitInterval(lower_bound=0.0, upper_bound=1.0),
                      'total_count': Positive(lower_bound=0.0)}
    support = IntegerNonnegative(lower_bound=0)

3.3.24 NegativeBinomial2

class NegativeBinomial2(mean, concentration, *, validate_args=None)
    Bases: GammaPoisson
    Another parameterization of GammaPoisson with rate is replaced by mean.
    arg_constraints = {'concentration': Positive(lower_bound=0.0), 'mean':
                      Positive(lower_bound=0.0)}
    support = IntegerNonnegative(lower_bound=0)
    pytree_data_fields = ('concentration',)

3.3.25 Poisson

class Poisson(rate, *, is_sparse=False, validate_args=None)
    Bases: Distribution
    Creates a Poisson distribution parameterized by rate, the rate parameter.
    Samples are nonnegative integers, with a pmf given by
    \[ P(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \]
Parameters

- **rate** (*numpy.ndarray*) – The rate parameter
- **is_sparse** (*bool*) – Whether to assume value is mostly zero when computing `log_prob()`, which can speed up computation when data is sparse.

```python
arg_constraints = {'rate': Positive(lower_bound=0.0)}
support = IntegerNonnegative(lower_bound=0)
pytree_aux_fields = ('is_sparse',)
sample(key, sample_shape=())
Returns a sample from the distribution having shape given by `sample_shape + batch_shape + event_shape`. Note that when `sample_shape` is non-empty, leading dimensions (of size `sample_shape`) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

- **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.
- **sample_shape** (*tuple*) – the sample shape for the distribution.

Returns
an array of shape `sample_shape + batch_shape + event_shape`

Return type
*numpy.ndarray*
```
```

log_prob(*args, **kwargs)
Evaluates the log probability density for a batch of samples given by `value`.

Parameters

- **value** – A batch of samples from the distribution.

Returns
an array with shape `value.shape[:-self.event_shape]`

Return type
*numpy.ndarray*
```
```

property mean
Mean of the distribution.

property variance
Variance of the distribution.
```
```
cdf(value)
The cumulative distribution function of this distribution.

Parameters

- **value** – samples from this distribution.

Returns
output of the cumulative distribution function evaluated at `value`. 
```
```
3.3.26 ZeroInflatedDistribution

ZeroInflatedDistribution\((base_{dist}, *, gate=\text{None}, gate_{logits}=\text{None}, validate\_args=\text{None})\)

Generic Zero Inflated distribution.

Parameters

- `base_{dist}` (Distribution) – the base distribution.
- `gate` (numpy.ndarray) – probability of extra zeros given via a Bernoulli distribution.
- `gate_{logits}` (numpy.ndarray) – logits of extra zeros given via a Bernoulli distribution.

3.3.27 ZeroInflatedPoisson

class ZeroInflatedPoisson\((gate, rate=1.0, *, validate\_args=\text{None})\)

Bases: ZeroInflatedProbs

A Zero Inflated Poisson distribution.

Parameters

- `gate` (numpy.ndarray) – probability of extra zeros.
- `rate` (numpy.ndarray) – rate of Poisson distribution.

arg_constraints = \{'gate': UnitInterval(lower_bound=0.0, upper_bound=1.0), 'rate': Positive(lower_bound=0.0)\}

support = IntegerNonnegative(lower_bound=0)

pytree_data_fields = ('rate',)

3.3.28 ZeroInflatedNegativeBinomial2

ZeroInflatedNegativeBinomial2\((mean, concentration, *, gate=\text{None}, gate_{logits}=\text{None}, validate\_args=\text{None})\)

3.4 Mixture Distributions

3.4.1 Mixture

Mixture\((mixing\_distribution, component\_distributions, *, validate\_args=\text{None})\)

A marginalized finite mixture of component distributions

The returned distribution will be either a:

1. MixtureGeneral, when component\_distributions is a list, or
2. MixtureSameFamily, when component\_distributions is a single distribution.

and more details can be found in the documentation for each of these classes.

Parameters

- `mixing\_distribution` – A Categorical specifying the weights for each mixture components. The size of this distribution specifies the number of components in the mixture, `mixture\_size`. 

• **component_distributions** – Either a list of component distributions or a single vectorized distribution. When a list is provided, the number of elements must equal `mixture_size`. Otherwise, the last batch dimension of the distribution must equal `mixture_size`.

**Returns**
The mixture distribution.

### 3.4.2 MixtureSameFamily

```python
class MixtureSameFamily(mixing_distribution, component_distribution, *, validate_args=None)
Bases: _MixtureBase

A finite mixture of component distributions from the same family

This mixture only supports a mixture of component distributions that are all of the same family. The different components are specified along the last batch dimension of the input `component_distribution`. If you need a mixture of distributions from different families, use the more general implementation in `MixtureGeneral`.

**Parameters**

- **mixing_distribution** – A `Categorical` specifying the weights for each mixture components. The size of this distribution specifies the number of components in the mixture, `mixture_size`.

- **component_distribution** – A single vectorized `Distribution`, whose last batch dimension equals `mixture_size` as specified by `mixing_distribution`.

**Example**

```python
gm
>>> import jax
>>> import jax.numpy as jnp
>>> import numpyro.distributions as dist

>>> mixing_dist = dist.Categorical(probs=jnp.ones(3) / 3.)
>>> component_dist = dist.Normal(loc=jnp.zeros(3), scale=jnp.ones(3))
>>> mixture = dist.MixtureSameFamily(mixing_dist, component_dist)

>>> mixture.sample(jax.random.PRNGKey(42)).shape
```

Pytree data fields:
- `_mixing_distribution`
- `_component_distribution`

Pytree aux fields:
- `_mixture_size`

**property component_distribution**

Return the vectorized distribution of components being mixed.

**Returns**
Component distribution

**Return type**
`Distribution`

**property support**

**property is_discrete**

**property component_mean**
property component_variance

component_cdf(samples)

component_sample(key, sample_shape=())

component_log_probs(value)

3.4.3 MixtureGeneral

class MixtureGeneral(mixing_distribution, component_distributions, *, support=None, validate_args=None)

Bases: _MixtureBase

A finite mixture of component distributions from different families

If all of the component distributions are from the same family, the more specific implementation in MixtureSameFamily will be somewhat more efficient.

Parameters

- **mixing_distribution** – A Categorical specifying the weights for each mixture components. The size of this distribution specifies the number of components in the mixture, `mixture_size`.

- **component_distributions** – A list of `mixture_size` Distribution objects.

- **support** – A `Constraint` object specifying the support of the mixture distribution. If not provided, the support will be inferred from the component distributions.

Example

```python
>>> import jax
>>> import jax.numpy as jnp
>>> import numpyro.distributions as dist
>>> mixing_dist = dist.Categorical(probs=jnp.ones(3) / 3.)
>>> component_dists = [
...     dist.Normal(loc=0.0, scale=1.0),
...     dist.Normal(loc=-0.5, scale=0.3),
...     dist.Normal(loc=0.6, scale=1.2),
... ]
>>> mixture = dist.MixtureGeneral(mixing_dist, component_dists)
>>> mixture.sample(jax.random.PRNGKey(42)).shape
()```

```python
>>> import jax
>>> import jax.numpy as jnp
>>> import numpyro.distributions as dist
>>> mixing_dist = dist.Categorical(probs=jnp.ones(2) / 2.)
>>> component_dists = [
...     dist.Normal(loc=0.0, scale=1.0),
...     dist.HalfNormal(scale=0.3),
... ]
>>> mixture = dist.MixtureGeneral(mixing_dist, component_dists, support=dist.constraints.real)
>>> mixture.sample(jax.random.PRNGKey(42)).shape
()```

```
pytree_data_fields = ('_mixing_distribution', '_component_distributions', '_support')

pytree_aux_fields = ('_mixture_size',)

property component_distributions
    The list of component distributions in the mixture
    Returns
    The list of component distributions
    Return type
    list[Distribution]

property support

property is_discrete

property component_mean

property component_variance

component_cdf(samples)

component_sample(key, sample_shape=())

component_log_probs(value)
```

### 3.5 Directional Distributions

#### 3.5.1 ProjectedNormal

```python
class ProjectedNormal(concentration, *, validate_args=None)
    Bases: Distribution

Projected isotropic normal distribution of arbitrary dimension.

This distribution over directional data is qualitatively similar to the von Mises and von Mises-Fisher distributions, but permits tractable variational inference via reparametrized gradients.

To use this distribution with autoguides and HMC, use `handlers.reparam` with a `ProjectedNormalReparam` reparametrizer in the model, e.g.:

```python
@handlers.reparam(config={"direction": ProjectedNormalReparam()})
def model():
    direction = numpyro.sample("direction",
                               ProjectedNormal(zeros(3)))
    ...
```

⚠️ Note

This implements `log_prob()` only for dimensions {2,3}.
arg_constraints = {'concentration': RealVector(Real(), 1)}
reparametrized_params = ['concentration']
support = Sphere()

property mean
  Note this is the mean in the sense of a centroid in the submanifold that minimizes expected squared geodesic distance.

property mode

sample(key, sample_shape=())
  Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
  • key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
  • sample_shape (tuple) – the sample shape for the distribution.

Returns
  an array of shape sample_shape + batch_shape + event_shape

Return type
  numpy.ndarray

log_prob(value)
  Evaluates the log probability density for a batch of samples given by value.

Parameters
  value – A batch of samples from the distribution.

Returns
  an array with shape value.shape[:-self.event_shape]

Return type
  numpy.ndarray

static infer_shapes(concentration)
  Infers batch_shape and event_shape given shapes of args to __init__().

**Note**
This assumes distribution shape depends only on the shapes of tensor inputs, not in the data contained in those inputs.

Parameters
  • *args – Positional args replacing each input arg with a tuple representing the sizes of each tensor input.
  • **kwargs – Keywords mapping name of input arg to tuple representing the sizes of each tensor input.
Returns

A pair `(batch_shape, event_shape)` of the shapes of a distribution that would be created with input args of the given shapes.

Return type
tuple

3.5.2 SineBivariateVonMises

class SineBivariateVonMises(phi_loc, psi_loc, phi_concentration, psi_concentration, correlation=None, weighted_correlation=None, validate_args=None)

Bases: Distribution

Unimodal distribution of two dependent angles on the 2-torus \((S^1 \otimes S^1)\) given by

\[
C^{-1} \exp(\kappa_1 \cos(x_1 - \mu_1) + \kappa_2 \cos(x_2 - \mu_2) + \rho \sin(x_1 - \mu_1) \sin(x_2 - \mu_2))
\]

and

\[
C = (2\pi)^2 \sum_{i=0}^{\infty} \left( \frac{2^i}{i!} \right) \left( \frac{\rho^2}{4\kappa_1\kappa_2} \right)^i I_i(\kappa_1)I_i(\kappa_2),
\]

where \(I_i(\cdot)\) is the modified bessel function of first kind, mu’s are the locations of the distribution, kappa’s are the concentration, and rho gives the correlation between angles \(x_1\) and \(x_2\). This distribution is helpful for modeling coupled angles such as torsion angles in peptide chains.

To infer parameters, use NUTS or HMC with priors that avoid parameterizations where the distribution becomes bimodal; see note below.

** Note

Sample efficiency drops as

\[
\frac{\rho}{\kappa_1\kappa_2} \to 1
\]

because the distribution becomes increasingly bimodal. To avoid bimodality use the `weighted_correlation` parameter with a skew away from one (e.g., Beta(1,3)). The `weighted_correlation` should be in [0,1].

** Note

The correlation and weighted_correlation params are mutually exclusive.

** Note

In the context of SVI, this distribution can be used as a likelihood but not for latent variables.

** References: **

Parameters

- `phi_loc (np.ndarray)` – location of first angle
- `psi_loc (np.ndarray)` – location of second angle
- `phi_concentration (np.ndarray)` – concentration of first angle
- `psi_concentration (np.ndarray)` – concentration of second angle
- `correlation (np.ndarray)` – correlation between the two angles
- `weighted_correlation (np.ndarray)` – set correlation to `weighted_corr * sqrt(phi_conc*psi_conc)` to avoid bimodality (see note). The `weighted_correlation` should be in [0,1].

```
arg_constraints = {
    'correlation': Real(),
    'phi_concentration': Positive(lower_bound=0.0),
    'phi_loc': Circular(lower_bound=-3.141592653589793, upper_bound=3.141592653589793),
    'psi_concentration': Positive(lower_bound=0.0),
    'psi_loc': Circular(lower_bound=-3.141592653589793, upper_bound=3.141592653589793)}
```

```
support = IndependentConstraint(Circular(lower_bound=-3.141592653589793, upper_bound=3.141592653589793), 1)
```

```
max_sample_iter = 1000
```

```
norm_const()
```

```
log_prob(*args, **kwargs)
```

Evaluates the log probability density for a batch of samples given by `value`.

Parameters

- `value` – A batch of samples from the distribution.

Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

numpy.ndarray

```
sample(key, sample_shape=())
```

** References: **


```
property mean
```

Computes circular mean of distribution. Note: same as location when mapped to support [-pi, pi]

### 3.5.3 SineSkewed

```
class SineSkewed(base_dist: Distribution, skewness, *, validate_args=None)
```

Bases: `Distribution`

Sine-skewing [1] is a procedure for producing a distribution that breaks pointwise symmetry on a torus distribution. The new distribution is called the Sine Skewed X distribution, where X is the name of the (symmetric) base distribution. Torus distributions are distributions with support on products of circles (i.e., $\otimes S^1$ where $S^1 = [-\pi, \pi]$). So, a 0-torus is a point, the 1-torus is a circle, and the 2-torus is commonly associated with the donut shape.
The sine skewed X distribution is parameterized by a weight parameter for each dimension of the event of X. For example with a von Mises distribution over a circle (1-torus), the sine skewed von Mises distribution has one skew parameter. The skewness parameters can be inferred using HMC or NUTS. For example, the following will produce a prior over skewness for the 2-torus:

```python
@numpyro.handlers.reparam(config={'phi_loc': CircularReparam(), 'psi_loc': CircularReparam()})
def model(obs):
    # Sine priors
    phi_loc = numpyro.sample('phi_loc', VonMises(pi, 2.))
    psi_loc = numpyro.sample('psi_loc', VonMises(-pi / 2, 2.))
    phi_conc = numpyro.sample('phi_conc', Beta(1., 1.))
    psi_conc = numpyro.sample('psi_conc', Beta(1., 1.))
    corr_scale = numpyro.sample('corr_scale', Beta(2., 5.))

    # Skewing prior
    ball_trans = L1BallTransform()
    skewness = numpyro.sample('skewness', Normal(0, 0.5).expand((2,)))
    skewness = ball_trans(skewness)  # constraint sum |skewness_i| <= 1

    with numpyro.plate('obs_plate'):
        sine = SineBivariateVonMises(phi_loc=phi_loc, psi_loc=psi_loc,
                                    phi_concentration=70 * phi_conc,
                                    psi_concentration=70 * psi_conc,
                                    weighted_correlation=corr_scale)

        return numpyro.sample('phi_psi', SineSkewed(sine, skewness), obs=obs)
```

To ensure the skewing does not alter the normalization constant of the (sine bivariate von Mises) base distribution the skewness parameters are constraint. The constraint requires the sum of the absolute values of skewness to be less than or equal to one. We can use the L1BallTransform to achieve this.

In the context of SVI, this distribution can freely be used as a likelihood, but use as latent variables it will lead to slow inference for 2 and higher dim toruses. This is because the base_dist cannot be reparameterized.

**Note**

An event in the base distribution must be on a d-torus, so the event_shape must be \((d,)\).

**Note**

For the skewness parameter, it must hold that the sum of the absolute value of its weights for an event must be less than or equal to one. See eq. 2.1 in [1].

**References:**

1. Sine-skewed toroidal distributions and their application in protein bioinformatics

Parameters

- **base_dist** *(numpyro.distributions.Distribution) – base density on a d-dimensional torus. Supported base distributions include: 1D VonMises, SineBivariateVonMises, 1D ProjectedNormal, and Uniform (-pi, pi).*
• **skewness** (*jax.numpy.array*) – skewness of the distribution.

    arg_constraints = {"skewness": L1Ball()}

    pytree_data_fields = ("base_dist", "skewness")

    support = IndependentConstraint(Circular(lower_bound=-3.141592653589793, upper_bound=3.141592653589793), 1)

    sample(key, sample_shape=())

    Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

    Parameters

    • **key** (*jax.random.PRNGKey*) – the rng_key key to be used for the distribution.

    • **sample_shape** (*tuple*) – the sample shape for the distribution.

    Returns

    an array of shape sample_shape + batch_shape + event_shape

    Return type

    numpy.ndarray

    log_prob(value)

    Evaluates the log probability density for a batch of samples given by value.

    Parameters

    • **value** – A batch of samples from the distribution.

    Returns

    an array with shape value.shape[:-self.event_shape]

    Return type

    numpy.ndarray

    property mean

    Mean of the base distribution

### 3.5.4 VonMises

class VonMises(loc, concentration, *, validate_args=None)

    Bases: Distribution

    The von Mises distribution, also known as the circular normal distribution.

    This distribution is supported by a circular constraint from -pi to +pi. By default, the circular support behaves like constraints.interval(-math.pi, math.pi). To avoid issues at the boundaries of this interval during sampling, you should reparameterize this distribution using handlers.reparam with a CircularReparam reparameterizer in the model, e.g.:

    ```python
    @handlers.reparam(config={"direction": CircularReparam()})
    def model():
        direction = numpyro.sample("direction", VonMises(0.0, 4.0))
        ...
    ```
arg_constraints = {'concentration': Positive(lower_bound=0.0), 'loc': Real()}
reparametrized_params = ['loc']
support = Circular(lower_bound=-3.141592653589793, upper_bound=3.141592653589793)
sample(key, sample_shape=())
    Generate sample from von Mises distribution

Parameters
    • key – random number generator key
    • sample_shape – shape of samples

Returns
    samples from von Mises

log_prob(*args, **kwargs)
    Evaluates the log probability density for a batch of samples given by value.

Parameters
    value – A batch of samples from the distribution.

Returns
    an array with shape value.shape[:-self.event_shape]

Return type
    numpy.ndarray

property mean
    Computes circular mean of distribution. NOTE: same as location when mapped to support [-pi, pi]

property variance
    Computes circular variance of distribution

3.6 Truncated Distributions

3.6.1 LeftTruncatedDistribution
class LeftTruncatedDistribution(base_dist, low=0.0, *, validate_args=None)
Bases: Distribution

arg_constraints = {'low': Real()}
reparametrized_params = ['low']
pytree_data_fields = ('base_dist', 'low', '_support')

property support
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters
- key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.
- sample_shape (tuple) – the sample shape for the distribution.

Returns
an array of shape sample_shape + batch_shape + event_shape

Return type
numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters
value – A batch of samples from the distribution.

Returns
an array with shape value.shape[:-self.event_shape]

Return type
numpy.ndarray

property mean
Mean of the distribution.

property var

3.6.2 RightTruncatedDistribution

class RightTruncatedDistribution(base_dist, high=0.0, *, validate_args=None)

Bases: Distribution

arg_constraints = {'high': Real(),}

reparametrized_params = ['high']


pytree_data_fields = ('base_dist', 'high', '_support')

property support

sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.
Parameters

- **key** (`jax.random.PRNGKey`) – the rng_key key to be used for the distribution.
- **sample_shape** (`tuple`) – the sample shape for the distribution.

Returns

an array of shape `sample_shape + batch_shape + event_shape`

Return type

`numpy.ndarray`

`log_prob(*args, **kwargs)`

Evaluates the log probability density for a batch of samples given by `value`.

Parameters

- **value** – A batch of samples from the distribution.

Returns

an array with shape `value.shape[:-self.event_shape]`

Return type

`numpy.ndarray`

property mean

Mean of the distribution.

property var

3.6.3 TruncatedCauchy

```python
class TruncatedCauchy(loc=0.0, scale=1.0, *, low=None, high=None, validate_args=None)
```

Bases:

3.6.4 TruncatedDistribution

```python
TruncatedDistribution(base_dist, low=None, high=None, *, validate_args=None)
```

A function to generate a truncated distribution.

Parameters

- **base_dist** – The base distribution to be truncated. This should be a univariate distribution. Currently, only the following distributions are supported: Cauchy, Laplace, Logistic, Normal, and StudentT.
- **low** – the value which is used to truncate the base distribution from below. Setting this parameter to `None` to not truncate from below.
- **high** – the value which is used to truncate the base distribution from above. Setting this parameter to `None` to not truncate from above.
3.6.5 TruncatedNormal

class TruncatedNormal(loc=0.0, scale=1.0, *, low=None, high=None, validate_args=None)

Bases:

3.6.6 TruncatedPolyaGamma

class TruncatedPolyaGamma(batch_shape=(), *, validate_args=None)

Bases: Distribution

documentation

truncation_point = 2.5
num_log_prob_terms = 7
num_gamma_variates = 8
arg_constraints = {}
support = Interval(lower_bound=0.0, upper_bound=2.5)
sample(key, sample_shape=())

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

Parameters

• key (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

• sample_shape (tuple) – the sample shape for the distribution.

Returns

an array of shape sample_shape + batch_shape + event_shape

Return type

numpy.ndarray

log_prob(*args, **kwargs)

Evaluates the log probability density for a batch of samples given by value.

Parameters

value – A batch of samples from the distribution.

Returns

an array with shape value.shape[:-self.event_shape]

Return type

numpy.ndarray
### 3.6.7 TwoSidedTruncatedDistribution

class TwoSidedTruncatedDistribution(base_dist, low=0.0, high=1.0, *, validate_args=None)

**Bases:** Distribution

**arg_constraints** = {'high': Dependent(), 'low': Dependent()}

**reparametrized_params** = ['low', 'high']


**pytree_data_fields** = ('base_dist', 'low', 'high', '_support')

**property support**

**sample(key, sample_shape=())**

Returns a sample from the distribution having shape given by sample_shape + batch_shape + event_shape. Note that when sample_shape is non-empty, leading dimensions (of size sample_shape) of the returned sample will be filled with iid draws from the distribution instance.

**Parameters**

- **key** (jax.random.PRNGKey) – the rng_key key to be used for the distribution.

**Returns**

An array of shape sample_shape + batch_shape + event_shape

**Return type**

numpy.ndarray

**log_prob(*args, **kwargs)**

Evaluates the log probability density for a batch of samples given by value.

**Parameters**

- **value** – A batch of samples from the distribution.

**Returns**

An array with shape value.shape[:-self.event_shape]

**Return type**

numpy.ndarray

**property mean**

Mean of the distribution.

**property var**
3.7 TensorFlow Distributions

Thin wrappers around TensorFlow Probability (TFP) distributions. For details on the TFP distribution interface, see its Distribution docs.

3.7.1 BijectorConstraint

class BijectorConstraint(bijector)

A constraint which is codomain of a TensorFlow bijector.

Parameters

bijector (Bijector) – a TensorFlow bijector

3.7.2 BijectorTransform

class BijectorTransform(bijector)

A wrapper for TensorFlow bijectors to make them compatible with NumPyro's transforms.

Parameters

bijector (Bijector) – a TensorFlow bijector

3.7.3 TFPDistribution

class TFPDistribution(batch_shape=(), event_shape=(), *, validate_args=None)

A thin wrapper for TensorFlow Probability (TFP) distributions. The constructor has the same signature as the corresponding TFP distribution.

This class can be used to convert a TFP distribution to a NumPyro-compatible one as follows:

\[
d = \text{TFPDistribution}[\text{tfd.Normal}(0, 1)]
\]

Note that typical use cases do not require explicitly invoking this wrapper, since NumPyro wraps TFP distributions automatically under the hood in model code, e.g.:

```python
from tensorflow_probability.substrates.jax import distributions as tfd

def model():
    numpyro.sample("x", tfd.Normal(0, 1))
```

3.8 Constraints

3.8.1 Constraint

class Constraint

Bases: object

Abstract base class for constraints.

A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.
is_discrete = False

event_dim = 0

cHECK(value)
    Returns a byte tensor of sample_shape + batch_shape indicating whether each event in value satisfies this constraint.

feasible_like(prototype)
    Get a feasible value which has the same shape as dtype as prototype.

classmethod tree_unflatten(aux_data, params)

3.8.2 boolean

boolean = Boolean()

3.8.3 circular

circular = Circular(lower_bound=-3.141592653589793, upper_bound=3.141592653589793)

3.8.4 corr_cholesky

corr_cholesky = CorrCholesky()

3.8.5 corr_matrix

corr_matrix = CorrMatrix()

3.8.6 dependent

dependent = Dependent()
    Placeholder for variables whose support depends on other variables. These variables obey no simple coordinate-wise constraints.

Parameters

- is_discrete (bool) – Optional value of .is_discrete in case this can be computed statically. If not provided, access to the .is_discrete attribute will raise a NotImplementedError.

- event_dim (int) – Optional value of .event_dim in case this can be computed statically. If not provided, access to the .event_dim attribute will raise a NotImplementedError.
3.8.7 greater_than

greater_than(lower_bound)
    Abstract base class for constraints.
    A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

3.8.8 integer_interval

integer_interval(lower_bound, upper_bound)
    Abstract base class for constraints.
    A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

3.8.9 integer_greater_than

integer_greater_than(lower_bound)
    Abstract base class for constraints.
    A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

3.8.10 interval

interval(lower_bound, upper_bound)
    Abstract base class for constraints.
    A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

3.8.11 l1_ball

l1_ball(x)
    Constrain to the L1 ball of any dimension.

3.8.12 less_than

less_than(upper_bound)
    Abstract base class for constraints.
    A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.
3.8.13 lower_cholesky

lower_cholesky = LowerCholesky()

3.8.14 multinomial

multinomial(upper_bound)

Abstract base class for constraints.

A constraint object represents a region over which a variable is valid, e.g. within which a variable can be opti-
mized.

3.8.15 nonnegative_integer

nonnegative_integer = IntegerNonnegative(lower_bound=0)

3.8.16 ordered_vector

ordered_vector = OrderedVector()

3.8.17 positive

positive = Positive(lower_bound=0.0)

3.8.18 positive_definite

positive_definite = PositiveDefinite()

3.8.19 positive_integer

positive_integer = IntegerPositive(lower_bound=1)

3.8.20 positive_ordered_vector

positive_ordered_vector = PositiveOrderedVector()

Constrains to a positive real-valued tensor where the elements are monotonically increasing along the event_shape dimension.
3.8.21 real

real = Real()

3.8.22 real_vector

real_vector = RealVector(Real(), 1)

3.8.23 scaled_unit_lower_cholesky

scaled_unit_lower_cholesky = ScaledUnitLowerCholesky()

3.8.24 softplus_positive

softplus_positive = SoftplusPositive(lower_bound=0.0)

3.8.25 softplus_lower_cholesky

softplus_lower_cholesky = SoftplusLowerCholesky()

3.8.26 simplex

simplex = Simplex()

3.8.27 sphere

sphere = Sphere()

Constrain to the Euclidean sphere of any dimension.

3.8.28 unit_interval

unit_interval = UnitInterval(lower_bound=0.0, upper_bound=1.0)

3.8.29 zero_sum

zero_sum = <class 'numpyro.distributions.constraints._ZeroSum'>

Abstract base class for constraints.

A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.
3.9 Transforms

3.9.1 biject_to

\textbf{biject_to}(\textit{constraint})

3.9.2 Transform

class Transform
    Bases: object
    \begin{align*}
    \text{domain} &= \text{Real}() \\
    \text{codomain} &= \text{Real}()
    \end{align*}

property \text{inv}

\text{log_abs_det_jacobian}(x, y, \text{intermediates}=\text{None})

\text{call_with_intermediates}(x)

\text{forward_shape}(\text{shape})
    \text{Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.}

\text{inverse_shape}(\text{shape})
    \text{Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.}

\text{classmethod} \text{tree_unflatten}(\text{aux_data, params})

3.9.3 AbsTransform

class AbsTransform
    Bases: ParameterFreeTransform
    \begin{align*}
    \text{domain} &= \text{Real}() \\
    \text{codomain} &= \text{Positive}(\text{lower_bound}=0.0)
    \end{align*}

3.9.4 AffineTransform

class AffineTransform(\text{loc, scale, domain=Real}())
    Bases: Transform

\text{property} \text{codomain}

\text{log_abs_det_jacobian}(x, y, \text{intermediates}=\text{None})

\textbf{Note}

When \textit{scale} is a JAX tracer, we always assume that \textit{scale} > 0 when calculating \textit{codomain}.
NumPyro Documentation

forward_shape(shape)
Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.

inverse_shape(shape)
Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

tree_flatten()

3.9.5 CholeskyTransform
class CholeskyTransform
Bases: ParameterFreeTransform
Transform via the mapping \( y = \text{cholesky}(x) \), where \( x \) is a positive definite matrix.
domain = PositiveDefinite()
codomain = LowerCholesky()
log_abs_det_jacobian(x, y, intermediates=None)

3.9.6 ComposeTransform
class ComposeTransform(parts)
Bases: Transform
property domain
property codomain
log_abs_det_jacobian(x, y, intermediates=None)
call_with_intermediates(x)
forward_shape(shape)
Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.

inverse_shape(shape)
Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

tree_flatten()

3.9.7 CorrCholeskyTransform
class CorrCholeskyTransform
Bases: ParameterFreeTransform
Transforms a unconstrained real vector \( x \) with length \( D \times (D - 1)/2 \) into the Cholesky factor of a D-dimension correlation matrix. This Cholesky factor is a lower triangular matrix with positive diagonals and unit Euclidean norm for each row. The transform is processed as follows:
1. First we convert $x$ into a lower triangular matrix with the following order:

$$
\begin{bmatrix}
1 & 0 & 0 & 0 \\
x_0 & 1 & 0 & 0 \\
x_1 & x_2 & 1 & 0 \\
x_3 & x_4 & x_5 & 1
\end{bmatrix}
$$

2. For each row $X_i$ of the lower triangular part, we apply a signed version of class \texttt{StickBreakingTransform} to transform $X_i$ into a unit Euclidean length vector using the following steps:
   
   a. Scales into the interval $(-1, 1)$ domain: $r_i = \tanh(X_i)$.
   
   b. Transforms into an unsigned domain: $z_i = r_i^2$.
   
   c. Applies $s_i = \text{StickBreakingTransform}(z_i)$.
   
   d. Transforms back into signed domain: $y_i = (\text{sign}(r_i), 1) * \sqrt{s_i}$.

   ```python
   domain = RealVector(Real(), 1)
codomain = CorrCholesky()
   log_abs_det_jacobian(x, y, intermediates=None)
   forward_shape(shape)
   Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.
   inverse_shape(shape)
   Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.
   ```

### 3.9.8 CorrMatrixCholeskyTransform

```python
class CorrMatrixCholeskyTransform
Bases: CholeskyTransform
Transform via the mapping $y = \text{cholesky}(x)$, where $x$ is a correlation matrix.
domain = CorrMatrix()
codomain = CorrCholesky()
log_abs_det_jacobian(x, y, intermediates=None)
```

### 3.9.9 ExpTransform

```python
class ExpTransform(domain=Real())
Bases: Transform
property codomain
log_abs_det_jacobian(x, y, intermediates=None)
tree_flatten()```
3.9.10 IdentityTransform

class IdentityTransform
   Bases: ParameterFreeTransform
   log_abs_det_jacobian(x, y, intermediates=None)

3.9.11 L1BallTransform

class L1BallTransform
   Bases: ParameterFreeTransform
   Transforms a unconstrained real vector $x$ into the unit L1 ball.
   domain = RealVector(Real(), 1)
   codomain = L1Ball()
   log_abs_det_jacobian(x, y, intermediates=None)

3.9.12 LowerCholeskyAffine

class LowerCholeskyAffine(loc, scale_tril)
   Bases: Transform
   Transform via the mapping $y = loc + scale\_tril \odot x$.
   Parameters
      * loc – a real vector.
      * scale_tril – a lower triangular matrix with positive diagonal.

   Example

>>> import jax.numpy as jnp
>>> from numpyro.distributions.transforms import LowerCholeskyAffine
>>> base = jnp.ones(2)
>>> loc = jnp.zeros(2)
>>> scale_tril = jnp.array([[0.3, 0.0], [1.0, 0.5]])
>>> affine = LowerCholeskyAffine(loc=loc, scale_tril=scale_tril)
>>> affine(base)
Array([0.3, 1.5], dtype=float32)

domain = RealVector(Real(), 1)
codomain = RealVector(Real(), 1)
log_abs_det_jacobian(x, y, intermediates=None)
forward_shape(shape)
   Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.
inverse_shape(shape)
   Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.
tree_flatten()
3.9.13 LowerCholeskyTransform

class LowerCholeskyTransform
    Bases: ParameterFreeTransform
    Transform a real vector to a lower triangular cholesky factor, where the strictly lower triangular submatrix is unconstrained and the diagonal is parameterized with an exponential transform.
    
    domain = RealVector(Real(), 1)
    codomain = LowerCholesky()
    log_abs_det_jacobian

    forward_shape(shape)
        Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.
    inverse_shape(shape)
        Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

3.9.14 OrderedTransform

class OrderedTransform
    Bases: ParameterFreeTransform
    Transform a real vector to an ordered vector.
    
    References:
    1. Stan Reference Manual v2.20, section 10.6, Stan Development Team
    
    Example
    
    ```python
    >>> import jax.numpy as jnp
    >>> from numpyro.distributions.transforms import OrderedTransform
    >>> base = jnp.ones(3)
    >>> transform = OrderedTransform()
    >>> assert jnp.allclose(transform(base), jnp.array([1., 3.7182817, 6.4365635]), rtol=1e-3, atol=1e-3)
    
    domain = RealVector(Real(), 1)
    codomain = OrderedVector()
    log_abs_det_jacobian(x, y, intermediates=None)
    
    3.9.15 PermuteTransform

class PermuteTransform(permutation)
    Bases: Transform
    
    domain = RealVector(Real(), 1)
    codomain = RealVector(Real(), 1)
3.9.16 PowerTransform

class PowerTransform(exponent)
   Bases: Transform
   domain = Positive(lower_bound=0.0)
   codomain = Positive(lower_bound=0.0)
   log_abs_det_jacobian(x, y, intermediates=None)
   forward_shape(shape)
      Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.
   inverse_shape(shape)
      Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.
   tree_flatten()
3.9.18 RecursiveLinearTransform

```python
from jax import random
from jax import numpy as jnp
import numpyro
from numpyro import distributions as dist

def cauchy_random_walk():
    return numpyro.sample(
        "x",
        dist.TransformedDistribution(
            dist.Cauchy(0, 1).expand([10, 1]).to_event(1),
            dist.transforms.RecursiveLinearTransform(jnp.eye(1)),
        )
    )

def rocket_trajectory():
    scale = numpyro.sample(
        "scale",
        dist.HalfCauchy(1).expand([2]).to_event(1),
    )
    transition_matrix = jnp.array([[1, 1], [0, 1]])
    return numpyro.sample(
        "x",
        dist.TransformedDistribution(
            dist.Normal(0, scale).expand([10, 2]).to_event(1),
            dist.transforms.RecursiveLinearTransform(transition_matrix),
        )
    )
```

domain = RealMatrix(Real(), 2)
codomain = RealMatrix(Real(), 2)

log_abs_det_jacobian(x: Array, y: Array, intermediates=None)
tree_flatten()
3.9.19 ScaledUnitLowerCholeskyTransform

class ScaledUnitLowerCholeskyTransform
  Bases: LowerCholeskyTransform

  Like LowerCholeskyTransform this Transform transforms a real vector to a lower triangular cholesky factor. However it does so via a decomposition

  \[ y = \text{loc} + \text{unit} \cdot \text{scale}_{\text{tril}} \otimes \text{scale}_{\text{diag}} \otimes x. \]

  where unit_{scale}_{tril} has ones along the diagonal and scale_{diag} is a diagonal matrix with all positive entries that is parameterized with a softplus transform.

  domain = RealVector(Real(), 1)
  codomain = ScaledUnitLowerCholesky()
  log_abs_det_jacobian(x, y, intermediates=None)

3.9.20 SigmoidTransform

class SigmoidTransform
  Bases: ParameterFreeTransform

  codomain = UnitInterval(lower_bound=0.0, upper_bound=1.0)
  log_abs_det_jacobian(x, y, intermediates=None)

3.9.21 SimplexToOrderedTransform

class SimplexToOrderedTransform(anchor_point=0.0)
  Bases: Transform

  Transform a simplex into an ordered vector (via difference in Logistic CDF between cutpoints) Used in [1] to induce a prior on latent cutpoints via transforming ordered category probabilities.

  Parameters

  anchor_point – Anchor point is a nuisance parameter to improve the identifiability of the transform. For simplicity, we assume it is a scalar value, but it is broadcastable x.shape[:-1]. For more details please refer to Section 2.2 in [1]

  References:


  Example

  >>> import jax.numpy as jnp
  >>> from numpyro.distributions.transforms import SimplexToOrderedTransform
  >>> base = jnp.array([0.3, 0.1, 0.4, 0.2])
  >>> transform = SimplexToOrderedTransform()
  >>> assert jnp.allclose(transform(base), jnp.array([-0.8472978, -0.40546507, 1.3862944]), rtol=1e-3, atol=1e-3)

  domain = Simplex()
codomain = OrderedVector()

log_abs_det_jacobian(x, y, intermediates=None)

tree_flatten()

forward_shape(shape)
    Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.

inverse_shape(shape)
    Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

3.9.22 SoftplusLowerCholeskyTransform

class SoftplusLowerCholeskyTransform
    Bases: ParameterFreeTransform

    Transform from unconstrained vector to lower-triangular matrices with nonnegative diagonal entries. This is
    useful for parameterizing positive definite matrices in terms of their Cholesky factorization.

domain = RealVector(Real(), 1)

codomain = SoftplusLowerCholesky()

log_abs_det_jacobian(x, y, intermediates=None)

forward_shape(shape)
    Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.

inverse_shape(shape)
    Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

3.9.23 SoftplusTransform

class SoftplusTransform
    Bases: ParameterFreeTransform

    Transform from unconstrained space to positive domain via softplus \( y = \log(1 + \exp(x)) \). The inverse is
    computed as \( x = \log(\exp(y) - 1) \).

domain = Real()

codomain = SoftplusPositive(lower_bound=0.0)

log_abs_det_jacobian(x, y, intermediates=None)
3.9.24 StickBreakingTransform

class StickBreakingTransform
    Bases: ParameterFreeTransform
domain = RealVector(Real(), 1)
codomain = Simplex()
log_abs_det_jacobian(x, y, intermediates=None)
forward_shape(shape)
    Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.
inverse_shape(shape)
    Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

3.9.25 ZeroSumTransform

class ZeroSumTransform(transform_ndims: int = 1)
    Bases: Transform
    A transform that constrains an array to sum to zero, adapted from PyMC [1] as described in [2,3]

    Parameters
        transform_ndims – Number of trailing dimensions to transform.

    References
        [1] https://github.com/pymc-devs/pymc/blob/244fb97b01ad0f3dadf5c3837b65839e2a59a0e8/pymc/distributions/transforms.py#L266

    property domain: Constraint

    property codomain: Constraint

    extend_axis_rev(array: Array, axis: int) → Array

    extend_axis(array: Array, axis: int) → Array

    log_abs_det_jacobian(x: Array, y: Array, intermediates: None = None) → Array

    forward_shape(shape: tuple) → tuple
    Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.

    inverse_shape(shape: tuple) → tuple
    Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

tree_flatten()
3.10 Flows

3.10.1 InverseAutoregressiveTransform

class InverseAutoregressiveTransform(autoregressive_nn, log_scale_min_clip=-5.0, log_scale_max_clip=3.0)
    Bases: Transform
    An implementation of Inverse Autoregressive Flow, using Eq (10) from Kingma et al., 2016,
    \[ y = \mu_t + \sigma_t \odot x \]
    where \( x \) are the inputs, \( y \) are the outputs, \( \mu_t, \sigma_t \) are calculated from an autoregressive network on \( x \), and \( \sigma_t > 0 \).

References

domain = RealVector(Real(), 1)
codomain = RealVector(Real(), 1)
call_with_intermediates(x)
log_abs_det_jacobian(x, y, intermediates=None)
    Calculates the elementwise determinant of the log jacobian.

Parameters
- \( x \) (numpy.ndarray) – the input to the transform
- \( y \) (numpy.ndarray) – the output of the transform

tree_flatten()

3.10.2 BlockNeuralAutoregressiveTransform

class BlockNeuralAutoregressiveTransform(bn_arn)
    Bases: Transform
    An implementation of Block Neural Autoregressive flow.

References
1. Block Neural Autoregressive Flow, Nicola De Cao, Ivan Titov, Wilker Aziz

domain = RealVector(Real(), 1)
codomain = RealVector(Real(), 1)
call_with_intermediates(x)
log_abs_det_jacobian(x, y, intermediates=None)
    Calculates the elementwise determinant of the log jacobian.

Parameters
- \( x \) (numpy.ndarray) – the input to the transform
- \( y \) (numpy.ndarray) – the output of the transform
tree_flatten()
CHAPTER
FOUR

INFERENCE

4.1 Markov Chain Monte Carlo (MCMC)

We provide a high-level overview of the MCMC algorithms in NumPyro:

- **NUTS**, which is an adaptive variant of **HMC**, is probably the most commonly used MCMC algorithm in NumPyro. Note that NUTS and HMC are not directly applicable to models with discrete latent variables, but in cases where the discrete variables have finite support and summing them out (i.e. enumeration) is tractable, NumPyro will automatically sum out discrete latent variables and perform NUTS/HMC on the remaining continuous latent variables. As discussed above, model reparameterization may be important in some cases to get good performance. Note that, generally speaking, we expect inference to be harder as the dimension of the latent space increases. See the bad geometry tutorial for additional tips and tricks.

- **MixedHMC** can be an effective inference strategy for models that contain both continuous and discrete latent variables.

- **HMCECS** can be an effective inference strategy for models with a large number of data points. It is applicable to models with continuous latent variables. See this example for detailed usage.

- **BarkerMH** is a gradient-based MCMC method that may be competitive with HMC and NUTS for some models. It is applicable to models with continuous latent variables.

- **HMCGibbs** combines HMC/NUTS steps with custom Gibbs updates. Gibbs updates must be specified by the user.

- **DiscreteHMCGibbs** combines HMC/NUTS steps with Gibbs updates for discrete latent variables. The corresponding Gibbs updates are computed automatically.

- **SA** is a gradient-free MCMC method. It is only applicable to models with continuous latent variables. It is expected to perform best for models whose latent dimension is low to moderate. It may be a good choice for models with non-differentiable log densities. Note that SA generally requires a very large number of samples, as mixing tends to be slow. On the plus side individual steps can be fast.

- **AIES** is a gradient-free ensemble MCMC method that informs Metropolis-Hastings proposals by sharing information between chains. It is only applicable to models with continuous latent variables. It is expected to perform best for models whose latent dimension is low to moderate. It may be a good choice for models with non-differentiable log densities, and can be robust to likelihood-free models. AIES generally requires the number of chains to be twice as large as the number of latent parameters, (and ideally larger).

- **ESS** is a gradient-free ensemble MCMC method that shares information between chains to find good slice sampling directions. It tends to be more sample efficient than AIES. It is only applicable to models with continuous latent variables. It is expected to perform best for models whose latent dimension is low to moderate and may be a good choice for models with non-differentiable log densities. ESS generally requires the number of chains to be twice as large as the number of latent parameters, (and ideally larger).
Like HMC/NUTS, all remaining MCMC algorithms support enumeration over discrete latent variables if possible (see restrictions). Enumerated sites need to be marked with `infer={'enumerate': 'parallel'}` like in the annotation example.

```python
class MCMC(sampler, *, num_warmup, num_samples, num_chains=1, thinning=1, postprocess_fn=None,  
chain_method='parallel', progress_bar=True, jit_model_args=False)
```

Bases: object

Provides access to Markov Chain Monte Carlo inference algorithms in NumPyro.

**Note**

`chain_method` is an experimental arg, which might be removed in a future version.

**Note**

Setting `progress_bar=False` will improve the speed for many cases. But it might require more memory than the other option.

**Note**

If setting `num_chains` greater than 1 in a Jupyter Notebook, then you will need to have installed `ipywidgets` in the environment from which you launched Jupyter in order for the progress bars to render correctly. If you are using Jupyter Notebook or Jupyter Lab, please also install the corresponding extension package like `widgetsnbextension` or `jupyterlab_widgets`.

**Note**

If your dataset is large and you have access to multiple acceleration devices, you can distribute the computation across multiple devices. Make sure that your jax version is v0.4.4 or newer. For example,

```python
import jax
from jax.experimental import mesh_utils
from jax.sharding import PositionalSharding
import numpy as np
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

X = np.random.randn(128, 3)
y = np.random.randn(128)

def model(X, y):
    beta = numpyro.sample("beta", dist.Normal(0, 1).expand([3]))
    numpyro.sample("obs", dist.Normal(X @ beta, 1), obs=y)

mcmc = MCMC(NUTS(model), num_warmup=10, num_samples=10)
# See https://jax.readthedocs.io/en/latest/notebooks/Distributed_arrays_and_ ˓→automatic_parallelization.html
sharding = PositionalSharding(mesh_utils.create_device_mesh((8,)))
```
X_shard = jax.device_put(X, sharding.reshape(8, 1))
y_shard = jax.device_put(y, sharding.reshape(8))
mcmc.run(jax.random.PRNGKey(0), X_shard, y_shard)

Parameters

- **sampler** (*MCMCKernel*) – an instance of *MCMCKernel* that determines the sampler for running MCMC. Currently, only *HMC* and *NUTS* are available.
- **num_warmup** (*int*) – Number of warmup steps.
- **num_samples** (*int*) – Number of samples to generate from the Markov chain.
- **thinning** (*int*) – Positive integer that controls the fraction of post-warmup samples that are retained. For example if thinning is 2 then every other sample is retained. Defaults to 1, i.e. no thinning.
- **num_chains** (*int*) – Number of MCMC chains to run. By default, chains will be run in parallel using *jax.pmap()* . If there are not enough devices available, chains will be run in sequence.
- **postprocess_fn** – Post-processing callable - used to convert a collection of unconstrained sample values returned from the sampler to constrained values that lie within the support of the sample sites. Additionally, this is used to return values at deterministic sites in the model.
- **chain_method** (*str*) – A callable jax transform like *jax.vmap* or one of ‘parallel’ (default), ‘sequential’, ‘vectorized’. The method ‘parallel’ is used to execute the drawing process in parallel on XLA devices (CPUs/GPUs/TPUs). If there are not enough devices for ‘parallel’, we fall back to ‘sequential’ method to draw chains sequentially. ‘vectorized’ method is an experimental feature which vectorizes the drawing method, hence allowing us to collect samples in parallel on a single device.
- **progress_bar** (*bool*) – Whether to enable progress bar updates. Defaults to True.
- **jit_model_args** (*bool*) – If set to True, this will compile the potential energy computation as a function of model arguments. As such, calling *MCMC.run* again on a same sized but different dataset will not result in additional compilation cost. Note that currently, this does not take effect for the case num_chains > 1 and chain_method == 'parallel'.

Note

It is possible to mix parallel and vectorized sampling, i.e., run vectorized chains on multiple devices using explicit *pmap*. Currently, doing so requires disabling the progress bar. For example,

```python
def do_mcmc(rng_key, n_vectorized=8):
    nuts_kernel = NUTS(model)
    mcmc = MCMC(
        nuts_kernel,
        progress_bar=False,
        num_chains=n_vectorized,
        chain_method='vectorized'
    )
    mcmc.run(
        rng_key,
        extra_fields=("potential_energy"),
    )
```
```python
return {**mcmc.get_samples(), **mcmc.get_extra_fields()}
```

# Number of devices to pmap over
n_parallel = jax.local_device_count()
rng_keys = jax.random.split(PRNGKey(rng_seed), n_parallel)
traces = pmap(do_mcmc)(rng_keys)
# concatenate traces along pmap ed axis
trace = {k: np.concatenate(v) for k, v in traces.items()}

property post_warmup_state

The state before the sampling phase. If this attribute is not None, `run()` will skip the warmup phase and start with the state specified in this attribute.

Note

This attribute can be used to sequentially draw MCMC samples. For example,

```python
mcmc = MCMC(NUTS(model), num_warmup=100, num_samples=100)
mcmc.run(random.PRNGKey(0))
first_100_samples = mcmc.get_samples()
mcmc.post_warmup_state = mcmc.last_state
mcmc.run(mcmc.post_warmup_state.rng_key)  # or mcmc.run(random.PRNGKey(1))
second_100_samples = mcmc.get_samples()
```

property last_state

The final MCMC state at the end of the sampling phase.

```python
warmup(rng_key, *args, extra_fields=(), collect_warmup=False, init_params=None, **kwargs)
```

Run the MCMC warmup adaptation phase. After this call, `self.post_warmup_state` will be set and the `run()` method will skip the warmup adaptation phase. To run `warmup` again for the new data, it is required to run `warmup()` again.

Parameters

- **rng_key** (`random.PRNGKey`) – Random number generator key to be used for the sampling.
- **args** – Arguments to be provided to the `numpyro.infer.mcmc.MCMCKernel.init()` method. These are typically the arguments needed by the `model`.
- **extra_fields** (tuple or list) – Extra fields (aside from `default_fields()`) from the state object (e.g. `numpyro.infer.hmc.HMCState` for HMC) to collect during the MCMC run. Exclude sample sites from collection with “~`sample_field`.sample_site” e.g. “~z.a” will prevent site “a” from being collected if you’re using the NUTS sampler.
- **collect_warmup** (bool) – Whether to collect samples from the warmup phase. Defaults to `False`.
- **init_params** – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn` provided to the kernel. If the kernel is instantiated by a numpyro model, the initial parameters here correspond to latent values in unconstrained space.
- **kwargs** – Keyword arguments to be provided to the `numpyro.infer.mcmc.MCMCKernel.init()` method. These are typically the keyword arguments needed by the `model`.
run(rng_key, *args, extra_fields=(), init_params=None, **kwargs)

Run the MCMC samplers and collect samples.

Parameters

- **rng_key** (`random.PRNGKey`) – Random number generator key to be used for the sampling. For multi-chains, a batch of num_chains keys can be supplied. If rng_key does not have batch_size, it will be split into a batch of num_chains keys.

- **args** – Arguments to be provided to the `numpyro.infer.mcmc.MCMCKernel.init()` method. These are typically the arguments needed by the `model`.

- **extra_fields** (`tuple` or `list` of `str`) – Extra fields (aside from “z”, “diverging”) from the state object (e.g. `numpyro.infer.hmc.HMCState` for HMC) to be collected during the MCMC run. Note that subfields can be accessed using dots, e.g. “adapt_state.step_size” can be used to collect step sizes at each step. Exclude sample sites from collection with “~sampler.sample_field.sample_site”. e.g. “~z.a” will prevent site “a” from being collected if you’re using the NUTS sampler.

- **init_params** – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn` provided to the kernel. If the kernel is instantiated by a numpyro model, the initial parameters here correspond to latent values in unconstrained space.

- **kwargs** – Keyword arguments to be provided to the `numpyro.infer.mcmc.MCMCKernel.init()` method. These are typically the keyword arguments needed by the `model`.

Note

JAX allows python code to continue even when the compiled code has not finished yet. This can cause troubles when trying to profile the code for speed. See [https://jax.readthedocs.io/en/latest/async_dispatch.html](https://jax.readthedocs.io/en/latest/async_dispatch.html) and [https://jax.readthedocs.io/en/latest/profiling.html](https://jax.readthedocs.io/en/latest/profiling.html) for pointers on profiling JAX programs.

get_samples(group_by_chain=False)

Get samples from the MCMC run.

Parameters

- **group_by_chain** (`bool`) – Whether to preserve the chain dimension. If True, all samples will have num_chains as the size of their leading dimension.

Returns

Samples having the same data type as `init_params`. The data type is a `dict` keyed on site names if a model containing Pyro primitives is used, but can be any `jaxlib.pytree()`, more generally (e.g. when defining a `potential_fn` for HMC that takes list args).

Example:

You can then pass those samples to `Predictive`:

```python
posterior_samples = mcmc.get_samples()
predictive = Predictive(model, posterior_samples=posterior_samples)
samples = predictive(rng_key1, *model_args, **model_kwargs)
```

get_extra_fields(group_by_chain=False)

Get extra fields from the MCMC run.

4.1. Markov Chain Monte Carlo (MCMC) 137
Parameters

**group_by_chain** (bool) – Whether to preserve the chain dimension. If True, all samples will have num_chains as the size of their leading dimension.

Returns

Extra fields keyed by field names which are specified in the *extra_fields* keyword of *run()*.

**print_summary** *(prob=0.9, exclude_deterministic=True)*

Print the statistics of posterior samples collected during running this MCMC instance.

Parameters

- **prob** (float) – the probability mass of samples within the credible interval.
- **exclude_deterministic** (bool) – whether or not print out the statistics at deterministic sites.

**transfer_states_to_host()**

Reduce the memory footprint of collected samples by transferring them to the host device.

### 4.1.1 MCMC Kernels

**MCMCKernel**

class MCMCKernel

Bases: ABC

Defines the interface for the Markov transition kernel that is used for MCMC inference.

Example:

```python
>>> from collections import namedtuple
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC

>>> MHState = namedtuple("MHState", ["u", "rng_key"])

>>> class MetropolisHastings(numpyro.infer.mcmc.MCMCKernel):
...     sample_field = "u"
...     ...
...     def __init__(self, potential_fn, step_size=0.1):
...         self.potential_fn = potential_fn
...         self.step_size = step_size
...     ...
...     def init(self, rng_key, num_warmup, init_params, model_args, model_kwargs):
...         return MHState(init_params, rng_key)
...     ...

...     def sample(self, state, model_args, model_kwargs):
...         u, rng_key = state
...         rng_key, key_proposal, key_accept = random.split(rng_key, 3)
...         u_proposal = dist.Normal(u, self.step_size).sample(key_proposal)
...         accept_prob = jnp.exp(self.potential_fn(u) - self.potential_fn(u_proposal))

(continues on next page)```
... u_new = jnp.where(dist.Uniform().sample(key_accept) < accept_prob, u_proposal, u)
... return MHState(u_new, rng_key)

```python
>>> def f(x):
...    return ((x - 2) ** 2).sum()

>>> kernel = MetropolisHastings(f)
>>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=1000)
>>> mcmc.run(random.PRNGKey(0), init_params=jnp.array([1., 2.]))
>>> posterior_samples = mcmc.get_samples()
>>> mcmc.print_summary()
```

**postprocess_fn**(model_args, model_kwargs)
Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

**Parameters**
- **model_args** – Arguments to the model.
- **model_kwargs** – Keyword arguments to the model.

**abstract init**(rng_key, num_warmup, init_params, model_args, model_kwargs)
Initialize the `MCMCKernel` and return an initial state to begin sampling from.

**Parameters**
- **rng_key** *(random.PRNGKey)* – Random number generator key to initialize the kernel.
- **num_warmup** *(int)* – Number of warmup steps. This can be useful when doing adaptation during warmup.
- **init_params** *(tuple)* – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn`.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

**Returns**
The initial state representing the state of the kernel. This can be any class that is registered as a `pytree`.

**abstract sample**(state, model_args, model_kwargs)
Given the current `state`, return the next `state` using the given transition kernel.

**Parameters**
- **state** – A `pytree` class representing the state for the kernel. For HMC, this is given by `HMCState`. In general, this could be any class that supports `getattr`.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

**Returns**
Next `state`.
property sample_field

The attribute of the state object passed to sample() that denotes the MCMC sample. This is used by postprocess_fn() and for reporting results in MCMC.print_summary().

property default_fields

The attributes of the state object to be collected by default during the MCMC run (when MCMC.run() is called).

property is_ensemble_kernel

Denotes whether the kernel is an ensemble kernel. If True, diagnostics_str will be displayed during the MCMC run (when MCMC.run() is called) if chain_method = “vectorized”.

get_diagnostics_str(state)

Given the current state, returns the diagnostics string to be added to progress bar for diagnostics purpose.

BarkerMH

class BarkerMH(model=None, potential_fn=None, step_size=1.0, adapt_step_size=True, adapt_mass_matrix=True, dense_mass=False, target_accept_prob=0.4, init_strategy=<function init_to_uniform>)

Bases: MCMCKernel

This is a gradient-based MCMC algorithm of Metropolis-Hastings type that uses a skew-symmetric proposal distribution that depends on the gradient of the potential (the Barker proposal; see reference [1]). In particular the proposal distribution is skewed in the direction of the gradient at the current sample.

We expect this algorithm to be particularly effective for low to moderate dimensional models, where it may be competitive with HMC and NUTS.

Note

We recommend to use this kernel with progress_bar=False in MCMC to reduce JAX’s dispatch overhead.

References:


Parameters

- **model** – Python callable containing Pyro primitives. If model is provided, potential_fn will be inferred using the model.
- **potential_fn** – Python callable that computes the potential energy given input parameters. The input parameters to potential_fn can be any python collection type, provided that init_params argument to init() has the same type.
- **step_size** (float) – (Initial) step size to use in the Barker proposal.
- **adapt_step_size** (bool) – Whether to adapt the step size during warm-up. Defaults to adapt_step_size==True.
- **adapt_mass_matrix** (bool) – Whether to adapt the mass matrix during warm-up. Defaults to adapt_mass_matrix==True.
- **dense_mass** (bool) – Whether to use a dense (i.e. full-rank) or diagonal mass matrix. (defaults to dense_mass=False).
• **target_accept_prob** (*float*) – The target acceptance probability that is used to guide step size adaptation. Defaults to target_accept_prob=0.4.

• **init_strategy** (*callable*) – a per-site initialization function. See *Initialization Strategies* section for available functions.

### Example

```python
>>> import jax
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC, BarkerMH

>>> def model():
...     x = numpyro.sample("x", dist.Normal().expand([10]))
...     numpyro.sample("obs", dist.Normal(x, 1.0), obs=jnp.ones(10))

>>> kernel = BarkerMH(model)
>>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=1000, progress_bar=True)
>>> mcmc.run(jax.random.PRNGKey(0))
>>> mcmc.print_summary()
```

### property model

#### property sample_field

The attribute of the *state* object passed to *sample()* that denotes the MCMC sample. This is used by *postprocess_fn()* and for reporting results in *MCMC.print_summary()*.

### get_diagnostics_str(*state*)

Given the current *state*, returns the diagnostics string to be added to progress bar for diagnostics purpose.

### init(*rng_key*, num_warmup, init_params, model_args, model_kwargs)

Initialize the *MCMKKernel* and return an initial state to begin sampling from.

**Parameters**

- **rng_key** (*random.PRNGKey*) – Random number generator key to initialize the kernel.

- **num_warmup** (*int*) – Number of warmup steps. This can be useful when doing adaptation during warmup.

- **init_params** (*tuple*) – Initial parameters to begin sampling. The type must be consistent with the input type to *potential_fn*.

- **model_args** – Arguments provided to the model.

- **model_kwargs** – Keyword arguments provided to the model.

**Returns**

The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

### postprocess_fn(*args, kwargs*)

Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

**Parameters**

- **model_args** – Arguments to the model.
**model_kwargs** – Keyword arguments to the model.

```
sample(state, model_args, model_kwargs)
```

Given the current `state`, return the next `state` using the given transition kernel.

**Parameters**

- **state** – A pytree class representing the state for the kernel. For HMC, this is given by `HMCState`. In general, this could be any class that supports `getattr`.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

**Returns**

Next `state`.

---

**HMC**

```
class HMC(model=None, potential_fn=None, kinetic_fn=None, step_size=1.0, inverse_mass_matrix=None, adapt_step_size=True, adapt_mass_matrix=True, dense_mass=False, target_accept_prob=0.8, num_steps=None, trajectory_length=6.283185307179586, init_strategy=<function init_to_uniform>, find_heuristic_step_size=False, forward_mode_differentiation=False, regularize_mass_matrix=True)
```

Bases: `MCMCKernel`

Hamiltonian Monte Carlo inference, using fixed trajectory length, with provision for step size and mass matrix adaptation.

**Note**

Until the kernel is used in an MCMC run, `postprocess_fn` will return the identity function.

**Note**

The default init strategy `init_to_uniform` might not be a good strategy for some models. You might want to try other init strategies like `init_to_median`.

**References:**

1. *MCMC Using Hamiltonian Dynamics*, Radford M. Neal

**Parameters**

- **model** – Python callable containing Pyro `primitives`. If model is provided, `potential_fn` will be inferred using the model.
- **potential_fn** – Python callable that computes the potential energy given input parameters. The input parameters to `potential_fn` can be any python collection type, provided that `init_params` argument to `initialize` has the same type.
- **kinetic_fn** – Python callable that returns the kinetic energy given inverse mass matrix and momentum. If not provided, the default is euclidean kinetic energy.
- **step_size** (`float`) – Determines the size of a single step taken by the verlet integrator while computing the trajectory using Hamiltonian dynamics. If not specified, it will be set to 1.
• **inverse_mass_matrix** *(numpy.ndarray or dict)* – Initial value for inverse mass matrix. This may be adapted during warmup if `adapt_mass_matrix = True`. If no value is specified, then it is initialized to the identity matrix. For a potential_fn with general JAX pytree parameters, the order of entries of the mass matrix is the order of the flattened version of pytree parameters obtained with `jax.tree_flatten`, which is a bit ambiguous (see more at https://jax.readthedocs.io/en/latest/pytrees.html). If `model` is not None, here we can specify a structured block mass matrix as a dictionary, where keys are tuple of site names and values are the corresponding block of the mass matrix. For more information about structured mass matrix, see **dense_mass** argument.

• **adapt_step_size** *(bool)* – A flag to decide if we want to adapt step_size during warm-up phase using Dual Averaging scheme.

• **adapt_mass_matrix** *(bool)* – A flag to decide if we want to adapt mass matrix during warm-up phase using Welford scheme.

• **dense_mass** *(bool or list)* – This flag controls whether mass matrix is dense (i.e. full-rank) or diagonal (defaults to `dense_mass=False`). To specify a structured mass matrix, users can provide a list of tuples of site names. Each tuple represents a block in the joint mass matrix. For example, assuming that the model has latent variables “x”, “y”, “z” (where each variable can be multi-dimensional), possible specifications and corresponding mass matrix structures are as follows:
  – dense_mass=[(“x”, “y”)]: use a dense mass matrix for the joint (x, y) and a diagonal mass matrix for z
  – dense_mass=[] (equivalent to dense_mass=False): use a diagonal mass matrix for the joint (x, y, z)
  – dense_mass=[(“x”, “y”, “z”)] (equivalent to full_mass=True): use a dense mass matrix for the joint (x, y, z)
  – dense_mass=[(“x’”,), (“y’”), (“z’”): use dense mass matrices for each of x, y, and z (i.e. block-diagonal with 3 blocks)

• **target_accept_prob** *(float)* – Target acceptance probability for step size adaptation using Dual Averaging. Increasing this value will lead to a smaller step size, hence the sampling will be slower but more robust. Defaults to 0.8.

• **num_steps** *(int)* – if different than None, fix the number of steps allowed for each iteration.

• **trajectory_length** *(float)* – Length of a MCMC trajectory for HMC. Default value is \(2\pi\).

• **init_strategy** *(callable)* – a per-site initialization function. See `Initialization Strategies` section for available functions.

• **find_heuristic_step_size** *(bool)* – whether or not to use a heuristic function to adjust the step size at the beginning of each adaptation window. Defaults to False.

• **forward_mode_differentiation** *(bool)* – whether to use forward-mode differentiation or reverse-mode differentiation. By default, we use reverse mode but the forward mode can be useful in some cases to improve the performance. In addition, some control flow utility on JAX such as `jax.lax.while_loop` or `jax.lax.fori_loop` only supports forward-mode differentiation. See JAX’s The Autodiff Cookbook for more information.

• **regularize_mass_matrix** *(bool)* – whether or not to regularize the estimated mass matrix for numerical stability during warmup phase. Defaults to True. This flag does not take effect if `adapt_mass_matrix == False`.
property model

property sample_field
    The attribute of the state object passed to sample() that denotes the MCMC sample. This is used by postprocess_fn() and for reporting results in MCMC.print_summary().

property default_fields
    The attributes of the state object to be collected by default during the MCMC run (when MCMC.run() is called).

get_diagnostics_str(state)
    Given the current state, returns the diagnostics string to be added to progress bar for diagnostics purpose.

init(rng_key, num_warmup, init_params=None, model_args=(), model_kwargs={})
    Initialize the MCMCKernel and return an initial state to begin sampling from.

    Parameters
    • rng_key (random.PRNGKey) – Random number generator key to initialize the kernel.
    • num_warmup (int) – Number of warmup steps. This can be useful when doing adaptation during warmup.
    • init_params (tuple) – Initial parameters to begin sampling. The type must be consistent with the input type to potential_fn.
    • model_args – Arguments provided to the model.
    • model_kwargs – Keyword arguments provided to the model.

    Returns
    The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

postprocess_fn(args, kwargs)
    Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

    Parameters
    • model_args – Arguments to the model.
    • model_kwargs – Keyword arguments to the model.

sample(state, model_args, model_kwargs)
    Run HMC from the given HMCState and return the resulting HMCState.

    Parameters
    • state (HMCState) – Represents the current state.
    • model_args – Arguments provided to the model.
    • model_kwargs – Keyword arguments provided to the model.

    Returns
    Next state after running HMC.
NUTS

```python
class NUTS(model=None, potential_fn=None, kinetic_fn=None, step_size=1.0, inverse_mass_matrix=None, adapt_step_size=True, adapt_mass_matrix=True, dense_mass=False, target_accept_prob=0.8, trajectory_length=None, max_tree_depth=10, init_strategy=<function init_to_uniform>, find_heuristic_step_size=False, forward_mode_differentiation=False, regularize_mass_matrix=True)
```

Bases: `HMC`

Hamiltonian Monte Carlo inference, using the No U-Turn Sampler (NUTS) with adaptive path length and mass matrix adaptation.

**Note**

Until the kernel is used in an MCMC run, `postprocess_fn` will return the identity function.

**Note**

The default init strategy `init_to_uniform` might not be a good strategy for some models. You might want to try other init strategies like `init_to_median`.

**References:**

1. MCMC Using Hamiltonian Dynamics, Radford M. Neal
2. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo, Matthew D. Hoffman, and Andrew Gelman.
3. A Conceptual Introduction to Hamiltonian Monte Carlo, Michael Betancourt

**Parameters**

- `model` – Python callable containing Pyro primitives. If model is provided, `potential_fn` will be inferred using the model.
- `potential_fn` – Python callable that computes the potential energy given input parameters. The input parameters to `potential_fn` can be any python collection type, provided that `init_params` argument to `init_kernel` has the same type.
- `kinetic_fn` – Python callable that returns the kinetic energy given inverse mass matrix and momentum. If not provided, the default is euclidean kinetic energy.
- `step_size` (float) – Determines the size of a single step taken by the verlet integrator while computing the trajectory using Hamiltonian dynamics. If not specified, it will be set to 1.
- `inverse_mass_matrix` (numpy.ndarray or dict) – Initial value for inverse mass matrix. This may be adapted during warmup if `adapt_mass_matrix = True`. If no value is specified, then it is initialized to the identity matrix. For a potential_fn with general JAX pytree parameters, the order of entries of the mass matrix is the order of the flattened version of pytree parameters obtained with `jax.tree_flatten`, which is a bit ambiguous (see more at https://jax.readthedocs.io/en/latest/pytrees.html). If `model` is not None, here we can specify a structured block mass matrix as a dictionary, where keys are tuple of site names and values are the corresponding block of the mass matrix. For more information about structured mass matrix, see `dense_mass` argument.
• **adapt_step_size** (*bool*) – A flag to decide if we want to adapt step_size during warm-up phase using Dual Averaging scheme.

• **adapt_mass_matrix** (*bool*) – A flag to decide if we want to adapt mass matrix during warm-up phase using Welford scheme.

• **dense_mass** (*bool or list*) – This flag controls whether mass matrix is dense (i.e. full-rank) or diagonal (defaults to dense_mass=False). To specify a structured mass matrix, users can provide a list of tuples of site names. Each tuple represents a block in the joint mass matrix. For example, assuming that the model has latent variables “x”, “y”, “z” (where each variable can be multi-dimensional), possible specifications and corresponding mass matrix structures are as follows:
  - dense_mass=[(“x”, “y”)]: use a dense mass matrix for the joint (x, y) and a diagonal mass matrix for z
  - dense_mass=[] (equivalent to dense_mass=False): use a diagonal mass matrix for the joint (x, y, z)
  - dense_mass=[(“x”, “y”, “z”)] (equivalent to full_mass=True): use a dense mass matrix for the joint (x, y, z)
  - dense_mass=[(“x”), (“y”), (“z”)]: use dense mass matrices for each of x, y, and z (i.e. block-diagonal with 3 blocks)

• **target_accept_prob** (*float*) – Target acceptance probability for step size adaptation using Dual Averaging. Increasing this value will lead to a smaller step size, hence the sampling will be slower but more robust. Defaults to 0.8.

• **trajectory_length** (*float*) – Length of a MCMC trajectory for HMC. This arg has no effect in NUTS sampler.

• **max_tree_depth** (*int*) – Max depth of the binary tree created during the doubling scheme of NUTS sampler. Defaults to 10. This argument also accepts a tuple of integers (d1, d2), where d1 is the max tree depth during warmup phase and d2 is the max tree depth during post warmup phase.

• **init_strategy** (*callable*) – A per-site initialization function. See Initialization Strategies section for available functions.

• **find_heuristic_step_size** (*bool*) – whether or not to use a heuristic function to adjust the step size at the beginning of each adaptation window. Defaults to False.

• **forward_mode_differentiation** (*bool*) – whether to use forward-mode differentiation or reverse-mode differentiation. By default, we use reverse mode but the forward mode can be useful in some cases to improve the performance. In addition, some control flow utility on JAX such as jax.lax.while_loop or jax.lax.fori_loop only supports forward-mode differentiation. See JAX’s The Autodiff Cookbook for more information.

**HMCGibbs**

class HMCGibbs(inner_kernel, gibbs_fn, gibbs_sites)

    Bases: MCMCKernel

    [EXPERIMENTAL INTERFACE]

    HMC-within-Gibbs. This inference algorithm allows the user to combine general purpose gradient-based inference (HMC or NUTS) with custom Gibbs samplers.

    Note that it is the user’s responsibility to provide a correct implementation of gibbs_fn that samples from the corresponding posterior conditional.
Parameters

- **inner_kernel** – One of HMC or NUTS.
- **gibbs_fn** – A Python callable that returns a dictionary of Gibbs samples conditioned on the HMC sites. Must include an argument `rng_key` that should be used for all sampling. Must also include arguments `hmc_sites` and `gibbs_sites`, each of which is a dictionary with keys that are site names and values that are sample values. Note that a given `gibbs_fn` may not need make use of all these sample values.
- **gibbs_sites** *(list)* – a list of site names for the latent variables that are covered by the Gibbs sampler.

Example

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC, NUTS, HMCGibbs
...
>>> def model():
...     x = numpyro.sample("x", dist.Normal(0.0, 2.0))
...     y = numpyro.sample("y", dist.Normal(0.0, 2.0))
...     numpyro.sample("obs", dist.Normal(x + y, 1.0), obs=jnp.array([1.0]))
...     return

>>> def gibbs_fn(rng_key, gibbs_sites, hmc_sites):
...     y = hmc_sites['y']
...     new_x = dist.Normal(0.8 * (1-y), jnp.sqrt(0.8)).sample(rng_key)
...     return {'x': new_x}

>>> hmc_kernel = NUTS(model)
>>> kernel = HMCGibbs(hmc_kernel, gibbs_fn=gibbs_fn, gibbs_sites=['x'])
>>> mcmc = MCMC(kernel, num_warmup=100, num_samples=100, progress_bar=False)
>>> mcmc.run(random.PRNGKey(0))
>>> mcmc.print_summary()

sample_field = 'z'

property model

get_diagnostics_str(state)

Given the current state, returns the diagnostics string to be added to progress bar for diagnostics purpose.

postprocess_fn(args, kwargs)

Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

Parameters

- **model_args** – Arguments to the model.
- **model_kwvars** – Keyword arguments to the model.

init(rng_key, num_warmup, init_params, model_args, model_kwvars)

Initialize the MCMCKernel and return an initial state to begin sampling from.

Parameters

- **rng_key** *(random.PRNGKey)* – Random number generator key to initialize the kernel.
• **num_warmup** (*int*) – Number of warmup steps. This can be useful when doing adaptation during warmup.

• **init_params** (*tuple*) – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn`.

• **model_args** – Arguments provided to the model.

• **model_kwargs** – Keyword arguments provided to the model.

**Returns**

The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

```python
sample(state, model_args, model_kwargs)
```

Given the current `state`, return the next `state` using the given transition kernel.

**Parameters**

• **state** – A pytree class representing the state for the kernel. For HMC, this is given by `HMCState`. In general, this could be any class that supports `getattr`.

• **model_args** – Arguments provided to the model.

• **model_kwargs** – Keyword arguments provided to the model.

**Returns**

Next `state`.

**DiscreteHMCGibbs**

```python
class DiscreteHMCGibbs(inner_kernel, *, random_walk=False, modified=False)
```

A subclass of `HMCGibbs` which performs Metropolis updates for discrete latent sites.

**Parameters**

• **inner_kernel** – One of `HMC` or `NUTS`.

• **random_walk** (*bool*) – If False, Gibbs sampling will be used to draw a sample from the conditional `p(gibbs_site | remaining sites)`. Otherwise, a sample will be drawn uniformly from the domain of `gibbs_site`. Defaults to False.

**Note**

The site update order is randomly permuted at each step.

**Note**

This class supports enumeration of discrete latent variables. To marginalize out a discrete latent site, we can specify `infer={'enumerate': 'parallel'}` keyword in its corresponding `sample()` statement.
• **modified** (*bool*) – whether to use a modified proposal, as suggested in reference [1], which always proposes a new state for the current Gibbs site. Defaults to False. The modified scheme appears in the literature under the name “modified Gibbs sampler” or “Metropolised Gibbs sampler”.

References:


Example

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import DiscreteHMCGibbs, MCMC, NUTS
...
>>> def model(probs, locs):
...    c = numpyro.sample("c", dist.Categorical(probs))
...    numpyro.sample("x", dist.Normal(locs[c], 0.5))
...
>>> probs = jnp.array([0.15, 0.3, 0.3, 0.25])
>>> locs = jnp.array([-2, 0, 2, 4])
>>> kernel = DiscreteHMCGibbs(NUTS(model), modified=True)
>>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=100000, progress_bar=False)
>>> mcmc.run(random.PRNGKey(0), probs, locs)
>>> mcmc.print_summary()
>>> samples = mcmc.get_samples()["x"]
>>> assert abs(jnp.mean(samples) - 1.3) < 0.1
>>> assert abs(jnp.var(samples) - 4.36) < 0.5
```

`init(rng_key, num_warmup, init_params, model_args, model_kwargs)`

Initialize the `MCMCKernel` and return an initial state to begin sampling from.

Parameters

- **rng_key** (*random.PRNGKey*) – Random number generator key to initialize the kernel.
- **num_warmup** (*int*) – Number of warmup steps. This can be useful when doing adaptation during warmup.
- **init_params** (*tuple*) – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn`.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

Returns

The initial state representing the state of the kernel. This can be any class that is registered as a `pytree`.

`sample(state, model_args, model_kwargs)`

Given the current `state`, return the next `state` using the given transition kernel.

Parameters

- **state** – A `pytree` class representing the state for the kernel. For HMC, this is given by `HMCState`. In general, this could be any class that supports `getattr`.

4.1. Markov Chain Monte Carlo (MCMC)
• **model_args** – Arguments provided to the model.
• **model_kwargs** – Keyword arguments provided to the model.

**Returns**

Next state.

**MixedHMC**

class MixedHMC(inner_kernel, *, num_discrete_updates=None, random_walk=False, modified=False)

Bases: DiscreteHMCGibbs

Implementation of Mixed Hamiltonian Monte Carlo (reference [1]).

**Note**

The number of discrete sites to update at each MCMC iteration (\(n_D\) in reference [1]) is fixed at value 1.

**References**


**Parameters**

• **inner_kernel** – A HMC kernel.
• **num_discrete_updates** *(int)* – Number of times to update discrete variables. Defaults to the number of discrete latent variables.
• **random_walk** *(bool)* – If False, Gibbs sampling will be used to draw a sample from the conditional \(p(\text{gibbs\_site} | \text{remaining sites})\), where \(\text{gibbs\_site}\) is one of the discrete sample sites in the model. Otherwise, a sample will be drawn uniformly from the domain of \(\text{gibbs\_site}\). Defaults to False.
• **modified** *(bool)* – whether to use a modified proposal, as suggested in reference [2], which always proposes a new state for the current Gibbs site (i.e. discrete site). Defaults to False. The modified scheme appears in the literature under the name “modified Gibbs sampler” or “Metropolised Gibbs sampler”.

**Example**

```python
groove from jax import random
groove import jax.numpy as jnp
groove import numpyro
groove import numpyro.distributions as dist
groove from numpyro.infer import HMC, MCMC, MixedHMC
groove...
groove def model(probs, locs):
groove ... c = numpyro.sample("c", dist.Categorical(probs))
groove ... x = numpyro.sample("x", dist.Normal(locs[c], 0.5))
groove ...
groove probs = jnp.array([0.15, 0.3, 0.3, 0.25])
groove locs = jnp.array([-2, 0, 2, 4])
groove kernel = MixedHMC(HMC(model, trajectory_length=1.2), num_discrete_updates=20)
```
(continues on next page)
init(rng_key, num_warmup, init_params, model_args, model_kwargs)

Initialize the MCMCKernel and return an initial state to begin sampling from.

Parameters

- **rng_key** (*random.PRNGKey*) – Random number generator key to initialize the kernel.
- **num_warmup** (*int*) – Number of warmup steps. This can be useful when doing adaptation during warmup.
- **init_params** (*tuple*) – Initial parameters to begin sampling. The type must be consistent with the input type to potential_fn.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

Returns

The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

sample(state, model_args, model_kwargs)

Given the current state, return the next state using the given transition kernel.

Parameters

- **state** – A pytree class representing the state for the kernel. For HMC, this is given by HMCState. In general, this could be any class that supports getattr.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

Returns

Next state.
New subsample indices are proposed randomly with replacement at each MCMC step.

References:


Parameters

- `inner_kernel` – One of *HMC* or *NUTS*.
- `num_blocks` *(int)* – Number of blocks to partition subsample into.
- `proxy` – Either `taylor_proxy()` for likelihood estimation, or, None for naive (in-between trajectory) subsampling as outlined in [4].

Example

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import HMCECS, MCMC, NUTS
...
>>> def model(data):
...     x = numpyro.sample("x", dist.Normal(0, 1))
...     with numpyro.plate("N", data.shape[0], subsample_size=100):
...         batch = numpyro.subsample(data, event_dim=0)
...         numpyro.sample("obs", dist.Normal(x, 1), obs=batch)
...
>>> data = random.normal(random.PRNGKey(0), (10000,)) + 1
>>> kernel = HMCECS(NUTS(model), num_blocks=10)
>>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=1000)
>>> mcmc.run(random.PRNGKey(0), data)
>>> samples = mcmc.get_samples()["x"]
>>> assert abs(jnp.mean(samples) - 1.) < 0.1
```

`postprocess_fn(args, kwargs)`

Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

Parameters

- `model_args` – Arguments to the model.
- `model_kwargs` – Keyword arguments to the model.
init(rng_key, num_warmup, init_params, model_args, model_kwargs)

Initialize the MCMCKernel and return an initial state to begin sampling from.

**Parameters**

- **rng_key** (*random.PRNGKey*) – Random number generator key to initialize the kernel.
- **num_warmup** (*int*) – Number of warmup steps. This can be useful when doing adaptation during warmup.
- **init_params** (*tuple*) – Initial parameters to begin sampling. The type must be consistent with the input type to potential_fn.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

**Returns**

The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

sample(state, model_args, model_kwargs)

Given the current state, return the next state using the given transition kernel.

**Parameters**

- **state** – A pytree class representing the state for the kernel. For HMC, this is given by HMCState. In general, this could be any class that supports getattr.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

**Returns**

Next state.

taylor_proxy(reference_params, degree=2)

This is just a convenient static method which calls taylor_proxy().

### SA

class SA(model=None, potential_fn=None, adapt_state_size=None, dense_mass=True, init_strategy=<function init_to_uniform>)

Bases: MCMCKernel

Sample Adaptive MCMC, a gradient-free sampler.

This is a very fast (in term of n_eff / s) sampler but requires many warmup (burn-in) steps. In each MCMC step, we only need to evaluate potential function at one point.

Note that unlike in reference [1], we return a randomly selected (i.e. thinned) subset of approximate posterior samples of size num_chains x num_samples instead of num_chains x num_samples x adapt_state_size.

**Note**

We recommend to use this kernel with progress_bar=False in MCMC to reduce JAX’s dispatch overhead.

**References:**

Parameters

- **model** – Python callable containing Pyro primitives. If model is provided, `potential_fn` will be inferred using the model.

- **potential_fn** – Python callable that computes the potential energy given input parameters. The input parameters to `potential_fn` can be any python collection type, provided that `init_params` argument to `init()` has the same type.

- **adapt_state_size (int)** – The number of points to generate proposal distribution. Defaults to 2 times latent size.

- **dense_mass (bool)** – A flag to decide if mass matrix is dense or diagonal (default to `dense_mass=True`)

- **init_strategy (callable)** – a per-site initialization function. See Initialization Strategies section for available functions.

```python
init(rng_key, num_warmup, init_params=None, model_args=(), model_kwargs={})
```

Initialize the MCMCKernel and return an initial state to begin sampling from.

Parameters

- **rng_key** (`random.PRNGKey`) – Random number generator key to initialize the kernel.

- **num_warmup** (`int`) – Number of warmup steps. This can be useful when doing adaptation during warmup.

- **init_params** (`tuple`) – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn`.

- **model_args** – Arguments provided to the model.

- **model_kwargs** – Keyword arguments provided to the model.

Returns

The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

**property model**

**property sample_field**

The attribute of the state object passed to `sample()` that denotes the MCMC sample. This is used by `postprocess_fn()` and for reporting results in `MCMC.print_summary()`.

**property default_fields**

The attributes of the state object to be collected by default during the MCMC run (when `MCMC.run()` is called).

**get_diagnostics_str(state)**

Given the current state, returns the diagnostics string to be added to progress bar for diagnostics purpose.

**postprocess_fn(args, kwargs)**

Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

Parameters

- **model_args** – Arguments to the model.

- **model_kwargs** – Keyword arguments to the model.
sample(state, model_args, model_kwarg)

Run SA from the given SASate and return the resulting SASate.

Parameters

- state (SASate) – Represents the current state.
- model_args – Arguments provided to the model.
- model_kwarg – Keyword arguments provided to the model.

Returns

Next state after running SA.

EnsembleSampler

class EnsembleSampler(model=None, potential_fn=None, *, randomize_split, init_strategy)

Bases: MCMCKernel, ABC

Abstract class for ensemble samplers. Each MCMC sample is divided into two sub-iterations in which half of the ensemble is updated.

Parameters

- model – Python callable containing Pyro primitives. If model is provided, potential_fn will be inferred using the model.
- potential_fn – Python callable that computes the potential energy given input parameters. The input parameters to potential_fn can be any python collection type, provided that init_params argument to init() has the same type.
- randomize_split (bool) – whether or not to permute the chain order at each iteration.
- init_strategy (callable) – a per-site initialization function. See Initialization Strategies section for available functions.

property model

property sample_field

The attribute of the state object passed to sample() that denotes the MCMC sample. This is used by postprocess_fn() and for reporting results in MCMC.print_summary().

property is_ensemble_kernel

Denotes whether the kernel is an ensemble kernel. If True, diagnostics_str will be displayed during the MCMC run (when MCMC.run() is called) if chain_method = “vectorized”.

abstract init_inner_state(rng_key)

return inner_state

abstract update_active_chains(active, inactive, inner_state)

return (updated active set of chains, updated inner state)

init(rng_key, num_warmup, init_params=None, model_args=(), model_kwarg=())

Initialize the MCMCKernel and return an initial state to begin sampling from.

Parameters

- rng_key (random.PRNGKey) – Random number generator key to initialize the kernel.
- num_warmup (int) – Number of warmup steps. This can be useful when doing adaptation during warmup.
• **init_params** *(tuple)* – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn`.

• **model_args** – Arguments provided to the model.

• **model_kwargs** – Keyword arguments provided to the model.

**Returns**

The initial state representing the state of the kernel. This can be any class that is registered as a pytree.

**postprocess_fn** *(args, kwargs)*

Get a function that transforms unconstrained values at sample sites to values constrained to the site’s support, in addition to returning deterministic sites in the model.

**Parameters**

• **model_args** – Arguments to the model.

• **model_kwargs** – Keyword arguments to the model.

**sample** *(state, model_args, model_kwargs)*

Given the current `state`, return the next `state` using the given transition kernel.

**Parameters**

• **state** – A pytree class representing the state for the kernel. For HMC, this is given by `HMCState`. In general, this could be any class that supports `getattr`.

• **model_args** – Arguments provided to the model.

• **model_kwargs** – Keyword arguments provided to the model.

**Returns**

Next `state`.

**AIES**

class **AIES** *(model=None, potential_fn=None, randomize_split=False, moves=None, init_strategy=<function init_to_uniform>)*

**Bases:** `EnsembleSampler`

Affine-Invariant Ensemble Sampling: a gradient free method that informs Metropolis-Hastings proposals by sharing information between chains. Suitable for low to moderate dimensional models. Generally, `num_chains` should be at least twice the dimensionality of the model.

**Note**

This kernel must be used with `num_chains > 1` and `chain_method="vectorized` in MCMC. The number of chains must be divisible by 2.

**References:**

   Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman.

**Parameters**
• **model** – Python callable containing Pyro primitives. If model is provided, potential_fn will be inferred using the model.

• **potential_fn** – Python callable that computes the potential energy given input parameters. The input parameters to potential_fn can be any python collection type, provided that init_params argument to init() has the same type.

• **randomize_split** (bool) – whether or not to permute the chain order at each iteration. Defaults to False.

• **moves** – a dictionary mapping moves to their respective probabilities of being selected. Valid keys are AIES.DEMove() and AIES.StretchMove(). Both tend to work well in practice. If the sum of probabilities exceeds 1, the probabilities will be normalized. Defaults to {AIES.DEMove(): 1.0}.

• **init_strategy**(callable) – a per-site initialization function. See Initialization Strategies section for available functions.

Example

```python
>>> import jax
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC, AIES

>>> def model():
...     x = numpyro.sample("x", dist.Normal().expand([10]))
...     numpyro.sample("obs", dist.Normal(x, 1.0), obs=jnp.ones(10))

>>> kernel = AIES(model, moves={AIES.DEMove(): 0.5,
...                              AIES.StretchMove(): 0.5})
>>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=2000, num_chains=20, chain_method='vectorized')
>>> mcmc.run(jax.random.PRNGKey(0))
```

get_diagnostics_str(state)

Given the current state, returns the diagnostics string to be added to progress bar for diagnostics purpose.

init_inner_state(rng_key)

return inner_state

update_active_chains(active, inactive, inner_state)

return (updated active set of chains, updated inner state)

static DEMove(sigma=1e-05, g0=None)

A proposal using differential evolution.

This Differential evolution proposal is implemented following Nelson et al. (2013).

Parameters

• **sigma** – (optional) The standard deviation of the Gaussian used to stretch the proposal vector. Defaults to 1.0.e-5.

• **(optional) (g0)** – The mean stretch factor for the proposal vector. By default, it is \(2.38 \sqrt{2^{ndim}}\) as recommended by the two references.
static StretchMove(a=2.0)


Parameters

a – (optional) The stretch scale parameter. (default: 2.0)

ESS
class ESS(model=None, potential_fn=None, randomize_split=True, moves=None, max_steps=10000,
max_iter=10000, init_mu=1.0, tune_mu=True, init_strategy=<function init_to_uniform>)

Bases: EnsembleSampler

Ensemble Slice Sampling: a gradient free method that finds better slice sampling directions by sharing information between chains. Suitable for low to moderate dimensional models. Generally, num_chains should be at least twice the dimensionality of the model.

Note

This kernel must be used with num_chains > 1 and chain_method="vectorized" in MCMC. The number of chains must be divisible by 2.

References:


Parameters

- model – Python callable containing Pyro primitives. If model is provided, potential_fn will be inferred using the model.
- potential_fn – Python callable that computes the potential energy given input parameters. The input parameters to potential_fn can be any python collection type, provided that init_params argument to init() has the same type.
- randomize_split (bool) – whether or not to permute the chain order at each iteration. Defaults to True.
- moves – a dictionary mapping moves to their respective probabilities of being selected. If the sum of probabilities exceeds 1, the probabilities will be normalized. Valid keys include: ESS.DifferentialMove() -> default proposal, works well along a wide range of target distributions, ESS.GaussianMove() -> for approximately normally distributed targets, ESS.KDEMove() -> for multimodal posteriors - requires large num_chains, and they must be well initialized ESS.RandomMove() -> no chain interaction, useful for debugging. Defaults to {ESS.DifferentialMove(): 1.0}.
- max_steps (int) – number of maximum stepping-out steps per sample. Defaults to 10,000.
- max_iter (int) – number of maximum expansions/contractions per sample. Defaults to 10,000.
- init_mu (float) – initial scale factor. Defaults to 1.0.
- `tune_mu (bool)` – whether or not to tune the initial scale factor. Defaults to True.
- `init_strategy (callable)` – a per-site initialization function. See *Initialization Strategies* section for available functions.

Example

```python
>>> import jax
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC, ESS

>>> def model():
...     x = numpyro.sample("x", dist.Normal().expand([10]))
...     numpyro.sample("obs", dist.Normal(x, 1.0), obs=jnp.ones(10))

>>> kernel = ESS(model, moves={ESS.DifferentialMove() : 0.8,
...                             ESS.RandomMove() : 0.2})

>>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=2000, num_chains=20, chain_method='vectorized')

>>> mcmc.run(jax.random.PRNGKey(0))
```

```
init_inner_state(rng_key)
    return inner_state

update_active_chains(active, inactive, inner_state)
    return (updated active set of chains, updated inner state)

static RandomMove()
    The Karamanis & Beutler (2020) “Random Move” with parallelization. When this move is used the walkers
    move along random directions. There is no communication between the walkers and this Move corresponds
    to the vanilla Slice Sampling method. This Move should be used for debugging purposes only.

static KDEMove(bw_method=None)
    The Karamanis & Beutler (2020) “KDE Move” with parallelization. When this Move is used the distribu-
    tion of the walkers of the complementary ensemble is traced using a Gaussian Kernel Density Estimation
    methods. The walkers then move along random direction vectos sampled from this distribution.

static GaussianMove()
    The Karamanis & Beutler (2020) “Gaussian Move” with parallelization. When this Move is used the walkers
    move along directions defined by random vectors sampled from the Gaussian approximation of the
    walkers of the complementary ensemble.

static DifferentialMove()
    The Karamanis & Beutler (2020) “Differential Move” with parallelization. When this Move is used the walkers
    move along directions defined by random pairs of walkers sampled (with no replacement) from the complementary ensemble. This is the default choice and performs well along a wide range of target distributions.

hmc(potential_fn=None, potential_fn_gen=None, kinetic_fn=None, algo='NUTS')
    Hamiltonian Monte Carlo inference, using either fixed number of steps or the No U-Turn Sampler (NUTS) with
    adaptive path length.

References:

1. MCMC Using Hamiltonian Dynamics, Radford M. Neal

4.1. Markov Chain Monte Carlo (MCMC)
2. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo, Matthew D. Hoffman, and Andrew Gelman.

3. A Conceptual Introduction to Hamiltonian Monte Carlo, Michael Betancourt

Parameters

- **potential_fn** – Python callable that computes the potential energy given input parameters. The input parameters to potential_fn can be any python collection type, provided that init_params argument to init_kernel has the same type.

- **potential_fn_gen** – Python callable that when provided with model arguments/keyword arguments returns potential_fn. This may be provided to do inference on the same model with changing data. If the data shape remains the same, we can compile sample_kernel once, and use the same for multiple inference runs.

- **kinetic_fn** – Python callable that returns the kinetic energy given inverse mass matrix and momentum. If not provided, the default is euclidean kinetic energy.

- **algo (str)** – Whether to run HMC with fixed number of steps or NUTS with adaptive path length. Default is NUTS.

Returns

a tuple of callables (init_kernel, sample_kernel), the first one to initialize the sampler, and the second one to generate samples given an existing one.

⚠️ Warning

Instead of using this interface directly, we would highly recommend you to use the higher level MCMC API instead.

Example

```python
>>> import jax
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer.hmc import hmc
>>> from numpyro.infer.util import initialize_model
>>> from numpyro.util import fori_collect

>>> true_coefs = jnp.array([1., 2., 3.])
>>> data = random.normal(random.PRNGKey(2), (2000, 3))
>>> labels = dist.Bernoulli(logits=(true_coefs * data).sum(-1)).sample(random.PRNGKey(3))

>>> def model(data, labels):
...     coefs = numpyro.sample('coefs', dist.Normal(jnp.zeros(3), jnp.ones(3)))
...     intercept = numpyro.sample('intercept', dist.Normal(0., 10.))
...     return numpyro.sample('y', dist.Bernoulli(logits=(coefs * data + intercept).sum(-1)), obs=labels)

>>> model_info = initialize_model(random.PRNGKey(0), model, model_args=(data, labels))
```

(continues on next page)
```python
>>> init_kernel, sample_kernel = hmc(model_info.potential_fn, algo='NUTS')
>>> hmc_state = init_kernel(model_info.param_info,
...                           trajectory_length=10,
...                           num_warmup=300)
>>> samples = fori_collect(0, 500, sample_kernel, hmc_state,
...                         transform=lambda state: model_info.postprocess_fn(state.
...                                                                  -> z))
>>> print(jnp.mean(samples['coefs'], axis=0))
[0.9153987 2.0754058 2.9621222]
```

### init_kernel

```python
def init_kernel(init_params, num_warmup, step_size=1.0, inverse_mass_matrix=None, adapt_step_size=True, adapt_mass_matrix=True, dense_mass=False, target_accept_prob=0.8, *, num_steps=None, trajectory_length=6.283185307179586, max_tree_depth=10, find_heuristic_step_size=False, forward_mode_differentiation=False, regularize_mass_matrix=True, model_args=(), model_kwargs=None, rng_key=None)
```

Initializes the HMC sampler.

**Parameters**

- **init_params** – Initial parameters to begin sampling. The type must be consistent with the input type to `potential_fn`.
- **num_warmup** (`int`) – Number of warmup steps; samples generated during warmup are discarded.
- **step_size** (`float`) – Determines the size of a single step taken by the verlet integrator while computing the trajectory using Hamiltonian dynamics. If not specified, it will be set to 1.
- **inverse_mass_matrix** (`numpy.ndarray` or `dict`) – Initial value for inverse mass matrix. This may be adapted during warmup if `adapt_mass_matrix = True`. If no value is specified, then it is initialized to the identity matrix. For a potential_fn with general JAX pytree parameters, the order of entries of the mass matrix is the order of the flattened version of pytree parameters obtained with `jax.tree_flatten`, which is a bit ambiguous (see more at https://jax.readthedocs.io/en/latest/pytrees.html). If `model` is not None, here we can specify a structured block mass matrix as a dictionary, where keys are tuple of site names and values are the corresponding block of the mass matrix. For more information about structured mass matrix, see `dense_mass` argument.
- **adapt_step_size** (`bool`) – A flag to decide if we want to adapt step_size during warm-up phase using Dual Averaging scheme.
- **adapt_mass_matrix** (`bool`) – A flag to decide if we want to adapt mass matrix during warm-up phase using Welford scheme.
- **dense_mass** (`bool` or `list`) – This flag controls whether mass matrix is dense (i.e. full-rank) or diagonal (defaults to `dense_mass=False`). To specify a structured mass matrix, users can provide a list of tuples of site names. Each tuple represents a block in the joint mass matrix. For example, assuming that the model has latent variables “x”, “y”, “z” (where each variable can be multi-dimensional), possible specifications and corresponding mass matrix structures are as follows:
  - `dense_mass=[("x", "y")]: use a dense mass matrix for the joint (x, y) and a diagonal mass matrix for z
  - `dense_mass=[] (equivalent to `dense_mass=False`): use a diagonal mass matrix for the joint (x, y, z)
• dense_mass=[("x", "y", "z")]
  (equivalent to full_mass=True): use a dense mass matrix for the joint (x, y, z)
• dense_mass=[("x"), ("y"), ("z")]: use dense mass matrices for each of x, y, and z (i.e. block-diagonal with 3 blocks)

• target_accept_prob (float) – Target acceptance probability for step size adaptation using Dual Averaging. Increasing this value will lead to a smaller step size, hence the sampling will be slower but more robust. Defaults to 0.8.

• num_steps (int) – if different than None, fix the number of steps allowed for each iteration.

• trajectory_length (float) – Length of a MCMC trajectory for HMC. Default value is $2\pi$.

• max_tree_depth (int) – Max depth of the binary tree created during the doubling scheme of NUTS sampler. Defaults to 10. This argument also accepts a tuple of integers ($d1$, $d2$), where $d1$ is the max tree depth during warmup phase and $d2$ is the max tree depth during post warmup phase.

• find_heuristic_step_size (bool) – whether to a heuristic function to adjust the step size at the beginning of each adaptation window. Defaults to False.

• forward_mode_differentiation (bool) – whether to use forward-mode differentiation or reverse-mode differentiation. By default, we use reverse mode but the forward mode can be useful in some cases to improve the performance. In addition, some control flow utility on JAX such as `jax.lax.while_loop` or `jax.lax.fori_loop` only supports forward-mode differentiation. See JAX’s The Autodiff Cookbook for more information.

• regularize_mass_matrix (bool) – whether or not to regularize the estimated mass matrix for numerical stability during warmup phase. Defaults to True. This flag does not take effect if `adapt_mass_matrix == False`.

• model_args (tuple) – Model arguments if `potential_fn_gen` is specified.

• model_kwargs (dict) – Model keyword arguments if `potential_fn_gen` is specified.

• rng_key (jax.random.PRNGKey) – random key to be used as the source of randomness.

```python
sample_kernel(hmc_state, model_args=(), model_kwargs=None)
```

Given an existing HMCState, run HMC with fixed (possibly adapted) step size and return a new HMCState.

Parameters

• hmc_state – Current sample (and associated state).

• model_args (tuple) – Model arguments if `potential_fn_gen` is specified.

• model_kwargs (dict) – Model keyword arguments if `potential_fn_gen` is specified.

Returns

new proposed HMCState from simulating Hamiltonian dynamics given existing state.

```python
taylor_proxy(reference_params, degree)
```

Control variate for unbiased log likelihood estimation using a Taylor expansion around a reference parameter. Suggested for subsampling in [1].

Parameters

• reference_params (dict) – Model parameterization at MLE or MAP-estimate.

• degree – number of terms in the Taylor expansion, either one or two.

References:
BarkerMHState = <class 'numpyro.infer.barker.BarkerMHState'>
A namedtuple() consisting of the following fields:

- \( i \) - iteration. This is reset to 0 after warmup.
- \( z \) - Python collection representing values (unconstrained samples from the posterior) at latent sites.
- potential_energy - Potential energy computed at the given value of \( z \).
- \( z_{grad} \) - Gradient of potential energy w.r.t. latent sample sites.
- accept_prob - Acceptance probability of the proposal. Note that \( z \) does not correspond to the proposal if it is rejected.
- mean_accept_prob - Mean acceptance probability until current iteration during warmup adaptation or sampling (for diagnostics).
- adapt_state - A HMCAdaptState namedtuple which contains adaptation information during warmup:
  - step_size - Step size to be used by the integrator in the next iteration.
  - inverse_mass_matrix - The inverse mass matrix to be used for the next iteration.
  - mass_matrix_sqrt - The square root of mass matrix to be used for the next iteration. In case of dense mass, this is the Cholesky factorization of the mass matrix.
- rng_key - random number generator seed used for generating proposals, etc.

HMCState = <class 'numpyro.infer.hmc.HMCState'>
A namedtuple() consisting of the following fields:

- \( i \) - iteration. This is reset to 0 after warmup.
- \( z \) - Python collection representing values (unconstrained samples from the posterior) at latent sites.
- \( z_{grad} \) - Gradient of potential energy w.r.t. latent sample sites.
- potential_energy - Potential energy computed at the given value of \( z \).
- energy - Sum of potential energy and kinetic energy of the current state.
- \( r \) - The current momentum variable. If this is None, a new momentum variable will be drawn at the beginning of each sampling step.
- trajectory_length - The amount of time to run HMC dynamics in each sampling step. This field is not used in NUTS.
- num_steps - Number of steps in the Hamiltonian trajectory (for diagnostics). In HMC sampler, trajectory_length should be None for step_size to be adapted. In NUTS sampler, the tree depth of a trajectory can be computed from this field with tree_depth = np.log2(num_steps).astype(int) + 1.
- accept_prob - Acceptance probability of the proposal. Note that \( z \) does not correspond to the proposal if it is rejected.
- mean_accept_prob - Mean acceptance probability until current iteration during warmup adaptation or sampling (for diagnostics).
- diverging - A boolean value to indicate whether the current trajectory is diverging.
- adapt_state - A HMCAdaptState namedtuple which contains adaptation information during warmup:
  - step_size - Step size to be used by the integrator in the next iteration.
  - inverse_mass_matrix - The inverse mass matrix to be used for the next iteration.
– **mass_matrix_sqrt** - The square root of mass matrix to be used for the next iteration. In case of dense mass, this is the Cholesky factorization of the mass matrix.

- **rng_key** - random number generator seed used for the iteration.

**HMGibbsState** = `<class 'numpyro.infer.hmc_gibbs.HMGibbsState'>`

- **z** - a dict of the current latent values (both HMC and Gibbs sites)
- **hmc_state** - current **HMCState**
- **rng_key** - random key for the current step

**SAState** = `<class 'numpyro.infer.sa.SAState'>`

A namedtuple() used in Sample Adaptive MCMC. This consists of the following fields:

- **i** - iteration. This is reset to 0 after warmup.
- **z** - Python collection representing values (unconstrained samples from the posterior) at latent sites.
- **potential_energy** - Potential energy computed at the given value of **z**.
- **accept_prob** - Acceptance probability of the proposal. Note that **z** does not correspond to the proposal if it is rejected.
- **mean_accept_prob** - Mean acceptance probability until current iteration during warmup or sampling (for diagnostics).
- **diverging** - A boolean value to indicate whether the new sample potential energy is diverging from the current one.
- **adapt_state** - A **SAAdaptState** namedtuple which contains adaptation information:
  - **zs** - Step size to be used by the integrator in the next iteration.
  - **pes** - Potential energies of **zs**.
  - **loc** - Mean of those **zs**.
  - **inv_mass_matrix_sqrt** - If using dense mass matrix, this is Cholesky of the covariance of **zs**. Otherwise, this is standard deviation of those **zs**.
- **rng_key** - random number generator seed used for the iteration.

**EnsembleSamplerState** = `<class 'numpyro.infer.ensemble.EnsembleSamplerState'>`

A namedtuple() consisting of the following fields:

- **z** - Python collection representing values (unconstrained samples from the posterior) at latent sites.
- **inner_state** - A namedtuple containing information needed to update half the ensemble.
- **rng_key** - random number generator seed used for generating proposals, etc.

**AIESState** = `<class 'numpyro.infer.ensemble.AIESState'>`

A namedtuple() consisting of the following fields:

- **i** - iteration.
- **accept_prob** - Acceptance probability of the proposal. Note that **z** does not correspond to the proposal if it is rejected.
- **mean_accept_prob** - Mean acceptance probability until current iteration during warmup adaptation or sampling (for diagnostics).
- **rng_key** - random number generator seed used for generating proposals, etc.
ESSState = <class 'numpyro.infer.ensemble.ESSState'>

A namedtuple() used as an inner state for Ensemble Sampler. This consists of the following fields:

- **i** - iteration.
- **n_expansions** - number of expansions in the current batch. Used for tuning mu.
- **n_contractions** - number of contractions in the current batch. Used for tuning mu.
- **mu** - Scale factor. This is tuned if tune_mu=True.
- **rng_key** - random number generator seed used for generating proposals, etc.

### 4.1.2 TensorFlow Kernels

Thin wrappers around TensorFlow Probability (TFP) MCMC kernels. For details on the TFP MCMC kernel interface, see its TransitionKernel docs.

**TFPKernel**

```python
from numpyro.contrib.tfp.mcmc import TFKernel

kernel = TFPKernel[tfp.mcmc.NoUTurnSampler](model, step_size=1.)
```

**Note**

By default, uncalibrated kernels will be inner kernels of the `MetropolisHastings` kernel.

**Note**

For `ReplicaExchangeMC`, TFP requires that the shape of `step_size` of the inner kernel must be `[len(inverse_temperatures), 1]` or `[len(inverse_temperatures), latent_size]`.

**Parameters**

- **model** – Python callable containing Pyro primitives. If model is provided, `potential_fn` will be inferred using the model.
- **potential_fn** – Python callable that computes the target potential energy given input parameters. The input parameters to `potential_fn` can be any python collection type, provided that `init_params` argument to `init()` has the same type.
- **init_strategy** (callable) – a per-site initialization function. See [Initialization Strategies](#) section for available functions.
- **kernel_kwargs** – other arguments to be passed to TFP kernel constructor.
HamiltonianMonteCarlo

class HamiltonianMonteCarlo(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.HamiltonianMonteCarlo with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

MetropolisAdjustedLangevinAlgorithm

class MetropolisAdjustedLangevinAlgorithm(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.langevin.MetropolisAdjustedLangevinAlgorithm with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

NoUTurnSampler

class NoUTurnSampler(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.nuts.NoUTurnSampler with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

RandomWalkMetropolis

class RandomWalkMetropolis(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.random_walk_metropolis.RandomWalkMetropolis with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

ReplicaExchangeMC

class ReplicaExchangeMC(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.replica_exchange_mc.ReplicaExchangeMC with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

SliceSampler

class SliceSampler(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.slice_sampler_kernel.SliceSampler with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.
UncalibratedHamiltonianMonteCarlo

class UncalibratedHamiltonianMonteCarlo:
(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.UncalibratedHamiltonianMonteCarlo with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

UncalibratedLangevin

class UncalibratedLangevin:
(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.UncalibratedLangevin with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

UncalibratedRandomWalk

class UncalibratedRandomWalk:
(model=None, potential_fn=None, init_strategy=<function init_to_uniform>, **kernel_kwargs)

Wraps tensorflow_probability.substrates.jax.mcmc.UncalibratedRandomWalk with TFPKernel. The first argument target_log_prob_fn in TFP kernel construction is replaced by either model or potential_fn.

4.1.3 MCMC Utilities

initialize_model:
(rng_key, model, *, init_strategy=<function init_to_uniform>, dynamic_args=False, model_args=(), model_kwargs=None, forward_mode_differentiation=False, validate_grad=True)

(EXPERIMENTAL INTERFACE) Helper function that calls get_potential_fn() and find_valid_initial_params() under the hood to return a tuple of (init_params_info, potential_fn, postprocess_fn, model_trace).

Parameters

- **rng_key** (jax.random.PRNGKey) – random number generator seed to sample from the prior. The returned init_params will have the batch shape rng_key.shape[:-1].
- **model** – Python callable containing Pyro primitives.
- **init_strategy** (callable) – a per-site initialization function. See Initialization Strategies section for available functions.
- **dynamic_args** (bool) – if True, the potential_fn and constraints_fn are themselves dependent on model arguments. When provided a *model_args, **model_kwargs, they return potential_fn and constraints_fn callables, respectively.
- **model_args** (tuple) – args provided to the model.
- **model_kwargs** (dict) – kwargs provided to the model.
- **forward_mode_differentiation** (bool) – whether to use forward-mode differentiation or reverse-mode differentiation. By default, we use reverse mode but the forward mode can be useful in some cases to improve the performance. In addition, some control flow utility on JAX such as jax.lax.while_loop or jax.lax.fori_loop only supports forward-mode differentiation. See JAX’s The Autodiff Cookbook for more information.
• **validate_grad** *(bool)* – whether to validate gradient of the initial params. Defaults to True.

**Returns**
a namedtuple `ModelInfo` which contains the fields (`param_info`, `potential_fn`, `postprocess_fn`, `model_trace`), where `param_info` is a namedtuple `ParamInfo` containing values from the prior used to initiate MCMC, their corresponding potential energy, and their gradients; `postprocess_fn` is a callable that uses inverse transforms to convert unconstrained HMC samples to constrained values that lie within the site’s support, in addition to returning values at deterministic sites in the model.

**fori_collect** *(lower, upper, body_fun, init_val, transform=<function identity>, progbars=True, return_last_val=False, collection_size=None, thinning=1, **progbar_opts)*

This looping construct works like `fori_loop()` but with the additional effect of collecting values from the loop body. In addition, this allows for post-processing of these samples via `transform`, and progress bar updates. Note that, `progbars=False` will be faster, especially when collecting a lot of samples. Refer to example usage in `hmc()`.

**Parameters**

• **lower** *(int)* – the index to start the collective work. In other words, we will skip collecting the first `lower` values.

• **upper** *(int)* – number of times to run the loop body.

• **body_fun** – a callable that takes a collection of `np.ndarray` and returns a collection with the same shape and dtype.

• **init_val** – initial value to pass as argument to `body_fun`. Can be any Python collection type containing `np.ndarray` objects.

• **transform** – a callable to post-process the values returned by `body_fun`.

• **progbars** – whether to post progress bar updates.

• **return_last_val** *(bool)* – If True, the last value is also returned. This has the same type as `init_val`.

• **thinning** – Positive integer that controls the thinning ratio for retained values. Defaults to 1, i.e. no thinning.

• **collection_size** *(int)* – Size of the returned collection. If not specified, the size will be `(upper - lower) // thinning`. If the size is larger than `(upper - lower) // thinning`, only the top `(upper - lower) // thinning` entries will be non-zero.

• **progbar_opts** – optional additional progress bar arguments. A `diagnostics_fn` can be supplied which when passed the current value from `body_fun` returns a string that is used to update the progress bar postfix. Also a `progbar_desc` keyword argument can be supplied which is used to label the progress bar.

**Returns**
collection with the same type as `init_val` with values collected along the leading axis of `np.ndarray` objects.

**consensus** *(subposteriors, num_draws=None, diagonal=False, rng_key=None)*

Merges subposteriors following consensus Monte Carlo algorithm.

**References:**

• **subposteriors** (*list*) – a list in which each element is a collection of samples.
• **num_draws** (*int*) – number of draws from the merged posterior.
• **diagonal** (*bool*) – whether to compute weights using variance or covariance, defaults to *False* (using covariance).
• **rng_key** (*jax.random.PRNGKey*) – source of the randomness, defaults to *jax.random.PRNGKey(0)*.

**Returns**

if **num_draws** is None, merges subposteriors without resampling; otherwise, returns a collection of **num_draws** samples with the same data structure as each subposterior.

**parametric**(*subposteriors, diagonal=False*)

Merges subposteriors following (embarrassingly parallel) parametric Monte Carlo algorithm.

**References:**

1. *Asymptotically Exact, Embarrassingly Parallel MCMC*, Willie Neiswanger, Chong Wang, Eric Xing

**Parameters**

• **subposteriors** (*list*) – a list in which each element is a collection of samples.
• **diagonal** (*bool*) – whether to compute weights using variance or covariance, defaults to *False* (using covariance).

**Returns**

the estimated mean and variance/covariance parameters of the joined posterior

**parametric_draws**(*subposteriors, num_draws, diagonal=False, rng_key=None*)

Merges subposteriors following (embarrassingly parallel) parametric Monte Carlo algorithm.

**References:**

1. *Asymptotically Exact, Embarrassingly Parallel MCMC*, Willie Neiswanger, Chong Wang, Eric Xing

**Parameters**

• **subposteriors** (*list*) – a list in which each element is a collection of samples.
• **num_draws** (*int*) – number of draws from the merged posterior.
• **diagonal** (*bool*) – whether to compute weights using variance or covariance, defaults to *False* (using covariance).
• **rng_key** (*jax.random.PRNGKey*) – source of the randomness, defaults to *jax.random.PRNGKey(0)*.

**Returns**

a collection of **num_draws** samples with the same data structure as each subposterior.
4.2 Stochastic Variational Inference (SVI)

We offer a brief overview of the three most commonly used ELBO implementations in NumPyro:

- **Trace_ELBO** is our basic ELBO implementation.
- **TraceMeanField_ELBO** is like Trace_ELBO but computes part of the ELBO analytically if doing so is possible.
- **TraceGraph_ELBO** offers variance reduction strategies for models with discrete latent variables. Generally speaking, this ELBO should always be used for models with discrete latent variables.
- **TraceEnum_ELBO** offers variable enumeration strategies for models with discrete latent variables. Generally speaking, this ELBO should always be used for models with discrete latent variables when enumeration is possible.

```python
class SVI(model, guide, optim, loss, **static_kwargs)
Bases: object
Stochastic Variational Inference given an ELBO loss objective.

References
1. SVI Part I: An Introduction to Stochastic Variational Inference in Pyro, (http://pyro.ai/examples/svi_part_i.html)

Example:
```
posterior_samples = predictive(random.PRNGKey(1), data=None)
# use posterior samples to make predictive
predictive = Predictive(model, posterior_samples, params=params, num_samples=1000)
samples = predictive(random.PRNGKey(1), data=None)

Parameters

- **model** – Python callable with Pyro primitives for the model.
- **guide** – Python callable with Pyro primitives for the guide (recognition network).
- **optim** – An instance of _NumpyroOptim, a jax.example_libraries.optimizers.Optimizer or an Optax GradientTransformation. If you pass an Optax optimizer it will automatically be wrapped using numpyro.optim.optax_to_numpyro().

from optax import adam, chain, clip
svi = SVI(model, guide, chain(clip(10.0), adam(1e-3)),
loss=Trace_ELBO())

- **loss** – ELBO loss, i.e. negative Evidence Lower Bound, to minimize.
- **static_kwargs** – static arguments for the model / guide, i.e. arguments that remain constant during fitting.

Returns tuple of (init_fn, update_fn, evaluate).

**init**

**Parameters**

- **rng_key** (jax.random.PRNGKey) – random number generator seed.
- **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
- **init_params** (dict) – if not None, initialize numpyro.param sites with values from this dictionary instead of using init_value in numpyro.param primitives.
- **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**

the initial SVIState

**get_params**

**Parameters**

- **svi_state** – current state of SVI.

**Returns**

the corresponding parameters

**update**

**Parameters**

- **args**, **forward_mode_differentiation**, **kwargs**
• **svi_state** – current state of SVI.
• **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
• **forward_mode_differentiation** – boolean flag indicating whether to use forward mode differentiation. Defaults to False.
• **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**
tuple of *(svi_state, loss)*.

**stable_update**(svi_state, *args, forward_mode_differentiation=False, **kwargs)

Similar to **update** but returns the current state if the the loss or the new state contains invalid values.

**Parameters**
• **svi_state** – current state of SVI.
• **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
• **forward_mode_differentiation** – boolean flag indicating whether to use forward mode differentiation. Defaults to False.
• **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**
tuple of *(svi_state, loss)*.

**run**(rng_key, num_steps, *args, progress_bar=True, stable_update=False,
forward_mode_differentiation=False, init_state=None, init_params=None, **kwargs)

(EXPERIMENTAL INTERFACE) Run SVI with num_steps iterations, then return the optimized parameters and the stacked losses at every step. If num_steps is large, setting progress_bar=False can make the run faster.

**Note**
For a complex training process (e.g. the one requires early stopping, epoch training, varying args/kwargs,...), we recommend to use the more flexible methods **init**, **update**, **evaluate** to customize your training procedure.

**Parameters**
• **rng_key** (*jax.random.PRNGKey*) – random number generator seed.
• **num_steps** (*int*) – the number of optimization steps.
• **args** – arguments to the model / guide
• **progress_bar** (*bool*) – Whether to enable progress bar updates. Defaults to True.
• **stable_update** (*bool*) – whether to use **stable_update** to update the state. Defaults to False.
• **forward_mode_differentiation** (*bool*) – whether to use forward-mode differentiation or reverse-mode differentiation. By default, we use reverse mode but the forward mode can be useful in some cases to improve the performance. In addition, some control flow utility on JAX such as jax.lax.while_loop or jax.lax.fori_loop only supports forward-mode differentiation. See JAX’s The Autodiff Cookbook for more information.
• **init_state** (*SVIState*) – if not None, begin SVI from the final state of previous SVI run. Usage:

```python
svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
svi_result = svi.run(random.PRNGKey(0), 2000, data)
# upon inspection of svi_result the user decides that the model has not converged
# continue from the end of the previous svi run rather than beginning again from iteration 0
svi_result = svi.run(random.PRNGKey(1), 2000, data, init_state=svi_result.state)
```

• **init_params** (*dict*) – if not None, initialize `numpyro.param` sites with values from this dictionary instead of using `init_value` in `numpyro.param` primitives.

• **kwargs** – keyword arguments to the model / guide

Returns

a namedtuple with fields `params` and `losses` where `params` holds the optimized values at `numpyro.param` sites, and `losses` is the collected loss during the process.

Return type

`SVIRunResult`

`evaluate(svi_state, *args, **kwargs)`

Take a single step of SVI (possibly on a batch / minibatch of data).

Parameters

• **svi_state** – current state of SVI.

• **args** – arguments to the model / guide (these can possibly vary during the course of fitting).

• **kwargs** – keyword arguments to the model / guide.

Returns

evaluate ELBO loss given the current parameter values (held within `svi_state.optim_state`).

`SVIState = <class 'numpyro.infer.svi.SVIState'>`

A namedtuple() consisting of the following fields:

• **optim_state** - current optimizer’s state.

• **mutable_state** - extra state to store values of “mutable” sites

• **rng_key** - random number generator seed used for the iteration.

`SVIRunResult = <class 'numpyro.infer.svi.SVIRunResult'>`

A namedtuple() consisting of the following fields:

• **params** - the optimized parameters.

• **state** - the last `SVIState`

• **losses** - the losses collected at every step.


## 4.2.1 ELBO

```python
class ELBO(num_particles=1, vectorize_particles=True):
    Bases: object

    Base class for all ELBO objectives.
    Subclasses should implement either `loss()` or `loss_with Mutable_state()`.

    Parameters
    • `num_particles` – The number of particles/samples used to form the ELBO (gradient) estimators.
    • `vectorize_particles` – Whether to use `jax.vmap` to compute ELBOs over the num_particles-many particles in parallel. If False use `jax.lax.map`. Defaults to True.

    `can_infer_discrete` = False

    `loss(rng_key, param_map, model, guide, *args, **kwargs)`
    Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

    Parameters
    • `rng_key` (`jax.random.PRNGKey`) – random number generator seed.
    • `param_map` (`dict`) – dictionary of current parameter values keyed by site name.
    • `model` – Python callable with NumPyro primitives for the model.
    • `guide` – Python callable with NumPyro primitives for the guide.
    • `args` – arguments to the model / guide (these can possibly vary during the course of fitting).
    • `kwargs` – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

    Returns
    negative of the Evidence Lower Bound (ELBO) to be minimized.

    `loss_with Mutable_state(rng_key, param_map, model, guide, *args, **kwargs)`
    Like `loss()` but also update and return the mutable state, which stores the values at mutable() sites.

    Parameters
    • `rng_key` (`jax.random.PRNGKey`) – random number generator seed.
    • `param_map` (`dict`) – dictionary of current parameter values keyed by site name.
    • `model` – Python callable with NumPyro primitives for the model.
    • `guide` – Python callable with NumPyro primitives for the guide.
    • `args` – arguments to the model / guide (these can possibly vary during the course of fitting).
    • `kwargs` – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

    Returns
    dictionary containing ELBO loss and the mutable state
```

---

174 Chapter 4. Inference
4.2.2 Trace_ELBO

```python
class Trace_ELBO(num_particles=1, vectorize_particles=True, multi_sample_guide=False)
```

Bases: `ELBO`

A trace implementation of ELBO-based SVI. The estimator is constructed along the lines of references [1] and [2]. There are no restrictions on the dependency structure of the model or the guide.

This is the most basic implementation of the Evidence Lower Bound, which is the fundamental objective in Variational Inference. This implementation has various limitations (for example it only supports random variables with reparameterized samplers) but can be used as a template to build more sophisticated loss objectives.

For more details, refer to http://pyro.ai/examples/svi_part_i.html.

References:

2. Black Box Variational Inference, Rajesh Ranganath, Sean Gerrish, David M. Blei

**Parameters**

- `num_particles` – The number of particles/samples used to form the ELBO (gradient) estimators.
- `vectorize_particles` – Whether to use `jax.vmap` to compute ELBOs over the num_particles-many particles in parallel. If False use `jax.lax.map`. Defaults to True.
- `multi_sample_guide` – Whether to make an assumption that the guide proposes multiple samples.

**loss_with_mutable_state**(rng_key, param_map, model, guide, *args, **kwargs)

Like `loss()` but also update and return the mutable state, which stores the values at `mutable()` sites.

**Parameters**

- `rng_key` (*jax.random.PRNGKey*) – random number generator seed.
- `param_map` (*dict*) – dictionary of current parameter values keyed by site name.
- `model` – Python callable with NumPyro primitives for the model.
- `guide` – Python callable with NumPyro primitives for the guide.
- `args` – arguments to the model / guide (these can possibly vary during the course of fitting).
- `kwargs` – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**

dictionary containing ELBO loss and the mutable state
4.2.3 TraceEnum_ELBO

class TraceEnum_ELBO(num_particles=1, max_plate_nesting=inf, vectorize_particles=True)

Bases: ELBO

(EXPERIMENTAL) A TraceEnum implementation of ELBO-based SVI. The gradient estimator is constructed along the lines of reference [1] specialized to the case of the ELBO. It supports arbitrary dependency structure for the model and guide.

Fine-grained conditional dependency information as recorded in the trace is used to reduce the variance of the gradient estimator. In particular provenance tracking [2] is used to find the cost terms that depend on each non-reparameterizable sample site. Enumerated variables are eliminated using the TVE algorithm for plated factor graphs [3].

## References


[2] *Nonstandard Interpretations of Probabilistic Programs for Efficient Inference*, David Wingate, Noah Goodman, Andreas Stuhlmüller, Jeffrey Siskind


\`
\text{can\_infer\_discrete} = \text{True}
\`

loss(rng_key, param_map, model, guide, *args, **kwargs)

Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

**Parameters**

- **rng_key** (*jax.random.PRNGKey*) – random number generator seed.
- **param_map** (*dict*) – dictionary of current parameter values keyed by site name.
- **model** – Python callable with NumPyro primitives for the model.
- **guide** – Python callable with NumPyro primitives for the guide.
- **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
- **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**

negative of the Evidence Lower Bound (ELBO) to be minimized.
4.2.4 TraceGraph_ELBO

class TraceGraph_ELBO(num_particles=1, vectorize_particles=True)

Bases: ELBO

A TraceGraph implementation of ELBO-based SVI. The gradient estimator is constructed along the lines of reference [1] specialized to the case of the ELBO. It supports arbitrary dependency structure for the model and guide. Fine-grained conditional dependency information as recorded in the trace is used to reduce the variance of the gradient estimator. In particular provenance tracking [2] is used to find the cost terms that depend on each non-reparameterizable sample site.

References


[2] Nonstandard Interpretations of Probabilistic Programs for Efficient Inference, David Wingate, Noah Goodman, Andreas Stuhlmüller, Jeffrey Siskind

can_infer_discrete = True

loss(rng_key, param_map, model, guide, *args, **kwargs)

Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

Parameters

- **rng_key** (jax.random.PRNGKey) – random number generator seed.
- **param_map** (dict) – dictionary of current parameter values keyed by site name.
- **model** – Python callable with NumPyro primitives for the model.
- **guide** – Python callable with NumPyro primitives for the guide.
- **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
- **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

Returns

negative of the Evidence Lower Bound (ELBO) to be minimized.

4.2.5 TraceMeanField_ELBO

class TraceMeanField_ELBO(num_particles=1, vectorize_particles=True)

Bases: ELBO

A trace implementation of ELBO-based SVI. This is currently the only ELBO estimator in NumPyro that uses analytic KL divergences when those are available.

⚠️ Warning

This estimator may give incorrect results if the mean-field condition is not satisfied. The mean field condition is a sufficient but not necessary condition for this estimator to be correct. The precise condition is that for every latent variable \( z \) in the guide, its parents in the model must not include any latent variables that are descendants of \( z \) in the guide. Here ‘parents in the model’ and ‘descendants in the guide’ is with respect to the corresponding (statistical) dependency structure. For example, this condition is always satisfied if the model and guide have identical dependency structures.
**loss_with_mutable_state**(*rng_key, param_map, model, guide, *args, **kwargs*)

Like `loss()` but also update and return the mutable state, which stores the values at `mutable()` sites.

**Parameters**

- **rng_key** (*jax.random.PRNGKey*) – random number generator seed.
- **param_map** (*dict*) – dictionary of current parameter values keyed by site name.
- **model** – Python callable with NumPyro primitives for the model.
- **guide** – Python callable with NumPyro primitives for the guide.
- **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
- **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**
dictionary containing ELBO loss and the mutable state

### 4.2.6 RenyiELBO

**class RenyiELBO(alpha=0, num_particles=2)**

Bases: `ELBO`

An implementation of Renyi’s $\alpha$-divergence variational inference following reference [1]. In order for the objective to be a strict lower bound, we require $\alpha \geq 0$. Note, however, that according to reference [1], depending on the dataset $\alpha < 0$ might give better results. In the special case $\alpha = 0$, the objective function is that of the important weighted autoencoder derived in reference [2].

**Parameters**

- **alpha** (*float*) – The order of $\alpha$-divergence. Here $\alpha \neq 1$. Default is 0.
- **num_particles** – The number of particles/samples used to form the objective (gradient) estimator. Default is 2.
- **vectorize_particles** – Whether to use `jax.vmap` to compute ELBOs over the num_particles-many particles in parallel. If False use `jax.lax.map`. Defaults to True.

**Example:**

```python
def model(data):
    with numpyro.plate("batch", 10000, subsample_size=100):
        latent = numpyro.sample("latent", dist.Normal(0, 1))
        batch = numpyro.subsample(data, event_dim=0)
        numpyro.sample("data", dist.Bernoulli(logits=latent), obs=batch)

def guide(data):
    w_loc = numpyro.param("w_loc", 1.)
    w_scale = numpyro.param("w_scale", 1.)
    with numpyro.plate("batch", 10000, subsample_size=100):
```

(continues on next page)
NumPyro Documentation

(continued from previous page)

```python
batch = numpyro.subsample(data, event_dim=0)
loc = w_loc * batch
scale = jnp.exp(w_scale * batch)
numpyro.sample("latent", dist.Normal(loc, scale))

elbo = RenyiELBO(num_particles=10)
svi = SVI(model, guide, optax.adam(0.1), elbo)
```

References:

2. *Importance Weighted Autoencoders*, Yuri Burda, Roger Grosse, Ruslan Salakhutdinov

**loss**(rng_key, param_map, model, guide, *args, **kwargs)

Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

**Parameters**

- **rng_key** (`jax.random.PRNGKey`) – random number generator seed.
- **param_map** (`dict`) – dictionary of current parameter values keyed by site name.
- **model** – Python callable with NumPyro primitives for the model.
- **guide** – Python callable with NumPyro primitives for the guide.
- **args** – arguments to the model / guide (these can possibly vary during the course of fitting).
- **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

**Returns**

Negative of the Evidence Lower Bound (ELBO) to be minimized.

4.3 Automatic Guide Generation

We provide a brief overview of the automatically generated guides available in NumPyro:

- **AutoNormal** and **AutoDiagonalNormal** are our basic mean-field guides. If the latent space is non-euclidean (due to e.g. a positivity constraint on one of the sample sites) an appropriate bijective transformation is automatically used under the hood to map between the unconstrained space (where the Normal variational distribution is defined) to the corresponding constrained space (note this is true for all automatic guides). These guides are a great place to start when trying to get variational inference to work on a model you are developing.

- **AutoMultivariateNormal** and **AutoLowRankMultivariateNormal** also construct Normal variational distributions but offer more flexibility, as they can capture correlations in the posterior. Note that these guides may be difficult to fit in the high-dimensional setting.

- **AutoDelta** is used for computing point estimates via MAP (maximum a posteriori estimation). See here for example usage.

- **AutoBNAFNormal** and **AutoIAFNormal** offer flexible variational distributions parameterized by normalizing flows.

- **AutoDAIS** is a powerful variational inference algorithm that leverages HMC. It can be a good choice for dealing with highly correlated posteriors but may be computationally expensive depending on the nature of the model.
• **AutoSurrogateLikelihoodDAIS** is a powerful variational inference algorithm that leverages HMC and that supports data subsampling.

• **AutoSemiDAIS** constructs a posterior approximation like **AutoDAIS** for local latent variables but provides support for data subsampling during ELBO training by utilizing a parametric guide for global latent variables.

• **AutoLaplaceApproximation** can be used to compute a Laplace approximation.

• **AutoGuideList** can be used to combine multiple automatic guides.

### 4.3.1 AutoGuide

**class AutoGuide** *(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, create_plates=None)*

Bases: ABC

Base class for automatic guides.

Derived classes must implement the **__call__()** method.

**Parameters**

- **model** (*callable*) – a pyro model
- **prefix** (*str*) – a prefix that will be prefixed to all param internal sites
- **init_loc_fn** (*callable*) – A per-site initialization function. See **Initialization Strategies** section for available functions.
- **create_plates** (*callable*) – An optional function inputing the same *args,**kwargs as model() and returning a numpyro.plate or iterable of plates. Plates not returned will be created automatically as usual. This is useful for data subsampling.

**abstract sample_posterior**(rng_key, params, *, sample_shape=())

Generate samples from the approximate posterior over the latent sites in the model.

**Parameters**

- **rng_key** (*jax.random.PRNGKey*) – random key to be used draw samples.
- **params** (*dict*) – Current parameters of model and autoguide. The parameters can be obtained using **get_params()** method from **SVI**.
- **sample_shape** (*tuple*) – sample shape of each latent site, defaults to ().

**Returns**

a dict containing samples drawn the this guide.

**Return type**

dict

**median**(params)

Returns the posterior median value of each latent variable.

**Parameters**

- **params** (*dict*) – A dict containing parameter values. The parameters can be obtained using **get_params()** method from **SVI**.

**Returns**

A dict mapping sample site name to median value.

**Return type**

dict
**quantiles**(params, quantiles)

Returns posterior quantiles each latent variable. Example:

```python
print(guide.quantiles(params, [0.05, 0.5, 0.95]))
```

**Parameters**

- **params (dict)** – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.
- **quantiles (list)** – A list of requested quantiles between 0 and 1.

**Returns**

A dict mapping sample site name to an array of quantile values.

**Return type**

dict

### 4.3.2 AutoGuideList

**class AutoGuideList**(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, create_plates=None)

Bases: `AutoGuide`

Container class to combine multiple automatic guides.

Example usage:

```python
rng_key_init = random.PRNGKey(0)
guide = AutoGuideList(my_model)
guide.append(AutoNormal(
    numpyro.handlers.block(numpyro.handlers.seed(model, rng_seed=0), hide=[
        "coefs"])
))
guide.append(AutoDelta(
    numpyro.handlers.block(numpyro.handlers.seed(model, rng_seed=1), expose=[
        "coefs"])
))
svi = SVI(model, guide, optim, Trace_ELBO())
svi_state = svi.init(rng_key_init, data, labels)
params = svi.get_params(svi_state)
```

**Parameters**

- **model (callable)** – a NumPyro model

**append**(part)

Add an automatic or custom guide for part of the model. The guide should have been created by blocking the model to restrict to a subset of sample sites. No two parts should operate on any one sample site.

**Parameters**

- **part (AutoGuide)** – a partial guide to add
sample_posterior(rng_key, params, *args, sample_shape=(), **kwargs)
Generate samples from the approximate posterior over the latent sites in the model.

Parameters

- rng_key (jax.random.PRNGKey) – random key to be used draw samples.
- params (dict) – Current parameters of model and autoguide. The parameters can be obtained using get_params() method from SVI.
- sample_shape (tuple) – sample shape of each latent site, defaults to ()

Returns
a dict containing samples drawn the this guide.

Return type
dict

median(params)
Returns the posterior median value of each latent variable.

Parameters

- params (dict) – A dict containing parameter values. The parameters can be obtained using get_params() method from SVI.

Returns
A dict mapping sample site name to median value.

Return type
dict

quantiles(params, quantiles)
Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(params, [0.05, 0.5, 0.95]))
```

Parameters

- params (dict) – A dict containing parameter values. The parameters can be obtained using get_params() method from SVI.
- quantiles (list) – A list of requested quantiles between 0 and 1.

Returns
A dict mapping sample site name to an array of quantile values.

Return type
dict

4.3.3 AutoContinuous
class AutoContinuous(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, create_plates=None)
Bases: AutoGuide

Base class for implementations of continuous-valued Automatic Differentiation Variational Inference [1].

Each derived class implements its own _get_posterior() method.

Assumes model structure and latent dimension are fixed, and all latent variables are continuous.

Reference:
1. **Automatic Differentiation Variational Inference**, Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, David M. Blei

**Parameters**

- **model** (`callable`) – A NumPyro model.
- **prefix** (`str`) – a prefix that will be prefixed to all param internal sites.
- **init_loc_fn** (`callable`) – A per-site initialization function. See *Initialization Strategies* section for available functions.

**get_base_dist()**

Returns the base distribution of the posterior when reparameterized as a `TransformedDistribution`. This should not depend on the model’s `*args, **kwargs`.

**get_transform(params)**

Returns the transformation learned by the guide to generate samples from the unconstrained (approximate) posterior.

**Parameters**

- **params** (`dict`) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from SVI.

**Returns**

the transform of posterior distribution

**Return type**

`Transform`

**get_posterior(params)**

Returns the posterior distribution.

**Parameters**

- **params** (`dict`) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from SVI.

**sample_posterior(rng_key, params, *, sample_shape=())**

Generate samples from the approximate posterior over the latent sites in the model.

**Parameters**

- **rng_key** (`jax.random.PRNGKey`) – random key to be used draw samples.
- **params** (`dict`) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from SVI.
- **sample_shape** (`tuple`) – sample shape of each latent site, defaults to ()

**Returns**

a dict containing samples drawn the this guide.

**Return type**

dict
4.3.4 AutoBNAFNormal

class AutoBNAFNormal(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, num_flows=1, hidden_factors=[8, 8])

    Bases: AutoContinuous

This implementation of AutoContinuous uses a Diagonal Normal distribution transformed via a BlockNeuralAutoregressiveTransform to construct a guide over the entire latent space. The guide does not depend on the model’s *args, **kwargs.

Usage:

    guide = AutoBNAFNormal(model, num_flows=1, hidden_factors=[50, 50], ...)
    svi = SVI(model, guide, ...)

References

1. Block Neural Autoregressive Flow, Nicola De Cao, Ivan Titov, Wilker Aziz

Parameters

- model (callable) – a generative model.
- prefix (str) – a prefix that will be prefixed to all param internal sites.
- init_loc_fn (callable) – A per-site initialization function.
- num_flows (int) – the number of flows to be used, defaults to 1.
- hidden_factors (list) – Hidden layer i has hidden_factors[i] hidden units per input dimension. This corresponds to both a and b in reference [1]. The elements of hidden_factors must be integers.

get_base_dist()  
Returns the base distribution of the posterior when reparameterized as a TransformedDistribution. This should not depend on the model’s *args, **kwargs.

4.3.5 AutoDiagonalNormal

class AutoDiagonalNormal(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, init_scale=0.1)

    Bases: AutoContinuous

This implementation of AutoContinuous uses a Normal distribution with a diagonal covariance matrix to construct a guide over the entire latent space. The guide does not depend on the model’s *args, **kwargs.

Usage:

    guide = AutoDiagonalNormal(model, ...)
    svi = SVI(model, guide, ...)

scale_constraint = SoftplusPositive(lower_bound=0.0)

get_base_dist()  
Returns the base distribution of the posterior when reparameterized as a TransformedDistribution. This should not depend on the model’s *args, **kwargs.
get_transform(params)
Returns the transformation learned by the guide to generate samples from the unconstrained (approximate) posterior.

Parameters
params (dict) – Current parameters of model and autoguide. The parameters can be obtained using get_params() method from SVI.

Returns
the transform of posterior distribution

Return type
Transform
get_posterior(params)
Returns a diagonal Normal posterior distribution.

median(params)
Returns the posterior median value of each latent variable.

Parameters
params (dict) – A dict containing parameter values. The parameters can be obtained using get_params() method from SVI.

Returns
A dict mapping sample site name to median value.

Return type
dict
quantiles(params, quantiles)
Returns posterior quantiles each latent variable. Example:

print(guide.quantiles(params, [0.05, 0.5, 0.95]))

Parameters

• params (dict) – A dict containing parameter values. The parameters can be obtained using get_params() method from SVI.

• quantiles (list) – A list of requested quantiles between 0 and 1.

Returns
A dict mapping sample site name to an array of quantile values.

Return type
dict

4.3.6 AutoMultivariateNormal

class AutoMultivariateNormal(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, init_scale=0.1)

Bases: AutoContinuous

This implementation of AutoContinuous uses a MultivariateNormal distribution to construct a guide over the entire latent space. The guide does not depend on the model’s *args, **kwargs.

Usage:
guide = AutoMultivariateNormal(model, ...)
svi = SVI(model, guide, ...)

scale_tril_constraint = ScaledUnitLowerCholesky()

get_base_dist()
Returns the base distribution of the posterior when reparameterized as a TransformedDistribution. This should not depend on the model's *args, **kwargs.

get_transform(params)
Returns the transformation learned by the guide to generate samples from the unconstrained (approximate) posterior.

Parameters
params (dict) – Current parameters of model and autoguide. The parameters can be obtained using get_params() method from SVI.

Returns
the transform of posterior distribution

Return type
Transform
get_posterior(params)
Returns a multivariate Normal posterior distribution.

median(params)
Returns the posterior median value of each latent variable.

Parameters
params (dict) – A dict containing parameter values. The parameters can be obtained using get_params() method from SVI.

Returns
A dict mapping sample site name to median value.

Return type
dict
quantiles(params, quantiles)
Returns posterior quantiles each latent variable. Example:

```python
print(guide.quantiles(params, [0.05, 0.5, 0.95]))
```

Parameters

- params (dict) – A dict containing parameter values. The parameters can be obtained using get_params() method from SVI.
- quantiles (list) – A list of requested quantiles between 0 and 1.

Returns
A dict mapping sample site name to an array of quantile values.

Return type
dict
4.3.7 AutoIAFNormal

```python
class AutoIAFNormal(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, num_flows=3, hidden_dims=None, skip_connections=False, nonlinearity=(<function elementwise.<locals>.<lambda>.), <function elementwise.<locals>.<lambda>>))
```

**Bases:** `AutoContinuous`

This implementation of `AutoContinuous` uses a Diagonal Normal distribution transformed via a `InverseAutoregressiveTransform` to construct a guide over the entire latent space. The guide does not depend on the model’s `*args, **kwargs`.

**Usage:**

```python
guide = AutoIAFNormal(model, hidden_dims=[20], skip_connections=True, ...)
svi = SVI(model, guide, ...)
```

**Parameters**

- `model` *(callable)* – a generative model.
- `prefix` *(str)* – a prefix that will be prefixed to all param internal sites.
- `init_loc_fn` *(callable)* – A per-site initialization function.
- `num_flows` *(int)* – the number of flows to be used, defaults to 3.
- `hidden_dims` *(list)* – the dimensionality of the hidden units per layer. Defaults to `[latent_dim, latent_dim]`.
- `skip_connections` *(bool)* – whether to add skip connections from the input to the output of each flow. Defaults to False.
- `nonlinearity` *(callable)* – the nonlinearity to use in the feedforward network. Defaults to `jax.example_libraries.stax.Elu()`.

**get_base_dist**

Returns the base distribution of the posterior when reparameterized as a `TransformedDistribution`. This should not depend on the model’s `*args, **kwargs`.

4.3.8 AutoLaplaceApproximation

```python
class AutoLaplaceApproximation(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, create_plates=None, hessian_fn=None)
```

**Bases:** `AutoContinuous`

Laplace approximation (quadratic approximation) approximates the posterior \( \log p(z|x) \) by a multivariate normal distribution in the unconstrained space. Under the hood, it uses Delta distributions to construct a MAP (i.e. point estimate) guide over the entire (unconstrained) latent space. Its covariance is given by the inverse of the hessian of \( -\log p(x, z) \) at the MAP point of \( z \).

**Usage:**

```python
guide = AutoLaplaceApproximation(model, ...)
svi = SVI(model, guide, ...)
```

**Parameters**

- `hessian_fn` *(callable)* – EXPERIMENTAL a function that takes a function \( f \) and a vector
x` and returns the hessian of \( f \) at \( x \). By default, we use \( \text{lambda } f, x: \text{jax.hessian}(f)(x) \). Other alternatives can be \( \text{lambda } f, x: \text{jax.jacobian}(\text{jax.jacobian}(f))(x) \) or \( \text{lambda } f, x: \text{jax.hessian}(f)(x) + 1e-3 * \text{jnp.eye}(x\text{.shape[0]}) \). The later example is helpful when the hessian of \( f \) at \( x \) is not positive definite. Note that the output hessian is the precision matrix of the laplace approximation.

**get_base_dist()**

Returns the base distribution of the posterior when reparameterized as a `TransformedDistribution`. This should not depend on the model’s `*args, **kwargs`.

**get_transform(params)**

Returns the transformation learned by the guide to generate samples from the unconstrained (approximate) posterior.

**Parameters**

- **params** (`dict`) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from `SVI`.

**Returns**

the transform of posterior distribution

**Return type**

`Transform`

**get_posterior(params)**

Returns a multivariate Normal posterior distribution.

**sample_posterior(rng_key, params, *, sample_shape=())**

Generate samples from the approximate posterior over the latent sites in the model.

**Parameters**

- **rng_key** (`jax.random.PRNGKey`) – random key to be used draw samples.
- **params** (`dict`) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from `SVI`.
- **sample_shape** (`tuple`) – sample shape of each latent site, defaults to ()

**Returns**

a dict containing samples drawn the this guide.

**Return type**

dict

**median(params)**

Returns the posterior median value of each latent variable.

**Parameters**

- **params** (`dict`) – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.

**Returns**

A dict mapping sample site name to median value.

**Return type**

dict

**quantiles(params, quantiles)**

Returns posterior quantiles each latent variable. Example:
```python
print(guide.quantiles(params, [0.05, 0.5, 0.95]))
```

**Parameters**

- **params** (`dict`) – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.
- **quantiles** (`list`) – A list of requested quantiles between 0 and 1.

**Returns**

A dict mapping sample site name to an array of quantile values.

**Return type**

`dict`

### 4.3.9 AutoLowRankMultivariateNormal

class **AutoLowRankMultivariateNormal**(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, init_scale=0.1, rank=None)

**Bases:** `AutoContinuous`

This implementation of `AutoContinuous` uses a LowRankMultivariateNormal distribution to construct a guide over the entire latent space. The guide does not depend on the model’s `*args, **kwargs`.

**Usage:**

```python
guide = AutoLowRankMultivariateNormal(model, rank=2, ...)
svi = SVI(model, guide, ...)
```

```python
scale_constraint = SoftplusPositive(lower_bound=0.0)
```

**get_base_dist()**

Returns the base distribution of the posterior when reparameterized as a `TransformedDistribution`. This should not depend on the model’s `*args, **kwargs`.

**get_transform(params)**

Returns the transformation learned by the guide to generate samples from the unconstrained (approximate) posterior.

**Parameters**

- **params** (`dict`) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from `SVI`.

**Returns**

the transform of posterior distribution

**Return type**

`Transform`

**get_posterior(params)**

Returns a lowrank multivariate Normal posterior distribution.

**median(params)**

Returns the posterior median value of each latent variable.

**Parameters**

- **params** (`dict`) – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`. 
Returns
A dict mapping sample site name to median value.

Return type
dict

quantiles(params, quantiles)
Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(params, [0.05, 0.5, 0.95]))
```

Parameters
- **params (dict)** – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.
- **quantiles (list)** – A list of requested quantiles between 0 and 1.

Returns
A dict mapping sample site name to an array of quantile values.

Return type
dict

### 4.3.10 AutoNormal

class AutoNormal(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, init_scale=0.1, create_plates=None)

Bases: `AutoGuide`

This implementation of `AutoGuide` uses Normal distributions to construct a guide over the entire latent space. The guide does not depend on the model's *args, **kwargs.

This should be equivalent to `AutoDiagonalNormal`, but with more convenient site names and with better support for mean field ELBO.

Usage:

```
guide = AutoNormal(model)
svi = SVI(model, guide, ...)
```

Parameters
- **model (callable)** – A NumPyro model.
- **prefix (str)** – a prefix that will be prefixed to all param internal sites.
- **init_loc_fn (callable)** – A per-site initialization function. See `Initialization Strategies` section for available functions.
- **init_scale (float)** – Initial scale for the standard deviation of each (unconstrained transformed) latent variable.
- **create_plates (callable)** – An optional function inputing the same *args, **kwargs as model() and returning a `numpyro.plate` or iterable of plates. Plates not returned will be created automatically as usual. This is useful for data subsampling.

```
scale_constraint = SoftplusPositive(lower_bound=0.0)
```
sample_posterior(rng_key, params, *, sample_shape=())

Generate samples from the approximate posterior over the latent sites in the model.

Parameters

- **rng_key** ([jax.random.PRNGKey]) – random key to be used to draw samples.
- **params** ([dict]) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from `SVI`.
- **sample_shape** ([tuple]) – sample shape of each latent site, defaults to ()

Returns

A dict containing samples drawn the this guide.

Return type
dict

median(params)

Returns the posterior median value of each latent variable.

Parameters

- **params** ([dict]) – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.

Returns

A dict mapping sample site name to median value.

Return type
dict

quantiles(params, quantiles)

Returns posterior quantiles each latent variable. Example:

```python
print(guide.quantiles(params, [0.05, 0.5, 0.95]))
```

Parameters

- **params** ([dict]) – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.
- **quantiles** ([list]) – A list of requested quantiles between 0 and 1.

Returns

A dict mapping sample site name to an array of quantile values.

Return type
dict

4.3.11 AutoDelta

class AutoDelta(model, *, prefix='auto', init_loc_fn=<function init_to_median>, create_plates=None)

Bases: AutoGuide

This implementation of `AutoGuide` uses Delta distributions to construct a MAP guide over the entire latent space. The guide does not depend on the model’s *args, **kwargs.
This class does MAP inference in constrained space.

Usage:

```python
guide = AutoDelta(model)
svi = SVI(model, guide, ...)
```

**Parameters**

- `model` (*callable*) – A NumPyro model.
- `prefix` (*str*) – a prefix that will be prefixed to all param internal sites.
- `init_loc_fn` (*callable*) – A per-site initialization function. See *Initialization Strategies* section for available functions.
- `create_plates` (*callable*) – An optional function inputing the same `*args, **kwargs` as `model()` and returning a `numpyro.plate` or iterable of plates. Plates not returned will be created automatically as usual. This is useful for data subsampling.

**sample_posterior** (`rng_key, params, *args, sample_shape=(), **kwargs`)

Generate samples from the approximate posterior over the latent sites in the model.

**Parameters**

- `rng_key` (*jax.random.PRNGKey*) – random key to be used draw samples.
- `params` (*dict*) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from `SVI`.
- `sample_shape` (*tuple*) – sample shape of each latent site, defaults to `()`.

**Returns**

da dict containing samples drawn the this guide.

**Return type**
dict

**median** (*params*)

Returns the posterior median value of each latent variable.

**Parameters**

- `params` (*dict*) – A dict containing parameter values. The parameters can be obtained using `get_params()` method from `SVI`.

**Returns**

A dict mapping sample site name to median value.

**Return type**
dict
4.3.12 AutoDAIS

class AutoDAIS(model, *, K=4, base_dist='diagonal', eta_init=0.01, eta_max=0.1, gamma_init=0.9, prefix='auto', init_loc_fn=<function init_to_uniform>, init_scale=0.1)

Bases: AutoContinuous

This implementation of AutoDAIS uses Differentiable Annealed Importance Sampling (DAIS) [1, 2] to construct a guide over the entire latent space. Samples from the variational distribution (i.e. guide) are generated using a combination of (uncorrected) Hamiltonian Monte Carlo and Annealed Importance Sampling. The same algorithm is called Uncorrected Hamiltonian Annealing in [1].

Note that AutoDAIS cannot be used in conjunction with data subsampling.

Reference:
1. MCMC Variational Inference via Uncorrected Hamiltonian Annealing, Tomas Geffner, Justin Domke
2. Differentiable Annealed Importance Sampling and the Perils of Gradient Noise, Guodong Zhang, Kyle Hsu, Jianing Li, Chelsea Finn, Roger Grosse

Usage:

guide = AutoDAIS(model)
svi = SVI(model, guide, ...)

Parameters

- **model** (callable) – A NumPyro model.
- **prefix** (str) – A prefix that will be prefixed to all param internal sites.
- **K** (int) – A positive integer that controls the number of HMC steps used. Defaults to 4.
- **base_dist** (str) – Controls whether the base Normal variational distribution is parameterized by a “diagonal” covariance matrix or a full-rank covariance matrix parameterized by a lower-triangular “cholesky” factor. Defaults to “diagonal”.
- **eta_init** (float) – The initial value of the step size used in HMC. Defaults to 0.01.
- **eta_max** (float) – The maximum value of the learnable step size used in HMC. Defaults to 0.1.
- **gamma_init** (float) – The initial value of the learnable damping factor used during partial momentum refreshments in HMC. Defaults to 0.9.
- **init_loc_fn** (callable) – A per-site initialization function. See Initialization Strategies section for available functions.
- **init_scale** (float) – Initial scale for the standard deviation of the base variational distribution for each (unconstrained transformed) latent variable. Defaults to 0.1.

sample_posterior(rng_key, params, *, sample_shape=())

Generate samples from the approximate posterior over the latent sites in the model.

Parameters

- **rng_key** (jax.random.PRNGKey) – random key to be used draw samples.
- **params** (dict) – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from `SVI`.
- **sample_shape** (tuple) – sample shape of each latent site, defaults to ()

4.3. Automatic Guide Generation 193
NumPyro Documentation

Returns
a dict containing samples drawn the this guide.

Return type
dict

4.3.13 AutoSemiDAIS

class AutoSemiDAIS(model, local_model, global_guide=None, *, prefix='auto', K=4, eta_init=0.01, eta_max=0.1, gamma_init=0.9, init_scale=0.1, subsample_plate=None, use_global_dais_params=False)

Bases: AutoGuide

This implementation of AutoSemiDAIS [1] combines a parametric variational distribution over global latent variables with Differentiable Annealed Importance Sampling (DAIS) [2, 3] to infer local latent variables. Unlike AutoDAIS this guide can be used in conjunction with data subsampling. Note that the resulting ELBO can be understood as a particular realization of a 'locally enhanced bound' as described in reference [4].

References:
1. Surrogate Likelihoods for Variational Annealed Importance Sampling, Martin Jankowiak, Du Phan
2. MCMC Variational Inference via Uncorrected Hamiltonian Annealing, Tomas Geffner, Justin Domke
3. Differentiable Annealed Importance Sampling and the Perils of Gradient Noise, Guodong Zhang, Kyle Hsu, Jianing Li, Chelsea Finn, Roger Grosse
4. Variational Inference with Locally Enhanced Bounds for Hierarchical Models, Tomas Geffner, Justin Domke

Usage:

def global_model():
    return numpyro.sample("theta", dist.Normal(0, 1))

def local_model(theta):
    with numpyro.plate("data", 8, subsample_size=2):
        tau = numpyro.sample("tau", dist.Gamma(5.0, 5.0))
        numpyro.sample("obs", dist.Normal(0.0, tau), obs=jnp.ones(2))

model = lambda: local_model(global_model())
global_guide = AutoNormal(global_model())
guide = AutoSemiDAIS(model, local_model, global_guide, K=4)
svi = SVI(model, guide, ...)

# sample posterior for particular data subset {3, 7}
with handlers.substitute(data=\{"data": jnp.array([3, 7])\}):
    samples = guide.sample_posterior(random.PRNGKey(1), params)

Parameters:
- model (callable) – A NumPyro model with global and local latent variables.
- local_model (callable) – The portion of model that includes the local latent variables only. The signature of local_model should be the return type of the global model with global latent variables only.
• **global_guide** *(callable)* – A guide for the global latent variables, e.g. an autoguide. The return type should be a dictionary of latent sample sites names and corresponding samples. If there is no global variable in the model, we can set this to None.

• **local_guide** *(callable)* – An optional guide for specifying the DAIS base distribution for local latent variables.

• **prefix** *(str)* – A prefix that will be prefixed to all internal sites.

• **K** *(int)* – A positive integer that controls the number of HMC steps used. Defaults to 4.

• **eta_init** *(float)* – The initial value of the step size used in HMC. Defaults to 0.01.

• **eta_max** *(float)* – The maximum value of the learnable step size used in HMC. Defaults to 0.1.

• **gamma_init** *(float)* – The initial value of the learnable damping factor used during partial momentum refreshments in HMC. Defaults to 0.9.

• **init_scale** *(float)* – Initial scale for the standard deviation of the variational distribution for each (unconstrained transformed) local latent variable. Defaults to 0.1.

• **subsample_plate** *(str)* – Optional name of the subsample plate site. This is required when the model has a subsample plate without subsample_size specified or the model has a subsample plate with subsample_size equal to the plate size.

• **use_global_dais_params** *(bool)* – Whether parameters controlling DAIS dynamic (HMC step size, HMC mass matrix, etc.) should be global (i.e. common to all data points in the subsample plate) or local (i.e. each data point in the subsample plate has individual parameters). Note that we do not use global parameters for the base distribution.

```python
def sample_posterior(rng_key, params, *args, sample_shape=(), **kwargs):
    # Generate samples from the approximate posterior over the latent sites in the model.
```

Parameters

- **rng_key** *(jax.random.PRNGKey)* – random key to be used draw samples.

- **params** *(dict)* – Current parameters of model and autoguide. The parameters can be obtained using `get_params()` method from SVI.

- **sample_shape** *(tuple)* – sample shape of each latent site, defaults to ()

Returns

a dict containing samples drawn the this guide.

Return type

dict

### 4.3.14 AutoSurrogateLikelihoodDAIS

```python
class AutoSurrogateLikelihoodDAIS(model, surrogate_model, *, K=4, eta_init=0.01, eta_max=0.1, 
gamma_init=0.9, prefix='auto', base_dist='diagonal', 
init_loc_fn=<function init_to_uniform>, init_scale=0.1)
```

Bases: `AutoDAIS`

This implementation of **AutoSurrogateLikelihoodDAIS** provides a mini-batchable family of variational distributions as described in [1]. It combines a user-provided surrogate likelihood with Differentiable Annealed Importance Sampling (DAIS) [2, 3]. It is not applicable to models with local latent variables (see **AutoSemiDAIS**), but unlike **AutoDAIS**, it *can* be used in conjunction with data subsampling.

---

4.3. Automatic Guide Generation 195
Reference:

1. Surrogate likelihoods for variational annealed importance sampling, Martin Jankowiak, Du Phan
2. MCMC Variational Inference via Uncorrected Hamiltonian Annealing, Tomas Geffner, Justin Domke
3. Differentiable Annealed Importance Sampling and the Perils of Gradient Noise, Guodong Zhang, Kyle Hsu, Jianing Li, Chelsea Finn, Roger Grosse

Usage:

```python
# logistic regression model for data {X, Y}
def model(X, Y):
    theta = numpyro.sample("theta", dist.Normal(jnp.zeros(2), jnp.ones(2)).to_event(1))
    with numpyro.plate("N", 100, subsample_size=10):
        X_batch = numpyro.subsample(X, event_dim=1)
        Y_batch = numpyro.subsample(Y, event_dim=0)
        numpyro.sample("obs", dist.Bernoulli(logits=theta @ X_batch.T), obs=Y_batch)

# surrogate model defined by prior and surrogate likelihood.
# a convenient choice for specifying the latter is to compute the likelihood on
# a randomly chosen data subset (here {X_surr, Y_surr} of size 20) and then use
# handlers.scale to scale the log likelihood by a vector of learnable weights.
def surrogate_model(X_surr, Y_surr):
    theta = numpyro.sample("theta", dist.Normal(jnp.zeros(2), jnp.ones(2)).to_event(1))
    omegas = numpyro.param("omegas", 5.0 * jnp.ones(20), constraint=dist.constraints.positive)
    with numpyro.plate("N", 20), numpyro.handlers.scale(scale=omegas):
        numpyro.sample("obs", dist.Bernoulli(logits=theta @ X_surr.T), obs=Y_surr)

guide = AutoSurrogateLikelihoodDAIS(model, surrogate_model)
svi = SVI(model, guide, ...)
```

Parameters

- `model` (callable): A NumPyro model.
- `surrogate_model` (callable): A NumPyro model that is used as a surrogate model for guiding the HMC dynamics that define the variational distribution. In particular, `surrogate_model` should contain the same prior as `model` but should contain a cheap-to-evaluate parametric ansatz for the likelihood. A simple ansatz for the latter involves computing the likelihood for a fixed subset of the data and scaling the resulting log likelihood by a learnable vector of positive weights. See the usage example above.
- `prefix` (str): A prefix that will be prefixed to all param internal sites.
- `K` (int): A positive integer that controls the number of HMC steps used. Defaults to 4.
- `base_dist` (str): Controls whether the base Normal variational distribution is parameterized by a “diagonal” covariance matrix or a full-rank covariance matrix parameterized by a lower-triangular “cholesky” factor. Defaults to “diagonal”.
- `eta_init` (float): The initial value of the step size used in HMC. Defaults to 0.01.
4.4 Reparameterizers

The `numpyro.infer.reparam` module contains reparameterization strategies for the `numpyro.handlers.reparam` effect. These are useful for altering geometry of a poorly-conditioned parameter space to make the posterior better shaped. These can be used with a variety of inference algorithms, e.g. Auto*Normal guides and MCMC.

class Reparam
Bases: ABC

Base class for reparameterizers.

4.4.1 Loc-Scale Decentering

class LocScaleReparam(centered=None, shape_params=())
Bases: Reparam

Generic decentering reparameterizer [1] for latent variables parameterized by `loc` and `scale` (and possibly additional `shape_params`).

This reparameterization works only for latent variables, not likelihoods.

References:


Parameters

- **centered** (`float`) – optional centered parameter. If None (default) learn a per-site per-element centering parameter in $[0, 1]$ initialized at value 0.5. To sample the parameter, consider using `lift` handler with a prior like `Uniform(0, 1)` to cast the parameter to a latent variable. If 0, fully decenter the distribution; if 1, preserve the centered distribution unchanged.

- **shape_params** (tuple or list) – list of additional parameter names to copy unchanged from the centered to decentered distribution.

__call__(name, fn, obs)

Parameters

- **name** (str) – A sample site name.
- **fn** (`Distribution`) – A distribution.
- **obs** (`numpy.ndarray`) – Observed value or None.
Returns
A pair \((\text{new}_\text{fn}, \text{value})\).

### 4.4.2 Neural Transport

class \textbf{NeuTraReparam}(\textit{guide, params})

\begin{itemize}
  \item \textbf{Bases:} \texttt{Reparam}
\end{itemize}

Neural Transport reparameterizer \cite{hoffman2019} of multiple latent variables.

This uses a trained \texttt{AutoContinuous} guide to alter the geometry of a model, typically for use e.g. in MCMC. Example usage:

```python
# Step 1. Train a guide
guide = AutoIAFNormal(model)
svi = SVI(model, guide, ...)
# ...train the guide...

# Step 2. Use trained guide in NeuTra MCMC
neutra = NeuTraReparam(guide)
model = neutra.reparam(model)
nuts = NUTS(model)
# ...now use the model in HMC or NUTS...
```

This reparameterization works only for latent variables, not likelihoods. Note that all sites must share a single common \texttt{NeuTraReparam} instance, and that the model must have static structure.

\cite{hoffman2019} Hoffman, M. et al. (2019)


Parameters

- \texttt{guide} (\texttt{AutoContinuous}) – A guide.
- \texttt{params} – trained parameters of the guide.

\texttt{reparam}(\textit{fn=\texttt{None}})

\texttt{__call__}(\textit{name, fn, obs})

Parameters

- \texttt{name} (\texttt{str}) – A sample site name.
- \texttt{fn} (\texttt{Distribution}) – A distribution.
- \texttt{obs} (\texttt{numpy.ndarray}) – Observed value or None.

Returns

A pair \((\text{new}_\text{fn}, \text{value})\).

\texttt{transform_sample}(\textit{latent})

Given latent samples from the warped posterior (with possible batch dimensions), return a \texttt{dict} of samples from the latent sites in the model.

Parameters

- \texttt{latent} – sample from the warped posterior (possibly batched).
Returns

a dict of samples keyed by latent sites in the model.

Return type
dict

4.4.3 Transformed Distributions

class TransformReparam
Bases: Reparam
Reparameterizer for TransformedDistribution latent variables.
This is useful for transformed distributions with complex, geometry-changing transforms, where the posterior has simple shape in the space of base_dist.
This reparameterization works only for latent variables, not likelihoods.
__call__(name, fn, obs)

Parameters

• name (str) – A sample site name.
• fn (Distribution) – A distribution.
• obs (numpy.ndarray) – Observed value or None.

Returns

A pair (new_fn, value).

4.4.4 Projected Normal Distributions

class ProjectedNormalReparam
Bases: Reparam
Reparametrizer for ProjectedNormal latent variables.
This reparameterization works only for latent variables, not likelihoods.
__call__(name, fn, obs)

Parameters

• name (str) – A sample site name.
• fn (Distribution) – A distribution.
• obs (numpy.ndarray) – Observed value or None.

Returns

A pair (new_fn, value).
4.4.5 Circular Distributions

class CircularReparam
   Bases: Reparam

   Reparametrizer for VonMises latent variables.

   __call__({name, fn, obs})

   Parameters
   - name (str) – A sample site name.
   - fn (Distribution) – A distribution.
   - obs (numpy.ndarray) – Observed value or None.

   Returns
   A pair (new_fn, value).

4.4.6 Explicit Reparameterization

class ExplicitReparam(transform)
   Bases: Reparam

   Explicit reparametrizer of a latent variable \( x \) to a transformed space \( y = \text{transform}(x) \) with more amenable geometry. This reparametrizer is similar to TransformReparam but allows reparametrizations to be decoupled from the model declaration.

   Parameters
   transform – Bijective transform to the reparameterized space.

   Example:

   ```python
   >>> from jax import random
   >>> from jax import numpy as jnp
   >>> import numpyro
   >>> from numpyro import handlers, distributions as dist
   >>> from numpyro.infer import MCMC, NUTS
   >>> from numpyro.infer.reparam import ExplicitReparam
   >>>
   >>> def model():
   ...   numpyro.sample("x", dist.Gamma(4, 4))
   >>>
   >>> # Sample in unconstrained space using a soft-plus instead of exp transform.
   >>> reparam = ExplicitReparam(dist.transforms.SoftplusTransform().inv)
   >>> reparameterized = handlers.reparam(model, {"x": reparam})
   >>> kernel = NUTS(model=reparameterized)
   >>> mcmc = MCMC(kernel, num_warmup=1000, num_samples=1000, num_chains=1)
   >>> mcmc.run(random.PRNGKey(2))
   sample: 100%|          | 2000/2000 [00:00<00:00, 2306.47it/s, 3 steps/sec of size 9.65e-01, acc. prob=0.93]
   ```

   __call__({name, fn, obs})

   Parameters
   - name (str) – A sample site name.
• **fn** (*Distribution*) – A distribution.
• **obs** (*numpy.ndarray*) – Observed value or None.

Returns
A pair (new_fn, value).

## 4.5 Funsor-based NumPyro

See the GitHub repo for more information about Funsor.

### 4.5.1 Effect handlers

#### class enum

```python
class enum(fn=None, first_available_dim=None)

Bases: BaseEnumMessenger

Enumerates in parallel over discrete sample sites marked infer={"enumerate": "parallel"}.

Parameters
- **fn** (callable) – Python callable with NumPyro primitives.
- **first_available_dim** (int) – The first tensor dimension (counting from the right) that is available for parallel enumeration. This dimension and all dimensions left may be used internally by Pyro. This should be a negative integer or None.

process_message(msg)
```

#### class infer_config

```python
class infer_config(fn=None, config_fn=None)

Bases: Messenger

Given a callable fn that contains NumPyro primitive calls and a callable config_fn taking a trace site and returning a dictionary, updates the value of the infer kwarg at a sample site to config_fn(site).

Parameters
- **fn** – a stochastic function (callable containing NumPyro primitive calls)
- **config_fn** – a callable taking a site and returning an infer dict

process_message(msg)
```

#### markov

```python
markov(fn=None, history=1, keep=False)

Markov dependency declaration.

This is a statistical equivalent of a memory management arena.

Parameters
- **fn** (callable) – Python callable with NumPyro primitives.
- **history** (int) – The number of previous contexts visible from the current context. Defaults to 1. If zero, this is similar to `numpyro.primitives.plate`.
- **keep** (bool) – If true, frames are replayable. This is important when branching: if keep=True, neighboring branches at the same level can depend on each other; if keep=False, neighboring branches are independent (conditioned on their shared ancestors).
```
class plate(name, size, subsample_size=None, dim=None)

Bases: GlobalNamedMessenger

An alternative implementation of numpyro.primitives.plate primitive. Note that only this version is compatible with enumeration.

There is also a context manager plate_to_enum_plate() which converts numpyro.plate statements to this version.

Parameters

• name (str) – Name of the plate.
• size (int) – Size of the plate.
• subsample_size (int) – Optional argument denoting the size of the mini-batch. This can be used to apply a scaling factor by inference algorithms. e.g. when computing ELBO using a mini-batch.
• dim (int) – Optional argument to specify which dimension in the tensor is used as the plate dim. If None (default), the rightmost available dim is allocated.

process_message(msg)

postprocess_message(msg)

to_data(x, name_to_dim=None, dim_type=DimType.LOCAL)

A primitive to extract a python object from a Funsor.

Parameters

• x (Funsor) – A funsor object
• name_to_dim (OrderedDict) – An optional inputs hint which maps dimension names from x to dimension positions of the returned value.
• dim_type (int) – Either 0, 1, or 2. This optional argument indicates a dimension should be treated as ‘local’, ‘global’, or ‘visible’, which can be used to interact with the global DimStack.

Returns

A non-funsor equivalent to x.

to_funsor(x, output=None, dim_to_name=None, dim_type=DimType.LOCAL)

A primitive to convert a Python object to a Funsor.

Parameters

• x – An object.
• output (funsor.domains.Domain) – An optional output hint to uniquely convert a data to a Funsor (e.g. when x is a string).
• dim_to_name (OrderedDict) – An optional mapping from negative batch dimensions to name strings.
• dim_type (int) – Either 0, 1, or 2. This optional argument indicates a dimension should be treated as ‘local’, ‘global’, or ‘visible’, which can be used to interact with the global DimStack.

Returns

A Funsor equivalent to x.
class `trace`(fn=None)

This version of `trace` handler records information necessary to do packing after execution. Each sample site is annotated with a “dim_to_name” dictionary, which can be passed directly to `to_funsor()`.

```python
postprocess_message(msg)
```

## 4.5.2 Inference Utilities

### config_enumerate(fn=None, default='parallel')

Configures enumeration for all relevant sites in a NumPyro model.

When configuring for exhaustive enumeration of discrete variables, this configures all sample sites whose distribution satisfies `.has_enumerate_support == True`.

This can be used as either a function:

```python
model = config_enumerate(model)
```

or as a decorator:

```python
@config_enumerate
def model(*args, **kwargs):
    ...
```

#### Note

Currently, only `default='parallel'` is supported.

**Parameters**

- `fn` (callable) – Python callable with NumPyro primitives.
- `default` (str) – Which enumerate strategy to use, one of “sequential”, “parallel”, or None. Defaults to “parallel”.

### infer_discrete(fn=None, first_available_dim=None, temperature=1, rng_key=None)

A handler that samples discrete sites marked with `site["infer"]["enumerate"] = "parallel"` from the posterior, conditioned on observations.

Example:

```python
@infer_discrete(first_available_dim=-1, temperature=0)
@config_enumerate
def viterbi_decoder(data, hidden_dim=10):
    transition = 0.3 / hidden_dim + 0.7 * jnp.eye(hidden_dim)
    means = jnp.arange(float(hidden_dim))
    states = [0]
    for t in markov(range(len(data))):
```

(continues on next page)
states.append(numpyro.sample("states_{t}\).format(t),
    dist.Categorical(transition[states[-1]]))

numpyro.sample("obs_{t}\).format(t),
    dist.Normal(means[states[-1]], 1.),
    obs=data[t])

return states  # returns maximum likelihood states

Parameters

- **fn** – a stochastic function (callable containing NumPyro primitive calls)
- **first_available_dim** (int) – The first tensor dimension (counting from the right) that is available for parallel enumeration. This dimension and all dimensions left may be used internally by Pyro. This should be a negative integer.
- **temperature** (int) – Either 1 (sample via forward-filter backward-sample) or 0 (optimize via Viterbi-like MAP inference). Defaults to 1 (sample).
- **rng_key** (jax.random.PRNGKey) – a random number generator key, to be used in cases temperature=1 or first_available_dim is None.

**log_density**(model, model_args, model_kwargs, params)

Similar to `numpyro.infer.util.log_density()` but works for models with discrete latent variables. Internally, this uses funsor to marginalize discrete latent sites and evaluate the joint log probability.

Parameters

- **model** – Python callable containing NumPyro primitives. Typically, the model has been enumerated by using `enum` handler:

  ```python
  def model(*args, **kwargs):
      ...
  
  log_joint = log_density(enum(config Enumerate(model)), args, kwargs, ...
  
  model_args (tuple) – args provided to the model.
- **model_kwargs** (dict) – kwargs provided to the model.
- **params** (dict) – dictionary of current parameter values keyed by site name.

Returns

log of joint density and a corresponding model trace

**plate_to_enum_plate()**

A context manager to replace `numpyro.plate` statement by a funsor-based `plate`

This is useful when doing inference for the usual NumPyro programs with `numpyro.plate` statements. For example, to get trace of a `model` whose discrete latent sites are enumerated, we can use:

```python
enum_model = numpyro.contrib.funsor.enum(model)
with plate_to_enum_plate():
    model_trace = numpyro.contrib.funsor.trace(enum_model).get_trace(
        *model_args, **model_kwargs)
```
4.6 Optimizers

Optimizer classes defined here are light wrappers over the corresponding optimizers sourced from jax.example_libraries.optimizers with an interface that is better suited for working with NumPyro inference algorithms.

4.6.1 Adam

class Adam(*args, **kwargs)
    Wrapper class for the JAX optimizer: adam()

    eval_and_stable_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState],
                           forward_mode_differentiation: bool = False)

    Like eval_and_update() but when the value of the objective function or the gradients are not finite, we
    will not update the input state and will set the objective output to nan.

    Parameters
    • fn – objective function.
    • state – current optimizer state.
    • forward_mode_differentiation – boolean flag indicating whether to use forward
      mode differentiation.

    Returns
    a pair of the output of objective function and the new optimizer state.

eval_and_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState],
                  forward_mode_differentiation: bool = False)

    Performs an optimization step for the objective function fn. For most optimizers, the update is performed
    based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as
    Minimize, the update is performed by reevaluating the function multiple times to get optimal parameters.

    Parameters
    • fn – an objective function returning a pair where the first item is a scalar loss function to
      be differentiated and the second item is an auxiliary output.
    • state – current optimizer state.
    • forward_mode_differentiation – boolean flag indicating whether to use forward
      mode differentiation.

    Returns
    a pair of the output of objective function and the new optimizer state.

get_params(state: tuple[int, _OptState]) → _Params

    Get current parameter values.

    Parameters
    state – current optimizer state.

    Returns
    collection with current value for parameters.

init(params: _Params) → tuple[int, _OptState]

    Initialize the optimizer with parameters designated to be optimized.
Parameters

params – a collection of numpy arrays.

Returns

initial optimizer state.

update\( (g: _\text{Params}, state: \text{tuple}[\text{int}, _\text{OptState}]) \rightarrow \text{tuple}[\text{int}, _\text{OptState}] \)

Gradient update for the optimizer.

Parameters

• g – gradient information for parameters.

• state – current optimizer state.

Returns

new optimizer state after the update.

### 4.6.2 Adagrad

class Adagrad(*args, **kwargs)

Wrapper class for the JAX optimizer: adagrad()

eval_and_stable_update\( (fn: \text{Callable}[[\text{Any}, \text{tuple}], state: \text{tuple}[\text{int}, _\text{OptState}], forward\_mode\_differentiation: \text{bool} = \text{False}]\)

Like eval_and_update() but when the value of the objective function or the gradients are not finite, we will not update the input state and will set the objective output to nan.

Parameters

• fn – objective function.

• state – current optimizer state.

• forward_mode_differentiation – boolean flag indicating whether to use forward mode differentiation.

Returns

a pair of the output of objective function and the new optimizer state.

eval_and_update\( (fn: \text{Callable}[[\text{Any}, \text{tuple}], state: \text{tuple}[\text{int}, _\text{OptState}], forward\_mode\_differentiation: \text{bool} = \text{False}]\)

Performs an optimization step for the objective function fn. For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as Minimize, the update is performed by reevaluating the function multiple times to get optimal parameters.

Parameters

• fn – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.

• state – current optimizer state.

• forward_mode_differentiation – boolean flag indicating whether to use forward mode differentiation.

Returns

a pair of the output of objective function and the new optimizer state.
get_params(state: tuple[int, _OptState]) → _Params
Get current parameter values.

Parameters
state – current optimizer state.

Returns
collection with current value for parameters.

init(params: _Params) → tuple[int, _OptState]
Initialize the optimizer with parameters designated to be optimized.

Parameters
params – a collection of numpy arrays.

Returns
initial optimizer state.

update(g: _Params, state: tuple[int, _OptState]) → tuple[int, _OptState]
Gradient update for the optimizer.

Parameters
• g – gradient information for parameters.
• state – current optimizer state.

Returns
new optimizer state after the update.

4.6.3 ClippedAdam
class ClippedAdam(*args, clip_norm=10.0, **kwargs)
Adam optimizer with gradient clipping.

Parameters
clip_norm (float) – All gradient values will be clipped between [-clip_norm, clip_norm].

Reference:
eval_and_stable_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)
Like eval_and_update() but when the value of the objective function or the gradients are not finite, we will not update the input state and will set the objective output to nan.

Parameters
• fn – objective function.
• state – current optimizer state.
• forward_mode_differentiation – boolean flag indicating whether to use forward mode differentiation.

Returns
a pair of the output of objective function and the new optimizer state.
eval_and_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)

Performs an optimization step for the objective function fn. For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as Minimize, the update is performed by reevaluating the function multiple times to get optimal parameters.

Parameters
- fn – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.
- state – current optimizer state.
- forward_mode_differentiation – boolean flag indicating whether to use forward mode differentiation.

Returns
- a pair of the output of objective function and the new optimizer state.

get_params(state: tuple[int, _OptState]) → _Params
Get current parameter values.

Parameters
- state – current optimizer state.

Returns
- collection with current value for parameters.

init(params: _Params) → tuple[int, _OptState]
Initialize the optimizer with parameters designated to be optimized.

Parameters
- params – a collection of numpy arrays.

Returns
- initial optimizer state.

update(g, state)
Gradient update for the optimizer.

Parameters
- g – gradient information for parameters.
- state – current optimizer state.

Returns
- new optimizer state after the update.

4.6.4 Minimize

class Minimize(method='BFGS', **kwargs)
Wrapper class for the JAX minimizer: minimize().

Example:

```python
>>> from numpy.testing import assert_allclose
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
```
>>> import numpyro.distributions as dist
>>> from numpyro.infer import SVI, Trace_ELBO
>>> from numpyro.infer.autoguide import AutoLaplaceApproximation

>>> def model(x, y):
    ...     a = numpyro.sample("a", dist.Normal(0, 1))
    ...     b = numpyro.sample("b", dist.Normal(0, 1))
    ...     with numpyro.plate("N", y.shape[0]):
    ...         numpyro.sample("obs", dist.Normal(a + b * x, 0.1), obs=y)

>>> x = jnp.linspace(0, 10, 100)
>>> y = 3 * x + 2

>>> optimizer = numpyro.optim.Minimize()

>>> guide = AutoLaplaceApproximation(model)

>>> svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

>>> init_state = svi.init(random.PRNGKey(0), x, y)

>>> optimal_state, loss = svi.update(init_state, x, y)

>>> params = svi.get_params(optimal_state)

>>> quantiles = guide.quantiles(params, 0.5)

>>> assert_allclose(quantiles["a"], 2., atol=1e-3)
>>> assert_allclose(quantiles["b"], 3., atol=1e-3)

```python
def eval_and_stable_update(fn: Callable[[Any], tuple], state: tuple[int, OptState], forward_mode_differentiation: bool = False)
```

Like `eval_and_update()` but when the value of the objective function or the gradients are not finite, we will not update the input `state` and will set the objective output to `nan`.

**Parameters**

- `fn` – objective function.
- `state` – current optimizer state.
- `forward_mode_differentiation` – boolean flag indicating whether to use forward mode differentiation.

**Returns**

a pair of the output of objective function and the new optimizer state.

```python
def eval_and_update(fn: Callable[[Any], tuple], state: tuple[int, OptState], forward_mode_differentiation=False)
```

Performs an optimization step for the objective function `fn`. For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as `Minimize`, the update is performed by reevaluating the function multiple times to get optimal parameters.

**Parameters**

- `fn` – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.
- `state` – current optimizer state.
- `forward_mode_differentiation` – boolean flag indicating whether to use forward mode differentiation.

**Returns**

a pair of the output of objective function and the new optimizer state.
**get_params** *(state: tuple[int, _OptState]) → _Params*

Get current parameter values.

**Parameters**

- `state` – current optimizer state.

**Returns**

collection with current value for parameters.

**init** *(params: _Params) → tuple[int, _OptState]*

Initialize the optimizer with parameters designated to be optimized.

**Parameters**

- `params` – a collection of numpy arrays.

**Returns**

initial optimizer state.

**update** *(g: _Params, state: tuple[int, _OptState]) → tuple[int, _OptState]*

Gradient update for the optimizer.

**Parameters**

- `g` – gradient information for parameters.
- `state` – current optimizer state.

**Returns**

new optimizer state after the update.

### 4.6.5 Momentum

**class Momentum** *(args, **kwargs)*

Wrapper class for the JAX optimizer: `momentum()`

**eval_and_stable_update** *(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)*

Like `eval_and_update()` but when the value of the objective function or the gradients are not finite, we will not update the input `state` and will set the objective output to `nan`.

**Parameters**

- `fn` – objective function.
- `state` – current optimizer state.
- `forward_mode_differentiation` – boolean flag indicating whether to use forward mode differentiation.

**Returns**

a pair of the output of objective function and the new optimizer state.

**eval_and_update** *(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)*

Performs an optimization step for the objective function `fn`. For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as `Minimize`, the update is performed by reevaluating the function multiple times to get optimal parameters.

**Parameters**
• **fn** – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.

• **state** – current optimizer state.

• **forward_mode_differentiation** – boolean flag indicating whether to use forward mode differentiation.

Returns
a pair of the output of objective function and the new optimizer state.

```python
get_params(state: tuple[int, _OptState]) → _Params
```
Get current parameter values.

Parameters
state – current optimizer state.

Returns
collection with current value for parameters.

```python
init(params: _Params) → tuple[int, _OptState]
```
Initialize the optimizer with parameters designated to be optimized.

Parameters
params – a collection of numpy arrays.

Returns
initial optimizer state.

```python
update(g: _Params, state: tuple[int, _OptState]) → tuple[int, _OptState]
```
Gradient update for the optimizer.

Parameters

• **g** – gradient information for parameters.

• **state** – current optimizer state.

Returns
new optimizer state after the update.

### 4.6.6 RMSProp

```python
class RMSProp(*args, **kwargs)
```
Wrapper class for the JAX optimizer: `rmsprop()`

```python
eval_and_stable_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)
```
Like `eval_and_update()` but when the value of the objective function or the gradients are not finite, we will not update the input state and will set the objective output to `nan`.

Parameters

• **fn** – objective function.

• **state** – current optimizer state.

• **forward_mode_differentiation** – boolean flag indicating whether to use forward mode differentiation.

Returns
a pair of the output of objective function and the new optimizer state.
**eval_and_update** *(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)*

Performs an optimization step for the objective function \( fn \). For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as **Minimize**, the update is performed by reevaluating the function multiple times to get optimal parameters.

**Parameters**

- \( fn \) – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.
- \( state \) – current optimizer state.
- \( forward_mode_differentiation \) – boolean flag indicating whether to use forward mode differentiation.

**Returns**

a pair of the output of objective function and the new optimizer state.

**get_params** *(state: tuple[int, _OptState]) \rightarrow _Params)*

Get current parameter values.

**Parameters**

- \( state \) – current optimizer state.

**Returns**

collection with current value for parameters.

**init** *(params: _Params) \rightarrow tuple[int, _OptState]*

Initialize the optimizer with parameters designated to be optimized.

**Parameters**

- \( params \) – a collection of numpy arrays.

**Returns**

initial optimizer state.

**update** *(g: _Params, state: tuple[int, _OptState]) \rightarrow tuple[int, _OptState]*

Gradient update for the optimizer.

**Parameters**

- \( g \) – gradient information for parameters.
- \( state \) – current optimizer state.

**Returns**

new optimizer state after the update.

### 4.6.7 RMSPropMomentum

**class** RMSPropMomentum(*args, **kwargs)*

Wrapper class for the JAX optimizer: **rmsprop_momentum()**

**eval_and_stable_update** *(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)*

Like **eval_and_update()** but when the value of the objective function or the gradients are not finite, we will not update the input \( state \) and will set the objective output to \( nan \).
• **fn** – objective function.
• **state** – current optimizer state.
• **forward_mode_differentiation** – boolean flag indicating whether to use forward
  mode differentiation.

**Returns**
a pair of the output of objective function and the new optimizer state.

`eval_and_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)`

Performs an optimization step for the objective function `fn`. For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as `Minimize`, the update is performed by reevaluating the function multiple times to get optimal parameters.

**Parameters**
- **fn** – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.
- **state** – current optimizer state.
- **forward_mode_differentiation** – boolean flag indicating whether to use forward mode differentiation.

**Returns**
a pair of the output of objective function and the new optimizer state.

`get_params(state: tuple[int, _OptState]) → _Params`
Get current parameter values.

**Parameters**
- **state** – current optimizer state.

**Returns**
collection with current value for parameters.

`init(params: _Params) → tuple[int, _OptState]`
Initialize the optimizer with parameters designated to be optimized.

**Parameters**
- **params** – a collection of numpy arrays.

**Returns**
initial optimizer state.

`update(g: _Params, state: tuple[int, _OptState]) → tuple[int, _OptState]`
Gradient update for the optimizer.

**Parameters**
- **g** – gradient information for parameters.
- **state** – current optimizer state.

**Returns**
new optimizer state after the update.
4.6.8 SGD

class SGD(*args, **kwargs)
    Wrapper class for the JAX optimizer: sgd()

eval_and_stable_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], 
                        forward_mode_differentiation: bool = False)
    Like eval_and_update() but when the value of the objective function or the gradients are not finite, we 
    will not update the input state and will set the objective output to nan.

    Parameters
    • fn – objective function.
    • state – current optimizer state.
    • forward_mode_differentiation – boolean flag indicating whether to use forward 
      mode differentiation.

    Returns
    a pair of the output of objective function and the new optimizer state.

eval_and_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: 
                 bool = False)
    Performs an optimization step for the objective function fn. For most optimizers, the update is performed 
    based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as 
    Minimize, the update is performed by reevaluating the function multiple times to get optimal parameters.

    Parameters
    • fn – an objective function returning a pair where the first item is a scalar loss function to 
      be differentiated and the second item is an auxiliary output.
    • state – current optimizer state.
    • forward_mode_differentiation – boolean flag indicating whether to use forward 
      mode differentiation.

    Returns
    a pair of the output of objective function and the new optimizer state.

get_params(state: tuple[int, _OptState]) → _Params
    Get current parameter values.

    Parameters
    state – current optimizer state.

    Returns
    collection with current value for parameters.

init(params: _Params) → tuple[int, _OptState]
    Initialize the optimizer with parameters designated to be optimized.

    Parameters
    params – a collection of numpy arrays.

    Returns
    initial optimizer state.

update(g: _Params, state: tuple[int, _OptState]) → tuple[int, _OptState]
    Gradient update for the optimizer.
Parameters

- \textbf{g} – gradient information for parameters.
- \textbf{state} – current optimizer state.

Returns

new optimizer state after the update.

### 4.6.9 SM3

```python
class SM3(*args, **kwargs)
    Wraper class for the JAX optimizer: sm3()

eval_and_stable_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState],
                       forward_mode_differentiation: bool = False)
```

Like \texttt{eval_and_update()} but when the value of the objective function or the gradients are not finite, we will not update the input \texttt{state} and will set the objective output to \texttt{nan}.

Parameters

- \textbf{fn} – objective function.
- \textbf{state} – current optimizer state.
- \textbf{forward_mode_differentiation} – boolean flag indicating whether to use forward mode differentiation.

Returns

a pair of the output of objective function and the new optimizer state.

```python
eval_and_update(fn: Callable[[Any], tuple], state: tuple[int, _OptState], forward_mode_differentiation: bool = False)
```

Performs an optimization step for the objective function \textit{fn}. For most optimizers, the update is performed based on the gradient of the objective function w.r.t. the current state. However, for some optimizers such as \texttt{Minimize}, the update is performed by reevaluating the function multiple times to get optimal parameters.

Parameters

- \textbf{fn} – an objective function returning a pair where the first item is a scalar loss function to be differentiated and the second item is an auxiliary output.
- \textbf{state} – current optimizer state.
- \textbf{forward_mode_differentiation} – boolean flag indicating whether to use forward mode differentiation.

Returns

a pair of the output of objective function and the new optimizer state.

```python
get_params(state: tuple[int, _OptState]) → _Params
```

Get current parameter values.

Parameters

- \textbf{state} – current optimizer state.

Returns

collection with current value for parameters.
**NumPyro Documentation**

\[
\text{init}(\text{params:}_\text{Params}) \rightarrow \text{tuple}[\text{int}, _\text{OptState}]
\]

Initialize the optimizer with parameters designated to be optimized.

**Parameters**

- \text{params} – a collection of numpy arrays.

**Returns**

initial optimizer state.

\[
\text{update}(g: _\text{Params}, \text{state: } \text{tuple}[\text{int}, _\text{OptState}]) \rightarrow \text{tuple}[\text{int}, _\text{OptState}]
\]

Gradient update for the optimizer.

**Parameters**

- \text{g} – gradient information for parameters.
- \text{state} – current optimizer state.

**Returns**

new optimizer state after the update.

### 4.6.10 Optax support

\[
\text{optax_to_numpyro}(\text{transformation}) \rightarrow _\text{NumPyroOptim}
\]

This function produces a \text{numpyro.optim._NumPyroOptim} instance from an \text{optax.GradientTransformation} so that it can be used with \text{numpyro.infer.svi.SVI}. It is a lightweight wrapper that recreates the \text{(init_fn, update_fn, get_params_fn)} interface defined by \text{jax.example_libraries.optimizers}.

**Parameters**

- \text{transformation} – An \text{optax.GradientTransformation} instance to wrap.

**Returns**

An instance of \text{numpyro.optim._NumPyroOptim} wrapping the supplied Optax optimizer.

### 4.7 Diagnostics

This provides a small set of utilities in NumPyro that are used to diagnose posterior samples.

#### 4.7.1 Autocorrelation

\[
\text{autocorrelation}(x, \text{axis=0})
\]

Computes the autocorrelation of samples at dimension \text{axis}.

**Parameters**

- \text{x} (\text{numpy.ndarray}) – the input array.
- \text{axis} (\text{int}) – the dimension to calculate autocorrelation.

**Returns**

autocorrelation of \text{x}.

**Return type**

\text{numpy.ndarray}
4.7.2 Autocovariance

**autocovariance**(x, *axis*=0)

Computes the autocovariance of samples at dimension *axis*.

**Parameters**
- x (*numpy.ndarray*) – the input array.
- axis (*int*) – the dimension to calculate autocovariance.

**Returns**
- autocovariance of x.

**Return type**
- numpy.ndarray

4.7.3 Effective Sample Size

**effective_sample_size**(x)

Computes effective sample size of input x, where the first dimension of x is chain dimension and the second dimension of x is draw dimension.

**References:**
1. *Introduction to Markov Chain Monte Carlo*, Charles J. Geyer

**Parameters**
- x (*numpy.ndarray*) – the input array.

**Returns**
- effective sample size of x.

**Return type**
- numpy.ndarray

4.7.4 Gelman Rubin

**gelman_rubin**(x)

Computes R-hat over chains of samples x, where the first dimension of x is chain dimension and the second dimension of x is draw dimension. It is required that x.shape[0] >= 2 and x.shape[1] >= 2.

**Parameters**
- x (*numpy.ndarray*) – the input array.

**Returns**
- R-hat of x.

**Return type**
- numpy.ndarray
4.7.5 Split Gelman Rubin

`split_gelman_rubin(x)`

Computes split R-hat over chains of samples `x`, where the first dimension of `x` is chain dimension and the second dimension of `x` is draw dimension. It is required that `x.shape[1] >= 4`.

**Parameters**

- `x (numpy.ndarray)`: the input array.

**Returns**

- split R-hat of `x`.

**Return type**

numpy.ndarray

4.7.6 HPDI

`hpdi(x, prob=0.9, axis=0)`

Computes “highest posterior density interval” (HPDI) which is the narrowest interval with probability mass `prob`.

**Parameters**

- `x (numpy.ndarray)`: the input array.
- `prob (float)`: the probability mass of samples within the interval.
- `axis (int)`: the dimension to calculate hpdi.

**Returns**

- quantiles of `x` at `(1 - prob) / 2` and `(1 + prob) / 2`.

**Return type**

numpy.ndarray

4.7.7 Summary

`summary(samples, prob=0.9, group_by_chain=True)`

Returns a summary table displaying diagnostics of `samples` from the posterior. The diagnostics displayed are mean, standard deviation, median, the 90% Credibility Interval `hpdi()`, `effective_sample_size()`, and `split_gelman_rubin()`.

**Parameters**

- `samples (dict or numpy.ndarray)`: a collection of input samples with left most dimension is chain dimension and second to left most dimension is draw dimension.
- `prob (float)`: the probability mass of samples within the HPDI interval.
- `group_by_chain (bool)`: If True, each variable in `samples` will be treated as having shape `num_chains x num_samples x sample_shape`. Otherwise, the corresponding shape will be `num_samples x sample_shape` (i.e. without chain dimension).

`print_summary(samples, prob=0.9, group_by_chain=True)`

Prints a summary table displaying diagnostics of `samples` from the posterior. The diagnostics displayed are mean, standard deviation, median, the 90% Credibility Interval `hpdi()`, `effective_sample_size()`, and `split_gelman_rubin()`.
Parameters

- **samples** (*dict* or *numpy.ndarray*) – a collection of input samples with left most dimension is chain dimension and second to left most dimension is draw dimension.
- **prob** (*float*) – the probability mass of samples within the HPDI interval.
- **group_by_chain** (*bool*) – If True, each variable in samples will be treated as having shape *num_chains x num_samples x sample_shape*. Otherwise, the corresponding shape will be *num_samples x sample_shape* (i.e. without chain dimension).

### 4.8 Runtime Utilities

#### 4.8.1 enable_validation

**enable_validation**(is_validate=True)

Enable or disable validation checks in NumPyro. Validation checks provide useful warnings and errors, e.g. NaN checks, validating distribution arguments and support values, etc. which is useful for debugging.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
</table>
| This utility does not take effect under JAX’s JIT compilation or vectorized transformation *jax.vmap()*.

Parameters

- **is_validate** (*bool*) – whether to enable validation checks.

#### 4.8.2 validation_enabled

**validation_enabled**(is_validate=True)

Context manager that is useful when temporarily enabling/disabling validation checks.

Parameters

- **is_validate** (*bool*) – whether to enable validation checks.

#### 4.8.3 enable_x64

**enable_x64**(use_x64=True)

Changes the default array type to use 64 bit precision as in NumPy.

Parameters

- **use_x64** (*bool*) – when True, JAX arrays will use 64 bits by default; else 32 bits.
4.8.4 set_platform

`set_platform(platform=None)`

Changes platform to CPU, GPU, or TPU. This utility only takes effect at the beginning of your program.

Parameters

- `platform (str)` – either 'cpu', 'gpu', or 'tpu'.

4.8.5 set_host_device_count

`set_host_device_count(n)`

By default, XLA considers all CPU cores as one device. This utility tells XLA that there are \( n \) host (CPU) devices available to use. As a consequence, this allows parallel mapping in JAX `jax.pmap()` to work in CPU platform.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>This utility only takes effect at the beginning of your program. Under the hood, this sets the environment variable <code>XLA_FLAGS=xla_force_host_platform_device_count=[num_devices]</code>, where ([num_device]) is the desired number of CPU devices ( n ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Warning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our understanding of the side effects of using the <code>xla_force_host_platform_device_count</code> flag in XLA is incomplete. If you observe some strange phenomenon when using this utility, please let us know through our issue or forum page. More information is available in this JAX issue.</td>
</tr>
</tbody>
</table>

Parameters

- `n (int)` – number of CPU devices to use.

4.9 Inference Utilities

4.9.1 Predictive

`class Predictive(model: Callable, posterior_samples: dict | None = None, *, guide: Callable | None = None, params: dict | None = None, num_samples: int | None = None, return_sites: list[str] | None = None, infer_discrete: bool = False, parallel: bool = False, batch_ndims: int | None = None, exclude_deterministic: bool = True)`

Bases: `object`

This class is used to construct predictive distribution. The predictive distribution is obtained by running model conditioned on latent samples from `posterior_samples`.

<table>
<thead>
<tr>
<th>Warning</th>
</tr>
</thead>
<tbody>
<tr>
<td>The interface for the <code>Predictive</code> class is experimental, and might change in the future.</td>
</tr>
</tbody>
</table>

Parameters
• **model** – Python callable containing Pyro primitives.

• **posterior_samples** (*dict*) – dictionary of samples from the posterior.

• **guide** (*callable*) – optional guide to get posterior samples of sites not present in *posterior_samples*.

• **params** (*dict*) – dictionary of values for param sites of model/guide.

• **num_samples** (*int*) – number of samples

• **return_sites** (*list*) – sites to return; by default only sample sites not present in *posterior_samples* are returned.

• **infer_discrete** (*bool*) – whether or not to sample discrete sites from the posterior, conditioned on observations and other latent values in *posterior_samples*. Under the hood, those sites will be marked with site["infer"]["enumerate"] = "parallel". See how *infer_discrete* works at the Pyro enumeration tutorial. Note that this requires funsor installation.

• **parallel** (*bool*) – whether to predict in parallel using JAX vectorized map `jax.vmap()`. Defaults to False.

• **batch_ndims** – the number of batch dimensions in posterior samples or parameters. If `None` defaults to 0 if guide is set (i.e. not `None`) and 1 otherwise. Usages for batched posterior samples:
  - set `batch_ndims=0` to get prediction for 1 single sample
  - set `batch_ndims=1` to get prediction for *posterior_samples* with shapes `(num_samples x ...)` (same as `batch_ndims=None` with `guide=None`)
  - set `batch_ndims=2` to get prediction for *posterior_samples* with shapes `(num_chains x N x ...)`. Note that if `num_samples` argument is not `None`, its value should be equal to `num_chains x N`.

Usages for batched parameters:
  - set `batch_ndims=0` to get 1 sample from the guide and parameters (same as `batch_ndims=None` with guide)
  - set `batch_ndims=1` to get predictions from a one dimensional batch of the guide and parameters with shapes `(num_samples x batch_size x ...)`

• **exclude_deterministic** – indicates whether to ignore deterministic sites from the posterior samples.

**Returns**

dict of samples from the predictive distribution.

**Example:**

Given a model:

```python
def model(X, y=None):
    ...
    return numpyro.sample("obs", likelihood, obs=y)
```

you can sample from the prior predictive:

```python
predictive = Predictive(model, num_samples=1000)
y_pred = predictive(rng_key, X)["obs"]
```
If you also have posterior samples, you can sample from the posterior predictive:

```python
predictive = Predictive(model, posterior_samples=posterior_samples)
y_pred = predictive(rng_key, X)["obs"]
```

See docstrings for `SVI` and `MCMCKernel` to see example code of this in context.

### 4.9.2 log_density

**log_density**(model, model_args, model_kwargs, params)

(EXPERIMENTAL INTERFACE) Computes log of joint density for the model given latent values `params`.

**Parameters**

- `model` – Python callable containing NumPyro primitives.
- `model_args` (tuple) – args provided to the model.
- `model_kwargs` (dict) – kwargs provided to the model.
- `params` (dict) – dictionary of current parameter values keyed by site name.

**Returns**

log of joint density and a corresponding model trace

### 4.9.3 get_transforms

**get_transforms**(model, model_args, model_kwargs, params)

(EXPERIMENTAL INTERFACE) Retrieve (inverse) transforms via `biject_to()` given a NumPyro model. This function supports ‘param’ sites. NB: Parameter values are only used to retrieve the model trace.

**Parameters**

- `model` – a callable containing NumPyro primitives.
- `model_args` (tuple) – args provided to the model.
- `model_kwargs` (dict) – kwargs provided to the model.
- `params` (dict) – dictionary of values keyed by site names.

**Returns**

dict of transformation keyed by site names.

### 4.9.4 transform_fn

**transform_fn**(transforms, params, invert=False)

(EXPERIMENTAL INTERFACE) Callable that applies a transformation from the `transforms` dict to values in the `params` dict and returns the transformed values keyed on the same names.

**Parameters**

- `params` – Dictionary of arrays keyed by names.
- `invert` – Whether to apply the inverse of the transforms.
Returns
dict of transformed params.

4.9.5 `constrain_fn`

`constrain_fn(model, model_args, model_kwargs, params, return_deterministic=False)`

(EXPERIMENTAL INTERFACE) Gets value at each latent site in `model` given unconstrained parameters `params`. The transforms is used to transform these unconstrained parameters to base values of the corresponding priors in `model`. If a prior is a transformed distribution, the corresponding base value lies in the support of base distribution. Otherwise, the base value lies in the support of the distribution.

Parameters

- `model` – a callable containing NumPyro primitives.
- `model_args` (tuple) – args provided to the model.
- `model_kwargs` (dict) – kwargs provided to the model.
- `params` (dict) – dictionary of unconstrained values keyed by site names.
- `return_deterministic` (bool) – whether to return the value of deterministic sites from the model. Defaults to `False`.

Returns
dict of transformed params.

4.9.6 `unconstrain_fn`

`unconstrain_fn(model, model_args, model_kwargs, params)`

(EXPERIMENTAL INTERFACE) Given a NumPyro model and a dict of parameters, this function applies the right transformation to convert parameter values from constrained space to unconstrained space.

Parameters

- `model` – a callable containing NumPyro primitives.
- `model_args` (tuple) – args provided to the model.
- `model_kwargs` (dict) – kwargs provided to the model.
- `params` (dict) – dictionary of constrained values keyed by site names.

Returns
dict of transformation keyed by site names.

4.9.7 `potential_energy`

`potential_energy(model, model_args, model_kwargs, params, enum=False)`

(EXPERIMENTAL INTERFACE) Computes potential energy of a model given unconstrained params. Under the hood, we will transform these unconstrained parameters to the values belong to the supports of the corresponding priors in `model`.

Parameters

- `model` – a callable containing NumPyro primitives.
- `model_args` (tuple) – args provided to the model.
• **model_kwargs** (*dict*) – kwargs provided to the model.
• **params** (*dict*) – unconstrained parameters of `model`.
• **enum** (*bool*) – whether to enumerate over discrete latent sites.

**Returns**

potential energy given unconstrained parameters.

### 4.9.8 log_likelihood

#### log_likelihood

```
log_likelihood(model, posterior_samples, *args, parallel=False, batch_ndims=1, **kwargs)
```

(EXPERIMENTAL INTERFACE) Returns log likelihood at observation nodes of model, given samples of all latent variables.

**Parameters**

- **model** – Python callable containing Pyro primitives.
- **posterior_samples** (*dict*) – dictionary of samples from the posterior.
- **args** – model arguments.
- **batch_ndims** – the number of batch dimensions in posterior samples. Some usages:
  - set `batch_ndims=0` to get log likelihoods for 1 single sample
  - set `batch_ndims=1` to get log likelihoods for `posterior_samples` with shapes `(num_samples x ...)`
  - set `batch_ndims=2` to get log likelihoods for `posterior_samples` with shapes `(num_chains x num_samples x ...)`
- **kwargs** – model kwargs.

**Returns**

dict of log likelihoods at observation sites.

### 4.9.9 find_valid_initial_params

#### find_valid_initial_params

```
find_valid_initial_params(rng_key, model, *, init_strategy=<function init_to_uniform>, enum=False, model_args=(), model_kwargs=None, prototype_params=None, forward_mode_differentiation=False, validate_grad=True)
```

(EXPERIMENTAL INTERFACE) Given a model with Pyro primitives, returns an initial valid unconstrained value for all the parameters. This function also returns the corresponding potential energy, the gradients, and an `is_valid` flag to say whether the initial parameters are valid. Parameter values are considered valid if the values and the gradients for the log density have finite values.

**Parameters**

- **rng_key** (*jax.random.PRNGKey*) – random number generator seed to sample from the prior. The returned `init_params` will have the batch shape `rng_key.shape[:-1]`.
- **model** – Python callable containing Pyro primitives.
- **init_strategy** (*callable*) – a per-site initialization function.
- **enum** (*bool*) – whether to enumerate over discrete latent sites.
- **model_args** (*tuple*) – args provided to the model.
- `model_kwargs (dict)` – kwargs provided to the model.
- `prototype_params (dict)` – an optional prototype parameters, which is used to define the shape for initial parameters.
- `forward_mode_differentiation (bool)` – whether to use forward-mode differentiation or reverse-mode differentiation. Defaults to False.
- `validate_grad (bool)` – whether to validate gradient of the initial params. Defaults to True.

Returns

tuple of `init_params_info` and `is_valid`, where `init_params_info` is the tuple containing the initial params, their potential energy, and their gradients.

### 4.9.10 Initialization Strategies

**init_to_feasible**

- `init_to_feasible(site=None)`
  
  Initialize to an arbitrary feasible point, ignoring distribution parameters.

**init_to_mean**

- `init_to_mean(site=None)`
  
  Initialize to the prior mean. For priors with no `.mean` property implemented, we defer to the `init_to_median()` strategy.

**init_to_median**

- `init_to_median(site=None, num_samples=15)`
  
  Initialize to the prior median. For priors with no `.sample` method implemented, we defer to the `init_to_uniform()` strategy.

  Parameters

  - `num_samples (int)` – number of prior points to calculate median.

**init_to_sample**

- `init_to_sample(site=None)`
  
  Initialize to a prior sample. For priors with no `.sample` method implemented, we defer to the `init_to_uniform()` strategy.
**init_to_uniform**

```python
init_to_uniform(site=None, radius=2)
```

Initialize to a random point in the area \((-\text{radius}, \text{radius})\) of unconstrained domain.

**Parameters**

- **radius** (float) – specifies the range to draw an initial point in the unconstrained domain.

**init_to_value**

```python
init_to_value(site=None, values={})
```

Initialize to the value specified in `values`. We defer to `init_to_uniform()` strategy for sites which do not appear in `values`.

**Parameters**

- **values** (dict) – dictionary of initial values keyed by site name.

### 4.9.11 Tensor Indexing

**vindex**(tensor, args)

Vectorized advanced indexing with broadcasting semantics.

See also the convenience wrapper `Vindex`.

This is useful for writing indexing code that is compatible with batching and enumeration, especially for selecting mixture components with discrete random variables.

For example suppose `x` is a parameter with `len(x.shape) == 3` and we wish to generalize the expression `x[i, :, j]` from integer `i,j` to tensors `i,j` with batch dims and enum dims (but no event dims). Then we can write the generalize version using `Vindex`

```python
xij = Vindex(x)[i, :, j]
```

```python
batch_shape = broadcast_shape(i.shape, j.shape)
event_shape = (x.size(1),)
assert xij.shape == batch_shape + event_shape
```

To handle the case when `x` may also contain batch dimensions (e.g. if `x` was sampled in a plated context as when using vectorized particles), `vindex()` uses the special convention that `Ellipsis` denotes batch dimensions (hence `...` can appear only on the left, never in the middle or in the right). Suppose `x` has event dim 3. Then we can write:

```python
old_batch_shape = x.shape[:-3]
old_event_shape = x.shape[-3:]

xij = Vindex(x)[..., i, :, j]  # The ... denotes unknown batch shape.
```

```python
new_batch_shape = broadcast_shape(old_batch_shape, i.shape, j.shape)
new_event_shape = (x.size(1),)
assert xij.shape == new_batch_shape + new_event_shape
```

Note that this special handling of `Ellipsis` differs from the NEP [1].

Formally, this function assumes:
1. Each arg is either `Ellipsis`, `slice(None)`, an integer, or a batched integer tensor (i.e. with empty event shape). This function does not support Nontrivial slices or boolean tensor masks. `Ellipsis` can only appear on the left as `args[0]`.

2. If `args[0]` is not `Ellipsis` then `tensor` is not batched, and its event dim is equal to `len(args)`.

3. If `args[0]` is `Ellipsis` then `tensor` is batched and its event dim is equal to `len(args[1:])`. Dims of `tensor` to the left of the event dims are considered batch dims and will be broadcasted with dims of `tensor` args.

Note that if none of the args is a tensor with `len(shape) > 0`, then this function behaves like standard indexing:

```python
if not any(isinstance(a, jnp.ndarray) and len(a.shape) > 0 for a in args):
    assert Vindex(x)[args] == x[args]
```

References

[1] https://www.numpy.org/neps/nep-0021-advanced-indexing.html introduces `vindex` as a helper for vectorized indexing. This implementation is similar to the proposed notation `x.vindex[]` except for slightly different handling of `Ellipsis`.

Parameters

- `tensor (jnp.ndarray)` – A tensor to be indexed.
- `args (tuple)` – An index, as args to `__getitem__`.

Returns

A nonstandard interpretation of `tensor[args]`.

Return type

`jnp.ndarray`

```python
class Vindex(tensor)
    Bases: object

    Convenience wrapper around `vindex()`.

    The following are equivalent:
    ```
    Vindex(x)[..., i, j, :]
    vindex(x, (Ellipsis, i, j, slice(None)))
    ```

    Parameters

    - `tensor (jnp.ndarray)` – A tensor to be indexed.

    Returns

    An object with a special `__getitem__()` method.
```
4.9.12 Model Inspection

get_dependencies

get_dependencies(model: Callable, model_args: tuple | None = None, model_kwargs: dict | None = None) \rightarrow dict[str, object]

Infers dependency structure about a conditioned model.

This returns a nested dictionary with structure like:

```python
{
    "prior_dependencies": {
        "variable1": {"variable1": set(),
                      "variable2": set()},
        "variable2": {"variable1": set()}
    },
    "posterior_dependencies": {
        "variable1": {"variable1": {"plate1"}, "variable2": set(),
                      "variable3": set()}
    }
}
```

where

- **prior_dependencies** is a dict mapping downstream latent and observed variables to dictionaries mapping upstream latent variables on which they depend to sets of plates inducing full dependencies. That is, included plates introduce quadratically many dependencies as in complete-bipartite graphs, whereas excluded plates introduce only linearly many dependencies as in independent sets of parallel edges. Prior dependencies follow the original model order.

- **posterior_dependencies** is a similar dict, but mapping latent variables to the latent or observed sits on which they depend in the posterior. Posterior dependencies are reversed from the model order.

Dependencies elide `numpyro.deterministic` sites and `numpyro.sample(..., Delta(...))` sites.

Examples

Here is a simple example with no plates. We see every node depends on itself, and only the latent variables appear in the posterior:

```python
def model_1():
    a = numpyro.sample("a", dist.Normal(0, 1))
    numpyro.sample("b", dist.Normal(a, 1), obs=0.0)

assert get_dependencies(model_1) == {
    "prior_dependencies": {
        "a": {"a": set(), "b": set()},
        "b": {"a": set()}
    },
    "posterior_dependencies": {
        "a": {"a": set(), "b": set()},
        "b": {"a": set()}
    }
}
```

Here is an example where two variables a and b start out conditionally independent in the prior, but become conditionally dependent in the posterior do the so-called collider variable c on which they both depend. This is called “moralization” in the graphical model literature:
def model_2():
    a = numpyro.sample("a", dist.Normal(0, 1))
    b = numpyro.sample("b", dist.LogNormal(0, 1))
    c = numpyro.sample("c", dist.Normal(a, b))
    numpyro.sample("d", dist.Normal(c, 1), obs=0.)

assert get_dependencies(model_2) == {
    "prior_dependencies": {
        "a": {"a": set()},
        "b": {"b": set()},
        "c": {"a": set(), "b": set(), "c": set()},
        "d": {"c": set(), "d": set()},
    },
    "posterior_dependencies": {
        "a": {"a": set(), "b": set(), "c": set()},
        "b": {"b": set(), "c": set()},
        "c": {"c": set(), "d": set()},
    },
}

Dependencies can be more complex in the presence of plates. So far all the dict values have been empty sets of plates, but in the following posterior we see that c depends on itself across the plate p. This means that, among the elements of c, e.g. c[0] depends on c[1] (this is why we explicitly allow variables to depend on themselves):

def model_3():
    with numpyro.plate("p", 5):
        a = numpyro.sample("a", dist.Normal(0, 1))
        numpyro.sample("b", dist.Normal(a.sum(), 1), obs=0.0)

assert get_dependencies(model_3) == {
    "prior_dependencies": {
        "a": {"a": set()},
        "b": {"a": set(), "b": set()},
    },
    "posterior Dependencies": {
        "a": {"a": set(), "p": set()},
        "b": {"b": set(), "c": set()},
    },
}

    “Faithful inversion of generative models for effective amortized inference”
    https://dl.acm.org/doi/10.5555/3327144.3327229

Parameters

- model (callable) – A model.
- model_args (tuple) – Optional tuple of model args.
- model_kwarg (dict) – Optional dict of model kwargs.

Returns

A dictionary of metadata (see above).
get_model_relations

get_model_relations(model, model_args=None, model_kwargs=None)

Infer relations of RVs and plates from given model and optionally data. See https://github.com/pyro-ppl/numpyro/issues/949 for more details.

This returns a dictionary with keys:

- “sample_sample” map each downstream sample site to a list of the upstream sample sites on which it depend;
- “sample_param” map each downstream sample site to a list of the upstream param sites on which it depend;
- “sample_dist” maps each sample site to the name of the distribution at that site;
- “param_constraint” maps each param site to the name of the constraints at that site;
- “plate_sample” maps each plate name to a lists of the sample sites within that plate; and
- “observe” is a list of observed sample sites.

For example for the model:

```python
def model(data):
    m = numpyro.sample('m', dist.Normal(0, 1))
    sd = numpyro.sample('sd', dist.LogNormal(m, 1))
    with numpyro.plate('N', len(data)):
        numpyro.sample('obs', dist.Normal(m, sd), obs=data)
```

the relation is:

```json
{'sample_sample': {'m': [], 'sd': ['m'], 'obs': ['m', 'sd']},
 'sample_dist': {'m': 'Normal', 'sd': 'LogNormal', 'obs': 'Normal'},
 'plate_sample': {'N': ['obs']},
 'observed': ['obs']}
```

Parameters

- model (callable) – A model to inspect.
- model_args – Optional tuple of model args.
- model_kwargs – Optional dict of model kwargs.

Return type
dict
4.10 Visualization Utilities

4.10.1 render_model

```python
def render_model(model, model_args=None, model_kwargs=None, filename=None, render_distributions=False, render_params=False):
    Wrap all functions needed to automatically render a model.
```

**Warning**

This utility does not support the `scan()` primitive. If you want to render a time-series model, you can try to rewrite the code using Python for loop.

**Parameters**

- `model` – Model to render.
- `model_args` – Positional arguments to pass to the model.
- `model_kwargs` – Keyword arguments to pass to the model.
- `filename` – File to save rendered model in.
- `render_distributions` – Whether to include RV distribution annotations in the plot.
- `render_params` – Whether to show params in the plot.

4.10.2 Trace Inspection

```python
def format_shapes(trace, *, compute_log_prob=False, title='Trace Shapes:', last_site=None):
    Given the trace of a function, returns a string showing a table of the shapes of all sites in the trace.
    Use trace handler (or funsor trace handler for enumeration) to produce the trace.
```

**Parameters**

- `trace` – The model trace to format.
- `compute_log_prob` – Compute log probabilities and display the shapes in the table. Accepts True / False or a function which when given a dictionary containing site-level metadata returns whether the log probability should be calculated and included in the table.
- `title` – Title for the table of shapes.
- `last_site` – Name of a site in the model. If supplied, subsequent sites are not displayed in the table.

**Usage:**

```python
def model(*args, **kwargs):
    ...

with numpyro.handlers.seed(rng_seed=1):
    trace = numpyro.handlers.trace(model).get_trace(*args, **kwargs)
print(numpyro.util.format_shapes(trace))
```
This provides a small set of effect handlers in NumPyro that are modeled after Pyro’s poutine module. For a tutorial on effect handlers more generally, readers are encouraged to read Poutine: A Guide to Programming with Effect Handlers in Pyro. These simple effect handlers can be composed together or new ones added to enable implementation of custom inference utilities and algorithms.

Example

As an example, we are using seed, trace and substitute handlers to define the `log_likelihood` function below. We first create a logistic regression model and sample from the posterior distribution over the regression parameters using MCMC(). The `log_likelihood` function uses effect handlers to run the model by substituting sample sites with values from the posterior distribution and computes the log density for a single data point. The `log_predictive_density` function computes the log likelihood for each draw from the joint posterior and aggregates the results for all the data points, but does so by using JAX’s auto-vectorize transform called `vmap` so that we do not need to loop over all the data points.

```python
>>> import jax.numpy as jnp
>>> from jax import random, vmap
>>> from jax.scipy.special import logsumexp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro import handlers
>>> from numpyro.infer import MCMC, NUTS

>>> N, D = 3000, 3
>>> def logistic_regression(data, labels):
...     coefs = numpyro.sample('coefs', dist.Normal(jnp.zeros(D), jnp.ones(D)))
...     intercept = numpyro.sample('intercept', dist.Normal(0., 10.))
...     logits = jnp.sum(coefs * data + intercept, axis=-1)
...     return numpyro.sample('obs', dist.Bernoulli(logits=logits), obs=labels)

>>> data = random.normal(random.PRNGKey(0), (N, D))
>>> true_coefs = jnp.arange(1., D + 1.)
>>> logits = jnp.sum(true_coefs * data, axis=-1)
>>> labels = dist.Bernoulli(logits=logits).sample(random.PRNGKey(1))

>>> num_warmup, num_samples = 1000, 1000
>>> mcmc = MCMC(NUTS(model=logistic_regression), num_warmup=num_warmup, num_samples=num_samples)
>>> mcmc.run(random.PRNGKey(2), data, labels)
sample: 100%|████████████████████████████| 1000/1000 [00:00<00:00, 1252.39it/s, 1 steps of 1.0000, size 5.83e-01. acc. prob=0.85]
```

(continues on next page)
>>> mcmc.print_summary()

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>sd</th>
<th>5.5%</th>
<th>94.5%</th>
<th>n_eff</th>
<th>Rhat</th>
</tr>
</thead>
<tbody>
<tr>
<td>coefs[0]</td>
<td>0.96</td>
<td>0.07</td>
<td>0.85</td>
<td>1.07</td>
<td>455.35</td>
<td>1.01</td>
</tr>
<tr>
<td>coefs[1]</td>
<td>2.05</td>
<td>0.09</td>
<td>1.91</td>
<td>2.20</td>
<td>332.00</td>
<td>1.01</td>
</tr>
<tr>
<td>coefs[2]</td>
<td>3.18</td>
<td>0.13</td>
<td>2.96</td>
<td>3.37</td>
<td>320.27</td>
<td>1.00</td>
</tr>
<tr>
<td>intercept</td>
<td>-0.03</td>
<td>0.02</td>
<td>-0.06</td>
<td>0.00</td>
<td>402.53</td>
<td>1.00</td>
</tr>
</tbody>
</table>

```python
>>> def log_likelihood(rng_key, params, model, *args, **kwargs):
...     model = handlers.substitute(handlers.seed(model, rng_key), params)
...     model_trace = handlers.trace(model).get_trace(*args, **kwargs)
...     obs_node = model_trace['obs']
...     return obs_node['fn'].log_prob(obs_node['value'])
```

```python
>>> def log_predictive_density(rng_key, params, model, *args, **kwargs):
...     n = list(params.values())[0].shape[0]
...     log_lk_fn = vmap(lambda rng_key, params: log_likelihood(rng_key, params, model, *args, **kwargs))
...     log_lk_vals = log_lk_fn(random.split(rng_key, n), params)
...     return jnp.sum(logsumexp(log_lk_vals, 0) - jnp.log(n))
```

```python
>>> print(log_predictive_density(random.PRNGKey(2), mcmc.get_samples(), logistic_regression, data, labels))
-874.89813
```

## 5.1 block

class block(fn=None, hide_fn=None, hide=None, expose_types=None, expose=None)

Bases: Messenger

Given a callable `fn`, return another callable that selectively hides primitive sites from other effect handlers on the stack. In the absence of parameters, all primitive sites are blocked. `hide_fn` takes precedence over `hide`, which has higher priority than `expose_types` followed by `expose`. Only the parameter with the precedence is considered.

Parameters

- **fn** (callable) – Python callable with NumPyro primitives.
- **hide_fn** (callable) – function which when given a dictionary containing site-level metadata returns whether it should be blocked.
- **hide** (list) – list of site names to hide.
- **expose_types** (list) – list of site types to expose, e.g. `['param']`.
- **expose** (list) – list of site names to expose.

Returns

Python callable with NumPyro primitives.

Example:
```python
>>> from jax import random
>>> import numpyro
>>> from numpyro.handlers import block, seed, trace
>>> import numpyro.distributions as dist

>>> def model():
...     a = numpyro.sample('a', dist.Normal(0., 1.))
...     return numpyro.sample('b', dist.Normal(a, 1.))

>>> model = seed(model, random.PRNGKey(0))
>>> block_all = block(model)
>>> block_a = block(model, lambda site: site['name'] == 'a')
>>> trace_block_all = trace(block_all).get_trace()
>>> assert not {'a', 'b'}.intersection(trace_block_all.keys())
>>> trace_block_a = trace(block_a).get_trace()
>>> assert 'a' not in trace_block_a
>>> assert 'b' in trace_block_a
```

### 5.2 collapse

class collapse(*args, **kwargs)

Bases: trace

EXPERIMENTAL Collapses all sites in the context by lazily sampling and attempting to use conjugacy relations. If no conjugacy is known this will fail. Code using the results of sample sites must be written to accept Funsors rather than Tensors. This requires funsor to be installed.

process_message(msg)

### 5.3 condition

class condition(fn=None, data=None, condition_fn=None)

Bases: Messenger

Conditions unobserved sample sites to values from data or condition_fn. Similar to substitute except that it only affects sample sites and changes the is_observed property to True.

Parameters

- **fn** – Python callable with NumPyro primitives.
- **data (dict)** – dictionary of numpy.ndarray values keyed by site names.
- **condition_fn** – callable that takes in a site dict and returns a numpy array or None (in which case the handler has no side effect).

Example:
>>> from jax import random
>>> import numpyro
>>> from numpyro.handlers import condition, seed, substitute, trace
>>> import numpyro.distributions as dist

>>> def model():
...    numpyro.sample('a', dist.Normal(0., 1.))

>>> model = seed(model, random.PRNGKey(0))
>>> exec_trace = trace(condition(model, {'a': -1})).get_trace()
>>> assert exec_trace['a']['value'] == -1
>>> assert exec_trace['a']['is_observed']

5.4 do

class do(fn=None, data=None)

Bases: Messenger

Given a stochastic function with some sample statements and a dictionary of values at names, set the return values of those sites equal to the values as if they were hard-coded to those values and introduce fresh sample sites with the same names whose values do not propagate.

Composes freely with condition() to represent counterfactual distributions over potential outcomes. See Single World Intervention Graphs [1] for additional details and theory.

This is equivalent to replacing \( z = \text{numpyro.sample("z", ...)} \) with \( z = 1. \) and introducing a fresh sample site \( \text{numpyro.sample("z", ...)} \) whose value is not used elsewhere.

References:

1. Single World Intervention Graphs: A Primer, Thomas Richardson, James Robins

Parameters

- fn – a stochastic function (callable containing Pyro primitive calls)
- data – a dict mapping sample site names to interventions

Example:

```python
>>> import jax.numpy as jnp
>>> import numpyro
>>> from numpyro.handlers import do, trace, seed
>>> import numpyro.distributions as dist
>>> def model(x):
...    s = numpyro.sample("s", dist.LogNormal())
...    z = numpyro.sample("z", dist.Normal(x, s))
...    return z ** 2
>>> intervened_model = handlers.do(model, data={'z': 1.})
>>> with trace() as exec_trace:
...    z_square = seed(intervened_model, 0)(1)
>>> assert exec_trace['z']['value'] != 1.
```
>>> assert not exec_trace['z']['is_observed']
>>> assert not exec_trace['z'].get('stop', None)
>>> assert z_square == 1

process_message(msg)

## 5.5 infer_config

class `infer_config`(fn=None, config_fn=None)

Bases: `Messenger`

Given a callable `fn` that contains NumPyro primitive calls and a callable `config_fn` taking a trace site and returning a dictionary, updates the value of the infer kwarg at a sample site to `config_fn(site)`.

**Parameters**

- `fn` – a stochastic function (callable containing NumPyro primitive calls)
- `config_fn` – a callable taking a site and returning an infer dict

process_message(msg)

## 5.6 lift

class `lift`(fn=None, prior=None)

Bases: `Messenger`

Given a stochastic function with `param` calls and a prior distribution, create a stochastic function where all `param` calls are replaced by sampling from prior. Prior should be a distribution or a dict of names to distributions.

Consider the following NumPyro program:

```python
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.handlers import lift

>>> def model(x):
...     s = numpyro.param("s", 0.5)
...     z = numpyro.sample("z", dist.Normal(x, s))
...     return z ** 2
>>> lifted_model = lift(model, prior={"s": dist.Exponential(0.3)})
```

`lift` makes `param` statements behave like `sample` statements using the distributions in `prior`. In this example, site `s` will now behave as if it was replaced with `s = numpyro.sample("s", dist.Exponential(0.3)).`

**Parameters**

- `fn` – function whose parameters will be lifted to random values
- `prior` – prior function in the form of a Distribution or a dict of Distributions

process_message(msg)
5.7 mask

class mask(fn=None, mask=True)
Bases: Messenger
This messenger masks out some of the sample statements elementwise.

Parameters
mask – a boolean or a boolean-valued array for masking elementwise log probability of sample sites (True includes a site, False excludes a site).

process_message(msg)

5.8 reparam

class reparam(fn=None, config=None)
Bases: Messenger
Reparametrizes each affected sample site into one or more auxiliary sample sites followed by a deterministic transformation [1].

To specify reparameterizers, pass a config dict or callable to the constructor. See the numpyro.infer.reparam module for available reparameterizers.

Note some reparameterizers can examine the *args,**kwargs inputs of functions they affect; these reparameterizers require using handlers.reparam as a decorator rather than as a context manager.


Parameters
config (dict or callable) – Configuration, either a dict mapping site name to Reparam, or a function mapping site to Reparam or None.

process_message(msg)

5.9 replay

class replay(fn=None, trace=None)
Bases: Messenger
Given a callable fn and an execution trace trace, return a callable which substitutes sample calls in fn with values from the corresponding site names in trace.

Parameters
• fn – Python callable with NumPyro primitives.
• trace – an OrderedDict containing execution metadata.

Example:
>>> from jax import random
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.handlers import replay, seed, trace

>>> def model():
...     numpyro.sample('a', dist.Normal(0., 1.))

>>> exec_trace = trace(seed(model, random.PRNGKey(0))).get_trace()
>>> print(exec_trace['a']['value'])
-0.20584235

>>> replayed_trace = trace(replay(model, exec_trace)).get_trace()
>>> print(exec_trace['a']['value'])
-0.20584235
>>> assert replayed_trace['a']['value'] == exec_trace['a']['value']

process_message(msg)

5.10 scale

class scale(fn=None, scale=1.0)
Bases: Messenger

This messenger rescales the log probability score.
This is typically used for data subsampling or for stratified sampling of data (e.g. in fraud detection where negatives vastly outnumber positives).

Parameters

scale (float or numpy.ndarray) – a positive scaling factor that is broadcastable to the shape of log probability.

process_message(msg)

5.11 scope

class scope(fn=None, prefix='', divider='\t', *, hide_types=None)
Bases: Messenger

This handler prepend a prefix followed by a divider to the name of sample sites.

Example:

>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.handlers import scope, seed, trace

>>> def model():
...     with scope(prefix="a"):
...         with scope(prefix="b", divider="."):  
...             return numpyro.sample("x", dist.Bernoulli(0.5))

(continues on next page)
...  
```python
>>> assert "a/b.x" in trace(seed(model, 0)).get_trace()
```

**Parameters**

- `fn` – Python callable with NumPyro primitives.
- `prefix` (*str*) – a string to prepend to sample names
- `divider` (*str*) – a string to join the prefix and sample name; default to ‘/’
- `hide_types` (*list*) – an optional list of side types to skip renaming.

`process_message(msg)`

## 5.12 seed

**class** `seed(fn=None, rng_seed=None, hide_types=None)`

Bases: `Messenger`

JAX uses a functional pseudo random number generator that requires passing in a seed `PRNGKey()` to every stochastic function. The `seed` handler allows us to initially seed a stochastic function with a `PRNGKey()`. Every call to the `sample()` primitive inside the function results in a splitting of this initial seed so that we use a fresh seed for each subsequent call without having to explicitly pass in a `PRNGKey` to each `sample` call.

**Parameters**

- `fn` – Python callable with NumPyro primitives.
- `rng_seed` (*int, jnp.ndarray scalar, or jax.random.PRNGKey*) – a random number generator seed.
- `hide_types` (*list*) – an optional list of side types to skip seeding, e.g. ['plate'].

---

**Note**

Unlike in Pyro, `numpyro.sample` primitive cannot be used without wrapping it in seed handler since there is no global random state. As such, users need to use `seed` as a contextmanager to generate samples from distributions or as a decorator for their model callable (See below).

---

**Note**

The seed handler has a mutable attribute `rng_key` which keeps changing after each `sample` call. Hence an instance of this class (e.g. `seed(model, rng_seed=0)`) might create tracer-leaks when jitted. A solution is to close the instance in a function, e.g., `seeded_model = lambda *args: seed(model, rng_seed=0)(*args)`. This `seeded_model` can be jitted.

**Example:**
```python
>>> from jax import random
>>> import numpyro
>>> import numpyro.handlers
>>> import numpyro.distributions as dist

>>> # as context manager
>>> with handlers.seed(rng_seed=1):
...   x = numpyro.sample('x', dist.Normal(0., 1.))

>>> def model():
...   return numpyro.sample('y', dist.Normal(0., 1.))

>>> # as function decorator (/modifier)

>>> y = handlers.seed(model, rng_seed=1)()

>>> assert x == y

process_message(msg)
```

### 5.13 substitute

**class substitute** *(fn=None, data=None, substitute_fn=None)*

**Bases:** Messenger

Given a callable `fn` and a dict `data` keyed by site names (alternatively, a callable `substitute_fn`), return a callable which substitutes all primitive calls in `fn` with values from `data` whose key matches the site name. If the site name is not present in `data`, there is no side effect.

If a `substitute_fn` is provided, then the value at the site is replaced by the value returned from the call to `substitute_fn` for the given site.

**Parameters**

- `fn` – Python callable with NumPyro primitives.
- `data` *(dict)* – dictionary of `numpy.ndarray` values keyed by site names.
- `substitute_fn` – callable that takes in a site dict and returns a numpy array or `None` (in which case the handler has no side effect).

**Example:**

```python
>>> from jax import random
>>> import numpyro
>>> import numpyro.handlers import seed, substitute, trace
>>> import numpyro.distributions as dist

>>> def model():
...     (continues on next page)
```

---

5.13. substitute 241
5.14 trace

class trace(fn=None)

Bases: Messenger

Returns a handler that records the inputs and outputs at primitive calls inside fn.

Example:

```python
>>> from jax import random
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.handlers import seed, trace
>>> import pprint as pp
>>> def model():
...     numpyro.sample('a', dist.Normal(0., 1.))
>>> exec_trace = trace(seed(model, random.PRNGKey(0))).get_trace()
>>> pp.pprint(exec_trace)
OrderedDict([('a',
               {'args': (),
                'fn': <numpyro.distributions.continuous.Normal object at
                      0x7f9e689b1eb8>,
                'is_observed': False,
                'kwargs': {'rng_key': Array([0, 0], dtype=uint32)},
                'name': 'a',
                'type': 'sample',
                'value': Array(-0.20584235, dtype=float32)})])
```

postprocess_message(msg)

get_trace(*args, **kwargs)

Run the wrapped callable and return the recorded trace.

Parameters

- *args – arguments to the callable.
- **kwargs – keyword arguments to the callable.

Returns

OrderedDict containing the execution trace.
CHAPTER

SIX

CONTRIBUTED CODE

6.1 Nested Sampling
Nested Sampling is a non-MCMC approach that works for arbitrary probability models, and is particularly well suited
to complex posteriors:
• NestedSampler offers a wrapper for jaxns. See JAXNS’s readthedocs for examples and Nested Sampling for
Gaussian Shells example for how to apply the sampler on numpyro models. Can handle arbitrary models, including ones with discrete RVs, and non-invertible transformations.
class NestedSampler(model, *, constructor_kwargs=None, termination_kwargs=None)
Bases: object
(EXPERIMENTAL) A wrapper for jaxns , a nested sampling package based on JAX.
See reference [1] for details on the meaning of each parameter. Please consider citing this reference if you use
the nested sampler in your research.
á Note
To enumerate over a discrete latent variable, you can add the keyword infer={“enumerate”: “parallel”} to
the corresponding sample statement.

á Note
To improve the performance, please consider enabling x64 mode at the beginning of your NumPyro program
numpyro.enable_x64().
References
abs/2012.15286)
Parameters
• model (callable) – a call with NumPyro primitives
• constructor_kwargs (dict) – additional keyword arguments to construct an upstream
jaxns.NestedSampler instance.
• termination_kwargs (dict) – keyword arguments to terminate the sampler. Please refer
to the upstream jaxns.NestedSampler.__call__() method.

243


Example

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.contrib.nested_sampling import NestedSampler

true_coefs = jnp.array([1., 2., 3.])
data = random.normal(random.PRNGKey(0), (2000, 3))
labels = dist.Bernoulli(logits=(true_coefs * data).sum(-1)).sample(random.PRNGKey(1))

def model(data, labels):
    coefs = numpyro.sample('coefs', dist.Normal(0, 1).expand([3]))
    intercept = numpyro.sample('intercept', dist.Normal(0., 10.))
    return numpyro.sample('y', dist.Bernoulli(logits=(coefs * data + intercept).sum(-1)),
                           obs=labels)

ns = NestedSampler(model)
ns.run(random.PRNGKey(2), data, labels)
samples = ns.get_samples(random.PRNGKey(3), num_samples=1000)
assert jnp.mean(jnp.abs(samples['intercept'])) < 0.05
print(jnp.mean(samples['coefs'], axis=0))
[0.93661342 1.95034876 2.86123884]
```

run(rng_key, *args, **kwargs)
Run the nested samplers and collect weighted samples.

Parameters

- **rng_key** (`random.PRNGKey`) – Random number generator key to be used for the sampling.
- **args** – The arguments needed by the model.
- **kwargs** – The keyword arguments needed by the model.

get_samples(rng_key, num_samples)
Draws samples from the weighted samples collected from the run.

Parameters

- **rng_key** (`random.PRNGKey`) – Random number generator key to be used to draw samples.
- **num_samples** (`int`) – The number of samples.

Returns

da dict of posterior samples

get_weighted_samples()
Gets weighted samples and their corresponding log weights.

print_summary()
Print summary of the result. This is a wrapper of jaxns.utils.summary().

diagnostics()
Plot diagnostics of the result. This is a wrapper of jaxns.plotting.plot_diagnostics() and jaxns.plotting.plot_cornerplot().
6.2 Stein Variational Inference

Stein Variational Inference (SteinVI) is a family of VI techniques for approximate Bayesian inference based on Stein’s method (see [1] for an overview). It is gaining popularity as it combines the scalability of traditional VI with the flexibility of non-parametric particle-based methods.

Stein variational gradient descent (SVGD) [2] is a recent SteinVI technique which uses iteratively moves a set of particles \( \{ z_i \}_{i=1}^N \) to approximate a distribution \( p(z) \). SVGD is well suited for capturing correlations between latent variables as a particle-based method. The technique preserves the scalability of traditional VI approaches while offering the flexibility and modeling scope of methods such as Markov chain Monte Carlo (MCMC). SVGD is good at capturing multi-modality [3][4].

`numpyro.contrib.einstein` is a framework for particle-based inference using the Stein mixture algorithm. The framework works on Stein mixtures, a restricted mixture of guide programs parameterized by Stein particles. Similarly to how SVGD works, Stein mixtures can approximate model posteriors by moving the Stein particles according to the Stein forces. Because the Stein particles parameterize a guide, they capture a neighborhood rather than a single point.

`numpyro.contrib.einstein` mimics the interface from `numpyro.infer.svi`, so trying SteinVI requires minimal change to the code for existing models inferred with SVI. For primary usage, see the Bayesian neural network example.

The framework currently supports several kernels, including:

- `RBFKernel`
- `LinearKernel`
- `RandomFeatureKernel`
- `MixtureKernel`
- `GraphicalKernel`
- `ProbabilityProductKernel`

For example, usage see:

- The Bayesian neural network example

References


6.2.1 SteinVI Interface

```python
class SteinVI(model: Callable, guide: Callable, optim: _NumPyroOptim, kernel_fn: SteinKernel, num_stein_particles: int = 10, num_elbo_particles: int = 10, loss_temperature: float = 1.0, repulsion_temperature: float = 1.0, non_mixture_guide_params_fn: Callable[[str], bool] = <function SteinVI.<lambda>>, enum=True, **static_kwargs)
```

Variational inference with Stein mixtures.

Example:

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.distributions import constraints
>>> from numpyro.contrib.einstein import MixtureGuidePredictive, SteinVI, RBFKernel

>>> def model(data):
...     f = numpyro.sample("latent_fairness", dist.Beta(10, 10))
...     with numpyro.plate("N", data.shape[0] if data is not None else 10):
...         numpyro.sample("obs", dist.Bernoulli(f), obs=data)

>>> def guide(data):
...     alpha_q = numpyro.param("alpha_q", 15., constraint=constraints.positive)
...     beta_q = numpyro.param("beta_q", lambda rng_key: random.exponential(rng_key), constraint=constraints.positive)
...     numpyro.sample("latent_fairness", dist.Beta(alpha_q, beta_q))

>>> data = jnp.concatenate([jnp.ones(6), jnp.zeros(4)])
>>> optimizer = numpyro.optim.Adam(step_size=0.0005)
>>> stein = SteinVI(model, guide, optimizer, kernel_fn=RBFKernel())
>>> stein_result = stein.run(random.PRNGKey(0), 2000, data)
>>> params = stein_result.params
>>> # use guide to make predictive
>>> predictive = MixtureGuidePredictive(model, guide, params, num_samples=1000, guide_sites=stein.guide_sites)
>>> samples = predictive(random.PRNGKey(1), data=None)
```

Parameters

- **model** (Callable) – Python callable with Pyro primitives for the model.
- **guide** – Python callable with Pyro primitives for the guide (recognition network).
- **optim** (_NumPyroOptim) – An instance of _NumPyroOptim.
- **kernel_fn** (SteinKernel) – Function that produces a logarithm of the statistical kernel to use with Stein mixture inference.
- **num_stein_particles** – Number of particles (i.e., mixture components) in the Stein mixture.
- **num_elbo_particles** – Number of Monte Carlo draws used to approximate the attractive force gradient. (More particles give better gradient approximations)
• **loss_temperature** (*Float*) – Scaling factor of the attractive force.

• **repulsion_temperature** (*Float*) – Scaling factor of the repulsive force (Non-linear Stein)

• **non_mixture_guide_param_fn** (*Callable*) – predicate on names of parameters in guide which should be optimized classically without Stein (E.g. parameters for large normal networks or other transformation)

• **static_kwargs** – Static keyword arguments for the model / guide, i.e. arguments that remain constant during inference.

### 6.2.2 SteinVI Kernels

**class RBFKernel**(mode='norm', matrix_mode='norm_diag', bandwidth_factor: ~collections.abc.Callable[[float], float] = <function RBFKernel.<lambda>>)

Calculates the Gaussian RBF kernel function, from [1], $k(x, y) = \exp(\frac{1}{h}||x - y||^2)$, where the bandwidth $h$ is computed using the median heuristic $h = \frac{1}{\log(n)}\text{med}(||x - y||)$.

**References:**
1. *Stein Variational Gradient Descent* by Liu and Wang

**Parameters**

- **mode** (*str*) – Either ‘norm’ (default) specifying to take the norm of each particle, ‘vector’ to return a component-wise kernel or ‘matrix’ to return a matrix-valued kernel

- **matrix_mode** (*str*) – Either ‘norm_diag’ (default) for diagonal filled with the norm kernel or ‘vector_diag’ for diagonal of vector-valued kernel

- **bandwidth_factor** – A multiplier to the bandwidth based on data size $n$ (default $1/\log(n)$)

**class LinearKernel**(mode='norm')

Calculates the linear kernel $k(x, y) = x \cdot y + 1$ from [1].

**References:**
1. *Stein Variational Gradient Descent as Moment Matching* by Liu and Wang

**class RandomFeatureKernel**(mode='norm', bandwidth_subset=None, bandwidth_factor: ~collections.abc.Callable[[float], float] = <function RandomFeatureKernel.<lambda>>)

Calculates the random kernel $k(x, y) = 1/m \sum_{l=1}^{m} \phi(x, w_l)\phi(y, w_l)$ from [1].

**References:**
1. *Stein Variational Gradient Descent as Moment Matching* by Liu and Wang

**Parameters**

- **bandwidth_subset** – How many particles should be used to calculate the bandwidth? (default None, meaning all particles)

- **random_indices** – The set of indices which to do random feature expansion on. (default None, meaning all indices)

- **bandwidth_factor** – A multiplier to the bandwidth based on data size $n$ (default $1/\log(n)$)
class MixtureKernel(ws: list[float], kernel_fns: list[SteinKernel], mode='norm')
Calculates a mixture of multiple kernels \( k(x, y) = \sum_i w_i k_i(x, y) \)

References:
1. Stein Variational Gradient Descent as Moment Matching by Liu and Wang

Parameters
- ws – Weight of each kernel in the mixture
- kernel_fns – Different kernel functions to mix together

Calculates graphical kernel \( k(x, y) = \text{diag}(K_l(x_l, y_l)) \) for local kernels \( K_l \) from [1][2].

References:
2. Stein Variational Gradient Descent with Matrix-Valued Kernels by Wang, Tang, Bajaj, and Liu

Parameters
- local_kernel_fns – A mapping between parameters and a choice of kernel function for that parameter (default to default_kernel_fn for each parameter)
- default_kernel_fn – The default choice of kernel function when none is specified for a particular parameter

class ProbabilityProductKernel(guide, scale=1.0)

6.3 Stochastic Support

class StochasticSupportInference(model, num_slp_samples, max_slps)
Bases: ABC

Base class for running inference in programs with stochastic support. Each subclass decomposes the input model into so called straight-line programs (SLPs) which are the different control-flow paths in the model. Inference is then run in each SLP separately and the results are combined to produce an overall posterior.

⚠️ Note

This implementation assumes that all stochastic branching is done based on the outcomes of discrete sampling sites that are annotated with infer={"branching": True}. For example,

```python
def model():
    model1 = numpyro.sample("model1", dist.Bernoulli(0.5), infer={"branching": True})
    if model1 == 0:
        mean = numpyro.sample("a1", dist.Normal(0.0, 1.0))
    else:
        mean = numpyro.sample("a2", dist.Normal(1.0, 1.0))
numpyro.sample("obs", dist.Normal(mean, 1.0), obs=0.2)
```
Parameters

- **model** – Python callable containing Pyro primitives *primitives*. local inference. Defaults to NUTS.

- **num_slp_samples** *(int)* – Number of samples to draw from the prior to discover the straight-line programs (SLPs).

- **max_slps** *(int)* – Maximum number of SLPs to discover. DCC will not run inference on more than *max_slps*.

**run** *(rng_key, *args, **kwargs)*

Run inference on each SLP separately and combine the results.

Parameters

- **rng_key** *(jax.random.PRNGKey)* – Random number generator key.

- **args** – Arguments to the model.

- **kwargs** – Keyword arguments to the model.

**class DCC**

Bases: *StochasticSupportInference*

Implements the Divide, Conquer, and Combine (DCC) algorithm for models with stochastic support from [1].

References:


**Example:**

```python
def model():
    model1 = numpyro.sample("model1", dist.Bernoulli(0.5), infer={"branching": True})
    if model1 == 0:
        mean = numpyro.sample("a1", dist.Normal(0.0, 1.0))
    else:
        mean = numpyro.sample("a2", dist.Normal(1.0, 1.0))
    numpyro.sample("obs", dist.Normal(mean, 1.0), obs=0.2)

mcmc_kwvars = dict(
    num_warmup=500, num_samples=1000
)
dcc = DCC(model, mcmc_kwvars=mcmc_kwvars)
dcc_result = dcc.run(random.PRNGKey(0))
```

Parameters

- **model** – Python callable containing Pyro primitives *primitives*.

- **mcmc_kwvars** *(dict)* – Dictionary of arguments passed to MCMC.

- **kernel_cls** *(numpyro.infer.mcmc.MCMCKernel)* – MCMC kernel class that is used for local inference. Defaults to NUTS.
• **num_slp_samples** (*int*) – Number of samples to draw from the prior to discover the straight-line programs (SLPs).

• **max_slps** (*int*) – Maximum number of SLPs to discover. DCC will not run inference on more than `max_slps`.

• **proposal_scale** (*float*) – Scale parameter for the proposal distribution for estimating the normalization constant of an SLP.

```python
class SDVI(model, optimizer, svi_num_steps=1000, combine_elbo_particles=1000, guide_init=<class 'numpyro.infer.autoguide.AutoNormal'>, loss=<numpyro.infer.elbo.Trace_ELBO object>, svi_progress_bar=False, num_slp_samples=1000, max_slps=124)
```

**Bases:** StochasticSupportInference

Implements the Support Decomposition Variational Inference (SDVI) algorithm for models with stochastic support from [1]. This implementation creates a separate guide for each SLP, trains the guides separately, and then combines the guides by weighting them proportional to their ELBO estimates.

**References:**

1. Rethinking Variational Inference for Probabilistic Programs with Stochastic Support, Tim Reichelt, Luke Ong, Tom Rainforth

**Example:**

```python
def model():
    model1 = numpyro.sample("model1", dist.Bernoulli(0.5), infer={"branching": True})
    if model1 == 0:
        mean = numpyro.sample("a1", dist.Normal(0.0, 1.0))
    else:
        mean = numpyro.sample("a2", dist.Normal(1.0, 1.0))
    numpyro.sample("obs", dist.Normal(mean, 1.0), obs=0.2)

sdvi = SDVI(model, numpyro.optim.Adam(step_size=0.001))
sdvi_result = sdvi.run(random.PRNGKey(0))
```

**Parameters**

• **model** – Python callable containing Pyro primitives `primitives`.

• **optimizer** – An instance of `_NumpyroOptim`, a `jax.example_libraries.optimizers.Optimizer` or an Optax GradientTransformation. Gets passed to SVI.

• **svi_num_steps** (*int*) – Number of steps to run SVI for each SLP.

• **combine_elbo_particles** (*int*) – Number of particles to estimate ELBO for computing SLP weights.

• **guide_init** – A constructor for the guide. This should be a callable that returns a `AutoGuide` instance. Defaults to `AutoNormal`.

• **loss** – ELBO loss for SVI. Defaults to `Trace_ELBO`.

• **svi_progress_bar** (*bool*) – Whether to use a progress bar for SVI.

• **num_slp_samples** (*int*) – Number of samples to draw from the prior to discover the straight-line programs (SLPs).
• `max_slps (int)` – Maximum number of SLPs to discover. DCC will not run inference on more than `max_slps`.

## 6.4 Hilbert Space Gaussian Processes Approximation

This module contains helper functions for use in the Hilbert Space Gaussian Process (HSGP) approximation method described in [1] and [2].

⚠️ **Warning**

This module is experimental.

### Why do we need an approximation?

Gaussian processes do not scale well with the number of data points. Recall we had to invert the kernel matrix! The computational complexity of the Gaussian process model is \( \mathcal{O}(n^3) \), where \( n \) is the number of data points. The HSGP approximation method is a way to reduce the computational complexity of the Gaussian process model to \( \mathcal{O}(mn + m) \), where \( m \) is the number of basis functions used in the approximation.

### Approximation Strategy Steps:


Here we provide the main steps and ingredients of the approximation method:

1. Each stationary kernel \( k \) has an associated spectral density \( S(\omega) \). There are closed formulas for the most common kernels. These formulas depend on the hyperparameters of the kernel (e.g. amplitudes and length scales).

2. We can approximate the spectral density \( S(\omega) \) as a polynomial series in \( ||\omega|| \). We call \( \omega \) the frequency.

3. We can interpret these polynomial terms as “powers” of the Laplacian operator. The key observation is that the Fourier transform of the Laplacian operator is \( ||\omega||^2 \).

4. Next, we impose Dirichlet boundary conditions on the Laplacian operator which makes it self-adjoint and with discrete spectrum.

5. We identify the expansion in (2) with the sum of powers of the Laplacian operator in the eigenbasis of (4).

Let \( m^* = \prod_{d=1}^{D} m_d \) be the total number of terms of the approximation, where \( m_d \) is the number of basis functions used in the approximation for the \( d \)-th dimension. Then, the approximation formula, in the non-centered parameterization, is:

\[
    f(x) \approx \sum_{j=1}^{m^*} \left( S(\sqrt{\lambda_j}) \right)^{1/2} \times \phi_j(x) \times \beta_j \sim \text{Normal}(0,1)
\]

where \( x \) is a \( D \) vector of inputs, \( \lambda_j \) are the eigenvalues of the Laplacian operator, \( \phi_j(x) \) are the eigenfunctions of the Laplacian operator, and \( \beta_j \) are the coefficients of the expansion (see Eq. (8) in [2]). We expect this to be a good approximation for a finite number of \( m^* \) terms in the series as long as the inputs values \( x \) are not too close to the boundaries \(-L_d \) and \( L_d \).
Even though the periodic kernel is not stationary, one can still adapt and find a similar approximation formula. However, these kernels are not supported for multidimensional inputs. See Appendix B in [2] for more details.

Example:
Here is an example of how to use the HSGP approximation method with NumPyro. We will use the squared exponential kernel. Other kernels can be used similarly.

```python
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> from numpyro.contrib.hsgp.approximation import hsgp_squared_exponential
>>> import numpyro.distributions as dist
>>> from numpyro.infer import MCMC, NUTS

>>> def generate_synthetic_data(rng_key, start, stop: float, num, scale):
...     # Generate synthetic data.
...     x = jnp.linspace(start=start, stop=stop, num=num)
...     y = jnp.sin(4 * jnp.pi * x) + jnp.sin(7 * jnp.pi * x)
...     y_obs = y + scale * random.normal(rng_key, shape=(num,))
...     return x, y_obs

>>> rng_key = random.PRNGKey(seed=42)
>>> rng_key, rng_subkey = random.split(rng_key)
>>> x, y_obs = generate_synthetic_data(rng_key=rng_subkey, start=0, stop=1, num=80, scale=0.3)

>>> def model(x, ell, m, non_centered, y=None):
...     # Priors
...     alpha = numpyro.sample("alpha", dist.InverseGamma(concentration=12, rate=10))
...     length = numpyro.sample("length", dist.InverseGamma(concentration=6, rate=1))
...     noise = numpyro.sample("noise", dist.InverseGamma(concentration=12, rate=10))
...     # Parametrization
...     f = hsgp_squared_exponential(x=x, alpha=alpha, length=length, ell=ell, m=m, non_centered=non_centered)
...     # Likelihood
...     with numpyro.plate("data", x.shape[0]):
...         numpyro.sample("likelihood", dist.Normal(loc=f, scale=noise), obs=y)
```

(continues on next page)
>>> sampler = NUTS(model)
>>> mcmc = MCMC(sampler=sampler, num_warmup=500, num_samples=1_000, num_chains=2)

>>> rng_key, rng_subkey = random.split(rng_key)

>>> ell = 1.3

>>> m = 20

>>> non_centered = True

>>> mcmc.run(rng_subkey, x, ell, m, non_centered, y_obs)

>>> mcmc.print_summary()

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>1.24</td>
<td>0.34</td>
<td>1.18</td>
<td>0.72</td>
<td>1.74</td>
<td>1804.01</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[0]</td>
<td>-0.10</td>
<td>0.66</td>
<td>-0.10</td>
<td>-1.24</td>
<td>0.92</td>
<td>1819.91</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[1]</td>
<td>0.00</td>
<td>0.71</td>
<td>-0.01</td>
<td>-1.09</td>
<td>1.26</td>
<td>1872.82</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[2]</td>
<td>-0.05</td>
<td>0.69</td>
<td>-0.03</td>
<td>-1.09</td>
<td>1.16</td>
<td>2105.88</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[3]</td>
<td>0.25</td>
<td>0.74</td>
<td>0.26</td>
<td>-0.98</td>
<td>1.42</td>
<td>2281.30</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[4]</td>
<td>-0.17</td>
<td>0.69</td>
<td>-0.17</td>
<td>-1.21</td>
<td>1.00</td>
<td>2551.39</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[5]</td>
<td>0.09</td>
<td>0.75</td>
<td>0.10</td>
<td>-1.13</td>
<td>1.30</td>
<td>3232.13</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[6]</td>
<td>-0.49</td>
<td>0.75</td>
<td>-0.49</td>
<td>-1.65</td>
<td>0.82</td>
<td>3042.31</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[7]</td>
<td>0.42</td>
<td>0.75</td>
<td>0.44</td>
<td>-0.78</td>
<td>1.65</td>
<td>2885.42</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[8]</td>
<td>0.69</td>
<td>0.71</td>
<td>0.71</td>
<td>-0.48</td>
<td>1.82</td>
<td>2811.68</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[9]</td>
<td>-1.43</td>
<td>0.75</td>
<td>-1.40</td>
<td>-2.63</td>
<td>-0.21</td>
<td>2858.68</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[10]</td>
<td>0.33</td>
<td>0.71</td>
<td>0.33</td>
<td>-0.77</td>
<td>1.51</td>
<td>2198.65</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[11]</td>
<td>1.09</td>
<td>0.73</td>
<td>1.11</td>
<td>-0.23</td>
<td>2.18</td>
<td>2765.99</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[12]</td>
<td>-0.91</td>
<td>0.72</td>
<td>-0.91</td>
<td>-2.06</td>
<td>0.31</td>
<td>2586.53</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[13]</td>
<td>0.05</td>
<td>0.70</td>
<td>0.04</td>
<td>-1.16</td>
<td>1.12</td>
<td>2569.59</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[14]</td>
<td>-0.44</td>
<td>0.71</td>
<td>-0.44</td>
<td>-1.58</td>
<td>0.73</td>
<td>2626.09</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[15]</td>
<td>0.69</td>
<td>0.73</td>
<td>0.70</td>
<td>-0.45</td>
<td>1.88</td>
<td>2626.32</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[16]</td>
<td>0.98</td>
<td>0.74</td>
<td>0.98</td>
<td>-0.15</td>
<td>2.28</td>
<td>2282.86</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[17]</td>
<td>-2.54</td>
<td>0.77</td>
<td>-2.52</td>
<td>-3.82</td>
<td>-1.29</td>
<td>3347.56</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[18]</td>
<td>1.35</td>
<td>0.66</td>
<td>1.35</td>
<td>0.30</td>
<td>2.46</td>
<td>2638.17</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[19]</td>
<td>1.10</td>
<td>0.54</td>
<td>1.09</td>
<td>0.25</td>
<td>2.01</td>
<td>2428.37</td>
<td>1.00</td>
</tr>
<tr>
<td>length</td>
<td>0.07</td>
<td>0.01</td>
<td>0.07</td>
<td>0.06</td>
<td>0.09</td>
<td>2321.67</td>
<td>1.00</td>
</tr>
<tr>
<td>noise</td>
<td>0.33</td>
<td>0.03</td>
<td>0.33</td>
<td>0.29</td>
<td>0.38</td>
<td>2472.83</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 0

### Note

Additional examples with code can be found in [3], [4] and [5].

### References:


---

6.4. Hilbert Space Gaussian Processes Approximation
4. Example: Hilbert space approximation for Gaussian processes.

Note

The code of this module is based on the code of the example Example: Hilbert space approximation for Gaussian processes by Omar Sosa Rodríguez.

6.4.1 eigenindices

\texttt{eigenindices(m: list[int] | int, dim: int) \rightarrow ArrayImpl}

Returns the indices of the first $D \times m^*$ eigenvalues of the laplacian operator.

$$m^* = \prod_{i=1}^{D} m_i$$

For more details see Eq. (10) in [1].

References:


Parameters

- \textbf{m} (list[int] | int) – The number of desired eigenvalue indices in each dimension. If an integer, the same number of eigenvalues is computed in each dimension.
- \textbf{dim} (int) – The dimension of the space.

Returns

An array of the indices of the first $D \times m^*$ eigenvalues.

Return type

ArrayImpl

Examples:

```python
>>> import jax.numpy as jnp
>>> from numpyro.contrib.hsgp.laplacian import eigenindices

>>> m = 10
>>> S = eigenindices(m, 1)
>>> assert S.shape == (1, m)
>>> S
Array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], dtype=int32)

>>> m = 10
>>> S = eigenindices(m, 2)
(continues on next page)```
>>> assert S.shape == (2, 100)

>>> m = [2, 2, 3]  # Riutort-Mayol et al eq (10)
>>> S = eigenindices(m, 3)
>>> assert S.shape == (3, 12)

6.4.2 sqrt_eigenvalues

The first $m \times D$ square root of eigenvalues of the laplacian operator in $[-L_1, L_1] \times ... \times [-L_D, L_D]$. See Eq. (56) in [1].

References:


Parameters

- `ell (int | float | list[int | float])` – The length of the interval in each dimension divided by 2. If a float, the same length is used in each dimension.
- `m (list[int] | int)` – The number of eigenvalues to compute in each dimension. If an integer, the same number of eigenvalues is computed in each dimension.
- `dim (int)` – The dimension of the space.

Returns

An array of the first $m \times D$ square root of eigenvalues.

Return type

ArrayImpl

6.4.3 eigenfunctions

The first $m \times D$ eigenfunctions of the laplacian operator in $[-L_1, L_1] \times ... \times [-L_D, L_D]$ evaluated at values of $x$. See Eq. (56) in [1]. If $x$ is 1D, the problem is assumed unidimensional. Otherwise, the dimension of the input space is inferred as the size of the last dimension of $x$. Other dimensions are treated as batch dimensions.

Example:

```python
>>> import jax.numpy as jnp

>>> from numpyro.contrib.hsgp.laplacian import eigenfunctions

>>> n = 100
>>> m = 10
```
```python
>>> x = jnp.linspace(-1, 1, n)
>>> basis = eigenfunctions(x=x, ell=1.2, m=m)
>>> assert basis.shape == (n, m)

>>> x = jnp.ones((n, 3))  # 2d input
>>> basis = eigenfunctions(x=x, ell=1.2, m=[2, 2, 3])
>>> assert basis.shape == (n, 12)
```

### References:


#### Parameters

- **x** (ArrayImpl) – The points at which to evaluate the eigenfunctions. If `x` is 1D the problem is assumed unidimensional. Otherwise, the dimension of the input space is inferred as the last dimension of `x`. Other dimensions are treated as batch dimensions.
- **ell** (float | list[float]) – The length of the interval in each dimension divided by 2. If a float, the same length is used in each dimension.
- **m** (int | list[int]) – The number of eigenvalues to compute in each dimension. If an integer, the same number of eigenvalues is computed in each dimension.

#### Returns

An array of the first $m^* \times D$ eigenfunctions evaluated at `x`.

#### Return type

ArrayImpl

---

### 6.4.4 eigenfunctions_periodic

**eigenfunctions_periodic**(x: ArrayImpl, w0: float, m: int)

Basis functions for the approximation of the periodic kernel.

#### Parameters

- **x** (ArrayImpl) – The points at which to evaluate the eigenfunctions.
- **w0** (float) – The frequency of the periodic kernel.
- **m** (int) – The number of eigenfunctions to compute.

#### Note

If you want to parameterize it with respect to the period use $w0 = 2 * jnp.pi / period$.

#### Warning

Multidimensional inputs are not supported.
### 6.4.5 spectral_density_squared_exponential

**spectral_density_squared_exponential**

```python
dim: int, w: ArrayImpl, alpha: float, length: float | ArrayImpl) → float
```

Spectral density of the squared exponential kernel.

See Section 4.2 in [1] and Section 2.1 in [2].

\[
S(\omega) = \alpha \left( \sqrt{2\pi} \right)^D \ell^D \exp \left( -\frac{1}{2} \ell^2 \omega^T \omega \right)
\]

**References:**


**Parameters**

- `dim` (int) – dimension
- `w` (ArrayImpl) – frequency
- `alpha` (float) – amplitude
- `length` (float) – length scale

**Returns**

spectral density value

**Return type**

float

### 6.4.6 spectral_density_matern

**spectral_density_matern**

```python
dim: int, nu: float, w: ArrayImpl, alpha: float, length: float | ArrayImpl) → float
```

Spectral density of the Matérn kernel.

See Eq. (4.15) in [1] and Section 2.1 in [2].

\[
S(\omega) = \alpha \left( \frac{2^\nu \Gamma(\nu+D/2)(2\nu)\nu}{\Gamma(\nu)\ell^2} \right)^{-\nu-D/2} \left( \frac{2\nu}{\ell^2 + \omega^T \omega} \right)^{-\nu-D/2}
\]

**References:**


**Parameters**

- `dim` (int) – dimension
- `nu` (float) – smoothness
- `w` (ArrayImpl) – frequency
- `alpha` (float) – amplitude
- `length` (float) – length scale

**Returns**

spectral density value

**Return type**

float

---

6.4. Hilbert Space Gaussian Processes Approximation
Returns
spectral density value

Return type
float

### 6.4.7 `diag_spectral_density_squared_exponential`

`diag_spectral_density_squared_exponential(alpha: float, length: float | list[float], ell: float | int | list[float | int], m: int | list[int], dim: int) → ArrayImpl`

Evaluates the spectral density of the squared exponential kernel at the first $D \times m^*$ square root eigenvalues of the laplacian operator in $[-L_1, L_1] \times \ldots \times [-L_D, L_D]$.

**Parameters**

- `alpha (float)` – amplitude of the squared exponential kernel
- `length (float)` – length scale of the squared exponential kernel
- `ell (float | int | list[float | int])` – The length of the interval divided by 2 in each dimension. If a float or int, the same length is used in each dimension.
- `m (int | list[int])` – The number of eigenvalues to compute for each dimension. If an integer, the same number of eigenvalues is computed in each dimension.
- `dim (int)` – The dimension of the space

**Returns**
spectral density vector evaluated at the first $D \times m^*$ square root eigenvalues

**Return type**
ArrayImpl

### 6.4.8 `diag_spectral_density_matern`

`diag_spectral_density_matern(nu: float, alpha: float, length: float, ell: float | int | list[float | int], m: int | list[int], dim: int) → ArrayImpl`

Evaluates the spectral density of the Matérn kernel at the first $D \times m^*$ square root eigenvalues of the laplacian operator in $[-L_1, L_1] \times \ldots \times [-L_D, L_D]$.

**Parameters**

- `nu (float)` – smoothness parameter
- `alpha (float)` – amplitude of the Matérn kernel
- `length (float)` – length scale of the Matérn kernel
- `ell (float | int | list[float | int])` – The length of the interval divided by 2 in each dimension. If a float or int, the same length is used in each dimension.
- `m (int | list[int])` – The number of eigenvalues to compute for each dimension. If an integer, the same number of eigenvalues is computed in each dimension.
- `dim (int)` – The dimension of the space

**Returns**
spectral density vector evaluated at the first $D \times m^*$ square root eigenvalues
6.4.9 `diag_spectral_density_periodic`

`diag_spectral_density_periodic(alpha: float, length: float, m: int) -> ArrayImpl`

Not actually a spectral density but these are used in the same way. These are simply the first $m$ coefficients of the low rank approximation for the periodic kernel. See Appendix B in [1].

References:


Parameters

• `alpha (float)` – amplitude
• `length (float)` – length scale
• `m (int)` – number of eigenvalues

Returns

“spectral density” vector

Return type

ArrayImpl

6.4.10 `hsgp_squared_exponential`

`hsgp_squared_exponential(x: ArrayImpl, alpha: float, length: float, ell: float | int | list[float | int], m: int | list[int], non_centered: bool = True) -> ArrayImpl`

Hilbert space Gaussian process approximation using the squared exponential kernel.

The main idea of the approach is to combine the associated spectral density of the squared exponential kernel and the spectrum of the Dirichlet Laplacian operator to obtain a low-rank approximation of the Gram matrix. For more details see [1, 2].

References:


Parameters

• `x (ArrayImpl)` – input data
• `alpha (float)` – amplitude of the squared exponential kernel
• `length (float)` – length scale of the squared exponential kernel
• `ell (float | int | list[float | int])` – positive value that parametrizes the length of the $D$-dimensional box so that the input data lies in the interval $[-L_1, L_1] \times ... \times [-L_D, L_E]$. We expect the approximation to be valid within this interval
• **m (int | list[m])** – number of eigenvalues to compute and include in the approximation for each dimension ($\{1, ..., D\}$). If an integer, the same number of eigenvalues is computed in each dimension.

• **non_centered (bool)** – whether to use a non-centered parameterization. By default, it is set to True.

**Returns**
the low-rank approximation linear model

**Return type**
ArrayImpl

### 6.4.11 hsgp_matern

**hsgp_matern**(*x: ArrayImpl, nu: float, alpha: float, length: float, ell: float | int | list[float | int], m: int | list[int], non_centered: bool = True*)

Hilbert space Gaussian process approximation using the Matérn kernel.

The main idea of the approach is to combine the associated spectral density of the Matérn kernel kernel and the spectrum of the Dirichlet Laplacian operator to obtain a low-rank approximation of the Gram matrix. For more details see [1, 2].

**References:**

**Parameters**

• **x (ArrayImpl)** – input data

• **nu (float)** – smoothness parameter

• **alpha (float)** – amplitude of the squared exponential kernel

• **length (float)** – length scale of the squared exponential kernel

• **ell (float | int | list[float | int])** – positive value that parameterizes the length of the D-dimensional box so that the input data lies in the interval $[-L_1, L_1] \times ... \times [-L_D, L_D]$. We expect the approximation to be valid within this interval

• **m (int | list[m])** – number of eigenvalues to compute and include in the approximation for each dimension ($\{1, ..., D\}$). If an integer, the same number of eigenvalues is computed in each dimension.

• **non_centered (bool)** – whether to use a non-centered parameterization. By default, it is set to True.

**Returns**
the low-rank approximation linear model

**Return type**
ArrayImpl
6.4.12 hsgp_periodic_non_centered

**hsgp_periodic_non_centered**

```python
(x: ArrayImpl, alpha: float, length: float, w0: float, m: int) \rightarrow ArrayImpl
```

Low rank approximation for the periodic squared exponential kernel in the non-centered parametrization.

See Appendix B in [1].

**References:**


**Parameters**

- `x` *(ArrayImpl)* – input data
- `alpha` *(float)* – amplitude
- `length` *(float)* – length scale
- `w0` *(float)* – frequency of the periodic kernel
- `m` *(int)* – number of eigenvalues to compute and include in the approximation

**Returns**

the low-rank approximation linear model

**Return type**

ArrayImpl
CHAPTER
SEVEN

BAYESIAN REGRESSION USING NUMPYRO

In this tutorial, we will explore how to do bayesian regression in NumPyro, using a simple example adapted from Statistical Rethinking [1]. In particular, we would like to explore the following:

- Write a simple model using the sample NumPyro primitive.
- Run inference using MCMC in NumPyro, in particular, using the No U-Turn Sampler (NUTS) to get a posterior distribution over our regression parameters of interest.
- Learn about inference utilities such as Predictive and log likelihood.
- Learn how we can use effect-handlers in NumPyro to generate execution traces from the model, condition on sample statements, seed models with RNG seeds, etc., and use this to implement various utilities that will be useful for MCMC, e.g. computing model log likelihood, generating empirical distribution over the posterior predictive, etc.

7.1 Tutorial Outline:

1. Dataset
2. Regression Model to Predict Divorce Rate
   - Model-1: Predictor-Marriage Rate
     - Posterior Distribution over the Regression Parameters
     - Prior Predictive Distribution
     - Posterior Predictive Distribution
     - Predictive Utility With Effect Handlers
     - Posterior Predictive Density
   - Model-2: Predictor-Median Age of Marriage
   - Model-3: Predictor-Marriage Rate and Median Age of Marriage
   - Divorce Rate Residuals by State
3. Regression Model with Measurement Error
   - Effect of Incorporating Measurement Noise on Residuals
4. References

[1]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
7.2 Dataset

For this example, we will use the WaffleDivorce dataset from Chapter 05, Statistical Rethinking [1]. The dataset contains divorce rates in each of the 50 states in the USA, along with predictors such as population, median age of marriage, whether it is a Southern state and, curiously, number of Waffle Houses.

```
[3]: DATASET_URL = "https://raw.githubusercontent.com/rmcelreath/rethinking/master/data/…WaffleDivorce.csv"
    dset = pd.read_csv(DATASET_URL, sep=";")
```

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Loc</th>
<th>Population1860</th>
<th>PropSlaves1860</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama AL</td>
<td>0</td>
<td>964201</td>
<td>0.450000</td>
</tr>
<tr>
<td>Alaska AK</td>
<td>1</td>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>Arizona AZ</td>
<td>2</td>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>Arkansas AR</td>
<td>3</td>
<td>435450</td>
<td>0.260000</td>
</tr>
<tr>
<td>California CA</td>
<td>4</td>
<td>379994</td>
<td>0.000000</td>
</tr>
<tr>
<td>Colorado CO</td>
<td>5</td>
<td>34277</td>
<td>0.000000</td>
</tr>
<tr>
<td>Connecticut CT</td>
<td>6</td>
<td>460147</td>
<td>0.000000</td>
</tr>
<tr>
<td>Delaware DE</td>
<td>7</td>
<td>112216</td>
<td>0.016000</td>
</tr>
<tr>
<td>District of Columbia DC</td>
<td>8</td>
<td>75080</td>
<td>0.000000</td>
</tr>
<tr>
<td>Florida FL</td>
<td>9</td>
<td>140424</td>
<td>0.440000</td>
</tr>
<tr>
<td>Georgia GA</td>
<td>10</td>
<td>1057286</td>
<td>0.440000</td>
</tr>
<tr>
<td>Hawaii HI</td>
<td>11</td>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>Idaho ID</td>
<td>12</td>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>Illinois IL</td>
<td>13</td>
<td>1711951</td>
<td>0.000000</td>
</tr>
<tr>
<td>Indiana IN</td>
<td>14</td>
<td>1350428</td>
<td>0.000000</td>
</tr>
<tr>
<td>Iowa IA</td>
<td>15</td>
<td>674913</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
```

(continues on next page)
Let us plot the pair-wise relationship amongst the main variables in the dataset, using `seaborn.pairplot`.

```python
[4]: vars = [
    "Population",
    "MedianAgeMarriage",
    "Marriage",
    "WaffleHouses",
    "South",
    "Divorce",
]

sns.pairplot(dset, x_vars=vars, y_vars=vars, palette="husl");
```
From the plots above, we can clearly observe that there is a relationship between divorce rates and marriage rates in a state (as might be expected), and also between divorce rates and median age of marriage.

There is also a weak relationship between number of Waffle Houses and divorce rates, which is not obvious from the plot above, but will be clearer if we regress Divorce against WaffleHouse and plot the results.

```
[5]: sns.regplot(x="WaffleHouses", y="Divorce", data=dset);
```
This is an example of a spurious association. We do not expect the number of Waffle Houses in a state to affect the divorce rate, but it is likely correlated with other factors that have an effect on the divorce rate. We will not delve into this spurious association in this tutorial, but the interested reader is encouraged to read Chapters 5 and 6 of [1] which explores the problem of causal association in the presence of multiple predictors.

For simplicity, we will primarily focus on marriage rate and the median age of marriage as our predictors for divorce rate throughout the remaining tutorial.

7.3 Regression Model to Predict Divorce Rate

Let us now write a regression model in NumPyro to predict the divorce rate as a linear function of marriage rate and median age of marriage in each of the states.

First, note that our predictor variables have somewhat different scales. It is a good practice to standardize our predictors and response variables to mean 0 and standard deviation 1, which should result in faster inference.

```
[6]: def standardize(x):
    return (x - x.mean()) / x.std()

dset["AgeScaled"] = dset.MedianAgeMarriage.pipe(standardize)
dset["MarriageScaled"] = dset.Marriage.pipe(standardize)
dset["DivorceScaled"] = dset.Divorce.pipe(standardize)
```

We write the NumPyro model as follows. While the code should largely be self-explanatory, take note of the following:

- In NumPyro, model code is any Python callable which can optionally accept additional arguments and keywords. For HMC which we will be using for this tutorial, these arguments and keywords remain static during inference, but we can reuse the same model to generate predictions on new data.

- In addition to regular Python statements, the model code also contains primitives like sample. These primitives can be interpreted with various side-effects using effect handlers. For more on effect handlers, refer to [3], [4]. For now, just remember that a sample statement makes this a stochastic function that samples some latent parameters from a prior distribution. Our goal is to infer the posterior distribution of these parameters conditioned on observed data.
There is the reason why we have kept our predictors as optional keyword arguments is to be able to reuse the same model as we vary the set of predictors. Likewise, the reason why the response variable is optional is that we would like to reuse this model to sample from the posterior predictive distribution. See the section on plotting the posterior predictive distribution, as an example.

```python
[7]: def model(marriage=None, age=None, divorce=None):
    a = numpyro.sample("a", dist.Normal(0.0, 0.2))
    M, A = 0.0, 0.0
    if marriage is not None:
        bM = numpyro.sample("bM", dist.Normal(0.0, 0.5))
        M = bM * marriage
    if age is not None:
        bA = numpyro.sample("bA", dist.Normal(0.0, 0.5))
        A = bA * age
    sigma = numpyro.sample("sigma", dist.Exponential(1.0))
    mu = a + M + A
    numpyro.sample("obs", dist.Normal(mu, sigma), obs=divorce)
```

7.3.1 Model 1: Predictor - Marriage Rate

We first try to model the divorce rate as depending on a single variable, marriage rate. As mentioned above, we can use the same model code as earlier, but only pass values for marriage and divorce keyword arguments. We will use the No U-Turn Sampler (see [5] for more details on the NUTS algorithm) to run inference on this simple model.

The Hamiltonian Monte Carlo (or, the NUTS) implementation in NumPyro takes in a potential energy function. This is the negative log joint density for the model. Therefore, for our model description above, we need to construct a function which given the parameter values returns the potential energy (or negative log joint density). Additionally, the verlet integrator in HMC (or, NUTS) returns sample values simulated using Hamiltonian dynamics in the unconstrained space. As such, continuous variables with bounded support need to be transformed into unconstrained space using bijective transforms. We also need to transform these samples back to their constrained support before returning these values to the user. Thankfully, this is handled on the backend for us, within a convenience class for doing MCMC inference that has the following methods:

- **run(...)**: runs warmup, adapts steps size and mass matrix, and does sampling using the sample from the warmup phase.
- **print_summary()**: print diagnostic information like quantiles, effective sample size, and the Gelman-Rubin diagnostic.
- **get_samples()**: gets samples from the posterior distribution.

Note the following:

- **JAX uses functional PRNGs.** Unlike other languages / frameworks which maintain a global random state, in JAX, every call to a sampler requires an explicit PRNGKey. We will split our initial random seed for subsequent operations, so that we do not accidentally reuse the same seed.
- **We run inference with the NUTS sampler.** To run vanilla HMC, we can instead use the HMC class.

```python
[8]: # Start from this source of randomness. We will split keys for subsequent operations.
    rng_key = random.PRNGKey(0)
    rng_key, rng_key_ = random.split(rng_key)

    # Run NUTS.
    kernel = NUTS(model)
    num_samples = 2000
```

(continues on next page)
mcmc = MCMC(kernel, num_warmup=1000, num_samples=num_samples)
mcmc.run(
    rng_key_, marriage=dset.MarriageScaled.values, divorce=dset.DivorceScaled.values
)
mcmc.print_summary()
samples_1 = mcmc.get_samples()

| sample: 100%| progress| 3000/3000 [00:04<00:00, 748.14it/s, 7 steps of...
--size 7.41e-01. acc. prob=0.92]

<table>
<thead>
<tr>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.00</td>
<td>0.11</td>
<td>0.00</td>
<td>-0.16</td>
<td>0.20</td>
<td>1510.96</td>
</tr>
<tr>
<td>bM</td>
<td>0.35</td>
<td>0.13</td>
<td>0.35</td>
<td>0.14</td>
<td>0.57</td>
<td>2043.12</td>
</tr>
<tr>
<td>sigma</td>
<td>0.95</td>
<td>0.10</td>
<td>0.94</td>
<td>0.78</td>
<td>1.10</td>
<td>1565.40</td>
</tr>
</tbody>
</table>

Number of divergences: 0

**Posterior Distribution over the Regression Parameters**

We notice that the progress bar gives us online statistics on the acceptance probability, step size and number of steps taken per sample while running NUTS. In particular, during warmup, we adapt the step size and mass matrix to achieve a certain target acceptance probability which is 0.8, by default. We were able to successfully adapt our step size to achieve this target in the warmup phase.

During warmup, the aim is to adapt hyper-parameters such as step size and mass matrix (the HMC algorithm is very sensitive to these hyper-parameters), and to reach the typical set (see [6] for more details). If there are any issues in the model specification, the first signal to notice would be low acceptance probabilities or very high number of steps. We use the sample from the end of the warmup phase to seed the MCMC chain (denoted by the second sample progress bar) from which we generate the desired number of samples from our target distribution.

At the end of inference, NumPyro prints the mean, std and 90% CI values for each of the latent parameters. Note that since we standardized our predictors and response variable, we would expect the intercept to have mean 0, as can be seen here. It also prints other convergence diagnostics on the latent parameters in the model, including effective sample size and the gelman rubin diagnostic ($\hat{R}$). The value for these diagnostics indicates that the chain has converged to the target distribution. In our case, the “target distribution” is the posterior distribution over the latent parameters that we are interested in. Note that this is often worth verifying with multiple chains for more complicated models. In the end, samples_1 is a collection (in our case, a dict since init_samples was a dict) containing samples from the posterior distribution for each of the latent parameters in the model.

To look at our regression fit, let us plot the regression line using our posterior estimates for the regression parameters, along with the 90% Credibility Interval (CI). Note that the hpd function in NumPyro’s diagnostics module can be used to compute CI. In the functions below, note that the collected samples from the posterior are all along the leading axis.

```python
[9]: def plot_regression(x, y_mean, y_hpdi):
    # Sort values for plotting by x axis
    idx = jnp.argsort(x)
    marriage = x[idx]
    mean = y_mean[idx]
    hpdi = y_hpdi[:, idx]
    divorce = dset.DivorceScaled.values[idx]

    # Plot
```

7.3. Regression Model to Predict Divorce Rate 269
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 6))
ax.plot(marriage, mean)
ax.plot(marriage, divorce, "o")
ax.fill_between(marriage, hpdi[0], hpdi[1], alpha=0.3, interpolate=True)
return ax

# Compute empirical posterior distribution over mu
posterior_mu = (jnp.expand_dims(samples_1["a"], -1) + jnp.expand_dims(samples_1["bM"], -1) * dset.MarriageScaled.values)
mean_mu = jnp.mean(posterior_mu, axis=0)
hpdi_mu = hpdi(posterior_mu, 0.9)
ax = plot_regression(dset.MarriageScaled.values, mean_mu, hpdi_mu)
ax.set(
xlabel="Marriage rate", ylabel="Divorce rate", title="Regression line with 90% CI")

We can see from the plot, that the CI broadens towards the tails where the data is relatively sparse, as can be expected.
Prior Predictive Distribution

Let us check that we have set sensible priors by sampling from the prior predictive distribution. NumPyro provides a handy `Predictive` utility for this purpose.

```python
from numpyro.infer import Predictive
rng_key, rng_key_ = random.split(rng_key)
prior_predictive = Predictive(model, num_samples=100)
prior_predictions = prior_predictive(rng_key_, marriage=dset.MarriageScaled.values)[
    "obs"
]
mean_prior_pred = jnp.mean(prior_predictions, axis=0)
hpdi_prior_pred = hpdi(prior_predictions, 0.9)
ax = plot_regression(dset.MarriageScaled.values, mean_prior_pred, hpdi_prior_pred)
ax.set(xlabel="Marriage rate", ylabel="Divorce rate", title="Predictions with 90% CI");
```

Posterior Predictive Distribution

Let us now look at the posterior predictive distribution to see how our predictive distribution looks with respect to the observed divorce rates. To get samples from the posterior predictive distribution, we need to run the model by substituting the latent parameters with samples from the posterior. Note that by default we generate a single prediction for each sample from the joint posterior distribution, but this can be controlled using the `num_samples` argument.

```python
rng_key, rng_key_ = random.split(rng_key)
predictive = Predictive(model, samples_1)
```
predictions = predictive(rng_key, marriage=dset.MarriageScaled.values)["obs"]
df = dset.filter(["Location"])
df["Mean Predictions"] = jnp.mean(predictions, axis=0)
df.head()

<table>
<thead>
<tr>
<th>Location</th>
<th>Mean Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.016434</td>
</tr>
<tr>
<td>1</td>
<td>0.501293</td>
</tr>
<tr>
<td>2</td>
<td>0.025105</td>
</tr>
<tr>
<td>3</td>
<td>0.600058</td>
</tr>
<tr>
<td>4</td>
<td>-0.082887</td>
</tr>
</tbody>
</table>

**Predictive Utility With Effect Handlers**

To remove the magic behind Predictive, let us see how we can combine effect handlers with the `vmap` JAX primitive to implement our own simplified predictive utility function that can do vectorized predictions.

```python
[12]: def predict(rng_key, post_samples, model, *args, **kwargs):
    model = handlers.seed(handlers.condition(model, post_samples), rng_key)
    model_trace = handlers.trace(model).get_trace(*args, **kwargs)
    return model_trace["obs"]["value"]

# vectorize predictions via vmap
predict_fn = vmap(
    lambda rng_key, samples: predict(  
        rng_key, samples, model, marriage=dset.MarriageScaled.values
    )
)
```

Note the use of the condition, seed and trace effect handlers in the `predict` function.

- The `seed` effect-handler is used to wrap a stochastic function with an initial PRNGKey seed. When a sample statement inside the model is called, it uses the existing seed to sample from a distribution but this effect-handler also splits the existing key to ensure that future `sample` calls in the model use the newly split key instead. This is to prevent us from having to explicitly pass in a PRNGKey to each `sample` statement in the model.

- The `condition` effect handler conditions the latent samples to certain values. In our case, we are conditioning on values from the posterior distribution returned by MCMC.

- The `trace` effect handler runs the model and records the execution trace within an `OrderedDict`. This trace object contains execution metadata that is useful for computing quantities such as the log joint density.

It should be clear now that the `predict` function simply runs the model by substituting the latent parameters with samples from the posterior (generated by the `mcmc` function) to generate predictions. Note the use of JAX’s auto-vectorization transform called `vmap` to vectorize predictions. Note that if we didn’t use `vmap`, we would have to use a native for loop which for each sample which is much slower. Each draw from the posterior can be used to get predictions over all the 50 states. When we vectorize this over all the samples from the posterior using `vmap`, we will get a `predictions_1` array of shape `(num_samples, 50)`. We can then compute the mean and 90% CI of these samples to plot the posterior predictive distribution. We note that our mean predictions match those obtained from the Predictive utility class.

```python
[13]: # Using the same key as we used for Predictive - note that the results are identical.
```

(continues on next page)
predictions_1 = predict_fn(random.split(rng_key_, num_samples), samples_1)

mean_pred = jnp.mean(predictions_1, axis=0)
df = dset.filter(['Location'])
df['Mean Predictions'] = mean_pred
df.head()

[13]:
<table>
<thead>
<tr>
<th>Location</th>
<th>Mean Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>0.016434</td>
</tr>
<tr>
<td>Alaska</td>
<td>0.501293</td>
</tr>
<tr>
<td>Arizona</td>
<td>0.025105</td>
</tr>
<tr>
<td>Arkansas</td>
<td>0.600058</td>
</tr>
<tr>
<td>California</td>
<td>-0.082887</td>
</tr>
</tbody>
</table>

[14]:
hpdi_pred = hpdi(predictions_1, 0.9)

ax = plot_regression(dset.MarriageScaled.values, mean_pred, hpdi_pred)
ax.set(xlabel='Marriage rate', ylabel='Divorce rate', title='Predictions with 90% CI');

We have used the same plot_regression function as earlier. We notice that our CI for the predictive distribution is much broader as compared to the last plot due to the additional noise introduced by the sigma parameter. Most data points lie well within the 90% CI, which indicates a good fit.
Posterior Predictive Density

Likewise, making use of effect-handlers and `vmap`, we can also compute the log likelihood for this model given the dataset, and the log posterior predictive density \[6\] which is given by

\[
\log \prod_{i=1}^{n} p(y_i|\theta)p_{post}(\theta)d\theta \approx \sum_{i=1}^{n} \log \frac{\sum_s p(\theta^s)}{S} = \sum_{i=1}^{n} (\log \sum_s p(\theta^s) - \log(S))
\]

Here, \(i\) indexes the observed data points \(y\) and \(s\) indexes the posterior samples over the latent parameters \(\theta\). If the posterior predictive density for a model has a comparatively high value, it indicates that the observed data-points have higher probability under the given model.

```
[15]: def log_likelihood(rng_key, params, model, *args, **kwargs):
    model = handlers.condition(model, params)
    model_trace = handlers.trace(model).get_trace(*args, **kwargs)
    obs_node = model_trace["obs"]
    return obs_node["fn"].log_prob(obs_node["value"])

def log_pred_density(rng_key, params, model, *args, **kwargs):
    n = list(params.values())[0].shape[0]
    log_lk_fn = vmap(
        lambda rng_key, params: log_likelihood(rng_key, params, model, *args, **kwargs)
    )
    log_lk_vals = log_lk_fn(random.split(rng_key, n), params)
    return (logsumexp(log_lk_vals, 0) - jnp.log(n)).sum()
```

Note that NumPyro provides the `log_likelihood` utility function that can be used directly for computing log likelihood as in the first function for any general model. In this tutorial, we would like to emphasize that there is nothing magical about such utility functions, and you can roll out your own inference utilities using NumPyro’s effect handling stack.

```
[16]: rng_key, rng_key_ = random.split(rng_key)
print(
    "Log posterior predictive density: {}".format(
        log_pred_density(
            rng_key_,
            samples_1,
            model,
            marriage=dset.MarriageScaled.values,
            divorce=dset.DivorceScaled.values,
        )
    )
)
```

Log posterior predictive density: -66.70008087158203
7.3.2 Model 2: Predictor - Median Age of Marriage

We will now model the divorce rate as a function of the median age of marriage. The computations are mostly a reproduction of what we did for Model 1. Notice the following:

- Divorce rate is inversely related to the age of marriage. Hence states where the median age of marriage is low will likely have a higher divorce rate.
- We get a higher log likelihood as compared to Model 2, indicating that median age of marriage is likely a much better predictor of divorce rate.

```python
rng_key, rng_key_ = random.split(rng_key)
mcmc.run(rng_key_, age=dset.AgeScaled.values, divorce=dset.DivorceScaled.values)
mcmc.print_summary()
samples_2 = mcmc.get_samples()

Number of divergences: 0
```

```python
posterior_mu = (  
jnp.expand_dims(samples_2["a"], -1)  
+ np.expand_dims(samples_2["bA"], -1) * dset.AgeScaled.values
)
mean_mu = jnp.mean(posterior_mu, axis=0)
hpdi_mu = hpdi(posterior_mu, 0.9)
ax = plot_regression(dset.AgeScaled.values, mean_mu, hpdi_mu)
ax.set(  
xlabel="Median marriage age",  
ylabel="Divorce rate",  
title="Regression line with 90% CI",  
);```
 rng_key, rng_key_ = random.split(rng_key)
predictions_2 = Predictive(model, samples_2)(rng_key_, age=dset.AgeScaled.values)['obs']

mean_pred = jnp.mean(predictions_2, axis=0)
hpdi_pred = hpdi(predictions_2, 0.9)

ax = plot_regression(dset.AgeScaled.values, mean_pred, hpdi_pred)
ax.set(xlabel="Median Age", ylabel="Divorce rate", title="Predictions with 90% CI");
7.3.3 Model 3: Predictor - Marriage Rate and Median Age of Marriage

Finally, we will also model divorce rate as depending on both marriage rate as well as the median age of marriage. Note that the model’s posterior predictive density is similar to Model 2 which likely indicates that the marginal information from marriage rate in predicting divorce rate is low when the median age of marriage is already known.

```
rng_key, rng_key_ = random.split(rng_key)
print(
    "Log posterior predictive density: {}\".format(
        log_pred_density(
            rng_key_,
            samples_2,
            model,
            age=dset.AgeScaled.values,
            divorce=dset.DivorceScaled.values,
        )
    )
)
Log posterior predictive density: -59.251956939697266
```

7.3. Regression Model to Predict Divorce Rate

(continues on next page)
```python
age=dset.AgeScaled.values,
  divorce=dset.DivorceScaled.values,
)
mcmc.print_summary()
samples_3 = mcmc.get_samples()

samples_3
```

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>-0.17</td>
<td>0.16</td>
<td>2007.41</td>
<td>1.00</td>
</tr>
<tr>
<td>bA</td>
<td>-0.61</td>
<td>0.16</td>
<td>-0.61</td>
<td>-0.89</td>
<td>-0.37</td>
<td>1225.02</td>
<td>1.00</td>
</tr>
<tr>
<td>bM</td>
<td>-0.07</td>
<td>0.16</td>
<td>-0.07</td>
<td>-0.34</td>
<td>0.19</td>
<td>1275.37</td>
<td>1.00</td>
</tr>
<tr>
<td>sigma</td>
<td>0.83</td>
<td>0.08</td>
<td>0.82</td>
<td>0.69</td>
<td>0.96</td>
<td>1820.77</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 0

```python
[22]:
rng_key, rng_key_ = random.split(rng_key)
print(
    "Log posterior predictive density: {}".format(
        log_pred_density(
            rng_key_,
            samples_3,
            model,
            marriage=dset.MarriageScaled.values,
            age=dset.AgeScaled.values,
            divorce=dset.DivorceScaled.values,
        )
    )
)
Log posterior predictive density: -59.06374740600586
```

## 7.3.4 Divorce Rate Residuals by State

The regression plots above shows that the observed divorce rates for many states differs considerably from the mean regression line. To dig deeper into how the last model (Model 3) under-predicts or over-predicts for each of the states, we will plot the posterior predictive and residuals (Observed divorce rate - Predicted divorce rate) for each of the states.

```python
[23]:
# Predictions for Model 3.
rng_key, rng_key_ = random.split(rng_key)
predictions_3 = Predictive(model, samples_3)(
    rng_key_,
    marriage=dset.MarriageScaled.values,
    age=dset.AgeScaled.values,
)["obs"]
y = jnp.arange(50)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 16))
pred_mean = jnp.mean(predictions_3, axis=0)
pred_hpdi = hpdi(predictions_3, 0.9)
(continues on next page)```
residuals_3 = dset.DivorceScaled.values - predictions_3  
residuals_mean = jnp.mean(residuals_3, axis=0)  
residuals_hpdi = hpdi(residuals_3, 0.9)  
idx = jnp.argsort(residuals_mean)

# Plot posterior predictive  
ax[0].plot(jnp.zeros(50), y, "--")  
ax[0].errorbar(  
    pred_mean[idx],  
y,  
xerr=pred_hpdi[1, idx] - pred_mean[idx],  
    marker="o",  
    ms=5,  
    mew=4,  
    ls="none",  
    alpha=0.8,  
)  
ax[0].plot(dset.DivorceScaled.values[idx], y, marker="o", ls="none", color="gray")  
ax[0].set(  
    xlabel="Posterior Predictive (red) vs. Actuals (gray)",  
    ylabel="State",  
    title="Posterior Predictive with 90% CI",  
)  
ax[0].set_yticks(y)  
ax[0].set_yticklabels(dset.Loc.values[idx], fontsize=10)

# Plot residuals  
residuals_3 = dset.DivorceScaled.values - predictions_3  
residuals_mean = jnp.mean(residuals_3, axis=0)  
residuals_hpdi = hpdi(residuals_3, 0.9)  
err = residuals_hpdi[1] - residuals_mean  
ax[1].plot(jnp.zeros(50), y, "--")  
ax[1].errorbar(  
    residuals_mean[idx],  
y,  
xerr=err[idx],  
    marker="o",  
    ms=5,  
    mew=4,  
    ls="none",  
    alpha=0.8  
)  
ax[1].set(xlabel="Residuals", ylabel="State", title="Residuals with 90% CI")  
ax[1].set_yticks(y)  
ax[1].set_yticklabels(dset.Loc.values[idx], fontsize=10);
The plot on the left shows the mean predictions with 90% CI for each of the states using Model 3. The gray markers indicate the actual observed divorce rates. The right plot shows the residuals for each of the states, and both these plots
are sorted by the residuals, i.e. at the bottom, we are looking at states where the model predictions are higher than the observed rates, whereas at the top, the reverse is true.

Overall, the model fit seems good because most observed data points like within a 90% CI around the mean predictions. However, notice how the model over-predicts by a large margin for states like Idaho (bottom left), and on the other end under-predicts for states like Maine (top right). This is likely indicative of other factors that we are missing out in our model that affect divorce rate across different states. Even ignoring other socio-political variables, one such factor that we have not yet modeled is the measurement noise given by Divorce SE in the dataset. We will explore this in the next section.

### 7.4 Regression Model with Measurement Error

Note that in our previous models, each data point influences the regression line equally. Is this well justified? We will build on the previous model to incorporate measurement error given by Divorce SE variable in the dataset. Incorporating measurement noise will be useful in ensuring that observations that have higher confidence (i.e. lower measurement noise) have a greater impact on the regression line. On the other hand, this will also help us better model outliers with high measurement errors. For more details on modeling errors due to measurement noise, refer to Chapter 14 of [1].

To do this, we will reuse Model 3, with the only change that the final observed value has a measurement error given by divorce_sd (notice that this has to be standardized since the divorce variable itself has been standardized to mean 0 and std 1).

```python
[24]: def model_se(marriage, age, divorce_sd, divorce=None):
    a = numpyro.sample("a", dist.Normal(0.0, 0.2))
    bM = numpyro.sample("bM", dist.Normal(0.0, 0.5))
    M = bM * marriage
    bA = numpyro.sample("bA", dist.Normal(0.0, 0.5))
    A = bA * age
    sigma = numpyro.sample("sigma", dist.Exponential(1.0))
    mu = a + M + A
    divorce_rate = numpyro.sample("divorce_rate", dist.Normal(mu, sigma))
    numpyro.sample("obs", dist.Normal(divorce_rate, divorce_sd), obs=divorce)

[25]: # Standardize
    dset["DivorceScaledSD"] = dset["Divorce SE"] / jnp.std(dset.Divorce.values)

[26]: rng_key, rng_key_ = random.split(rng_key)

kernel = NUTS(model_se, target_accept_prob=0.9)
mcmc = MCMC(kernel, num_warmup=1000, num_samples=3000)
mcmc.run(
    rng_key_,
    marriage=dset.MarriageScaled.values,
    age=dset.AgeScaled.values,
    divorce_sd=dset.DivorceScaledSD.values,
    divorce=dset.DivorceScaled.values,
)
mcmc.print_summary()
samples_4 = mcmc.get_samples()
```

```
sample: 100%|████████████████████████████████| 4000/4000 [00:06<00:00, 578.19it/s, 15 steps of␣
˓→size 2.58e-01. acc. prob=0.93]
```

### 7.4 Regression Model with Measurement Error

281
<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-0.06</td>
<td>0.10</td>
<td>-0.06</td>
<td>-0.20</td>
<td>0.11</td>
<td>3203.01</td>
<td>1.00</td>
</tr>
<tr>
<td>bA</td>
<td>-0.61</td>
<td>0.16</td>
<td>-0.61</td>
<td>-0.87</td>
<td>-0.35</td>
<td>2156.51</td>
<td>1.00</td>
</tr>
<tr>
<td>bM</td>
<td>0.06</td>
<td>0.17</td>
<td>0.06</td>
<td>-0.21</td>
<td>0.33</td>
<td>1943.15</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[0]</td>
<td>1.16</td>
<td>0.36</td>
<td>1.15</td>
<td>0.53</td>
<td>1.72</td>
<td>2488.98</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[1]</td>
<td>0.69</td>
<td>0.55</td>
<td>0.68</td>
<td>-0.15</td>
<td>1.65</td>
<td>4832.63</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[2]</td>
<td>0.42</td>
<td>0.34</td>
<td>0.42</td>
<td>-0.16</td>
<td>0.96</td>
<td>4419.13</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[3]</td>
<td>1.41</td>
<td>0.46</td>
<td>1.40</td>
<td>0.63</td>
<td>2.11</td>
<td>4782.86</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[4]</td>
<td>-0.90</td>
<td>0.13</td>
<td>-0.90</td>
<td>-1.12</td>
<td>-0.71</td>
<td>4269.33</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[5]</td>
<td>0.65</td>
<td>0.39</td>
<td>0.65</td>
<td>0.01</td>
<td>1.31</td>
<td>4139.51</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[6]</td>
<td>-1.36</td>
<td>0.35</td>
<td>-1.36</td>
<td>-1.96</td>
<td>-0.82</td>
<td>5180.21</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[7]</td>
<td>-0.33</td>
<td>0.49</td>
<td>-0.33</td>
<td>-1.15</td>
<td>0.45</td>
<td>4089.39</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[8]</td>
<td>-1.88</td>
<td>0.59</td>
<td>-1.88</td>
<td>-2.89</td>
<td>-0.93</td>
<td>3305.68</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[9]</td>
<td>-0.62</td>
<td>0.17</td>
<td>-0.61</td>
<td>-0.90</td>
<td>-0.34</td>
<td>4936.95</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[10]</td>
<td>0.76</td>
<td>0.29</td>
<td>0.76</td>
<td>0.28</td>
<td>1.24</td>
<td>3627.89</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[11]</td>
<td>-0.55</td>
<td>0.50</td>
<td>-0.55</td>
<td>-1.38</td>
<td>0.26</td>
<td>3822.80</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[12]</td>
<td>0.20</td>
<td>0.53</td>
<td>0.20</td>
<td>-0.74</td>
<td>0.99</td>
<td>1476.70</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[13]</td>
<td>-0.86</td>
<td>0.23</td>
<td>-0.87</td>
<td>-1.24</td>
<td>-0.48</td>
<td>5333.10</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[14]</td>
<td>0.55</td>
<td>0.30</td>
<td>0.55</td>
<td>0.09</td>
<td>1.05</td>
<td>5533.56</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[15]</td>
<td>0.28</td>
<td>0.38</td>
<td>0.28</td>
<td>-0.35</td>
<td>0.92</td>
<td>5179.68</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[16]</td>
<td>0.49</td>
<td>0.43</td>
<td>0.49</td>
<td>-0.23</td>
<td>1.16</td>
<td>5134.56</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[17]</td>
<td>1.25</td>
<td>0.35</td>
<td>1.24</td>
<td>0.69</td>
<td>1.84</td>
<td>4571.21</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[18]</td>
<td>0.42</td>
<td>0.38</td>
<td>0.41</td>
<td>-0.15</td>
<td>1.10</td>
<td>4946.86</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[19]</td>
<td>0.38</td>
<td>0.55</td>
<td>0.36</td>
<td>-0.50</td>
<td>1.29</td>
<td>2145.11</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[20]</td>
<td>-0.56</td>
<td>0.34</td>
<td>-0.56</td>
<td>-1.12</td>
<td>-0.02</td>
<td>5219.59</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[21]</td>
<td>-1.11</td>
<td>0.27</td>
<td>-1.11</td>
<td>-1.53</td>
<td>-0.65</td>
<td>3778.88</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[22]</td>
<td>-0.28</td>
<td>0.26</td>
<td>-0.28</td>
<td>-0.71</td>
<td>0.13</td>
<td>5751.65</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[23]</td>
<td>-0.99</td>
<td>0.30</td>
<td>-0.99</td>
<td>-1.46</td>
<td>-0.49</td>
<td>4385.57</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[24]</td>
<td>0.43</td>
<td>0.41</td>
<td>0.42</td>
<td>-0.26</td>
<td>1.08</td>
<td>3868.84</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[25]</td>
<td>-0.03</td>
<td>0.32</td>
<td>-0.03</td>
<td>-0.57</td>
<td>0.48</td>
<td>5927.41</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[26]</td>
<td>-0.01</td>
<td>0.49</td>
<td>-0.01</td>
<td>-0.79</td>
<td>0.81</td>
<td>4581.29</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[27]</td>
<td>-0.16</td>
<td>0.39</td>
<td>-0.15</td>
<td>-0.79</td>
<td>0.49</td>
<td>4522.45</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[28]</td>
<td>-0.27</td>
<td>0.50</td>
<td>-0.29</td>
<td>-1.08</td>
<td>0.53</td>
<td>3824.97</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[29]</td>
<td>-1.79</td>
<td>0.24</td>
<td>-1.78</td>
<td>-2.18</td>
<td>-1.39</td>
<td>5134.14</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[30]</td>
<td>0.17</td>
<td>0.42</td>
<td>0.16</td>
<td>-0.55</td>
<td>0.82</td>
<td>5978.21</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[31]</td>
<td>-1.66</td>
<td>0.16</td>
<td>-1.66</td>
<td>-1.93</td>
<td>-1.41</td>
<td>5976.18</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[32]</td>
<td>0.12</td>
<td>0.25</td>
<td>0.12</td>
<td>-0.27</td>
<td>0.52</td>
<td>5759.99</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[33]</td>
<td>-0.04</td>
<td>0.52</td>
<td>-0.04</td>
<td>-0.91</td>
<td>0.82</td>
<td>2926.68</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[34]</td>
<td>-0.13</td>
<td>0.22</td>
<td>-0.13</td>
<td>-0.50</td>
<td>0.23</td>
<td>4390.05</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[35]</td>
<td>1.27</td>
<td>0.43</td>
<td>1.27</td>
<td>0.53</td>
<td>1.94</td>
<td>4659.54</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[36]</td>
<td>0.22</td>
<td>0.36</td>
<td>0.22</td>
<td>-0.36</td>
<td>0.84</td>
<td>3758.16</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[37]</td>
<td>-1.02</td>
<td>0.23</td>
<td>-1.02</td>
<td>-1.38</td>
<td>-0.64</td>
<td>5954.84</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[38]</td>
<td>-0.93</td>
<td>0.54</td>
<td>-0.94</td>
<td>-1.84</td>
<td>-0.06</td>
<td>3289.66</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[39]</td>
<td>-0.67</td>
<td>0.33</td>
<td>-0.67</td>
<td>-1.18</td>
<td>-0.09</td>
<td>4787.55</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[40]</td>
<td>0.25</td>
<td>0.55</td>
<td>0.24</td>
<td>-0.67</td>
<td>1.16</td>
<td>4526.98</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[41]</td>
<td>0.73</td>
<td>0.34</td>
<td>0.73</td>
<td>0.17</td>
<td>1.29</td>
<td>4237.28</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[42]</td>
<td>0.20</td>
<td>0.18</td>
<td>0.20</td>
<td>-0.10</td>
<td>0.48</td>
<td>5156.91</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[43]</td>
<td>0.81</td>
<td>0.43</td>
<td>0.81</td>
<td>0.14</td>
<td>1.50</td>
<td>2067.24</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[44]</td>
<td>-0.42</td>
<td>0.51</td>
<td>-0.43</td>
<td>-1.23</td>
<td>0.45</td>
<td>3844.29</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[45]</td>
<td>-0.39</td>
<td>0.25</td>
<td>-0.39</td>
<td>-0.78</td>
<td>0.04</td>
<td>4611.94</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[46]</td>
<td>0.13</td>
<td>0.31</td>
<td>0.13</td>
<td>-0.36</td>
<td>0.64</td>
<td>5879.70</td>
<td>1.00</td>
</tr>
<tr>
<td>divorce_rate[47]</td>
<td>0.56</td>
<td>0.47</td>
<td>0.56</td>
<td>-0.15</td>
<td>1.37</td>
<td>4319.38</td>
<td>1.00</td>
</tr>
</tbody>
</table>
### 7.4.1 Effect of Incorporating Measurement Noise on Residuals

Notice that our values for the regression coefficients is very similar to Model 3. However, introducing measurement noise allows us to more closely match our predictive distribution to the observed values. We can see this if we plot the residuals as earlier.

```python
rng_key, rng_key_ = random.split(rng_key)
predictions_4 = Predictive(model_se, samples_4)(
    rng_key_,
    marriage=dset.MarriageScaled.values,
    age=dset.AgeScaled.values,
    divorce_sd=dset.DivorceScaledSD.values,
)['obs']
```

```python
sd = dset.DivorceScaledSD.values
residuals_4 = dset.DivorceScaled.values - predictions_4
residuals_mean = jnp.mean(residuals_4, axis=0)
residuals_hpdi = hpdi(residuals_4, 0.9)
err = residuals_hpdi[1] - residuals_mean
idx = jnp.argsort(residuals_mean)
y = jnp.arange(50)
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 16))

# Plot Residuals
ax.plot(jnp.zeros(50), y, "--")
ax.errorbar(residuals_mean[idx], y, xerr=err[idx], marker="o", ms=5, mew=4, ls="none", alpha=0.8)

# Plot SD
ax.errorbar(residuals_mean[idx], y, xerr=sd[idx], ls="none", color="orange", alpha=0.9)

# Plot earlier mean residual
ax.plot(jnp.mean(dset.DivorceScaled.values - predictions_3, 0)[idx], y, ls="none",
        marker="o", ms=6, color="black",
        alpha=0.6)
```

Number of divergences: 0
ax.set(xlabel="Residuals", ylabel="State", title="Residuals with 90% CI")
ax.set_yticks(y)
ax.set_yticklabels(dset.Loc.values[idx], fontsize=10)
ax.text(-2.8, -7, "Residuals (with error-bars) from current model (in red). "
"Black marker shows residuals from the previous model (Model 3). "
"Measurement error is indicated by orange bar.",);
Residuals with 90% CI

Residuals (with errorbars) from current model (in red). Black marker shows residuals from the previous model (Model 2). Measurement error is indicated by orange bar.

7.4. Regression Model with Measurement Error
The plot above shows the residuals for each of the states, along with the measurement noise given by inner error bar. The gray dots are the mean residuals from our earlier Model 3. Notice how having an additional degree of freedom to model the measurement noise has shrunk the residuals. In particular, for Idaho and Maine, our predictions are now much closer to the observed values after incorporating measurement noise in the model.

To better see how measurement noise affects the movement of the regression line, let us plot the residuals with respect to the measurement noise.

```python
[29]: fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10, 6))
x = dset.DivorceScaledSD.values
y1 = jnp.mean(residuals_3, 0)
y2 = jnp.mean(residuals_4, 0)
ax.plot(x, y1, ls="none", marker="o")
ax.plot(x, y2, ls="none", marker="o")
for i, (j, k) in enumerate(zip(y1, y2)):
    ax.plot([x[i], x[i]], [j, k], "--", color="gray")

ax.set(
    xlabel="Measurement Noise",
    ylabel="Residual",
    title="Mean residuals (Model 4: red, Model 3: blue)",
);
```

The plot above shows what has happened in more detail - the regression line itself has moved to ensure a better fit for observations with low measurement noise (left of the plot) where the residuals have shrunk very close to 0. That is to say that data points with low measurement error have a concomitantly higher contribution in determining the regression line. On the other hand, for states with high measurement error (right of the plot), incorporating measurement noise allows us to move our posterior distribution mass closer to the observations resulting in a shrinkage of residuals as well.
7.5 References

7. JAX Development Team (2018). Composable transformations of Python+NumPy programs: differentiate, vectorize, JIT to GPU/TPU, and more
8. Gelman, A., Hwang, J., and Vehtari A. Understanding predictive information criteria for Bayesian models
Probabilistic Machine Learning models can not only make predictions about future data, but also model uncertainty. In areas such as personalized medicine, there might be a large amount of data, but there is still a relatively small amount of data for each patient. To customize predictions for each person it becomes necessary to build a model for each person — with its inherent uncertainties — and to couple these models together in a hierarchy so that information can be borrowed from other similar people [1].

The purpose of this tutorial is to demonstrate how to implement a Bayesian Hierarchical Linear Regression model using NumPyro. To motivate the tutorial, I will use OSIC Pulmonary Fibrosis Progression competition, hosted at Kaggle.

### 8.1 1. Understanding the task

Pulmonary fibrosis is a disorder with no known cause and no known cure, created by scarring of the lungs. In this competition, we were asked to predict a patient’s severity of decline in lung function. Lung function is assessed based on output from a spirometer, which measures the forced vital capacity (FVC), i.e. the volume of air exhaled.

In medical applications, it is useful to evaluate a model's confidence in its decisions. Accordingly, the metric used to rank the teams was designed to reflect both the accuracy and certainty of each prediction. It’s a modified version of the Laplace Log Likelihood (more details on that later).

Let’s explore the data and see what’s that all about:

```python
[1]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro arviz

[2]: import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

[3]: train = pd.read_csv(
    "https://gist.githubusercontent.com/ucals/
    "2cf9d101992cb1b78c2cdd6e3bac6a4b/raw/
    "43034c39052d5974b894d2ec1bc3f90f3623d9/
    "osic_pulmonary_fibrosis.csv"
)
train.head()
```
In the dataset, we were provided with a baseline chest CT scan and associated clinical information for a set of patients. A patient has an image acquired at time Week = 0 and has numerous follow up visits over the course of approximately 1-2 years, at which time their FVC is measured. For this tutorial, I will use only the Patient ID, the weeks and the FVC measurements, discarding all the rest. Using only these columns enabled our team to achieve a competitive score, which shows the power of Bayesian hierarchical linear regression models especially when gauging uncertainty is an important part of the problem.

Since this is real medical data, the relative timing of FVC measurements varies widely, as shown in the 3 sample patients below:

```python
[4]:
def chart_patient(patient_id, ax):
    data = train[train["Patient"] == patient_id]
    x = data["Weeks"]
    y = data["FVC"]
    ax.set_title(patient_id)
    sns.regplot(x=x, y=y, ax=ax, ci=None, line_kws={"color": "red"})
```

On average, each of the 176 provided patients made 9 visits, when FVC was measured. The visits happened in specific weeks in the [-12, 133] interval. The decline in lung capacity is very clear. We see, though, they are very different from patient to patient.

We were are asked to predict every patient’s FVC measurement for every possible week in the [-12, 133] interval, and the confidence for each prediction. In other words: we were asked fill a matrix like the one below, and provide a confidence score for each prediction:
The task was perfect to apply Bayesian inference. However, the vast majority of solutions shared by Kaggle community used discriminative machine learning models, disconsidering the fact that most discriminative methods are very poor at providing realistic uncertainty estimates. Because they are typically trained in a manner that optimizes the parameters to minimize some loss criterion (e.g. the predictive error), they do not, in general, encode any uncertainty in either their parameters or the subsequent predictions. Though many methods can produce uncertainty estimates either as a by-product or from a post-processing step, these are typically heuristic based, rather than stemming naturally from a statistically principled estimate of the target uncertainty distribution [2].

8.2 2. Modelling: Bayesian Hierarchical Linear Regression with Partial Pooling

The simplest possible linear regression, not hierarchical, would assume all FVC decline curves have the same $\alpha$ and $\beta$. That’s the pooled model. In the other extreme, we could assume a model where each patient has a personalized FVC decline curve, and these curves are completely unrelated. That’s the unpooled model, where each patient has completely separate regressions.

Here, I’ll use the middle ground: Partial pooling. Specifically, I’ll assume that while $\alpha$’s and $\beta$’s are different for each patient as in the unpooled case, the coefficients all share similarity. We can model this by assuming that each individual coefficient comes from a common group distribution. The image below represents this model graphically:
Mathematically, the model is described by the following equations:

\[
\begin{align*}
\mu_\alpha &\sim \text{Normal}(0, 500) \\
\sigma_\alpha &\sim \text{Half-Normal}(100) \\
\mu_\beta &\sim \text{Normal}(0, 3) \\
\sigma_\beta &\sim \text{Half-Normal}(3) \\
\alpha_i &\sim \text{Normal}(\mu_\alpha, \sigma_\alpha) \\
\beta_i &\sim \text{Normal}(\mu_\beta, \sigma_\beta) \\
\sigma &\sim \text{Half-Normal}(100) \\
FVC_{ij} &\sim \text{Normal}(\alpha_i + t\beta_i, \sigma)
\end{align*}
\]

where \( t \) is the time in weeks. Those are very uninformative priors, but that’s ok: our model will converge!

Implementing this model in NumPyro is pretty straightforward:

```python
from jax import random
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive

assert numpyro.__version__.startswith("0.15.2")
```

where \( t \) is the time in weeks. Those are very uninformative priors, but that’s ok: our model will converge!

Implementing this model in NumPyro is pretty straightforward:
```python
[6]: def model(patient_code, Weeks, FVC_obs=None):
    \(\mu_α\) = numpyro.sample("\mu_α", dist.Normal(0.0, 500.0))
    \(\sigma_α\) = numpyro.sample("\sigma_α", dist.HalfNormal(100.0))
    \(\mu_β\) = numpyro.sample("\mu_β", dist.Normal(0.0, 3.0))
    \(\sigma_β\) = numpyro.sample("\sigma_β", dist.HalfNormal(3.0))

    n_patients = len(np.unique(patient_code))

    with numpyro.plate("plate_i", n_patients):
        \(\alpha\) = numpyro.sample("\alpha", dist.Normal(\(\mu_α\), \(\sigma_α\)))
        \(\beta\) = numpyro.sample("\beta", dist.Normal(\(\mu_β\), \(\sigma_β\)))

    \(\sigma\) = numpyro.sample("\sigma", dist.HalfNormal(100.0))
    FVC_est = \(\alpha\)[patient_code] + \(\beta\)[patient_code] * Weeks

    with numpyro.plate("data", len(patient_code)):
        numpyro.sample("obs", dist.Normal(FVC_est, \(\sigma\)), obs=FVC_obs)

That's all for modelling!

8.3 3. Fitting the model

A great achievement of Probabilistic Programming Languages such as NumPyro is to decouple model specification and inference. After specifying my generative model, with priors, condition statements and data likelihood, I can leave the hard work to NumPyro’s inference engine.

Calling it requires just a few lines. Before we do it, let’s add a numerical Patient ID for each patient code. That can be easily done with scikit-learn’s LabelEncoder:

```python
[7]: from sklearn.preprocessing import LabelEncoder

    patient_encoder = LabelEncoder()
    train["patient_code"] = patient_encoder.fit_transform(train["Patient"].values)

    FVC_obs = train["FVC"].values
    Weeks = train["Weeks"].values
    patient_code = train["patient_code"].values

Now, calling NumPyro’s inference engine:

```python
[8]: nuts_kernel = NUTS(model)

    mcmc = MCMC(nuts_kernel, num_samples=2000, num_warmup=2000)
    rng_key = random.PRNGKey(0)
    mcmc.run(rng_key, patient_code, Weeks, FVC_obs=FVC_obs)

    posterior_samples = mcmc.get_samples()
```

8.3. 3. Fitting the model 293
8.4 4. Checking the model

8.4.1 4.1. Inspecting the learned parameters

First, let's inspect the parameters learned. To do that, I will use ArviZ, which perfectly integrates with NumPyro:

```python
[9]: import arviz as az
data = az.from_numpyro(mcmc)
az.plot_trace(data, compact=True, figsize=(15, 25));
```
8.4. 4. Checking the model

![Graphs and plots related to model checking in NumPyro](image)
NumPyro Documentation

Looks like our model learned personalized alphas and betas for each patient!

### 8.4.2 4.2. Visualizing FVC decline curves for some patients

Now, let's visually inspect FVC decline curves predicted by our model. We will completely fill in the FVC table, predicting all missing values. The first step is to create a table to fill:

```python
[10]: def create_prediction_template(unique_patient_df, weeks_series):
    unique_patient_df["_temp"] = True
    weeks = pd.DataFrame(weeks_series, columns=["Weeks"])
    weeks["_temp"] = True
    return unique_patient_df.merge(weeks, on="_temp").drop(["_temp"], axis=1)
```

```python
[11]: patients = train["Patient", "patient_code"].drop_duplicates()
    start_week_number = -12
    end_week_number = 134
    predict_weeks = pd.Series(np.arange(start_week_number, end_week_number))
    pred_template = create_prediction_template(patients, predict_weeks)
```

Predicting the missing values in the FVC table and confidence (sigma) for each value becomes really easy:

```python
[12]: patient_code = pred_template["patient_code"].values
    Weeks = pred_template["Weeks"].values
    predictive = Predictive(model, posterior_samples, return_sites=["σ", "obs"])
    samples_predictive = predictive(random.PRNGKey(0), patient_code, Weeks, None)
```

Let's now put the predictions together with the true values, to visualize them:

```python
[13]: df = pred_template.copy()
    df["FVC_pred"] = samples_predictive["obs"].T.mean(axis=1)
    df["sigma"] = samples_predictive["obs"].T.std(axis=1)
    df["FVC_inf"] = df["FVC_pred"] - df["sigma"]
    df["FVC_sup"] = df["FVC_pred"] + df["sigma"]
    df = pd.merge(
        df, train[["Patient", "Weeks", "FVC"]], how="left", on=["Patient", "Weeks"]
    )
    df = df.rename(columns={"FVC": "FVC_true"})
    df.head()
```

```
<table>
<thead>
<tr>
<th>Patient</th>
<th>patient_code</th>
<th>Weeks</th>
<th>FVC_pred</th>
<th>sigma</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID0000763720177411956430</td>
<td>0</td>
<td>0</td>
<td>2226.545166</td>
<td>160.158493</td>
</tr>
<tr>
<td>ID0000763720177411956430</td>
<td>0</td>
<td>1</td>
<td>2216.172852</td>
<td>160.390778</td>
</tr>
<tr>
<td>ID0000763720177411956430</td>
<td>0</td>
<td>-10</td>
<td>2219.136963</td>
<td>155.339615</td>
</tr>
<tr>
<td>ID0000763720177411956430</td>
<td>0</td>
<td>-9</td>
<td>2214.727051</td>
<td>153.333313</td>
</tr>
<tr>
<td>ID0000763720177411956430</td>
<td>0</td>
<td>-8</td>
<td>2208.758545</td>
<td>157.368637</td>
</tr>
</tbody>
</table>
```

Finally, let's see our predictions for 3 patients:
def chart_patient_with_predictions(patient_id, ax):
    data = df[df["Patient"] == patient_id]
    x = data["Weeks"]
    ax.set_title(patient_id)
    ax.plot(x, data["FVC_true"], "o")
    ax.plot(x, data["FVC_pred"])
    ax = sns.regplot(x=x, y=data["FVC_true"], ax=ax, ci=None, line_kws={"color": "red"})
    ax.fill_between(x, data["FVC_inf"], data["FVC_sup"], alpha=0.5, color="#ffcd3c")
    ax.set_ylabel("FVC")

f, axes = plt.subplots(1, 3, figsize=(15, 5))
chart_patient_with_predictions("ID00007637202177411956430", axes[0])
chart_patient_with_predictions("ID00009637202177434476278", axes[1])
chart_patient_with_predictions("ID00011637202177653955184", axes[2])

The results are exactly what we expected to see! Highlight observations:

- The model adequately learned Bayesian Linear Regressions! The orange line (learned predicted FVC mean) is very inline with the red line (deterministic linear regression). But most important: it learned to predict uncertainty, showed in the light orange region (one sigma above and below the mean FVC line)
- The model predicts a higher uncertainty where the data points are more disperse (1st and 3rd patients). Conversely, where the points are closely grouped together (2nd patient), the model predicts a higher confidence (narrower light orange region)
- Finally, in all patients, we can see that the uncertainty grows as the look more into the future: the light orange region widens as the # of weeks grow!

### 8.4.3 4.3. Computing the modified Laplace Log Likelihood and RMSE

As mentioned earlier, the competition was evaluated on a modified version of the Laplace Log Likelihood. In medical applications, it is useful to evaluate a model’s confidence in its decisions. Accordingly, the metric is designed to reflect both the accuracy and certainty of each prediction.

For each true FVC measurement, we predicted both an FVC and a confidence measure (standard deviation $\sigma$). The metric was computed as:
\[\sigma_{\text{clipped}} = \max(\sigma, 70)\] \hspace{1cm} (8.9)

\[\delta = \min(|FVC_{\text{true}} - FVC_{\text{pred}}|, 1000)\] \hspace{1cm} (8.10)

\[\text{metric} = -\frac{\sqrt{2}\delta}{\sigma_{\text{clipped}}} - \ln(\sqrt{2}\sigma_{\text{clipped}})\] \hspace{1cm} (8.11)

The error was thresholded at 1000 ml to avoid large errors adversely penalizing results, while the confidence values were clipped at 70 ml to reflect the approximate measurement uncertainty in FVC. The final score was calculated by averaging the metric across all (Patient, Week) pairs. Note that metric values will be negative and higher is better.

Next, we calculate the metric and RMSE:

```python
[15]: y = df.dropna()
rmse = ((y["FVC_pred"] - y["FVC_true"])**2).mean()**(1/2)
print(f"RMSE: {rmse:.1f} ml")

sigma_c = y["sigma"].values
sigma_c[sigma_c < 70] = 70
delta = (y["FVC_pred"] - y["FVC_true"]).abs()
delta[delta > 1000] = 1000
lll = -np.sqrt(2) * delta / sigma_c - np.log(np.sqrt(2) * sigma_c)
print(f"Laplace Log Likelihood: {lll.mean():.4f}")
```

RMSE: 122.3 ml
Laplace Log Likelihood: -6.1406

What do these numbers mean? It means if you adopted this approach, you would outperform most of the public solutions in the competition. Curiously, the vast majority of public solutions adopt a standard deterministic Neural Network, modelling uncertainty through a quantile loss. Most of the people still adopt a frequentist approach.

Uncertainty for single predictions becomes more and more important in machine learning and is often a requirement. Especially when the consequences of a wrong prediction are high, we need to know what the probability distribution of an individual prediction is. For perspective, Kaggle just launched a new competition sponsored by Lyft, to build motion prediction models for self-driving vehicles. “We ask that you predict a few trajectories for every agent and provide a confidence score for each of them.”

### 8.5 5. Add layer to model hierarchy: Smoking Status

We can extend the model by including the column `SmokingStatus` as a pooling level, where model parameters will be partially pooled by the groups “Never smoked”, “Ex-smoker”, and “Currently smokes”. To do this, we need to:

1. Encode the `SmokingStatus` column
2. Map patient encoding to smoking status encodings
3. Refine and retrain the model with the additional hierarchy

```python
[16]: train["SmokingStatus"].value_counts()
[16]:
Ex-smoker    1038
Never smoked 429
Currently smokes 82
Name: SmokingStatus, dtype: int64
```
[17]:
patient_code = train["patient_code"].values
Weeks = train["Weeks"].values

[18]:
smoking_status_encoder = LabelEncoder()
train["smoking_status_code"] = smoking_status_encoder.fit_transform(
    train["SmokingStatus"]
)
smoking_status_code = train["smoking_status_code"].values

[19]:
map_patient_to_smoking_status = (
    train[["patient_code", "smoking_status_code"]]
    .drop_duplicates()
    .set_index("patient_code", verify_integrity=True)
    .sort_index()["smoking_status_code"]
    .values
)

[20]:
def model_smoking_hierarchy(
    patient_code, Weeks, map_patient_to_smoking_status, FVC_obs=None
):
    μ_α_global = numpyro.sample("μ_α_global", dist.Normal(0.0, 500.0))
    σ_α_global = numpyro.sample("σ_α_global", dist.HalfNormal(100.0))
    μ_β_global = numpyro.sample("μ_β_global", dist.Normal(0.0, 3.0))
    σ_β_global = numpyro.sample("σ_β_global", dist.HalfNormal(3.0))
    n_patients = len(np.unique(patient_code))
    n_smoking_statuses = len(np.unique(map_patient_to_smoking_status))
    with numpyro.plate("plate_smoking_status", n_smoking_statuses):
        μ_α_smoking_status = numpyro.sample("μ_α_smoking_status",
            dist.Normal(μ_α_global, σ_α_global)
        )
        μ_β_smoking_status = numpyro.sample("μ_β_smoking_status",
            dist.Normal(μ_β_global, σ_β_global)
        )
    with numpyro.plate("plate_i", n_patients):
        α = numpyro.sample("α",
            dist.Normal(μ_α_smoking_status[map_patient_to_smoking_status], σ_α_global),
        )
        β = numpyro.sample("β",
            dist.Normal(μ_β_smoking_status[map_patient_to_smoking_status], σ_β_global),
        )
    σ = numpyro.sample("σ", dist.HalfNormal(100.0))
    FVC_est = α[patient_code] + β[patient_code] * Weeks
    with numpyro.plate("data", len(patient_code)):
        numpyro.sample("obs", dist.Normal(FVC_est, σ), obs=FVC_obs)
8.5.1 Reparameterize the model

Hierarchical models often need to be reparameterized to enable MCMC to explore the full parameter space. NumPyro’s LocScaleReparam is used to do this below. For more details, see bad_posterior_geometry.ipynb and funnel.py. Thomas Wiecki also has a great post about developing non-centered models.

```python
from numpyro.handlers import reparam
from numpyro.infer.reparam import LocScaleReparam

reparam_config = {
    "μ_α_smoking_status": LocScaleReparam(0),
    "μ_β_smoking_status": LocScaleReparam(0),
    "α": LocScaleReparam(0),
    "β": LocScaleReparam(0),
}
reparam_model_smoking_hierarchy = reparam(
    model_smoking_hierarchy, config=reparam_config
)
```

```python
nuts_kernel = NUTS(reparam_model_smoking_hierarchy, target_accept_prob=0.97)
```

```python
mcmc = MCMC(nuts_kernel, num_samples=3000, num_warmup=5000)
rng_key = random.PRNGKey(0)
mcmc.run(rng_key, patient_code, Weeks, map_patient_to_smoking_status, FVC_obs=FVC_obs)
posterior_samples = mcmc.get_samples()
```

8.5.2 5.1. Inspect the learned parameters

```python
data = az.from_numpyro(mcmc)
az.plot_trace(data, compact=True, figsize=(15, 45));
```
8.5. 5. Add layer to model hierarchy: Smoking Status
Smoking Status distributions

Adding a legend for the smoking status distributions to help interpret the model results for that level.

<table>
<thead>
<tr>
<th>Smoking Status</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Currently smokes</td>
<td>0</td>
</tr>
<tr>
<td>Ex-smoker</td>
<td>1</td>
</tr>
<tr>
<td>Never smoked</td>
<td>2</td>
</tr>
</tbody>
</table>

```
[24]: # Check the label code for each SmokingStatus
    smoking_status_encoder.inverse_transform([0, 1, 2])
[24]: array(['Currently smokes', 'Ex-smoker', 'Never smoked'], dtype=object)

[25]: axes = az.plot_trace(
    data,
    var_names=['μ_α_smoking_status', 'μ_β_smoking_status'],
    legend=True,
    compact=True,
    figsize=(15, 15),
)
# The legend handles were not working for the first plot
axes[0, 0].legend();
```

WARNING:matplotlib.legend:No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
8.5.3 Interpret smoking status model parameters

The model parameters for each smoking status show interesting results, especially for trend, $\mu_{\beta_{\text{smoking status}}}$.

In the trace plots above and summary table below the trend for current smokers, $\mu_{\beta_{\text{smoking status}}[0]}$, has a positive mean, whereas the trend for patients that are ex-smokers and those that have never smoked are negative, $\mu_{\beta_{\text{smoking status}}[1]}$ and $\mu_{\beta_{\text{smoking status}}[2]}$.

```python
[26]: trace = az.from_numpyro(mcmc)
az.summary(
    trace,
    var_names=['$\mu_{\alpha_{\text{global}}}$', '$\mu_{\beta_{\text{global}}}$', '$\mu_{\alpha_{\text{smoking status}}}$', '$\mu_{\beta_{\text{smoking status}}}$'],
    (continues on next page)
```
Shape validation failed: input_shape: (1, 3000), minimum_shape: (chains=2, draws=4)

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>sd</th>
<th>hdi_3%</th>
<th>hdi_97%</th>
<th>mcse_mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>μ_α_global</td>
<td>1660.2</td>
<td>309.7</td>
<td>1118.0</td>
<td>2274.9</td>
<td>6.6</td>
</tr>
<tr>
<td>μ_β_global</td>
<td>-1.25</td>
<td>2.06</td>
<td>-5.01</td>
<td>2.67</td>
<td>0.004</td>
</tr>
<tr>
<td>μ_α_smoking_status[0]</td>
<td>2970.4</td>
<td>222.7</td>
<td>2572.9</td>
<td>3429.3</td>
<td>7.7</td>
</tr>
<tr>
<td>μ_α_smoking_status[1]</td>
<td>2907.9</td>
<td>68.0</td>
<td>2782.9</td>
<td>3035.1</td>
<td>2.0</td>
</tr>
<tr>
<td>μ_α_smoking_status[2]</td>
<td>2475.3</td>
<td>102.9</td>
<td>2286.0</td>
<td>2671.2</td>
<td>6.1</td>
</tr>
<tr>
<td>μ_β_smoking_status[0]</td>
<td>0.26</td>
<td>1.71</td>
<td>-1.28</td>
<td>5.07</td>
<td>0.032</td>
</tr>
<tr>
<td>μ_β_smoking_status[1]</td>
<td>-4.6</td>
<td>0.49</td>
<td>-5.56</td>
<td>-3.72</td>
<td>0.010</td>
</tr>
<tr>
<td>μ_β_smoking_status[2]</td>
<td>-4.51</td>
<td>0.79</td>
<td>-6.01</td>
<td>-3.05</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Let's look at these curves for individual patients to help interpret these model results.

### 8.5.4 5.2. Visualizing FVC decline curves for some patients

```python
[27]:
    patient_code = pred_template["patient_code"].values
    Weeks = pred_template["Weeks"].values
    predictive = Predictive(
        reparam_model_smoking_hierarchy, posterior_samples, return_sites=["σ", "obs"]
    )
    samples_predictive = predictive(
        random.PRNGKey(0), patient_code, Weeks, map_patient_to_smoking_status, None
    )

[28]:
    df = pred_template.copy()
    df["FVC_pred"] = samples_predictive["obs"].T.mean(axis=1)
    df["sigma"] = samples_predictive["obs"].T.std(axis=1)
    df["FVC_inf"] = df["FVC_pred"] - df["sigma"]
    df["FVC_sup"] = df["FVC_pred"] + df["sigma"]
    df = pd.merge(
        df, train["Patient", "Weeks", "FVC"], how="left", on=["Patient", "Weeks"]
    )
    df = df.rename(columns={"FVC": "FVC_true"})
    df.head()
```

```
<table>
<thead>
<tr>
<th>Patient</th>
<th>patient_code</th>
<th>Weeks</th>
<th>FVC_pred</th>
<th>sigma</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ID000007637202177411956430</td>
<td>0</td>
<td>-12</td>
<td>2229.098877 157.88753</td>
</tr>
<tr>
<td>1</td>
<td>ID000007637202177411956430</td>
<td>0</td>
<td>-11</td>
<td>2225.022461 157.358429</td>
</tr>
<tr>
<td>2</td>
<td>ID000007637202177411956430</td>
<td>0</td>
<td>-10</td>
<td>2224.487549 155.416016</td>
</tr>
</tbody>
</table>
```

(continues on next page)
8.5.5 Review patients that currently smoke

By plotting each patient with the smoking status “Currently smokes”, we see some patients with a clear positive trend and others without a clear trend or negative trend. The trend lines are less overfit than the unpooled trend lines and show relatively large uncertainty in the slope and intercept. Depending on the model use case, we could proceed in different ways:

- If we just wanted to get an understanding of different attributes as they relate to patient’s FVC over time, we could stop here with an understanding that current smokers might have an increase in FVC over time when being monitored for Pulmonary Fibrosis. We might hypothesize causes for this observation to design a new experiment to test that hypothesis.

- If we wanted to develop a model that generates predictions used to treat patients, then we will want to make sure we are not overfitting so that we can trust the model with new patients. We might adjust model parameters to shrink the “Currently smokes” group model parameters to be closer to global parameters or even combine the group with “Ex-smokers”. We could look into collecting more data for current smokers to help ensure the model is not overfitting.

[30]: f, axes = plt.subplots(3, 3, figsize=(15, 15))
for i, patient in enumerate(train[train["SmokingStatus"] == "Currently smokes"]["Patient"].unique()):

(continues on next page)
8.5.6 5.3 Modified Laplace Log Likelihood and RMSE for model with Smoking Status Level

We calculate the metrics for the updated model and compare to the original model.

```python
[31]: y = df.dropna()
rmse = ((y["FVC_pred"] - y["FVC_true"]) ** 2).mean() ** (1 / 2)
print(f"RMSE: {rmse:.1f} ml")
```
Both the Laplace Log Likelihood and RMSE show slightly worse performance for the smoking status model. We’ve learned that adding this hierarchy level as-is did not improve model performance, but we did find some interested results from the smoking status level that might be worth investigating. In addition, we might try to adjust priors or trying a different level (e.g. gender) to improve model performance.

### 8.6 Wrap-up

Finally, I hope the great work done by Pyro/NumPyro developers help democratize Bayesian methods, empowering an ever growing community of researchers and practitioners to create models that can not only generate predictions, but also assess uncertainty in their predictions.

### 8.7 References


CHAPTER NINE

EXAMPLE: BASEBALL BATTING AVERAGE

Original example from Pyro: https://github.com/pyro-ppl/pyro/blob/dev/examples/baseball.py

Example has been adapted from [1]. It demonstrates how to do Bayesian inference using various MCMC kernels in Pyro (HMC, NUTS, SA), and use of some common inference utilities.

As in the Stan tutorial, this uses the small baseball dataset of Efron and Morris [2] to estimate players’ batting average which is the fraction of times a player got a base hit out of the number of times they went up at bat.

The dataset separates the initial 45 at-bats statistics from the remaining season. We use the hits data from the initial 45 at-bats to estimate the batting average for each player. We then use the remaining season’s data to validate the predictions from our models.

Three models are evaluated:

- Complete pooling model: The success probability of scoring a hit is shared amongst all players.
- No pooling model: Each individual player’s success probability is distinct and there is no data sharing amongst players.
- Partial pooling model: A hierarchical model with partial data sharing.

We recommend Radford Neal’s tutorial on HMC ([3]) to users who would like to get a more comprehensive understanding of HMC and its variants, and to [4] for details on the No U-Turn Sampler, which provides an efficient and automated way (i.e. limited hyper-parameters) of running HMC on different problems.

Note that the Sample Adaptive (SA) kernel, which is implemented based on [5], requires large num_warmup and num_samples (e.g. 15,000 and 300,000). So it is better to disable progress bar to avoid dispatching overhead.

References:


```python
import argparse
import os
import jax.numpy as jnp
import jax.random as random
from jax.scipy.special import logsumexp
```
import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import BASEBALL, load_dataset
from numpyro.infer import HMC, MCMC, NUTS, SA, Predictive, log_likelihood

def fully_pooled(at_bats, hits=None):
    r""
    Number of hits in $K$ at bats for each player has a Binomial
    distribution with a common probability of success, $\phi$.

    :param (jnp.ndarray) at_bats: Number of at bats for each player.
    :param (jnp.ndarray) hits: Number of hits for the given at bats.
    :return: Number of hits predicted by the model.
    ""
    phi_prior = dist.Uniform(0, 1)
    phi = numpyro.sample("phi", phi_prior)
    num_players = at_bats.shape[0]
    with numpyro.plate("num_players", num_players):
        return numpyro.sample("obs", dist.Binomial(at_bats, probs=phi), obs=hits)

def not_pooled(at_bats, hits=None):
    r""
    Number of hits in $K$ at bats for each player has a Binomial
    distribution with independent probability of success, $\phi_i$.

    :param (jnp.ndarray) at_bats: Number of at bats for each player.
    :param (jnp.ndarray) hits: Number of hits for the given at bats.
    :return: Number of hits predicted by the model.
    ""
    num_players = at_bats.shape[0]
    with numpyro.plate("num_players", num_players):
        phi_prior = dist.Uniform(0, 1)
        phi = numpyro.sample("phi", phi_prior)
        return numpyro.sample("obs", dist.Binomial(at_bats, probs=phi), obs=hits)

def partially_pooled(at_bats, hits=None):
    r""
    Number of hits has a Binomial distribution with independent
    probability of success, $\phi_i$. Each $\phi_i$ follows a Beta
    distribution with concentration parameters $c_1$ and $c_2$, where
    $c_1 = m * kappa$, $c_2 = (1 - m) * kappa$, $m \sim Uniform(0, 1)$,
    and $kappa \sim Pareto(1, 1.5)$.

    :param (jnp.ndarray) at_bats: Number of at bats for each player.
    :param (jnp.ndarray) hits: Number of hits for the given at bats.
    :return: Number of hits predicted by the model.
    ""
    m = numpyro.sample("m", dist.Uniform(0, 1))
kappa = numpyro.sample("kappa", dist.Pareto(1, 1.5))
num_players = at_bats.shape[0]
with numpyro.plate("num_players", num_players):
    phi_prior = dist.Beta(m * kappa, (1 - m) * kappa)
    phi = numpyro.sample("phi", phi_prior)
    return numpyro.sample("obs", dist.Binomial(at_bats, probs=phi), obs=hits)

def partially_pooled_with_logit(at_bats, hits=None):
    r""
    Number of hits has a Binomial distribution with a logit link function. The logits $\alpha$ for each player is normally distributed with the mean and scale parameters sharing a common prior.
    
    :param (jnp.ndarray) at_bats: Number of at bats for each player.
    :param (jnp.ndarray) hits: Number of hits for the given at bats.
    :return: Number of hits predicted by the model.
    ""
    loc = numpyro.sample("loc", dist.Normal(-1, 1))
    scale = numpyro.sample("scale", dist.HalfCauchy(1))
    num_players = at_bats.shape[0]
    with numpyro.plate("num_players", num_players):
        alpha = numpyro.sample("alpha", dist.Normal(loc, scale))
        return numpyro.sample("obs", dist.Binomial(at_bats, logits=alpha), obs=hits)

def run_inference(model, at_bats, hits, rng_key, args):
    if args.algo == "NUTS":
        kernel = NUTS(model)
    elif args.algo == "HMC":
        kernel = HMC(model)
    elif args.algo == "SA":
        kernel = SA(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False
        if ("NUMPYRO_SPHINXBUILD" in os.environ or args.disable_progbar)
            else True,
    )
    mcmc.run(rng_key, at_bats, hits)
    return mcmc.get_samples()

def predict(model, at_bats, hits, z, rng_key, player_names, train=True):
    header = model.__name__ + (" - TRAIN" if train else " - TEST")
    predictions = Predictive(model, posterior_samples=z)(rng_key, at_bats)["obs"]
    print_results(
        "=" * 30 + header + "=" * 30, predictions, player_names, at_bats, hits
    )
if not train:
    post_loglik = log_likelihood(model, z, at_bats, hits)["obs"]
    # computes expected log predictive density at each data point
    exp_log_density = logsumexp(post_loglik, axis=0) - jnp.log(jnp.shape(post_loglik)[0])
    # reports log predictive density of all test points
    print("Log pointwise predictive density: {:.2f}\n".format(exp_log_density.sum()))

def print_results(header, preds, player_names, at_bats, hits):
    columns = ["", "At-bats", "ActualHits", "Pred(p25)", "Pred(p50)", "Pred(p75)"]
    header_format = "{:>20} {:>10} {:>10} {:>10} {:>10} {:>10}"
    row_format = "{:>20} {:>10.0f} {:>10.0f} {:>10.2f} {:>10.2f} {:>10.2f}"
    quantiles = jnp.quantile(preds, jnp.array([0.25, 0.5, 0.75]), axis=0)
    print("\n", header, "\n")
    print(header_format.format(*columns))
    for i, p in enumerate(player_names):
        print(row_format.format(p, at_bats[i], hits[i], *quantiles[:, i]), "\n")

def main(args):
    _, fetch_train = load_dataset(BASEBALL, split="train", shuffle=False)
    train, player_names = fetch_train()
    _, fetch_test = load_dataset(BASEBALL, split="test", shuffle=False)
    test, _ = fetch_test()
    at_bats, hits = train[:, 0], train[:, 1]
    season_at_bats, season_hits = test[:, 0], test[:, 1]
    for i, model in enumerate((fully_pooled, not_pooled, partially_pooled, partially_pooled_with_logit)):
        rng_key, rng_key_predict = random.split(random.PRNGKey(i + 1))
        zs = run_inference(model, at_bats, hits, rng_key, args)
        predict(model, at_bats, hits, zs, rng_key_predict, player_names)
        predict(model, season_at_bats, season_hits, zs, rng_key_predict, player_names, train=False,)

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Baseball batting average using MCMC")
    parser.add_argument("-n", "--num-samples", nargs="?", default=3000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1500, type=int)
parser.add_argument("--num-chains", nargs="?", default=1, type=int)
parser.add_argument(
    "--algo", default="NUTS", type=str, help='whether to run "HMC", "NUTS", or "SA"'
)
parser.add_argument(
    "-dp",
    "--disable-progbar",
    action="store_true",
    default=False,
    help="whether to disable progress bar",
)
parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu"."
args = parser.parse_args()
	numpyro.set_platform(args.device)
	numpyro.set_host_device_count(args.num_chains)

    main(args)
**EXAMPLE: VARIATIONAL AUTOENCODER**

```python
import argparse
import inspect
import os
import time

import matplotlib.pyplot as plt

from jax import jit, lax, random
from jax.example_libraries import stax
import jax.numpy as jnp
from jax.random import PRNGKey

import numpyro
from numpyro import optim
import numpyro.distributions as dist
from numpyro.examples.datasets import MNIST, load_dataset
from numpyro.infer import SVI, Trace_ELBO

RESULTS_DIR = os.path.abspath(os.path.join(os.path.dirname(inspect.getfile(lambda: None)), ".results"))
os.makedirs(RESULTS_DIR, exist_ok=True)

def encoder(hidden_dim, z_dim):
    return stax.serial(
        stax.Dense(hidden_dim, W_init=stax.randn()),
        stax.Softplus,
        stax.FanOut(2),
        stax.parallel(
            stax.Dense(z_dim, W_init=stax.randn()),
            stax.serial(stax.Dense(z_dim, W_init=stax.randn()), stax.Exp),
        ),
    )

def decoder(hidden_dim, out_dim):
    return stax.serial(
        stax.Dense(hidden_dim, W_init=stax.randn()),
        stax.Softplus,
```
stax.Dense(out_dim, W_init=stax.randn()),
   stax.Sigmoid,
)

def model(batch, hidden_dim=400, z_dim=100):
    batch = jnp.reshape(batch, (batch.shape[0], -1))
    batch_dim, out_dim = jnp.shape(batch)
    decode = numpyro.module("decoder", decoder(hidden_dim, out_dim), (batch_dim, z_dim))
    with numpyro.plate("batch", batch_dim):
        z = numpyro.sample("z", dist.Normal(0, 1).expand([z_dim]).to_event(1))
        img_loc = decode(z)
    return numpyro.sample("obs", dist.Bernoulli(img_loc).to_event(1), obs=batch)

def guide(batch, hidden_dim=400, z_dim=100):
    batch = jnp.reshape(batch, (batch.shape[0], -1))
    batch_dim, out_dim = jnp.shape(batch)
    encode = numpyro.module("encoder", encoder(hidden_dim, z_dim), (batch_dim, out_dim))
    z_loc, z_std = encode(batch)
    with numpyro.plate("batch", batch_dim):
        return numpyro.sample("z", dist.Normal(z_loc, z_std).to_event(1))

@jit
def binarize(rng_key, batch):
    return random.bernoulli(rng_key, batch).astype(batch.dtype)

def main(args):
    encoder_nn = encoder(args.hidden_dim, args.z_dim)
    decoder_nn = decoder(args.hidden_dim, 28 * 28)
    adam = optim.Adam(args.learning_rate)
    svi = SVI(
        model, guide, adam, Trace_ELBO(), hidden_dim=args.hidden_dim, z_dim=args.z_dim
    )
    rng_key = PRNGKey(0)
    train_init, train_fetch = load_dataset(
        MNIST, batch_size=args.batch_size, split="train"
    )
    test_init, test_fetch = load_dataset(
        MNIST, batch_size=args.batch_size, split="test"
    )
    num_train, train_idx = train_init()
    rng_key, rng_key_binarize, rng_key_init = random.split(rng_key, 3)
    sample_batch = binarize(rng_key_binarize, train_fetch(0, train_idx)[0])
    svi_state = svi.init(rng_key_init, sample_batch)

    @jit
def epoch_train(svi_state, rng_key, train_idx):
        def body_fn(i, val):
            loss_sum, svi_state = val
        (continues on next page)
rng_key_binarize = random.fold_in(rng_key, i)
batch = binarize(rng_key_binarize, train_fetch(i, train_idx)[0])
svi_state, loss = svi.update(svi_state, batch)
loss_sum += loss
return loss_sum, svi_state

return lax.fori_loop(0, num_train, body_fn, (0.0, svi_state))

@jit
def eval_test(svi_state, rng_key, test_idx):
    def body_fun(i, loss_sum):
        rng_key_binarize = random.fold_in(rng_key, i)
batch = binarize(rng_key_binarize, test_fetch(i, test_idx)[0])
        # FIXME: does this lead to a requirement for an rng_key arg in svi_eval?
        loss = svi.evaluate(svi_state, batch) / len(batch)
        loss_sum += loss
        return loss_sum

    loss = lax.fori_loop(0, num_test, body_fun, 0.0)
    loss = loss / num_test
    return loss

def reconstruct_img(epoch, rng_key):
    img = test_fetch(0, test_idx)[0][0]
    plt.imsave(os.path.join(RESULTS_DIR, "original_epoch=\{0\}.png".format(epoch)),
                img,
                cmap="gray",
            )
    rng_key_binarize, rng_key_sample = random.split(rng_key)
test_sample = binarize(rng_key_binarize, img)
params = svi.get_params(svi_state)
z_mean, z_var = encoder_nn[1](
    params["encoder$params"]
)
    test_sample.reshape([1, -1])
    )
z = dist.Normal(z_mean, z_var).sample(rng_key_sample)
img_loc = decoder_nn[1](params["decoder$params"], z).reshape([28, 28])
plt.imsave(os.path.join(RESULTS_DIR, "recons_epoch=\{0\}.png".format(epoch)),
                img_loc,
                cmap="gray",
            )

for i in range(args.num_epochs):
    rng_key, rng_key_train, rng_key_test, rng_key_reconstruct = random.split(
        rng_key, 4
    )
t_start = time.time()
num_train, train_idx = train_init()
_, svi_state = epoch_train(svi_state, rng_key_train, train_idx)
 rng_key, rng_key_test, rng_key_reconstruct = random.split(rng_key, 3)
num_test, test_idx = test_init()
test_loss = eval_test(svi_state, rng_key_test, test_idx)
reconstruct_img(i, rng_key_reconstruct)
print("Epoch {}: loss = {} ({} s.)".format(i, test_loss, time.time() - t_start)
      )
)

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="parse args")
    parser.add_argument("-n", "--num-epochs", default=15, type=int, help="number of training epochs")
    parser.add_argument("-lr", "--learning-rate", default=1.0e-3, type=float, help="learning rate")
    parser.add_argument("-batch-size", default=128, type=int, help="batch size")
    parser.add_argument("-z-dim", default=50, type=int, help="size of latent")
    parser.add_argument("-hidden-dim", default=400, type=int,
                        help="size of hidden layer in encoder/decoder networks")
    args = parser.parse_args()
    main(args)
EXAMPLE: NEAL’S FUNNEL

This example, which is adapted from [1], illustrates how to leverage non-centered parameterization using the `reparam` handler. We will examine the difference between two types of parameterizations on the 10-dimensional Neal’s funnel distribution. As we will see, HMC gets trouble at the neck of the funnel if centered parameterization is used. On the contrary, the problem can be solved by using non-centered parameterization.

Using non-centered parameterization through `LocScaleReparam` or `TransformReparam` in NumPyro has the same effect as the automatic reparameterisation technique introduced in [2].

References:


import argparse
import os
import matplotlib.pyplot as plt

from jax import random
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
from numpyro.handlers import reparam
from numpyro.infer import MCMC, NUTS, Predictive
from numpyro.infer.reparam import LocScaleReparam

(continues on next page)
```python
def model(dim=10):
    y = numpyro.sample("y", dist.Normal(0, 3))
    numpyro.sample("x", dist.Normal(jnp.zeros(dim - 1), jnp.exp(y / 2)))

reparam_model = reparam(model, config=\{"x": LocScaleReparam(0)\})

def run_inference(model, args, rng_key):
    kernel = NUTS(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key)
    mcmc.print_summary(exclude_deterministic=False)
    return mcmc

def main(args):
    rng_key = random.PRNGKey(0)

    # do inference with centered parameterization
    print("============================= Centered Parameterization␣
˓
⇒
==============================")
    mcmc = run_inference(model, args, rng_key)
    samples = mcmc.get_samples()
    diverging = mcmc.get_extra_fields()["diverging"]

    # do inference with non-centered parameterization
    print("\n\n=========================== Non-centered Parameterization␣
˓
⇒
============================")
    reparam_mcmc = run_inference(reparam_model, args, rng_key)
    reparam_samples = reparam_mcmc.get_samples()
    reparam_diverging = reparam_mcmc.get_extra_fields()["diverging"]

    # collect deterministic sites
    reparam_samples = Predictive(  
        reparam_model, reparam_samples, return_sites=["x", "y"]
    )(random.PRNGKey(1))

    # make plots
    fig, (ax1, ax2) = plt.subplots(  
        2, 1, sharex=True, figsize=(8, 8), constrained_layout=True
    )
```

(continues on next page)
ax1.plot(
samples["x"][-diverging, 0],
samples["y"][-diverging],
"o",
color="darkred",
alpha=0.3,
label="Non-diverging",
)
ax1.plot(
samples["x"][-diverging, 0],
samples["y"][-diverging],
"o",
color="lime",
label="Diverging",
)
ax1.set(
xlim=(-20, 20),
ylim=(-9, 9),
ylabel="y",
title="Funnel samples with centered parameterization",
)
ax1.legend()

ax2.plot(
reparam_samples["x"][-reparam_diverging, 0],
reparam_samples["y"][-reparam_diverging],
"o",
color="darkred",
alpha=0.3,
)
ax2.plot(
reparam_samples["x"][-reparam_diverging, 0],
reparam_samples["y"][-reparam_diverging],
"o",
color="lime",
)
ax2.set(
xlim=(-20, 20),
ylim=(-9, 9),
xlabel="x[0]",
ylabel="y",
title="Funnel samples with non-centered parameterization",
)

plt.savefig("funnel_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(
description="Non-centered reparameterization example"
)
parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
parser.add_argument("--num-chains", nargs="?", default=1, type=int)
parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu"."
args = parser.parse_args()

numpyro.set_platform(args.device)
numpyro.set_host_device_count(args.num_chains)

main(args)
EXAMPLE: STOCHASTIC VOLATILITY

Generative model:

\[
\begin{align*}
\sigma & \sim \text{Exponential}(50) \quad (12.1) \\
\nu & \sim \text{Exponential}(1) \quad (12.2) \\
s_i & \sim \text{Normal}(s_{i-1}, \sigma^{-2}) \quad (12.3) \\
r_i & \sim \text{StudentT}(\nu, 0, \exp(s_i)) \quad (12.4)
\end{align*}
\]

This example is from PyMC3 [1], which itself is adapted from the original experiment from [2]. A discussion about translating this in Pyro appears in [3].

We take this example to illustrate how to use the functional interface \texttt{hmc}. However, we recommend readers to use \texttt{MCMC} class as in other examples because it is more stable and has more features supported.

References:

import argparse
import os

import matplotlib
import matplotlib.dates as mdates
import matplotlib.pyplot as plt

import jax.numpy as jnp
import jax.random as random
import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import SP500, load_dataset
from numpyro.infer.hmc import hmc
from numpyro.infer.util import initialize_model
from numpyro.util import fori_collect

matplotlib.use("Agg")  # noqa: E402

def model(returns):
    step_size = numpyro.sample("sigma", dist.Exponential(50.0))
    s = numpyro.sample( (continues on next page)
"s", dist.GaussianRandomWalk(scale=step_size, num_steps=jnp.shape(returns)[0])
)

nu = numpyro.sample("nu", dist.Exponential(0.1))
return numpyro.sample(
  "r", dist.StudentT(df=nu, loc=0.0, scale=jnp.exp(s)), obs=returns
)

def print_results(posterior, dates):
    
def _print_row(values, row_name=""):  
        quantiles = jnp.array([0.2, 0.4, 0.5, 0.6, 0.8])  
        row_name_fmt = "{:>8}"  
        header_format = row_name_fmt + "{:>12}" * 5  
        row_format = row_name_fmt + "{:>12.3f}" * 5  
        columns = ["(p{{{}}})".format(int(q * 100)) for q in quantiles]  
        q_values = jnp.quantile(values, quantiles, axis=0)  
        print(header_format.format('', *columns))  
        print(row_format.format(row_name, *q_values))  
        print("\n")  
    
    print("=" * 20, "sigma", "=" * 20)  
    _print_row(posterior["sigma"])  
    print("=" * 20, "nu", "=" * 20)  
    _print_row(posterior["nu"])  
    print("=" * 20, "volatility", "=" * 20)  
    for i in range(0, len(dates), 180):  
        _print_row(jnp.exp(posterior["s"][:, i]), dates[i])


def main(args):
    _, fetch = load_dataset(SP500, shuffle=False)  
    dates, returns = fetch()  
    init_rng_key, sample_rng_key = random.split(random.PRNGKey(args.rng_seed))  
    model_info = initialize_model(init_rng_key, model, model_args=(returns,))  
    init_kernel, sample_kernel = hmc(model_info.potential_fn, algo="NUTS")  
    hmc_state = init_kernel(  
        model_info.param_info, args.num_warmup, rng_key=sample_rng_key  
    )  
    hmc_states = fori_collect(  
        args.num_warmup,  
        args.num_warmup + args.num_samples,  
        sample_kernel,  
        hmc_state,  
        transform=lambda hmc_state: model_info.postprocess_fn(hmc_state.z),  
        progbar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,  
    )  
    print_results(hmc_states, dates)

    fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)  
    dates = mdates.num2date(mdates.datestr2num(dates))  
    ax.plot(dates, returns, lw=0.5)  
    # format the ticks
ax.xaxis.set_major_locator(mdates.YearLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))
ax.xaxis.set_minor_locator(mdates.MonthLocator())

ax.plot(dates, jnp.exp(hmc_states["s"].T), "r", alpha=0.01)
legend = ax.legend(["returns", "volatility"], loc="upper right")
legend.legend_handles[1].set_alpha(0.6)
ax.set(xlabel="time", ylabel="returns", title="Volatility of S&P500 over time")

plt.savefig("stochastic_volatility_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Stochastic Volatility Model")
    parser.add_argument("-n", "--num-samples", nargs="?", default=600, type=int)
    parser.add_argument("--num-warmup", default=600, type=int)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    parser.add_argument("--rng_seed", default=21, type=int, help="random number generator seed")
    args = parser.parse_args()

    numpyro.set_platform(args.device)

    main(args)
In this example, we will follow [1] to implement the ProdLDA topic model from Autoencoding Variational Inference For Topic Models by Akash Srivastava and Charles Sutton [2]. This model returns consistently better topics than vanilla LDA and trains much more quickly. Furthermore, it does not require a custom inference algorithm that relies on complex mathematical derivations. This example also serves as an introduction to Flax and Haiku modules in NumPyro.

Note that unlike [1, 2], this implementation uses a Dirichlet prior directly rather than approximating it with a softmax-normal distribution.

For the interested reader, a nice extension of this model is the CombinedTM model [3] which utilizes a pre-trained sentence transformer (like https://www.sbert.net/) to generate a better representation of the encoded latent vector.

References:

import argparse

import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from wordcloud import WordCloud

import flax.linen as nn
import haiku as hk
import jax
from jax import device_put, random
import jax.numpy as jnp

(continues on next page)
import numpyro
from numpyro.contrib.module import flax_module, haiku_module
import numpyro.distributions as dist
from numpyro.infer import SVI, TraceMeanField_ELBO

class HaikuEncoder:
    def __init__(self, vocab_size, num_topics, hidden, dropout_rate):
        self._vocab_size = vocab_size
        self._num_topics = num_topics
        self._hidden = hidden
        self._dropout_rate = dropout_rate

    def __call__(self, inputs, is_training):
        dropout_rate = self._dropout_rate
        if is_training else 0.0
        h = jax.nn.softplus(hk.Linear(self._hidden)(inputs))
        h = jax.nn.softplus(hk.Linear(self._hidden)(h))
        h = hk.dropout(hk.next_rng_key(), dropout_rate, h)
        h = hk.Linear(self._num_topics)(h)

        # NB: here we set `create_scale=False` and `create_offset=False` to reduce
        # the number of learning parameters
        log_concentration = hk.BatchNorm(create_scale=False, create_offset=False, decay_rate=0.9)(h, is_training)
        return jnp.exp(log_concentration)

class HaikuDecoder:
    def __init__(self, vocab_size, dropout_rate):
        self._vocab_size = vocab_size
        self._dropout_rate = dropout_rate

    def __call__(self, inputs, is_training):
        dropout_rate = self._dropout_rate
        if is_training else 0.0
        h = hk.dropout(hk.next_rng_key(), dropout_rate, inputs)
        h = hk.Linear(self._vocab_size, with_bias=False)(h)
        return hk.BatchNorm(create_scale=False, create_offset=False, decay_rate=0.9)(h, is_training)

class FlaxEncoder(nn.Module):
    vocab_size: int
    num_topics: int
    hidden: int
    dropout_rate: float

    @nn.compact
    def __call__(self, inputs, is_training):
h = nn.softplus(nn.Dense(self.hidden)(inputs))
h = nn.softplus(nn.Dense(self.hidden)(h))
h = nn.Dropout(self.dropout_rate, deterministic=not is_training)(h)
h = nn.Dense(self.num_topics)(h)

log_concentration = nn.BatchNorm(
    use_bias=False,
    use_scale=False,
    momentum=0.9,
    use_running_average=not is_training,
    )(h)
return jnp.exp(log_concentration)

class FlaxDecoder(nn.Module):
    vocab_size: int
    dropout_rate: float

    @nn.compact
def __call__(self, inputs, is_training):
        h = nn.Dropout(self.dropout_rate, deterministic=not is_training)(inputs)
        h = nn.Dense(self.vocab_size, use_bias=False)(h)
        return nn.BatchNorm(
            use_bias=False,
            use_scale=False,
            momentum=0.9,
            use_running_average=not is_training,
            )(h)

def model(docs, hyperparams, is_training=False, nn_framework="flax"):
    if nn_framework == "flax":
        decoder = flax_module(
            "decoder",
            FlaxDecoder(hyperparams["vocab_size"], hyperparams["dropout_rate"]),
            input_shape=(1, hyperparams["num_topics"]),
            # ensure PRNGKey is made available to dropout layers
            apply_rng=["dropout"],
            # indicate mutable state due to BatchNorm layers
            mutable=["batch_stats"],
            # to ensure proper initialisation of BatchNorm we must
            # initialise with is_training=True
            is_training=True,
            )
    elif nn_framework == "haiku":
        decoder = haiku_module(
            "decoder",
            # use 'transform_with_state' for BatchNorm
            hk.transform_with_state(
                HaikuDecoder(hyperparams["vocab_size"], hyperparams["dropout_rate"])
            ),
            input_shape=(1, hyperparams["num_topics"]),
            )
apply_rng=True,
# to ensure proper initialisation of BatchNorm we must
# initialise with is_training=True
is_training=True,
)

else:
    raise ValueError(f"Invalid choice {nn_framework} for argument nn_framework")

with numpyro.plate(
    "documents", docs.shape[0], subsample_size=hyperparams["batch_size"]
):
    batch_docs = numpyro.subsample(docs, event_dim=1)
    theta = numpyro.sample(
        "theta", dist.Dirichlet(jnp.ones(hyperparams["num_topics"]))
    )

if nn_framework == "flax":
    logits = decoder(theta, is_training, rngs={"dropout": numpyro.prng_key()})
elif nn_framework == "haiku":
    logits = decoder(numpyro.prng_key(), theta, is_training)

total_count = batch_docs.sum(-1)
numpyro.sample(
    "obs", dist.Multinomial(total_count, logits=logits), obs=batch_docs
)

def guide(docs, hyperparams, is_training=False, nn_framework="flax"):
    if nn_framework == "flax":
        encoder = flax_module(
            "encoder",
            FlaxEncoder(
                hyperparams["vocab_size"],
                hyperparams["num_topics"],
                hyperparams["hidden"],
                hyperparams["dropout_rate"],
            ),
            input_shape=(1, hyperparams["vocab_size"]),
            # ensure PRNGKey is made available to dropout layers
            apply_rng=["dropout"],
            # indicate mutable state due to BatchNorm layers
            mutable=["batch_stats"],
            # to ensure proper initialisation of BatchNorm we must
            # initialise with is_training=True
            is_training=True,
        )
    elif nn_framework == "haiku":
        encoder = haiku_module(
            "encoder",
            # use `transform_with_state` for BatchNorm
            hk.transform_with_state(
                HaikuEncoder(
            )
        )
hyperparams["vocab_size"],
hyperparams["num_topics"],
hyperparams["hidden"],
hyperparams["dropout_rate"],
),
)

input_shape=(1, hyperparams["vocab_size"],
apply_rng=True,
# to ensure proper initialisation of BatchNorm we must
# initialise with is_training=True
is_training=True,
)
else:
    raise ValueError(f"Invalid choice {nn_framework} for argument nn_framework")

with numpyro.plate(
    "documents", docs.shape[0], subsample_size=hyperparams["batch_size"]
):
    batch_docs = numpyro.subsample(docs, event_dim=1)

if nn_framework == "flax":
    concentration = encoder(
        batch_docs, is_training, rngs={"dropout": numpyro.prng_key()}
    )
elif nn_framework == "haiku":
    concentration = encoder(numpyro.prng_key(), batch_docs, is_training)

numpyro.sample("theta", dist.Dirichlet(concentration))


def load_data():
    news = fetch_20newsgroups(subset="all")
    vectorizer = CountVectorizer(max_df=0.5, min_df=20, stop_words="english")
    docs = jnp.array(vectorizer.fit_transform(news["data"]).toarray())
    vocab = pd.DataFrame(columns=["word", "index"])
    vocab["word"] = vectorizer.get_feature_names_out()
    vocab["index"] = vocab.index

    return docs, vocab

def run_inference(docs, args):
    rng_key = random.PRNGKey(0)
    docs = device_put(docs)

    hyperparams = dict(
        vocab_size=docs.shape[1],
        num_topics=ars.num_topics,
        hidden=ars.hidden,
        dropout_rate=ars.dropout_rate,
        batch_size=ars.batch_size,
    )
optimizer = numpyro.optim.Adam(args.learning_rate)
svi = SVI(model, guide, optimizer, loss=TraceMeanField_ELBO())

return svi.run(
    rng_key,
    args.num_steps,
    docs,
    hyperparams,
    is_training=True,
    progress_bar=not args.disable_progbar,
    nn_framework=args.nn_framework,
)

def plot_word_cloud(b, ax, vocab, n):
    indices = jnp.argsort(b)[::-1]
top20 = indices[:20]
df = pd.DataFrame(top20, columns=["index"])
words = pd.merge(df, vocab[["index", "word"]], how="left", on="index")[["word"]].values.tolist()
sizes = b[top20].tolist()
freqs = {words[i]: sizes[i] for i in range(len(words))}
wc = WordCloud(background_color="white", width=800, height=500)
wcc = wc.generate_from_frequencies(freqs)
ax.set_title(f"Topic {n + 1}")
ax.imshow(wc, interpolation="bilinear")
ax.axis("off")

def main(args):
    docs, vocab = load_data()
    print(f"Dictionary size: {len(vocab)}")
    print(f"Corpus size: {docs.shape}")

    svi_result = run_inference(docs, args)

    if args.nn_framework == "flax":
        beta = svi_result.params["decoder$params"["Dense_0"]]["kernel"]
    elif args.nn_framework == "haiku":
        beta = svi_result.params["decoder$params"["linear"]]["w"]

    beta = jax.nn.softmax(beta)

    # the number of plots depends on the chosen number of topics.
    # add 2 to num topics to ensure we create a row for any remainder after division
    nrows = (args.num_topics + 2) // 3
    fig, axs = plt.subplots(nrows, 3, figsize=(14, 3 + 3 * nrows))
    axs = axs.flatten()
```python
for n in range(beta.shape[0]):
  plot_word_cloud(beta[n], axs[n], vocab, n)

  # hide any unused axes
for i in range(n, len(axs)):
  axs[i].axis("off")

fig.savefig("wordclouds.png")

if __name__ == "__main__":
  assert numpyro.__version__.startswith("0.15.2")
  parser = argparse.ArgumentParser(
      description="Probabilistic topic modelling with Flax and Haiku"
  )
  parser.add_argument("-n", "--num-steps", nargs="?", default=30_000, type=int)
  parser.add_argument("-t", "--num-topics", nargs="?", default=12, type=int)
  parser.add_argument("--batch-size", narg="?", default=32, type=int)
  parser.add_argument("--learning-rate", narg="?", default=1e-3, type=float)
  parser.add_argument("--hidden", narg="?", default=200, type=int)
  parser.add_argument("--dropout-rate", narg="?", default=0.2, type=float)
  parser.add_argument(
      "-dp",
      "--disable-progbar",
      action="store_true",
      default=False,
      help="Whether to disable progress bar",
  )
  parser.add_argument(
      "--device", default="cpu", type=str, help="use "cpu", "gpu" or "tpu"."
  )
  parser.add_argument(
      "--nn-framework",
      nargs="?",
      default="flax",
      help="The framework to use for constructing encoder / decoder. Options are "flax" or "haiku".
  ),
  
  args = parser.parse_args()

  numpyro.set_platform(args.device)
  main(args)
```
Author: Madhav Kanda

Occasionally, the Hamiltonian Monte Carlo (HMC) sampler encounters challenges in effectively sampling from the posterior distribution. One illustrative case is Neal’s funnel. In these situations, the conventional centered parameterization may prove inadequate, leading us to employ non-centered parameterization. However, there are instances where even non-centered parameterization may not suffice, necessitating the utilization of Variationally Inferred Parameterization to attain the desired centeredness within the range of 0 to 1.

The purpose of this tutorial is to implement Variationally Inferred Parameterization based on Automatic Reparameterization of Probabilistic Programs using LocScaleReparam in Numpyro.

```
[ ]: %pip -qq install numpyro
%pip -qq install ucimlrepo

[ ]: import arviz as az
import numpy as np
from ucimlrepo import fetch_ucirepo
import jax
import jax.numpy as jnp
import numpyro
distributions as dist
from numpyro.infer import MCMC, NUTS, SVI, Trace_ELBO
from numpyro.infer.autoguide import AutoDiagonalNormal
from numpyro.infer.reparam import LocScaleReparam
rng_key = jax.random.PRNGKey(0)
```

### 14.1 1. Dataset

We will be using the German Credit Dataset for this illustration. The dataset consists of 1000 entries with 20 categorical symbolic attributes prepared by Prof. Hofmann. In this dataset, each entry represents a person who takes a credit by a bank. Each person is classified as good or bad credit risks according to the set of attributes.

```
[ ]: def load_german_credit():
    statlog_german_credit_data = fetch_ucirepo(id=144)
    X = statlog_german_credit_data.data.features
    y = statlog_german_credit_data.data.targets
    return X, y
```
```python
X, y = load_german_credit()
```

<table>
<thead>
<tr>
<th>Attribute1</th>
<th>Attribute2</th>
<th>Attribute3</th>
<th>Attribute4</th>
<th>Attribute5</th>
<th>Attribute6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A11</td>
<td>6</td>
<td>A34</td>
<td>A43</td>
<td>1169</td>
</tr>
<tr>
<td>1</td>
<td>A12</td>
<td>48</td>
<td>A32</td>
<td>A43</td>
<td>5951</td>
</tr>
<tr>
<td>2</td>
<td>A14</td>
<td>12</td>
<td>A34</td>
<td>A46</td>
<td>2096</td>
</tr>
<tr>
<td>3</td>
<td>A11</td>
<td>42</td>
<td>A32</td>
<td>A42</td>
<td>7882</td>
</tr>
<tr>
<td>4</td>
<td>A11</td>
<td>24</td>
<td>A33</td>
<td>A40</td>
<td>4870</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>995</td>
<td>A14</td>
<td>12</td>
<td>A32</td>
<td>A42</td>
<td>1736</td>
</tr>
<tr>
<td>996</td>
<td>A11</td>
<td>30</td>
<td>A32</td>
<td>A41</td>
<td>3857</td>
</tr>
<tr>
<td>997</td>
<td>A14</td>
<td>12</td>
<td>A32</td>
<td>A43</td>
<td>804</td>
</tr>
<tr>
<td>998</td>
<td>A11</td>
<td>45</td>
<td>A32</td>
<td>A43</td>
<td>1845</td>
</tr>
<tr>
<td>999</td>
<td>A12</td>
<td>45</td>
<td>A34</td>
<td>A41</td>
<td>4576</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute7</th>
<th>Attribute8</th>
<th>Attribute9</th>
<th>Attribute10</th>
<th>Attribute11</th>
<th>Attribute12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A75</td>
<td>4</td>
<td>A93</td>
<td>A101</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>A73</td>
<td>2</td>
<td>A92</td>
<td>A101</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>A74</td>
<td>2</td>
<td>A93</td>
<td>A101</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>A74</td>
<td>3</td>
<td>A93</td>
<td>A101</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>995</td>
<td>A74</td>
<td>3</td>
<td>A92</td>
<td>A101</td>
<td>4</td>
</tr>
<tr>
<td>996</td>
<td>A73</td>
<td>4</td>
<td>A91</td>
<td>A101</td>
<td>4</td>
</tr>
<tr>
<td>997</td>
<td>A75</td>
<td>4</td>
<td>A93</td>
<td>A101</td>
<td>4</td>
</tr>
<tr>
<td>998</td>
<td>A73</td>
<td>4</td>
<td>A93</td>
<td>A101</td>
<td>4</td>
</tr>
<tr>
<td>999</td>
<td>A71</td>
<td>3</td>
<td>A93</td>
<td>A101</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute13</th>
<th>Attribute14</th>
<th>Attribute15</th>
<th>Attribute16</th>
<th>Attribute17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>67</td>
<td>A143</td>
<td>A152</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>A143</td>
<td>A152</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>49</td>
<td>A143</td>
<td>A152</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td>A143</td>
<td>A153</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>53</td>
<td>A143</td>
<td>A153</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>995</td>
<td>31</td>
<td>A143</td>
<td>A152</td>
<td>1</td>
</tr>
<tr>
<td>996</td>
<td>40</td>
<td>A143</td>
<td>A152</td>
<td>1</td>
</tr>
<tr>
<td>997</td>
<td>38</td>
<td>A143</td>
<td>A152</td>
<td>1</td>
</tr>
<tr>
<td>998</td>
<td>23</td>
<td>A143</td>
<td>A153</td>
<td>1</td>
</tr>
<tr>
<td>999</td>
<td>27</td>
<td>A143</td>
<td>A152</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute18</th>
<th>Attribute19</th>
<th>Attribute20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>A192</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A191</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A191</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>A191</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>A191</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>995</td>
<td>1</td>
<td>A191</td>
</tr>
<tr>
<td>996</td>
<td>1</td>
<td>A192</td>
</tr>
<tr>
<td>997</td>
<td>1</td>
<td>A191</td>
</tr>
<tr>
<td>998</td>
<td>1</td>
<td>A192</td>
</tr>
</tbody>
</table>

(continues on next page)
Here, X depicts 20 attributes and the values corresponding to these attributes for each person represented in the data entry and y is the output variable corresponding to these attributes

```python
def data_transform(X, y):
    def categorical_to_int(x):
        d = {u: i for i, u in enumerate(np.unique(x))}
        return np.array([d[i] for i in x])

categoricals = []
umericals = []
umericals.append(np.ones([len(y)]))
for column in X:
    column = X[column]
    if column.dtype == "O":
        categoricals.append(categorical_to_int(column))
    else:
        numericals.append((column - column.mean()) / column.std())
numericals = np.array(numericals).T
status = np.array(y == 1, dtype=np.int32)
status = np.squeeze(status)
return jnp.array(numericals), jnp.array(categoricals), jnp.array(status)
```

Data transformation for feeding it into the Numpyr model

```python
numericals, categoricals, status = data_transform(X, y)
```

```python
x_numeric = numericals.astype(jnp.float32)
x_categorical = [jnp.eye(c.max() + 1)[c] for c in categoricals]
all_x = jnp.concatenate([x_numeric] + x_categorical, axis=1)
um_features = all_x.shape[1]
y = status[jnp.newaxis, Ellipsis]
```

### 14.2. 2. Model

We will be using a logistic regression model with hierarchical prior on coefficient scales

\[
\begin{align*}
\log \tau_0 & \sim \mathcal{N}(0, 10) \\
\log \tau_i & \sim \mathcal{N}(\log \tau_0, 1) \\
\beta_i & \sim \mathcal{N}(0, \tau_i) \\
y & \sim \text{Bernoulli}\left(\sigma\left(\beta X^T\right)\right)
\end{align*}
\] (14.1)  (14.2)

```python
def german_credit():
    log_tau_zero = numpyro.sample("log_tau_zero", dist.Normal(0, 10))
    log_tau_i = numpyro.sample("log_tau_i", dist.Normal(log_tau_zero, jnp.ones(num_features)))
```

(continues on next page)
beta = numpyro.sample(
    'beta', dist.Normal(jnp.zeros(num_features), jnp.exp(log_tau_i))
)

numpyro.sample(
    'obs',
    dist.Bernoulli(logits=jnp.einsum("nd,md->mn", all_x, beta[jnp.newaxis, :]),
    obs=y,
)

nuts_kernel = NUTS(german_credit)
mcmc = MCMC(nuts_kernel, num_warmup=1000, num_samples=1000)
mcmc.run(rng_key, extra_fields=("num_steps",))

mcmc.print_summary()

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta[0]</td>
<td>0.13</td>
<td>0.38</td>
<td>0.05</td>
<td>-0.36</td>
<td>0.74</td>
<td>284.06</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[1]</td>
<td>-0.34</td>
<td>0.12</td>
<td>-0.34</td>
<td>-0.52</td>
<td>-0.15</td>
<td>621.55</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[2]</td>
<td>-0.27</td>
<td>0.13</td>
<td>-0.27</td>
<td>-0.45</td>
<td>-0.03</td>
<td>542.13</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[3]</td>
<td>-0.30</td>
<td>0.10</td>
<td>-0.30</td>
<td>-0.44</td>
<td>-0.11</td>
<td>566.55</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[4]</td>
<td>-0.00</td>
<td>0.07</td>
<td>-0.00</td>
<td>-0.12</td>
<td>0.11</td>
<td>782.35</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[5]</td>
<td>0.12</td>
<td>0.09</td>
<td>0.11</td>
<td>-0.02</td>
<td>0.27</td>
<td>728.28</td>
<td>1.01</td>
</tr>
<tr>
<td>beta[6]</td>
<td>-0.08</td>
<td>0.08</td>
<td>-0.07</td>
<td>-0.22</td>
<td>0.05</td>
<td>822.89</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[7]</td>
<td>-0.05</td>
<td>0.07</td>
<td>-0.04</td>
<td>-0.19</td>
<td>0.05</td>
<td>752.66</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[8]</td>
<td>-0.42</td>
<td>0.32</td>
<td>-0.39</td>
<td>-0.87</td>
<td>0.05</td>
<td>198.00</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[9]</td>
<td>-0.07</td>
<td>0.26</td>
<td>-0.02</td>
<td>-0.50</td>
<td>0.31</td>
<td>220.27</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[10]</td>
<td>0.26</td>
<td>0.31</td>
<td>0.18</td>
<td>-0.15</td>
<td>0.78</td>
<td>404.97</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[11]</td>
<td>1.23</td>
<td>0.34</td>
<td>1.25</td>
<td>0.68</td>
<td>1.79</td>
<td>227.34</td>
<td>1.01</td>
</tr>
<tr>
<td>beta[12]</td>
<td>-0.26</td>
<td>0.34</td>
<td>-0.17</td>
<td>-0.81</td>
<td>0.22</td>
<td>349.10</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[13]</td>
<td>-0.30</td>
<td>0.34</td>
<td>-0.21</td>
<td>-0.86</td>
<td>0.13</td>
<td>387.72</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[14]</td>
<td>0.07</td>
<td>0.20</td>
<td>0.04</td>
<td>-0.26</td>
<td>0.38</td>
<td>240.45</td>
<td>1.03</td>
</tr>
<tr>
<td>beta[15]</td>
<td>0.10</td>
<td>0.22</td>
<td>0.05</td>
<td>-0.18</td>
<td>0.50</td>
<td>287.41</td>
<td>1.02</td>
</tr>
<tr>
<td>beta[16]</td>
<td>0.76</td>
<td>0.30</td>
<td>0.76</td>
<td>0.22</td>
<td>1.24</td>
<td>364.73</td>
<td>1.03</td>
</tr>
<tr>
<td>beta[17]</td>
<td>-0.53</td>
<td>0.28</td>
<td>-0.55</td>
<td>-0.94</td>
<td>-0.05</td>
<td>269.95</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[18]</td>
<td>0.70</td>
<td>0.42</td>
<td>0.70</td>
<td>-0.02</td>
<td>1.29</td>
<td>367.28</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[19]</td>
<td>0.17</td>
<td>0.40</td>
<td>0.06</td>
<td>-0.43</td>
<td>0.77</td>
<td>333.54</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[20]</td>
<td>0.03</td>
<td>0.19</td>
<td>0.01</td>
<td>-0.23</td>
<td>0.39</td>
<td>381.57</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[21]</td>
<td>0.18</td>
<td>0.22</td>
<td>0.13</td>
<td>-0.14</td>
<td>0.53</td>
<td>335.48</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[22]</td>
<td>-0.05</td>
<td>0.32</td>
<td>-0.01</td>
<td>-0.56</td>
<td>0.46</td>
<td>439.54</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[23]</td>
<td>-0.10</td>
<td>0.30</td>
<td>-0.04</td>
<td>-0.63</td>
<td>0.30</td>
<td>508.20</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[24]</td>
<td>-0.34</td>
<td>0.36</td>
<td>-0.25</td>
<td>-0.94</td>
<td>0.12</td>
<td>283.15</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[25]</td>
<td>0.14</td>
<td>0.40</td>
<td>0.04</td>
<td>-0.46</td>
<td>0.71</td>
<td>433.69</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[26]</td>
<td>-0.01</td>
<td>0.19</td>
<td>-0.00</td>
<td>-0.34</td>
<td>0.28</td>
<td>438.64</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[27]</td>
<td>-0.36</td>
<td>0.27</td>
<td>-0.33</td>
<td>-0.78</td>
<td>0.04</td>
<td>377.33</td>
<td>1.01</td>
</tr>
<tr>
<td>beta[28]</td>
<td>-0.07</td>
<td>0.22</td>
<td>-0.03</td>
<td>-0.43</td>
<td>0.26</td>
<td>493.09</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[29]</td>
<td>0.01</td>
<td>0.22</td>
<td>0.00</td>
<td>-0.32</td>
<td>0.34</td>
<td>448.21</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[30]</td>
<td>0.35</td>
<td>0.43</td>
<td>0.22</td>
<td>-0.18</td>
<td>1.08</td>
<td>314.69</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[31]</td>
<td>0.41</td>
<td>0.33</td>
<td>0.40</td>
<td>-0.10</td>
<td>0.90</td>
<td>402.62</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[32]</td>
<td>-0.03</td>
<td>0.21</td>
<td>-0.01</td>
<td>-0.39</td>
<td>0.30</td>
<td>525.23</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[33]</td>
<td>-0.12</td>
<td>0.18</td>
<td>-0.09</td>
<td>-0.41</td>
<td>0.16</td>
<td>334.94</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[34]</td>
<td>-0.02</td>
<td>0.16</td>
<td>-0.01</td>
<td>-0.39</td>
<td>0.26</td>
<td>318.25</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[35]</td>
<td>0.42</td>
<td>0.27</td>
<td>0.42</td>
<td>-0.04</td>
<td>0.81</td>
<td>455.99</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[36]</td>
<td>-0.07</td>
<td>0.20</td>
<td>-0.04</td>
<td>-0.39</td>
<td>0.24</td>
<td>410.71</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[37]</td>
<td>0.56</td>
<td>0.46</td>
<td>0.49</td>
<td>-0.11</td>
<td>1.28</td>
<td>570.74</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[38]</td>
<td>-0.09</td>
<td>0.16</td>
<td>0.15</td>
<td>-0.14</td>
<td>0.50</td>
<td>379.76</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[39]</td>
<td>-0.00</td>
<td>0.16</td>
<td>0.00</td>
<td>-0.25</td>
<td>0.26</td>
<td>352.19</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[40]</td>
<td>0.19</td>
<td>0.24</td>
<td>0.24</td>
<td>-0.32</td>
<td>0.41</td>
<td>522.43</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[41]</td>
<td>-0.02</td>
<td>0.14</td>
<td>0.01</td>
<td>-0.22</td>
<td>0.23</td>
<td>562.00</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[42]</td>
<td>0.10</td>
<td>0.13</td>
<td>0.10</td>
<td>-0.24</td>
<td>0.21</td>
<td>638.20</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[43]</td>
<td>-0.07</td>
<td>0.23</td>
<td>-0.02</td>
<td>-0.45</td>
<td>0.30</td>
<td>437.47</td>
<td>1.02</td>
</tr>
<tr>
<td>beta[44]</td>
<td>0.38</td>
<td>0.28</td>
<td>0.37</td>
<td>-0.03</td>
<td>0.82</td>
<td>266.19</td>
<td>1.04</td>
</tr>
<tr>
<td>beta[45]</td>
<td>-0.14</td>
<td>0.22</td>
<td>-0.09</td>
<td>-0.52</td>
<td>0.16</td>
<td>406.31</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[46]</td>
<td>0.19</td>
<td>0.23</td>
<td>0.14</td>
<td>-0.14</td>
<td>0.55</td>
<td>338.97</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[47]</td>
<td>-0.04</td>
<td>0.22</td>
<td>0.02</td>
<td>-0.23</td>
<td>0.43</td>
<td>438.03</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[48]</td>
<td>0.05</td>
<td>0.24</td>
<td>0.02</td>
<td>-0.32</td>
<td>0.41</td>
<td>522.43</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[49]</td>
<td>-0.00</td>
<td>0.14</td>
<td>0.01</td>
<td>-0.22</td>
<td>0.23</td>
<td>562.00</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[50]</td>
<td>0.19</td>
<td>0.24</td>
<td>0.24</td>
<td>-0.32</td>
<td>0.41</td>
<td>522.43</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[51]</td>
<td>-0.02</td>
<td>0.14</td>
<td>0.01</td>
<td>-0.22</td>
<td>0.23</td>
<td>562.00</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[52]</td>
<td>0.10</td>
<td>0.13</td>
<td>0.10</td>
<td>-0.24</td>
<td>0.21</td>
<td>638.20</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[53]</td>
<td>-0.07</td>
<td>0.18</td>
<td>-0.04</td>
<td>-0.34</td>
<td>0.23</td>
<td>481.37</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[54]</td>
<td>0.13</td>
<td>0.19</td>
<td>0.09</td>
<td>-0.12</td>
<td>0.47</td>
<td>507.56</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[55]</td>
<td>-0.14</td>
<td>0.33</td>
<td>-0.06</td>
<td>-0.64</td>
<td>0.37</td>
<td>303.00</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[56]</td>
<td>-0.00</td>
<td>0.56</td>
<td>0.32</td>
<td>-0.18</td>
<td>1.41</td>
<td>438.86</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[57]</td>
<td>-1.51</td>
<td>0.95</td>
<td>-1.52</td>
<td>-3.03</td>
<td>0.11</td>
<td>290.78</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[58]</td>
<td>-1.07</td>
<td>0.67</td>
<td>-1.11</td>
<td>-2.12</td>
<td>0.03</td>
<td>641.04</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[59]</td>
<td>-1.24</td>
<td>0.76</td>
<td>-1.26</td>
<td>-2.47</td>
<td>0.03</td>
<td>666.31</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[60]</td>
<td>-1.16</td>
<td>0.65</td>
<td>-1.19</td>
<td>-2.20</td>
<td>-0.10</td>
<td>821.60</td>
<td>1.00</td>
</tr>
<tr>
<td>beta[61]</td>
<td>-2.11</td>
<td>0.88</td>
<td>-2.13</td>
<td>-3.50</td>
<td>-0.61</td>
<td>806.15</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[0]</td>
<td>-1.71</td>
<td>0.86</td>
<td>-1.68</td>
<td>-3.28</td>
<td>-0.44</td>
<td>697.00</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[1]</td>
<td>-1.88</td>
<td>0.84</td>
<td>-1.91</td>
<td>-3.30</td>
<td>-0.58</td>
<td>623.56</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[2]</td>
<td>-1.99</td>
<td>0.90</td>
<td>-1.98</td>
<td>-3.51</td>
<td>-0.65</td>
<td>710.21</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[3]</td>
<td>-2.00</td>
<td>0.86</td>
<td>-0.96</td>
<td>-2.23</td>
<td>0.52</td>
<td>445.30</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[4]</td>
<td>-1.69</td>
<td>0.93</td>
<td>-1.63</td>
<td>-3.17</td>
<td>-0.14</td>
<td>326.33</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[5]</td>
<td>-1.41</td>
<td>0.95</td>
<td>-1.35</td>
<td>-2.93</td>
<td>0.19</td>
<td>441.60</td>
<td>1.01</td>
</tr>
<tr>
<td>log_tau_i[6]</td>
<td>-0.11</td>
<td>0.57</td>
<td>-0.12</td>
<td>-0.97</td>
<td>0.80</td>
<td>539.60</td>
<td>1.00</td>
</tr>
<tr>
<td>logTau_i[7]</td>
<td>-1.36</td>
<td>0.96</td>
<td>-1.31</td>
<td>-3.16</td>
<td>0.01</td>
<td>336.11</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[8]</td>
<td>-1.30</td>
<td>0.95</td>
<td>-1.26</td>
<td>-2.85</td>
<td>0.28</td>
<td>335.94</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[9]</td>
<td>-1.72</td>
<td>0.89</td>
<td>-1.70</td>
<td>-3.05</td>
<td>-0.25</td>
<td>584.38</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[10]</td>
<td>-1.65</td>
<td>0.92</td>
<td>-1.63</td>
<td>-3.07</td>
<td>-0.10</td>
<td>345.77</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[11]</td>
<td>-0.51</td>
<td>0.65</td>
<td>-0.49</td>
<td>-1.42</td>
<td>0.59</td>
<td>676.64</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[12]</td>
<td>-0.84</td>
<td>0.76</td>
<td>-0.76</td>
<td>-2.09</td>
<td>0.34</td>
<td>303.14</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[13]</td>
<td>-0.69</td>
<td>0.82</td>
<td>-0.59</td>
<td>-2.03</td>
<td>0.61</td>
<td>359.35</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[14]</td>
<td>-1.45</td>
<td>0.99</td>
<td>-1.42</td>
<td>-2.97</td>
<td>0.25</td>
<td>397.18</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[15]</td>
<td>-1.75</td>
<td>0.94</td>
<td>-1.73</td>
<td>-3.39</td>
<td>-0.40</td>
<td>617.54</td>
<td>1.00</td>
</tr>
</tbody>
</table>
From `mcmc.print_summary` it is evident that there are **37 divergences**. Thus, we will use Variationally Inferred Parameterization (VIP) to reduce these divergences.

```python
[ ]: data = az.from_numpyro(mcmc)
az.plot_trace(data, compact=True);
```
14.3. 3. Reparameterization

We introduce a parameterization parameters $\lambda \in [0, 1]$ for any variable $z$, and transform:

$\Rightarrow z \sim N(z | \mu, \sigma)$

$\Rightarrow$ by defining $z \sim N(\lambda \mu, \sigma^\lambda)$

$\Rightarrow z = \mu + \sigma^{1-\lambda}(z - \lambda \mu)$.

Thus, using the above transformation the joint density can be transformed as follows:

$$p(\theta, \hat{\mu}, y) = \mathcal{N}(\theta | 0, 1) \times \mathcal{N}(\mu | \theta, \sigma_\mu) \times \mathcal{N}(y | \mu, \sigma)$$ (14.3)

$$p(\theta, \hat{\mu}, y) = \mathcal{N}(\theta | 0, 1) \times \mathcal{N}(\hat{\mu} | \lambda \theta, \sigma_\mu^\lambda) \times \mathcal{N}(y | \theta + \sigma_\mu^{1-\lambda}(\hat{\mu} - \lambda \theta), \sigma)$$ (14.4)

```python
[ ]: def german_credit_reparam(beta_centeredness=None):
    def model():
        log_tau_zero = numpyro.sample("log_tau_zero", dist.Normal(0, 10))
        log_tau_i = numpyro.sample("log_tau_i", dist.Normal(log_tau_zero, jnp.ones(num_features))
        with numpyro.handlers.reparam(config={"beta": LocScaleReparam(beta_centeredness)}):
            beta = numpyro.sample("beta", dist.Normal(jnp.zeros(num_features), jnp.exp(log_tau_i)))
```

(continues on next page)
NumPyro Documentation

(continued from previous page)

cumpyro.sample("obs",
    dist.Bernoulli(logits=jnp.einsum("nd,md->mn", all_x, beta[jnp.newaxis, :]),
    obs=y),
)

return model

Now, using SVI we optimize $\lambda$.

[ ]: model = german_credit_reparam()
guide = AutoDiagonalNormal(model)
svi = SVI(model, guide, numpyro.optim.Adam(3e-4), Trace_ELBO(10))
svi_results = svi.run(rng_key, 10000)

100%|████████████████████| 10000/10000 [00:16<00:00, 588.87it/s, init loss: 2165.2424, → avg. loss [9501-10000]: 576.7846]

[ ]: reparam_model = german_credit_reparam(
    beta_centeredness=svi_results.params["beta_centered"]
)

[ ]: nuts_kernel = NUTS(reparam_model)
mcmc_reparam = MCMC(nuts_kernel, num_warmup=1000, num_samples=1000)
mcmc_reparam.run(rng_key, extra_fields=("num_steps",))

sample: 100%|████████████████████| 2000/2000 [00:07<00:00, 285.41it/s, 31 steps of␣ → size 1.28e-01. acc. prob=0.89]

[ ]: mcmc_reparam.print_summary()

<table>
<thead>
<tr>
<th>beta_decentered[0]</th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12</td>
<td>0.40</td>
<td>0.06</td>
<td>-0.48</td>
<td>0.80</td>
<td>338.70</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.45</td>
<td>0.15</td>
<td>-0.45</td>
<td>0.70</td>
<td>-0.21</td>
<td>791.23</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.38</td>
<td>0.17</td>
<td>-0.38</td>
<td>-0.65</td>
<td>-0.09</td>
<td>691.79</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.41</td>
<td>0.13</td>
<td>-0.41</td>
<td>-0.61</td>
<td>-0.19</td>
<td>1022.79</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.01</td>
<td>0.11</td>
<td>-0.01</td>
<td>-0.18</td>
<td>0.20</td>
<td>1176.84</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.19</td>
<td>0.14</td>
<td>0.19</td>
<td>-0.04</td>
<td>0.41</td>
<td>1194.41</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.13</td>
<td>0.14</td>
<td>-0.13</td>
<td>-0.36</td>
<td>0.09</td>
<td>1227.24</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.07</td>
<td>0.12</td>
<td>-0.06</td>
<td>-0.24</td>
<td>0.14</td>
<td>1096.31</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.46</td>
<td>0.34</td>
<td>-0.46</td>
<td>-0.99</td>
<td>0.08</td>
<td>330.30</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.03</td>
<td>0.32</td>
<td>-0.02</td>
<td>-0.57</td>
<td>0.49</td>
<td>310.35</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.35</td>
<td>0.39</td>
<td>0.30</td>
<td>-0.26</td>
<td>1.00</td>
<td>426.11</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>1.29</td>
<td>0.31</td>
<td>1.30</td>
<td>0.81</td>
<td>1.82</td>
<td>433.16</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.32</td>
<td>0.39</td>
<td>-0.25</td>
<td>-0.96</td>
<td>0.24</td>
<td>521.05</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.38</td>
<td>0.40</td>
<td>-0.32</td>
<td>-1.00</td>
<td>0.24</td>
<td>410.05</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.08</td>
<td>0.28</td>
<td>0.06</td>
<td>-0.37</td>
<td>0.57</td>
<td>457.72</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.14</td>
<td>0.30</td>
<td>0.10</td>
<td>-0.28</td>
<td>0.66</td>
<td>612.31</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.85</td>
<td>0.31</td>
<td>0.86</td>
<td>0.41</td>
<td>1.45</td>
<td>432.14</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>-0.64</td>
<td>0.28</td>
<td>-0.65</td>
<td>-1.05</td>
<td>-0.14</td>
<td>523.15</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>0.78</td>
<td>0.42</td>
<td>0.78</td>
<td>0.07</td>
<td>1.46</td>
<td>545.52</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
<table>
<thead>
<tr>
<th>beta_decentered[19]</th>
<th>0.15</th>
<th>0.39</th>
<th>0.08</th>
<th>-0.50</th>
<th>0.80</th>
<th>662.60</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta_decentered[20]</td>
<td>0.04</td>
<td>0.25</td>
<td>0.03</td>
<td>-0.39</td>
<td>0.40</td>
<td>445.85</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[21]</td>
<td>0.24</td>
<td>0.27</td>
<td>0.21</td>
<td>-0.20</td>
<td>0.65</td>
<td>477.68</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[22]</td>
<td>-0.03</td>
<td>0.38</td>
<td>-0.01</td>
<td>-0.64</td>
<td>0.60</td>
<td>984.59</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[23]</td>
<td>-0.13</td>
<td>0.34</td>
<td>-0.08</td>
<td>-0.72</td>
<td>0.35</td>
<td>702.87</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[24]</td>
<td>-0.41</td>
<td>0.39</td>
<td>-0.37</td>
<td>-1.08</td>
<td>0.13</td>
<td>663.13</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[25]</td>
<td>0.19</td>
<td>0.47</td>
<td>0.09</td>
<td>-0.48</td>
<td>0.92</td>
<td>529.68</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[26]</td>
<td>0.00</td>
<td>0.25</td>
<td>0.01</td>
<td>-0.47</td>
<td>0.35</td>
<td>690.54</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[27]</td>
<td>-0.46</td>
<td>0.31</td>
<td>-0.46</td>
<td>-0.95</td>
<td>0.04</td>
<td>464.44</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[28]</td>
<td>-0.09</td>
<td>0.30</td>
<td>-0.06</td>
<td>-0.56</td>
<td>0.41</td>
<td>464.65</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[29]</td>
<td>0.02</td>
<td>0.30</td>
<td>0.01</td>
<td>-0.47</td>
<td>0.52</td>
<td>747.44</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[30]</td>
<td>0.38</td>
<td>0.44</td>
<td>0.31</td>
<td>-0.30</td>
<td>1.05</td>
<td>717.12</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[31]</td>
<td>0.47</td>
<td>0.36</td>
<td>0.47</td>
<td>-0.09</td>
<td>1.03</td>
<td>564.18</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[32]</td>
<td>-0.03</td>
<td>0.26</td>
<td>-0.02</td>
<td>-0.44</td>
<td>0.44</td>
<td>572.03</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[33]</td>
<td>-0.17</td>
<td>0.25</td>
<td>-0.15</td>
<td>-0.63</td>
<td>0.19</td>
<td>713.40</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[34]</td>
<td>-0.02</td>
<td>0.21</td>
<td>-0.01</td>
<td>-0.39</td>
<td>0.32</td>
<td>620.45</td>
<td>1.01</td>
</tr>
<tr>
<td>beta_decentered[35]</td>
<td>0.53</td>
<td>0.31</td>
<td>0.55</td>
<td>-0.03</td>
<td>1.00</td>
<td>681.60</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[36]</td>
<td>0.09</td>
<td>0.24</td>
<td>0.06</td>
<td>-0.27</td>
<td>0.49</td>
<td>610.06</td>
<td>1.01</td>
</tr>
<tr>
<td>beta_decentered[37]</td>
<td>-0.14</td>
<td>0.31</td>
<td>-0.10</td>
<td>-0.74</td>
<td>0.28</td>
<td>826.87</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[38]</td>
<td>-0.12</td>
<td>0.25</td>
<td>-0.11</td>
<td>-0.53</td>
<td>0.30</td>
<td>493.49</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[39]</td>
<td>0.44</td>
<td>0.28</td>
<td>0.44</td>
<td>-0.01</td>
<td>0.89</td>
<td>542.71</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[40]</td>
<td>0.05</td>
<td>0.26</td>
<td>0.03</td>
<td>-0.39</td>
<td>0.45</td>
<td>709.78</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[41]</td>
<td>0.02</td>
<td>0.30</td>
<td>0.01</td>
<td>-0.52</td>
<td>0.46</td>
<td>389.41</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[42]</td>
<td>-0.15</td>
<td>0.33</td>
<td>-0.10</td>
<td>-0.74</td>
<td>0.32</td>
<td>607.05</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[43]</td>
<td>0.66</td>
<td>0.47</td>
<td>0.65</td>
<td>-0.11</td>
<td>1.38</td>
<td>539.31</td>
<td>1.01</td>
</tr>
<tr>
<td>beta_decentered[44]</td>
<td>0.25</td>
<td>0.27</td>
<td>0.23</td>
<td>-0.19</td>
<td>0.66</td>
<td>686.63</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[45]</td>
<td>-0.01</td>
<td>0.22</td>
<td>-0.00</td>
<td>-0.36</td>
<td>0.35</td>
<td>909.70</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[46]</td>
<td>0.02</td>
<td>0.22</td>
<td>0.01</td>
<td>-0.33</td>
<td>0.39</td>
<td>741.33</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[47]</td>
<td>-0.25</td>
<td>0.31</td>
<td>-0.21</td>
<td>-0.72</td>
<td>0.25</td>
<td>487.26</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[48]</td>
<td>-0.18</td>
<td>0.30</td>
<td>-0.15</td>
<td>-0.67</td>
<td>0.30</td>
<td>400.57</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[49]</td>
<td>-0.07</td>
<td>0.30</td>
<td>-0.05</td>
<td>-0.56</td>
<td>0.41</td>
<td>594.38</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[50]</td>
<td>0.46</td>
<td>0.31</td>
<td>0.46</td>
<td>-0.09</td>
<td>0.88</td>
<td>384.33</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[51]</td>
<td>-0.23</td>
<td>0.27</td>
<td>-0.19</td>
<td>-0.69</td>
<td>0.16</td>
<td>561.35</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[52]</td>
<td>0.22</td>
<td>0.25</td>
<td>0.21</td>
<td>0.19</td>
<td>0.60</td>
<td>456.10</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[53]</td>
<td>0.07</td>
<td>0.29</td>
<td>0.04</td>
<td>-0.41</td>
<td>0.53</td>
<td>500.99</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[54]</td>
<td>0.05</td>
<td>0.30</td>
<td>0.02</td>
<td>-0.49</td>
<td>0.51</td>
<td>896.08</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[55]</td>
<td>0.01</td>
<td>0.23</td>
<td>0.01</td>
<td>-0.34</td>
<td>0.43</td>
<td>1033.13</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[56]</td>
<td>-0.02</td>
<td>0.19</td>
<td>-0.01</td>
<td>-0.33</td>
<td>0.28</td>
<td>717.87</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[57]</td>
<td>0.01</td>
<td>0.23</td>
<td>0.01</td>
<td>-0.33</td>
<td>0.41</td>
<td>684.61</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[58]</td>
<td>-0.09</td>
<td>0.26</td>
<td>-0.08</td>
<td>-0.55</td>
<td>0.30</td>
<td>455.66</td>
<td>1.01</td>
</tr>
<tr>
<td>beta_decentered[59]</td>
<td>0.20</td>
<td>0.27</td>
<td>0.18</td>
<td>-0.20</td>
<td>0.63</td>
<td>413.57</td>
<td>1.01</td>
</tr>
<tr>
<td>beta_decentered[60]</td>
<td>-0.14</td>
<td>0.37</td>
<td>-0.09</td>
<td>-0.76</td>
<td>0.46</td>
<td>411.83</td>
<td>1.00</td>
</tr>
<tr>
<td>beta_decentered[61]</td>
<td>0.58</td>
<td>0.57</td>
<td>0.50</td>
<td>-0.23</td>
<td>1.50</td>
<td>495.86</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[0]</td>
<td>-1.56</td>
<td>0.91</td>
<td>-1.53</td>
<td>-2.94</td>
<td>-0.01</td>
<td>480.69</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[1]</td>
<td>-1.07</td>
<td>0.64</td>
<td>-1.08</td>
<td>-1.99</td>
<td>0.06</td>
<td>950.02</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[2]</td>
<td>-1.25</td>
<td>0.74</td>
<td>-1.23</td>
<td>-2.36</td>
<td>-0.06</td>
<td>796.37</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[3]</td>
<td>-1.15</td>
<td>0.65</td>
<td>-1.18</td>
<td>-2.25</td>
<td>-0.14</td>
<td>838.40</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[4]</td>
<td>-2.09</td>
<td>0.90</td>
<td>-2.12</td>
<td>-3.51</td>
<td>-0.49</td>
<td>1120.31</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[5]</td>
<td>-1.70</td>
<td>0.84</td>
<td>-1.69</td>
<td>-3.10</td>
<td>-0.35</td>
<td>1291.74</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[6]</td>
<td>-1.87</td>
<td>0.92</td>
<td>-1.84</td>
<td>-3.67</td>
<td>-0.58</td>
<td>1066.38</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[7]</td>
<td>-2.06</td>
<td>0.89</td>
<td>-2.07</td>
<td>-3.42</td>
<td>-0.42</td>
<td>641.48</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[8]</td>
<td>-1.11</td>
<td>0.91</td>
<td>-1.03</td>
<td>-2.79</td>
<td>0.19</td>
<td>482.69</td>
<td>1.00</td>
</tr>
</tbody>
</table>

(continues on next page)
<table>
<thead>
<tr>
<th>log_tau_i[9]</th>
<th>-1.65</th>
<th>0.91</th>
<th>-1.62</th>
<th>-3.05</th>
<th>-0.04</th>
<th>772.73</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_tau_i[10]</td>
<td>-1.31</td>
<td>0.94</td>
<td>-1.27</td>
<td>-2.77</td>
<td>0.24</td>
<td>578.34</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[11]</td>
<td>-0.04</td>
<td>0.49</td>
<td>-0.08</td>
<td>-0.81</td>
<td>0.77</td>
<td>736.20</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[12]</td>
<td>-1.37</td>
<td>0.98</td>
<td>-1.32</td>
<td>-2.86</td>
<td>0.31</td>
<td>623.14</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[13]</td>
<td>-1.28</td>
<td>0.97</td>
<td>-1.21</td>
<td>-2.78</td>
<td>0.30</td>
<td>653.51</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[14]</td>
<td>-1.70</td>
<td>0.95</td>
<td>-1.71</td>
<td>-3.20</td>
<td>-0.15</td>
<td>831.88</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[15]</td>
<td>-1.67</td>
<td>0.97</td>
<td>-1.65</td>
<td>-3.15</td>
<td>0.04</td>
<td>726.23</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[16]</td>
<td>-0.53</td>
<td>0.65</td>
<td>-0.49</td>
<td>-1.55</td>
<td>0.54</td>
<td>558.56</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[17]</td>
<td>-0.81</td>
<td>0.68</td>
<td>-0.80</td>
<td>-1.91</td>
<td>0.27</td>
<td>655.36</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[18]</td>
<td>-0.71</td>
<td>0.83</td>
<td>-0.60</td>
<td>-1.89</td>
<td>0.75</td>
<td>536.55</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[19]</td>
<td>-1.55</td>
<td>0.98</td>
<td>-1.53</td>
<td>-3.17</td>
<td>-0.02</td>
<td>675.27</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[20]</td>
<td>-1.81</td>
<td>0.99</td>
<td>-1.79</td>
<td>-3.36</td>
<td>-0.41</td>
<td>823.93</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[21]</td>
<td>-1.53</td>
<td>0.93</td>
<td>-1.51</td>
<td>-3.03</td>
<td>-0.04</td>
<td>767.64</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[22]</td>
<td>-1.60</td>
<td>0.99</td>
<td>-1.58</td>
<td>-3.25</td>
<td>-0.01</td>
<td>840.71</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[23]</td>
<td>-1.64</td>
<td>0.97</td>
<td>-1.60</td>
<td>-3.15</td>
<td>0.10</td>
<td>615.62</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[24]</td>
<td>-1.21</td>
<td>0.96</td>
<td>-1.11</td>
<td>-2.61</td>
<td>0.47</td>
<td>674.58</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[25]</td>
<td>-1.54</td>
<td>1.04</td>
<td>-1.49</td>
<td>-3.39</td>
<td>-0.01</td>
<td>599.69</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[26]</td>
<td>-1.77</td>
<td>0.95</td>
<td>-1.74</td>
<td>-3.27</td>
<td>-0.13</td>
<td>915.91</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[27]</td>
<td>-1.14</td>
<td>0.89</td>
<td>-1.07</td>
<td>-2.53</td>
<td>0.36</td>
<td>569.10</td>
<td>1.01</td>
</tr>
<tr>
<td>log_tau_i[28]</td>
<td>-1.72</td>
<td>0.93</td>
<td>-1.66</td>
<td>-3.25</td>
<td>-0.20</td>
<td>911.70</td>
<td>1.01</td>
</tr>
<tr>
<td>log_tau_i[29]</td>
<td>-1.70</td>
<td>0.91</td>
<td>-1.71</td>
<td>-3.16</td>
<td>-0.14</td>
<td>648.29</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[30]</td>
<td>-1.28</td>
<td>1.02</td>
<td>-1.28</td>
<td>-2.99</td>
<td>0.34</td>
<td>575.90</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[31]</td>
<td>-1.10</td>
<td>0.86</td>
<td>-1.04</td>
<td>-2.39</td>
<td>0.45</td>
<td>730.21</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[32]</td>
<td>-1.79</td>
<td>0.95</td>
<td>-1.82</td>
<td>-3.32</td>
<td>-0.19</td>
<td>667.95</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[33]</td>
<td>-1.64</td>
<td>0.86</td>
<td>-1.62</td>
<td>-3.04</td>
<td>-0.32</td>
<td>948.64</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[34]</td>
<td>-1.87</td>
<td>0.92</td>
<td>-1.88</td>
<td>-3.29</td>
<td>-0.25</td>
<td>999.49</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[35]</td>
<td>-1.00</td>
<td>0.85</td>
<td>-0.95</td>
<td>-2.44</td>
<td>0.25</td>
<td>672.42</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[36]</td>
<td>-1.76</td>
<td>0.92</td>
<td>-1.73</td>
<td>-3.19</td>
<td>-0.22</td>
<td>889.56</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[37]</td>
<td>-1.63</td>
<td>1.00</td>
<td>-1.58</td>
<td>-3.11</td>
<td>0.12</td>
<td>973.29</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[38]</td>
<td>-1.72</td>
<td>0.85</td>
<td>-1.70</td>
<td>-3.13</td>
<td>-0.30</td>
<td>837.73</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[39]</td>
<td>-1.15</td>
<td>0.80</td>
<td>-1.13</td>
<td>-2.32</td>
<td>0.18</td>
<td>627.15</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[40]</td>
<td>-1.76</td>
<td>0.93</td>
<td>-1.70</td>
<td>-3.36</td>
<td>-0.34</td>
<td>686.88</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[41]</td>
<td>-1.75</td>
<td>0.96</td>
<td>-1.74</td>
<td>-3.20</td>
<td>-0.09</td>
<td>612.83</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[42]</td>
<td>-1.62</td>
<td>0.91</td>
<td>-1.62</td>
<td>-3.21</td>
<td>-0.25</td>
<td>596.18</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[43]</td>
<td>-0.79</td>
<td>0.93</td>
<td>-0.74</td>
<td>-2.20</td>
<td>0.83</td>
<td>560.52</td>
<td>1.01</td>
</tr>
<tr>
<td>log_tau_i[44]</td>
<td>-1.52</td>
<td>0.94</td>
<td>-1.48</td>
<td>-3.08</td>
<td>0.01</td>
<td>888.27</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[45]</td>
<td>-1.83</td>
<td>0.90</td>
<td>-1.84</td>
<td>-3.27</td>
<td>-0.44</td>
<td>1122.53</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[46]</td>
<td>-1.83</td>
<td>0.89</td>
<td>-1.78</td>
<td>-3.21</td>
<td>-0.31</td>
<td>990.03</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[47]</td>
<td>-1.56</td>
<td>0.93</td>
<td>-1.48</td>
<td>-3.20</td>
<td>-0.21</td>
<td>736.01</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[48]</td>
<td>-1.59</td>
<td>0.94</td>
<td>-1.54</td>
<td>-3.20</td>
<td>-0.01</td>
<td>588.77</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[49]</td>
<td>-1.71</td>
<td>0.93</td>
<td>-1.68</td>
<td>-3.17</td>
<td>-0.17</td>
<td>813.94</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[50]</td>
<td>-1.15</td>
<td>0.83</td>
<td>-1.11</td>
<td>-2.51</td>
<td>0.14</td>
<td>514.68</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[51]</td>
<td>-1.54</td>
<td>0.89</td>
<td>-1.51</td>
<td>-2.97</td>
<td>-0.13</td>
<td>780.67</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[52]</td>
<td>-1.59</td>
<td>0.93</td>
<td>-1.56</td>
<td>-3.21</td>
<td>-0.28</td>
<td>807.47</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[53]</td>
<td>-1.74</td>
<td>0.92</td>
<td>-1.72</td>
<td>-3.10</td>
<td>-0.15</td>
<td>657.89</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[54]</td>
<td>-1.74</td>
<td>0.93</td>
<td>-1.74</td>
<td>-3.09</td>
<td>-0.10</td>
<td>867.29</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[55]</td>
<td>-1.89</td>
<td>0.94</td>
<td>-1.87</td>
<td>-3.34</td>
<td>-0.27</td>
<td>1132.90</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[56]</td>
<td>-1.89</td>
<td>0.92</td>
<td>-1.89</td>
<td>-3.27</td>
<td>-0.27</td>
<td>980.72</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[57]</td>
<td>-1.84</td>
<td>0.91</td>
<td>-1.84</td>
<td>-3.33</td>
<td>-0.38</td>
<td>813.32</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[58]</td>
<td>-1.75</td>
<td>0.93</td>
<td>-1.74</td>
<td>-3.25</td>
<td>-0.34</td>
<td>583.52</td>
<td>1.00</td>
</tr>
<tr>
<td>log_tau_i[59]</td>
<td>-1.59</td>
<td>0.96</td>
<td>-1.53</td>
<td>-3.11</td>
<td>-0.06</td>
<td>754.39</td>
<td>1.01</td>
</tr>
<tr>
<td>log_tau_i[60]</td>
<td>-1.58</td>
<td>0.98</td>
<td>-1.52</td>
<td>-3.23</td>
<td>-0.08</td>
<td>561.05</td>
<td>1.00</td>
</tr>
</tbody>
</table>
```
log_tau_i[61]  -0.98  1.03  -0.85  -2.55  0.79  502.63  1.00
log_tau_zero   -1.49  0.26  -1.49  -1.91  -1.10  220.32  1.00
```

The number of divergences have significantly reduced from 37 to 1.

```
data = az.from_numpyro(mcmc_reparam)
az.plot_trace(data, compact=True, figsize=(15, 25));
```
14.4 4. References:

2. https://github.com/mgorinova/autoreparam/tree/master
In this tutorial we will demonstrate how to create beautiful visualizations of your probabilistic graphical models using `numpyro.render_model`.

```python
[1]: pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
```

```python
import numpy as np
import flax.linen as flax_nn
from jax import nn
import jax.numpy as jnp

import numpyro
from numpyro.contrib.module import flax_module
import numpyro.distributions as dist
import numpyro.distributions.constraints as constraints

assert numpyro.__version__.startswith("0.15.2")
```

## 15.1 A Simple Example

The visualization interface can be readily used with your models:

```python
[2]:
def model(data):
    m = numpyro.sample("m", dist.Normal(0, 1))
    sd = numpyro.sample("sd", dist.LogNormal(m, 1))
    with numpyro.plate("N", len(data)):
        numpyro.sample("obs", dist.Normal(m, sd), obs=data)
```

```python
[3]:
data = jnp.ones(10)
numpyro.render_model(model, model_args=(data,))
```

**WARNING:jax._src.lib.xla_bridge**: No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)
The visualization can be saved to a file by providing `filename='path'` to `numpyro.render_model`. You can use different formats such as PDF or PNG by changing the filename's suffix. When not saving to a file (filename=None), you can also change the format with `graph.format = 'pdf'` where `graph` is the object returned by `numpyro.render_model`.

```python
[4]: graph = numpyro.render_model(model, model_args=(data,), filename="model.pdf")
```

## 15.2 Tweaking the visualization

As `numpyro.render_model` returns an object of type `graphviz.dot.Digraph`, you can further improve the visualization of this graph. For example, you could use the `unflatten` preprocessor to improve the layout aspect ratio for more complex models.

```python
[5]: def mace(positions, annotations):
    ""
    This model corresponds to the plate diagram in Figure 3 of https://www.aclweb.org/anthology/Q18-1040.pdf.
    ""
    num_annotators = int(np.max(positions)) + 1
    num_classes = int(np.max(annotations)) + 1
    num_items, num_positions = annotations.shape

    with numpyro.plate("annotator", num_annotators):
        epsilon = numpyro.sample("epsilon", dist.Dirichlet(jnp.full(num_classes, 10)))
        theta = numpyro.sample("theta", dist.Beta(0.5, 0.5))

    with numpyro.plate("item", num_items, dim=-2):
        c = numpyro.sample("c", dist.DiscreteUniform(0, num_classes - 1))

    with numpyro.plate("position", num_positions):
        s = numpyro.sample("s", dist.Bernoulli(1 - theta[positions]))
        probs = jnp.where(
            s[... , None] == 0, nn.one_hot(c, num_classes), epsilon[positions]
        )
        numpyro.sample("y", dist.Categorical(probs), obs=annotations)
```

```python
positions = np.array([1, 1, 1, 2, 3, 4, 5])
# fmt: off
annotations = np.array([
    [1, 3, 1, 2, 2, 1, 3, 2, 2, 4, 2, 1, 2, 1,
    1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1, 1,
    1, 3, 1, 2, 2, 4, 2, 2, 3, 1, 1, 1, 2, 1, 2],
    [1, 3, 1, 2, 2, 2, 2, 3, 2, 3, 4, 2, 1, 2, 2,
    1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 3, 1, 1, 1, 1,
    1, 3, 1, 2, 2, 3, 2, 3, 3, 1, 1, 2, 3, 2, 2],
    [1, 3, 2, 2, 2, 2, 2, 3, 2, 2, 4, 2, 1, 2, 2,
    1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 3, 1, 1, 1, 1,
    1, 3, 1, 2, 2, 3, 2, 3, 3, 1, 1, 2, 3, 2, 2],
    [1, 3, 2, 2, 2, 2, 2, 3, 2, 2, 4, 2, 1, 2, 1,
    1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1, 1, 2,
    1, 3, 1, 2, 2, 3, 1, 2, 3, 1, 1, 1, 2, 1, 2],
    [1, 4, 2, 3, 3, 3, 2, 2, 2, 4, 3, 1, 3, 1,
    2, 1, 1, 2, 1, 2, 2, 3, 2, 1, 2, 1, 1, 1,
                               (continues on next page)
```
```python
[1, 3, 1, 2, 3, 4, 2, 3, 3, 1, 1, 2, 2, 1, 2],
[1, 3, 1, 1, 2, 3, 1, 4, 2, 2, 4, 3, 1, 2, 1,
  1, 1, 1, 2, 3, 2, 2, 2, 2, 1, 1, 2, 1, 1, 1,
  1, 2, 1, 2, 2, 3, 2, 2, 4, 1, 1, 1, 2, 1, 2],
[1, 3, 2, 2, 2, 2, 1, 3, 2, 2, 4, 4, 1, 1, 1,
  1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1, 2,
  1, 3, 1, 2, 3, 4, 3, 3, 1, 1, 1, 2, 1, 2],
[1, 4, 2, 1, 2, 1, 3, 3, 4, 3, 1, 2, 1,
  1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 2, 1, 1, 1,
  1, 3, 1, 2, 2, 3, 2, 3, 2, 1, 1, 2, 1, 2],
]).T
# we subtract 1 because the first index starts with 0 in Python
positions -= 1
annotations -= 1

mace_graph = numpyro.render_model(mace, model_args=(positions, annotations))
```

[6]: # default layout
mace_graph
# layout after processing the layout with unflatten
mace_graph.unflatten(stagger=2)

15.2. Tweaking the visualization
15.3 Rendering the parameters

We can render the parameters defined as `numpyro.param` by setting `render_params=True` in `numpyro.render_model`.

```python
[8]: def model(data):
    m = numpyro.param("m", 0.0)
    sd = numpyro.param("sd", 1.0, constraint=constraints.positive)
    lambd = numpyro.sample("lambda", dist.LogNormal(m, sd))
    with numpyro.plate("N", len(data)):
        numpyro.sample("obs", dist.Exponential(lambd), obs=data)
```

```python
[9]: data = jnp.ones(10)
numpyro.render_model(model, model_args=(data,), render_params=True)
```
15.3. Rendering the parameters
15.4 Distribution and Constraint annotations

It is possible to display the distribution of each RV in the generated plot by providing `render_distributions=True` when calling `numpyro.render_model`. The constraints associated with parameters are also displayed when `render_distributions=True`.

```
[10]: numpyro.render_model(
    model, model_args=(data,), render_params=True, render_distributions=True
)
```

In the above plot ‘$\sim$’ denotes the distribution of RV and ‘$\textit{math:in}$’ denotes the constraint of parameter.

15.5 Rendering deterministic sites

We can also render deterministic sites defined via `numpyro.deterministic`. Such sites will be drawn with a dashed-line to distinguish from random sites. The following example illustrates this:

```
[11]: def model(data):
    m = numpyro.sample("m", dist.Normal(0, 1))
    sd = numpyro.sample("sd", dist.LogNormal(m, 1))
    # deterministic site
    m_transformed = numpyro.deterministic("m_transformed", m + 1)
    with numpyro.plate("N", len(data)):
        numpyro.sample("obs", dist.Normal(m_transformed, sd), obs=data)
```
15.6 Rendering neural network’s parameters

```python
[12]: def model(data):
    lambda_base = numpyro.sample("lambda", dist.Normal(0, 1))
    net = flax_module("affine_net", flax_nn.Dense(1), input_shape=(1,))
    lambd = jnp.exp(net(jnp.expand_dims(lambda_base, -1)).squeeze(-1))
    with numpyro.plate("N", len(data)):
        numpyro.sample("obs", dist.Exponential(lambd), obs=data)
```

15.6. Rendering neural network’s parameters
15.7 Overlapping non-nested plates

Note that overlapping non-nested plates may be drawn as multiple rectangles.

```python
[15]: def model():
    plate1 = numpyro.plate("plate1", 2, dim=-2)
    plate2 = numpyro.plate("plate2", 3, dim=-1)
    with plate1:
        x = numpyro.sample("x", dist.Normal(0, 1))
    with plate1, plate2:
        y = numpyro.sample("y", dist.Normal(x, 1))
    with plate2:
        numpyro.sample("z", dist.Normal(y.sum(-2, keepdims=True), 1), obs=jnp.zeros(3))

[16]: numpyro.render_model(model)
```
15.7. Overlapping non-nested plates
BAD POSTERIOR GEOMETRY AND HOW TO DEAL WITH IT

HMC and its variant NUTS use gradient information to draw (approximate) samples from a posterior distribution. These gradients are computed in a particular coordinate system, and different choices of coordinate system can make HMC more or less efficient. This is analogous to the situation in constrained optimization problems where, for example, parameterizing a positive quantity via an exponential versus softplus transformation results in distinct optimization dynamics.

Consequently it is important to pay attention to the geometry of the posterior distribution. Reparameterizing the model (i.e. changing the coordinate system) can make a big practical difference for many complex models. For the most complex models it can be absolutely essential. For the same reason it can be important to pay attention to some of the hyperparameters that control HMC/NUTS (in particular the max_tree_depth and dense_mass).

In this tutorial we explore models with bad posterior geometries—and what one can do to get achieve better performance—with a few concrete examples.

```python
[1]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[1]: from functools import partial
    import numpy as np
    from jax import random
    import jax.numpy as jnp
    import numpyro
    from numpyro.diagnostics import summary
    import numpyro.distributions as dist
    from numpyro.infer import MCMC, NUTS

    assert numpyro.__version__.startswith("0.15.2")

    # NB: replace cpu by gpu to run this notebook on gpu
    numpyro.set_platform("cpu")

We begin by writing a helper function to do NUTS inference.

```python
[2]: def run_inference(
        model, num_warmup=1000, num_samples=1000, max_tree_depth=10, dense_mass=False
    ):
    kernel = NUTS(model, max_tree_depth=max_tree_depth, dense_mass=dense_mass)
    mcmc = MCMC(
        kernel,
```
16.1 Evaluating HMC/NUTS

In general it is difficult to assess whether the samples returned from HMC or NUTS represent accurate (approximate) samples from the posterior. Two general rules of thumb, however, are to look at the effective sample size (ESS) and $r_{\text{hat}}$ diagnostics returned by `mcmc.print_summary()`. If we see values of $r_{\text{hat}}$ in the range $(1.0, 1.05)$ and effective sample sizes that are comparable to the total number of samples `num_samples` (assuming thinning=1) then we have good reason to believe that HMC is doing a good job. If, however, we see low effective sample sizes or large $r_{\text{hat}}$s for some of the variables (e.g. $r_{\text{hat}} = 1.15$) then HMC is likely struggling with the posterior geometry. In the following we will use $r_{\text{hat}}$ as our primary diagnostic metric.

16.2 Model reparameterization

16.2.1 Example #1

We begin with an example (horseshoe regression; see examples/horseshoe_regression.py for a complete example script) where reparameterization helps a lot. This particular example demonstrates a general reparameterization strategy that is useful in many models with hierarchical/multi-level structure. For more discussion of some of the issues that can arise in hierarchical models see reference [1].

To deal with the bad geometry that results from this coordinate system we change coordinates using the following re-write logic. Instead of

$$\beta \sim \text{Normal}(0, \lambda \tau)$$

we write

$$\beta' \sim \text{Normal}(0, 1)$$
and

\[ \beta \equiv \lambda \tau \beta' \]

where \( \beta \) is now defined \textit{deterministically} in terms of \( \lambda, \tau, \) and \( \beta' \). In effect we’ve changed to a coordinate system where the different latent variables are less correlated with one another. In this new coordinate system we can expect HMC with a diagonal mass matrix to behave much better than it would in the original coordinate system.

There are basically two ways to implement this kind of reparameterization in NumPyro:

- manually (i.e. by hand)
- using \texttt{numpyro.infer.reparam}, which automates a few common reparameterization strategies

To begin with let’s do the reparameterization by hand.

---

[4]: # In this reparameterized model none of the parameters of the distributions # explicitly depend on other parameters. This model is exactly equivalent # to \_unrep\_hs\_model but is expressed in a different coordinate system.

```python
def \_rep\_hs\_model1(X, Y):
    lambdas = numpyro.sample("lambdas", dist.HalfCauchy(jnp.ones(X.shape[1])))
    tau = numpyro.sample("tau", dist.HalfCauchy(jnp.ones(1)))
    unscaled_betas = numpyro.sample("unscaled\_betas", dist.Normal(scale=jnp.ones(X.shape[1])))
    scaled_betas = numpyro.deterministic("betas", tau * lambdas * unscaled_betas)
    mean_function = jnp.dot(X, scaled_betas)
numpyro.sample("Y", dist.Normal(mean_function, 0.05), obs=Y)
```

Next we do the reparameterization using \texttt{numpyro.infer.reparam}. There are at least two ways to do this. First let’s use \texttt{LocScaleReparam}.

---

[5]: from numpyro.infer.reparam import LocScaleReparam

```python
# LocScaleReparam with centered=0 fully "decenters" the prior over betas.
config = {"betas": LocScaleReparam(centered=0)}
# The coordinate system of this model is equivalent to that in \_rep\_hs\_model1 above.

\_rep\_hs\_model2 = numpyro.handlers.reparam(\_unrep\_hs\_model, config=config)
```

To show the versatility of the \texttt{numpyro.infer.reparam} library let’s do the reparameterization using \texttt{TransformReparam} instead.

---

[6]: from numpyro.distributions.transforms import AffineTransform
from numpyro.infer.reparam import TransformReparam

```python
# In this reparameterized model none of the parameters of the distributions # explicitly depend on other parameters. This model is exactly equivalent # to \_unrep\_hs\_model but is expressed in a different coordinate system.

def \_rep\_hs\_model3(X, Y):
    lambdas = numpyro.sample("lambdas", dist.HalfCauchy(jnp.ones(X.shape[1])))
    tau = numpyro.sample("tau", dist.HalfCauchy(jnp.ones(1)))

    # instruct NumPyro to do the reparameterization automatically.
    reparam_config = {"betas": TransformReparam()}
    with numpyro.handlers.reparam(config=reparam_config):
```

(continues on next page)
betas_root_variance = tau * lambdas
# in order to use TransformReparam we have to express the prior
# over betas as a TransformedDistribution
betas = numpyro.sample("betas",
    dist.TransformedDistribution(
        dist.Normal(0.0, jnp.ones(X.shape[1])),
        AffineTransform(0.0, betas_root_variance),
    ),
)

mean_function = jnp.dot(X, betas)
numpyro.sample("Y", dist.Normal(mean_function, 0.05), obs=Y)

Finally we verify that _rep_hs_model1, _rep_hs_model2, and _rep_hs_model3 do indeed achieve better r_hats than _unrep_hs_model.

[8]: # create fake dataset
    X = np.random.RandomState(0).randn(100, 500)
    Y = X[:, 0]
    print("unreparameterized model (very bad r_hats)")
    run_inference(partial(_unrep_hs_model, X, Y))
    print("\nreparameterized model with manual reparameterization (good r_hats)")
    run_inference(partial(_rep_hs_model1, X, Y))
    print("\nreparameterized model with LocScaleReparam (good r_hats)")
    run_inference(partial(_rep_hs_model2, X, Y))
    print("\nreparameterized model with TransformReparam (good r_hats)")
    run_inference(partial(_rep_hs_model3, X, Y))

unreparameterized model (very bad r_hats)
[betas] max r_hat: 1.0775
[lambdas] max r_hat: 3.2450
[tau] max r_hat: 2.1926

reparameterized model with manual reparameterization (good r_hats)
[betas] max r_hat: 1.0074
[lambdas] max r_hat: 1.0146
[tau] max r_hat: 1.0036
[unscaled_betas] max r_hat: 1.0059

reparameterized model with LocScaleReparam (good r_hats)
[betas] max r_hat: 1.0103
[betas_decentered] max r_hat: 1.0060
[lambdas] max r_hat: 1.0124
[tau] max r_hat: 0.9998

reparameterized model with TransformReparam (good r_hats)
[betas] max r_hat: 1.0087
[betas_base] max r_hat: 1.0080

(continues on next page)
16.2.2 Aside: numpyro.deterministic

In `_rep_hs_model1` above we used `numpyro.deterministic` to define `scaled_betas`. We note that using this primitive is not strictly necessary; however, it has the consequence that `scaled_betas` will appear in the trace and will thus appear in the summary reported by `mcmc.print_summary()`. In other words we could also have written:

```python
scaled_betas = tau * lambdas * unscaled_betas
```

without invoking the `deterministic` primitive.

16.3 Mass matrices

By default HMC/NUTS use diagonal mass matrices. For models with complex geometries it can pay to use a richer set of mass matrices.

16.3.1 Example #2

In this first simple example we show that using a full-rank (i.e. dense) mass matrix leads to a better $r_{hat}$.

```
# Because rho is very close to 1.0 the posterior geometry
# is extremely skewed and using the "diagonal" coordinate system
# implied by dense_mass=False leads to bad results
rho = 0.9999
cov = jnp.array([[10.0, rho], [rho, 0.1]])

def mvn_model():
    numpyro.sample("x", dist.MultivariateNormal(jnp.zeros(2), covariance_matrix=cov))

print("dense_mass = False (bad r_hat)")
run_inference(mvn_model, dense_mass=False, max_tree_depth=3)

print("dense_mass = True (good r_hat)")
run_inference(mvn_model, dense_mass=True, max_tree_depth=3)
```

```bash
dense_mass = False (bad r_hat)
[x] max r_hat: 1.3810

dense_mass = True (good r_hat)
[x] max r_hat: 0.9992
```
16.3.2 Example #3

Using `dense_mass=True` can be very expensive when the dimension of the latent space $D$ is very large. In addition it can be difficult to estimate a full-rank mass matrix with $D^2$ parameters using a moderate number of samples if $D$ is large. In these cases `dense_mass=True` can be a poor choice. Luckily, the argument `dense_mass` can also be used to specify structured mass matrices that are richer than a diagonal mass matrix but more constrained (i.e. have fewer parameters) than a full-rank mass matrix (see the docs). In this second example we show how we can use `dense_mass` to specify such a structured mass matrix.

```python
[10]: rho = 0.9
    cov = jnp.array([[10.0, rho], [rho, 0.1]])

# In this model x1 and x2 are highly correlated with one another
# but not correlated with y at all.

def partially_correlated_model():
    x1 = numpyro.sample("x1", dist.MultivariateNormal(jnp.zeros(2), covariance_matrix=cov))
    x2 = numpyro.sample("x2", dist.MultivariateNormal(jnp.zeros(2), covariance_matrix=cov))
    numpyro.sample("y", dist.Normal(jnp.zeros(100), 1.0))
    numpyro.sample("obs", dist.Normal(x1 - x2, 0.1), jnp.ones(2))
```

Now let's compare two choices of `dense_mass`.

```python
[11]: print("dense_mass = False (very bad r_hats)")
run_inference(partially_correlated_model, dense_mass=False, max_tree_depth=3)

print("\ndense_mass = True (bad r_hats)"
run_inference(partially_correlated_model, dense_mass=True, max_tree_depth=3)

# We use dense_mass=[("x1", "x2")] to specify
# a structured mass matrix in which the y-part of the mass matrix is diagonal
# and the (x1, x2) block of the mass matrix is full-rank.

# Graphically:
#
#   x1  x2  y
#   | * * 0 |
#   x2 | * * 0 |
#   y  | 0 0 * |

print("\nstructured mass matrix (good r_hats)"
run_inference(partially_correlated_model, dense_mass=[("x1", "x2")], max_tree_depth=3)
```

```
dense_mass = False (very bad r_hats)
[x1] max r_hat: 1.5882
[x2] max r_hat: 1.5410
[y] max r_hat: 2.0179

dense_mass = True (bad r_hats)
[x1] max r_hat: 1.0697
(continues on next page)
```
structured mass matrix (good r_hats)

<table>
<thead>
<tr>
<th></th>
<th>max r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1.0023</td>
</tr>
<tr>
<td>x2</td>
<td>1.0024</td>
</tr>
<tr>
<td>y</td>
<td>1.0030</td>
</tr>
</tbody>
</table>

16.4 max_tree_depth

The hyperparameter `max_tree_depth` can play an important role in determining the quality of posterior samples generated by NUTS. The default value in NumPyro is `max_tree_depth=10`. In some models, in particular those with especially difficult geometries, it may be necessary to increase `max_tree_depth` above 10. In other cases where computing the gradient of the model log density is particularly expensive, it may be necessary to decrease `max_tree_depth` below 10 to reduce compute. As an example where large `max_tree_depth` is essential, we return to a variant of example #2. (We note that in this particular case another way to improve performance would be to use `dense_mass=True`).

16.4.1 Example #4

```
# Because rho is very close to 1.0 the posterior geometry is extremely
# skewed and using small max_tree_depth leads to bad results.
rho = 0.999
dim = 200
cov = rho * jnp.ones((dim, dim)) + (1 - rho) * jnp.eye(dim)

def mvn_model():
    numpyro.sample("x", dist.MultivariateNormal(jnp.zeros(dim), covariance_matrix=cov))

print("max_tree_depth = 5 (bad r_hat)")
run_inference(mvn_model, max_tree_depth=5)

print("max_tree_depth = 10 (good r_hat)")
run_inference(mvn_model, max_tree_depth=10)
```

max_tree_depth = 5 (bad r_hat)
[ ] max r_hat: 1.1159
max_tree_depth = 10 (good r_hat)
[ ] max r_hat: 1.0166
16.5 Other strategies

- In some cases it can make sense to use variational inference to learn a new coordinate system. For details see examples/neutra.py and reference [2].

16.6 References


TRUNCATED AND FOLDED DISTRIBUTIONS

This tutorial will cover how to work with truncated and folded distributions in NumPyro. It is assumed that you’re already familiar with the basics of NumPyro. To get the most out of this tutorial you’ll need some background in probability.

17.1 Table of contents

- 0. Setup
- 1. What is a truncated distribution?
- 2. What is a folded distribution?
- 3. Sampling from truncated and folded distributions
- 4. Ready-to-use truncated and folded distributions
- 5. Building your own truncated distributions
  - 5.1 Recap of NumPyro distributions
  - 5.2 Right-truncated normal
  - 5.3 Left-truncated Poisson
- 6. References and related material

17.2 Setup

To run this notebook, we are going to need the following imports

```python
[ ]: !pip install -q git+https://github.com/pyro-ppl/numpyro.git

[2]:
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import poisson as sp_poisson

import jax
from jax import lax, random
import jax.numpy as jnp
from jax.scipy.special import ndtri
from jax.scipy.stats import norm, poisson
```

(continues on next page)
import numpyro
import numpyro.distributions as dist
from numpyro.distributions import (Distribution, FoldedDistribution, SoftLaplace, StudentT, TruncatedDistribution, TruncatedNormal, constraints,
)
from numpyro.distributions.util import promote_shapes
from numpyro.infer import MCMC, NUTS, DiscreteHMCGibbs, Predictive

numpyro.enable_x64()
RNG = random.PRNGKey(0)
PRIOR_RNG, MCMC_RNG, PRED_RNG = random.split(RNG, 3)
MCMC_KWARGS = dict(
    num_warmup=2000,
    num_samples=2000,
    num_chains=4,
    chain_method="sequential",
)

17.3 1. What are truncated distributions?

The **support** of a probability distribution is the set of values in the domain with non-zero probability. For example, the support of the normal distribution is the whole real line (even if the density gets very small as we move away from the mean, technically speaking, it is never quite zero). The support of the uniform distribution, as coded in jax. random.uniform with the default arguments, is the interval \([0, 1)\), because any value outside of that interval has zero probability. The support of the Poisson distribution is the set of non-negative integers, etc.

**Truncating** a distribution makes its support smaller so that any value outside our desired domain has zero probability. In practice, this can be useful for modelling situations in which certain biases are introduced during data collection. For example, some physical detectors only get triggered when the signal is above some minimum threshold, or sometimes the detectors fail if the signal exceeds a certain value. As a result, the **observed values are constrained to be within a limited range of values**, even though the true signal does not have the same constraints. See, for example, section 3.1 of *Information Theory and Learning Algorithms* by David Mackay. Naively, if \(S\) is the support of the original density \(p_Y(y)\), then by truncating to a new support \(T \subset S\) we are effectively defining a new random variable \(Z\) for which the density is

\[
p_Z(z) \propto \begin{cases} 
p_Y(z) & \text{if } z \text{ is in } T \\
0 & \text{if } z \text{ is outside } T \end{cases} \tag{17.1}
\]

The reason for writing a \(\propto\) (proportional to) sign instead of a strict equation is that, defined in the above way, the resulting function does not integrate to 1 and so it cannot be strictly considered a probability density. To make it into a probability density we need to re-distribute the truncated mass among the part of the distribution that remains. To do this, we simply re-weight every point by the same constant:

\[
p_Z(z) = \begin{cases} 
\frac{1}{M}p_Y(z) & \text{if } z \text{ is in } T \\
0 & \text{if } z \text{ is outside } T \end{cases} \tag{17.2}
\]
where \( M = \int_T p_Y(y) \, dy \).

In practice, the truncation is often one-sided. This means that if, for example, the support before truncation is the interval \((a, b)\), then the support after truncation is of the form \((a, c)\) or \((c, b)\), with \(a < c < b\). The figure below illustrates a left-sided truncation at zero of a normal distribution \(N(1, 1)\).

The original distribution (left side) is truncated at the vertical dotted line. The truncated mass (orange region) is redistributed in the new support (right side image) so that the total area under the curve remains equal to 1 even after truncation. This method of re-weighting ensures that the density ratio between any two points, \(p(a)/p(b)\) remains the same before and after the reweighting is done (as long as the points are inside the new support, of course).

**Note:** Truncated data is different from *censored* data. Censoring also hides values that are outside some desired support but, contrary to truncated data, we know when a value has been censored. The typical example is the household scale which does not report values above 300 pounds. Censored data will not be covered in this tutorial.

### 17.4 2. What is a folded distribution?

**Folding** is achieved by taking the absolute value of a random variable, \(Z = |Y|\). This obviously modifies the support of the original distribution since negative values now have zero probability:

\[
p_Z(z) = \begin{cases} 
p_Y(z) + p_Y(-z) & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}
\]

(17.3)

The figure below illustrates a folded normal distribution \(N(1, 1)\).

As you can see, the resulting distribution is different from the truncated case. In particular, the density ratio between points, \(p(a)/p(b)\), is in general not the same after folding. For some examples in which folding is relevant see references.
If the original distribution is symmetric around zero, then folding and truncating at zero have the same effect.

### 17.5 3. Sampling from truncated and folded distributions

**Truncated distributions**

Usually, we already have a sampler for the pre-truncated distribution (e.g. `np.random.normal`). So, a seemingly simple way of generating samples from the truncated distribution would be to sample from the original distribution, and then discard the samples that are outside the desired support. For example, if we wanted samples from a normal distribution truncated to the support $(-\infty, 1)$, we'd simply do:

```python
upper = 1
samples = np.random.normal(size=1000)
truncated_samples = samples[samples < upper]
```

This is called rejection sampling but it is not very efficient. If the region we truncated had a sufficiently high probability mass, then we’d be discarding a lot of samples and it might be a while before we accumulate sufficient samples for the truncated distribution. For example, the above snippet would only result in approximately 840 truncated samples even though we initially drew 1000. This can easily get a lot worse for other combinations of parameters. A more efficient approach is to use a method known as inverse transform sampling. In this method, we first sample from a uniform distribution in (0, 1) and then transform those samples with the inverse cumulative distribution of our truncated distribution. This method ensures that no samples are wasted in the process, though it does have the slight complication that we need to calculate the inverse CDF (ICDF) of our truncated distribution. This might sound too complicated at first but, with a bit of algebra, we can often calculate the truncated ICDF in terms of the untruncated ICDF. The untruncated ICDF for many distributions is already available.

**Folded distributions**

This case is a lot simpler. Since we already have a sampler for the pre-folded distribution, all we need to do is to take the absolute value of those samples:

```python
samples = np.random.normal(size=1000)
folded_samples = np.abs(samples)
```

### 17.6 4. Ready to use truncated and folded distributions

The later sections in this tutorial will show you how to construct your own truncated and folded distributions, but you don’t have to reinvent the wheel. NumPyro has a bunch of truncated distributions already implemented.

Suppose, for example, that you want a normal distribution truncated on the right. For that purpose, we use the `TruncatedNormal` distribution. The parameters of this distribution are `loc` and `scale`, corresponding to the `loc` and `scale` of the untruncated normal, and `low` and/or `high` corresponding to the truncation points. Importantly, the `low` and `high` are keyword only arguments, only `loc` and `scale` are valid as positional arguments. This is how you can use this class in a model:

```python
[3]: def truncated_normal_model(num_observations, high, x=None):
    loc = numpyro.sample("loc", dist.Normal())
    scale = numpyro.sample("scale", dist.LogNormal())
    with numpyro.plate("observations", num_observations):
        numpyro.sample("x", TruncatedNormal(loc, scale, high=high), obs=x)
```
Let’s now check that we can use this model in a typical MCMC workflow.

**Prior simulation**

```python
[4]: high = 1.2
num_observations = 250
num_prior_samples = 100

prior = Predictive(truncated_normal_model, num_samples=num_prior_samples)
prior_samples = prior(PRIOR_RNG, num_observations, high)
```

**Inference**

To test our model, we run mcmc against some synthetic data. The synthetic data can be any arbitrary sample from the prior simulation.

```python
[5]: # -- select an arbitrary prior sample as true data
true_idx = 0
true_loc = prior_samples["loc"][true_idx]
true_scale = prior_samples["scale"][true_idx]
true_x = prior_samples["x"][true_idx]

[6]: plt.hist(true_x.copy(), bins=20)
plt.axvline(high, linestyle=":\", color="k")
plt.xlabel("x")
plt.show()
```

```python
[7]: # --- Run MCMC and check estimates and diagnostics
mcmc = MCMC(NUTS(truncated_normal_model), **MCMC_KWARGS)
mcmc.run(MCMC_RNG, num_observations, high, true_x)
mcmc.print_summary()

# --- Compare to ground truth
print(f"True loc : {true_loc:.2f}"
print(f"True scale: {true_scale:.2f}"
```

```text
17.6. 4. Ready to use truncated and folded distributions
```
Removing the truncation

Once we have inferred the parameters of our model, a common task is to understand what the data would look like without the truncation. In this example, this is easily done by simply “pushing” the value of $\text{high}$ to infinity.

```python
[8]: pred = Predictive(truncated_normal_model, posterior_samples=mcmc.get_samples())
pred_samples = pred(PRED_RNG, num_observations, high=float("inf"))
```

Let’s finally plot these samples and compare them to the original, observed data.

```python
[9]: # thin the samples to not saturate matplotlib
samples_thinned = pred_samples["x"].ravel()[::1000]

[10]: f, axes = plt.subplots(1, 2, figsize=(15, 5), sharex=True)
    
    axes[0].hist(
        samples_thinned.copy(), label="Untruncated posterior", bins=20, density=True
    )
    axes[0].set_title("Untruncated posterior")
    
    vals, bins, _ = axes[1].hist(
        samples_thinned[samples_thinned < high].copy(),
        label="Tail of untruncated posterior",
        bins=10,
        density=True,
    )
    axes[1].hist(
        true_x.copy(), bins=bins, label="Observed, truncated data", density=True, alpha=0.5
    )
    axes[1].set_title("Comparison to observed data")
    
    for ax in axes:
        ax.axvline(high, linestyle=":", color="k", label="Truncation point")
```

Number of divergences: 0
True loc : -0.56
True scale: 1.4
The plot on the left shows data simulated from the posterior distribution with the truncation removed, so we are able to see how the data would look like if it were not truncated. To sense check this, we discard the simulated samples that are above the truncation point and make histogram of those and compare it to a histogram of the true data (right plot).

**The TruncatedDistribution class**

The source code for the `TruncatedNormal` in NumPyro uses a class called `TruncatedDistribution` which abstracts away the logic for `sample` and `log_prob` that we will discuss in the next sections. At the moment, though, this logic only works continuous, symmetric distributions with real support.

We can use this class to quickly construct other truncated distributions. For example, if we need a truncated SoftLaplace we can use the following pattern:

```python
def TruncatedSoftLaplace(loc=0.0, scale=1.0, *, low=None, high=None, validate_args=None):
    return TruncatedDistribution(base_dist=SoftLaplace(loc, scale),
                                 low=low, high=high,
                                 validate_args=validate_args,
                                )
```

```python
def truncated_soft_laplace_model(num_observations, high, x=None):
    loc = numpyro.sample("loc", dist.Normal())
    scale = numpyro.sample("scale", dist.LogNormal())
    with numpyro.plate("obs", num_observations):
        numpyro.sample("x", TruncatedSoftLaplace(loc, scale, high=high), obs=x)
```

And, as before, we check that we can use this model in the steps of a typical workflow:

```python
high = 2.3
num_observations = 200
num_prior_samples = 100
```
prior = Predictive(truncated_soft_laplace_model, num_samples=num_prior_samples)
prior_samples = prior(PRIOR_RNG, num_observations, high)

true_idx = 0
true_x = prior_samples["x"][true_idx]
true_loc = prior_samples["loc"][true_idx]
true_scale = prior_samples["scale"][true_idx]

mcmc = MCMC(
    NUTS(truncated_soft_laplace_model),
    **MCMC_KWARGS,
)

mcmc.run(
    MCMC_RNG,
    num_observations,
    high,
    true_x,
)

mcmc.print_summary()

print(f"True loc : {true_loc:.2f}")
print(f"True scale: {true_scale:.2f}")

<table>
<thead>
<tr>
<th>sample: 100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4000/4000 [00:02&lt;00:00, 1745.70it/s, 1 steps of size 6.78e-01. acc. prob=0.93]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sample: 100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4000/4000 [00:00&lt;00:00, 9294.56it/s, 1 steps of size 7.02e-01. acc. prob=0.93]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sample: 100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4000/4000 [00:00&lt;00:00, 10412.30it/s, 1 steps of size 7.20e-01. acc. prob=0.92]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sample: 100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4000/4000 [00:00&lt;00:00, 10583.85it/s, 3 steps of size 7.01e-01. acc. prob=0.93]</th>
</tr>
</thead>
</table>

| mean  std  median  5.0%  95.0%  n_eff  r_hat  
|-----  ----  -------  ----  ----  -----  ----  
| loc  -0.37  0.17  -0.38  -0.65  -0.10  4034.96  1.00  
| scale  1.46  0.12  1.45  1.27  1.65  3618.77  1.00  

Number of divergences: 0
True loc : -0.56
True scale: 1.4

Important

The sample method of the TruncatedDistribution class relies on inverse-transform sampling. This has the implicit requirement that the base distribution should have an icdf method already available. If this is not the case, we will not be able to call the sample method on any instances of our distribution, nor use it with the Predictive class. However, the log_prob method only depends on the cdf method (which is more frequently available than the icdf).
If the log_prob method is available, then we can use our distribution as prior/likelihood in a model.

The FoldedDistribution class

Similar to truncated distributions, NumPyro has the FoldedDistribution class to help you quickly construct folded distributions. Popular examples of folded distributions are the so-called “half-normal”, “half-student” or “half-cauchy”. As the name suggests, these distributions keep only (the positive) half of the distribution. Implicit in the name of these “half” distributions is that they are centered at zero before folding. But, of course, you can fold a distribution even if its not centered at zero. For instance, this is how you would define a folded student-t distribution.

```python
[14]: def FoldedStudentT(df, loc=0.0, scale=1.0):
    return FoldedDistribution(StudentT(df, loc=loc, scale=scale))

[15]: def folded_student_model(num_observations, x=None):
    df = numpyro.sample("df", dist.Gamma(6, 2))
    loc = numpyro.sample("loc", dist.Normal())
    scale = numpyro.sample("scale", dist.LogNormal())
    with numpyro.plate("obs", num_observations):
        numpyro.sample("x", FoldedStudentT(df, loc, scale), obs=x)
```

And we check that we can use our distribution in a typical workflow:

```python
[16]: # --- prior sampling
    num_observations = 500
    num_prior_samples = 100
    prior = Predictive(folded_student_model, num_samples=num_prior_samples)
    prior_samples = prior(PRIOR_RNG, num_observations)

    # --- choose any prior sample as the ground truth
    true_idx = 0
    true_df = prior_samples["df"][true_idx]
    true_loc = prior_samples["loc"][true_idx]
    true_scale = prior_samples["scale"][true_idx]
    true_x = prior_samples["x"][true_idx]

    # --- do inference with MCMC
    mcmc = MCMC(  
        NUTS(folded_student_model),  
        **MCMC_KWARGS,  
    )
    mcmc.run(MCMC_RNG, num_observations, true_x)

    # --- Check diagnostics
    mcmc.print_summary()

    # --- Compare to ground truth:
    print(f"True df : {true_df:.2f}"
    print(f"True loc : {true_loc:.2f}"
    print(f"True scale: {true_scale:.2f}"
```

sample: 100
...
```
(continues on next page)
Number of divergences: 33
True df : 3.01
True loc : 0.37
True scale: 2.41

17.7.5. Building your own truncated distribution

If the TruncatedDistribution and FoldedDistribution classes are not sufficient to solve your problem, you might want to look into writing your own truncated distribution from the ground up. This can be a tedious process, so this section will give you some guidance and examples to help you with it.

17.7.1.5.1 Recap of NumPyro distributions

A NumPyro distribution should subclass Distribution and implement a few basic ingredients:

Class attributes

The class attributes serve a few different purposes. Here we will mainly care about two: 1. arg_constraints: Impose some requirements on the parameters of the distribution. Errors are raised at instantiation time if the parameters passed do not satisfy the constraints. 2. support: It is used in some inference algorithms like MCMC and SVI with autoguides, where we need to perform the algorithm in the unconstrained space. Knowing the support, we can automatically reparametrize things under the hood.

We’ll explain other class attributes as we go.

The __init__ method

This is where we define the parameters of the distribution. We also use jax and lax to promote the parameters to shapes that are valid for broadcasting. The __init__ method of the parent class is also required because that's where the validation of our parameters is done.

The log_prob method

Implementing the log_prob method ensures that we can do inference. As the name suggests, this method returns the logarithm of the density evaluated at the argument.

The sample method
This method is used for drawing independent samples from our distribution. It is particularly useful for doing prior and posterior predictive checks. Note, in particular, that this method is not needed if you only need to use your distribution as prior in a model - the log_prob method will suffice.

The place-holder code for any of our implementations can be written as

```python
class MyDistribution(Distribution):
    # class attributes
    arg_constraints = {}
    support = None
    def __init__(self):
        pass
    def log_prob(self, value):
        pass
    def sample(self, key, sample_shape=()):
        pass
```

17.7.2 5.2 Example: Right-truncated normal

We are going to modify a normal distribution so that its new support is of the form \((-\infty, \text{high})\), with \text{high} a real number. This could be done with the TruncatedNormal distribution but, for the sake of illustration, we are not going to rely on it. We'll call our distribution RightTruncatedNormal. Let's write the skeleton code and then proceed to fill in the blanks.

```python
class RightTruncatedNormal(Distribution):
    # <class attributes>
    def __init__(self):
        pass
    def log_prob(self, value):
        pass
    def sample(self, key, sample_shape=()):
        pass
```

**Class attributes**

Remember that a non-truncated normal distribution is specified in NumPyro by two parameters, loc and scale, which correspond to the mean and standard deviation. Looking at the source code for the Normal distribution we see the following lines:

```
arg_constraints = {"loc": constraints.real, "scale": constraints.positive}
support = constraints.real
reparameterized_params = ["loc", "scale"]
```

The reparameterized_params attribute is used by variational inference algorithms when constructing gradient estimators. The parameters of many common distributions with continuous support (e.g. the Normal distribution) are reparameterizable, while the parameters of discrete distributions are not. Note that reparameterized_params is irrelevant for MCMC algorithms like HMC. See SVI Part III for more details.

We must adapt these attributes to our case by including the "high" parameter, but there are two issues we need to deal with:
1. `constraints.real` is a bit too restrictive. We’d like `jnp.inf` to be a valid value for `high` (equivalent to no truncation), but at the moment infinity is not a valid real number. We deal with this situation by defining our own constraint. The source code for `constraints.real` is easy to imitate:

```python
class _RightExtendedReal(constraints.Constraint):
    """Any number in the interval (-inf, inf]."""
    def __call__(self, x):
        return (x == x) & (x != float("-inf"))
    def feasible_like(self, prototype):
        return jnp.zeros_like(prototype)
right_extended_real = _RightExtendedReal()
```

2. `support` can no longer be a class attribute as it will depend on the value of `high`. So instead we implement it as a dependent property.

Our distribution then looks as follows:

```python
class RightTruncatedNormal(Distribution):
    arg_constraints = {
        "loc": constraints.real,
        "scale": constraints.positive,
        "high": right_extended_real,
    }
    reparametrized_params = ["loc", "scale", "high"]
    # ...

    @constraints.dependent_property
    def support(self):
        return constraints.lower_than(self.high)
```

The `__init__` method

Once again we take inspiration from the source code for the normal distribution. The key point is the use of `lax` and `jax` to check the shapes of the arguments passed and make sure that such shapes are consistent for broadcasting. We follow the same pattern for our use case – all we need to do is include the `high` parameter.

In the source implementation of `Normal`, both parameters `loc` and `scale` are given defaults so that one recovers a standard normal distribution if no arguments are specified. In the same spirit, we choose `float("inf")` as a default for `high` which would be equivalent to no truncation.

```python
# ...
    def __init__(self, loc=0.0, scale=1.0, high=float("inf"), validate_args=None):
        batch_shape = lax.broadcast_shapes(
            jnp.shape(loc),
            jnp.shape(scale),
            jnp.shape(high),
        )
        self.loc, self.scale, self.high = promote_shapes(loc, scale, high)
        super().__init__(batch_shape, validate_args=validate_args)
    # ...
```
The log_prob method

For a truncated distribution, the log density is given by

\[
\log p_Z(z) = \begin{cases} 
\log p_Y(z) - \log M & \text{if } z \text{ is in } T \\
-\infty & \text{if } z \text{ is outside } T
\end{cases}
\] (17.4)

where, again, \(p_Z\) is the density of the truncated distribution, \(p_Y\) is the density before truncation, and \(M = \int_T p_Y(y) dy\). For the specific case of truncating the normal distribution to the interval \((-\infty, \text{high})\), the constant \(M\) is equal to the cumulative density evaluated at the truncation point. We can easily implement this log-density method because \texttt{jax.scipy.stats} already has a \texttt{norm} module that we can use.

```python
# ...
    def log_prob(self, value):
        log_m = norm.logcdf(self.high, self.loc, self.scale)
        log_p = norm.logpdf(value, self.loc, self.scale)
        return jnp.where(value < self.high, log_p - log_m, -jnp.inf)
# ...
```

The sample method

To implement the sample method using inverse-transform sampling, we need to also implement the inverse cumulative distribution function. For this, we can use the \texttt{ndtri} function that lives inside \texttt{jax.scipy.special}. This function returns the inverse cdf for the standard normal distribution. We can do a bit of algebra to obtain the inverse cdf of the truncated, non-standard normal. First recall that if \(X \sim \text{Normal}(0, 1)\) and \(Y = \mu + \sigma X\), then \(Y \sim \text{Normal}(\mu, \sigma)\). Then if \(Z\) is the truncated \(Y\), its cumulative density is given by:

\[
F_Z(y) = \int_{-\infty}^y p_Z(r) dr = \frac{1}{M} \int_{-\infty}^y p_Y(s) ds \quad \text{if } y < \text{high} = \frac{1}{M} F_Y(y)
\] (17.5)

And so its inverse is

\[
F_Z^{-1}(u) = \left(\frac{1}{M} F_Y\right)^{-1}(u) = F_Y^{-1}(Mu) = F_{\mu+\sigma X}^{-1}(Mu) = \mu + \sigma F_X^{-1}(Mu)
\] (17.6)

The translation of the above math into code is

```python
# ...
    def sample(self, key, sample_shape=()):
        shape = sample_shape + self.batch_shape
        minval = jnp.finfo(jnp.result_type(float)).tiny
        u = random.uniform(key, shape, minval=minval)
        return self.icdf(u)

def icdf(self, u):
    m = norm.cdf(self.high, self.loc, self.scale)
    return self.loc + self.scale * ndtri(m * u)
# ...
```

With everything in place, the final implementation is as below.

```python
[17]: class _RightExtendedReal(constraints.Constraint):
    
    Any number in the interval \((-\infty, \infty]\).
    
(continues on next page)
```python
def __call__(self, x):
    return (x == x) & (x != float("-inf"))

def feasible_like(self, prototype):
    return jnp.zeros_like(prototype)

right_extended_real = _RightExtendedReal()

class RightTruncatedNormal(Distribution):
    
    A truncated Normal distribution.
    
    :param numpy.ndarray loc: location parameter of the untruncated normal
    :param numpy.ndarray scale: scale parameter of the untruncated normal
    :param numpy.ndarray high: point at which the truncation happens
    
    arg_constraints = {
        "loc": constraints.real,
        "scale": constraints.positive,
        "high": right_extended_real,
    }
    reparametrized_params = ["loc", "scale", "high"]

def __init__(self, loc=0.0, scale=1.0, high=float("inf"), validate_args=True):
    batch_shape = lax.broadcast_shapes(
        jnp.shape(loc),
        jnp.shape(scale),
        jnp.shape(high),
    )
    self.loc, self.scale, self.high = promote_shapes(loc, scale, high)
    super().__init__(batch_shape, validate_args=validate_args)

def log_prob(self, value):
    log_m = norm.logcdf(self.high, self.loc, self.scale)
    log_p = norm.logpdf(value, self.loc, self.scale)
    return jnp.where(value < self.high, log_p - log_m, -jnp.inf)

def sample(self, key, sample_shape=()):
    shape = sample_shape + self.batch_shape
    minval = jnp.finfo(jnp.result_type(float)).tiny
    u = random.uniform(key, shape, minval=minval)
    return self.icdf(u)

def icdf(self, u):
    m = norm.cdf(self.high, self.loc, self.scale)
    return self.loc + self.scale * ndtri(m * u)

@constraints.dependent_property
def support(self):
    return constraints.less_than(self.high)
```

Chapter 17. Truncated and folded distributions
Let’s try it out!

```python
[18]: def truncated_normal_model(num_observations, x=None):
    loc = numpyro.sample("loc", dist.Normal())
    scale = numpyro.sample("scale", dist.LogNormal())
    high = numpyro.sample("high", dist.Normal())
    with numpyro.plate("observations", num_observations):
        numpyro.sample("x", RightTruncatedNormal(loc, scale, high), obs=x)
```

```python
[19]: num_observations = 1000
    num_prior_samples = 100
    prior = Predictive(truncated_normal_model, num_samples=num_prior_samples)
    prior_samples = prior(PRIOR_RNG, num_observations)
```

As before, we run mcmc against some synthetic data. We select any random sample from the prior as the ground truth:

```python
[20]: true_idx = 0
    true_loc = prior_samples["loc"][true_idx]
    true_scale = prior_samples["scale"][true_idx]
    true_high = prior_samples["high"][true_idx]
    true_x = prior_samples["x"][true_idx]
```

```python
[21]: plt.hist(true_x.copy())
    plt.axvline(true_high, linestyle=":\", color="k")
    plt.xlabel("x")
    plt.show()
```

Run MCMC and check the estimates:

```python
[22]: mcmc = MCMC(NUTS(truncated_normal_model), **MCMC_KWARGS)
    mcmc.run(MCMC_RNG, num_observations, true_x)
    mcmc.print_summary()
```

(continues on next page)
NumPyro Documentation

(continued from previous page)

4000/4000 [00:00<00:00, 7434.51it/s, 5 steps of size 1.56e-01. acc. prob=0.78]
sample: 100

4000/4000 [00:00<00:00, 7792.94it/s, 54 steps of size 5.41e-02. acc. prob=0.91]
sample: 100

4000/4000 [00:00<00:00, 7404.07it/s, 9 steps of size 1.77e-01. acc. prob=0.78]

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>0.88</td>
<td>0.01</td>
<td>0.88</td>
<td>0.88</td>
<td>0.89</td>
<td>590.13</td>
<td>1.01</td>
</tr>
<tr>
<td>loc</td>
<td>-0.58</td>
<td>0.07</td>
<td>-0.58</td>
<td>-0.70</td>
<td>-0.46</td>
<td>671.04</td>
<td>1.01</td>
</tr>
<tr>
<td>scale</td>
<td>1.40</td>
<td>0.05</td>
<td>1.40</td>
<td>1.32</td>
<td>1.48</td>
<td>678.30</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Number of divergences: 6310

Compare estimates against the ground truth:

```python
[23]: print(f"True high : {true_high:3.2f}")
print(f"True loc : {true_loc:3.2f}")
print(f"True scale: {true_scale:3.2f}")
```

True high : 0.88
True loc : -0.56
True scale: 1.45

Note that, even though we can recover good estimates for the true values, we had a very high number of divergences. These divergences happen because the data can be outside of the support that we are allowing with our priors. To fix this, we can change the prior on high so that it depends on the observations:

```python
[24]: def truncated_normal_model_2(num_observations, x=None):
    loc = numpyro.sample("loc", dist.Normal())
    scale = numpyro.sample("scale", dist.LogNormal())
    if x is None:
        high = numpyro.sample("high", dist.Normal())
    else:
        # high is greater or equal to the max value in x:
        delta = numpyro.sample("delta", dist.HalfNormal())
        high = numpyro.deterministic("high", delta + x.max())

    with numpyro.plate("observations", num_observations):
        numpyro.sample("x", RightTruncatedNormal(loc, scale, high), obs=x)
```

```python
[25]: mcmc = MCMC(NUTS(truncated_normal_model_2), **MCMC_KWARGS)
mcmc.run(MCMC_RNG, num_observations, true_x)
mcmc.print_summary(exclude_deterministic=False)
```

sample: 100

4000/4000 [00:03<00:00, 1089.76it/s, 15 steps of size 4.85e-01. acc. prob=0.93]
sample: 100

(continues on next page)
And the divergences are gone.

In practice, we usually want to understand how the data would look like without the truncation. To do that in NumPyro, there is no need of writing a separate model, we can simply rely on the `condition` handler to push the truncation point to infinity:

```python
model_without_truncation = numpyro.handlers.condition(
    truncated_normal_model,
    {"high": float("inf")},
)
estimates = mcmc.get_samples().copy()
estimates.pop("high")  # Drop to make sure these are not used
pred = Predictive(
    model_without_truncation,
    posterior_samples=estimates,
)
pred_samples = pred(PRED_RNG, num_observations=1000)
```

```python
# thin the samples for a faster histogram
samples_thinned = pred_samples["x"].ravel()[::1000]
```

```python
f, axes = plt.subplots(1, 2, figsize=(15, 5))

axes[0].hist(
    samples_thinned.copy(), label="Untruncated posterior", bins=20, density=True
)
axes[0].axvline(true_high, linestyle=":", color="k", label="Truncation point")
axes[0].set_title("Untruncated posterior")
axes[0].legend()

axes[1].hist(
    samples_thinned[samples_thinned < true_high].copy(),
    label="Tail of untruncated posterior",
)```
17.7.3 5.3 Example: Left-truncated Poisson

As a final example, we now implement a left-truncated Poisson distribution. Note that a right-truncated Poisson could be reformulated as a particular case of a categorical distribution, so we focus on the less trivial case.

Class attributes

For a truncated Poisson we need two parameters, the rate of the original Poisson distribution and a low parameter to indicate the truncation point. As this is a discrete distribution, we need to clarify whether or not the truncation point is included in the support. In this tutorial, we’ll take the convention that the truncation point low is part of the support.

The low parameter has to be given a ‘non-negative integer’ constraint. As it is a discrete parameter, it will not be possible to do inference for this parameter using NUTS. This is likely not a problem since the truncation point is often known in advance. However, if we really must infer the low parameter, it is possible to do so with DiscreteHMCGibbs though one is limited to using priors with enumerate support.

Like in the case of the truncated normal, the support of this distribution will be defined as a property and not as a class attribute because it depends on the specific value of the low parameter.

class LeftTruncatedPoisson:
    arg_constraints = {
        "low": constraints.nonnegative_integer,
        "rate": constraints.positive,
    }

    # ...
    @constraints.dependent_property(is_discrete=True)
    def support(self):
        return constraints.integer_greater_than(self.low - 1)
The `is_discrete` argument passed in the `dependent_property` decorator is used to tell the inference algorithms which variables are discrete latent variables.

**The `__init__` method**

Here we just follow the same pattern as in the previous example.

```python
# ...
def __init__(self, rate=1.0, low=0, validate_args=None):
    batch_shape = lax.broadcast_shapes(
        jnp.shape(low), jnp.shape(rate)
    )
    self.low, self.rate = promote_shapes(low, rate)
    super().__init__(batch_shape, validate_args=validate_args)
# ...
```

**The `log_prob` method**

The logic is very similar to the truncated normal case. But this time we are truncating on the left, so the correct normalization is the complementary cumulative density:

\[
M = \sum_{n=L}^{\infty} p_Y(n) = 1 - \sum_{n=0}^{L-1} p_Y(n) = 1 - F_Y(L-1)
\]  
(17.7)

For the code, we can rely on the `poisson` module that lives inside `jax.scipy.stats`.

```python
# ...
def log_prob(self, value):
    m = 1 - poisson.cdf(self.low - 1, self.rate)
    log_p = poisson.logpmf(value, self.rate)
    return jnp.where(value >= self.low, log_p - jnp.log(m), -jnp.inf)
# ...
```

**The `sample` method**

Inverse-transform sampling also works for discrete distributions. The “inverse” cdf of a discrete distribution being defined as:

\[
F^{-1}(u) = \max \{ n \in \mathbb{N} | F(n)u \}
\]  
(17.8)

Or, in plain English, \(F^{-1}(u)\) is the highest number for which the cumulative density is less than \(u\). However, there’s currently no implementation of \(F^{-1}\) for the Poisson distribution in Jax (at least, at the moment of writing this tutorial).

We have to rely on our own implementation. Fortunately, we can take advantage of the discrete nature of the distribution and easily implement a “brute-force” version that will work for most cases. The brute force approach consists of simply scanning all non-negative integers in order, one by one, until the value of the cumulative density exceeds the argument \(u\). The implicit requirement is that we need a way to evaluate the cumulative density for the truncated distribution, but we can calculate that:

\[
F_Z(z) = \sum_{n=0}^{z} p_Z(n) = \frac{1}{M} \sum_{n=L}^{\infty} p_Y(n) \quad \text{assuming } z \geq L = \frac{1}{M} \left( \sum_{n=0}^{z} p_Y(n) - \sum_{n=0}^{L-1} p_Y(n) \right) = \frac{1}{M} (F_Y(z) - F_Y(L-1))
\]  
(17.9)

And, of course, the value of \(F_Z(z)\) is equal to zero if \(z < L\). (As in the previous example, we are using \(Y\) to denote the original, un-truncated variable, and we are using \(Z\) to denote the truncated variable)
```python
# ...  
def sample(self, key, sample_shape=()):
    shape = sample_shape + self.batch_shape
    minval = jnp.finfo(jnp.result_type(float)).tiny
    u = random.uniform(key, shape, minval=minval)
    return self.icdf(u)

def icdf(self, u):
    def cond_fn(val):
        n, cdf = val
        return jnp.any(cdf < u)
    def body_fn(val):
        n, cdf = val
        n_new = jnp.where(cdf < u, n + 1, n)
        return n_new, self.cdf(n_new)
    low = self.low * jnp.ones_like(u)
    cdf = self.cdf(low)
    n, _ = lax.while_loop(cond_fn, body_fn, (low, cdf))
    return n.astype(jnp.result_type(int))

def cdf(self, value):
    m = 1 - poisson.cdf(self.low - 1, self.rate)
    f = poisson.cdf(value, self.rate) - poisson.cdf(self.low - 1, self.rate)
    return jnp.where(k >= self.low, f / m, 0)
```

A few comments with respect to the above implementation:  
* Even with double precision, if \( rate \) is much less than \( low \), the above code will not work. Due to numerical limitations, one obtains that \( \text{poisson.cdf}(low - 1, rate) \) is equal (or very close) to 1.0. This makes it impossible to re-weight the distribution accurately because the normalization constant would be 0.0.  
* The brute-force \( \text{icdf} \) is of course very slow, particularly when \( rate \) is high. If you need faster sampling, one option would be to rely on a faster search algorithm. For example:

```python
def icdf_faster(self, u):
    num_bins = 200  # Choose a reasonably large value
    bins = jnp.arange(num_bins)
    cdf = self.cdf(bins)
    indices = jnp.searchsorted(cdf, u)
    return bins[indices]
```

The obvious limitation here is that the number of bins has to be fixed a priori (jax does not allow for dynamically sized arrays). Another option would be to rely on an approximate implementation, as proposed in this article.

* Yet another alternative for the \( \text{icdf} \) is to rely on scipy’s implementation and make use of Jax’s host_callback module. This feature allows you to use Python functions without having to code them in Jax. This means that we can simply make use of scipy’s implementation of the Poisson ICDF! From the last equation, we can write the truncated \( \text{icdf} \) as:

\[
F_{Z}^{-1}(u) = F_{Y}^{-1}(Mu + F_{Y}(L - 1))
\]

And in python:

```python
def scipy_truncated_poisson_icdf(args):  # Note: all arguments are passed inside a tuple
    rate, low, u = args
    (continues on next page)```
rate = np.asarray(rate)
low = np.asarray(low)
u = np.asarray(u)
density = sp_poisson(rate)
low_cdf = density.cdf(low - 1)
normalizer = 1.0 - low_cdf
x = normalizer * u + low_cdf
return density.ppf(x)

In principle, it wouldn't be possible to use the above function in our NumPyro distribution because it is not coded in Jax. The jax.experimental.host_callback.call function solves precisely that problem. The code below shows you how to use it, but keep in mind that this is currently an experimental feature so you should expect changes to the module. See the host_callback docs for more details.

# ...
def icdf_scipy(self, u):
    result_shape = jax.ShapeDtypeStruct(
        u.shape,
        jnp.result_type(float) # int type not currently supported
    )
    result = jax.experimental.host_callback.call(
        scipy_truncated_poisson_icdf,
        (self.rate, self.low, u),
        result_shape=result_shape
    )
    return result.astype(jnp.result_type(int))
# ...

Putting it all together, the implementation is as below:

[29]: def scipy_truncated_poisson_icdf(args):  # Note: all arguments are passed inside a tuple
    rate, low, u = args
    rate = np.asarray(rate)
    low = np.asarray(low)
    u = np.asarray(u)
    density = sp_poisson(rate)
    low_cdf = density.cdf(low - 1)
    normalizer = 1.0 - low_cdf
    x = normalizer * u + low_cdf
    return density.ppf(x)

class LeftTruncatedPoisson(Distribution):
    """
    A truncated Poisson distribution.
    :param numpy.ndarray low: lower bound at which truncation happens
    :param numpy.ndarray rate: rate of the Poisson distribution.
    """

    arg_constraints = {
        "low": constraints.nonnegative_integer,
        "rate": constraints.positive,
```python
def __init__(self, rate=1.0, low=0, validate_args=None):
    batch_shape = lax.broadcast_shapes(jnp.shape(low), jnp.shape(rate))
    self.low, self.rate = promote_shapes(low, rate)
    super().__init__(batch_shape, validate_args=validate_args)

def log_prob(self, value):
    m = 1 - poisson.cdf(self.low - 1, self.rate)
    log_p = poisson.logpmf(value, self.rate)
    return jnp.where(value >= self.low, log_p - jnp.log(m), -jnp.inf)

def sample(self, key, sample_shape=()):
    shape = sample_shape + self.batch_shape
    float_type = jnp.result_type(float)
    minval = jnp.finfo(float_type).tiny
    u = random.uniform(key, shape, minval=minval)
    # return self.icdf(u)  # Brute force
    # return self.icdf_faster(u)  # For faster sampling.
    return self.icdf_scipy(u)  # Using 'host_callback'

def icdf(self, u):
    def cond_fn(val):
        n, cdf = val
        return jnp.any(cdf < u)
    def body_fn(val):
        n, cdf = val
        n_new = jnp.where(cdf < u, n + 1, n)
        return n_new, self.cdf(n_new)
    low = self.low * jnp.ones_like(u)
    cdf = self.cdf(low)
    n, _ = lax.while_loop(cond_fn, body_fn, (low, cdf))
    return n.astype(jnp.result_type(int))

def icdf_faster(self, u):
    num_bins = 200  # Choose a reasonably large value
    bins = jnp.arange(num_bins)
    cdf = self.cdf(bins)
    indices = jnp.searchsorted(cdf, u)
    return bins[indices]

def icdf_scipy(self, u):
    result_shape = jax.ShapeDtypeStruct(u.shape, jnp.result_type(float))
    result = jax.experimental.host_callback.call(
        scipy_truncated_poisson_icdf,
        (self.rate, self.low, u),
        result_shape=result_shape,
    )
    return result.astype(jnp.result_type(int))
```

(continues on next page)
def cdf(self, value):
m = 1 - poisson.cdf(self.low - 1, self.rate)
f = poisson.cdf(value, self.rate) - poisson.cdf(self.low - 1, self.rate)
return jnp.where(value >= self.low, f / m, 0)

@constraints.dependent_property(is_discrete=True)
def support(self):
    return constraints.integer_greater_than(self.low - 1)

Let’s try it out!

[30]: def discrete_distplot(samples, ax=None, **kwargs):
    
    Utility function for plotting the samples as a barplot.
    
x, y = np.unique(samples, return_counts=True)
y = y / sum(y)
if ax is None:
    ax = plt.gca()

    ax.bar(x, y, **kwargs)

    return ax

[31]: def truncated_poisson_model(num_observations, x=None, k=5):
    
    zeros = jnp.zeros((k,))
    low = numpyro.sample("low", dist.Categorical(logits=zeros))
    rate = numpyro.sample("rate", dist.LogNormal(1, 1))

    with numpyro.plate("observations", num_observations):
        numpyro.sample("x", LeftTruncatedPoisson(rate, low), obs=x)

Prior samples

[32]:

    # -- prior samples
    num_observations = 1000
    num_prior_samples = 100
    prior = Predictive(truncated_poisson_model, num_samples=num_prior_samples)
    prior_samples = prior(PRIOR_RNG, num_observations)

Inference

As in the case for the truncated normal, here it is better to replace the prior on the low parameter so that it is consistent with the observed data. We’d like to have a categorical prior on low (so that we can use DiscreteHMCGibbs) whose highest category is equal to the minimum value of x (so that prior and data are consistent). However, we have to be careful in the way we write such model because Jax does not allow for dynamically sized arrays. A simple way of coding this model is to simply specify the number of categories as an argument:

[33]: def truncated_poisson_model(num_observations, x=None, k=5):
    
    zeros = jnp.zeros((k,))
    low = numpyro.sample("low", dist.Categorical(logits=zeros))
    rate = numpyro.sample("rate", dist.LogNormal(1, 1))

    with numpyro.plate("observations", num_observations):
        numpyro.sample("x", LeftTruncatedPoisson(rate, low), obs=x)
# Take any prior sample as the true process.
true_idx = 6
ture_low = prior_samples["low"] [true_idx]
ture_rate = prior_samples["rate"] [true_idx]
ture_x = prior_samples["x"] [true_idx]
discrete_distplot(ture_x.copy());

To do inference, we set \( k = \min(x) + 1 \). Note also the use of `DiscreteHMCGibbs`:

```python
mcmc = MCMC(DiscreteHMCGibbs(NUTS(truncated_poisson_model)), **MCMC_KWARGS)
mcmc.run(MCMC_RNG, num_observations, ture_x, k=true_x.min() + 1)
mcmc.print_summary()
```

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>4.13</td>
<td>2.43</td>
<td>4.00</td>
<td>0.00</td>
<td>7.00</td>
<td>7433.79</td>
<td>1.00</td>
</tr>
<tr>
<td>rate</td>
<td>18.16</td>
<td>0.14</td>
<td>18.16</td>
<td>17.96</td>
<td>18.40</td>
<td>3074.46</td>
<td>1.00</td>
</tr>
</tbody>
</table>

As before, one needs to be extra careful when estimating the truncation point. If the truncation point is known is best to provide it.
And note we can use NUTS directly because there's no need to infer any discrete parameters.

```
[38]: mcmc = MCMC(
    NUTS(model_with_known_low),
    **MCMC_KWARGS,
)
```

```
[39]: mcmc.run(MCMC_RNG, num_observations, true_x)
mcmc.print_summary()
```

```
mean    std  median   5.0%    95.0%   n_eff   r_hat
rate  18.17  0.13   18.17   17.95   18.39  3406.81   1.00
```

Removing the truncation

```
[40]: model_without_truncation = numpyro.handlers.condition(
    truncated_poisson_model,
    {"low": 0},
)
pred = Predictive(model_without_truncation, posterior_samples=mcmc.get_samples())
pred_samples = pred(PRED_RNG, num_observations)
thinned_samples = pred_samples["x"][::500]
```

```
[41]: discrete_distplot(thinned_samples.copy());
```
17.8 References and related material

1. Wikipedia page on inverse transform sampling
2. David Mackay’s book on information theory
3. Composite models with underlying folded distributions
4. Application of the generalized folded-normal distribution to the process capability measures
5. Pyro SVI tutorial part 3
6. Approximation of the inverse Poisson cumulative distribution function
CHAPTER EIGHTEEN

BAYESIAN CENSORING DATA MODELING

In this example we show how to model censored data using NumPyro. Censoring is a condition in which the value of a measurement or observation is only partially known. We work out two examples: a continuous and a discrete likelihood. Both cases follow the same structure but differ in a small detail, which we want to highlight. The key ingredient of the modeling strategy is the cumulative distribution function (CDF) parametrization described in Uber’s blog post “Modeling Censored Time-to-Event Data Using Pyro, an Open Source Probabilistic Programming Language” (see here the original gist code).

This example is based in the notebook Bayesian Censoring Data Modeling.

18.1 Prepare Notebook

```python
[ ]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro arviz matplotlib preliz

[ ]:
import os
import arviz as az
from IPython.display import set_matplotlib_formats
from jaxlib.xla_extension import ArrayImpl
import matplotlib.pyplot as plt
import preliz as pz
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
from numpyro.handlers import mask
from numpyro.infer import MCMC, NUTS, Predictive

plt.style.use("bmh")
if "NUMPYRO_SPHINXBUILD" in os.environ:
    set_matplotlib_formats("svg")

plt.rcParams["figure.figsize"] = [8, 6]
numpyro.set_host_device_count(n=4)
```

(continues on next page)
18.2 Part I: Continuous Distribution

We start by considering the continuous case. We assume that we have a dataset of observations $y$ that are censored from below and above at a known thresholds. We want to estimate the parameters of a distribution that generated these observations. We assume that the distribution is a gamma distribution (with parameters $\alpha$ and $\beta$), but the same approach can be used for other distributions.

18.2.1 Generate Censored Data

We generate data from a gamma distribution and we clip the data to be between the thresholds lower and upper. We then plot the histogram of the data and the true gamma distribution.

```python
rng_key = random.PRNGKey(seed=0)
assert numpyro.__version__.startswith("0.15.2")
%
load_ext autoreload
%
autoreload 2
%
config InlineBackend.figure_format = "retina"

class CensoredGammaDataParams(BaseModel):
    alpha: float = Field(..., description="Concentration parameter", gt=0)
    beta: float = Field(..., description="Rate parameter", gt=0)
    lower: float = Field(..., description="Lower censoring bound", gt=0)
    upper: float = Field(..., description="Upper censoring bound", gt=0)
    n: int = Field(..., description="Number of samples", gt=0)

def generate_censored_gamma_samples(
    rng_key: ArrayImpl, params: CensoredGammaDataParams
) -> ArrayImpl:
    raw_samples = dist.Gamma(concentration=params.alpha, rate=params.beta).sample(
        rng_key, (params.n,)
    )
    return jnp.clip(raw_samples, params.lower, params.upper)

censored_gamma_data_params = CensoredGammaDataParams(
    alpha=3.0, beta=1.0, lower=0.5, upper=5.0, n=100
)
rng_key, rng_subkey = random.split(rng_key)
censored_gamma_samples = generate_censored_gamma_samples(
    rng_key=rng_subkey, params=censored_gamma_data_params
)

fig, ax = plt.subplots()
_ = ax.hist(
    censored_gamma_samples,
    bins=20,
)
18.2.2 Naive Model

Before we implement the censoring model, we start with a naive model that does not take into account the censoring component. We simply use a gamma distribution with parameters $\alpha$ and $\beta$ to model the data. We then plot the posterior distribution of the parameters and compare them with the true values.

```python
[3]: def gamma_model(y: ArrayImpl) -> None:
    alpha = numpyro.sample("alpha", dist.Exponential(1.0))
    beta = numpyro.sample("beta", dist.Exponential(1.0))
    numpyro.sample("obs", dist.Gamma(concentration=alpha, rate=beta), obs=y)
```

(continues on next page)
class InferenceParams(BaseModel):
    num_warmup: int = Field(2_000, ge=1)
    num_samples: int = Field(3_000, ge=1)
    num_chains: int = Field(4, ge=1)

inference_params = InferenceParams()

gamma_kernel = NUTS(gamma_model)
gamma_mcmc = MCMC(
    gamma_kernel,
    num_warmup=inference_params.num_warmup,
    num_samples=inference_params.num_samples,
    num_chains=inference_params.num_chains,
)

rng_key, rng_subkey = random.split(rng_key)
gamma_mcmc.run(rng_key, y=censored_gamma_samples)

We also generate posterior predictive samples.

[4]: gamma_predictive = Predictive(
    model=gamma_model, posterior_samples=gamma_mcmc.get_samples()
)
rng_key, rng_subkey = random.split(rng_key)
gamma_posterior_predictive = gamma_predictive(rng_subkey, y=None)

gamma_idata = az.from_numpyro(posterior=gamma_mcmc)
gamma_idata.extend(az.from_numpyro(posterior_predictive=gamma_posterior_predictive))

Now we plot the trace and compare the true and estimated parameters.

[5]: axes = az.plot_trace(
    data=gamma_idata,
    compact=True,
    lines=[
        ("alpha", {}), censored_gamma_data_params.alpha),
        ("beta", {}), censored_gamma_data_params.beta),
    )

(continues on next page)
backend_kwargs={"figsize": (10, 6), "layout": "constrained"},
)
plt.gcf().suptitle("Gamma Model", fontsize=18, fontweight="bold");

[6]: axes = az.plot_posterior(
    data=gamma_idata,
    ref_val=[censored_gamma_data_params.alpha, censored_gamma_data_params.beta],
    round_to=3,
    figsize=(12, 5),
)
plt.gcf().suptitle("Gamma Model Parameters", fontsize=18, fontweight="bold", y=1.03);
Despite the fact that the true parameters are in the 94% HDI, we do see a bias in the estimation of the parameters. This is expected since we are not taking into account the censoring in the model.

We can visualize the inferred distribution and compare it with the true distribution from the data generating process.

```python
fig, ax = plt.subplots()
_ = ax.hist(
    censored_gamma_samples,
    bins=20,
    density=True,
    color="C1",
    alpha=0.5,
    label="Censored data",
)
ax.axvline(censored_gamma_data_params.lower, color="k", linestyle="--")
ax.axvline(
    censored_gamma_data_params.upper,
    color="k",
    linestyle="--",
    label="Censoring bounds",
)
pz.Gamma(
    alpha=censored_gamma_data_params.alpha, beta=censored_gamma_data_params.beta
).plot_pdf(color="C0", ax=ax)
az.plot_kde(
    gamma_idata["posterior_predictive"]['obs'].to_numpy().flatten(),
    plot_kwargs={"color": "C2", "label": "Posterior predictive"},
    ax=ax,
)
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))
ax.set_title("Gamma Model", fontsize=16, fontweight="bold");
```
Here see that the distributions do not match well.

### 18.2.3 Censored Gamma Model

As mentioned in the introduction, the main idea to implement the censoring model is to use the CDF parametrization. The reason is because the probability density function PDF of a censored distribution is

\[
\begin{align*}
0 & \quad \text{for } y < \text{lower}, \\
\text{CDF}(\text{lower}, \text{dist}) & \quad \text{for } y = \text{lower}, \\
\text{PDF}(y, \text{dist}) & \quad \text{for } \text{lower} < y < \text{upper}, \\
1 - \text{CDF}(\text{upper}, \text{dist}) & \quad \text{for } y = \text{upper}, \\
0 & \quad \text{for } y > \text{upper},
\end{align*}
\]

Hence, we just need too implement these conditions as a custom likelihood function. To do this in NumPyro we follow the approach described in the gist of Uber’s blog post. To follow this strategy we need an indicator function to encode the censoring:

```python
[8]:
def get_truncation_label(y: ArrayImpl, lower: float, upper: float) -> ArrayImpl:
    return jnp.where(y == lower, -1, jnp.where(y == upper, 1, 0))
```

```python
truncation_label = get_truncation_label(
    y=censored_gamma_samples,
    lower=censored_gamma_data_params.lower,
    upper=censored_gamma_data_params.upper,
)
```
We are now ready to implement the model. For points in the interval $lower < y < upper$ we simply sample form a gamma distribution. On the other hand, points on the boundary follow Bernoulli distribution (if we model the censoring mask value) where the truncation probability can be parametrized by the CDF. Let’s see how to do this concretely:

```python
[9]: def censored_gamma_model(
    y: ArrayImpl, lower: float, upper: float, truncation_label: ArrayImpl
) -> None:
    alpha = numpyro.sample("alpha", dist.Exponential(1.0))
    beta = numpyro.sample("beta", dist.Exponential(1.0))

    distribution = dist.Gamma(concentration=alpha, rate=beta)

    with mask(mask=truncation_label == -1):
        truncation_prob_lower = distribution.cdf(lower)
        numpyro.sample(
            "truncated_label_lower", dist.Bernoulli(truncation_prob_lower), obs=1
        )

    with mask(mask=truncation_label == 0):
        numpyro.sample("obs", distribution, obs=y)

    with mask(mask=truncation_label == 1):
        truncation_prob_upper = 1 - distribution.cdf(upper)
        numpyro.sample(
            "truncated_label_upper", dist.Bernoulli(truncation_prob_upper), obs=1
        )

[10]: censored_gamma_kernel = NUTS(censored_gamma_model)
censored_gamma_mcmc = MCMC(
    censored_gamma_kernel,
    num_warmup=inference_params.num_warmup,
    num_samples=inference_params.num_samples,
    num_chains=inference_params.num_chains,
)
	ring_key, rng_subkey = random.split(rng_key)
censored_gamma_mcmc.run(
    rng_key,
    y=censored_gamma_samples,
    lower=censored_gamma_data_params.lower,
    upper=censored_gamma_data_params.upper,
    truncation_label=truncation_label,
)
```

404 Chapter 18. Bayesian Censoring Data Modeling
Remark: Note that is straightforward to add lower and upper censoring components to the model (Uber’s blog post has just one censoring component).

We now fit the model and evaluate the results:

```python
[c11]: censored_gamma_predictive = Predictive(  
    model=censored_gamma_model,  
    posterior_samples=censored_gamma_mcmc.get_samples(),  
    return_sites=['obs'],
  )
  
rng_key, rng_subkey = random.split(rng_key)
censored_gamma_posterior_predictive = censored_gamma_predictive(  
    rng_subkey,
    y=None,  
    lower=censored_gamma_data_params.lower,  
    upper=censored_gamma_data_params.upper,  
    truncation_label=truncation_label,
  )

censored_gamma_idata = az.from_numpyro(posterior=censored_gamma_mcmc)
censored_gamma_idata.extend(  
    az.from_numpyro(posterior_predictive=censored_gamma_posterior_predictive)
)
```

```python
[c12]: axes = az.plot_trace(  
    data=censored_gamma_idata,  
    compact=True,  
    lines=[  
      ('alpha', {}, censored_gamma_data_params.alpha),  
      ('beta', {}, censored_gamma_data_params.beta),  
    ],  
    backend_kwargs={'figsize': (10, 6), 'layout': 'constrained'},
  )
  
  plt.gcf().suptitle("Censored Gamma Model", fontsize=18, fontweight="bold");
```
```python
[13]: axes = az.plot_posterior(
    data=censored_gamma_idata,
    ref_val=[censored_gamma_data_params.alpha, censored_gamma_data_params.beta],
    round_to=3,
    figsize=(12, 5),
)
plt.gcf().suptitle(
    "Censored Gamma Model Parameters", fontsize=18, fontweight="bold", y=1.03
);
```
We clearly see that the true parameters are in correspondence to the mean and mode of the posterior distribution. The censoring model is able to recover the true parameters.

As we did for the naive model, we visualize the inferred distribution and compare it with the true distribution from the data generating process.

```python
[14]: fig, ax = plt.subplots()
    _ = ax.hist(
        censored_gamma_samples,
        bins=20,
        density=True,
        color="C1",
        alpha=0.5,
        label="Censored data",
    )
    ax.axvline(censored_gamma_data_params.lower, color="k", linestyle="--")
    ax.axvline(
        censored_gamma_data_params.upper,
        color="k",
        linestyle="--",
        label="Censoring bounds",
    )
    pz.Gamma(
        alpha=censored_gamma_data_params.alpha, beta=censored_gamma_data_params.beta
    ).plot_pdf(color="C0", ax=ax)
    az.plot_kde(
        censored_gamma_idata["posterior_predictive"][:"obs"].to_numpy().flatten(),
        plot_kwargs={"color": "C2", "label": "Posterior predictive"},
        ax=ax,
    )
    ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))
    ax.set_title("Censored Gamma Model (no clipping)", fontsize=16, fontweight="bold");
```
This looks much better than the naive model. However, there something wring as we are just sampling from the obs variable! We should not have posterior samples outside the censoring bounds. We can easily fix this by clipping the samples to the bounds.

```python
[15]: fig, ax = plt.subplots()
    _ = ax.hist(
        censored_gamma_samples,
        bins=20,
        density=True,
        color="C1",
        alpha=0.5,
        label="Censored data",
    )
    ax.axvline(censored_gamma_data_params.lower, color="k", linestyle="--")
    ax.axvline(
        censored_gamma_data_params.upper,
        color="k",
        linestyle="--",
        label="Censoring bounds",
    )
    pz.Gamma(
        alpha=censored_gamma_data_params.alpha, beta=censored_gamma_data_params.beta
    ).plot_pdf(color="C0", ax=ax)
    az.plot_kde(
        censored_gamma_idata["posterior_predictive"]["obs"]
        .clip(min=censored_gamma_data_params.lower, max=censored_gamma_data_params.upper)
        .to_numpy()
        .flatten(),
```
In this second part we do the same as in the first part but for a discrete distribution. Again, we assume that we have a dataset of observations $y$ that are censored from below and above at known thresholds. This time we assume the data follows a Poisson distribution (with parameter $\lambda$). We want to recover the true parameter $\lambda$. The censoring model has the same structure as in the continuous case, but there is a small detail in the upper bound. We need to add term to the truncation probability to account for the probability of the upper bound. Specifically we need to add the probability of the upper bound using the PDF.

### 18.3.1 Generate Censored Data

We generate data from a Poisson distribution and we *clip* the data to be between the thresholds `lower` and `upper`. We then plot the histogram of the data and the true Poisson distribution.

```python
[16]: class CensoredPoissonDataParams(BaseModel):
    rate: float = Field(..., description="Rate parameter", gt=0)
    lower: float = Field(..., description="Lower censoring bound", gt=0)
    upper: float = Field(..., description="Upper censoring bound", gt=0)
    n: int = Field(..., description="Number of samples", gt=0)
```
```python
def generate_censored_poisson_samples(
    rng_key: ArrayImpl, params: CensoredPoissonDataParams
) -> ArrayImpl:
    raw_samples = dist.Poisson(rate=params.rate).sample(rng_key, (params.n,))
    return jnp.clip(raw_samples, params.lower, params.upper)

censored_poisson_data_params = CensoredPoissonDataParams(
    rate=1, lower=1, upper=4, n=100
)
 rng_key, rng_subkey = random.split(rng_key)
censored_poisson_samples = generate_censored_poisson_samples(
    rng_key=rng_subkey, params=censored_poisson_data_params
)

fig, ax = plt.subplots()
_ = ax.hist(
    censored_poisson_samples,
    bins=20,
    density=True,
    color="C1",
    alpha=0.5,
    label="Censored data",
    align="left",
)
ax.axvline(censored_poisson_data_params.lower, color="k", linestyle="--")
ax.axvline(
    censored_poisson_data_params.upper,
    color="k",
    linestyle="--",
    label="Censoring bounds",
)
pz.Poisson(mu=censored_poisson_data_params.rate).plot_pdf(color="C0", ax=ax)
ax.set_title("Censored Poisson Sample Data", fontsize=16, fontweight="bold");
```
### 18.3.2 Naive Model

Just as in the continuous case, we start with a naive model that does not take into account the censoring component. We simply use a Poisson distribution with parameter $\lambda$ to model the data.

```python
[17]:
def poisson_model(y: ArrayImpl) -> None:
    rate = numpyro.sample("rate", dist.Exponential(1.0))
    numpyro.sample("obs", dist.Poisson(rate), obs=y)
```

```python
poisson_kernel = NUTS(poisson_model)
poisson_mcmc = MCMC(
    poisson_kernel,
    num_warmup=inference_params.num_warmup,
    num_samples=inference_params.num_samples,
    num_chains=inference_params.num_chains,
)
```

```python
rng_key, rng_subkey = random.split(rng_key)
poisson_mcmc.run(rng_key, y=censored_poisson_samples)
```

(continues on next page)
Running chain 0: 100% | 5000/5000 [00:02<00:00, 2123.00it/s]
Running chain 1: 100% | 5000/5000 [00:02<00:00, 2126.03it/s]
Running chain 2: 100% | 5000/5000 [00:02<00:00, 2129.97it/s]
Running chain 3: 100% | 5000/5000 [00:02<00:00, 2134.66it/s]

[18]: poisson_predictive = Predictive(
    model=poisson_model, posterior_samples=poisson_mcmc.get_samples()
)
    rng_key, rng_subkey = random.split(rng_key)
poisson_posterior_predictive = poisson_predictive(rng_subkey, y=None)

poisson_idata = az.from_numpyro(posterior=poisson_mcmc)
poisson_idata.extend(az.from_numpyro(posterior_predictive=poisson_posterior_predictive))

[19]: axes = az.plot_trace(
    data=poisson_idata,
    compact=True,
    lines=[("rate", {}, censored_poisson_data_params.rate)],
    backend_kwargs="figsize": (10, 6), "layout": "constrained"),
)
plt.gcf().suptitle("Poisson Model", fontsize=18, fontweight="bold");

[20]: axes = az.plot_posterior(
    data=poisson_idata,
We clearly see a bias in the estimation of the parameter. The true value of the rate parameter $\lambda$ is not even in the 94% HDI.

We can also take a look into the posterior predictive distribution.

```python
fig, ax = plt.subplots()
_ = ax.hist(censored_poisson_samples, bins=20, density=True, align="left", color="C1", alpha=0.5, label="Censored data",
)
ax.axvline(censored_poisson_data_params.lower, color="k", linestyle="--")
ax.axvline(censored_poisson_data_params.upper,
```
color="k",
linestyle="--",
label="Censoring bounds",
)
pz.Poisson(mu=censored_poisson_data_params.rate).plot_pdf(color="C0", ax=ax)
_ = ax.hist(
    poisson_idata["posterior_predictive"]['obs'].to_numpy().flatten(),
    bins=50,
    density=True,
    align="right",
    color="C2",
    alpha=0.5,
    label="Censored data",
)
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))
ax.set_title("Poisson Model", fontsize=16, fontweight="bold");
18.3.3 Censored Poisson Model

Now we work out the censored model. As before, we need to get the censoring labels to manage the likelihood components:

```python
[22]: truncation_label = get_truncation_label(
    y=censored_poisson_samples,
    lower=censored_poisson_data_params.lower,
    upper=censored_poisson_data_params.upper,
)
```

Now, finally regarding the detail about the upper censoring component.

- We want to consider the closed interval \( P(lower \leq y \leq upper) \).
- For the upper interval we use \( 1 - CDF(upper) = 1 - P(y \leq upper) = P(y > upper) \), but we actually need \( P(y \geq upper) \).
- Hence we need to add \( P(y = upper) \) in the upper truncation probability.
- Observe that this discussion is irrelevant for the continuous case as the probability of a single point is zero so \( P(y > upper) = P(y \geq upper) \).

The implementation of the model should be clearer now.

```python
[23]: def censored_poisson_model(
    y: ArrayImpl, lower: float, upper: float, truncation_label: ArrayImpl
) -> None:
    rate = numpyro.sample("rate", dist.Exponential(1.0))
    distribution = dist.Poisson(rate)

    with mask(mask=truncation_label == -1):
        truncation_prob_lower = distribution.cdf(lower)
        numpyro.sample("truncated_label_lower", dist.Bernoulli(truncation_prob_lower), obs=1

    with mask(mask=truncation_label == 0):
        numpyro.sample("obs", distribution, obs=y)

    with mask(mask=truncation_label == 1):
        ccdf_upper = 1 - distribution.cdf(upper)
        pmf_upper = jnp.exp(distribution.log_prob(upper))
        truncation_prob_upper = ccdf_upper + pmf_upper
        numpyro.sample("truncated_label_upper", dist.Bernoulli(truncation_prob_upper), obs=1
```

Remark: This was an important detail pointed out by Kyle Caron in his blog post “Modeling Anything With First Principles: Demand under extreme stockouts”.

We proceed to fit the model and compare the results with the true parameter.

```python
[24]: censored_poisson_kernel = NUTS(censored_poisson_model)
censored_poisson_mcmc = MCMC(
    censored_poisson_kernel,
```
num_warmup=inference_params.num_warmup,
num_samples=inference_params.num_samples,
num_chains=inference_params.num_chains,
)

censored_poisson_mcmc.run(
    rng_key,
y=censored_poisson_samples,
    lower=censored_poisson_data_params.lower,
    upper=censored_poisson_data_params.upper,
    truncation_label=truncation_label,
)

Running chain 0: 100% | 5000/5000 [00:03<00:00, 1257.60it/s]
Running chain 1: 100% | 5000/5000 [00:03<00:00, 1259.63it/s]
Running chain 2: 100% | 5000/5000 [00:03<00:00, 1260.75it/s]
Running chain 3: 100% | 5000/5000 [00:03<00:00, 1261.95it/s]

censored_poisson_predictive = Predictive(
    model=censored_poisson_model,
    posterior_samples=censored_poisson_mcmc.get_samples(),
    return_sites="obs",
)

censored_poisson_posterior_predictive = censored_poisson_predictive(
    rng_subkey,
y=None,
    lower=censored_poisson_data_params.lower,
    upper=censored_poisson_data_params.upper,
    truncation_label=truncation_label,
)

censored_poisson_idata = az.from_numpyro(posterior=censored_poisson_mcmc)

censored_poisson_idata.extend(
    az.from_numpyro(posterior_predictive=censored_poisson_posterior_predictive)
)

axes = az.plot_trace(
    data=censored_poisson_idata,
    compact=True,
    lines=[("rate", {}), censored_poisson_data_params.rate],
    backend_kwargs={"figsize": (10, 6), "layout": "constrained"},
)
plt.gcf().suptitle("Censored Poisson Model", fontsize=18, fontweight="bold");

Censored Poisson Model

[27]: axes = az.plot_posterior(
    data=censored_poisson_idata,
    ref_val=[censored_poisson_data_params.rate],
    round_to=3,
    figsize=(10, 6),
)
plt.gcf().suptitle("Censored Poisson Model Parameter", fontsize=18, fontweight="bold");
We see that we have recovered the true value of the rate parameter $\lambda = 1$.

Here is the posterior predictive distribution:

```python
[28]: fig, ax = plt.subplots()
  _ = ax.hist(
    censored_poisson_samples,
    bins=20,
    density=True,
    align="left",
    color="C1",
    alpha=0.5,
    label="Censored data",
  )
ax.axvline(censored_poisson_data_params.lower, color="k", linestyle="--")
ax.axvline(
    censored_poisson_data_params.upper,
    color="k",
    linestyle="--",
    label="Censoring bounds",
)
pz.Poisson(mu=censored_poisson_data_params.rate).plot_pdf(color="C0", ax=ax)
  _ = ax.hist(
    censored_poisson_idata["posterior_predictive"]['obs'].to_numpy().flatten(),
  )
```

(continues on next page)
As before, to have a correct posterior predictive distribution we need to clip the samples to the bounds.

```
[29]: fig, ax = plt.subplots()
     _ = ax.hist(
              censored_poisson_samples,
              bins=20,
              density=True,
              align="left",
              color="C1",
              alpha=0.5,
              label="Censored data",
          )
     ax.axvline(censored_poisson_data_params.lower, color="k", linestyle="--")
     ax.axvline(
              censored_poisson_data_params.upper,
              color="k",
              linestyle="--",
          )
```
Here we see that the observed and predicted distributions match well.
HILBERT SPACE APPROXIMATION GAUSSIAN PROCESS MODULE

In this notebook we provide an example on how to use the Hilbert Space Gaussian Process module. We use a synthetic data set to illustrate the usage of some of the kernel approximation functions provided in the module.

**Remark:** This example was taken from the original blog post A Conceptual and Practical Introduction to Hilbert Space GPs Approximation Methods.

### 19.1 Prepare Notebook

```python
#1: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro arviz

#2: import os
import arviz as az
from IPython.display import set_matplotlib_formats
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator
from jax import random
import jax.numpy as jnp
import numpyro
from numpyro.contrib.hsgp.approximation import hsgp_squared_exponential
from numpyro.contrib.hsgp.laplacian import eigenfunctions
from numpyro.contrib.hsgp.spectral_densities import (diag_spectral_density_squared_exponential,
)
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive
plt.style.use("bmh")
if "NUMPYRO_SPHINXBUILD" in os.environ:
    set_matplotlib_formats("svg")
plt.rcParams["figure.figsize"] = [10, 6]
numpyro.set_host_device_count(n=4)
rng_key = random.PRNGKey(seed=42)
```

(continues on next page)
19.2 Generate Synthetic Data

We generate synthetic data in a one-dimensional space. We split the data into training and test sets.

```python
[3]: def generate_synthetic_data(rng_key, start, stop: float, num, scale):
    x = jnp.linspace(start=start, stop=stop, num=num)
    y = jnp.sin(4 * jnp.pi * x) + jnp.sin(7 * jnp.pi * x)
    y_obs = y + scale * random.normal(rng_key, shape=(num,))
    return x, y, y_obs

n_train = 80
n_test = 100
scale = 0.3
rng_key, rng_subkey = random.split(rng_key)
x_train, y_train, y_train_obs = generate_synthetic_data(
                                    rng_key=rng_subkey, start=0, stop=1, num=n_train, scale=scale)

rng_key, rng_subkey = random.split(rng_key)
x_test, y_test, y_test_obs = generate_synthetic_data(
                                    rng_key=rng_subkey, start=-0.2, stop=1.2, num=n_test, scale=scale)
```

```python
[4]: fig, ax = plt.subplots()
ax.scatter(x_train, y_train_obs, c="C0", label="observed (train)")
ax.scatter(x_test, y_test_obs, c="C1", label="observed (test)")
ax.plot(x_train, y_train, color="black", linewidth=3, label="mean (latent)")
ax.axvline(x=0, color="C2", alpha=0.8, linestyle="--", linewidth=2)
ax.axvline(x=1, color="C2", linestyle="--", alpha=0.8, linewidth=2)
ax.legend(loc="upper center", bbox_to_anchor=(0.5, -0.1), ncol=4)
ax.set(xlabel="x", ylabel="y")
ax.set_title("Synthetic Data", fontsize=16, fontweight="bold");
```
It is recommended to center the data before using the approximation functions.

```
[5]: train_mean = x_train.mean()
x_train_centered = x_train - train_mean
x_test_centered = x_test - train_mean
```

19.3 Specify the Model

We now specify the model. We will use the squared exponential kernel to model the mean of a Gaussian likelihood. This kernel function depends on two parameters:

- The amplitude \( \alpha \).
- The length scale \( \ell \).

For these two parameters, we need to specify prior distributions.

Next, we use the function `hsgp_squared_exponential` to approximate the kernel function with the basis functions. We need to specify if we want the centered or non-centered parameterization of the linear model approximation.

```
[6]: def model(x, ell, m, non_centered, y=None):
    # --- Priors ---
    alpha = numpyro.sample("alpha", dist.InverseGamma(concentration=12, rate=10))
    length = numpyro.sample("length", dist.InverseGamma(concentration=6, rate=1))
    noise = numpyro.sample("noise", dist.InverseGamma(concentration=12, rate=10))
```

(continues on next page)
# --- Parametrization ---
f = hsgp_squared_exponential(
    x=x, alpha=alpha, length=length, ell=ell, m=m, non_centered=non_centered
)

# --- Likelihood ---
with numpyro.plate("data", x.shape[0]):
    numpyro.sample("likelihood", dist.Normal(loc=f, scale=noise), obs=y)

19.4 Fit the Model

For this example we will use ell=0.8 (since we centered the data), m=20 and the non-centered parameterization.

Now we fit the model to the data using the NUTS sampler.

```
sampler = NUTS(model)
mcmc = MCMC(sampler=sampler, num_warmup=1_000, num_samples=2_000, num_chains=4)
rng_key, rng_subkey = random.split(rng_key)
ell = 0.8
m = 20
non_centered = True
mcmc.run(rng_subkey, x_train_centered, ell, m, non_centered, y_train_obs)
```

Let’s see the model diagnostics and posterior distribution of the parameters.

```
idata = az.from_numpyro(posterior=mcmc)

az.summary(
    data=idata,
    var_names=['alpha', 'length', 'noise', 'beta'],
)
```

(continues on next page)
beta[7]  1.172  0.536  0.128  2.136  0.008  0.006  3991.0
beta[8] -0.688  0.531 -1.702  0.307  0.008  0.006  3923.0
beta[9]  0.685  0.537 -0.433  1.572  0.008  0.006  4623.0
beta[10]  2.957  0.680  1.756  4.278  0.009  0.007  5641.0
beta[11] -1.360  0.648 -2.620 -0.189  0.008  0.006  4858.0
beta[12] -0.750  0.657 -1.991  0.466  0.008  0.006  6727.0
beta[13]  0.474  0.747 -0.888  1.926  0.009  0.007  7606.0
beta[14]  0.054  0.790 -1.472  1.474  0.009  0.009  7786.0
beta[15] -0.228  0.796 -1.775  1.241  0.009  0.008  8051.0
beta[16]  0.239  0.845 -1.324  1.840  0.009  0.009  9099.0

    ess_tail  r_hat
alpha     4996.0  1.0
length    5364.0  1.0
noise     6009.0  1.0
beta[0]   5326.0  1.0
beta[1]   4980.0  1.0
beta[2]   4845.0  1.0
beta[3]   4762.0  1.0
beta[4]   4314.0  1.0
beta[5]   4779.0  1.0
beta[6]   3904.0  1.0
beta[7]   4807.0  1.0
beta[8]   5092.0  1.0
beta[9]   4779.0  1.0
beta[10]  5873.0  1.0
beta[11]  5675.0  1.0
beta[12]  6227.0  1.0
beta[13]  5729.0  1.0
beta[14]  6141.0  1.0
beta[15]  6457.0  1.0
beta[16]  6069.0  1.0
beta[17]  5954.0  1.0
beta[18]  6031.0  1.0
beta[19]  5907.0  1.0

[9]: axes = az.plot_trace(
     data=idata,
     var_names=["alpha", "length", "noise", "beta"],
     compact=True,
     kind="rank_bars",
     backend_kwargs={"figsize": (10, 7), "layout": "constrained"},
    )
plt.gcf().suptitle("Posterior Distributions", fontsize=16, fontweight="bold");

19.4. Fit the Model 425
Overall, the model seems to have converged well.

19.5 Posterior Predictive Distribution

Finally, we generate samples from the posterior predictive distribution on the test set and plot the results.

\[10\]: predictive = Predictive(model, mcmc.get_samples())
posterior_predictive = predictive(rng_subkey, x_test_centered, ell, m, non_centered)
                      rng_key, rng_subkey = random.split(rng_key)
idata.extend(az.from_numpyro(posterior_predictive=posterior_predictive))

\[11\]: fig, ax = plt.subplots()
ax.axvline(x=0, color="C2", alpha=0.8, linestyle="--", linewidth=2)
ax.axvline(x=1, color="C2", linestyle="--", alpha=0.8, linewidth=2, label="training range")
az.plot_hdi(x_test, idata.posterior_predictive["likelihood"],
             hdi_prob=0.94, color="C1",
             )

(continues on next page)
smooth=False,
fill_kwargs={"alpha": 0.1, "label": "$94\%$ HDI (test)"},
ax=ax,
)

az.plot_hdi(
    x_test,
    idata.posterior_predictive["likelihood"],
    hdi_prob=0.5,
    color="C1",
    smooth=False,
    fill_kwargs={"alpha": 0.3, "label": "$50\%$ HDI (test)"},
    ax=ax,
)

ax.plot(
    x_test,
    idata.posterior_predictive["likelihood"].mean(dim="chain", "draw"),
    color="C1",
    linewidth=3,
    label="posterior predictive mean (test)",
)

ax.scatter(x_train, y_train_obs, c="C0", label="observed (train)")
ax.plot(x_train, y_test, color="black", linewidth=3, alpha=0.7, label="mean (latent)")
ax.legend(loc="upper center", bbox_to_anchor=(0.5, -0.1), ncol=4)
ax.set(xlabel="x", ylabel="y")
ax.set_title("Posterior Predictive", fontsize=16, fontweight="bold");
The model did a good job of capturing the underlying function in the training set region. The uncertainty increases as we move away from the training set.

19.6 Idea of the Hilbert Space Approximation?

In this notebook we do not go into the details of the Hilbert Space Approximation. However, here we sketch the main idea.

We approximate the kernel function with a set of basis functions $\phi_j$ coming from the spectrum of the Dirichlet Laplacian in the box $[-\ell, \ell]$. There basis functions are independent of the kernel hyperparameters $\alpha$ and $\ell$.

The weights of these basis functions come from evaluating the spectral density $S(\omega)$ of the kernel function at the square roots of the eigenvalues $\lambda_j$ of the Dirichlet Laplacian. The final approximation formula looks like:

$$
 f(x) \approx \sum_{j=1}^{m} \left( S(\sqrt{\lambda_j}) \right)^{1/2} \times \phi_j(x) \times \beta_j \sim \text{Normal}(0,1)
$$

Let’s see the approximation components. First, we plot the basis functions.
These are weighted by spectral density values. The following plot shows the spectral evaluated on the square roots of the eigenvalues of the Dirichlet Laplacian. We use various values of the hyperparameters alpha and length to see how the spectral density changes. We also include in black the corresponding spectral density using the posterior mean inferred from the model above.

```python
alpha_posterior_mean = idata.posterior["alpha"].mean(dim=("chain", "draw")).item()
length_posterior_mean = idata.posterior["length"].mean(dim=("chain", "draw")).item()
```

```python
for alpha_value in (1.0, 1.5):
    for length_value in (0.05, 0.1):
        diag_sd = diag_spectral_density_squared_exponential(
            alpha=alpha_value,
            length=length_value,
            ell=ell,
            m=m,
            dim=1,
        )
```

(continues on next page)
ax.plot(
    range(1, m + 1),
    diag_sd,
    marker="o",
    linewidth=1.5,
    markersize=4,
    alpha=0.8,
    label=f"alpha = {alpha_value}, length = {length_value}"
)

diag_sd = diag_spectral_density_squared_exponential(
    alpha=alpha_posterior_mean,
    length=length_posterior_mean,
    ell=ell,
    m=m,
    dim=1,
)
ax.plot(
    range(1, m + 1),
    diag_sd,
    marker="o",
    color="black",
    linewidth=3,
    markersize=6,
    label=f"posterior mean (alpha = {alpha_posterior_mean:.2f}, length = {length_posterior_mean:.2f})",
)
ax.xaxis.set_major_locator(MultipleLocator())
ax.legend(loc="upper right", title="Hyperparameters")
ax.set_title(
    r"Spectral Density on the First $m$ (square root) Eigenvalues",
    fontsize=16,
    fontweight="bold",
);
As the spectral density decays to zero at higher frequencies, the effect of the larger eigenvalues becomes smaller. One can prove that in the limit, the Hilbert space approximation converges to the true kernel function.
GAUSSIAN MIXTURE MODEL

This tutorial demonstrates how to marginalize out discrete latent variables in NumPyro through the motivating example of a mixture model. We’ll focus on the mechanics of parallel enumeration, keeping the model simple by training a trivial 1-D Gaussian model on a tiny 5-point dataset. See also the enumeration tutorial for a broader introduction to parallel enumeration.

20.1 Table of contents

- Overview
- Training a MAP estimator
- Serving the model: predicting membership
  - Predicting membership using discrete inference
  - Predicting membership by enumerating in the guide
- MCMC

[ ]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[1]: from collections import defaultdict
import os
import matplotlib.pyplot as plt
import scipy.stats

import jax.scipy.stats

from jax import pure_callback, random
import jax.numpy as jnp
import optax

import numpyro
from numpyro import handlers
from numpyro.contrib.funsor import config_enumerate, infer_discrete
from numpyro.distributions import dist
from numpyro.distributions import constraints
from numpyro.infer import SVI, TraceEnum_ELBO, init_to_value
from numpyro.infer.autoguide import AutoDelta

%matplotlib inline

(import continues on next page)
NumPyro Documentation

20.1.1 Overview

NumPyro’s TraceEnum_ELBO can automatically marginalize out variables in both the guide and the model. When enumerating guide variables, NumPyro enumerates in parallel by allocating on the left a new array dimension and using nonstandard evaluation to create an array of possible values at the variable’s sample site. These nonstandard values are then replayed in the model. When enumerating variables in the model, the variables are enumerated in parallel and must not appear in the guide. Mathematically, guide-side enumeration simply reduces variance in stochastic ELBO estimates by exactly integrating out an enumerated variable, whereas model-side enumeration avoids an application of Jensen’s inequality by exactly marginalizing out an enumerated variable.

Here is our tiny dataset. It has five points.

```
[2]: data = jnp.array([0.0, 1.0, 10.0, 11.0, 12.0])
```

No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)

20.1.2 Training a MAP estimator

Let’s start by learning model parameters weights, locs, and scale given priors and data. We will learn point estimates of these using an AutoDelta guide (named after its delta distributions). Our model will learn global mixture weights, the location of each mixture component, and a shared scale that is common to both components. During inference, TraceEnum_ELBO will marginalize out the assignments of datapoints to clusters.

```
[3]: K = 2  # Fixed number of components.
```

```
@config_enumerate
def model(data):
  # Global variables.
  weights = numpyro.sample("weights", dist.Dirichlet(0.5 * jnp.ones(K)))
  scale = numpyro.sample("scale", dist.LogNormal(0.0, 2.0))
  with numpyro.plate("components", K):
    locs = numpyro.sample("locs", dist.Normal(0.0, 10.0))

  with numpyro.plate("data", len(data)):
    # Local variables.
    assignment = numpyro.sample("assignment", dist.Categorical(weights))
    numpyro.sample("obs", dist.Normal(locs[assignment], scale), obs=data)
```

To run inference with this (model, guide) pair, we use NumPyro’s config_enumerate handler to enumerate over all assignments in each iteration. Since we’ve wrapped the batched Categorical assignments in a numpyro.plate independence context along the data batch dimension, this enumeration can happen in parallel: we enumerate only 2 possibilities, rather than \(2^{\text{len(data)}} = 32\).

Before inference we’ll initialize to plausible values. Mixture models are very susceptible to local modes. A common approach is to choose the best among many random initializations, where the cluster means are initialized from random subsamples of the data. Since we’re using an AutoDelta guide, we can initialize by using a init_to_value helper function.
```python
[4]: elbo = TraceEnum_ELBO()

def initialize(seed):
    global global_guide
    init_values = {
        "weights": jnp.ones(K) / K,
        "scale": jnp.sqrt(data.var() / 2),
        "locs": data[  
            random.categorical(  
                random.PRNGKey(seed), jnp.ones(len(data)) / len(data), shape=(K,)  
            )  
        ],
    }

    global_model = handlers.block(  
        handlers.seed(model, random.PRNGKey(0)),
        hide_fn=lambda site: site["name"] not in ["weights", "scale", "locs", "components"],
    )

    global_guide = AutoDelta(  
        global_model, init_loc_fn=init_to_value(values=init_values)  
    )

    handlers.seed(global_guide, random.PRNGKey(0))(data)  
    # warm up the guide
    return elbo.loss(random.PRNGKey(0), {}, model, global_guide, data)

# Choose the best among 100 random initializations.
loss, seed = min((initialize(seed), seed) for seed in range(100))
ninitialize(seed)  
print(f"seed = {seed}, initial_loss = {loss}")

seed = 8, initial_loss = 25.149845123291016

During training, we’ll collect both losses and gradient norms to monitor convergence. We can do this using hook_optax helper function below.

[5]: # Helper function to collect gradient norms during training
def hook_optax(optimizer):
    gradient_norms = defaultdict(list)

    def append_grad(grad):
        for name, g in grad.items():
            gradient_norms[name].append(float(jnp.linalg.norm(g)))
        return grad

    def update_fn(grads, state, params=None):
        grads = pure_callback(append_grad, grads, grads)
        return optimizer.update(grads, state, params=params)

    return optax.GradientTransformation(optimizer.init, update_fn), gradient_norms

optim, gradient_norms = hook_optax(optax.adam(learning_rate=0.1, b1=0.8, b2=0.99))
global_svi = SVI(model, global_guide, optim, loss=elbo)
```
Now train the model:

```
[6]: global_svi_result = global_svi.run(
    random.PRNGKey(0), 200 if not smoke_test else 2, data
)
```

```
100 
˓→200/200 [00:00<00:00, 287.42it/s, init loss: 25.1498, avg. loss [191-200]: 17.4433]
```

```
[7]: plt.figure(figsize=(10, 3), dpi=100).set_facecolor("white")
plt.plot(global_svi_result.losses)
plt.xlabel("iters")
plt.ylabel("loss")
plt.yscale("log")
plt.title("Convergence of SVI")
plt.show()
```

```
[8]: plt.figure(figsize=(10, 4), dpi=100).set_facecolor("white")
  for name, grad_norms in gradient_norms.items():
    plt.plot(grad_norms, label=name)
plt.xlabel("iters")
plt.ylabel("gradient norm")
plt.yscale("log")
plt.legend(loc="best")
plt.title("Gradient norms during SVI")
plt.show()
```
Here are the learned parameters:

```python
map_estimates = global_svi_result.params
weights = map_estimates["weights_auto_loc"]
locs = map_estimates["locs_auto_loc"]
scale = map_estimates["scale_auto_loc"]
print(f"weights = {weights}")
print(f"locs = {locs}")
print(f"scale = {scale}")
weights = [0.375 0.625]
locs = [ 0.4989534 10.984944 ]
scale = 0.6514341831207275
```

The model’s weights are as expected, with about 2/5 of the data in the first component and 3/5 in the second component. Next let’s visualize the mixture model.

```python
X = jnp.arange(-3, 15, 0.1)
Y1 = weights[0] * scipy.stats.norm.pdf((X - locs[0]) / scale)
Y2 = weights[1] * scipy.stats.norm.pdf((X - locs[1]) / scale)

plt.figure(figsize=(10, 4), dpi=100).set_facecolor("white")
plt.plot(X, Y1, "r-")
plt.plot(X, Y2, "b-"
plt.plot(X, Y1 + Y2, "k--")
plt.plot(data, jnp.zeros(len(data)), "k*")
plt.title("Density of two-component mixture model")
plt.ylabel("probability density")
plt.show()
```
Finally note that optimization with mixture models is non-convex and can often get stuck in local optima. For example in this tutorial, we observed that the mixture model gets stuck in an everything-in-one-cluster hypothesis if `scale` is initialized to be too large.

### 20.1.3 Serving the model: predicting membership

Now that we’ve trained a mixture model, we might want to use the model as a classifier. During training we marginalized out the assignment variables in the model. While this provides fast convergence, it prevents us from reading the cluster assignments from the guide. We’ll discuss two options for treating the model as a classifier: first using `infer_discrete` (much faster) and second by training a secondary guide using enumeration inside SVI (slower but more general).

**Predicting membership using discrete inference**

The fastest way to predict membership is to use the `infer_discrete` handler, together with `trace` and `replay`. Let’s start out with a MAP classifier, setting `infer_discrete`’s temperature parameter to zero. For a deeper look at effect handlers like `trace`, `replay`, and `infer_discrete`, see the effect handler tutorial.

```python
[11]: trained_global_guide = handlers.substitute(  
    global_guide, global_svi_result.params  
  )  
  # substitute trained params

guide_trace = handlers.trace(trained_global_guide).get_trace(data)  
  # record the globals
trained_model = handlers.replay(model, trace=guide_trace)  
  # replay the globals

def classifier(data, temperature=0, rng_key=None):
    inferred_model = infer_discrete(  
      trained_model, temperature=temperature, first_available_dim=-2, rng_key=rng_key  
    )  
      # set first_available_dim to avoid conflict with data plate
    seeded_inferred_model = handlers.seed(inferred_model, random.PRNGKey(0))
    trace = handlers.trace(seeded_inferred_model).get_trace(data)
    return trace["assignment"]["value"]
```

(continues on next page)
Indeed we can run this classifier on new data.

```
[12]: new_data = jnp.arange(-3, 15, 0.1)
assignment = classifier(new_data)
plt.figure(figsize=(8, 2), dpi=100).set_facecolor("white")
plt.plot(new_data, assignment)
plt.title("MAP assignment")
plt.xlabel("data value")
plt.ylabel("class assignment")
plt.show()
```

To generate random posterior assignments rather than MAP assignments, we could set temperature=1.

```
[13]: print(classifier(data, temperature=1, rng_key=random.PRNGKey(0)))
[0 0 1 1 1]
```

Since the classes are very well separated, we zoom in to the boundary between classes, around 5.75.

```
[14]: new_data = jnp.arange(5.5, 6.0, 0.005)
assignment = classifier(new_data, temperature=1, rng_key=random.PRNGKey(0))
plt.figure(figsize=(8, 2), dpi=100).set_facecolor("white")
plt.plot(new_data, assignment, "x", color="C0")
plt.title("Random posterior assignment")
plt.xlabel("data value")
plt.ylabel("class assignment")
plt.show()
```
Predicting membership by enumerating in the guide

A second way to predict class membership is to enumerate in the guide. This doesn’t work well for serving classifier models, since we need to run stochastic optimization for each new input data batch, but it is more general in that it can be embedded in larger variational models.

To read cluster assignments from the guide, we’ll define a new full_guide that fits both global parameters (as above) and local parameters (which were previously marginalized out). Since we’ve already learned good values for the global variables, we will block SVI from updating those by using handlers.block.

```python
[15]: @config_enumerate
def full_guide(data):
    # Global variables.
    with handlers.block(
        hide=["weights_auto_loc", "locs_auto_loc", "scale_auto_loc"]
    ):
        # Keep our learned values of global parameters.
        trained_global_guide(data)

    # Local variables.
    with numpyro.plate("data", len(data)):
        assignment_probs = numpyro.param("assignment_probs",
                                          jnp.ones((len(data), K)) / K,
                                          constraint=constraints.simplex,
                                          )
        numpyro.sample("assignment", dist.Categorical(assignment_probs))

[16]: optim, gradient_norms = hook_optax(optax.adam(learning_rate=0.2, b1=0.8, b2=0.99))
elbo = TraceEnum_ELBO()
full_svi = SVI(model, full_guide, optim, loss=elbo)

full_svi_result = full_svi.run(random.PRNGKey(0), 200 if not smoke_test else 2, data)
```

```
100 ˓→%|
˓→200/200 [00:00<00:00, 298.62it/s, init loss: 338.6479, avg. loss [191-200]: 18.2659]
```

Chapter 20. Gaussian Mixture Model
We can now examine the guide's local assignment_probs variable.
```python
assignment_probs = full_svi_result.params["assignment_probs"]
plt.figure(figsize=(8, 3), dpi=100).set_facecolor("white")
plt.plot(    data,
    assignment_probs[:, 0],
    "ro",
    label=f"component with mean {locs[0]:.2g}",
)
plt.plot(    data,
    assignment_probs[:, 1],
    "bo",
    label=f"component with mean {locs[1]:.2g}",
)
plt.title("Mixture assignment probabilities")
plt.xlabel("data value")
plt.ylabel("assignment probability")
plt.legend(loc="center")
plt.show()
```

### 20.1.4 MCMC

Next we'll explore the full posterior over component parameters using collapsed NUTS, i.e. we'll use NUTS and marginalize out all discrete latent variables.

```python
from numpyro.infer import MCMC, NUTS

kernel = NUTS(model)
mcmc = MCMC(kernel, num_warmup=50, num_samples=250)
mcmc.run(random.PRNGKey(2), data)
mcmc.print_summary()
posterior_samples = mcmc.get_samples()
```
```python
[21]: X, Y = posterior_samples["locs"].T

[22]: plt.figure(figsize=(8, 8), dpi=100).set_facecolor("white")
   h, xs, ys, image = plt.hist2d(X, Y, bins=[20, 20])
   plt.contour(
       jnp.log(h + 3).T,
       extent=[xs.min(), xs.max(), ys.min(), ys.max()],
       colors="white",
       alpha=0.8,
   )
   plt.title("Posterior density as estimated by collapsed NUTS")
   plt.xlabel("loc of component 0")
   plt.ylabel("loc of component 1")
   plt.tight_layout()
   plt.show()
```

<table>
<thead>
<tr>
<th>locs[0]</th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.45</td>
<td>4.30</td>
<td>0.62</td>
<td>-0.97</td>
<td>11.41</td>
<td>8.92</td>
<td>1.14</td>
<td></td>
</tr>
<tr>
<td>locs[1]</td>
<td>8.72</td>
<td>4.19</td>
<td>10.75</td>
<td>0.23</td>
<td>11.78</td>
<td>8.12</td>
<td>1.16</td>
</tr>
<tr>
<td>scale</td>
<td>1.58</td>
<td>1.75</td>
<td>1.02</td>
<td>0.57</td>
<td>3.46</td>
<td>19.19</td>
<td>1.02</td>
</tr>
<tr>
<td>weights[0]</td>
<td>0.47</td>
<td>0.20</td>
<td>0.48</td>
<td>0.17</td>
<td>0.76</td>
<td>8.78</td>
<td>1.03</td>
</tr>
<tr>
<td>weights[1]</td>
<td>0.53</td>
<td>0.20</td>
<td>0.52</td>
<td>0.24</td>
<td>0.83</td>
<td>8.78</td>
<td>1.03</td>
</tr>
</tbody>
</table>

Number of divergences: 11
Note that due to nonidentifiability of the mixture components the likelihood landscape has two equally likely modes, near $(11,0.5)$ and $(0.5,11)$. NUTS has difficulty switching between the two modes.

```python
plt.figure(figsize=(8, 3), dpi=100).set_facecolor("white")
plt.plot(X, color="red")
plt.plot(Y, color="blue")
plt.xlabel("NUTS step")
plt.ylabel("loc")
plt.title("Trace plot of loc parameter during NUTS inference")
plt.tight_layout()
plt.show()
```
EXAMPLE: TOY MIXTURE MODEL WITH DISCRETE ENUMERATION

A toy mixture model to provide a simple example for implementing discrete enumeration:

\[(A) \rightarrow [B] \rightarrow (C)\]

A is an observed Bernoulli variable with Beta prior. B is a hidden variable which is a mixture of two Bernoulli distributions (with Beta priors), chosen by A being true or false. C is observed, and like B, is a mixture of two Bernoulli distributions (with Beta priors), chosen by B being true or false. There is a plate over the three variables for num_obs independent observations of data.

Because B is hidden and discrete we wish to marginalize it out of the model. This is done by:

1. marking the model with @config Enumerate
2. marking the B sample site in the model with infer="enumerate": "parallel"
3. passing SVI the TraceEnum_ELBO loss function

```python
import argparse
import matplotlib.pyplot as plt
from jax import random
import jax.numpy as jnp
import optax
import numpyro
from numpyro import handlers
from numpyro.contrib.funsor import config_enumerate
import numpyro.distributions as dist
from numpyro.distributions import constraints
from numpyro.infer import SVI, TraceEnum_ELBO
from numpyro.ops.indexing import Vindex

def main(args):
    num_obs = args.num_obs
    num_steps = args.num_steps
    prior, CPDs, data = handlers.seed(generate_data, random.PRNGKey(0))(num_obs)
    posterior_params = train(prior, data, num_steps, num_obs)
    evaluate(CPDs, posterior_params)

def generate_data(num_obs):
```

(continues on next page)
# domain = [False, True]

prior = {
    "A": jnp.array([1.0, 10.0]),
    "B": jnp.array([[10.0, 1.0], [1.0, 10.0]]),
    "C": jnp.array([[10.0, 1.0], [1.0, 10.0]]),
}

CPDs = {
    "p_A": numpyro.sample("p_A", dist.Beta(prior["A"][:, 0], prior["A"][:, 1])),
    "p_B": numpyro.sample("p_B", dist.Beta(prior["B"][:, 0], prior["B"][:, 1])),
    "p_C": numpyro.sample("p_C", dist.Beta(prior["C"][:, 0], prior["C"][:, 1])),
}

data = {
    "A": numpyro.sample("A", dist.Bernoulli(jnp.ones(num_obs) * CPDs["p_A"])),
    "B": numpyro.sample("B", dist.Bernoulli(CPDs["p_B"][:, 0] * CPDs["p_A"])),
    "C": numpyro.sample("C", dist.Bernoulli(CPDs["p_C"][:, 0] * CPDs["p_B"][:, 0])),
}

@config_enumerate
def model(prior, obs, num_obs):
    p_A = numpyro.sample("p_A", dist.Beta(1, 1))
    p_B = numpyro.sample("p_B", dist.Beta(jnp.ones(2), jnp.ones(2)).to_event(1))
    p_C = numpyro.sample("p_C", dist.Beta(jnp.ones(2), jnp.ones(2)).to_event(1))
    with numpyro.plate("data_plate", num_obs):
        A = numpyro.sample("A", dist.Bernoulli(p_A), obs=obs["A"])
        B = numpyro.sample(
            "B",
            dist.Bernoulli(Vindex(p_B)[A]),
            infer={"enumerate": "parallel"},
        )
        numpyro.sample("C", dist.Bernoulli(Vindex(p_C)[B]), obs=obs["C"])

def guide(prior, obs, num_obs):
    a = numpyro.param("a", prior["A"], constraint=constraints.positive)
    numpyro.sample("p_A", dist.Beta(a[0], a[1]))
    b = numpyro.param("b", prior["B"], constraint=constraints.positive)
    numpyro.sample("p_B", dist.Beta(b[:, 0], b[:, 1]).to_event(1))
    c = numpyro.param("c", prior["C"], constraint=constraints.positive)
    numpyro.sample("p_C", dist.Beta(c[:, 0], c[:, 1]).to_event(1))

def train(prior, data, num_steps, num_obs):
    elbo = TraceEnum_ELBO()
    svi = SVI(model, guide, optax.adam(learning_rate=0.01), loss=elbo)
    svi_result = svi.run(random.PRNGKey(0), num_steps, prior, data, num_obs)
    plt.figure()
    plt.plot(svi_result.losses)
    plt.show()
    posterior_params = svi_result.params.copy()
    posterior_params["a"] = posterior_params["a"][None, :]

    (continues on next page)
def evaluate(CPDs, posterior_params):
    true_p_A, pred_p_A = get_true_pred_CPDs(CPDs["p_A"], posterior_params["a"], posterior_params["c"])
    true_p_B, pred_p_B = get_true_pred_CPDs(CPDs["p_B"], posterior_params["b"])
    true_p_C, pred_p_C = get_true_pred_CPDs(CPDs["p_C"], posterior_params["c"])
    print("p_A = True")
    print("actual: ", true_p_A)
    print("predicted:", pred_p_A)
    print("p_B = True | A = False/True")
    print("actual: ", true_p_B)
    print("predicted:", pred_p_B)
    print("p_C = True | B = False/True")
    print("actual: ", true_p_C)
    print("predicted:", pred_p_C)

def get_true_pred_CPDs(CPD, posterior_param):
    true_p = CPD
    pred_p = posterior_param[:, 0] / jnp.sum(posterior_param, axis=1)
    return true_p, pred_p

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Toy mixture model")
    parser.add_argument("-n", "--num-steps", default=4000, type=int)
    parser.add_argument("-o", "--num-obs", default=10000, type=int)
    args = parser.parse_args()
    main(args)
EXAMPLE: BAYESIAN MODELS OF ANNOTATION

In this example, we run MCMC for various crowdsourced annotation models in [1].

All models have discrete latent variables. Under the hood, we enumerate over (marginalize out) those discrete latent sites in inference. Those models have different complexity so they are great references for those who are new to Pyro/NumPyro enumeration mechanism. We recommend readers compare the implementations with the corresponding plate diagrams in [1] to see how concise a Pyro/NumPyro program is.

The interested readers can also refer to [3] for more explanation about enumeration.

The data is taken from Table 1 of reference [2].

Currently, this example does not include postprocessing steps to deal with “Label Switching” issue (mentioned in section 6.2 of [1]).

References:


```python
import argparse
import os

import numpy as np
from jax import nn, random, vmap
import jax.numpy as jnp

import numpyro
from numpyro import handlers
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive
from numpyro.infer.reparam import LocScaleReparam
from numpyro.ops.indexing import Vindex

def get_data():
    """
    :return: a tuple of annotator indices and class indices. The first term has shape `num_positions` whose entries take values from `0` to `num_annotators - 1`.
    """
```
The second term has shape `num_items x num_positions` whose entries take values from `0` to `num_classes - 1`.

# NB: the first annotator assessed each item 3 times
positions = np.array([1, 1, 1, 2, 3, 4, 5])
# fmt: off
annotations = np.array(
    [[1, 1, 1, 1, 1, 1, 1], [3, 3, 3, 4, 3, 3, 4], [1, 1, 2, 2, 1, 2, 2],
     [2, 2, 2, 3, 1, 2, 2], [2, 2, 2, 3, 3, 2, 2], [2, 2, 2, 2, 3, 2, 2],
     [1, 1, 1, 1, 1, 1, 1], [3, 3, 3, 3, 4, 3, 3], [2, 2, 2, 2, 2, 2, 2],
     [2, 3, 2, 2, 2, 3, 2], [4, 4, 4, 4, 4, 4, 4], [2, 2, 2, 3, 3, 4, 3],
     [1, 1, 1, 1, 1, 1, 1], [3, 3, 3, 3, 3, 3, 3], [2, 2, 2, 2, 2, 2, 2],
     [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1],
     [2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2],
     [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1],
     [1, 1, 1, 1, 1, 1], [3, 3, 3, 3, 2, 3, 3], [1, 1, 1, 1, 1, 1, 1],
     [2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2], [4, 3, 3, 4, 3, 4, 3],
     [2, 2, 2, 2, 2, 2], [2, 3, 2, 2, 2, 3, 3], [3, 3, 3, 3, 4, 3, 2],
     [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1],
     [2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2],
    ])
# fmt: on
return positions - 1, annotations - 1

def multinomial(annotations):
    # This model corresponds to the plate diagram in Figure 1 of reference [1].
    num_classes = int(np.max(annotations)) + 1
    num_items, num_positions = annotations.shape

    with numpyro.plate("class", num_classes):
        zeta = numpyro.sample("zeta", dist.Dirichlet(jnp.ones(num_classes)))

    pi = numpyro.sample("pi", dist.Dirichlet(jnp.ones(num_classes)))

    with numpyro.plate("item", num_items, dim=-2):
        c = numpyro.sample("c", dist.Categorical(pi), infer={"enumerate": "parallel"})

    with numpyro.plate("position", num_positions):
        numpyro.sample("y", dist.Categorical(zeta[c]), obs=annotations)

def dawid_skene(positions, annotations):
    # This model corresponds to the plate diagram in Figure 2 of reference [1].
    num_annotators = int(np.max(positions)) + 1
    num_classes = int(np.max(annotations)) + 1
```python
num_items, num_positions = annotations.shape

with numpyro.plate("annotator", num_annotators, dim=-2):
    with numpyro.plate("class", num_classes):
        beta = numpyro.sample("beta", dist.Dirichlet(jnp.ones(num_classes)))

pi = numpyro.sample("pi", dist.Dirichlet(jnp.ones(num_classes)))

with numpyro.plate("item", num_items, dim=-2):
    c = numpyro.sample("c", dist.Categorical(pi), infer={"enumerate": "parallel"})

    # here we use Vindex to allow broadcasting for the second index 'c'

    with numpyro.plate("position", num_positions):
        numpyro.sample("y", dist.Categorical(Vindex(beta)[positions, c, :]), obs=annotations)

def mace(positions, annotations):
    ""
    This model corresponds to the plate diagram in Figure 3 of reference [1].
    ""
    num_annotators = int(np.max(positions)) + 1
    num_classes = int(np.max(annotations)) + 1
    num_items, num_positions = annotations.shape

    with numpyro.plate("annotator", num_annotators):
        epsilon = numpyro.sample("epsilon", dist.Dirichlet(jnp.full(num_classes, 10)))
        theta = numpyro.sample("theta", dist.Beta(0.5, 0.5))

    with numpyro.plate("item", num_items, dim=-2):
        c = numpyro.sample("c",
            dist.DiscreteUniform(0, num_classes - 1),
            infer={"enumerate": "parallel"},
        )

    with numpyro.plate("position", num_positions):
        s = numpyro.sample("s",
            dist.Bernoulli(1 - theta[positions]),
            infer={"enumerate": "parallel"},
        )

        probs = jnp.where(
            s[... None] == 0, nn.one_hot(c, num_classes), epsilon[positions]
        )

        numpyro.sample("y", dist.Categorical(probs), obs=annotations)
```

```
This model corresponds to the plate diagram in Figure 4 of reference [1].

```python
num_annotators = int(np.max(positions)) + 1
num_classes = int(np.max(annotations)) + 1
num_items, num_positions = annotations.shape

with numpyro.plate("class", num_classes):
    # NB: we define 'beta' as the 'logits' of 'y' likelihood; but 'logits' is
    # invariant up to a constant, so we'll follow [1]: fix the last term of 'beta'
    # to 0 and only define hyperpriors for the first 'num_classes - 1' terms.
    zeta = numpyro.sample("zeta", dist.Normal(0, 1).expand([num_classes - 1]).to_event(1))
    omega = numpyro.sample("Omega", dist.HalfNormal(1).expand([num_classes - 1]).to_event(1))

    with numpyro.plate("annotator", num_annotators, dim=-2):
        with numpyro.plate("class", num_classes):
            # non-centered parameterization
            with handlers.reparam(config={"beta": LocScaleReparam(0)}):
                beta = numpyro.sample("beta", dist.Normal(zeta, omega).to_event(1))
            # pad 0 to the last item
            beta = jnp.pad(beta, [(0, 0)] * (jnp.ndim(beta) - 1) + [(0, 1)])

    pi = numpyro.sample("pi", dist.Dirichlet(jnp.ones(num_classes)))

with numpyro.plate("item", num_items, dim=-2):
    c = numpyro.sample("c", dist.Categorical(pi), infer="enumerate": "parallel")

with numpyro.plate("position", num_positions):
    logits = Vindex(beta)[positions, c, :]
    numpyro.sample("y", dist.Categorical(logits=logits), obs=annotations)
```

---

This model corresponds to the plate diagram in Figure 5 of reference [1].

```python
num_classes = int(np.max(annotations)) + 1
num_items, num_positions = annotations.shape

with numpyro.plate("class", num_classes):
    eta = numpyro.sample("eta", dist.Normal(0, 1).expand([num_classes - 1]).to_event(1))
    chi = numpyro.sample("Chi", dist.HalfNormal(1).expand([num_classes - 1]).to_event(1))

    pi = numpyro.sample("pi", dist.Dirichlet(jnp.ones(num_classes)))
```

---

(continues on next page)
with numpyro.plate("item", num_items, dim=-2):
c = numpyro.sample("c", dist.Categorical(pi), infer={"enumerate": "parallel"})

with handlers.reparam(config={"theta": LocScaleReparam(0)}):
    theta = numpyro.sample("theta", dist.Normal(eta[c], chi[c]).to_event(1))
    theta = jnp.pad(theta, [(0, 0)] * (jnp.ndim(theta) - 1) + [(0, 1)])

with numpyro.plate("position", annotations.shape[-1]):
numpyro.sample("y", dist.Categorical(logits=theta), obs=annotations)

def logistic_random_effects(positions, annotations):
    """
    This model corresponds to the plate diagram in Figure 5 of reference [1].
    """
    num_annotators = int(np.max(positions)) + 1
    num_classes = int(np.max(annotations)) + 1
    num_items, num_positions = annotations.shape

    with numpyro.plate("class", num_classes):
        zeta = numpyro.sample("zeta", dist.Normal(0, 1).expand([num_classes - 1]).to_event(1))
        omega = numpyro.sample("Omega", dist.HalfNormal(1).expand([num_classes - 1]).to_event(1))
        chi = numpyro.sample("Chi", dist.HalfNormal(1).expand([num_classes - 1]).to_event(1))

    with numpyro.plate("annotator", num_annotators, dim=-2):
        with numpyro.plate("class", num_classes):
            with handlers.reparam(config={"beta": LocScaleReparam(0)}):
                beta = numpyro.sample("beta", dist.Normal(zeta, omega).to_event(1))
                beta = jnp.pad(beta, [(0, 0)] * (jnp.ndim(beta) - 1) + [(0, 1)])

        pi = numpyro.sample("pi", dist.Dirichlet(jnp.ones(num_classes)))

    with numpyro.plate("item", num_items, dim=-2):
        c = numpyro.sample("c", dist.Categorical(pi), infer={"enumerate": "parallel"})

    with handlers.reparam(config={"theta": LocScaleReparam(0)}):
        theta = numpyro.sample("theta", dist.Normal(0, chi[c]).to_event(1))
        theta = jnp.pad(theta, [(0, 0)] * (jnp.ndim(theta) - 1) + [(0, 1)])

    with numpyro.plate("position", num_positions):
        logits = Vindex(beta)[positions, c, :] - theta
        numpyro.sample("y", dist.Categorical(logits=logits), obs=annotations)

NAME_TO_MODEL = {
(continues on next page)
"mn": multinomial,
"ds": dawid_skene,
"mace": mace,
"hds": hierarchical_dawid_skene,
"id": item_difficulty,
"lre": logistic_random_effects,
}

```python
def main(args):
    annotators, annotations = get_data()
    model = NAME_TO_MODEL[args.model]
    data = (
        (annotations,)
        if model in [multinomial, item_difficulty]
        else (annotators, annotations)
    )
    mcmc = MCMC(
        NUTS(model),
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(random.PRNGKey(0), *data)
    mcmc.print_summary()
    posterior_samples = mcmc.get_samples()
    predictive = Predictive(model, posterior_samples, infer_discrete=True)
    discrete_samples = predictive(random.PRNGKey(1), *data)
    item_class = vmap(lambda x: jnp.bincount(x, length=4), in_axes=1)(
        discrete_samples["c"].squeeze(-1)
    )
    print("Histogram of the predicted class of each item:")
    row_format = "{:10}" * 5
    print(row_format.format("", *["c={}".format(i) for i in range(4)]))
    for i, row in enumerate(item_class):
        print(row_format.format(f"item[{i}]", *row))
```

### Note

In the above inference code, we marginalized the discrete latent variables $c$ hence `mcmc.get_samples(...)` does not include samples of $c$. We then utilize `predictive(...., infer_discrete=True)` to get posterior samples for $c$, which is stored in `discrete_samples`. To merge those discrete samples into the `mcmc` instance, we can use the following pattern:

```python
chain_discrete_samples = jax.tree.map(
    lambda x: x.reshape((args.num_chains, args.num_samples) + x.shape[1:]),
    discrete_samples)
mcmc.get_samples().update(discrete_samples)
mcmc.get_samples(group_by_chain=True).update(chain_discrete_samples)
```
This is useful when we want to pass the `mcmc` instance to `arviz` through `arviz.from_numpyro(mcmc)`.

```python
if __name__ == '__main__':
    assert numpyro.__version__.startswith('0.15.2')
    parser = argparse.ArgumentParser(description='Bayesian Models of Annotation')
    parser.add_argument('--num-samples', nargs='?', default=1000, type=int)
    parser.add_argument('--num-warmup', nargs='?', default=1000, type=int)
    parser.add_argument('--num-chains', nargs='?', default=1, type=int)
    parser.add_argument('--model', nargs='?', default='ds', help='one of "mn" (multinomial), "ds" (dawid_skene), "mace",
    "hds" (hierarchical_dawid_skene),
    "id" (item_difficulty), "lre" (logistic_random_effects)',
    )
    parser.add_argument('--device', default='cpu', type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```
CHAPTER TWENTYTHREE

EXAMPLE: ENUMERATE HIDDEN MARKOV MODEL

This example is ported from [1], which shows how to marginalize out discrete model variables in Pyro.

This combines MCMC with a variable elimination algorithm, where we use enumeration to exactly marginalize out some variables from the joint density.

To marginalize out discrete variables $x$:

1. Verify that the variable dependency structure in your model admits tractable inference, i.e. the dependency graph among enumerated variables should have narrow treewidth.

2. Ensure your model can handle broadcasting of the sample values of those variables.

Note that difference from [1], which uses Python loop, here we use scan() to reduce compilation times (only one step needs to be compiled) of the model. Under the hood, scan stacks all the priors’ parameters and values into an additional time dimension. This allows us computing the joint density in parallel. In addition, the stacked form allows us to use the parallel-scan algorithm in [2], which reduces parallel complexity from $O(\text{length})$ to $O(\log(\text{length}))$.

Data are taken from [3]. However, the original source of the data seems to be the Institut fuer Algorithmen und Kognitive Systeme at Universitaet Karlsruhe.

References:

1. Pyro’s Hidden Markov Model example, (https://pyro.ai/examples/hmm.html)


```python
import argparse
import logging
import os
import time
from jax import random
import jax.numpy as jnp
import numpyro
from numpyro.contrib.control_flow import scan
import numpyro.distributions as dist
from numpyro.examples.datasets import JSB_CHORALES, load_dataset
from numpyro.handlers import mask
```

(continues on next page)
from numpyro.infer import HMC, MCMC, NUTS
from numpyro.ops.indexing import Vindex

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

Let's start with a simple Hidden Markov Model.

```python
# x[t-1] --> x[t] --> x[t+1]
# | | |
# V V V
# y[t-1] y[t] y[t+1]
#
# This model includes a plate for the data_dim = 44 keys on the piano. This
# model has two "style" parameters probs_x and probs_y that we'll draw from a
# prior. The latent state is x, and the observed state is y.
def model_1(sequences, lengths, args, include_prior=True):
    num_sequences, max_length, data_dim = sequences.shape
    with mask(mask=include_prior):
        probs_x = numpyro.sample(
            "probs_x", dist.Dirichlet(0.9 * jnp.eye(args.hidden_dim) + 0.1).to_event(1)
        )
        probs_y = numpyro.sample(
            "probs_y",
            dist.Beta(0.1, 0.9).expand([args.hidden_dim, data_dim]).to_event(2),
        )

def transition_fn(carry, y):
    x_prev, t = carry
    with numpyro.plate("sequences", num_sequences, dim=-2):
        x = numpyro.sample(
            "x",
            dist.Categorical(probs_x[x_prev]),
            infer={"enumerate": "parallel"},
        )
        with numpyro.plate("tones", data_dim, dim=-1):
            numpyro.sample("y", dist.Bernoulli(probs_y[x.squeeze(-1)]), obs=y)
    return (x, t + 1), None

x_init = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
scan(transition_fn, (x_init, 0), jnp.swapaxes(sequences, 0, 1))
```

Next let's add a dependency of y[t] on y[t-1].

```python
# x[t-1] --> x[t] --> x[t+1]
# | | |
# V V V
# y[t-1] --> y[t] --> y[t+1]
def model_2(sequences, lengths, args, include_prior=True):
    num_sequences, max_length, data_dim = sequences.shape
```

(continues on next page)
with mask(mask=include_prior):
    probs_x = numpyro.sample(
        "probs_x", dist.Dirichlet(0.9 * jnp.eye(args.hidden_dim) + 0.1).to_event(1)
    )

    probs_y = numpyro.sample(
        "probs_y",
        dist.Beta(0.1, 0.9).expand([args.hidden_dim, 2, data_dim]).to_event(3),
    )

def transition_fn(carry, y):
    x_prev, y_prev, t = carry
    with numpyro.plate("sequences", num_sequences, dim=-2):
        with mask(mask=(t < lengths)[..., None]):
            x = numpyro.sample(
                "x",
                dist.Categorical(probs_x[x_prev]),
                infer={"enumerate": "parallel"},
            )
            # Note the broadcasting tricks here: to index probs_y on tensors x and y,
            # we also need a final tensor for the tones dimension. This is conveniently
            # provided by the plate associated with that dimension.
            with numpyro.plate("tones", data_dim, dim=-1) as tones:
                y = numpyro.sample(
                    "y", dist.Bernoulli(probs_y[x, y_prev, tones]), obs=y
                )
    return (x, y, t + 1), None

x_init = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
y_init = jnp.zeros((num_sequences, data_dim), dtype=jnp.int32)
scan(transition_fn, (x_init, y_init, 0), jnp.swapaxes(sequences, 0, 1))

Next consider a Factorial HMM with two hidden states.

```
# w[t-1] ----> w[t] ----> w[t+1]
# \ x[t-1] --\-> x[t] --\-> x[t+1]
# \ / \ / \ / \\
# \ / \ / \ /
# y[t-1] y[t] y[t+1]
#
# Note that since the joint distribution of each y[t] depends on two variables,
# those two variables become dependent. Therefore during enumeration, the
# entire joint space of these variables w[t],x[t] needs to be enumerated.
# For that reason, we set the dimension of each to the square root of the
# target hidden dimension.

def model_3(sequences, lengths, args, include_prior=True):
    num_sequences, max_length, data_dim = sequences.shape
    hidden_dim = int((args.hidden_dim)**0.5)  # split between w and x
    with mask(mask=include_prior):
        probs_w = numpyro.sample(
            "probs_w", dist.Dirichlet(0.9 * jnp.eye(hidden_dim) + 0.1).to_event(1)
        )
```
```python
probs_x = numpyro.sample("probs_x", dist.Dirichlet(0.9 * jnp.eye(hidden_dim) + 0.1).to_event(1))
probs_y = numpyro.sample("probs_y",
dist.Beta(0.1, 0.9).expand([args.hidden_dim, 2, data_dim]).to_event(3),
)

def transition_fn(carry, y):
    w_prev, x_prev, t = carry
    with numpyro.plate("sequences", num_sequences, dim=-2):
        with mask(mask=(t < lengths)[..., None]):
            w = numpyro.sample("w",
dist.Categorical(probs_w[w_prev]),
infer={"enumerate": "parallel"},
        )
            x = numpyro.sample("x",
dist.Categorical(probs_x[x_prev]),
infer={"enumerate": "parallel"},
        )
            # Note the broadcasting tricks here: to index probs_y on tensors x and y,
            # we also need a final tensor for the tones dimension. This is
            # provided by the plate associated with that dimension.
            with numpyro.plate("tones", data_dim, dim=-1) as tones:
                numpyro.sample("y", dist.Bernoulli(probs_y[w, x, tones]), obs=y)
    return (w, x, t + 1), None

w_init = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
x_init = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
scan(transition_fn, (w_init, x_init, 0), jnp.swapaxes(sequences, 0, 1))
```

By adding a dependency of x on w, we generalize to a Dynamic Bayesian Network.

```python
# w[t-1] ----> w[t] ----> w[t+1]
# | \ | \ | \
# | x[t-1] ----> x[t] ----> x[t+1]
# | / | / | /
# y[t-1] y[t] y[t+1]
# # Note that message passing here has roughly the same cost as with the
# # Factorial HMM, but this model has more parameters.

def model_4(sequences, lengths, args, include_prior=True):
    num_sequences, max_length, data_dim = sequences.shape
    hidden_dim = int(args.hidden_dim**0.5)  # split between w and x
    with mask(mask=include_prior):
        probs_w = numpyro.sample("probs_w", dist.Dirichlet(0.9 * jnp.eye(hidden_dim) + 0.1).to_event(1)
```

(continues on next page)
probs_x = numpyro.sample("probs_x",
    dist.Dirichlet(0.9 * jnp.eye(hidden_dim) + 0.1)
        .expand_by([hidden_dim]).to_event(2),
    )
probs_y = numpyro.sample("probs_y",
    dist.Beta(0.1, 0.9).expand([hidden_dim, hidden_dim, data_dim]).to_event(3),
)

def transition_fn(carry, y):
    w_prev, x_prev, t = carry
    with numpyro.plate("sequences", num_sequences, dim=-2):
        with mask(mask=(t < lengths)[...,
334
        w = numpyro.sample("w",
            dist.Categorical(probs_w[w_prev]),
            infer="enumerate": "parallel"),
        )
        x = numpyro.sample("x",
            dist.Categorical(Vindex(probs_x)[w, x_prev]),
            infer="enumerate": "parallel"),
        )
        with numpyro.plate("tones", data_dim, dim=-1) as tones:
            numpyro.sample("y",
                dist.Bernoulli(probs_y[w, x, tones]), obs=y)
    return (w, x, t + 1), None

w_init = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
x_init = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
scan(transition_fn, (w_init, x_init, 0), jnp.swapaxes(sequences, 0, 1))

Next let’s consider a second-order HMM model in which x[t+1] depends on both x[t] and x[t-1].

# _______>______
# _____>_____/______ \
# / / \
# x[t-1] --> x[t] --> x[t+1] --> x[t+2]
# | | |
# V V V
# y[t-1] y[t] y[t+1] y[t+2]
#
# Note that in this model (in contrast to the previous model) we treat
# the transition and emission probabilities as parameters (so they have no prior).
# Note that this is the "2HMM" model in reference [4].

def model_6(sequences, lengths, args, include_prior=False):
    num_sequences, max_length, data_dim = sequences.shape
    with mask(mask=include_prior):
        (continues on next page)
# Explicitly parameterize the full tensor of transition probabilities, which has hidden_dim cubed entries.
probs_x = numpyro.sample('probs_x',
    dist.Dirichlet(0.9 * jnp.eye(args.hidden_dim) + 0.1)
    .expand([args.hidden_dim, args.hidden_dim])
    .to_event(2),
)

probs_y = numpyro.sample('probs_y',
    dist.Beta(0.1, 0.9).expand([args.hidden_dim, data_dim]).to_event(2),
)

def transition_fn(carry, y):
    x_prev, x_curr, t = carry
    with numpyro.plate("sequences", num_sequences, dim=-2):
        with mask(mask=(t < lengths)[...,
            None]):
            probs_x_t = Vindex(probs_x)[x_prev, x_curr]
            x_prev, x_curr = (
                x_curr,
                numpyro.sample("x",
                    dist.Categorical(probs_x_t),
                    infer={'enumerate': "parallel"},
                ),
            )
    with numpyro.plate("tones", data_dim, dim=-1):
        probs_y_t = probs_y[x_curr.squeeze(-1)]
        numpyro.sample("y", dist.Bernoulli(probs_y_t), obs=y)
    return (x_prev, x_curr, t + 1), None

x_prev = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
x_curr = jnp.zeros((num_sequences, 1), dtype=jnp.int32)
scan(transition_fn, (x_prev, x_curr, 0), jnp.swapaxes(sequences, 0, 1), history=2)

Do inference

models = {
    name[len("model_") :]: model
    for name, model in globals().items()
    if name.startswith("model_")
}

def main(args):
    model = models[args.model]
    _, fetch = load_dataset(JSB_CHORALES, split="train", shuffle=False)
    lengths, sequences = fetch()
    if args.num_sequences:
        sequences = sequences[0 : args.num_sequences]
lengths = lengths[0 : args.num_sequences]

logger.info("-" * 40)
logger.info("Training {} on {} sequences".format(model.__name__, len(sequences)))

# find all the notes that are present at least once in the training set
present_notes = (sequences == 1).sum(0).sum(0) > 0
# remove notes that are never played (we remove 37/88 notes with default args)
sequences = sequences[:, :, present_notes]

if args.truncate:
    lengths = lengths.clip(0, args.truncate)
    sequences = sequences[:, : args.truncate]

logger.info("Each sequence has shape {}".format(sequences[0].shape))
logger.info("Starting inference...")
rng_key = random.PRNGKey(2)
start = time.time()
kern...
numpyro.set_host_device_count(args.num_chains)

main(args)
EXAMPLE: CJS CAPTURE-RECAPTURE MODEL FOR ECOLOGICAL DATA

This example is ported from [8].

We show how to implement several variants of the Cormack-Jolly-Seber (CJS) [4, 5, 6] model used in ecology to analyze animal capture-recapture data. For a discussion of these models see reference [1].

We make use of two datasets:

- the European Dipper (Cinclus cinclus) data from reference [2] (this is Norway’s national bird).
- the meadow voles data from reference [3].

Compare to the Stan implementations in [7].

References


7. https://github.com/stan-dev/example-models/tree/master/BPA/Ch.07


```python
import argparse
import os
from jax import random
import jax.numpy as jnp
from jax.scipy.special import expit, logit
import numpyro
from numpyro import handlers
from numpyro.contrib.control_flow import scan
import numpyro.distributions as dist
```

(continues on next page)
Our first and simplest CJS model variant only has two continuous (scalar) latent random variables: i) the survival probability \( \phi \); and ii) the recapture probability \( \rho \). These are treated as fixed effects with no temporal or individual/group variation.

```python
def model_1(capture_history, sex):
    N, T = capture_history.shape
    phi = numpyro.sample("phi", dist.Uniform(0.0, 1.0))  # survival probability
    rho = numpyro.sample("rho", dist.Uniform(0.0, 1.0))  # recapture probability

    def transition_fn(carry, y):
        first_capture_mask, z = carry
        with numpyro.plate("animals", N, dim=-1):
            with handlers.mask(mask=first_capture_mask):
                mu_z_t = first_capture_mask * phi * z + (1 - first_capture_mask)
                z = numpyro.sample("z", dist.Bernoulli(dist.util.clamp_probs(mu_z_t)),
                                  infer={"enumerate": "parallel"},
                                  )
                mu_y_t = rho * z
                numpyro.sample("y", dist.Bernoulli(dist.util.clamp_probs(mu_y_t)), obs=y
        first_capture_mask = first_capture_mask | y.astype(bool)
        return (first_capture_mask, z), None

    z = jnp.ones(N, dtype=jnp.int32)
    # we use this mask to eliminate extraneous log probabilities
    # that arise for a given individual before its first capture.
    first_capture_mask = capture_history[:, 0].astype(bool)
    # NB swapaxes: we move time dimension of `capture_history` to the front to scan over...it
    scan(transition_fn,
          (first_capture_mask, z),
          jnp.swapaxes(capture_history[:, 1:], 0, 1),
          )
```

In our second model variant there is a time-varying survival probability \( \phi_t \) for T-1 of the T time periods of the capture data; each \( \phi_t \) is treated as a fixed effect.

```python
def model_2(capture_history, sex):
    N, T = capture_history.shape
    rho = numpyro.sample("rho", dist.Uniform(0.0, 1.0))  # recapture probability
```

```python
def transition_fn(carry, y):
```

```python
(continues on next page)
```
first_capture_mask, z = carry
# note that phi_t needs to be outside the plate, since
# phi_t is shared across all N individuals
phi_t = numpyro.sample("phi", dist.Uniform(0.0, 1.0))

with numpyro.plate("animals", N, dim=-1):
    with handlers.mask(mask=first_capture_mask):
        mu_z_t = first_capture_mask * phi_t * z + (1 - first_capture_mask)
        z = numpyro.sample("z",
            dist.Bernoulli(dist.util.clamp_probs(mu_z_t)),
            infer={"enumerate": "parallel"},
            )
        mu_y_t = rho * z
        numpyro.sample("y", dist.Bernoulli(dist.util.clamp_probs(mu_y_t)), obs=y)

first_capture_mask = first_capture_mask | y.astype(bool)
return (first_capture_mask, z), None

z = jnp.ones(N, dtype=jnp.int32)
# we use this mask to eliminate extraneous log probabilities
# that arise for a given individual before its first capture.
first_capture_mask = capture_history[:, 0].astype(bool)
# NB swapaxes: we move time dimension of `capture_history` to the front to scan over
transition_fn,
    (first_capture_mask, z),
    jnp.swapaxes(capture_history[:, 1:], 0, 1),
)

In our third model variant there is a survival probability phi_t for T-1 of the T time periods of the capture data (just like in model_2), but here each phi_t is treated as a random effect.

def model_3(capture_history, sex):
    N, T = capture_history.shape
    phi_mean = numpyro.sample("phi_mean", dist.Uniform(0.0, 1.0))
    # mean survival probability
    phi_logit_mean = logit(phi_mean)
    # controls temporal variability of survival probability
    phi_sigma = numpyro.sample("phi_sigma", dist.Uniform(0.0, 10.0))
    rho = numpyro.sample("rho", dist.Uniform(0.0, 1.0))  # recapture probability

def transition_fn(carry, y):
    first_capture_mask, z = carry
    with handlers.reparam(config={"phi_logit": LocScaleReparam(0)}):
        phi_logit_t = numpyro.sample("phi_logit", dist.Normal(phi_logit_mean, phi_sigma))
        # (continues on next page)
phi_t = expit(phi_logit_t)

```python
with numpyro.plate("animals", N, dim=-1):
    with handlers.mask(mask=first_capture_mask):
        mu_z_t = first_capture_mask * phi_t * z + (1 - first_capture_mask)
        # NumPyro exactly sums out the discrete states z_t.
        z = numpyro.sample("z",
            dist.Bernoulli(dist.util.clamp_probs(mu_z_t)),
            infer={"enumerate": "parallel"},
        )
        mu_y_t = rho * z
        numpyro.sample("y", dist.Bernoulli(dist.util.clamp_probs(mu_y_t)), obs=y)
```

```python
first_capture_mask = first_capture_mask | y.astype(bool)
return (first_capture_mask, z) , None
```

In our fourth model variant we include group-level fixed effects for sex (male, female).

```python
def model_4(capture_history, sex):
    N, T = capture_history.shape
    # survival probabilities for males/females
    phi_male = numpyro.sample("phi_male", dist.Uniform(0.0, 1.0))
    phi_female = numpyro.sample("phi_female", dist.Uniform(0.0, 1.0))
    # we construct a N-dimensional vector that contains the appropriate
    # phi for each individual given its sex (female = 0, male = 1)
    phi = sex * phi_male + (1.0 - sex) * phi_female
    rho = numpyro.sample("rho", dist.Uniform(0.0, 1.0)) # recapture probability

def transition_fn(carry, y):
    first_capture_mask, z = carry
    with numpyro.plate("animals", N, dim=-1):
        with handlers.mask(mask=first_capture_mask):
            mu_z_t = first_capture_mask * phi * z + (1 - first_capture_mask)
            # NumPyro exactly sums out the discrete states z_t.
            z = numpyro.sample("z",
                dist.Bernoulli(dist.util.clamp_probs(mu_z_t)),
            )
            mu_y_t = rho * z
            numpyro.sample("y", dist.Bernoulli(dist.util.clamp_probs(mu_y_t)), obs=y)
    return (first_capture_mask, z) , None
```

```python
z = jnp.ones(N, dtype=jnp.int32)
# we use this mask to eliminate extraneous log probabilities
# that arise for a given individual before its first capture.
first_capture_mask = capture_history[:, 0].astype(bool)
# NB swapaxes: we move time dimension of 'capture_history' to the front to scan over it
scan(transition_fn,
    (first_capture_mask, z),
    jnp.swapaxes(capture_history[:, 1:], 0, 1),
)
```
In our final model variant we include both fixed group effects and fixed time effects for the survival probability $\phi$:

$$\text{logit}(\phi_t) = \beta_{\text{group}} + \gamma_t$$

We need to take care that the model is not overparameterized; to do this we effectively let a single scalar $\beta$ encode the difference in male and female survival probabilities.

```python
def model_5(capture_history, sex):
    N, T = capture_history.shape
    # $\phi_{\beta}$ controls the survival probability differential
    # for males versus females (in logit space)
    phi_beta = numpyro.sample("phi_beta", dist.Normal(0.0, 10.0))
    phi_beta = sex * phi_beta
    rho = numpyro.sample("rho", dist.Uniform(0.0, 1.0))  # recapture probability

    def transition_fn(carry, y):
        first_capture_mask, z = carry
        phi_gamma_t = numpyro.sample("phi_gamma", dist.Normal(0.0, 10.0))
        phi_t = expit(phi_beta + phi_gamma_t)
        with numpyro.plate("animals", N, dim=-1):
            with handlers.mask(mask=first_capture_mask):
                mu_z_t = first_capture_mask * phi_t * z + (1 - first_capture_mask)
                # NumPyro exactly sums out the discrete states $z_t$.
                z = numpyro.sample("z",
                    dist.Bernoulli(dist.util.clamp_probs(mu_z_t)),
                    infer={"enumerate": "parallel"},
                )
                mu_y_t = rho * z
                numpyro.sample("y",
                    dist.Bernoulli(dist.util.clamp_probs(mu_y_t)), obs=y
                )
```

(continues on next page)
```python
first_capture_mask = first_capture_mask | y.astype(bool)
return (first_capture_mask, z), None

z = jnp.ones(N, dtype=jnp.int32)
# we use this mask to eliminate extraneous log probabilities
# that arise for a given individual before its first capture.
first_capture_mask = capture_history[:, 0].astype(bool)
# NB swapaxes: we move time dimension of `capture_history` to the front to scan over it
scan(transition_fn,
    (first_capture_mask, z),
    jnp.swapaxes(capture_history[:, 1:], 0, 1),
)

Do inference

models = {
    name[len("model_") :]: model
    for name, model in globals().items()
    if name.startswith("model_")
}

def run_inference(model, capture_history, sex, rng_key, args):
    if args.algo == "NUTS":
        kernel = NUTS(model)
    elif args.algo == "HMC":
        kernel = HMC(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, capture_history, sex)
    mcmc.print_summary()
    return mcmc.get_samples()

def main(args):
    # load data
    if args.dataset == "dipper":
        capture_history, sex = load_dataset(DIPPER_VOLE, split="dipper", shuffle=False)[
        1
    ]()
    elif args.dataset == "vole":
        if args.model in ["4", "5"]:
            raise ValueError(
                "Cannot run model_{} on meadow voles data, since we lack sex ")
```

(continues on next page)
"information for these animals.".format(args.model)

    (capture_history,) = load_dataset(DIPPER_VOLE, split="vole", shuffle=False)[1]()
    sex = None
else:
        raise ValueError("Available datasets are 'dipper' and 'vole'.")

    N, T = capture_history.shape
    print(
        "Loaded {} capture history for {} individuals collected over {} time periods.".
        "{}"
    ,
        args.dataset, N, T
    )

    model = models[args.model]
    rng_key = random.PRNGKey(args.rng_seed)
    run_inference(model, capture_history, sex, rng_key, args)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="CJS capture-recapture model for ecological data"
    )
    parser.add_argument(
        "-m", "--model",
        default="1",
        type=str,
        help="one of: {}.format("", .join(sorted(models.keys()))),
    )
    parser.add_argument("-d", "--dataset", default="dipper", type=str)
    parser.add_argument("-n", "--num-samples", narg="?", default=1000, type=int)
    parser.add_argument("--num-warmup", narg="?", default=1000, type=int)
    parser.add_argument("--num-chains", narg="?", default=1, type=int)
    parser.add_argument(
        "--rng_seed", default=0, type=int, help="random number generator seed"
    )
    parser.add_argument(
        "--algo", default="NUTS", type=str, help='whether to run "NUTS" or "HMC"
    )
    args = parser.parse_args()
    main(args)
EXAMPLE: NESTED SAMPLING FOR GAUSSIAN SHELLS

This example illustrates the usage of the contrib class NestedSampler, which is a wrapper of jaxns library ([1]) to be used for NumPyro models.

Here we will replicate the Gaussian Shells demo at [2] and compare against NUTS sampler.

References:
1. jaxns library: https://github.com/Joshuaalbert/jaxns

```python
import argparse
import matplotlib.pyplot as plt

from jax import random
import jax.numpy as jnp

import numpyro
from numpyro.contrib.nested_sampling import NestedSampler
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, DiscreteHMCGibbs

class GaussianShell(dist.Distribution):
    support = dist.constraints.real_vector

    def __init__(self, loc, radius, width):
        self.loc, self.radius, self.width = loc, radius, width
        super().__init__(batch_shape=loc.shape[:-1], event_shape=loc.shape[-1:]

    def sample(self, key, sample_shape=()):
        return jnp.zeros(sample_shape + self.shape())

    def log_prob(self, value):
        normalizer = (-0.5) * (jnp.log(2.0 * jnp.pi) + 2.0 * jnp.log(self.width))
        d = jnp.linalg.norm(value - self.loc, axis=-1)
        return normalizer - 0.5 * ((d - self.radius) / self.width) ** 2
```

(continues on next page)
def model(center1, center2, radius, width, enum=False):
    z = numpyro.sample("z", dist.Bernoulli(0.5), infer={"enumerate": "parallel"} if enum else {})
    x = numpyro.sample("x", dist.Uniform(-6.0, 6.0).expand([2]).to_event(1))
    center = jnp.stack([center1, center2])[z]
    numpyro.sample("shell", GaussianShell(center, radius, width), obs=x)

def run_inference(args, data):
    print("=== Performing Nested Sampling ===")
    ns = NestedSampler(model)
    ns.run(random.PRNGKey(0), **data, enum=args.enum)
    ns.print_summary()
    # samples obtained from nested sampler are weighted, so
    # we need to provide random key to resample from those weighted samples
    ns_samples = ns.get_samples(random.PRNGKey(1), num_samples=args.num_samples)
    print("\n
    === Performing MCMC Sampling ===")
    if args.enum:
        mcmc = MCMC(NUTS(model), num_warmup=args.num_warmup, num_samples=args.num_samples)
    else:
        mcmc = MCMC(DiscreteHMCGibbs(NUTS(model)),
                     num_warmup=args.num_warmup,
                     num_samples=args.num_samples,
                     )
    mcmc.run(random.PRNGKey(2), **data, enum=args.enum)
    mcmc.print_summary()
    mcmc_samples = mcmc.get_samples()
    return ns_samples["x"], mcmc_samples["x"]

def main(args):
    data = dict(radius=2.0,
                 width=0.1,
                 center1=jnp.array([-3.5, 0.0]),
                 center2=jnp.array([3.5, 0.0]),
                 )
    ns_samples, mcmc_samples = run_inference(args, data)
    # plotting
    fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(8, 8), constrained_layout=True)
    ax1.plot(mcmc_samples[:, 0], mcmc_samples[:, 1], "ro", alpha=0.2)
    ax1.set
```python
xlim=(-6, 6),
ylim=(-2.5, 2.5),
ylabel="x[1]",
title="Gaussian-shell samples using NUTS",
)

ax2.plot(ns_samples[:, 0], ns_samples[:, 1], "ro", alpha=0.2)
ax2.set(
    xlim=(-6, 6),
    ylim=(-2.5, 2.5),
    xlabel="x[0]",
    ylabel="x[1]",
    title="Gaussian-shell samples using Nested Sampler",
)

plt.savefig("gaussian_shells_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Nested sampler for Gaussian shells")
    parser.add_argument("-n", "--num-samples", nargs="?", default=10000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
    parser.add_argument("--enum", action="store_true", default=False,
                       help="whether to enumerate over the discrete latent variable",
    )
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)

    main(args)
```
MISSING VALUES IN DISCRETE COVARIATES

Missing data is a very widespread problem in practical applications, both in covariates ('explanatory variables') and outcomes. When performing bayesian inference with MCMC, imputing discrete missing values is not possible using Hamiltonian Monte Carlo techniques. One way around this problem is to create a new model that enumerates the discrete variables and does inference over the new model, which, for a single discrete variable, is a mixture model. (see e.g. Stan's user guide on Latent Discrete Parameters) Enumerating the discrete latent sites requires some manual math work that can get tedious for complex models. Inference by automatic enumeration of discrete variables is implemented in numpyro and allows for a very convenient way of dealing with missing discrete data.

```
[ ]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro funsor
```

```
[1]: from math import inf
    from graphviz import Digraph
    from jax import numpy as jnp, random
    from jax.scipy.special import expit

import numpyro
from numpyro import distributions as dist, sample
from numpyro.infer.hmc import NUTS
from numpyro.infer.mcmc import MCMC

simkeys = random.split(random.PRNGKey(0), 10)
nsim = 5000
mcmc_key = random.PRNGKey(1)
```

First we will simulate data with correlated binary covariates. The assumption is that we wish to estimate parameter for some parametric model without bias (e.g. for inferring a causal effect). For several different missing data patterns we will see how to impute the values to lead to unbiased models.

The basic data structure is as follows. Z is a latent variable that gives rise to the marginal dependence between A and B, the observed covariates. We will consider different missing data mechanisms for variable A, where variable B and Y are fully observed. The effects of A and B on Y are the effects of interest.

```
[2]: dot = Digraph()
    dot.node("A")
    dot.node("B")
    dot.node("Z")
    dot.node("Y")
```

(continues on next page)
dot.edges(["ZA", "ZB", "AY", "BY"])
According to Rubin’s classic definitions there are 3 distinct of missing data mechanisms:

1. missing completely at random (MCAR)
2. missing at random, conditional on observed data (MAR)
3. missing not at random, even after conditioning on observed data (MNAR)

Missing data mechanisms 1. and 2. are ‘easy’ to handle as they depend on observed data only. Mechanism 3. (MNAR) is trickier as it depends on data that is not observed, but may still be relevant to the outcome you are modeling (see below for a concrete example).

First we will generate missing values in A, conditional on the value of Y (thus it is a MAR mechanism).
This graph depicts the datagenerating mechanism, where Y is the only cause of missingness in A, denoted M. This means that the missingness in M is random, conditional on Y.

As an example consider this simplified scenario:

- A represents a history of heart illness
- B represents the age of a patient
- Y represents whether or not the patient will visit the general practitioner

A general practitioner wants to find out why patients that are assigned to her clinic will visit the clinic or not. She thinks
that having a history of heart illness and age are potential causes of doctor visits. Data on patient ages are available through their registration forms, but information on prior heart illness may be available only after they have visited the clinic. This makes the missingness in \(A\) (history of heart disease), dependent on the outcome (visiting the clinic).

\[
\text{A}\_\text{isobs} = \text{random.bernoulli}(\text{simkeys}[4], \expit(3 * (Y - Y.\text{mean}())))
\]

\[
A\_\text{obs} = \text{jnp.where}(A\_\text{isobs}, A, -1)
\]

\[
A\_\text{obsidx} = \text{jnp.where}(A\_\text{isobs})
\]

# generate complete case arrays
Acc = A\_\text{obs}[A\_\text{obsidx}]
Bcc = B[A\_\text{obsidx}]
Ycc = Y[A\_\text{obsidx}]

We will evaluate 2 approaches:

1. complete case analysis (which will lead to biased inferences)
2. with imputation (conditional on B)

Note that explicitly including \(Y\) in the imputation model for \(A\) is unnecessary. The sampled imputations for \(A\) will condition on \(Y\) indirectly as the likelihood of \(Y\) is conditional on \(A\). So values of \(A\) that give high likelihood to \(Y\) will be sampled more often than other values.

```
[6]: def ccmodel(A, B, Y):
    ntotal = A.shape[0]
    # get parameters of outcome model
    b_A = sample("b_A", dist.Normal(0, 2.5))
    b_B = sample("b_B", dist.Normal(0, 2.5))
    s_Y = sample("s_Y", dist.HalfCauchy(2.5))

    with numpyro.plate("obs", ntotal):
        ### outcome model
        eta_Y = b_A * A + b_B * B
        sample("obs_Y", dist.Normal(eta_Y, s_Y), obs=Y)
```

```
[7]: cckernel = NUTS(ccmodel)
ccmcmc = MCMC(cckernel, num_warmup=250, num_samples=750)
ccmcmc.run(mcmc_key, Acc, Bcc, Ycc)
ccmcmc.print_summary()
```

```
<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_A</td>
<td>0.30</td>
<td>0.01</td>
<td>0.30</td>
<td>0.29</td>
<td>0.31</td>
<td>500.83</td>
<td>1.00</td>
</tr>
<tr>
<td>b_B</td>
<td>0.28</td>
<td>0.01</td>
<td>0.28</td>
<td>0.27</td>
<td>0.29</td>
<td>546.34</td>
<td>1.00</td>
</tr>
<tr>
<td>s_Y</td>
<td>0.25</td>
<td>0.00</td>
<td>0.25</td>
<td>0.24</td>
<td>0.25</td>
<td>559.55</td>
<td>1.00</td>
</tr>
</tbody>
</table>
```

Number of divergences: 0

```
[8]: def impmodel(A, B, Y):
    ntotal = A.shape[0]
    A_isobs = A >= 0
```

(continues on next page)
# get parameters of imputation model
mu_A = sample("mu_A", dist.Normal(0, 2.5))
b_B_A = sample("b_B_A", dist.Normal(0, 2.5))

# get parameters of outcome model
b_A = sample("b_A", dist.Normal(0, 2.5))
b_B = sample("b_B", dist.Normal(0, 2.5))
s_Y = sample("s_Y", dist.HalfCauchy(2.5))

with numpyro.plate("obs", ntotal):
    ### imputation model
    # get linear predictor for missing values
    eta_A = mu_A + B * b_B_A
    # sample imputation values for A
    # mask out to not add log_prob to total likelihood right now
    Aimp = sample("A",
        dist.Bernoulli(logits=eta_A).mask(False),
        infer={"enumerate": "parallel"},
    )

    # 'manually' calculate the log_prob
    log_prob = dist.Bernoulli(logits=eta_A).log_prob(Aimp)
    # cancel out enumerated values that are not equal to observed values
    log_prob = jnp.where(A_isobs & (Aimp != A), -inf, log_prob)
    # add to total likelihood for sampler
    numpyro.factor("A_obs", log_prob)

    ### outcome model
    eta_Y = b_A * Aimp + b_B * B
    sample("obs_Y", dist.Normal(eta_Y, s_Y), obs=Y)

[9]: impkernel = NUTS(imppmodel)
impmcmc = MCMC(impkernel, num_warmup=250, num_samples=750)
impmcmc.run(mcmc_key, Aobs, B, Y)
impmcmc.print_summary()

sample: 100%| 1000/1000 [00:05<00:00, 174.83it/s, 7 steps of␣size 4.41e-01. acc. prob=0.91]

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_A</td>
<td>0.25</td>
<td>0.01</td>
<td>0.25</td>
<td>0.24</td>
<td>0.27</td>
<td>447.79</td>
<td>1.01</td>
</tr>
<tr>
<td>b_B</td>
<td>0.25</td>
<td>0.01</td>
<td>0.25</td>
<td>0.24</td>
<td>0.26</td>
<td>570.66</td>
<td>1.01</td>
</tr>
<tr>
<td>b_B_A</td>
<td>0.74</td>
<td>0.08</td>
<td>0.74</td>
<td>0.60</td>
<td>0.86</td>
<td>316.36</td>
<td>1.00</td>
</tr>
<tr>
<td>mu_A</td>
<td>-0.39</td>
<td>0.06</td>
<td>-0.39</td>
<td>-0.48</td>
<td>-0.29</td>
<td>290.86</td>
<td>1.00</td>
</tr>
<tr>
<td>s_Y</td>
<td>0.25</td>
<td>0.00</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>527.97</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 0
As we can see, when data are missing conditionally on Y, imputation leads to consistent estimation of the parameter of interest (b_A and b_B).

### 26.2 MNAR conditional on covariate

When data are missing conditional on unobserved data, things get more tricky. Here we will generate missing values in A, conditional on the value of A itself (missing not at random (MNAR), but missing at random conditional on A).

As an example consider patients who have cancer:

- A represents weight loss
- B represents age
- Y represents overall survival time

Both A and B can be related to survival time in cancer patients. For patients who have extreme weight loss, it is more likely that this will be noted by the doctor and registered in the electronic health record. For patients with no weight loss or little weight loss, it may be that the doctor forgets to ask about it and therefore does not register it in the records.

```python
[10]: dot_mnar_x = Digraph()
    with dot_mnar_y.subgraph() as s:
        s.attr(rank="same")
        s.node("A")
        s.node("M")
    dot_mnar_x.node("B")
    dot_mnar_x.node("Z")
    dot_mnar_x.node("Y")
    dot_mnar_x.edges(["AM", "ZA", "ZB", "AY", "BY"])
    dot_mnar_x
```
26.2. MNAR conditional on covariate
A_isobs = random.bernoulli(simkeys[5], 0.9 - 0.8 * A)
Aobs = jnp.where(A_isobs, A, -1)
A_obsidx = jnp.where(A_isobs)

# generate complete case arrays
Acc = Aobs[A_obsidx]
Bcc = B[A_obsidx]
Ycc = Y[A_obsidx]

cckernel = NUTS(ccmodel)
ccmcmc = MCMC(cckernel, num_warmup=250, num_samples=750)
ccmcmc.run(mcmc_key, Acc, Bcc, Ycc)
ccmcmc.print_summary()

sample: 100%| 1000/1000 [00:02<00:00, 342.07it/s, 3 steps of␣ size 5.97e-01. acc. prob=0.92]

mean std  median  5.0%  95.0%  n_eff  r_hat
b_A  0.27 0.02  0.26   0.24   0.29  667.08  1.01
b_B  0.25 0.01  0.25   0.24   0.26  811.49  1.00
s_Y  0.25 0.00  0.25   0.24   0.25  547.51  1.00

Number of divergences: 0

impkernel = NUTS(imppmodel)
imppmcmc = MCMC(impkernel, num_warmup=250, num_samples=750)
imppmcmc.run(mcmc_key, Aobs, B, Y)
imppmcmc.print_summary()

sample: 100%| 1000/1000 [00:06<00:00, 166.36it/s, 7 steps of␣ size 4.10e-01. acc. prob=0.94]

mean std  median  5.0%  95.0%  n_eff  r_hat
b_A  0.34 0.01  0.34   0.32   0.35  576.15  1.00
b_B  0.33 0.01  0.33   0.32   0.34  800.58  1.00
b_B_A  0.32 0.12  0.32   0.12   0.51  342.21  1.01
mu_A  -1.81 0.09  -1.81  -1.95  -1.67  288.57  1.00
s_Y  0.26 0.00  0.26   0.25   0.26  820.20  1.00

Number of divergences: 0

Perhaps surprisingly, imputing missing values when the missingness mechanism depends on the variable itself will actually lead to bias, while complete case analysis is unbiased! See e.g. Bias and efficiency of multiple imputation compared with complete-case analysis for missing covariate values.

However, complete case analysis may be undesirable as well. E.g. due to leading to lower precision in estimating the parameter from B to Y, or maybe when there is an expected difference interaction between the value of A and the parameter from A to Y. To deal with this situation, an explicit model for the reason of missingness (/observation) is required. We will add one below.
def impmissmodel(A, B, Y):
    ntotal = A.shape[0]
    A_isobs = A >= 0

    # get parameters of imputation model
    mu_A = sample("mu_A", dist.Normal(0, 2.5))
    b_B_A = sample("b_B_A", dist.Normal(0, 2.5))

    # get parameters of outcome model
    b_A = sample("b_A", dist.Normal(0, 2.5))
    b_B = sample("b_B", dist.Normal(0, 2.5))
    s_Y = sample("s_Y", dist.HalfCauchy(2.5))

    # get parameter of model of missingness
    with numpyro.plate("obsmodel", 2):
        p_Aobs = sample("p_Aobs", dist.Beta(1, 1))

    with numpyro.plate("obs", ntotal):
        ### imputation model
        # get linear predictor for missing values
        eta_A = mu_A + B * b_B_A

        # sample imputation values for A
        # mask out to not add log_prob to total likelihood right now
        Aimp = sample("A",
                      dist.Bernoulli(logits=eta_A).mask(False),
                      infer="enumerate": "parallel"),
                  )

        # 'manually' calculate the log_prob
        log_prob = dist.Bernoulli(logits=eta_A).log_prob(Aimp)

        # cancel out enumerated values that are not equal to observed values
        log_prob = jnp.where(A_isobs & (Aimp != A), -inf, log_prob)

        # add to total likelihood for sampler
        numpyro.factor("obs_A", log_prob)

        ### outcome model
        eta_Y = b_A * Aimp + b_B * B
        sample("obs_Y", dist.Normal(eta_Y, s_Y), obs=Y)

        ### missingness / observationmodel
        eta_Aobs = jnp.where(Aimp, p_Aobs[0], p_Aobs[1])
        sample("obs_Aobs", dist.Bernoulli(probs=eta_Aobs), obs=A_isobs)

impmisskernel = NUTS(impmissmodel)
impmissmcmc = MCMC(impmisskernel, num_warmup=250, num_samples=750)
impmissmcmc.run(mcmc_key, Aobs, B, Y)
impmissmcmc.print_summary()
We can now estimate the parameters $b_A$ and $b_B$ without bias, while still utilizing all observations. Obviously, modeling the missingness mechanism relies on assumptions that need either be substantiated with prior evidence, or possibly analyzed through sensitivity analysis.

For more reading on missing data in bayesian inference, see:

- Presentation Bayesian Methods for missing data (pdf)
- Bayesian Approaches for Missing Not at Random Outcome Data: The Role of Identifying Restrictions (doi:10.1214/17-STS630)
In this tutorial, we will demonstrate how to build a model for time series forecasting in NumPyro. Specifically, we will replicate the Seasonal, Global Trend (SGT) model from the Rlgt: Bayesian Exponential Smoothing Models with Trend Modifications package. The time series data that we will use for this tutorial is the lynx dataset, which contains annual numbers of lynx trappings from 1821 to 1934 in Canada.

```python
[ ]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[1]: import os
from IPython.display import set_matplotlib_formats
import matplotlib.pyplot as plt
import pandas as pd
from jax import random
import jax.numpy as jnp
import numpyro
from numpyro.contrib.control_flow import scan
from numpyro.diagnostics import autocorrelation, hpdi
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive

if "NUMPYRO_SPHINXBUILD" in os.environ:
    set_matplotlib_formats("svg")

numpyro.set_host_device_count(4)
assert numpyro.__version__.startswith("0.15.2")

27.1 Data

First, let's import and take a look at the dataset.

```python
[2]: URL = "https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/...datasets/lynx.csv"
lynx = pd.read_csv(URL, index_col=0)
data = lynx["value"].values
print("Length of time series:", data.shape[0])
plt.figure(figsize=(8, 4))
(continues on next page)"
plt.plot(lynx["time"], data)
plt.show()

Length of time series: 114

The time series has a length of 114 (a data point for each year), and by looking at the plot, we can observe seasonality in this dataset, which is the recurrence of similar patterns at specific time periods. e.g. in this dataset, we observe a cyclical pattern every 10 years, but there is also a less obvious but clear spike in the number of trappings every 40 years. Let us see if we can model this effect in NumPyro.

In this tutorial, we will use the first 80 values for training and the last 34 values for testing.

```python
[3]: y_train, y_test = jnp.array(data[:80], dtype=jnp.float32), data[80:]
```

## 27.2 Model

The model we are going to use is called **Seasonal, Global Trend**, which when tested on 3003 time series of the M-3 competition, has been known to outperform other models originally participating in the competition:

\[
\begin{align*}
\text{exp-val}_t &= \text{level}_{t-1} + \text{coef-trend} \times \text{level}_{t-1}^{\text{pow-trend}} + s_t \times \text{level}_{t-1}^{\text{pow-season}}, \\
\sigma_t &= \sigma \times \text{exp-val}_{t-1}^{\text{powx}} + \text{offset}, \\
y_t &\sim \text{StudentT}(\nu, \text{exp-val}_t, \sigma_t)
\end{align*}
\]

where \(\text{level}\) and \(s\) follows the following recursion rules:

\[
\begin{align*}
\text{level-p} &= \begin{cases} \\
\text{level}_{t-1}^{\text{pow-season}} - s_t \times \text{level}_{t-1}^{\text{pow-season}} \\
\text{Average}[y(t - \text{seasonality} + 1), \ldots, y(t)]
\end{cases} & \text{if } t \leq \text{seasonality}, \\
\text{level}_t &= \text{level-sm} \times \text{level-p} + (1 - \text{level-sm}) \times \text{level}_{t-1}, \\
\text{s}^{t+\text{seasonality}} &= \text{s-sm} \times \frac{y_t - \text{level}_t}{\text{level}_{t-1}^{\text{pow-trend}}} + (1 - \text{s-sm}) \times s_t
\end{align*}
\]
A more detailed explanation for SGT model can be found in this vignette from the authors of the Rlgt package. Here we summarize the core ideas of this model:

- **Student’s t-distribution**, which has heavier tails than normal distribution, is used for the likelihood.

- The expected value \( \text{exp_val} \) consists of a trending component and a seasonal component:
  
  - The trend is governed by the map \( x \mapsto x + ax^b \), where \( x \) is level, \( a \) is \( \text{coef\_trend} \), and \( b \) is \( \text{pow\_trend} \). Note that when \( b \approx 0 \), the trend is linear with \( a \) is the slope, and when \( b \approx 1 \), the trend is exponential with \( a \) is the rate. So that function can cover a large family of trend.

  - When time changes, level and \( s \) are updated to new values. Coefficients \( \text{level\_sm} \) and \( \text{s\_sm} \) are used to make the transition smoothly.

- When \( \text{powx} \) is near 0, the error \( \sigma \) will be nearly constant while when \( \text{powx} \) is near 1, the error will be proportional to the expected value.

- There are several varieties of SGT. In this tutorial, we use generalized seasonality and seasonal average method.

We are ready to specify the model using \textit{NumPyro} primitives. In NumPyro, we use the primitive \texttt{sample(name, prior)} to declare a latent random variable with a corresponding prior. These primitives can have custom interpretations depending on the effect handlers that are used by NumPyro inference algorithms in the backend. e.g. we can condition on specific values using the condition handler, or record values at these sample sites in the execution trace using the trace handler. Note that these details are not important for specifying the model, or running inference, but curious readers are encouraged to read the tutorial on effect handlers in Pyro.

```python
[4]: def sgt(y, seasonality, future=0):
    # heuristically, standard derivation of Cauchy prior depends on
    # the max value of data
    cauchy_sd = jnp.max(y) / 150

    # NB: priors' parameters are taken from
    nu = numpyro.sample("nu", dist.Uniform(2, 20))
    powx = numpyro.sample("powx", dist.Uniform(0, 1))
    sigma = numpyro.sample("sigma", dist.HalfCauchy(cauchy_sd))
    offset_sigma = numpyro.sample(
        "offset_sigma", dist.TruncatedCauchy(low=1e-10, loc=1e-10, scale=cauchy_sd)
    )

    coef_trend = numpyro.sample("coef_trend", dist.Cauchy(0, cauchy_sd))
    pow_trend_beta = numpyro.sample("pow_trend_beta", dist.Beta(1, 1))
    # pow_trend takes values from -0.5 to 1
    pow_trend = 1.5 * pow_trend_beta - 0.5
    pow_season = numpyro.sample("pow_season", dist.Beta(1, 1))

    level_sm = numpyro.sample("level_sm", dist.Beta(1, 2))
    s_sm = numpyro.sample("s_sm", dist.Uniform(0, 1))
    init_s = numpyro.sample("init_s", dist.Cauchy(0, y[:seasonality] * 0.3))

    def transition_fn(carry, t):
        level, s, moving_sum = carry
        season = s[0] * level**pow_season
        exp_val = level + coef_trend * level**pow_trend + season
        exp_val = jnp.clip(exp_val, 0)
        # use expected value when forecasting
        y_t = jnp.where(t >= N, exp_val, y[t])
```
moving_sum = 
    moving_sum + y[t] - jnp.where(t >= seasonality, y[t - seasonality], 0.0)

level_p = jnp.where(t >= seasonality, moving_sum / seasonality, y_t - season)
level = level_sm * level_p + (1 - level_sm) * level
level = jnp.clip(level, 0)

new_s = (s_sm * (y_t - level) / season + (1 - s_sm)) * s[0]
# repeat s when forecasting
new_s = jnp.where(t >= N, s[0], new_s)
s = jnp.concatenate([s[1:], new_s[None]], axis=0)

omega = sigma * exp_val**powx + offset_sigma
y_ = numpyro.sample("y", dist.StudentT(nu, exp_val, omega))

return (level, s, moving_sum), y_

N = y.shape[0]
level_init = y[0]
s_init = jnp.concatenate([init_s[1:], init_s[:1]], axis=0)
moving_sum = level_init

with numpyro.handlers.condition(data={"y": y[1:]})
    _, ys = scan(transition_fn, (level_init, s_init, moving_sum), jnp.arange(1, N + future))

if future > 0:
    numpyro.deterministic("y_forecast", ys[-future:])

Note that level and s are updated recursively while we collect the expected value at each time step. NumPyro uses JAX in the backend to JIT compile many critical parts of the NUTS algorithm, including the verlet integrator and the tree building process. However, doing so using Python's for loop in the model will result in a long compilation time for the model, so we use scan - which is a wrapper of lax.scan with supports for NumPyro primitives and handlers. A detailed explanation for using this utility can be found in NumPyro documentation. Here we use it to collect y values while the triple (level, s, moving_sum) plays the role of carrying state.

Another note is that instead of declaring the observation site y in transition_fn

```
numpyro.sample("y", dist.StudentT(nu, exp_val, omega), obs=y[t])
```

we have used condition handler here. The reason is we also want to use this model for forecasting. In forecasting, future values of y are non-observable, so obs=y[t] does not make sense when t >= len(y) (caution: index out-of-bound errors do not get raised in JAX, e.g. jnp.arange(3)[10] == 2). Using condition, when the length of scan is larger than the length of the conditioned/observed site, unobserved values will be sampled from the distribution of that site.
27.3 Inference

First, we want to choose a good value for `seasonality`. Following the demo in Rlgt, we will set `seasonality=38`. Indeed, this value can be guessed by looking at the plot of the training data, where the second order seasonality effect has a periodicity around 40 years. Note that 38 is also one of the highest-autocorrelation lags.

```
[5]: print("Lag values sorted according to their autocorrelation values:

print(jnp.argsort(autocorrelation(y_train))[:,::-1])
```

```
Lag values sorted according to their autocorrelation values:

[ 0 67 57 38 68 1 29 58 37 56 28 10 19 39 66 78 47 77 9 79 48 76 30 18
 20 11 46 59 69 27 55 36 2 8 40 49 17 21 75 12 65 45 31 26 7 54 35 41
 50 3 22 60 70 16 44 32 23 43 51 4 15 14 34 24 5 52 73 64 33 71 72
 61 63 62]
```

Now, let us run 4 MCMC chains (using the No-U-Turn Sampler algorithm) with 5000 warmup steps and 5000 sampling steps per each chain. The returned value will be a collection of 20000 samples.

```
[6]: %%time
    kernel = NUTS(sgt)
    mcmc = MCMC(kernel, num_warmup=5000, num_samples=5000, num_chains=4)
    mcmc.run(random.PRNGKey(0), y_train, seasonality=38)
    mcmc.print_summary()
    samples = mcmc.get_samples()
```

```

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>coef_trend</td>
<td>32.33</td>
<td>123.99</td>
<td>12.07</td>
<td>-91.43</td>
<td>157.37</td>
<td>1307.74</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[0]</td>
<td>84.35</td>
<td>105.16</td>
<td>61.08</td>
<td>-59.71</td>
<td>232.37</td>
<td>4053.35</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[1]</td>
<td>-21.48</td>
<td>72.05</td>
<td>-26.11</td>
<td>-130.13</td>
<td>94.34</td>
<td>1038.51</td>
<td>1.01</td>
</tr>
<tr>
<td>init_s[2]</td>
<td>26.08</td>
<td>92.13</td>
<td>13.57</td>
<td>-114.83</td>
<td>156.16</td>
<td>1559.02</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[3]</td>
<td>122.52</td>
<td>123.56</td>
<td>102.67</td>
<td>-59.39</td>
<td>305.43</td>
<td>4317.17</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[4]</td>
<td>443.91</td>
<td>254.12</td>
<td>395.89</td>
<td>69.08</td>
<td>789.27</td>
<td>3090.34</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[5]</td>
<td>1163.56</td>
<td>491.37</td>
<td>1079.23</td>
<td>481.92</td>
<td>1861.90</td>
<td>1562.40</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[6]</td>
<td>1968.70</td>
<td>649.68</td>
<td>1860.94</td>
<td>902.00</td>
<td>2910.49</td>
<td>1974.42</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[7]</td>
<td>3652.34</td>
<td>1107.27</td>
<td>3505.37</td>
<td>1967.67</td>
<td>5383.26</td>
<td>1669.91</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[8]</td>
<td>2593.04</td>
<td>831.42</td>
<td>2452.27</td>
<td>1317.67</td>
<td>3858.55</td>
<td>1805.87</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[9]</td>
<td>947.28</td>
<td>422.29</td>
<td>885.72</td>
<td>311.39</td>
<td>1589.56</td>
<td>3355.27</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[10]</td>
<td>44.09</td>
<td>102.92</td>
<td>28.38</td>
<td>-105.25</td>
<td>283.73</td>
<td>1367.99</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[11]</td>
<td>-2.25</td>
<td>52.92</td>
<td>-2.71</td>
<td>-86.51</td>
<td>72.90</td>
<td>611.35</td>
<td>1.01</td>
</tr>
<tr>
<td>init_s[12]</td>
<td>-12.22</td>
<td>64.98</td>
<td>-13.67</td>
<td>-110.07</td>
<td>85.65</td>
<td>892.13</td>
<td>1.01</td>
</tr>
<tr>
<td>init_s[13]</td>
<td>74.43</td>
<td>106.48</td>
<td>53.13</td>
<td>-79.73</td>
<td>225.92</td>
<td>658.08</td>
<td>1.01</td>
</tr>
<tr>
<td>init_s[14]</td>
<td>332.98</td>
<td>255.28</td>
<td>281.72</td>
<td>-11.18</td>
<td>697.96</td>
<td>3685.55</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[15]</td>
<td>965.80</td>
<td>389.00</td>
<td>893.29</td>
<td>373.98</td>
<td>1521.59</td>
<td>2575.80</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[16]</td>
<td>1261.12</td>
<td>469.99</td>
<td>1191.83</td>
<td>557.98</td>
<td>1937.38</td>
<td>2300.48</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[17]</td>
<td>1372.34</td>
<td>559.14</td>
<td>1274.21</td>
<td>483.96</td>
<td>2151.94</td>
<td>2007.79</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[18]</td>
<td>611.20</td>
<td>313.13</td>
<td>546.56</td>
<td>167.97</td>
<td>1087.74</td>
<td>2854.06</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[19]</td>
<td>17.81</td>
<td>87.79</td>
<td>8.93</td>
<td>-118.64</td>
<td>143.96</td>
<td>5689.95</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[20]</td>
<td>-31.84</td>
<td>66.70</td>
<td>-23.15</td>
<td>-146.89</td>
<td>58.97</td>
<td>3083.09</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[21]</td>
<td>-14.01</td>
<td>44.74</td>
<td>-5.80</td>
<td>-86.03</td>
<td>42.99</td>
<td>2118.09</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[22]</td>
<td>-2.26</td>
<td>42.99</td>
<td>-2.39</td>
<td>-61.40</td>
<td>66.34</td>
<td>3022.51</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[23]</td>
<td>43.53</td>
<td>90.60</td>
<td>29.14</td>
<td>-82.56</td>
<td>167.89</td>
<td>3230.17</td>
<td>1.00</td>
</tr>
<tr>
<td>init_s[24]</td>
<td>509.69</td>
<td>326.73</td>
<td>453.22</td>
<td>44.04</td>
<td>975.15</td>
<td>2087.02</td>
<td>1.00</td>
</tr>
</tbody>
</table>
```

(continues on next page)
27.4 Forecasting

Given samples from mcmc, we want to do forecasting for the testing dataset y_test. NumPyro provides a convenient utility Predictive to get predictive distribution. Let’s see how to use it to get forecasting values.

Notice that in the sgt model defined above, there is a keyword future which controls the execution of the model - depending on whether future > 0 or future == 0. The following code predicts the last 34 values from the original time-series.

```python
predictive = Predictive(sgt, samples, return_sites=['y_forecast'])
forecast_marginal = predictive(random.PRNGKey(1), y_train, seasonality=38, future=34)['y_forecast']
```

Let’s get sMAPE, root mean square error of the prediction, and visualize the result with the mean prediction and the 90% highest posterior density interval (HPDI).

```python
y_pred = jnp.mean(forecast_marginal, axis=0)
sMAPE = jnp.mean(jnp.abs(y_pred - y_test) / (y_pred + y_test)) * 200
msqrt = jnp.sqrt(jnp.mean((y_pred - y_test) ** 2))
print(f"sMAPE: {:.2f}, rmse: {:.2f}".format(sMAPE, msqrt))
```

```
sMAPE: 63.93, rmse: 1249.29
```

Finally, let’s plot the result to verify that we get the expected one.
As we can observe, the model has been able to learn both the first and second order seasonality effects, i.e. a cyclical pattern with a periodicity of around 10, as well as spikes that can be seen once every 40 or so years. Moreover, we not only have point estimates for the forecast but can also use the uncertainty estimates from the model to bound our forecasts.

### 27.5 Acknowledgements

We would like to thank Slawek Smyl for many helpful resources and suggestions. Fast inference would not have been possible without the support of JAX and the XLA teams, so we would like to thank them for providing such a great open-source platform for us to build on, and for their responsiveness in dealing with our feature requests and bug reports.

### 27.6 References

Some data are discrete but intrinsically ordered, these are called ordinal data. One example is the Liekert scale for questionairs (“this is an informative tutorial”: 1. strongly disagree / 2. disagree / 3. neither agree nor disagree / 4. agree / 5. strongly agree). Ordinal data is also ubiquitous in the medical world (e.g. the Glasgow Coma Scale for measuring neurological disfunctioning).

This poses a challenge for statistical modeling as the data do not fit the most well known modelling approaches (e.g. linear regression). Modeling the data as categorical is one possibility, but it disregards the inherent ordering in the data, and may be less statistically efficient. There are multiple appoaches for modeling ordered data. Here we will show how to use the OrderedLogistic distribution using cutpoints that are sampled from Improper priors, from a Normal distribution and induced via categories’ probabilities from Dirichlet distribution. For a more in-depth discussion of Bayesian modeling of ordinal data, see e.g. Michael Betancourt’s Ordinal Regression case study


```
[1]: # !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[2]: import pandas as pd
   import seaborn as sns

   from jax import numpy as np, random

   import numpyro
   from numpyro import handlers, sample
   from numpyro.distributions import (Categorical,
   Dirichlet,
   ImproperUniform,
   Normal,
   OrderedLogistic,
   TransformedDistribution,
   constraints,
   transforms,
   )

   from numpyro.infer import MCMC, NUTS
   from numpyro.infer.reparam import TransformReparam

   assert numpyro.__version__.startswith("0.15.2")
```
28.1 Data Generation

First, generate some data with ordinal structure

```python
simkeys = random.split(random.PRNGKey(1), 2)
nsim = 50
nclasses = 3
Y = Categorical(logits=np.zeros(nclasses)).sample(simkeys[0], sample_shape=(nsim,))
X = Normal().sample(simkeys[1], sample_shape=(nsim,))
X += Y

print("value counts of Y:")
df = pd.DataFrame({"X": X, "Y": Y})
print(df.Y.value_counts())
for i in range(nclasses):
    print(f"mean(X) for Y == {i}: {X[np.where(Y==i)].mean():.3f}"")

value counts of Y:
0   15
1   19
2   16
Name: Y, dtype: int64
mean(X) for Y == 0: 0.042
mean(X) for Y == 1: 0.832
mean(X) for Y == 2: 1.448
```

```python
sns.violinplot(x="Y", y="X", data=df);
```

## 28.2 Improper Prior

We will model the outcomes $Y$ as coming from an OrderedLogistic distribution, conditional on $X$. The OrderedLogistic distribution in NumPyro requires ordered cutpoints. We can use the `ImproperUniform` distribution to introduce a parameter with an arbitrary support that is otherwise completely uninformative, and then add an `ordered_vector` constraint.

```python
[5]: def model1(X, Y, nclasses=3):
    b_X_eta = sample("b_X_eta", Normal(0, 5))
    c_y = sample(
        "c_y",
        ImproperUniform(
            support=constraints.ordered_vector,
            batch_shape=(),
            event_shape=(nclasses - 1,),
        ),
    )
    with numpyro.plate("obs", X.shape[0]):
        eta = X * b_X_eta
        sample("Y", OrderedLogistic(eta, c_y), obs=Y)
```

```python
mcmc_key = random.PRNGKey(1234)
kernel = NUTS(model1)
mcmc = MCMC(kernel, num_warmup=250, num_samples=750)
mcmc.run(mcmc_key, X, Y, nclasses)
mcmc.print_summary()
```

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_X_eta</td>
<td>1.44</td>
<td>0.37</td>
<td>1.42</td>
<td>0.83</td>
<td>2.05</td>
<td>349.38</td>
<td>1.01</td>
</tr>
<tr>
<td>c_y[0]</td>
<td>-0.10</td>
<td>0.38</td>
<td>-0.10</td>
<td>-0.71</td>
<td>0.51</td>
<td>365.63</td>
<td>1.00</td>
</tr>
<tr>
<td>c_y[1]</td>
<td>2.15</td>
<td>0.49</td>
<td>2.13</td>
<td>1.38</td>
<td>2.99</td>
<td>376.45</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Number of divergences: 0

The `ImproperUniform` distribution allows us to use parameters with constraints on their domain, without adding any additional information e.g. about the location or scale of the prior distribution on that parameter.

If we want to incorporate such information, for instance that the values of the cut-points should not be too far from zero, we can add an additional `sample` statement that uses another prior, coupled with an `obs` argument. In the example below we first sample cutpoints $c_y$ from the `ImproperUniform` distribution with `constraints.ordered_vector` as before, and then sample a dummy parameter from a `Normal` distribution while conditioning on $c_y$ using `obs=c_y`. Effectively, we’ve created an improper / unnormalized prior that results from restricting the support of a `Normal` distribution to the ordered domain.

```python
[6]: def model2(X, Y, nclasses=3):
    b_X_eta = sample("b_X_eta", Normal(0, 5))
    c_y = sample(
        "c_y",
        ImproperUniform(
```

(continues on next page)
support=constraints.ordered_vector,
batch_shape=(),
event_shape=(nclasses - 1),
)

sample("c_y_smp", Normal(0, 1), obs=c_y)

with numpyro.plate("obs", X.shape[0]):
    eta = X * b_X_eta
    sample("Y", OrderedLogistic(eta, c_y), obs=Y)

kernel = NUTS(model2)
mcmc = MCMC(kernel, num_warmup=250, num_samples=750)
mcmc.run(mcmc_key, X, Y, nclasses)
mcmc.print_summary()

<table>
<thead>
<tr>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_X_eta</td>
<td>1.23</td>
<td>0.31</td>
<td>1.23</td>
<td>0.64</td>
<td>501.31</td>
<td>1.01</td>
</tr>
<tr>
<td>c_y[0]</td>
<td>-0.24</td>
<td>0.34</td>
<td>-0.23</td>
<td>-0.76</td>
<td>0.38</td>
<td>492.91</td>
</tr>
<tr>
<td>c_y[1]</td>
<td>1.77</td>
<td>0.40</td>
<td>1.76</td>
<td>1.11</td>
<td>2.42</td>
<td>628.46</td>
</tr>
</tbody>
</table>

Number of divergences: 0

### 28.3 Proper Prior

If having a proper prior for those cutpoints `c_y` is desirable (e.g. to sample from that prior and get prior predictive), we can use `TransformedDistribution` with an `OrderedTransform` transform as follows.

```python
[7]: def model3(X, Y, nclasses=3):
    b_X_eta = sample("b_X_eta", Normal(0, 5))
    c_y = sample(
        "c_y",
        TransformedDistribution(
            Normal(0, 1).expand([nclasses - 1]), transforms.OrderedTransform()
        ),
    )
    with numpyro.plate("obs", X.shape[0]):
        eta = X * b_X_eta
        sample("Y", OrderedLogistic(eta, c_y), obs=Y)
```

kernel = NUTS(model3)
mcmc = MCMC(kernel, num_warmup=250, num_samples=750)
mcmc.run(mcmc_key, X, Y, nclasses)
mcmc.print_summary()

<table>
<thead>
<tr>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_X_eta</td>
<td>1.77</td>
<td>0.40</td>
<td>1.76</td>
<td>1.11</td>
<td>2.42</td>
<td>628.46</td>
</tr>
<tr>
<td>c_y[0]</td>
<td>-0.24</td>
<td>0.34</td>
<td>-0.23</td>
<td>-0.76</td>
<td>0.38</td>
<td>492.91</td>
</tr>
<tr>
<td>c_y[1]</td>
<td>1.77</td>
<td>0.40</td>
<td>1.76</td>
<td>1.11</td>
<td>2.42</td>
<td>628.46</td>
</tr>
</tbody>
</table>

Number of divergences: 0
28.4 Principled prior with Dirichlet Distribution

It is non-trivial to apply our expertise over the cutpoints in latent space (even more so when we are having to provide a prior before applying the OrderedTransform).

Natural inclination would be to apply Dirichlet prior model to the ordinal probabilities. We will follow proposal by M.Betancourt ([1], Section 2.2) and use Dirichlet prior model to induce cutpoints indirectly via SimplexToOrdered-Transform. This approach should be advantageous when there is a need for strong prior knowledge to be added to our Ordinal model, e.g. when one of the categories is missing in our dataset or when some categories are strongly separated (leading to non-identifiability of the cutpoints). Moreover, such parametrization allows us to sample our model and conduct prior predictive checks (unlike model1 with ImproperUniform).

We can sample cutpoints directly from TransformedDistribution(Dirichlet(concentration), transforms.SimplexToOrderedTransform(anchor_point)). However, if we use the Transform within the reparam handler context, we can capture not only the induced cutpoints, but also the sampled Ordinal probabilities implied by the concentration parameter. anchor_point is a nuisance parameter to improve identifiability of our transformation (for details please see [1], Section 2.2)

Please note that we cannot compare latent cutpoints or b_X_eta separately across the various models as they are inherently linked.

[8]: # We will apply a nudge towards equal probability for each category # (corresponds to equal logits of the true data generating process)
concentration = np.ones((nclasses,)) * 10.0

[9]: def model4(X, Y, nclasses, concentration, anchor_point=0.0):
    b_X_eta = sample("b_X_eta", Normal(0, 5))

    with handlers.reparam(config={"c_y": TransformReparam()}):
        c_y = sample(
            "c_y",
            TransformedDistribution(
                Dirichlet(concentration),
                transforms.SimplexToOrderedTransform(anchor_point),
            ),
        )

    with numpyro.plate("obs", X.shape[0]):
        eta = X * b_X_eta
        sample("Y", OrderedLogistic(eta, c_y), obs=Y)

kernel = NUTS(model4)
mcmc = MCMC(kernel, num_warmup=250, num_samples=750)
mcmc.run(mcmc_key, X, Y, nclasses, concentration)
# with `exclude_deterministic=False`, we will also show the ordinal probabilities sampled from Dirichlet (vis. `c_y_base`)

```python
mcmc.print_summary(exclude_deterministic=False)
```

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_X_eta</td>
<td>1.01</td>
<td>0.26</td>
<td>1.01</td>
<td>0.59</td>
<td>1.42</td>
<td>388.46</td>
<td>1.00</td>
</tr>
<tr>
<td>c_y[0]</td>
<td>-0.42</td>
<td>0.26</td>
<td>-0.42</td>
<td>-0.88</td>
<td>-0.05</td>
<td>491.73</td>
<td>1.00</td>
</tr>
<tr>
<td>c_y[1]</td>
<td>1.34</td>
<td>0.29</td>
<td>1.34</td>
<td>0.86</td>
<td>1.80</td>
<td>617.53</td>
<td>1.00</td>
</tr>
<tr>
<td>c_y_base[0]</td>
<td>0.40</td>
<td>0.06</td>
<td>0.40</td>
<td>0.29</td>
<td>0.49</td>
<td>488.71</td>
<td>1.00</td>
</tr>
<tr>
<td>c_y_base[1]</td>
<td>0.39</td>
<td>0.06</td>
<td>0.39</td>
<td>0.29</td>
<td>0.48</td>
<td>523.65</td>
<td>1.00</td>
</tr>
<tr>
<td>c_y_base[2]</td>
<td>0.21</td>
<td>0.05</td>
<td>0.21</td>
<td>0.13</td>
<td>0.29</td>
<td>610.33</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 0
Real-world datasets often contain many missing values. In those situations, we have to either remove those missing data (also known as “complete case”) or replace them by some values. Though using complete case is pretty straightforward, it is only applicable when the number of missing entries is so small that throwing away those entries would not affect much the power of the analysis we are conducting on the data. The second strategy, also known as imputation, is more applicable and will be our focus in this tutorial.

Probably the most popular way to perform imputation is to fill a missing value with the mean, median, or mode of its corresponding feature. In that case, we implicitly assume that the feature containing missing values has no correlation with the remaining features of our dataset. This is a pretty strong assumption and might not be true in general. In addition, it does not encode any uncertainty that we might put on those values. Below, we will construct a Bayesian setting to resolve those issues. In particular, given a model on the dataset, we will

• create a generative model for the feature with missing value
• and consider missing values as unobserved latent variables.

```python
[1]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[1]: # first, we need some imports
    import os
    from IPython.display import set_matplotlib_formats
    from matplotlib import pyplot as plt
    import numpy as np
    import pandas as pd

    from jax import numpy as jnp, random
    import numpyro
    from numpyro import distributions as dist
    from numpyro.infer import MCMC, NUTS, Predictive

    plt.style.use("seaborn")
    if "NUMPYRO_SPHINXBUILD" in os.environ:
        set_matplotlib_formats("svg")

    assert numpyro.__version__.startswith("0.15.2")
```
29.1 Dataset

The data is taken from the competition Titanic: Machine Learning from Disaster hosted on kaggle. It contains information of passengers in the Titanic accident such as name, age, gender,... And our target is to predict if a person is more likely to survive.

```
[2]: train = pd.read_csv(
    "https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv"
)
train.info()
train.head()
```

```
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
# Column  Non-Null Count   Dtype
0  PassengerId  891 non-null  int64
1  Survived    891 non-null  int64
2  Pclass      891 non-null  int64
3  Name        891 non-null  object
4  Sex         891 non-null  object
5  Age         714 non-null  float64
6  SibSp       891 non-null  int64
7  Parch       891 non-null  int64
8  Ticket      891 non-null  object
9  Fare        891 non-null  float64
10 Cabin      204 non-null  object
11 Embarked   889 non-null  object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

```
[2]:  
  PassengerId  Survived  Pclass  
0      1         0       3
1      2         1       1
2      3         1       3
3      4         1       1
4      5         0       3

<table>
<thead>
<tr>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>SibSp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Braund, Mr. Owen Harris</td>
<td>male</td>
<td>22.0</td>
<td>1</td>
</tr>
<tr>
<td>Cumings, Mrs. John Bradley (Florence Briggs Th...</td>
<td>female</td>
<td>38.0</td>
<td>1</td>
</tr>
<tr>
<td>Heikkinen, Miss. Laina</td>
<td>female</td>
<td>26.0</td>
<td>0</td>
</tr>
<tr>
<td>Futrelle, Mrs. Jacques Heath (Lily May Peel)</td>
<td>female</td>
<td>35.0</td>
<td>1</td>
</tr>
<tr>
<td>Allen, Mr. William Henry</td>
<td>male</td>
<td>35.0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parch</th>
<th>Ticket</th>
<th>Fare</th>
<th>Cabin</th>
<th>Embarked</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A/5 21171</td>
<td>7.2500</td>
<td>NaN</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>PC 17599</td>
<td>71.2833</td>
<td>C85</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>STON/02. 3101282</td>
<td>7.9250</td>
<td>NaN</td>
<td>S</td>
</tr>
<tr>
<td>3</td>
<td>113803</td>
<td>53.1000</td>
<td>C123</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>373450</td>
<td>8.0500</td>
<td>NaN</td>
<td>S</td>
</tr>
</tbody>
</table>
```

Look at the data info, we know that there are missing data at Age, Cabin, and Embarked columns. Although Cabin is an important feature (because the position of a cabin in the ship can affect the chance of people in that cabin to...
survive), we will skip it in this tutorial for simplicity. In the dataset, there are many categorical columns and two numerical columns Age and Fare. Let’s first look at the distribution of those categorical columns:

```python
[3]: for col in ['Survived', 'Pclass', 'Sex', 'SibSp', 'Parch', 'Embarked']:
    print(train_df[col].value_counts(), end='\n
  0 549
  1 342
Name: Survived, dtype: int64

  3 491
  1 216
  2 184
Name: Pclass, dtype: int64

  male 577
  female 314
Name: Sex, dtype: int64

  0 608
  1 209
  2 28
  4 18
  3 16
  8 7
  5 5
Name: SibSp, dtype: int64

  0 678
  1 118
  2 80
  3 5
  5 5
  4 4
  6 1
Name: Parch, dtype: int64

  S 644
  C 168
  Q 77
Name: Embarked, dtype: int64
```

### 29.2 Prepare data

First, we will merge rare groups in SibSp and Parch columns together. In addition, we’ll fill 2 missing entries in Embarked by the mode S. Note that we can make a generative model for those missing entries in Embarked but let’s skip doing so for simplicity.

```python
[4]: train_df.SibSp.clip(0, 1, inplace=True)
train_df.Parch.clip(0, 2, inplace=True)
train_df.Embarked.fillna('S', inplace=True)
```
Looking closer at the data, we can observe that each name contains a title. We know that age is correlated with the title of the name: e.g. those with Mrs. would be older than those with Miss. (on average) so it might be good to create that feature. The distribution of titles is:

```python
[5]: train_df.Name.str.split(" ",").str.get(1).str.split(" ").str.get(0).value_counts()
```

<table>
<thead>
<tr>
<th>Title</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mr.</td>
<td>517</td>
</tr>
<tr>
<td>Miss.</td>
<td>182</td>
</tr>
<tr>
<td>Mrs.</td>
<td>125</td>
</tr>
<tr>
<td>Master.</td>
<td>40</td>
</tr>
<tr>
<td>Dr.</td>
<td>7</td>
</tr>
<tr>
<td>Rev.</td>
<td>6</td>
</tr>
<tr>
<td>Mlle.</td>
<td>2</td>
</tr>
<tr>
<td>Col.</td>
<td>2</td>
</tr>
<tr>
<td>Major.</td>
<td>2</td>
</tr>
<tr>
<td>Lady.</td>
<td>1</td>
</tr>
<tr>
<td>Sir.</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>Ms.</td>
<td>1</td>
</tr>
<tr>
<td>Capt.</td>
<td>1</td>
</tr>
<tr>
<td>Mme.</td>
<td>1</td>
</tr>
<tr>
<td>Jonkheer.</td>
<td>1</td>
</tr>
<tr>
<td>Don.</td>
<td>1</td>
</tr>
<tr>
<td>Name:</td>
<td>Name, dtype: int64</td>
</tr>
</tbody>
</table>

We will make a new column `Title`, where rare titles are merged into one group `Misc.`.

```python
[6]: train_df["Title"] = (  
        train_df.Name.str.split(" ",").str.get(1).str.split(" ").str.get(0).apply(  
            lambda x: x if x in ["Mr.", "Miss.", "Mrs.", "Master."] else "Misc.")
    )
```

Now, it is ready to turn the dataframe, which includes categorical values, into numpy arrays. We also perform standardization (a good practice for regression models) for `Age` column.

```python
[7]: title_cat = pd.CategoricalDtype(  
        categories=["Mr.", "Miss.", "Mrs.", "Master.", "Misc."], ordered=True
    )
embarked_cat = pd.CategoricalDtype(categories=["S", "C", "Q"], ordered=True)
age_mean, age_std = train_df.Age.mean(), train_df.Age.std()
data = dict(  
        age=train_df.Age.pipe(lambda x: (x - age_mean) / age_std).values,  
        pclass=train_df.Pclass.values - 1,  
        title=train_df.Title.astype(title_cat).cat.codes.values,  
        sex=(train_df.Sex == "male").astype(int).values,  
        sibsp=train_df.SibSp.values,  
        parch=train_df.Parch.values,  
        embarked=train_df.Embarked.astype(embarked_cat).cat.codes.values,
    )
survived = train_df.Survived.values
# compute the age mean for each title
```

(continues on next page)
age_notnan = data["age"][jnp.isfinite(data["age")]]
title_notnan = data["title"][]jnp.isfinite(data["age")])
age_mean_by_title = jnp.stack([age_notnan[title_notnan == i].mean() for i in range(5)])

29.3 Modelling

First, we want to note that in NumPyro, the following models

```python
def model1a():
    x = numpyro.sample("x", dist.Normal(0, 1).expand([10]))
```

and

```python
def model1b():
    x = numpyro.sample("x", dist.Normal(0, 1).expand([10]).mask(False))
numpyro.sample("x_obs", dist.Normal(0, 1).expand([10]), obs=x)
```

are equivalent in the sense that both of them have

- the same latent sites \( x \) drawn from \( \text{dist.Normal}(0, 1) \) prior,
- and the same log densities \( \text{dist.Normal}(0, 1).\log\text{prob}(x) \).

Now, assume that we observed the last 6 values of \( x \) (non-observed entries take value NaN), the typical model will be

```python
def model2a(x):
    x_impute = numpyro.sample("x_impute", dist.Normal(0, 1).expand([4]))
    x_obs = numpyro.sample("x_obs", dist.Normal(0, 1).expand([6]), obs=x[4:])
    x_imputed = jnp.concatenate([x_impute, x_obs])
```

or with the usage of mask,

```python
def model2b(x):
    x_impute = numpyro.sample("x_impute", dist.Normal(0, 1).expand([4]).mask(False))
    x_imputed = jnp.concatenate([x_impute, x[4:]])
numpyro.sample("x", dist.Normal(0, 1).expand([10]), obs=x_imputed)
```

Both approaches to model the partial observed data \( x \) are equivalent. For the model below, we will use the latter method.

```python
[8]: def model(
    age, pclass, title, sex, sibsp, parch, embarked, survived=None, bayesian_impute=True
):
    b_pclass = numpyro.sample("b_Pclass", dist.Normal(0, 1).expand([3]))
    b_title = numpyro.sample("b_Title", dist.Normal(0, 1).expand([5]))
    b_sex = numpyro.sample("b_Sex", dist.Normal(0, 1).expand([2]))
    b_sibsp = numpyro.sample("b_SibSp", dist.Normal(0, 1).expand([2]))
    b_parch = numpyro.sample("b_Parch", dist.Normal(0, 1).expand([3]))
    b_embarked = numpyro.sample("b_Embarked", dist.Normal(0, 1).expand([3]))
    # impute age by Title
    isnan = np.isnan(age)
age_nanidx = np.nonzero(isnan)[0]
```

(continues on next page)
if bayesian_impute:
    age_mu = numpyro.sample("age_mu", dist.Normal(0, 1).expand([5]))
    age_mu = age_mu[title]
    age_sigma = numpyro.sample("age_sigma", dist.Normal(0, 1).expand([5]))
    age_sigma = age_sigma[title]
    age_impute = numpyro.sample(
        "age_impute",
        dist.Normal(age_mu[age_nanidx], age_sigma[age_nanidx]).mask(False),
    )
    age = jnp.asarray(age).at[age_nanidx].set(age_impute)
    numpyro.sample("age", dist.Normal(age_mu, age_sigma), obs=age)
else:
    # fill missing data by the mean of ages for each title
    age_impute = age_mean_by_title[title][age_nanidx]
    age = jnp.asarray(age).at[age_nanidx].set(age_impute)

    a = numpyro.sample("a", dist.Normal(0, 1))
    b_age = numpyro.sample("b_Age", dist.Normal(0, 1))
    logits = a + b_age * age
    logits = logits + b_title[title] + b_pclass[pclass] + b_sex[sex]
    logits = logits + b_sibsp[sibsp] + b_parch[parch] + b_embarked[embarked]
    numpyro.sample("survived", dist.Bernoulli(logits=logits), obs=survived)

Note that in the model, the prior for age is dist.Normal(age_mu, age_sigma), where the values of age_mu and age_sigma depend on title. Because there are missing values in age, we will encode those missing values in the latent parameter age_impute. Then we can replace NaN entries in age with the vector age_impute.

29.4 Sampling

We will use MCMC with NUTS kernel to sample both regression coefficients and imputed values.

```python
mcmc = MCMC(NUTS(model), num_warmup=1000, num_samples=1000)
mcmc.run(random.PRNGKey(0), **data, survived=survived)
mcmc.print_summary()
```

<table>
<thead>
<tr>
<th>Sample: 100%</th>
<th>2000/2000 [00:15&lt;00:00]</th>
<th>132.15it/s</th>
<th>63 steps of...</th>
<th>size 5.68e-02</th>
<th>acc. prob=0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>mean</td>
<td>std</td>
<td>median</td>
<td>5.0%</td>
<td>95.0%</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>age_impute[0]</td>
<td>0.20</td>
<td>0.84</td>
<td>0.18</td>
<td>-1.22</td>
<td>1.53</td>
</tr>
<tr>
<td>age_impute[1]</td>
<td>-0.06</td>
<td>0.86</td>
<td>-0.08</td>
<td>-1.41</td>
<td>1.26</td>
</tr>
<tr>
<td>age_impute[2]</td>
<td>0.38</td>
<td>0.73</td>
<td>0.39</td>
<td>-0.80</td>
<td>1.58</td>
</tr>
<tr>
<td>age_impute[3]</td>
<td>0.25</td>
<td>0.84</td>
<td>0.23</td>
<td>-0.99</td>
<td>1.86</td>
</tr>
<tr>
<td>age_impute[4]</td>
<td>-0.63</td>
<td>0.91</td>
<td>-0.59</td>
<td>-1.99</td>
<td>0.87</td>
</tr>
<tr>
<td>age_impute[5]</td>
<td>0.21</td>
<td>0.89</td>
<td>0.19</td>
<td>-1.02</td>
<td>1.97</td>
</tr>
<tr>
<td>age_impute[6]</td>
<td>0.45</td>
<td>0.82</td>
<td>0.46</td>
<td>-0.90</td>
<td>1.73</td>
</tr>
<tr>
<td>age_impute[7]</td>
<td>-0.62</td>
<td>0.86</td>
<td>-0.62</td>
<td>-2.13</td>
<td>0.72</td>
</tr>
<tr>
<td>age_impute[8]</td>
<td>-0.13</td>
<td>0.90</td>
<td>-0.14</td>
<td>-1.64</td>
<td>1.38</td>
</tr>
<tr>
<td>age_impute[9]</td>
<td>0.24</td>
<td>0.84</td>
<td>0.26</td>
<td>-1.06</td>
<td>1.77</td>
</tr>
<tr>
<td>age_impute[10]</td>
<td>0.20</td>
<td>0.89</td>
<td>0.21</td>
<td>-1.26</td>
<td>1.65</td>
</tr>
<tr>
<td>age_impute[11]</td>
<td>0.17</td>
<td>0.91</td>
<td>0.19</td>
<td>-1.59</td>
<td>1.48</td>
</tr>
<tr>
<td>age_impute[12]</td>
<td>-0.65</td>
<td>0.89</td>
<td>-0.68</td>
<td>-2.12</td>
<td>0.77</td>
</tr>
<tr>
<td>age_impute[13]</td>
<td>0.21</td>
<td>0.85</td>
<td>0.18</td>
<td>-1.24</td>
<td>1.65</td>
</tr>
<tr>
<td>age_impute[14]</td>
<td>0.05</td>
<td>0.92</td>
<td>0.05</td>
<td>-1.40</td>
<td>1.65</td>
</tr>
<tr>
<td>age_impute[15]</td>
<td>0.37</td>
<td>0.94</td>
<td>0.37</td>
<td>-1.02</td>
<td>1.98</td>
</tr>
<tr>
<td>age_impute[16]</td>
<td>-1.74</td>
<td>0.26</td>
<td>-1.74</td>
<td>-2.13</td>
<td>-1.32</td>
</tr>
<tr>
<td>age_impute[17]</td>
<td>0.21</td>
<td>0.89</td>
<td>0.22</td>
<td>-1.30</td>
<td>1.60</td>
</tr>
<tr>
<td>age_impute[18]</td>
<td>0.18</td>
<td>0.90</td>
<td>0.18</td>
<td>-1.26</td>
<td>1.58</td>
</tr>
<tr>
<td>age_impute[19]</td>
<td>-0.67</td>
<td>0.86</td>
<td>-0.66</td>
<td>-1.97</td>
<td>0.85</td>
</tr>
<tr>
<td>age_impute[20]</td>
<td>0.23</td>
<td>0.89</td>
<td>0.27</td>
<td>-1.19</td>
<td>1.71</td>
</tr>
<tr>
<td>age_impute[21]</td>
<td>0.21</td>
<td>0.87</td>
<td>0.20</td>
<td>-1.11</td>
<td>1.68</td>
</tr>
<tr>
<td>age_impute[22]</td>
<td>0.19</td>
<td>0.90</td>
<td>0.18</td>
<td>-1.26</td>
<td>1.63</td>
</tr>
<tr>
<td>age_impute[23]</td>
<td>-0.15</td>
<td>0.85</td>
<td>-0.15</td>
<td>-1.57</td>
<td>1.24</td>
</tr>
<tr>
<td>age_impute[24]</td>
<td>-0.71</td>
<td>0.89</td>
<td>-0.73</td>
<td>-2.05</td>
<td>0.82</td>
</tr>
<tr>
<td>age_impute[25]</td>
<td>0.20</td>
<td>0.85</td>
<td>0.19</td>
<td>-1.20</td>
<td>1.62</td>
</tr>
<tr>
<td>age_impute[26]</td>
<td>0.21</td>
<td>0.88</td>
<td>0.21</td>
<td>-1.20</td>
<td>1.68</td>
</tr>
<tr>
<td>age_impute[27]</td>
<td>-0.69</td>
<td>0.91</td>
<td>-0.73</td>
<td>-2.20</td>
<td>0.77</td>
</tr>
<tr>
<td>age_impute[28]</td>
<td>0.60</td>
<td>0.77</td>
<td>0.60</td>
<td>-0.61</td>
<td>1.95</td>
</tr>
<tr>
<td>age_impute[29]</td>
<td>0.20</td>
<td>0.89</td>
<td>0.17</td>
<td>-1.23</td>
<td>1.71</td>
</tr>
<tr>
<td>age_impute[30]</td>
<td>0.24</td>
<td>0.87</td>
<td>0.23</td>
<td>-1.14</td>
<td>1.60</td>
</tr>
<tr>
<td>age_impute[31]</td>
<td>-1.72</td>
<td>0.26</td>
<td>-1.72</td>
<td>-2.11</td>
<td>-1.28</td>
</tr>
<tr>
<td>age_impute[32]</td>
<td>0.44</td>
<td>0.77</td>
<td>0.43</td>
<td>-0.83</td>
<td>1.58</td>
</tr>
<tr>
<td>age_impute[33]</td>
<td>0.34</td>
<td>0.89</td>
<td>0.32</td>
<td>-1.14</td>
<td>1.73</td>
</tr>
<tr>
<td>age_impute[34]</td>
<td>-1.72</td>
<td>0.26</td>
<td>-1.71</td>
<td>-2.11</td>
<td>-1.26</td>
</tr>
<tr>
<td>age_impute[35]</td>
<td>-0.45</td>
<td>0.90</td>
<td>-0.47</td>
<td>-2.06</td>
<td>0.92</td>
</tr>
<tr>
<td>age_impute[36]</td>
<td>0.30</td>
<td>0.84</td>
<td>0.30</td>
<td>-1.03</td>
<td>1.73</td>
</tr>
<tr>
<td>age_impute[37]</td>
<td>0.33</td>
<td>0.88</td>
<td>0.32</td>
<td>-1.10</td>
<td>1.81</td>
</tr>
<tr>
<td>age_impute[38]</td>
<td>0.33</td>
<td>0.76</td>
<td>0.35</td>
<td>-0.94</td>
<td>1.56</td>
</tr>
<tr>
<td>age_impute[39]</td>
<td>0.19</td>
<td>0.93</td>
<td>0.21</td>
<td>-1.32</td>
<td>1.82</td>
</tr>
<tr>
<td>age_impute[40]</td>
<td>-0.67</td>
<td>0.88</td>
<td>-0.69</td>
<td>-1.94</td>
<td>0.88</td>
</tr>
<tr>
<td>age_impute[41]</td>
<td>0.17</td>
<td>0.89</td>
<td>0.14</td>
<td>-1.30</td>
<td>1.43</td>
</tr>
<tr>
<td>age_impute[42]</td>
<td>0.23</td>
<td>0.82</td>
<td>0.25</td>
<td>-1.12</td>
<td>1.48</td>
</tr>
<tr>
<td>age_impute[43]</td>
<td>0.22</td>
<td>0.82</td>
<td>0.21</td>
<td>-1.19</td>
<td>1.45</td>
</tr>
<tr>
<td>age_impute[44]</td>
<td>-0.41</td>
<td>0.85</td>
<td>-0.42</td>
<td>-1.96</td>
<td>0.78</td>
</tr>
<tr>
<td>age_impute[45]</td>
<td>-0.36</td>
<td>0.89</td>
<td>-0.35</td>
<td>-2.01</td>
<td>0.94</td>
</tr>
<tr>
<td>age_impute[46]</td>
<td>-0.33</td>
<td>0.91</td>
<td>-0.32</td>
<td>-1.76</td>
<td>1.27</td>
</tr>
<tr>
<td>age_impute[47]</td>
<td>-0.71</td>
<td>0.85</td>
<td>-0.69</td>
<td>-2.12</td>
<td>0.64</td>
</tr>
<tr>
<td>age_impute[48]</td>
<td>0.21</td>
<td>0.85</td>
<td>0.24</td>
<td>-1.21</td>
<td>1.64</td>
</tr>
<tr>
<td>age_impute[49]</td>
<td>0.42</td>
<td>0.82</td>
<td>0.41</td>
<td>-0.83</td>
<td>1.77</td>
</tr>
<tr>
<td>age_impute[50]</td>
<td>0.26</td>
<td>0.86</td>
<td>0.24</td>
<td>-1.18</td>
<td>1.63</td>
</tr>
<tr>
<td>age_impute[51]</td>
<td>-0.29</td>
<td>0.91</td>
<td>-0.30</td>
<td>-1.83</td>
<td>1.15</td>
</tr>
<tr>
<td>age_impute[52]</td>
<td>0.36</td>
<td>0.85</td>
<td>0.34</td>
<td>-1.12</td>
<td>1.68</td>
</tr>
<tr>
<td>age_impute[53]</td>
<td>-0.68</td>
<td>0.89</td>
<td>-0.65</td>
<td>-2.09</td>
<td>0.75</td>
</tr>
<tr>
<td>age_impute[54]</td>
<td>0.27</td>
<td>0.90</td>
<td>0.25</td>
<td>-1.24</td>
<td>1.68</td>
</tr>
<tr>
<td>age_impute[55]</td>
<td>0.36</td>
<td>0.89</td>
<td>0.36</td>
<td>-0.96</td>
<td>1.86</td>
</tr>
<tr>
<td>age_impute[56]</td>
<td>0.38</td>
<td>0.86</td>
<td>0.40</td>
<td>-1.00</td>
<td>1.75</td>
</tr>
<tr>
<td>age_impute[57]</td>
<td>0.01</td>
<td>0.91</td>
<td>0.03</td>
<td>-1.33</td>
<td>1.56</td>
</tr>
<tr>
<td>age_impute[58]</td>
<td>-0.69</td>
<td>0.91</td>
<td>-0.66</td>
<td>-2.13</td>
<td>0.78</td>
</tr>
<tr>
<td>age_impute[59]</td>
<td>-0.14</td>
<td>0.85</td>
<td>-0.16</td>
<td>-1.44</td>
<td>1.37</td>
</tr>
<tr>
<td>age_impute[60]</td>
<td>-0.59</td>
<td>0.94</td>
<td>-0.61</td>
<td>-2.19</td>
<td>0.93</td>
</tr>
<tr>
<td>age_impute[61]</td>
<td>0.24</td>
<td>0.92</td>
<td>0.25</td>
<td>-1.35</td>
<td>1.65</td>
</tr>
<tr>
<td>age_impute[62]</td>
<td>-0.55</td>
<td>0.91</td>
<td>-0.57</td>
<td>-2.01</td>
<td>0.96</td>
</tr>
<tr>
<td>age_impute[63]</td>
<td>0.21</td>
<td>0.90</td>
<td>0.19</td>
<td>-1.42</td>
<td>1.60</td>
</tr>
<tr>
<td>age_impute[64]</td>
<td>-0.66</td>
<td>0.88</td>
<td>-0.68</td>
<td>-2.04</td>
<td>0.73</td>
</tr>
<tr>
<td>age_impute[65]</td>
<td>0.44</td>
<td>0.78</td>
<td>0.48</td>
<td>-0.93</td>
<td>1.57</td>
</tr>
<tr>
<td>age_impute[66]</td>
<td>0.22</td>
<td>0.94</td>
<td>0.20</td>
<td>-1.35</td>
<td>1.69</td>
</tr>
<tr>
<td>age_impute[67]</td>
<td>0.33</td>
<td>0.76</td>
<td>0.34</td>
<td>-0.85</td>
<td>1.63</td>
</tr>
<tr>
<td>age_impute[68]</td>
<td>0.31</td>
<td>0.84</td>
<td>0.33</td>
<td>-1.08</td>
<td>1.60</td>
</tr>
<tr>
<td>age_impute[69]</td>
<td>0.26</td>
<td>0.91</td>
<td>0.25</td>
<td>-1.29</td>
<td>1.75</td>
</tr>
<tr>
<td>age_impute[70]</td>
<td>-0.67</td>
<td>0.86</td>
<td>-0.70</td>
<td>-2.02</td>
<td>0.70</td>
</tr>
<tr>
<td>age_impute[71]</td>
<td>-0.70</td>
<td>0.90</td>
<td>-0.69</td>
<td>-2.21</td>
<td>0.75</td>
</tr>
<tr>
<td>age_impute[72]</td>
<td>0.24</td>
<td>0.86</td>
<td>0.24</td>
<td>-1.07</td>
<td>1.66</td>
</tr>
<tr>
<td>age_impute[73]</td>
<td>0.34</td>
<td>0.72</td>
<td>0.35</td>
<td>-0.77</td>
<td>1.55</td>
</tr>
<tr>
<td>age_impute[74]</td>
<td>-0.64</td>
<td>0.85</td>
<td>-0.64</td>
<td>-2.10</td>
<td>0.77</td>
</tr>
<tr>
<td>age_impute[75]</td>
<td>0.41</td>
<td>0.78</td>
<td>0.42</td>
<td>-0.96</td>
<td>1.60</td>
</tr>
<tr>
<td>age_impute[76]</td>
<td>0.18</td>
<td>0.89</td>
<td>0.21</td>
<td>-1.19</td>
<td>1.74</td>
</tr>
<tr>
<td>age_impute[77]</td>
<td>0.21</td>
<td>0.84</td>
<td>0.22</td>
<td>-1.22</td>
<td>1.63</td>
</tr>
<tr>
<td>age_impute[78]</td>
<td>-0.36</td>
<td>0.87</td>
<td>-0.33</td>
<td>-1.81</td>
<td>1.01</td>
</tr>
<tr>
<td>age_impute[79]</td>
<td>0.20</td>
<td>0.84</td>
<td>0.19</td>
<td>-1.35</td>
<td>1.37</td>
</tr>
<tr>
<td>age_impute[80]</td>
<td>0.23</td>
<td>0.84</td>
<td>0.24</td>
<td>-1.09</td>
<td>1.61</td>
</tr>
<tr>
<td>age_impute[81]</td>
<td>0.28</td>
<td>0.90</td>
<td>0.32</td>
<td>-1.08</td>
<td>1.83</td>
</tr>
<tr>
<td>age_impute[82]</td>
<td>0.61</td>
<td>0.80</td>
<td>0.60</td>
<td>-0.61</td>
<td>2.03</td>
</tr>
<tr>
<td>age_impute[83]</td>
<td>0.24</td>
<td>0.89</td>
<td>0.26</td>
<td>-1.22</td>
<td>1.66</td>
</tr>
<tr>
<td>age_impute[84]</td>
<td>0.21</td>
<td>0.91</td>
<td>0.21</td>
<td>-1.35</td>
<td>1.65</td>
</tr>
<tr>
<td>age_impute[85]</td>
<td>0.24</td>
<td>0.92</td>
<td>0.21</td>
<td>-1.33</td>
<td>1.63</td>
</tr>
<tr>
<td>age_impute[86]</td>
<td>0.31</td>
<td>0.81</td>
<td>0.30</td>
<td>-0.86</td>
<td>1.76</td>
</tr>
<tr>
<td>age_impute[87]</td>
<td>-0.11</td>
<td>0.84</td>
<td>-0.10</td>
<td>-1.42</td>
<td>1.23</td>
</tr>
<tr>
<td>age_impute[88]</td>
<td>0.21</td>
<td>0.94</td>
<td>0.22</td>
<td>-1.31</td>
<td>1.77</td>
</tr>
<tr>
<td>age_impute[89]</td>
<td>0.24</td>
<td>0.86</td>
<td>0.23</td>
<td>-1.08</td>
<td>1.67</td>
</tr>
<tr>
<td>age_impute[90]</td>
<td>0.41</td>
<td>0.84</td>
<td>0.45</td>
<td>-0.88</td>
<td>1.90</td>
</tr>
<tr>
<td>age_impute[91]</td>
<td>0.21</td>
<td>0.86</td>
<td>0.20</td>
<td>-1.21</td>
<td>1.58</td>
</tr>
<tr>
<td>age_impute[92]</td>
<td>0.21</td>
<td>0.84</td>
<td>0.20</td>
<td>-1.21</td>
<td>1.57</td>
</tr>
<tr>
<td>age_impute[93]</td>
<td>0.22</td>
<td>0.87</td>
<td>0.23</td>
<td>-1.29</td>
<td>1.50</td>
</tr>
<tr>
<td>age_impute[94]</td>
<td>0.22</td>
<td>0.87</td>
<td>0.18</td>
<td>-1.09</td>
<td>1.70</td>
</tr>
<tr>
<td>age_impute[95]</td>
<td>0.22</td>
<td>0.87</td>
<td>0.23</td>
<td>-1.16</td>
<td>1.65</td>
</tr>
<tr>
<td>age_impute[96]</td>
<td>0.30</td>
<td>0.84</td>
<td>0.26</td>
<td>-1.18</td>
<td>1.57</td>
</tr>
<tr>
<td>age_impute[97]</td>
<td>0.23</td>
<td>0.87</td>
<td>0.25</td>
<td>-1.22</td>
<td>1.63</td>
</tr>
<tr>
<td>age_impute[98]</td>
<td>-0.36</td>
<td>0.91</td>
<td>-0.37</td>
<td>-1.96</td>
<td>1.03</td>
</tr>
<tr>
<td>age_impute[99]</td>
<td>0.15</td>
<td>0.87</td>
<td>0.14</td>
<td>-1.22</td>
<td>1.61</td>
</tr>
<tr>
<td>age_impute[100]</td>
<td>0.27</td>
<td>0.85</td>
<td>0.30</td>
<td>-1.27</td>
<td>1.45</td>
</tr>
<tr>
<td>age_impute[101]</td>
<td>0.25</td>
<td>0.87</td>
<td>0.25</td>
<td>-1.19</td>
<td>1.57</td>
</tr>
<tr>
<td>age_impute[102]</td>
<td>-0.29</td>
<td>0.85</td>
<td>-0.28</td>
<td>-1.70</td>
<td>1.10</td>
</tr>
<tr>
<td>age_impute[103]</td>
<td>0.01</td>
<td>0.89</td>
<td>0.01</td>
<td>-1.46</td>
<td>1.39</td>
</tr>
<tr>
<td>age_impute[104]</td>
<td>0.21</td>
<td>0.86</td>
<td>0.24</td>
<td>-1.16</td>
<td>1.64</td>
</tr>
<tr>
<td>age_impute[105]</td>
<td>0.24</td>
<td>0.93</td>
<td>0.21</td>
<td>-1.14</td>
<td>1.90</td>
</tr>
<tr>
<td>age_impute[106]</td>
<td>0.21</td>
<td>0.83</td>
<td>0.21</td>
<td>-1.09</td>
<td>1.55</td>
</tr>
<tr>
<td>age_impute[107]</td>
<td>0.22</td>
<td>0.85</td>
<td>0.22</td>
<td>-1.09</td>
<td>1.64</td>
</tr>
<tr>
<td>age_impute[108]</td>
<td>0.31</td>
<td>0.88</td>
<td>0.30</td>
<td>-1.10</td>
<td>1.76</td>
</tr>
<tr>
<td>age_impute[109]</td>
<td>0.22</td>
<td>0.86</td>
<td>0.23</td>
<td>-1.25</td>
<td>1.56</td>
</tr>
<tr>
<td>age_impute[110]</td>
<td>0.33</td>
<td>0.78</td>
<td>0.35</td>
<td>-0.95</td>
<td>1.62</td>
</tr>
<tr>
<td>age_impute[111]</td>
<td>0.22</td>
<td>0.88</td>
<td>0.21</td>
<td>-1.11</td>
<td>1.71</td>
</tr>
<tr>
<td>age_impute[112]</td>
<td>-0.03</td>
<td>0.90</td>
<td>-0.02</td>
<td>-1.38</td>
<td>1.55</td>
</tr>
<tr>
<td>age_impute[113]</td>
<td>0.24</td>
<td>0.85</td>
<td>0.23</td>
<td>-1.17</td>
<td>1.62</td>
</tr>
</tbody>
</table>

(continues on next page)
| age_impute[114] | 0.36 | 0.86 | 0.37 | -0.99 | 1.76 | 1155.67 | 1.00 |
| age_impute[115] | 0.26 | 0.96 | 0.28 | -1.37 | 1.81 | 1245.97 | 1.00 |
| age_impute[116] | 0.21 | 0.86 | 0.24 | -1.18 | 1.69 | 1565.59 | 1.00 |
| age_impute[117] | -0.31 | 0.94 | -0.33 | -1.91 | 1.19 | 1593.65 | 1.00 |
| age_impute[118] | -0.69 | 0.88 | -0.74 | -2.00 | 0.90 | 1536.44 | 1.00 |
| age_impute[119] | 0.63 | 0.81 | 0.66 | -0.65 | 1.89 | 899.61 | 1.00 |
| age_impute[120] | 0.27 | 0.90 | 0.26 | -1.16 | 1.74 | 1744.32 | 1.00 |
| age_impute[121] | 0.18 | 0.87 | 0.18 | -1.23 | 1.60 | 1625.58 | 1.00 |
| age_impute[122] | -0.39 | 0.88 | -0.38 | -1.71 | 1.12 | 1266.58 | 1.00 |
| age_impute[123] | 0.21 | 0.87 | 0.22 | -1.20 | 1.64 | 1315.42 | 1.00 |
| age_impute[124] | -0.69 | 0.88 | -0.74 | -2.00 | 0.90 | 1536.44 | 1.00 |
| age_impute[125] | 0.21 | 0.87 | 0.22 | -1.20 | 1.64 | 1315.42 | 1.00 |
| age_impute[126] | -0.31 | 0.94 | -0.33 | -1.91 | 1.19 | 1593.65 | 1.00 |

(continues on next page)
To double check that the assumption “age is correlated with title” is reasonable, let’s look at the inferred age by title. Recall that we performed standardization on \( \text{age} \), so here we need to scale back to original domain.

\[
\text{age}_\text{by_title} = \text{age}_\text{mean} + \text{age}_\text{std} \times \text{mcmc.get_samples()['age\_mu'].mean(\text{axis}=0)}
\]

```
[10]: age_by_title = age_mean + age_std * mcmc.get_samples()\[\text{"age\_mu"}\].mean(\text{axis}=0)
  dict(zip(title_cat.categories, age_by_title))
```

```
[10]: {
'Mr.': 32.434227,
'Miss.': 21.763992,
'Mrs.': 35.852997,
'Master.': 4.6297398,

Number of divergences: 0

(continues on next page)
The inferred result confirms our assumption that Age is correlated with Title:

- those with Master. title has pretty small age (in other words, they are children in the ship) comparing to the other groups,
- those with Mrs. title have larger age than those with Miss. title (in average).

We can also see that the result is similar to the actual statistical mean of Age given Title in our training dataset:

```python
train_df.groupby("Title")["Age"].mean()
```

<table>
<thead>
<tr>
<th>Title</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>4.574167</td>
</tr>
<tr>
<td>Misc.</td>
<td>42.384615</td>
</tr>
<tr>
<td>Miss.</td>
<td>21.773973</td>
</tr>
<tr>
<td>Mr.</td>
<td>32.368090</td>
</tr>
<tr>
<td>Mrs.</td>
<td>35.898148</td>
</tr>
</tbody>
</table>

Name: Age, dtype: float64

So far so good, we have many information about the regression coefficients together with imputed values and their uncertainties. Let's inspect those results a bit:

- The mean value -0.44 of b_Age implies that those with smaller ages have better chance to survive.
- The mean value (1.11, -1.07) of b_Sex implies that female passengers have higher chance to survive than male passengers.

### 29.5 Prediction

In NumPyro, we can use `Predictive` utility for making predictions from posterior samples. Let's check how well the model performs on the training dataset. For simplicity, we will get a survived prediction for each posterior sample and perform the majority rule on the predictions.

```python
posterior = mcmc.get_samples()
survived_pred = Predictive(model, posterior)(random.PRNGKey(1), **data)["survived"]
survived_pred = (survived_pred.mean(axis=0) >= 0.5).astype(jnp.uint8)
print("Accuracy:", (survived_pred == survived).sum() / survived.shape[0])
confusion_matrix = pd.crosstab(pd.Series(survived, name="actual"), pd.Series(survived_pred, name="predict")) / confusion_matrix.sum(axis=1)
```

```
Accuracy: 0.8271605
```

This is a pretty good result using a simple logistic regression model. Let’s see how the model performs if we don’t use Bayesian imputation here.
We can see that Bayesian imputation does a little bit better here.

Remark. When using posterior samples to perform prediction on the new data, we need to marginalize out age_impute because those imputing values are specific to the training data:

```python
posterior.pop("age_impute")
survived_pred = Predictive(model, posterior)(random.PRNGKey(3), **new_data)
```

### 29.6 References

2. Kaggle competition: Titanic: Machine Learning from Disaster
In this example we show how to use NUTS to sample from the posterior over the hyperparameters of a gaussian process.

```python
import argparse
import os
import time
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)
import jax
from jax import vmap
import jax.numpy as jnp
import jax.random as random
import numpyro
import numpyro.distributions as dist
from numpyro.infer import (MCMC,
                          NUTS,
                          init_to_feasible,
                          init_to_median,
                          init_to_sample,
                          init_to_uniform,
                          init_to_value,
                          )

matplotlib.use("Agg")  # noqa: E402

# squared exponential kernel with diagonal noise term
def kernel(X, Z, var, length, noise, jitter=1.0e-6, include_noise=True):
    deltaXsq = jnp.power((X[:, None] - Z) / length, 2.0)
    k = var * jnp.exp(-0.5 * deltaXsq)
    if include_noise:
        k += (noise + jitter) * jnp.eye(X.shape[0])
    return k

def model(X, Y):
    # set uninformative log-normal priors on our three kernel hyperparameters
    var = numpyro.sample("kernel_var", dist.LogNormal(0.0, 10.0))
    noise = numpyro.sample("kernel_noise", dist.LogNormal(0.0, 10.0))
    length = numpyro.sample("kernel_length", dist.LogNormal(0.0, 10.0))

    # compute kernel
    k = kernel(X, X, var, length, noise)

    # sample Y according to the standard gaussian process formula
    numpyro.sample("Y", dist.MultivariateNormal(loc=jnp.zeros(X.shape[0]), covariance_matrix=k),
                  obs=Y,
                  )

# helper function for doing hmc inference
def run_inference(model, args, rng_key, X, Y):
    start = time.time()
    # demonstrate how to use different HMC initialization strategies
    if args.init_strategy == "value":
        init_strategy = "value"
values={"kernel_var": 1.0, "kernel_noise": 0.05, "kernel_length": 0.5}

elif args.init_strategy == "median":
    init_strategy = init_to_median(num_samples=10)
elif args.init_strategy == "feasible":
    init_strategy = init_to_feasible()
elif args.init_strategy == "sample":
    init_strategy = init_to_sample()
elif args.init_strategy == "uniform":
    init_strategy = init_to_uniform(radius=1)
kernel = NUTS(model, init_strategy=init_strategy)
mcmc = MCMC(
    kernel,
    num_warmup=args.num_warmup,
    num_samples=args.num_samples,
    num_chains=args.num_chains,
    thinning=args.thinning,
    progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
)
mcmc.run(rng_key, X, Y)
mcmc.print_summary()
print("\nMCMC elapsed time:", time.time() - start)
return mcmc.get_samples()

# do GP prediction for a given set of hyperparameters. this makes use of the well-known
# formula for Gaussian process predictions

def predict(rng_key, X, Y, X_test, var, length, noise, use_cholesky=True):
    # compute kernels between train and test data, etc.
    k_pp = kernel(X_test, X_test, var, length, noise, include_noise=True)
    k_pX = kernel(X_test, X, var, length, noise, include_noise=False)
    k_XX = kernel(X, X, var, length, noise, include_noise=True)

    # since K_xx is symmetric positive-definite, we can use the more efficient and
    # stable Cholesky decomposition instead of matrix inversion
    if use_cholesky:
        K_xx_cho = jax.scipy.linalg.cho_factor(k_XX)
        K = k_pp - jnp.matmul(k_pX, jax.scipy.linalg.cho_solve(K_xx_cho, k_pX.T))
        mean = jnp.matmul(k_pX, jax.scipy.linalg.cho_solve(K_xx_cho, Y))
    else:
        K_xx_inv = jnp.linalg.inv(k_XX)
        K = k_pp - jnp.matmul(k_pX, jnp.matmul(K_xx_inv, jnp.transpose(k_pX)))
        mean = jnp.matmul(k_pX, jnp.matmul(K_xx_inv, Y))

    sigma_noise = jnp.sqrt(jnp.clip(jnp.diag(K), 0.0)) * jax.random.normal(
        rng_key, X_test.shape[:1])

    # we return both the mean function and a sample from the posterior predictive for the
    # given set of hyperparameters
    return mean, mean + sigma_noise

(continues on next page)
# create artificial regression dataset

def get_data(N=30, sigma_obs=0.15, N_test=400):
    np.random.seed(0)
    X = jnp.linspace(-1, 1, N)
    Y = X + 0.2 * jnp.power(X, 3.0) + 0.5 * jnp.power(0.5 + X, 2.0) * jnp.sin(4.0 * X)
    Y += sigma_obs * np.random.randn(N)
    Y -= jnp.mean(Y)
    Y /= jnp.std(Y)

    assert X.shape == (N,)
    assert Y.shape == (N,)

    X_test = jnp.linspace(-1.3, 1.3, N_test)

    return X, Y, X_test


def main(args):
    X, Y, X_test = get_data(N=args.num_data)

    # do inference
    rng_key, rng_key_predict = random.split(random.PRNGKey(0))
    samples = run_inference(model, args, rng_key, X, Y)

    # do prediction
    vmap_args = (
        random.split(rng_key_predict, samples["kernel_var"].shape[0]),
        samples["kernel_var"],
        samples["kernel_length"],
        samples["kernel_noise"],
    )
    means, predictions = vmap(
        lambda rng_key, var, length, noise: predict(
            rng_key, X, Y, X_test, var, length, noise, use_cholesky=args.use_cholesky
        )
    )(vmap_args)

    mean_prediction = np.mean(means, axis=0)
    percentiles = np.percentile(predictions, [5.0, 95.0], axis=0)

    # make plots
    fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)

    # plot training data
    ax.plot(X, Y, "kx")

    # plot 90% confidence level of predictions
    ax.fill_between(X_test, percentiles[0, :], percentiles[1, :], color="lightblue")

    # plot mean prediction
    ax.plot(X_test, mean_prediction, "blue", ls="solid", lw=2.0)
    ax.set(xlabel="X", ylabel="Y", title="Mean predictions with 90% CI")
plt.savefig("gp_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Gaussian Process example")
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--thinning", nargs="?", default=2, type=int)
    parser.add_argument("--num-data", nargs="?", default=25, type=int)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu"."")
    parser.add_argument("--init-strategy",
        default="median",
        type=str,
        choices=['median', 'feasible', 'value', 'uniform', 'sample'],
    )
    parser.add_argument("--no-cholesky", dest="use_cholesky", action="store_false")
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
EXAMPLE: BAYESIAN NEURAL NETWORK

We demonstrate how to use NUTS to do inference on a simple (small) Bayesian neural network with two hidden layers.

```
import argparse
import os
import time

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)
from jax import vmap
import jax.numpy as jnp
import jax.random as random

import numpyro
from numpyro import handlers
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

matplotlib.use("Agg")  # noqa: E402

# the non-linearity we use in our neural network
def nonlin(x):
    return jnp.tanh(x)

# a two-layer bayesian neural network with computational flow
# given by D_X => D_H => D_H => D_Y where D_H is the number of
# hidden units. (note we indicate tensor dimensions in the comments)
def model(X, Y, D_H, D_Y=1):
    N, D_X = X.shape
    # sample first layer (we put unit normal priors on all weights)
    w1 = numpyro.sample("w1", dist.Normal(jnp.zeros((D_X, D_H)), jnp.ones((D_X, D_H)))).to_event(1)
    assert w1.shape == (D_X, D_H)
    z1 = nonlin(jnp.matmul(X, w1))  # <= first layer of activations
    assert z1.shape == (N, D_H)

    # sample second layer
    w2 = numpyro.sample("w2", dist.Normal(jnp.zeros((D_H, D_H)), jnp.ones((D_H, D_H)))).to_event(1)
    assert w2.shape == (D_H, D_H)
    z2 = nonlin(jnp.matmul(z1, w2))  # <= second layer of activations
    assert z2.shape == (N, D_H)

    # sample final layer of weights and neural network output
    w3 = numpyro.sample("w3", dist.Normal(jnp.zeros((D_H, D_Y)), jnp.ones((D_H, D_Y)))).to_event(1)
    assert w3.shape == (D_H, D_Y)
    z3 = jnp.matmul(z2, w3)  # <= output of the neural network
    assert z3.shape == (N, D_Y)

    if Y is not None:
        assert z3.shape == Y.shape

    # we put a prior on the observation noise
    prec_obs = numpyro.sample("prec_obs", dist.Gamma(3.0, 1.0)).to_event(1)
    sigma_obs = 1.0 / jnp.sqrt(prec_obs)

    # observe data
    with numpyro.plate("data", N):
        # note we use to_event(1) because each observation has shape (1,)
        numpyro.sample("Y", dist.Normal(z3, sigma_obs).to_event(1), obs=Y)
# helper function for HMC inference

```python
def run_inference(model, args, rng_key, X, Y, D_H):
    start = time.time()
    kernel = NUTS(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, X, Y, D_H)
    mcmc.print_summary()
    print("MCMC elapsed time:", time.time() - start)
    return mcmc.get_samples()
```

# helper function for prediction

```python
def predict(model, rng_key, samples, X, D_H):
    model = handlers.substitute(handlers.seed(model, rng_key), samples)
    # note that Y will be sampled in the model because we pass Y=None here
    model_trace = handlers.trace(model).get_trace(X=X, Y=None, D_H=D_H)
    return model_trace["Y"]["value"]
```

# create artificial regression dataset

```python
def get_data(N=50, D_X=3, sigma_obs=0.05, N_test=500):
    D_Y = 1  # create 1d outputs
    np.random.seed(0)
    X = jnp.linspace(-1, 1, N)
    X = jnp.power(X[:, np.newaxis], jnp.arange(D_X))
    W = 0.5 * np.random.randn(D_X)
    Y = jnp.dot(X, W) + 0.5 * jnp.power(0.5 + X[:, 1], 2.0) * jnp.sin(4.0 * X[:, 1])
    Y += sigma_obs * np.random.randn(N)
    Y -= jnp.mean(Y)
    Y /= jnp.std(Y)

    assert X.shape == (N, D_X)
    assert Y.shape == (N, D_Y)

    X_test = jnp.linspace(-1.3, 1.3, N_test)
    X_test = jnp.power(X_test[:, np.newaxis], jnp.arange(D_X))

    return X, Y, X_test
```

```python
def main(args):
    N, D_X, D_H = args.num_data, 3, args.num_hidden
    X, Y, X_test = get_data(N=N, D_X=D_X)
```

# do inference
rng_key, rng_key_predict = random.split(random.PRNGKey(0))
samples = run_inference(model, args, rng_key, X, Y, D_H)

# predict Y_test at inputs X_test
vmap_args = (
samples,
    random.split(rng_key_predict, args.num_samples * args.num_chains),
)
predictions = vmap(
    lambda samples, rng_key: predict(model, rng_key, samples, X_test, D_H)
)(*vmap_args)
predictions = predictions[..., 0]

# compute mean prediction and confidence interval around median
mean_prediction = jnp.mean(predictions, axis=0)
percentiles = np.percentile(predictions, [5.0, 95.0], axis=0)

# make plots
fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)

# plot training data
ax.plot(X[:, 1], Y[:, 0], "kx")

# plot 90% confidence level of predictions
ax.fill_between(
    X_test[:, 1], percentiles[0, :], percentiles[1, :], color="lightblue"
)

# plot mean prediction
ax.plot(X_test[:, 1], mean_prediction, "blue", ls="solid", lw=2.0)
ax.set(xlabel="X", ylabel="Y", title="Mean predictions with 90% CI")

plt.savefig("bnn_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Bayesian neural network example")
    parser.add_argument("-n", "--num-samples", nargs="?", default=2000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--num-data", nargs="?", default=100, type=int)
    parser.add_argument("--num-hidden", nargs="?", default=5, type=int)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
EXAMPLE: AUTODAIS

AutoDAIS constructs a guide that combines elements of Hamiltonian Monte Carlo, Annealed Importance Sampling, and Variational Inference.

In this demo script we construct a somewhat artificial example involving a gaussian process binary classifier. We aim to demonstrate that:

- DAIS can achieve better ELBOs than e.g. mean field variational inference
- DAIS can achieve better posterior approximations than e.g. mean field variational inference
- DAIS improves as you increase $K$, the number of HMC steps used in the sampler

References:

[1] “MCMC Variational Inference via Uncorrected Hamiltonian Annealing,”
    Tomas Geffner, Justin Domke.

    Guodong Zhang, Kyle Hsu, Jianing Li, Chelsea Finn, Roger Grosse.
import argparse
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import expit
import seaborn as sns
from jax import random
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, SVI, Trace_ELBO, autoguide
from numpyro.util import enable_x64

AutoDAIS (K=8)

AutoDAIS (K=128)

AutoDiagonalNormal

NUTS

(continues on next page)
matplotlib.use("Agg")  # noqa: E402

# squared exponential kernel

def kernel(X, Z, length, jitter=1.0e-6):
    deltaXsq = jnp.power((X[:, None] - Z) / length, 2.0)
    k = jnp.exp(-0.5 * deltaXsq) + jitter * jnp.eye(X.shape[0])
    return k

def model(X, Y, length=0.2):
    # compute kernel
    k = kernel(X, X, length)

    # sample from gaussian process prior
    f = numpyro.sample("f",
        dist.MultivariateNormal(loc=jnp.zeros(X.shape[0]), covariance_matrix=k),
    )
    # we use a non-standard link function to induce extra non-gaussianity
    numpyro.sample("obs", dist.Bernoulli(logits=jnp.power(f, 3.0)), obs=Y)

# create artificial binary classification dataset

def get_data(N=16):
    np.random.seed(0)
    X = np.linspace(-1, 1, N)
    Y = X + 0.2 * np.power(X, 3.0) + 0.5 * np.power(0.5 + X, 2.0) * np.sin(4.0 * X)
    Y -= np.mean(Y)
    Y /= np.std(Y)
    Y = np.random.binomial(1, expit(Y))

    assert X.shape == (N,)
    assert Y.shape == (N,)

    return X, Y

# helper function for running SVI with a particular autoguide

def run_svi(rng_key, X, Y, guide_family="AutoDiagonalNormal", K=8):
    assert guide_family in ["AutoDiagonalNormal", "AutoDAIS"]

    if guide_family == "AutoDAIS":
        guide = autoguide.AutoDAIS(model, K=K, eta_init=0.02, eta_max=0.5)
        step_size = 5e-4
    elif guide_family == "AutoDiagonalNormal":
        guide = autoguide.AutoDiagonalNormal(model)
        step_size = 3e-3

    optimizer = numpyro.optim.Adam(step_size=step_size)
    svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
svi_result = svi.run(rng_key, args.num_svi_steps, X, Y)
params = svi_result.params

final_elbo = -Trace_ELBO(num_particles=1000).loss(
    rng_key, params, model, guide, X, Y
)

guide_name = guide_family
if guide_family == "AutoDAIS":
    guide_name += "-{}".format(K)

print("{} final elbo: {:.2f}".format(guide_name, final_elbo))
return guide.sample_posterior(
    random.PRNGKey(1), params, sample_shape=(args.num_samples,)
)

# helper function for running mcmc
def run_nuts(mcmc_key, args, X, Y):
    mcmc = MCMC(NUTS(model), num_warmup=args.num_warmup, num_samples=args.num_samples)
    mcmc.run(mcmc_key, X, Y)
    mcmc.print_summary()
    return mcmc.get_samples()

def main(args):
    X, Y = get_data()

    rng_keys = random.split(random.PRNGKey(0), 4)

    # run SVI with an AutoDAIS guide for two values of K
dais8_samples = run_svi(rng_keys[1], X, Y, guide_family="AutoDAIS", K=8)
dais128_samples = run_svi(rng_keys[2], X, Y, guide_family="AutoDAIS", K=128)

    # run SVI with an AutoDiagonalNormal guide
meanfield_samples = run_svi(rng_keys[3], X, Y, guide_family="AutoDiagonalNormal")

    # run MCMC inference
nuts_samples = run_nuts(rng_keys[0], args, X, Y)

    # make 2d density plots of the (f_0, f_1) marginal posterior
if args.num_samples >= 1000:
    sns.set_style("white")

    coord1, coord2 = 0, 1

    fig, axes = plt.subplots(
        2, 2, sharex=True, figsize=(6, 6), constrained_layout=True
    )

    xlim = (-3, 3)
ylim = (-3, 3)

```python
def add_fig(samples, title, ax):
    sns.kdeplot(x=samples["f"][:, coord1], y=samples["f"][:, coord2], ax=ax)
    ax.set(title=title, xlim=xlim, ylim=ylim)

add_fig(dais8_samples, "AutoDAIS (K=8)", axes[0][0])
add_fig(dais128_samples, "AutoDAIS (K=128)", axes[0][1])
add_fig(meanfield_samples, "AutoDiagonalNormal", axes[1][0])
add_fig(nuts_samples, "NUTS", axes[1][1])

plt.savefig("dais_demo.png")
```

```python
if __name__ == "__main__":
    parser = argparse.ArgumentParser("Usage example for AutoDAIS guide.")
    parser.add_argument("--num-svi-steps", type=int, default=80 * 1000)
    parser.add_argument("--num-warmup", type=int, default=2000)
    parser.add_argument("--num-samples", type=int, default=10 * 1000)
    parser.add_argument("--device", default="cpu", type=str, choices=["cpu", "gpu"])

    args = parser.parse_args()

    enable_x64()
    numpyro.set_platform(args.device)

    main(args)
```
EXAMPLE: SPARSE REGRESSION

We demonstrate how to do (fully Bayesian) sparse linear regression using the approach described in [1]. This approach is particularly suitable for situations with many feature dimensions (large P) but not too many datapoints (small N). In particular we consider a quadratic regressor of the form:

\[ f(X) = \text{constant} + \sum_i \theta_i X_i + \sum_{i<j} \theta_{ij} X_i X_j + \text{observation noise} \]

References:

```python
import argparse
import itertools
import os
import time
import numpy as np
from jax import vmap
import jax.numpy as jnp
import jax.random as random
from jax.scipy.linalg import cho_factor, cho_solve, solve_triangular
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

def dot(X, Z):
    return jnp.dot(X, Z[..., None])[..., 0]

# The kernel that corresponds to our quadratic regressor.

def kernel(X, Z, eta1, eta2, c, jitter=1.0e-4):
    etasq = jnp.square(eta1)
    eta2sq = jnp.square(eta2)
    k1 = 0.5 * eta2sq * jnp.square(1.0 + dot(X, Z))
    k2 = -0.5 * eta2sq * dot(jnp.square(X), jnp.square(Z))
    k3 = (etasq - eta2sq) * dot(X, Z)
    k4 = jnp.square(c) - 0.5 * eta2sq
    if X.shape == Z.shape:
```

(continues on next page)
k4 += jitter * jnp.eye(X.shape[0])
return k1 + k2 + k3 + k4

# Most of the model code is concerned with constructing the sparsity inducing prior.
def model(X, Y, hypers):
    S, P, N = hypers["expected_sparsity"], X.shape[1], X.shape[0]

    sigma = numpyro.sample("sigma", dist.HalfNormal(hypers["alpha3"]))
    phi = sigma * (S / jnp.sqrt(N)) / (P - S)
    etal1 = numpyro.sample("etal1", dist.HalfCauchy(phi))

    msq = numpyro.sample("msq", dist.InverseGamma(hypers["alpha1"], hypers["beta1"]))
    xisq = numpyro.sample("xisq", dist.InverseGamma(hypers["alpha2"], hypers["beta2"]))

    eta2 = jnp.square(etal1) * jnp.sqrt(xisq) / msq
    lam = numpyro.sample("lambda", dist.HalfCauchy(jnp.ones(P)))
    kappa = jnp.sqrt(msq) * lam / jnp.sqrt(msq + jnp.square(etal1 * lam))

    # compute kernel
    kX = kappa * X
    k = kernel(kX, kX, etal1, eta2, hypers["c"] + sigma**2 * jnp.eye(N))
    assert k.shape == (N, N)

    # sample Y according to the standard gaussian process formula
    numpyro.sample("Y", dist.MultivariateNormal(loc=jnp.zeros(X.shape[0]), covariance_matrix=k), obs=Y,
    )

# Compute the mean and variance of coefficient theta_i (where i = dimension) for a
# MCMC sample of the kernel hyperparameters (etal1, xisq, ...).
# Compare to theorem 5.1 in reference [1].
def compute_singleton_mean_variance(X, Y, dimension, msq, lam, etal1, xisq, c, sigma):
    P, N = X.shape[1], X.shape[0]

    probe = jnp.zeros((2, P))
    probe = probe.at[:, dimension].set(jnp.array([1.0, -1.0]))

    eta2 = jnp.square(etal1) * jnp.sqrt(xisq) / msq
    kappa = jnp.sqrt(msq) * lam / jnp.sqrt(msq + jnp.square(etal1 * lam))

    kX = kappa * X
    kprobe = kappa * probe

    k_xx = kernel(kX, kX, etal1, eta2, c) + sigma**2 * jnp.eye(N)
    k_xx_inv = jnp.linalg.inv(k_xx)
    k_probeX = kernel(kprobe, kX, etal1, eta2, c)
    k_prbprb = kernel(kprobe, kprobe, etal1, eta2, c)
vec = jnp.array([0.50, -0.50])
mu = jnp.matmul(k_probeX, jnp.matmul(k_xx_inv, Y))
mu = jnp.dot(mu, vec)

var = k_prbprb - jnp.matmul(k_probeX, jnp.matmul(k_xx_inv, jnp.transpose(k_probeX)))
var = jnp.matmul(var, vec)
var = jnp.dot(var, vec)

return mu, var

# Compute the mean and variance of coefficient theta_ij for a MCMC sample of the
# kernel hyperparameters (eta1, xisq, ...). Compare to theorem 5.1 in reference [1].
def compute_pairwise_mean_variance(X, Y, dim1, dim2, msq, lam, eta1, xisq, c, sigma):
    P, N = X.shape[1], X.shape[0]

    probe = jnp.zeros((4, P))
    probe = probe.at[:, dim1].set(jnp.array([1.0, 1.0, -1.0, -1.0]))
    probe = probe.at[:, dim2].set(jnp.array([1.0, -1.0, 1.0, -1.0]))

    eta2 = jnp.square(eta1) * jnp.sqrt(xisq) / msq
    kappa = jnp.sqrt(msq) * lam / jnp.sqrt(msq + jnp.square(eta1 * lam))

    kX = kappa * X
    kprobe = kappa * probe

    k_xx = kernel(kX, kX, eta1, eta2, c) + sigma**2 * jnp.eye(N)
    k_xx_inv = jnp.linalg.inv(k_xx)
    k_probeX = kernel(kprobe, kX, eta1, eta2, c)
    k_prbprb = kernel(kprobe, kprobe, eta1, eta2, c)

    vec = jnp.array([0.25, -0.25, -0.25, 0.25])
    mu = jnp.matmul(k_probeX, jnp.matmul(k_xx_inv, Y))
    mu = jnp.dot(mu, vec)

    var = k_prbprb - jnp.matmul(k_probeX, jnp.matmul(k_xx_inv, jnp.transpose(k_probeX)))
    var = jnp.matmul(var, vec)
    var = jnp.dot(var, vec)

    return mu, var

# Sample coefficients theta from the posterior for a given MCMC sample.
# The first P returned values are {theta_1, theta_2, ...., theta_P}, while
# the remaining values are {theta_ij} for i,j in the list `active_dims`,
# sorted so that i < j.
def sample_theta_space(X, Y, active_dims, msq, lam, eta1, xisq, c, sigma):
    P, N, M = X.shape[1], X.shape[0], len(active_dims)
    num_coefficients = P + M * (M - 1) // 2
probe = jnp.zeros((2 * P + 2 * M * (M - 1), P))
vec = jnp.zeros((num_coefficients, 2 * P + 2 * M * (M - 1)))
start1 = 0
start2 = 0

for dim in range(P):
    probe = probe.at[start1 : start1 + 2, dim].set(jnp.array([1.0, -1.0]))
    vec = vec.at[start2, start1 : start1 + 2].set(jnp.array([0.5, -0.5]))
    start1 += 2
    start2 += 1

for dim1 in active_dims:
    for dim2 in active_dims:
        if dim1 >= dim2:
            continue
        probe = probe.at[start1 : start1 + 4, dim1].set(
            jnp.array([1.0, 1.0, -1.0, -1.0]))
        probe = probe.at[start1 : start1 + 4, dim2].set(
            jnp.array([1.0, -1.0, 1.0, -1.0]))
        vec = vec.at[start2, start1 : start1 + 4].set(
            jnp.array([0.25, -0.25, -0.25, 0.25]))
        start1 += 4
        start2 += 1

eta2 = jnp.square(eta1) * jnp.sqrt(xisq) / msq
kappa = jnp.sqrt(msq) * lam / jnp.sqrt(msq + jnp.square(eta1 * lam))
kX = kappa * X
kprobe = kappa * probe
k_xx = kernel(kX, kX, eta1, eta2, c) + sigma**2 * jnp.eye(N)
L = cho_factor(k_xx, lower=True)[0]
k_probeX = kernel(kprobe, kX, eta1, eta2, c)
k_prbprb = kernel(kprobe, kprobe, eta1, eta2, c)

mu = jnp.matmul(k_probeX, cho_solve((L, True), Y))
mu = jnp.sum(mu * vec, axis=-1)

Linv_k_probeX = solve_triangular(L, jnp.transpose(k_probeX), lower=True)
covar = k_prbprb - jnp.matmul(jnp.transpose(Linv_k_probeX), Linv_k_probeX)
covar = jnp.matmul(vec, jnp.matmul(covar, jnp.transpose(vec)))

# sample from N(mu, covar)
L = jnp.linalg.cholesky(covar)
sample = mu + jnp.matmul(L, np.random.randn(num_coefficients))

return sample
# Helper function for doing HMC inference

def run_inference(model, args, rng_key, X, Y, hypers):
    start = time.time()
    kernel = NUTS(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, X, Y, hypers)
    mcmc.print_summary()
    print("MCMC elapsed time:", time.time() - start)
    return mcmc.get_samples()

# Get the mean and variance of a gaussian mixture

def gaussian_mixture_stats(mus, variances):
    mean_mu = jnp.mean(mus)
    mean_var = jnp.mean(variances) + jnp.mean(jnp.square(mus)) - jnp.square(mean_mu)
    return mean_mu, mean_var

# Create artificial regression dataset where only S out of P feature
# dimensions contain signal and where there is a single pairwise interaction
# between the first and second dimensions.

def get_data(N=20, S=2, P=10, sigma_obs=0.05):
    assert S < P and P > 1 and S > 0
    np.random.seed(0)
    X = np.random.randn(N, P)
    # generate S coefficients with non-negligible magnitude
    W = 0.5 + 2.5 * np.random.rand(S)
    # generate data using the S coefficients and a single pairwise interaction
    Y = (
        np.sum(X[:, :S] * W, axis=-1)
        + X[:, 0] * X[:, 1]
        + sigma_obs * np.random.randn(N)
    )
    Y -= jnp.mean(Y)
    Y_std = jnp.std(Y)
    assert X.shape == (N, P)
    assert Y.shape == (N,)

    return X, Y / Y_std, W / Y_std, 1.0 / Y_std

# Helper function for analyzing the posterior statistics for coefficient theta_i

def analyze_dimension(samples, X, Y, dimension, hypers):
    vmap_args = (  
    (continues on next page)
samples["msq"],
samples["lambda"],
samples["eta1"],
samples["xisq"],
samples["sigma"],
)
mus, variances = vmap(
    lambda msq, lam, eta1, xisq, sigma: compute_singleton_mean_variance(
        X, Y, dimension, msq, lam, eta1, xisq, hypers["c"], sigma
    )
)(*vmap_args)
mean, variance = gaussian_mixture_stats(mus, variances)
std = jnp.sqrt(variance)
return mean, std

# Helper function for analyzing the posterior statistics for coefficient theta_ij
def analyze_pair_of_dimensions(samples, X, Y, dim1, dim2, hypers):
vmap_args = (
    samples["msq"],
    samples["lambda"],
    samples["eta1"],
    samples["xisq"],
    samples["sigma"],
)
mus, variances = vmap(
    lambda msq, lam, eta1, xisq, sigma: compute_pairwise_mean_variance(
        X, Y, dim1, dim2, msq, lam, eta1, xisq, hypers["c"], sigma
    )
)(*vmap_args)
mean, variance = gaussian_mixture_stats(mus, variances)
std = jnp.sqrt(variance)
return mean, std

def main(args):
    X, Y, expected_thetas, expected_pairwise = get_data(
        N=args.num_data, P=args.num_dimensions, S=args.active_dimensions
    )

    # setup hyperparameters
    hypers = {
        "expected_sparsity": max(1.0, args.num_dimensions / 10),
        "alpha1": 3.0,
        "beta1": 1.0,
        "alpha2": 3.0,
        "beta2": 1.0,
        "alpha3": 1.0,
        "c": 1.0,
    }

    # do inference
    (continues on next page)
rng_key = random.PRNGKey(0)
samples = run_inference(model, args, rng_key, X, Y, hypers)

# compute the mean and square root variance of each coefficient \( \theta_i \)
means, stds = vmap(
    lambda dim: analyze_dimension(samples, X, Y, dim, hypers))(
    jnp.arange(args.num_dimensions)
)

print(
    "Coefficients \( \theta_1 \) to \( \theta_{%d} \) used to generate the data:"
    % args.active_dimensions,
    expected_thetas,
)
print(
    "The single quadratic coefficient \( \theta_{\{1,2\}} \) used to generate the data:",
    expected_pairwise,
)
active_dimensions = []
for dim, (mean, std) in enumerate(zip(means, stds)):
    # we mark the dimension as inactive if the interval \([\text{mean} - 3 \times \text{std}, \text{mean} + 3 \times \text{std}]\) contains zero
    lower, upper = mean - 3.0 * std, mean + 3.0 * std
    inactive = "inactive" if lower < 0.0 and upper > 0.0 else "active"
    if inactive == "active":
        active_dimensions.append(dim)
    print(
        "[dimension %02d/%02d] %s: \(t%.2e \pm %.2e\)"
        % (dim + 1, args.num_dimensions, inactive, mean, std)
    )

print(
    "Identified a total of %d active dimensions; expected %d."
    % (len(active_dimensions), args.active_dimensions)
)

# Compute the mean and square root variance of coefficients \( \theta_{ij} \) for \( i, j \) active dimensions.
# Note that the resulting numbers are only meaningful for \( i \neq j \).
if len(active_dimensions) > 0:
    dim_pairs = jnp.array(
        list(itertools.product(active_dimensions, active_dimensions))
    )
    means, stds = vmap(
        lambda dim_pair: analyze_pair_of_dimensions(
            samples, X, Y, dim_pair[0], dim_pair[1], hypers
        ))(dim_pairs)
    for dim_pair, mean, std in zip(dim_pairs, means, stds):
        dim1, dim2 = dim_pair
        if dim1 >= dim2:
            continue
lower, upper = mean - 3.0 * std, mean + 3.0 * std
if not (lower < 0.0 and upper > 0.0):
    format_str = "Identified pairwise interaction between dimensions %d and
˓→%d: %.2e +- %.2e"
    print(format_str % (dim1 + 1, dim2 + 1, mean, std))

    # Draw a single sample of coefficients theta from the posterior, where we return
˓→all singleton
    # coefficients theta_i and pairwise coefficients theta_ij for i, j active
˓→dimensions. We use the
    # final MCMC sample obtained from the HMC sampler.
    thetas = sample_theta_space(
        X,
        Y,
        active_dimensions,
        samples["msq"][-1],
        samples["lambda"][-1],
        samples["eta1"][-1],
        samples["xisq"][-1],
        hypers["c"],
        samples["sigma"][-1],
    )
    print("Single posterior sample theta:\n", thetas)

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Gaussian Process example")
    parser.add_argument("--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=500, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--num-dimensions", nargs="?", default=100, type=int)
    parser.add_argument("--active-dimensions", nargs="?", default=20, type=int)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
We demonstrate how to use NUTS to do sparse regression using the Horseshoe prior [1] for both continuous- and binary-valued responses. For a more complex modeling and inference approach that also supports quadratic interaction terms in a way that is efficient in high dimensions see examples/sparse_regression.py.

References:
[1] “Handling Sparsity via the Horseshoe,”
    Carlos M. Carvalho, Nicholas G. Polson, James G. Scott.

```python
import argparse
import os
import time

import numpy as np
from scipy.special import expit
import jax.numpy as jnp
import jax.random as random
import numpyro
from numpyro.diagnostics import summary
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

# regression model with continuous-valued outputs/responses
def model_normal_likelihood(X, Y):
    D_X = X.shape[1]

    # sample from horseshoe prior
    lambdas = numpyro.sample("lambdas", dist.HalfCauchy(jnp.ones(D_X)))
    tau = numpyro.sample("tau", dist.HalfCauchy(jnp.ones(1)))

    # note that in practice for a normal likelihood we would probably want to
    # integrate out the coefficients (as is done for example in sparse_regression.py).
    # however, this trick wouldn't be applicable to other likelihoods
    # (e.g. bernoulli, see below) so we don't make use of it here.
    unscaled_betas = numpyro.sample("unscaled_betas", dist.Normal(0.0, jnp.ones(D_X)))
    scaled_betas = numpyro.deterministic("betas", tau * lambdas * unscaled_betas)

    # compute mean function using linear coefficients
```

(continues on next page)
mean_function = jnp.dot(X, scaled_betas)

prec_obs = numpyro.sample("prec_obs", dist.Gamma(3.0, 1.0))
sigma_obs = 1.0 / jnp.sqrt(prec_obs)

# observe data
numpyro.sample("Y", dist.Normal(mean_function, sigma_obs), obs=Y)

# regression model with binary-valued outputs/responses
def model_bernoulli_likelihood(X, Y):
    D_X = X.shape[1]

    # sample from horseshoe prior
    lambdas = numpyro.sample("lambdas", dist.HalfCauchy(jnp.ones(D_X)))
    tau = numpyro.sample("tau", dist.HalfCauchy(jnp.ones(1)))

    # note that this reparameterization (i.e. coordinate transformation) improves
    # posterior geometry and makes NUTS sampling more efficient
    unscaled_betas = numpyro.sample("unscaled_betas", dist.Normal(0.0, jnp.ones(D_X)))
    scaled_betas = numpyro.deterministic("betas", tau * lambdas * unscaled_betas)

    # compute mean function using linear coefficients
    mean_function = jnp.dot(X, scaled_betas)

    # observe data
    numpyro.sample("Y", dist.Bernoulli(logits=mean_function), obs=Y)

# helper function for HMC inference
def run_inference(model, args, rng_key, X, Y):
    start = time.time()
    kernel = NUTS(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )

    mcmc.run(rng_key, X, Y)
    mcmc.print_summary(exclude_deterministic=False)

    samples = mcmc.get_samples()
    summary_dict = summary(samples, group_by_chain=False)

    print("\nMCMC elapsed time:", time.time() - start)

    return summary_dict

(continues on next page)
def get_data(N=50, D_X=3, sigma_obs=0.05, response="continuous"):
    assert response in ["continuous", "binary"]
    assert D_X >= 3
    np.random.seed(0)
    X = np.random.randn(N, D_X)

    # the response only depends on X_0, X_1, and X_2
    W = np.array([2.0, -1.0, 0.50])
    Y = jnp.dot(X[:, :3], W)
    Y -= jnp.mean(Y)

    if response == "continuous":
        Y += sigma_obs * np.random.randn(N)
    elif response == "binary":
        Y = np.random.binomial(1, expit(Y))

    assert X.shape == (N, D_X)
    assert Y.shape == (N,)

    return X, Y

def main(args):
    N, D_X = args.num_data, 32

    print("[Experiment with continuous-valued responses]")
    # first generate and analyze data with continuous-valued responses
    X, Y = get_data(N=N, D_X=D_X, response="continuous")

    # do inference
    rng_key, rng_key_predict = random.split(random.PRNGKey(0))
    summary = run_inference(model_normal_likelihood, args, rng_key, X, Y)

    print("Posterior median over lambdas (leading 5 dimensions):")
    print(summary["lambdas"]["median"][:5])
    print("Posterior mean over betas (leading 5 dimensions):")
    print(summary["betas"]["mean"][:5])

    print("[Experiment with binary-valued responses]")
    # note we use more data for the case of binary-valued responses,
    # since each response carries less information than a real number
    X, Y = get_data(N=4 * N, D_X=D_X, response="binary")

    # do inference
    rng_key, rng_key_predict = random.split(random.PRNGKey(0))
    summary = run_inference(model_bernoulli_likelihood, args, rng_key, X, Y)
# lambda should only be large for the first 3 dimensions, which correspond to relevant covariates (see get_data)
print("Posterior median over lambdas (leading 5 dimensions):")
print(summary["lambdas"]["median"][5])
print("Posterior mean over betas (leading 5 dimensions):")
print(summary["betas"]["mean"][5])

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
parser = argparse.ArgumentParser(description="Horseshoe regression example")
parser.add_argument("-n", "--num-samples", nargs="?", default=2000, type=int)
parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
parser.add_argument("--num-chains", nargs="?", default=1, type=int)
parser.add_argument("--num-data", nargs="?", default=100, type=int)
parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
args = parser.parse_args()
numpyro.set_platform(args.device)
numpyro.set_host_device_count(args.num_chains)

main(args)
EXAMPLE: PROPORTION TEST

You are managing a business and want to test if calling your customers will increase their chance of making a purchase. You get 100,000 customer records and call roughly half of them and record if they make a purchase in the next three months. You do the same for the half that did not get called. After three months, the data is in - did calling help?

This example answers this question by estimating a logistic regression model where the covariates are whether the customer got called and their gender. We place a multivariate normal prior on the regression coefficients. We report the 95% highest posterior density interval for the effect of making a call.

```python
import argparse
import os

from jax import random
import jax.numpy as jnp
from jax.scipy.special import expit

import numpyro
from numpyro.diagnostics import hpdi
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

def make_dataset(rng_key) -> tuple[jnp.ndarray, jnp.ndarray]:
    """
    Make simulated dataset where potential customers who get a sales calls have ~2% higher chance of making another purchase.
    """
    key1, key2, key3 = random.split(rng_key, 3)

    num_calls = 51342
    num_no_calls = 48658

    made_purchase_got_called = dist.Bernoulli(0.084).sample(
        key1, sample_shape=(num_calls,)
    )
    made_purchase_no_calls = dist.Bernoulli(0.061).sample(
        key2, sample_shape=(num_no_calls,)
    )

    made_purchase = jnp.concatenate([made_purchase_got_called, made_purchase_no_calls])

    is_female = dist.Bernoulli(0.5).sample(
```
key3, sample_shape=(num_calls + num_no_calls,)
)
got_called = jnp.concatenate([jnp.ones(num_calls), jnp.zeros(num_no_calls)])
design_matrix = jnp.hstack(
    [jnp.ones((num_no_calls + num_calls, 1)),
     got_called.reshape(-1, 1),
     is_female.reshape(-1, 1),
    ]
)
return design_matrix, made_purchase

def model(design_matrix: jnp.ndarray, outcome: jnp.ndarray = None) -> None:
    """
    Model definition: Log odds of making a purchase is a linear combination
    of covariates. Specify a Normal prior over regression coefficients.
    :param design_matrix: Covariates. All categorical variables have been one-hot
    encoded.
    :param outcome: Binary response variable. In this case, whether or not the
    customer made a purchase.
    """
    beta = numpyro.sample("coefficients",
        dist.MultivariateNormal(
            loc=0.0, covariance_matrix=jnp.eye(design_matrix.shape[1])
        ),
    )
    logits = design_matrix.dot(beta)
    with numpyro.plate("data", design_matrix.shape[0]):
        numpyro.sample("obs", dist.Bernoulli(logits=logits), obs=outcome)

def print_results(coef: jnp.ndarray, interval_size: float = 0.95) -> None:
    """
    Print the confidence interval for the effect size with interval_size
    probability mass.
    """
    baseline_response = expit(coef[:, 0])
    response_with_calls = expit(coef[:, 0] + coef[:, 1])

    impact_on_probability = hpdi(response_with_calls - baseline_response, prob=interval_size)

    effect_of_gender = hpdi(coef[:, 2], prob=interval_size)

    print(
There is a \(\{\text{interval}\_\text{size} \times 100\}\)% probability that calling customers increase the chance they'll make a purchase by \(\{(100 \times \text{impact}\_\text{on}\_\text{probability}[0]):.2\}\) to \(\{(100 \times \text{impact}\_\text{on}\_\text{probability}[1]):.2\}\) percentage points.

print(
    f"There is a \{\text{interval}\_\text{size} \times 100\}% probability the effect of gender on the log-odds of conversion lies in the interval \((\text{effect}\_\text{of}\_\text{gender}[0]:.2f, \text{effect}\_\text{of}\_\text{gender}[1]:.2f)\)."  
    " Since this interval contains 0, we can conclude gender does not impact the conversion rate."  
)

def run_inference(
    design_matrix: jnp.ndarray,  
    outcome: jnp.ndarray,  
    rng_key: jnp.ndarray,  
    num_warmup: int,  
    num_samples: int,  
    num_chains: int,  
    interval_size: float = 0.95,
) -> None:
    """
    Estimate the effect size.
    """
    kernel = NUTS(model)
    mcmc = MCMC(
        kernel,
        num_warmup=num_warmup,
        num_samples=num_samples,
        num_chains=num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, design_matrix, outcome)
    # 0th column is intercept (not getting called)
    # 1st column is effect of getting called
    # 2nd column is effect of gender (should be none since assigned at random)
    coef = mcmc.get_samples()["coefficients"]
    print_results(coef, interval_size)

def main(args):
    rng_key, _ = random.split(random.PRNGKey(3))
    design_matrix, response = make_dataset(rng_key)
    run_inference(
        design_matrix,  
        response,  
        rng_key,
    )
```python
    args.num_warmup,
    args.num_samples,
    args.num_chains,
    args.interval_size,
    )

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Testing whether ")
    parser.add_argument("-n", "--num-samples", nargs="?", default=500, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1500, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--interval-size", nargs="?", default=0.95, type=float)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```
EXAMPLE: GENERALIZED LINEAR MIXED MODELS

The UCBadmit data is sourced from the study [1] of gender biased in graduate admissions at UC Berkeley in Fall 1973:

Table 1: UCBadmit dataset

<table>
<thead>
<tr>
<th>dept</th>
<th>male</th>
<th>applications</th>
<th>admit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>825</td>
<td>512</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>108</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>560</td>
<td>353</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>25</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>325</td>
<td>120</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>593</td>
<td>202</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>417</td>
<td>138</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>375</td>
<td>131</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>191</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>393</td>
<td>94</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>373</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>341</td>
<td>24</td>
</tr>
</tbody>
</table>

This example replicates the multilevel model $m_{ghmm5}$ at [3], which is used to evaluate whether the data contain evidence of gender biased in admissions across departments. This is a form of Generalized Linear Mixed Models for binomial regression problem, which models

- varying intercepts across departments,
- varying slopes (or the effects of being male) across departments,
- correlation between intercepts and slopes,

and uses non-centered parameterization (or whitening).

A more comprehensive explanation for binomial regression and non-centered parameterization can be found in Chapter 10 (Counting and Classification) and Chapter 13 (Adventures in Covariance) of [2].

References:

import argparse
import os
import matplotlib.pyplot as plt
import numpy as np
from jax import random
import jax.numpy as jnp
from jax.scipy.special import expit
import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import UCBADMIT, load_dataset
from numpyro.infer import MCMC, NUTS, Predictive

def glmm(dept, male, applications, admit=None):
    v_mu = numpyro.sample("v_mu", dist.Normal(0, jnp.array([4.0, 1.0])))
    sigma = numpyro.sample("sigma", dist.HalfNormal(jnp.ones(2)))
    L_Rho = numpyro.sample("L_Rho", dist.LKJCholesky(2, concentration=2))
    scale_tril = sigma[..., jnp.newaxis] * L_Rho
    # non-centered parameterization

    (continues on next page)
num_dept = len(np.unique(dept))
z = numpyro.sample("z", dist.Normal(jnp.zeros((num_dept, 2)), 1))
v = jnp.dot(scale_tril, z.T).

logits = v_mu[0] + v[dept, 0] + (v_mu[1] + v[dept, 1]) * male
if admit is None:
    # we use a Delta site to record probs for predictive distribution
    probs = expit(logits)
    numpyro.sample("probs", dist.Delta(probs), obs=probs)
    numpyro.sample("admit", dist.Binomial(applications, logits=logits), obs=admit)

def run_inference(dept, male, applications, admit, rng_key, args):
    kernel = NUTS(glmm)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, dept, male, applications, admit)
    return mcmc.get_samples()

def print_results(header, preds, dept, male, probs):
    columns = ["Dept", "Male", "ActualProb", "Pred(p25)", "Pred(p50)", "Pred(p75)"
    header_format = "{:10} {:10} {:10} {:10} {:10} {:10}"
    row_format = "{:10.0f} {:10.0f} {:10.2f} {:10.2f} {:10.2f} {:10.2f}"
    quantiles = jnp.quantile(preds, jnp.array([0.25, 0.5, 0.75]), axis=0)
    print("\n", header, "\n")
    print(header_format.format(*columns))
    for i in range(len(dept)):
        print(row_format.format(dept[i], male[i], probs[i], *quantiles[:, i]), "\n")

def main(args):
    _, fetch_train = load_dataset(UCBADMIT, split="train", shuffle=False)
    dept, male, applications, admit = fetch_train()
    rng_key, rng_key_predict = random.split(random.PRNGKey(1))
    zs = run_inference(dept, male, applications, admit, rng_key, args)
    pred_probs = Predictive(glmm, zs)(rng_key_predict, dept, male, applications)["probs"
    header = "=" * 30 + "glmm - TRAIN" + "=" * 30
    print_results(header, pred_probs, dept, male, admit / applications)

    # make plots
    fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)
    ax.plot(range(1, 13), admit / applications, "o", ms=7, label="actual rate")
    ax.errorbar(...

(continues on next page)
range(1, 13),
jnp.mean(pred_probs, 0),
jnp.std(pred_probs, 0),
fmt="o",
c="k",
mfc="none",
ms=7,
elinewidth=1,
label=r"mean \pm std",
)
ax.plot(range(1, 13), jnp.percentile(pred_probs, 5, 0), "k+)
ax.plot(range(1, 13), jnp.percentile(pred_probs, 95, 0), "k+)

ax.set(
   xlabel="cases",
ylabel="admit rate",
title="Posterior Predictive Check with 90% CI",
)
ax.legend()

plt.savefig("ucbadmit_plot.pdf")

if __name__ == "__main__":
   assert numpyro.__version__.startswith("0.15.2")
   parser = argparse.ArgumentParser(
      description="UCBadmit gender discrimination using HMC"
   )
   parser.add_argument("-n", "--num-samples", nargs="?", default=2000, type=int)
   parser.add_argument("--num-warmup", nargs="?", default=500, type=int)
   parser.add_argument("--num-chains", nargs="?", default=1, type=int)
   parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".")
   args = parser.parse_args()
   numpyro.set_platform(args.device)
   numpyro.set_host_device_count(args.num_chains)
   main(args)
EXAMPLE: HIDDEN MARKOV MODEL

In this example, we will follow [1] to construct a semi-supervised Hidden Markov Model for a generative model with observations being words and latent variables being categories. Instead of automatically marginalizing all discrete latent variables (as in [2]), we will use the “forward algorithm” (which exploits the conditional independence of a Markov model - see [3]) to iteratively do this marginalization.

The semi-supervised problem is chosen instead of an unsupervised one because it is hard to make the inference work for an unsupervised model (see the discussion [4]). On the other hand, this example also illustrates the usage of JAX’s lax.scan primitive. The primitive will greatly improve compiling for the model.

References:

import argparse
import os
import time

import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import gaussian_kde
from jax import lax, random
import jax.numpy as jnp
from jax.scipy.special import logsumexp
import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

def simulate_data(
    rng_key, num_categories, num_words, num_supervised_data, num_unsupervised_data
):
    rng_key, rng_key_transition, rng_key_emission = random.split(rng_key, 3)

    transition_prior = jnp.ones(num_categories)
emission_prior = jnp.repeat(0.1, num_words)

transition_prob = dist.Dirichlet(transition_prior).sample(
    key=rng_key_transition, sample_shape=(num_categories,)
)
emission_prob = dist.Dirichlet(emission_prior).sample(
    key=rng_key_emission, sample_shape=(num_categories,)
)

start_prob = jnp.repeat(1.0 / num_categories, num_categories)
categories, words = [], []
for t in range(num_supervised_data + num_unsupervised_data):
    rng_key, rng_key_transition, rng_key_emission = random.split(rng_key, 3)
    if t == 0 or t == num_supervised_data:
        category = dist.Categorical(start_prob).sample(key=rng_key_transition)
    else:
        category = dist.Categorical(transition_prob[category]).sample(
            key=rng_key_transition
        )
    word = dist.Categorical(emission_prob[category]).sample(key=rng_key_emission)
    categories.append(category)
    words.append(word)

# split into supervised data and unsupervised data
categories, words = jnp.stack(categories), jnp.stack(words)
supervised_categories = categories[:num_supervised_data]
supervised_words = words[:num_supervised_data]
unsupervised_words = words[num_supervised_data:]
return (
    transition_prior,
    emission_prior,
    transition_prob,
    emission_prob,
    supervised_categories,
    supervised_words,
    unsupervised_words,
)

def forward_one_step(prev_log_prob, curr_word, transition_log_prob, emission_log_prob):
    log_prob_tmp = jnp.expand_dims(prev_log_prob, axis=1) + transition_log_prob
    log_prob = log_prob_tmp + emission_log_prob[:, curr_word]
    return logsumexp(log_prob, axis=0)

def forward_log_prob(
    init_log_prob, words, transition_log_prob, emission_log_prob, unroll_loop=False
):
    # Note: The following naive implementation will make it very slow to compile
    # and do inference. So we use lax.scan instead.
    #
    # >>> log_prob = init_log_prob
# >>> for word in words:
# ...   log_prob = forward_one_step(log_prob, word, transition_log_prob, emission_log_prob)

```python
def scan_fn(log_prob, word):
    return
    forward_one_step(log_prob, word, transition_log_prob, emission_log_prob),
    None,  # we don't need to collect during scan

if unroll_loop:
    log_prob = init_log_prob
    for word in words:
        log_prob = forward_one_step(
            log_prob, word, transition_log_prob, emission_log_prob
        )
else:
    log_prob, _ = lax.scan(scan_fn, init_log_prob, words)
return log_prob
```

def semi_supervised_hmm(
    transition_prior,
    emission_prior,
    supervised_categories,
    supervised_words,
    unsupervised_words,
    unroll_loop=False,
):
    num_categories, num_words = transition_prior.shape[0], emission_prior.shape[0]
    transition_prob = numpyro.sample(
        "transition_prob",
        dist.Dirichlet(jnp.broadcast_to(transition_prior, (num_categories, num_categories))
    ),
    )
    emission_prob = numpyro.sample(
        "emission_prob",
        dist.Dirichlet(jnp.broadcast_to(emission_prior, (num_categories, num_words))),
    )

    # models supervised data;
    # here we don't make any assumption about the first supervised category, in other words,
    # we place a flat/uniform prior on it.
    numpyro.sample(
        "supervised_categories",
        dist.Categorical(transition_prob[supervised_categories[:-1]]),
        obs=supervised_categories[1:],
    )
    numpyro.sample(
        "supervised_words",
        dist.Categorical(emission_prob[supervised_categories]),
    )
    (continues on next page)
obs=supervised_words,
)

# computes log prob of unsupervised data
transition_log_prob = jnp.log(transition_prob)
emission_log_prob = jnp.log(emission_prob)
init_log_prob = emission_log_prob[:, unsupervised_words[0]]
log_prob = forward_log_prob(
    init_log_prob,
    unsupervised_words[1:],
    transition_log_prob,
    emission_log_prob,
    unroll_loop,
)
log_prob = logsumexp(log_prob, axis=0, keepdims=True)
# inject log_prob to potential function
numpyro.factor("forward_log_prob", log_prob)

def print_results(posterior, transition_prob, emission_prob):
    header = semi_supervised_hmm.__name__ + " - TRAIN"
    columns = ["", "ActualProb", "Pred(p25)", "Pred(p50)", "Pred(p75)"]
    header_format = "{:>20} {:>10} {:>10} {:>10} {:>10}"
    row_format = "{:>20} {:>10.2f} {:>10.2f} {:>10.2f} {:>10.2f}"
    print("\n", "=" * 20 + header + "=" * 20, "\n")
    print(header_format.format(*columns))
    quantiles = np.quantile(posterior["transition_prob"], [0.25, 0.5, 0.75], axis=0)
    for i in range(transition_prob.shape[0]):
        for j in range(transition_prob.shape[1]):
            idx = "transition[{},{}]".format(i, j)
            print(
                row_format.format(idx, transition_prob[i, j], *quantiles[:, i, j]), "\n"
            )
    quantiles = np.quantile(posterior["emission_prob"], [0.25, 0.5, 0.75], axis=0)
    for i in range(emission_prob.shape[0]):
        for j in range(emission_prob.shape[1]):
            idx = "emission[{},{}]".format(i, j)
            print(
                row_format.format(idx, emission_prob[i, j], *quantiles[:, i, j]), "\n"
            )

def main(args):
    print("Simulating data...")
    (transition_prior,
        emission_prior,
        transition_prob,
        emission_prob,
        supervised_categories,
        )
supervised_words,
unsupervised_words,
) = simulate_data(
    random.PRNGKey(1),
    num_categories=args.num_categories,
    num_words=args.num_words,
    num_supervised_data=args.num_supervised,
    num_unsupervised_data=args.num_unsupervised,
)
print("Starting inference...")
rng_key = random.PRNGKey(2)
start = time.time()
kernel = NUTS(semi_supervised_hmm)
mcmc = MCMC(
    kernel,
    num_warmup=args.num_warmup,
    num_samples=args.num_samples,
    num_chains=args.num_chains,
    progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
)
mcmc.run(
    rng_key,
    transition_prior,
    emission_prior,
    supervised_categories,
    supervised_words,
    unsupervised_words,
    args.unroll_loop,
)
samples = mcmc.get_samples()
print_results(samples, transition_prob, emission_prob)
print("MCMC elapsed time:", time.time() - start)

# make plots
fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)
x = np.linspace(0, 1, 101)
for i in range(transition_prob.shape[0]):
    for j in range(transition_prob.shape[1]):
        ax.plot(
            x,
            gaussian_kde(samples["transition_prob":[, i, j])(x),
            label="trans_prob[{}, {}], true value = {:.2f}".format(
                i, j, transition_prob[i, j]
            ),
        )
    ax.set(
        xlabel="Probability",
        ylabel="Frequency",
        title="Transition probability posterior",
    )
ax.legend()
```python
plt.savefig("hmm_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Semi-supervised Hidden Markov Model")
    parser.add_argument("--num-categories", default=3, type=int)
    parser.add_argument("--num-words", default=10, type=int)
    parser.add_argument("--num-supervised", default=100, type=int)
    parser.add_argument("--num-unsupervised", default=500, type=int)
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=500, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--unroll-loop", action="store_true")
    parser.add_argument("--device", default="cpu", type=str, help="use "cpu" or "gpu".")
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```
EXAMPLE: HILBERT SPACE APPROXIMATION FOR GAUSSIAN PROCESSES.

This example replicates the model in the excellent case study by Aki Vehtari [1] (originally written using R and Stan). The case study uses approximate Gaussian processes [2] to model the relative number of births per day in the US from 1969 to 1988. The Hilbert space approximation is way faster than the exact Gaussian processes because it circumvents the need for inverting the covariance matrix.

The original case study also emphasizes the iterative process of building a Bayesian model, which is excellent as a pedagogical resource. Here, however, we replicate only the model that includes all components (long term trend, smooth year seasonality, slowly varying day of week effect, day of the year effect and special floating days effects).

The different components of the model are isolated into separate functions so that they can easily be reused in different contexts. To combine the multiple components into a single birthdays model, here we make use of Numpyro’s scope handler which modifies the site names of the components by adding a prefix to them. By doing this, we avoid duplication of site names within the model. Following this pattern, it is straightforward to construct the other models in [1] with the code provided here.

There are a few minor differences in the mathematical details of our models, which we had to make for the chains to mix properly or for ease of implementation. We have commented on the places where our models are different.


References:


561
import argparse
import os

import matplotlib.pyplot as plt
import pandas as pd

import jax
import jax.numpy as jnp
from tensorflow_probability.substrates import jax as tfp

import numpyro
from numpyro import deterministic, plate, sample
from numpyro.distributions as dist
from numpyro.handlers import scope
from numpyro.infer import MCMC, NUTS, init_to_median

# --- Data processing functions

def get_labour_days(dates):
    
    First monday of September
    
    is_september = dates.dt.month.eq(9)
    is_monday = dates.dt.weekday.eq(0)
    is_first_week = dates.dt.day.le(7)

    is_labour_day = is_september & is_monday & is_first_week
    is_day_after = is_labour_day.shift(fill_value=False)
return is_labour_day | is_day_after

def get_memorial_days(dates):
    """
    Last monday of May
    """
    is_may = dates.dt.month.eq(5)
    is_monday = dates.dt.weekday.eq(0)
    is_last_week = dates.dt.day.ge(25)

    is_memorial_day = is_may & is_monday & is_last_week
    is_day_after = is_memorial_day.shift(fill_value=False)

    return is_memorial_day | is_day_after

def get_thanksgiving_days(dates):
    """
    Third thursday of November
    """
    is_november = dates.dt.month.eq(11)
    is_thursday = dates.dt.weekday.eq(3)
    is_third_week = dates.dt.day.between(22, 28)

    is_thanksgiving = is_november & is_thursday & is_third_week
    is_day_after = is_thanksgiving.shift(fill_value=False)

    return is_thanksgiving | is_day_after

def get_floating_days_indicators(dates):
    def encode(x):
        return jnp.array(x.values, dtype=jnp.result_type(int))

    return {
        "labour_days_indicator": encode(get_labour_days(dates)),
        "memorial_days_indicator": encode(get_memorial_days(dates)),
        "thanksgiving_days_indicator": encode(get_thanksgiving_days(dates)),
    }

def load_data():
    URL = "https://raw.githubusercontent.com/avehtari/casestudies/master/Birthdays/data/births_usa_1969.csv"
    data = pd.read_csv(URL, sep="",
    day0 = pd.to_datetime("31-Dec-1968")
    dates = [day0 + pd.Timedelta(f"{i}d") for i in data["id"]]
    data["date"] = dates
    data["births_relative"] = data["births"] / data["births"].mean()
    return data
def make_birthdays_data_dict(data):
    x = data["id"][].values
    y = data["births_relative"][].values
    dates = data["date"]
    xsd = jnp.array((x - x.mean()) / x.std())
    ysd = jnp.array((y - y.mean()) / y.std())
    day_of_week = jnp.array((data["day_of_week"] - 1).values)
    day_of_year = jnp.array((data["day_of_year"] - 1).values)
    floating_days = get_floating_days_indicators(dates)
    period = 365.25
    w0 = x.std() * (jnp.pi * 2 / period)
    L = 1.5 * max(xsd)
    M1 = 10
    M2 = 10  # 20 in original case study
    M3 = 5
    return {
        "x": xsd,
        "day_of_week": day_of_week,
        "day_of_year": day_of_year,
        "w0": w0,
        "L": L,
        "M1": M1,
        "M2": M2,
        "M3": M3,
        **floating_days,
        "y": ysd,
    }

# --- Modelling utility functions --- #

def spectral_density(w, alpha, length):
    c = alpha * jnp.sqrt(2 * jnp.pi * length)
    e = jnp.exp(-0.5 * (length**2) * (w**2))
    return c * e

def diag_spectral_density(alpha, length, L, M):
    sqrt_eigenvalues = jnp.arange(1, 1 + M) * jnp.pi / 2 / L
    return spectral_density(sqrt_eigenvalues, alpha, length)

def eigenfunctions(x, L, M):
    """
    The first M eigenfunctions of the laplacian operator in [-L, L]
    evaluated at x. These are used for the approximation of the
    squared exponential kernel.
    """
    m1 = (jnp.pi / (2 * L)) * jnp.tile(L + x[:, None], M)
    m2 = jnp.diag(jnp.linspace(1, M, num=M))
num = jnp.sin(m1 @ m2)
den = jnp.sqrt(L)
return num / den

def modified_bessel_first_kind(v, z):
    v = jnp.asarray(v, dtype=float)
    return jnp.exp(jnp.abs(z)) * tfp.math.bessel_ive(v, z)

def diag_spectral_density_periodic(alpha, length, M):
    ""
    Not actually a spectral density but these are used in the same way. These are simply the first M coefficients of the low rank approximation for the periodic kernel.
    ""
    a = length ** (-2)
    J = jnp.arange(0, M)
    c = jnp.where(J > 0, 2, 1)
    q2 = (c * alpha**2 / jnp.exp(a)) * modified_bessel_first_kind(J, a)
    return q2

def eigenfunctions_periodic(x, w0, M):
    ""
    Basis functions for the approximation of the periodic kernel.
    ""
    m1 = jnp.tile(w0 * x[:, None], M)
    m2 = jnp.diag(jnp.arange(M, dtype=jnp.float32))
    mw0x = m1 @ m2
    cosines = jnp.cos(mw0x)
sines = jnp.sin(mw0x)
    return cosines, sines

# --- Approximate Gaussian processes --- #
def approx_se_ncp(x, alpha, length, L, M):
    ""
    Hilbert space approximation for the squared exponential kernel in the non-centered parametrisation.
    ""
    phi = eigenfunctions(x, L, M)
    spd = jnp.sqrt(diag_spectral_density(alpha, length, L, M))
    with plate("basis", M):
        beta = sample("beta", dist.Normal(0, 1))

    f = deterministic("f", phi @ (spd * beta))
    return f

def approx_periodic_gp_ncp(x, alpha, length, w0, M):
    """
Low rank approximation for the periodic squared exponential kernel in the non-centered parametrisation.

```python
def with plate("cos_basis", M):
    beta_cos = sample("beta_cos", dist.Normal(0, 1))

    with plate("sin_basis", M - 1):
        beta_sin = sample("beta_sin", dist.Normal(0, 1))

    # The first eigenfunction for the sine component
    # is zero, so the first parameter wouldn't contribute to the approximation.
    # We set it to zero to identify the model and avoid divergences.
    zero = jnp.array([0.0])
    beta_sin = jnp.concatenate((zero, beta_sin))

    f = deterministic("f", cosines @ (q2 * beta_cos) + sines @ (q2 * beta_sin))
    return f
```

# --- Components of the Birthdays model --- #

```python
def trend_gp(x, L, M):
    alpha = sample("alpha", dist.HalfNormal(1.0))
    length = sample("length", dist.InverseGamma(10.0, 2.0))
    f = approx_se_ncp(x, alpha, length, L, M)
    return f
```

```python
def year_gp(x, w0, M):
    alpha = sample("alpha", dist.HalfNormal(1.0))
    length = sample("length", dist.HalfNormal(0.2))  # scale=0.1 in original
    f = approx_periodic_gp_ncp(x, alpha, length, w0, M)
    return f
```

```python
def weekday_effect(day_of_week):
    with plate("plate_day_of_week", 6):
        weekday = sample("_beta", dist.Normal(0, 1))

    monday = jnp.array([-jnp.sum(weekday)])  # Monday = 0 in original
    beta = deterministic("beta", jnp.concatenate((monday, weekday)))
    return beta[day_of_week]
```

```python
def yearday_effect(day_of_year):
    slab_df = 50  # 100 in original case study
    slab_scale = 2
    scale_global = 0.1
    tau = sample("tau", dist.HalfNormal(2 * scale_global))
```

# Original uses half-t with 100df

```python
    c_aux = sample("c_aux", dist.InverseGamma(0.5 * slab_df, 0.5 * slab_df))
    c = slab_scale * jnp.sqrt(c_aux)
```

# Jan 1st: Day 0
# Feb 29th: Day 59
# Dec 31st: Day 365

```python
with plate("plate_day_of_year", 366):
    lam = sample("lam", dist.HalfCauchy(scale=1))
    lam_tilde = jnp.sqrt(c) * lam / jnp.sqrt(c + (tau * lam) ** 2)
    beta = sample("beta", dist.Normal(loc=0, scale=tau * lam_tilde))

return beta[day_of_year]
```

```python
def special_effect(indicator):
    beta = sample("beta", dist.Normal(0, 1))
    return beta * indicator
```

# --- Model --- #

```python
def birthdays_model(
    x,  
    day_of_week,  
    day_of_year,  
    memorial_days_indicator,  
    labour_days_indicator,  
    thanksgiving_days_indicator,  
    w0,  
    L,  
    M1,  
    M2,  
    M3,  
    y=None):
    intercept = sample("intercept", dist.Normal(0, 1))
    f1 = scope(trend_gp, "trend")(x, L, M1)
    f2 = scope(year_gp, "year")(x, w0, M2)
    g3 = scope(trend_gp, "week-trend")(  
        x, L, M3
    )  
    # length ~ lognormal(-1, 1) in original
    weekday = scope(weekday_effect, "week")(day_of_week)
    yearday = scope(yearday_effect, "day")(day_of_year)

    # --- special days
    memorial = scope(special_effect, "memorial")(memorial_days_indicator)
    labour = scope(special_effect, "labour")(labour_days_indicator)
    thanksgiving = scope(special_effect, "thanksgiving")(thanksgiving_days_indicator)

    day = yearday + memorial + labour + thanksgiving
    # --- Combine components
    f = deterministic("f", intercept + f1 + f2 + jnp.exp(g3) * weekday + day)
```

(continues on next page)
sigma = sample("sigma", dist.HalfNormal(0.5))

with plate("obs", x.shape[0]):
    sample("y", dist.Normal(f, sigma), obs=y)

# --- plotting function --- #
DATA_STYLE = dict(marker=".", alpha=0.8, lw=0, label="data", c="lightgray")
MODEL_STYLE = dict(lw=2, color="k")

def plot_trend(data, samples, ax=None):
    y = data["births_relative"]
    x = data["date"]
    fsd = samples["intercept"][:, None] + samples["trend/f"]
    f = jnp.quantile(fsd * y.std() + y.mean(), 0.50, axis=0)

    if ax is None:
        ax = plt.gca()

    ax.plot(x, y, **DATA_STYLE)
    ax.plot(x, f, **MODEL_STYLE)
    return ax

def plot_seasonality(data, samples, ax=None):
    y = data["births_relative"]
    sdev = y.std()
    mean = y.mean()
    baseline = (samples["intercept"][:, None] + samples["trend/f"]) * sdev
    y_detrended = y - baseline.mean(0)
    y_year_mean = y_detrended.groupby(data["day_of_year"]).mean()
    x = y_year_mean.index

    f_median = (pd.DataFrame(samples["year/f"] * sdev + mean, columns=data["day_of_year"])
                .melt(var_name="day_of_year")
                .groupby("day_of_year")["value"]
                .median()
            )

    if ax is None:
        ax = plt.gca()

    ax.plot(x, y_year_mean, **DATA_STYLE)
    ax.plot(x, f_median, **MODEL_STYLE)
    return ax

def plot_week(data, samples, ax=None):
    if ax is None:
        ax = plt.gca()

    # (continues on next page)
weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
y = data['births_relative']
x = data['day_of_week'] - 1
f = jnp.median(samples['week/beta'] * y.std() + y.mean(), 0)

ax.plot(x, y, **DATA_STYLE)
ax.plot(range(7), f, **MODEL_STYLE)
ax.set_xticks(range(7))
ax.set_xticklabels(weekdays)
return ax

def plot_weektrend(data, samples, ax=None):
dates = data['date']
weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
y = data['births_relative']
mean, sdev = y.mean(), y.std()
intercept = samples['intercept'][:, None]
f1 = samples['trend/f']
f2 = samples['year/f']
g3 = samples['week-trend/f']
baseline = ((intercept + f1 + f2) * y.std()).mean(0)

if ax is None:
    ax = plt.gca()

ax.plot(dates, y - baseline, **DATA_STYLE)
for n, day in enumerate(weekdays):
    week_beta = samples['week/beta'][:, n][:, None]
    fsd = jnp.exp(g3) * week_beta
    f = jnp.quantile(fsd * sdev + mean, 0.50, axis=0)
    ax.plot(dates, f, **MODEL_STYLE)
    ax.text(dates.iloc[-1], f[-1], day)

return ax

def plot_1988(data, samples, ax=None):
indicators = get_floating_days_indicators(data['date'])
memorial_beta = samples['memorial/beta'][:, None]
labour_beta = samples['labour/beta'][:, None]
thanks_beta = samples['thanksgiving/beta'][:, None]

memorials = indicators['memorial_days_indicator'] * memorial_beta
labour = indicators['labour_days_indicator'] * labour_beta
thanksgiving = indicators['thanksgiving_days_indicator'] * thanks_beta
floating_days = memorials + labour + thanksgiving

is_1988 = data['date'].dt.year == 1988
days_in_1988 = data['day_of_year'][is_1988] - 1
days_effect = samples['day/beta'][[:, days_in_1988.values]
floating_effect = floating_days[:, jnp.argwhere(is_1988.values).ravel()]
y = data['births_relative']
f = (days_effect + floating_effect) * y.std() + y.mean()
f_median = jnp.median(f, axis=0)

special_days = {
    'Valentine\'s': "1988-02-14",
    'Leap day': "1988-02-29",
    'Halloween': "1988-10-31",
    'Christmas eve': "1988-12-24",
    'Christmas day': "1988-12-25",
    'New year': "1988-01-01",
    'New year\'s eve': "1988-12-31",
    'April 1st': "1988-04-01",
    'Independence day': "1988-07-04",
    'Labour day': "1988-09-05",
    'Memorial day': "1988-05-30",
    'Thanksgiving': "1988-11-24",
}

if ax is None:
    ax = plt.gca()

ax.plot(days_in_1988, f_median, color="k", lw=2)

for name, date in special_days.items():
    xs = pd.to_datetime(date).day_of_year - 1
    ys = f_median[xs]
    text = ax.text(xs - 3, ys, name, horizontalalignment="right")
    text.set_bbox(dict(facecolor="white", alpha=0.5, edgecolor="none"))

is_day_13 = data['date'].dt.day == 13
bad_luck_days = data.loc[is_1988 & is_day_13, 'day_of_year'] - 1
ax.plot(bad_luck_days, f_median[bad_luck_days.values], marker="o", mec="gray", c="none", ms=10, lw=0)

return ax

def make_figure(data, samples):
    import matplotlib.ticker as mtick

    fig = plt.figure(figsize=(15, 9))
    grid = plt.GridSpec(2, 3, wspace=0.1, hspace=0.25)
    axes = (570 Chapter 38. Example: Hilbert space approximation for Gaussian processes.
NumPyro Documentation

(continued from previous page)

plt.subplot(grid[0,
plt.subplot(grid[1,
plt.subplot(grid[1,
plt.subplot(grid[1,

:]),
0]),
1]),
2]),

)
plot_1988(data, samples, ax=axes[0])
plot_trend(data, samples, ax=axes[1])
plot_seasonality(data, samples, ax=axes[2])
plot_week(data, samples, ax=axes[3])
for ax in axes:
ax.axhline(y=1, linestyle="--", color="gray", lw=1)
if not ax.get_subplotspec().is_first_row():
ax.set_ylim(0.65, 1.35)
if not ax.get_subplotspec().is_first_col():
ax.set_yticks([])
ax.set_ylabel("")
else:
ax.yaxis.set_major_formatter(mtick.PercentFormatter(xmax=1))
ax.set_ylabel("Relative number of births")
axes[0].set_title("Special day effect")
axes[0].set_xlabel("Day of year")
axes[1].set_title("Long term trend")
axes[1].set_xlabel("Year")
axes[2].set_title("Year seasonality")
axes[2].set_xlabel("Day of year")
axes[3].set_title("Day of week effect")
axes[3].set_xlabel("Day of week")
return fig

# --- functions for running the model --- #
def parse_arguments():
parser = argparse.ArgumentParser(description="Hilbert space approx for GPs")
parser.add_argument("--num-samples", nargs="?", default=1000, type=int)
parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
parser.add_argument("--num-chains", nargs="?", default=1, type=int)
parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
parser.add_argument("--x64", action="store_true", help="Enable double precision")
parser.add_argument(
"--save-figure",
default="",
type=str,
help="Path where to save the plot with matplotlib.",
)
args = parser.parse_args()
return args

def main(args):
(continues on next page)

571


is_sphinxbuild = "NUMPYRO_SPHINXBUILD" in os.environ
data = load_data()
data_dict = make_birthdays_data_dict(data)
mcmc = MCMC(
    NUTS(birthdays_model, init_strategy=init_to_median),
    num_warmup=args.num_warmup,
    num_samples=args.num_samples,
    num_chains=args.num_chains,
    progress_bar=(not is_sphinxbuild),
)
mcmc.run(jax.random.PRNGKey(0), **data_dict)
if not is_sphinxbuild:
mcmc.print_summary()

if args.save_figure:
samples = mcmc.get_samples()
print(f"Saving figure at {args.save_figure}"
fig = make_figure(data, samples)
fig.savefig(args.save_figure)
plt.close()

return mcmc

if __name__ == '__main__':
    args = parse_arguments()
numpyro.enable_x64(args.x64)
numpyro.set_platform(args.device)
numpyro.set_host_device_count(args.num_chains)
main(args)
EXAMPLE: HILBERT SPACE APPROXIMATION FOR GAUSSIAN PROCESSES (MULTIDIMENSIONAL)

Gaussian process models (see Example: Gaussian Process) are a flexible class of models for regression, classification, and unsupervised learning. Because they have poor scaling properties, they are not suitable for large datasets. The Hilbert space approximation (see Example: Hilbert space approximation for Gaussian processes) provides a scalable alternative. This example extends the univariate case studied in the previous example to the multidimensional input case and demonstrates the usage of the contributed HSGP module.

First, load the required libraries and set configure jax and numpyro.

```
[1]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[2]: from typing import Sequence

import arviz as az
import matplotlib.pyplot as plt
import numpy as np
from numpy.typing import NDArray
import jax
from jax import random
import jax.numpy as jnp
import optax
from optax import linear_onecycle_schedule
from numpyro import distributions as dist
from numpyro.contrib.hsgp.approximation import hsgp_squared_exponential
from numpyro.infer import Predictive
from numpyro.infer.autoguide import AutoNormal
from numpyro.infer.elbo import Trace_ELBO
from numpyro.infer.hmc import NUTS
from numpyro.infer.initialization import init_to_median, init_to_uniform
from numpyro.infer.mcmc import MCMC
from numpyro.infer.svi import SVI
from numpyro.optim import Adam

[3]: num_devices = 4
numpyro.set_host_device_count(num_devices)
jax.config.update(
    "jax_enable_x64", True
```

(continues on next page)
# additional precision for to avoid numerical issues with Cholesky decomposition of the covariance matrix

## 39.1 Draw Simulated Data

We begin by sampling \(N\) points from a \(D\)-dimensional Gaussian process with a squared exponential kernel function. The input points are drawn from a uniform distribution on a square/hypercube spanning the domain \([-L, L]^D\). We also sample a (noise-free) set of points from a uniformly-spaced grid of inputs in order to visualize the generative process. Because our model will assume that the Gaussian process is centered, we de-mean points in the output space prior to returning them. The `sample_grid_and_data` function returns the gridded values and the data points. The `se_kernel` function implements the covariance function for the Gaussian process.

```python
[4]:
def se_kernel(
    X: jax.Array,
    Z: jax.Array,
    amplitude: float,
    length: float,
    noise: float | None,
    jitter=1.0e-6,
) -> jax.Array:
    r = jnp.linalg.norm(X[:, jnp.newaxis] - Z, axis=-1)
    delta = (r / length) ** 2
    k = (amplitude**2) * jnp.exp(-0.5 * delta)
    if noise is None:
        return k
    else:
        return k + (noise**2 + jitter) * jnp.eye(k.shape[0])

def sample_grid_and_data(
    N_grid: int,
    N: int,
    L: float,
    amplitude: float,
    lengthscale: float,
    noise: float,
    key: int,
    D: int,
) -> tuple[jax.Array, jax.Array, jax.Array, jax.Array]:
    # draw points on a grid for plotting surface of the noiseless function
    x_linspace = jnp.linspace(-L, L, N_grid)
    x_mesh = jnp.meshgrid(*[x_linspace for _ in range(D)])
    X_grid = jnp.concatenate([x_mesh[i].ravel() for i in range(D)], axis=1)

    # draw data points from a uniform distribution on the support of the grid
    X = random.uniform(key, shape=(N, D), minval=-L, maxval=L)
```

(continues on next page)
# concatenate grid and data points
X_all = jnp.concatenate([X_grid, X], axis=0)

# sample from the GP
cov = se_kernel(X_all, X_all, amplitude, lengthscale, 0.0)  # noiseless
_, key = random.split(key)
_y = random.multivariate_normal(key, mean=jnp.zeros(cov.shape[0]), cov=cov)

# separate the grid and data points
y_grid = _y[0 : N_grid**D].reshape((N_grid,) * D)
_, key = random.split(key)
y = _y[N_grid**D :] + (random.normal(key, shape=(N,)) * noise)
  # add noise to the data points
y_mean = y.mean()

return X_grid, y_grid - y_mean, X, y - y_mean

We fix D=2 for this example but the code is fully general. Plots will be rendered for the unidimensional and bidimensional cases. We set N=1000 and L=3.0.

[5]: # parameters for the synthetic data
D = 2
N_grid = 25 if D == 2 else 100
N = 1_000
L = 3.0

# kernel parameters
amplitude = 1.0
lengthscale = 2.0

# noise level
noise = 0.5 if D == 2 else 0.15

# sample the grid and data
seed = 0
key = jax.random.key(seed)

X_grid, y_grid, X, y = sample_grid_and_data(
    N_grid, N, L, amplitude, lengthscale, noise, key, D
)

Before proceeding to the models, we’ll set up a few plotting functions to help visualize the generative process and the posterior predictive distribution. We’ll use plot_surface_scatter in the 2 dimensional case and plot_line_scatter in the 1 dimensional case.

[6]: def plot_surface_scatter(
    N_grid: int,
    X_grid: NDArray | None = None,
    y_grid: NDArray | None = None,
X: NDArray | None = None,
y: NDArray | None = None,
test_ind: jax.Array | None = None,
post_y: jax.Array | None = None,
xz_lines: list[tuple[jax.Array, jax.Array, float]] | None = None,
yz_lines: list[tuple[jax.Array, jax.Array, float]] | None = None,
xy_annotate_lines: Sequence[tuple[tuple[float, float], tuple[float, float]]] | None
] = None,
fig_size: float = 8.0,
label_size: float = 8.0,
grid_alpha: float = 0.1,
y_wireframe_alpha: float = 1.0,
post_alpha: float = 0.1,
point_size: float = 1.0,
point_alpha: float = 0.5,
alpha: float = 0.1,
) -> None:
    # setup figure
    fig = plt.figure(figsize=(fig_size, fig_size))

    # plot the surface of the noiseless function and the data points
    x0_grid, x1_grid = (  
        X_grid[:, 0].reshape((N_grid, N_grid)),  
        X_grid[:, 1].reshape((N_grid, N_grid)),  
    )
    ax = fig.add_subplot(projection="3d")
    ax.plot_wireframe(  
        x0_grid,  
        x1_grid,  
        post_y_grid,  
        rstride=1,  
        cstride=1,  
        alpha=post_alpha,  
        color="tab:blue",  
    )

    # plot wireframes from draws from the posterior
    if post_y is not None:
        for i in range(post_y.shape[0]):
            post_y_grid = post_y[i, :].reshape((N_grid, N_grid))
            ax.plot_wireframe(  
                x0_grid,  
                x1_grid,  
                post_y_grid,  
                rstride=1,  
                cstride=1,  
                linewidth=1.0,  
                alpha=post_alpha,  
                color="tab:blue",  
            )

    # plot the data points
    if X is not None and y is not None:
        color = (  
            "tab:blue"  
            if test_ind is None
            else np.where(test_ind, "tab:green", "tab:blue")
        )
        ax.scatter(  
            xs=X[:, 0],  
            color=color,  
            alpha=alpha,  
            s=point_size,  
            label=label_size,  
            marker="s",  
            linewidth=1.0,  
            edgecolor="tab:blue",  
            capsize=2.0,  
            capstyle="round",  
            clip_on=True,  
            rasterized=True,  
            antialiased=True,  
            alpha=1.0,  
            zorder=5,  
        )

    # plot the posterior
    if post_y is not None:
        ax.plot_wireframe(  
            x0_grid,  
            x1_grid,  
            post_y_grid,  
            rstride=1,  
            cstride=1,  
            linewidth=1.0,  
            alpha=post_alpha,  
            color="tab:blue",  
        )

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")

    # plot the data points
    if X is not None and y is not None:
        ax.scatter(  
            xs=X[:, 0],  
            ys=y[X[:, 0]],  
            color="tab:blue",  
            alpha=alpha,  
            s=point_size,  
            label=label_size,  
            marker="s",  
            linewidth=1.0,  
            edgecolor="tab:blue",  
            capsize=2.0,  
            capstyle="round",  
            clip_on=True,  
            rasterized=True,  
            antialiased=True,  
            alpha=1.0,  
            zorder=5,  
        )

    # plot the posterior
    if post_y is not None:
        ax.plot_wireframe(  
            x0_grid,  
            x1_grid,  
            post_y_grid,  
            rstride=1,  
            cstride=1,  
            linewidth=1.0,  
            alpha=post_alpha,  
            color="tab:blue",  
        )

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
ys = X[:, 1],
zs = y,
c = color,
s = point_size,
alpha = point_alpha,
)

# add confidence intervals at the boundaries
if xz_lines:
    for line in xz_lines:
        x, z, y = line
        ax.plot(
            x, z, zs=y, zdir="y", color="tab:green", linestyle="--", alpha=ci_alpha
        )
if yz_lines:
    for line in yz_lines:
        y, z, x = line
        ax.plot(
            y, z, zs=x, zdir="x", color="tab:green", linestyle="--", alpha=ci_alpha
        )

# plot the surface of the noiseless function
if y_grid is not None:
    ax.plot_wireframe(
        x0_grid,
        x1_grid,
        y_grid,
        rstride=1,
        cstride=1,
        linewidths=1.0,
        alpha=y_wireframe_alpha,
        color="tab:orange",
    )

# add box in xy plane
z_min = ax.get_zlim()[0]
ax.set_zlim(ax.get_zlim())
if xy_annotate_lines:
    for line in xy_annotate_lines:
        x_bounds, y_bounds = line
        z_bounds = (z_min, z_min)
        ax.plot(
            x_bounds,
            y_bounds,
            z_bounds,
            color="tab:gray",
            alpha=0.5,
            linestyle="--",
        )

# remove background panes
ax.xaxis.pane.fill = False
ax.yaxis.pane.fill = False
ax.zaxis.pane.fill = False
ax.yaxis.pane.set_edgecolor("w")
ax.zaxis.pane.set_edgecolor("w")
ax.xaxis.pane.set_edgecolor("w")

# configure grid
ax.xaxis._axinfo["grid"]["color"] = ("tab:gray", grid_alpha)
ax.yaxis._axinfo["grid"]["color"] = ("tab:gray", grid_alpha)
ax.zaxis._axinfo["grid"]["color"] = ("tab:gray", grid_alpha)

# set labels and ticks
ax.xaxis.set_tick_params(labelsize=label_size)
ax.set_xlabel("x0", fontsize=label_size)
ax.yaxis.set_tick_params(labelsize=label_size)
ax.set_ylabel("x1", fontsize=label_size)
ax.zaxis.set_tick_params(labelsize=label_size)
ax.set_zlabel("y", fontsize=label_size)

ax.set_box_aspect(aspect=None, zoom=0.9)
return ax

def plot_line_scatter(
    X_grid: jax.Array,
    y_grid: jax.Array,
    X: jax.Array | None = None,
    y: jax.Array | None = None,
    test_ind: jax.Array | None = None,
    post_y: jax.Array | None = None,
    v_lines: Sequence[float] | None = None,
    ci: tuple[jax.Array, jax.Array] | None = None,
    fig_size: float = 5.0,
    label_size: float = 8.0,
    post_alpha: float = 0.25,
    point_size: float = 1.0,
    point_alpha: float = 0.25,
    ci_alpha: float = 0.1,
):  
    fig = plt.figure(figsize=(fig_size, fig_size))
    ax = fig.add_subplot()

    # plot draws of the function from the posterior
    if post_y is not None:
        for i in range(post_y.shape[0]):
            ax.plot(
                X_grid, post_y[i, :], linewidth=1.0, alpha=post_alpha, color="tab:blue"
            )

    # plot the data points
    if X is not None and y is not None:
        if test_ind is None:

(continues on next page)
color = "tab:blue"

else:
    test_ind = np.array(test_ind).squeeze()
    color = np.where(test_ind, "tab:green", "tab:blue")
    ax.scatter(X, y, c=color, s=point_size, alpha=point_alpha)

    # add confidence intervals
    if ci:
        ax.fill_between(
            X_grid.squeeze(), ci[0], ci[1], color="tab:blue", alpha=ci_alpha
        )

    # add the noiseless function
    ax.plot(X_grid, y_grid, linewidth=1.0, alpha=1.0, color="tab:orange")

    # add vertical lines denoting boundaries of the training data
    if v_lines:
        for v_line in v_lines:
            plt.axvline(v_line, color="tab:gray", linestyle="--", alpha=0.5)
            plt.axvline(v_line, color="tab:gray", linestyle="--", alpha=0.5)

    # set labels and ticks
    ax.set_xlabel("x", fontsize=label_size)
    ax.set_ylabel("y", fontsize=label_size)
    ax.xaxis.set_tick_params(labelsize=label_size)
    ax.yaxis.set_tick_params(labelsize=label_size)

    return ax

We can plot the surface of the noise-free function as a wireframe and the noisy observations as points.

[7]:

    if D == 2:
        plot_surface_scatter(N_grid, X_grid, y_grid, X, y)
    elif D == 1:
        plot_line_scatter(
            X_grid,
            y_grid,
            X,
            y,
        )
    plt.show()
39.2 Exact covariance Gaussian process (baseline)

We begin by fitting an exact Gaussian process model to the noisy points. We infer the hyperparameters of the kernel function and the noise level. To compute the covariance function, we can re-use the `se_kernel` function from above. Because exact Gaussian process models require persisting the train set, we store the training data $X$ and $y$ as attributes on the model so that we can compute the posterior predictive distribution later. When $X_{\text{test}}$ is provided, $f_{\text{star}}$ and $y_{\text{test}}$ are returned as outputs, corresponding to the posterior mean and the samples from the noisy emission distribution at the test points.
Before fitting the model, we split the data into train and test sets. We will train on the data set away from the boundary and test the model’s ability to extrapolate to the boundary.

```
tr_frac = 0.8  # train on data contained within the inner tr_frac fraction of the domain
tr_idx = ((X > -L * tr_frac) & (X < L * tr_frac)).sum(axis=1) == D
tr_idx_grid = ((X_grid > -L * tr_frac) & (X_grid < L * tr_frac)).sum(axis=1) == D
X_tr = X[tr_idx]  # train on values set away from the edges
X_test = X[~tr_idx]
y_tr = y[tr_idx]
y_test = y[~tr_idx]
```
fit_mcmc and fit_svi are helper functions that do inference on the model using MCMC and SVI, respectively. We will use MCMC here, but we can easily switch to SVI to achieve faster inference under a mean-field approximation of the posterior.

```python
InfERENCE = "mcmc"

def fit_mcmc(
    seed: int,
    model: callable,
    num_warmup: int = 500,
    num_samples: int = 500,
    target_accept_prob: float = 0.8,
    init_strategy: callable = init_to_uniform,
    **model_kwargs,
)
    rng_key = random.PRNGKey(seed)
    kernel = NUTS(
        model, target_accept_prob=target_accept_prob, init_strategy=init_strategy
    )
    mcmc = MCMC(
        kernel,
        num_warmup=num_warmup,
        num_samples=num_samples,
        num_chains=4,
        progress_bar=False,
    )
    mcmc.run(rng_key, **model_kwargs)
    return mcmc

def fit_svi(
    seed: int,
    model: callable,
    guide: callable,
    num_steps: int = 5000,
    peak_lr: float = 0.01,
    **model_kwargs,
):
    lr_scheduler = linear_onecycle_schedule(num_steps, peak_lr)
    svi = SVI(model, guide, Adam(lr_scheduler), Trace_ELBO())
    return svi.run(random.PRNGKey(seed), num_steps, progress_bar=False, **model_kwargs)

[10]: INFERENCE = "mcmc"

[11]: if INFERENCE == "mcmc":
    mcmc = fit_mcmc(seed, m.model)
else:
    guide = AutoNormal(m.model, init_loc_fn=init_to_median(num_samples=25))
    svi_res = fit_svi(seed=seed, model=m.model, guide=guide)

We see that the model accurately recovers the generative kernel hyperparameters. (Recall that the value of the kernel amplitude was set to 1.0 and the lengthscale was set to 2.0)
if INFERENCE == "mcmc":
    idata = az.from_numpyro(posterior=mcmc)
mcmc.print_summary()

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std</th>
<th>median</th>
<th>5.0%</th>
<th>95.0%</th>
<th>n_eff</th>
<th>r_hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>amplitude</td>
<td>1.31</td>
<td>0.57</td>
<td>1.17</td>
<td>0.58</td>
<td>2.11</td>
<td>868.30</td>
<td>1.00</td>
</tr>
<tr>
<td>lengthscale</td>
<td>2.37</td>
<td>0.50</td>
<td>2.33</td>
<td>1.56</td>
<td>3.14</td>
<td>925.36</td>
<td>1.00</td>
</tr>
<tr>
<td>noise</td>
<td>0.49</td>
<td>0.01</td>
<td>0.49</td>
<td>0.47</td>
<td>0.52</td>
<td>1307.16</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Number of divergences: 0

Inspecting the individual chains demonstrates convergence.

if INFERENCE == "mcmc":
    VAR_NAMES = ["amplitude", "lengthscale", "noise"]
    axes = az.plot_trace(
        data=idata,
        var_names=VAR_NAMES,
        kind="rank_bars",
        backend_kwargs={"figsize": (10, 7), "layout": "constrained"},
    )

Now, we can set up helper functions to generate draws from the posterior predictive and assess the model’s ability to recover the known functional form and extrapolate to the boundary.
Using the posterior samples, we can predict the values of the function on the set of grid points and test its ability to recover the true function.

```
if INFERENCE == "mcmc":
    post_y = posterior_predictive_mcmc(seed, m.model, mcmc, X_test=X_grid)
else:
    post_y = posterior_predictive_svi(
        seed, m.model, guide, svi_res.params, X_test=X_grid)
```

With posterior predictive samples drawn from the model, we can visualize samples of the latent function and the associated 80 percent credible intervals.

```
def plot_fit_result(
    N_post: int, post: dict[str, jax.Array], q_lower: float = 0.1, q_upper: float = 0.9):
    ci = np.quantile(post["y_test"], jnp.array([q_lower, q_upper]), axis=0)
    if D == 1:
        ci_lower, ci_upper = ci[0, :], ci[1, :]
        test_ind = (X < -L * tr_frac) | (X > L * tr_frac)
        ax = plot_line_scatter(
            X_grid, y_grid, X, y,
            test_ind=test_ind,
            post_y=post["f_star"][0:N_post, :],
            post_alpha=0.1,
            point_alpha=0.15,
            v_lines=[-L * tr_frac, L * tr_frac],
            ci=(ci_lower, ci_upper),
```

(continues on next page)
elif D == 2:
    # compute confidence intervals at the edges of the grid
    yz_ind1 = X_grid[:, 0] == L
    yz_lower1 = ci[0, :] [yz_ind1]
    yz_upper1 = ci[1, :] [yz_ind1]
    yz_ind2 = X_grid[:, 0] == -L
    yz_lower2 = ci[0, :] [yz_ind2]
    yz_upper2 = ci[1, :] [yz_ind2]

    xz_ind1 = X_grid[:, 1] == L
    xz_lower1 = ci[0, :] [xz_ind1]
    xz_upper1 = ci[1, :] [xz_ind1]
    xz_ind2 = X_grid[:, 1] == -L
    xz_lower2 = ci[0, :] [xz_ind2]
    xz_upper2 = ci[1, :] [xz_ind2]

    ax = plot_surface_scatter(
        N_grid=N_grid,
        X_grid=X_grid,
        y_grid=y_grid,
        X=X,
        y=y,
        test_ind=~tr_idx,
        post_y=post["f_star"] [0:N_post, :],
        post_alpha=0.1,
        xy_annotate_lines=[
            (-L * tr_frac, -L * tr_frac), (-L * tr_frac, L * tr_frac),
            (-L * tr_frac, L * tr_frac), (L * tr_frac, L * tr_frac),
            (L * tr_frac, L * tr_frac), (-L * tr_frac, L * tr_frac),
            (L * tr_frac, -L * tr_frac), (-L * tr_frac, -L * tr_frac),
        ],
        yz_lines=[
            (X_grid[yz_ind1, 1], yz_lower1, L),
            (X_grid[yz_ind1, 1], yz_upper1, L),
            (X_grid[yz_ind2, 1], yz_lower2, -L),
            (X_grid[yz_ind2, 1], yz_upper2, -L),
        ],
        xz_lines=[
            (X_grid[xz_ind1, 0], xz_lower1, L),
            (X_grid[xz_ind1, 0], xz_upper1, L),
            (X_grid[xz_ind2, 0], xz_lower2, -L),
            (X_grid[xz_ind2, 0], xz_upper2, -L),
        ],
        y_wireframe_alpha=0.4,
        ci_alpha=0.75,
    )

    return ax

[17]: plot_fit_result(20, post_y)
plt.show()
Here, we plot several draws of the posterior mean ($f_{\text{star}}$) as blue wireframes overlaid on the true function in orange. The square in the x0-x1 plane denotes the region of data on which the model was trained. The green dashed lines on the function’s boundary denote its 80 percent credible intervals at the boundary points. We also plot the set of training points in blue and the set of test points in green.

We can also directly inspect the model’s calibration against the set of grid points. The plot_calibration function compares the true function values to the posterior predictive mean.

```python
[18]: def plot_calibration(
    y_true: jax.Array,
    y_pred: jax.Array,
    test_ind: jax.Array | None = None,
    fig_size: float = 5.0,
):  
    (...)
```

(continues on next page)
label_size: float = 8.0,
point_size: float = 1.0,
x_label: str = "True",
y_label: str = "Predicted",
)
fig = plt.figure(figsize=(fig_size, fig_size))
ax = fig.add_subplot()
color = (
"tab:blue" if test_ind is None else np.where(test_ind, "tab:green", "tab:blue")
)
ax.scatter(y_true, y_pred, c=color, alpha=0.5, s=point_size)
ax.plot(
[y_true.min(), y_true.max()],
[y_true.min(), y_true.max()],
color="tab:orange",
linestyle="--",
)
ax.set_xlim([y_true.min(), y_true.max()])
ax.set_ylim([y_true.min(), y_true.max()])
ax.xaxis.set_tick_params(labelsize=label_size)
ax.set_xlabel(x_label, fontsize=label_size)
ax.yaxis.set_tick_params(labelsize=label_size)
ax.set_ylabel(y_label, fontsize=label_size)
return ax

[19]: ax = plot_calibration(
    y_grid,
    post_y["f_star"].mean(axis=0),
    test_ind=~tr_idx_grid,
    point_size=1.0 if D == 2 else 5.0,
)
plt.show()
Grid points points included in the bounds of the test set are displayed in blue. Points outside the training boundary are displayed in green. The dashed line in orange is the identity line (True=predicted).

Finally, as a benchmark to compare the HSGP approximation against, we compute the root mean squared error on the set of (noisy) test points.

```python
[20]: if INERENCE == "mcmc":
    post_y_test = posterior_predictive_mcmc(seed, m.model, mcmc, X_test=X_test)
else:
    post_y_test = posterior_predictive_svi(
        seed, m.model, guide, svi_res.params, X_test=X_test
    )
print(
    "Test RMSE:",
    jnp.sqrt(jnp.mean((post_y_test["y_test"].mean(axis=0) - y_test) ** 2)),
)
```

Test RMSE: 0.5625003036885073
39.3 HSGP Alternative

We now turn to the Hilbert Space approximation. Mayol et al 2020 provide an accessible and practical introduction to the methodology. Orduz 2024 additionally provides a detailed tutorial with numpyro code for a unidimensional example. Below, we demonstrate the usage of the numpyro.contrib.hsgp module on a multidimensional problem. The full approximation is given by

\[ f(x) = \sum_{j=1}^{m^*} \left( S_\phi \left( \sqrt{\lambda_j} \right) \right)^{1/2} \phi_j(x) \beta_j \]

(Mayol et al 2020, Eq. 14)

Here, \( S_\phi \) is the spectral density of the squared exponential kernel, \( \lambda_j \) are the eigenvalues of the Laplacian, \( \phi_j \) are the eigenfunctions of the Laplacian, and \( \beta_j \) are the coefficients of the expansion (\( \sim \mathcal{N}(0, 1) \)). The total number of eigenfunctions is \( m^* \) which is the product of the number of approximating functions for each dimension.

This approximation is conveniently implemented by the numpyro.contrib.hsgp.approximation module’s \texttt{hsgp_squared_exponential} function. For the full model, we can simply sample kernel hyperparameters, feed these to the \texttt{hsgp_squared_exponential} function, and define the emission distribution. The dimension of the problem is inferred from the trailing dimension of \( X \). We set the number of basis functions for each dimension (\( m \)) to 5 and the support of the model ([\(-L, L\)]) to be 2.5 times the support of the data. If desired, we could pass length-D lists to \( m \) and \( L \) to allow the number of basis functions of the approximation and the length of the approximation interval to vary by dimension.

```python
@jax.tree_util.register_pytree_node_class
class HSGPModel:
    def __init__(self, m: int, L: float) -> None:
        self.m = m
        self.L = L

    def model(self, X: jax.Array, y: jax.Array | None = None):
        amplitude = npyro.sample("amplitude", dist.LogNormal(0, 1))
        length = npyro.sample("lengthscale", dist.Exponential(1))
        noise = npyro.sample("noise", dist.LogNormal(0, 1))

        f = npyro.deterministic(
            "f_star",
            npyro.contrib.hsgp_squared_exponential(  
                X, alpha=amplitude, length=length, ell=self.L, m=self.m
            ),
        )

        site = "y" if y is not None else "y_test"
        npyro.sample(site, dist.Normal(f, noise), obs=y)

    def tree_flatten(self):
        children = ()  # arrays / dynamic values
        aux_data = (  
            self.L,
            self.m,
        )  # static values
```

(continues on next page)
We can fit the model using the same utility functions from above.

```python
[22]: hsgp_m = HSGPModel(m=5, L=L * 2.5)

if INFERENCE == "mcmc":
    hsgp_mcmc = fit_mcmc(
        2,
        hsgp_m.model,
        X=X_tr,
        y=y_tr,
        num_warmup=500,
        num_samples=500,
        target_accept_prob=0.95,
        init_strategy=init_to_median(num_samples=25),
    )
else:
    hsgp_guide = AutoNormal(hsgp_m.model, init_loc_fn=init_to_median(num_samples=25))
    hsgp_res = fit_svi(seed, hsgp_m.model, hsgp_guide, X=X_tr, y=y_tr, num_steps=10_000)
```

We see that the inferred kernel hyperparameters closely (although not exactly) match those of the exact model.

```python
[23]: if INFERENCE == "mcmc":
    idata_hsgp = az.from_numpyro(posterior=hsgp_mcmc)
    hsgp_mcmc.print_summary()
```

(continues on next page)
Number of divergences: 0

Chains are well-mixed, like the exact model.

```python
[24]: if INFERENC == "mcmc":
    axes = az.plot_trace(
        data=idata_hsgp,
        var_names=VAR_NAMES,
        kind="rank_bars",
        backend_kwargs={"figsize": (10, 7), "layout": "constrained"},
    )
```

We can also overlay the posterior samples of the kernel hyperparameters from the exact model and the approximation using `arviz`'s `plot_density` function.
We can generate predictions for the grid points as above.

```python
[26]: if INFERENCE == "mcmc":
    post_y_hsgp = posterior_predictive_mcmc(seed, hsgp_m.model, hsgp_mcmc, X=X_grid)
else:
    post_y_hsgp = posterior_predictive_svi(
        seed,
        hsgp_m.model,
        hsgp_guide,
        hsgp_res.params,
        X=X_grid,
    )
```

Like the exact model, the approximation accurately recovers the shape of the function and extrapolates well to the boundary.

```python
[27]: plot_fit_result(20, post_y_hsgp)
plt.show()
```
We can now compare the posterior predictive mean from the approximate model to the predictions from the exact model. We see the approximate model closely matches the exact model.

```
[28]: ax = plot_calibration(
    post_y["f_star"].mean(axis=0),
    post_y_hsgp["f_star"].mean(axis=0),
    point_size=1.0 if D == 2 else 5.0,
    x_label="Exact GP",
    y_label="HSGP",
)
plt.show()
```
And the approximate model performs well on the test set.

```python
[29]: if INFERENE == "mcmc":
        post_y_test_hsgp = posterior_predictive_mcmc(
            seed, hsgp_m.model, hsgp_mcmc, X=X_test
        )
    else:
        post_y_test_hsgp = posterior_predictive_svi(
            seed, hsgp_m.model, hsgp_guide, hsgp_res.params, X=X_test
        )
    print(
        "Test RMSE:",
        jnp.sqrt(jnp.mean((post_y_test_hsgp["y_test"].mean(axis=0) - y_test) ** 2)),
    )

Test RMSE: 0.564753977438082
```
EXAMPLE: PREDATOR-PREY MODEL

This example replicates the great case study [1], which leverages the Lotka-Volterra equation [2] to describe the dynamics of Canada lynx (predator) and snowshoe hare (prey) populations. We will use the dataset obtained from [3] and run MCMC to get inferences about parameters of the differential equation governing the dynamics.

References:
3. http://people.whitman.edu/~hundledr/courses/M250F03/M250.html
import argparse
import os

import matplotlib
import matplotlib.pyplot as plt

from jax.experimental.ode import odeint
import jax.numpy as jnp
from jax.random import PRNGKey

import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import LYNXHARE, load_dataset
from numpyro.infer import MCMC, NUTS, Predictive

matplotlib.use("Agg")  # noqa: E402

def dz_dt(z, t, theta):
    """
    Lotka–Volterra equations. Real positive parameters \`alpha`, \`beta`, \`gamma`, \`delta` describes the interaction of two species.
    """
    u = z[0]
v = z[1]
    alpha, beta, gamma, delta = (
        theta[..., 0],
        theta[..., 1],
        theta[..., 2],
        theta[..., 3],
    )
du_dt = (alpha - beta * v) * u
dv_dt = (-gamma + delta * u) * v
    return jnp.stack([du_dt, dv_dt])

def model(N, y=None):
    """
    :param int N: number of measurement times
    :param numpy.ndarray y: measured populations with shape (N, 2)
    """
    # initial population
    z_init = numpyro.sample("z_init", dist.LogNormal(jnp.log(10), 1).expand([2]))
    # measurement times
    ts = jnp.arange(float(N))
    # parameters alpha, beta, gamma, delta of dz_dt
    theta = numpyro.sample("theta",
        "theta",
        dist.TruncatedNormal(  
            low=0.0,  
            loc=jnp.array([1.0, 0.05, 1.0, 0.05]),  
            scale=jnp.array([0.5, 0.05, 0.5, 0.05]),  
        ),
    )

(continues on next page)
# integrate \(dz/dt\), the result will have shape \(N \times 2\)

\[
z = \text{odeint}(dz\_dt, z\_init, ts, theta, rtol=1e-6, atol=1e-5, mxstep=1000)
\]

# measurement errors

\[
\sigma = \text{numpyro.sample}("\sigma", \text{dist.LogNormal}(-1, 1).\text{expand}([2]))
\]

# measured populations

\[
\text{numpyro.sample}("y", \text{dist.LogNormal}(\text{jnp.log}(z), \sigma), \text{obs}=y)
\]

def main(args):
    _, fetch = load_dataset(LYNXHARE, shuffle=False)
    year, data = fetch()  # data is in hare -> lynx order

    # use dense_mass for better mixing rate
    mcmc = MCMC(
        NUTS(model, dense_mass=True),
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(PRNGKey(1), N=data.shape[0], y=data)
    mcmc.print_summary()

    # predict populations
    pop_pred = Predictive(model, mcmc.get_samples())(PRNGKey(2), data.shape[0])['y']
    mu = jnp.mean(pop_pred, 0)
    pi = jnp.percentile(pop_pred, jnp.array([10, 90]), 0)
    plt.figure(figsize=(8, 6), constrained_layout=True)
    plt.plot(year, data[:, 0], 'ko', mfc="none", ms=4, label="true hare", alpha=0.67)
    plt.plot(year, data[:, 1], 'bx', label="true lynx")
    plt.plot(year, mu[:, 0], 'k--', label="pred hare", lw=1, alpha=0.67)
    plt.plot(year, mu[:, 1], 'b--', label="pred lynx")
    plt.fill_between(year, pi[0, :, 0], pi[1, :, 0], color="k", alpha=0.2)
    plt.fill_between(year, pi[0, :, 1], pi[1, :, 1], color="b", alpha=0.3)
    plt.gca().set(ylim=(0, 160), xlabel="year", ylabel="population (in thousands)"
    plt.title("Posterior predictive (80% CI) with predator-prey pattern.")
    plt.legend()
    plt.savefig("ode_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Predator-Prey Model")
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("-num-warmup", narg="?", default=1000, type=int)
    parser.add_argument("-num-chains", narg="?", default=1, type=int)
    parser.add_argument("-device", default="cpu", type=str, help='use "cpu" or "gpu"')
    args = parser.parse_args()
    numpyro.set_platform(args.device)
```python
numpyro.set_host_device_count(args.num_chains)
main(args)
```
SOLVING DIFFERENTIAL EQUATIONS (ODES) FOR MULTIPLE INITIAL CONDITIONS.

Ordinary differential equations (ODEs) find applications in various fields, including epidemiology, physics, chemistry, banking, and more. Oftentimes, an ODE system requires integration for multiple initial conditions while keeping parameters constant. Additionally, typical datasets often contain missing values, exhibit different durations, and have irregularly spaced data points. This tutorial expands upon the previous Predator-Prey Model tutorial to address these challenges. We will:

1. Define ODEs and the probabilistic model.
2. Generate synthetic datasets with imperfections.
3. Perform parameter estimation using the MCMC algorithm.

```
[ ]: #!pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[2]: import funcitools

import matplotlib.pyplot as plt

import jax
from jax.experimental.ode import odeint
import jax.numpy as jnp
from jax.random import PRNGKey

import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive, init_to_sample

# Numerical instabilities may arise during ODE solving,
# so one has sometimes to play around with solver settings,
# change solver, or change numeric precision as we do here.
numpyro.enable_x64(True)
```
41.1 Model

Let’s start by defining our differential equations, \(dz/\text{dt}\), and the probabilistic model, model. The differential equations remain the same as in the Lotka-Volterra tutorial. However, notable changes are introduced in the model to accommodate multiple initial conditions simultaneously. We begin by sampling initial conditions, \(z_{\text{init}}\), and parameters, \(\theta\). Subsequently, the ODE system is solved in a vectorized form. Vectorization is achieved using \(\text{jax.vmap}\), with the use of \(\text{functools.partial}\) for passing kwargs. Next, we sample sigma to represent measurement error. Finally, we sample the measured populations. Given that missing values may exist in the observed \(y\), we mask non-finite values.

\[
\text{def } dz/\text{dt}(z, t, \theta):
\]

```
Lotka–Volterra equations. Real positive parameters \('alpha', 'beta', 'gamma', 'delta'\)
describes the interaction of two species.
```

\[
u, v = z
\alpha, \beta, \gamma, \delta = \theta
\]

\[
du/\text{dt} = (\alpha - \beta \cdot v) \cdot u
dv/\text{dt} = (-\gamma + \delta \cdot u) \cdot v
\]

\[
\text{return } \text{jnp.stack([du/\text{dt}, dv/\text{dt}])}
\]

\[
def model(ts, y_{\text{init}}, y=\text{None}):
\]

```
:param numpy.ndarray ts: measurement times
:param numpy.ndarray y_{\text{init}}: measured initial conditions
:param numpy.ndarray y: measured populations

# initial population
z_{\text{init}} = \text{numpyro.sample('z_{\text{init}}', dist.LogNormal(jnp.log(y_{\text{init}}), jnp.ones_like(y_{\text{init}})))}
```

\[
\# parameters alpha, beta, gamma, delta of dz/\text{dt}
\theta = \text{numpyro.sample('theta', dist.TruncatedNormal(}
\quad \text{low}=0.0,
\quad \text{loc}=\text{jnp.array([1.0, 0.05, 1.0, 0.05]),}
\quad \text{scale}=\text{jnp.array([0.2, 0.01, 0.2, 0.01]),}
\quad \text{)}
\]

\[
\# helpers to solve ODEs in a vectorized form
\text{odeint_with_kwvars} = \text{functools.partial(odeint, rtol=1e-6, atol=1e-5, mxstep=1000)}
\text{vect_solve_ode} = \text{jax.vmap(}
\quad \text{odeint_with_kwvars},
\quad \text{in_axes=(None, 0, 0, None),}
\quad \text{)}
\]

\[
\# integrate dz/\text{dt}
\text{zs} = \text{vect_solve_ode(dz/\text{dt}, z_{\text{init}}, ts, \theta)}
\# measurement errors
```

(continues on next page)
sigma = numpyro.sample("sigma", dist.LogNormal(-1, 1).expand([2]))

# measured populations
if y is not None:
    # mask missing observations in the observed y
    mask = jnp.isfinite(jnp.log(y))
    numpyro.sample("y", dist.LogNormal(jnp.log(zs), sigma).mask(mask), obs=y)
else:
    numpyro.sample("y", dist.LogNormal(jnp.log(zs), sigma))

### 41.2 Dataset

For the purpose of this tutorial, we will utilize synthetic datasets generated by sampling from the previously defined model. To emulate the non-ideal properties of real-life datasets, we will introduce missing values, varying durations, and irregular spacing between timepoints. It’s important to note that JAX works with vectorized and compiled calculations, requiring datasets to have the same length. In our case, although we have different spacing, we maintain the same number of points. If it’s not the case one can use `jnp.pad` to extend all datasets to the same length with dummy fill values, which can later be masked.

First, let’s establish simulation settings. The datasets will exhibit varying timespans between $t_{min}$ and $t_{max}$, with the number of points constrained between $n_{points_{min}}$ and $n_{points_{max}}$. Additionally, we will introduce missing values with a probability of $p_{missing}$.

```python
[4]: n_datasets = 3  # int n_datasets: number of datasets to generate
t_min = 100  # int t_min: minimal allowed length of the generated time array
t_max = 200  # int t_min: maximal allowed length of the generated time array
n_points_min = 80  # int n_points_min: minimal allowed number of points in a data set
n_points_max = 120  # int n_points_max: maximal allowed number of points in a data set
y0_min = 2.0  # float y0_min: minimal allowed value for initial conditions
y0_max = 10.0  # float y0_max: maximal allowed value for initial conditions
p_missing = 0.1  # float p_missing: probability of having missing values
```

Generate an array with initial conditions

```python
[5]: # generate an array with initial conditions
z_inits = jnp.array(
    [jnp.linspace(y0_min, y0_max, n_datasets), jnp.linspace(y0_max, y0_min, n_datasets)]
).T

print(f"Initial conditions are: \\n  {z_inits}"
)

Initial conditions are:
[[ 2. 10.]
 [ 6.  6.]
[10.  2.]]
```

Next, let’s create a time matrix $t_s$ to store the time points for each individual dataset. We will generate random integers in `rand_duration` between $t_{min}$ and $t_{max}$ to represent varying durations. Similarly, `rand_n_points` will correspond to different spacings in each dataset. Since JAX requires a matrix with a constant shape, we will use `jnp.pad` to pad individual observations to the common length of the longest array.
```python
# generate array with random integers between t_min and t_max, representing time duration in the data set
rand_duration = jax.random.randint(
    PRNGKey(1), shape=(n_datasets,), minval=t_min, maxval=t_max
)

# generate array with random integers between n_points_min and n_points_max, representing number of time points per dataset
rand_n_points = jax.random.randint(
    PRNGKey(1), shape=(n_datasets,), minval=n_points_min, maxval=n_points_max
)

# Note that arrays have different length and are stored in a list
time_arrays = [
    jnp.linspace(0, j, num=rand_n_points[i]).astype(float)
    for i, j in enumerate(rand_duration)
]
longest = jnp.max(jnp.array([len(i) for i in time_arrays]))

# Make a time matrix
ts = jnp.array(
    [jnp.pad(arr, pad_width=(0, longest - len(arr)), constant_values=jnp.nan)
     for arr in time_arrays
]
)

print(f"The shape of the time matrix is {ts.shape}"
print(f"First values are 

{ts[:, :10]}"
print(f"Last values are 

{ts[:, -10:]}"

The shape of the time matrix is (3, 108)
First values are
[[ 0.  1.00934579  2.01869159  3.02803738  4.03738318  5.04672897  6.05607477  7.06542056  8.07476636  9.08411215]
 [ 0.  1.23863636  2.47727273  3.71590909  4.95454545  6.19318182  7.43181818  8.67045455  9.90909091 11.14772727]
 [ 0.  1.21212121  2.42424242  3.63636364  4.84848485  6.06060606  7.27272727  8.48484848  9.69696969 10.90909091]]
Last values are
[[ 98.91588785  99.92523364 100.93457944 101.94392523 102.95327103
 103.96261682 104.97196262 105.98130841 106.99065421 108.     ]
 [ nan  nan  nan  nan  nan  nan  nan  nan  nan  nan]
 [118.78787879 120.     nan  nan  nan  nan  nan  nan  nan  nan]
 [ nan  nan  nan  nan  nan  nan  nan  nan  nan  nan]]

We’ll utilize the Predictive mode from NumPyro to draw a single sample, representing our synthetic dataset. Subsequently, we’ll apply a mask with NaNs to the data to simulate missing values. For simplicity, we’ll ensure that initial values are non-missing. In real datasets where this may not hold true, then various imputation methods can be applied.
```

```python
# take a single sample that will be our synthetic data
sample = Predictive(model, num_samples=1)(PRNGKey(100), ts, z_inits)
```

(continues on next page)
data = sample["y"][0]

# create a mask that will add missing values to the data
missing_obs_mask = jax.random.choice(
    PRNGKey(1),
    jnp.array([True, False]),
    shape=data.shape,
    p=jnp.array([p_missing, 1 - p_missing]),
)

# make sure that initial values are not missing
missing_obs_mask = missing_obs_mask.at[:, 0, :].set(False)

# data with missing values
data = data.at[missing_obs_mask].set(jnp.nan)

Finally, for compatibility with NUTS later on, we need to fill NaN values in the time matrix `ts` with dummy variables. The `odeint` function from JAX requires these values to be in increasing order. We fill them with values greater than `t_max` from the time matrix. Importantly, these values do not affect the MCMC estimation, as the corresponding values in the `data` are missing and thereby ignored during the posterior estimation.

[8]: # fill_nans
def fill_nans(ts):
    n_nan = jnp.sum(jnp.isnan(ts))
    if n_nan > 0:
        loc_first_nan = jnp.where(jnp.isnan(ts))[0][0]
        ts_filled_nans = ts.at[loc_first_nan:].set(
            jnp.linspace(t_max, t_max + 20, n_nan)
        )
    return ts_filled_nans
else:
    return ts

ts_filled_nans = jnp.array([fill_nans(t) for t in ts])

Let's briefly summarize our synthetic dataset:

[9]:
print("The dataset has the shape {data.shape}, (n_datasets, n_points, n_observables)")
print("The time matrix has the shape {ts.shape}, (n_datasets, n_timepoints)"
print("The time matrix has different spacing between timepoints: \n {ts[:, :, 5]}"
print("The final timepoints are: {jnp.nanmax(ts, 1)} years.")
print("The dataset has {jnp.sum(jnp.isnan(data))/jnp.size(data):.0%} missing observations"
)
print("True params mean: {sample[\'theta\'][0]}")

The dataset has the shape (3, 108, 2), (n_datasets, n_points, n_observables)
The time matrix has the shape (3, 108), (n_datasets, n_timepoints) The time matrix has different spacing between timepoints:
[[0. 1.00934579 2.01869159 3.02803738 4.03738318]
 [0. 1.23863636 2.47727273 3.71590909 4.95454545]
 [0. 1.2121212 2.42424242 3.63636364 4.84848485]]
The final timepoints are: [108. 109. 120.] years.
The dataset has 19% missing observations
True params mean: [0.78770691 0.05049109 0.89073622 0.05296055]

Let’s visualize the dataset, with solid lines helping to guide the eye. You’ll notice line breaks where NaN values occur.

```python
fig, axs = plt.subplots(2, n_datasets, figsize=(15, 4))
for i in range(n_datasets):
    loc = jnp.where(jnp.isfinite(data[:, :, 0]))[0][-1]
    axs[0, i].plot(ts[i, :], data[i, :, 0], "ko", mfc="none", ms=4, label="true hare", alpha=0.67)
    axs[0, i].plot(ts[i, :], data[i, :, 0], label="true hare", alpha=0.67)
    axs[0, i].set_xlabel("Time, year")
    axs[0, i].set_ylabel("Population")
    axs[0, i].set_xlim([-5, jnp.nanmax(ts)])
    axs[1, i].plot(ts[i, :], data[i, :, 1], "bx", label="true lynx")
    axs[1, i].plot(ts[i, :], data[i, :, 1], label="true lynx")
    axs[1, i].set_xlabel("Time, year")
    axs[1, i].set_ylabel("Population")
    axs[1, i].set_xlim([-5, jnp.nanmax(ts)])
fig.tight_layout()
```

41.3 Perform MCMC.

To achieve a balance between accuracy and speed, one has to adjust the parameters of both the MCMC solver and the ODE solver to suit the specific problem.

```python
y_init = data[:, :, 0]
mcmc = MCMC(
    NUTS(
        model,
        dense_mass=True,
        init_strategy=init_to_sample(),
    )
)```

(continues on next page)
mcmc.run(PRNGKey(1031410), ts=ts_filled_nans, y_init=y_init, y=data)
mcmc.print_summary()

print(f"True params mean: {sample['theta'][0]}")
print(f"Estimated params mean: {jnp.mean(mcmc.get_samples()]['theta'], axis = 0)}")

mean std median 5.0% 95.0% n_eff r_hat
sigma[0] 0.29 0.01 0.29 0.27 0.31 1064.77 1.00
sigma[1] 0.51 0.02 0.51 0.47 0.55 1593.43 1.00
theta[0] 0.77 0.02 0.77 0.74 0.79 760.41 1.00
theta[1] 0.05 0.00 0.05 0.05 0.05 888.74 1.00
theta[2] 0.91 0.02 0.91 0.87 0.94 842.09 1.00
theta[3] 0.06 0.00 0.06 0.05 0.06 858.84 1.00
z_init[0, 0] 1.51 0.05 1.51 1.43 1.60 782.07 1.00
z_init[0, 1] 9.11 0.55 9.09 8.06 9.88 1072.88 1.00
z_init[1, 0] 3.83 0.14 3.83 3.63 4.07 986.01 1.00
z_init[1, 1] 8.54 0.57 8.54 7.66 9.52 945.91 1.00
z_init[2, 0] 3.87 0.15 3.86 3.64 4.11 1210.24 1.00
z_init[2, 1] 3.70 0.19 3.69 3.39 4.02 1342.93 1.00

Number of divergences: 0
True params mean: [0.78770691 0.05049109 0.89073622 0.05296055]
Estimated params mean: [0.7684689 0.05000161 0.90749349 0.05559383]

41.4 Run predictions.

[12]: # predict
ts_pred = jnp.tile(jnp.linspace(0, 200, 1000), (n_datasets, 1))
pop_pred = Predictive(model, mcmc.get_samples())(PRNGKey(1041140), ts_pred, y_init)['y']
mu = jnp.mean(pop_pred, 0)
pi = jnp.percentile(pop_pred, jnp.array([10, 90]), 0)

print(f"True params mean: {sample['theta'][0]}")
print(f"Estimated params mean: {jnp.mean(mcmc.get_samples()]['theta'], axis = 0)}")

True params mean: [0.78770691 0.05049109 0.89073622 0.05296055]
Estimated params mean: [0.7684689 0.05000161 0.90749349 0.05559383]

Plot the observed points and predicted mean with prediction intervals.

41.4. Run predictions. 605
```python
fig, axs = plt.subplots(2, n_datasets, figsize=(15, 4))
for i in range(n_datasets):
    loc = jnp.where(jnp.isfinite(data[i, :, 0]))[0][-1]
    axs[0, i].plot(ts_pred[i, :], mu[i, :, 0], "k-.", label="pred hare", lw=1, alpha=0.67)
    axs[0, i].plot(ts[i, :], data[i, :, 0], "ko", mfc="none", ms=4, label="true hare", alpha=0.67)
    axs[0, i].fill_between(ts_pred[i, :], pi[0, i, :, 0], pi[1, i, :, 0], color="k", alpha=0.2)
    axs[0, i].set_xlabel("Time, year")
    axs[0, i].set_ylabel("Population")
    axs[0, i].set_xlim([-5, jnp.nanmax(ts)])

axs[1, i].plot(ts_pred[i, :], mu[i, :, 1], "b--", label="pred lynx")
axs[1, i].plot(ts[i, :], data[i, :, 1], "bx", label="true lynx")
axs[1, i].fill_between(ts_pred[i, :], pi[0, i, :, 1], pi[1, i, :, 1], color="b", alpha=0.2)
axs[1, i].set_xlabel("Time, year")
axs[1, i].set_ylabel("Population")
axs[1, i].set_xlim([-5, jnp.nanmax(ts)])

fig.tight_layout()
```

606 Chapter 41. Solving differential equations (ODEs) for multiple initial conditions.
EXAMPLE: NEURAL TRANSPORT

This example illustrates how to use a trained AutoBNAFNormal autoguide to transform a posterior to a Gaussian-like one. The transform will be used to get better mixing rate for NUTS sampler.

References:


```python
import argparse
import os
from matplotlib.gridspec import GridSpec
import matplotlib.pyplot as plt
import seaborn as sns
```

(continues on next page)
from jax import random
import jax.numpy as jnp
from jax.scipy.special import logsumexp

import numpyro
from numpyro import optim
from numpyro.diagnostics import print_summary
import numpyro.distributions as dist
from numpyro.distributions import constraints
from numpyro.infer import MCMC, NUTS, SVI, Trace_ELBO
from numpyro.infer.autoguide import AutoBNAFNormal
from numpyro.infer.reparam import NeuTraReparam

class DualMoonDistribution(dist.Distribution):
    support = constraints.real_vector

    def __init__(self):
        super(DualMoonDistribution, self).__init__(event_shape=(2,))

    def sample(self, key, sample_shape=()):
        # it is enough to return an arbitrary sample with correct shape
        return jnp.zeros(sample_shape + self.event_shape)

    def log_prob(self, x):
        term1 = 0.5 * ((jnp.linalg.norm(x, axis=-1) - 2) / 0.4) ** 2
        term2 = -0.5 * ((x[..., :1] + jnp.array([-2.0, 2.0])) / 0.6) ** 2
        pe = term1 - logsumexp(term2, axis=-1)
        return -pe

def dual_moon_model():
    numpyro.sample("x", DualMoonDistribution())

def main(args):
    print("Start vanilla HMC...")
    nuts_kernel = NUTS(dual_moon_model)
    mcmc = MCMC(
        nuts_kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(random.PRNGKey(0))
    mcmc.print_summary()
    vanilla_samples = mcmc.get_samples()["x"].copy()

    guide = AutoBNAFNormal(
        dual_moon_model, hidden_factors=[args.hidden_factor, args.hidden_factor]
    )
(continues on next page)
svi = SVI(dual_moon_model, guide, optim.Adam(0.003), Trace_ELBO())

print("Start training guide...")
svi_result = svi.run(random.PRNGKey(1), args.num_iters)
print("Finish training guide. Extract samples...")
guide_samples = guide.sample_posterior(
    random.PRNGKey(2), svi_result.params, sample_shape=(args.num_samples,))
)["x"].copy()

print("\nStart NeuTra HMC...")
nutra = NeuTraReparam(guide, svi_result.params)
nutra_model = neutra.reparam(dual_moon_model)
nuts_kernel = NUTS(nutra_model)
mcmc = MCMC(
    nuts_kernel,
    num_warmup=args.num_warmup,
    num_samples=args.num_samples,
    num_chains=args.num_chains,
    progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
)
mcmc.run(random.PRNGKey(3))
mcmc.print_summary()
zs = mcmc.get_samples(group_by_chain=True)["auto_shared_latent"]
print("Transform samples into unwarped space...")
samples = neutra.transform_sample(zs)
print_summary(samples)
zs = zs.reshape(-1, 2)
samples = samples["x"].reshape(-1, 2).copy()

# make plots

# guide samples (for plotting)
guide_base_samples = dist.Normal(jnp.zeros(2), 1.0).sample(
    random.PRNGKey(4), (1000,))
guide_trans_samples = neutra.transform_sample(guide_base_samples)["x"]
x1 = jnp.linspace(-3, 3, 100)
x2 = jnp.linspace(-3, 3, 100)
X1, X2 = jnp.meshgrid(x1, x2)
P = jnp.exp(DualMoonDistribution().log_prob(jnp.stack([X1, X2], axis=-1)))

fig = plt.figure(figsize=(12, 8), constrained_layout=True)
gs = GridSpec(2, 3, figure=fig)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
ax3 = fig.add_subplot(gs[0, 1])
ax4 = fig.add_subplot(gs[1, 1])
ax5 = fig.add_subplot(gs[0, 2])
ax6 = fig.add_subplot(gs[1, 2])

ax1.plot(svi_result.losses[1000:])

(continues on next page)
ax1.set_title("Autoguide training loss
(after 1000 steps")

ax2.contourf(X1, X2, P, cmap="OrRd")
sns.kdeplot(x=guide_samples[:, 0], y=guide_samples[:, 1], n_levels=30, ax=ax2)
ax2.set(
    xlim=[-3, 3],
    ylim=[-3, 3],
    xlabel="x0",
    ylabel="x1",
    title="Posterior using
AutoBNAFNormal guide",
)

sns.scatterplot(
    x=guide_base_samples[:, 0],
    y=guide_base_samples[:, 1],
    ax=ax3,
    hue=guide_trans_samples[:, 0] < 0.0,
)
ax3.set(
    xlim=[-3, 3],
    ylim=[-3, 3],
    xlabel="x0",
    ylabel="x1",
    title="AutoBNAFNormal base samples
(True=left moon; False=right moon)",
)

ax4.contourf(X1, X2, P, cmap="OrRd")
sns.kdeplot(x=vanilla_samples[:, 0], y=vanilla_samples[:, 1], n_levels=30, ax=ax4)
av行政机关[-50::, 0], vanilla_samples[-50::, 1], "bo-", alpha=0.5
ax4.set(
    xlim=[-3, 3],
    ylim=[-3, 3],
    xlabel="x0",
    ylabel="x1",
    title="Posterior using
vanilla HMC sampler",
)

sns.scatterplot(
    x=zs[:, 0],
    y=zs[:, 1],
    ax=ax5,
    hue=samples[:, 0] < 0.0,
    s=30,
    alpha=0.5,
    edgecolor="none",
)
ax5.set(
    xlim=[-5, 5],
    ylim=[-5, 5],
    xlabel="x0",
    ylabel="x1",
    title="Samples from the
warped posterior - p(z)",
)(continues on next page)
```python
ax6.contourf(X1, X2, P, cmap="OrRd")
sns.kdeplot(x=samples[:, 0], y=samples[:, 1], n_levels=30, ax=ax6)
ax6.plot(samples[-50:, 0], samples[-50:, 1], "bo", alpha=0.2)
ax6.set(
    xlim=[-3, 3],
ylim=[-3, 3],
xlabel="x0",
ylabel="x1",
title="Posterior using NeuTra HMC sampler",
)
plt.savefig("neutra.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="NeuTra HMC")
    parser.add_argument("-n", "--num-samples", nargs="?", default=4000, type=int)
    parser.add_argument("--num-warmup", narg="?", default=1000, type=int)
    parser.add_argument("--num-chains", narg="?", default=1, type=int)
    parser.add_argument("--hidden-factor", narg="?", default=8, type=int)
    parser.add_argument("--num-iters", narg="?", default=10000, type=int)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```

EXAMPLE: THOMPSON SAMPLING FOR BAYESIAN OPTIMIZATION WITH GPS

In this example we show how to implement Thompson sampling for Bayesian optimization with Gaussian processes. The implementation is based on this tutorial: https://gdmarmerola.github.io/ts-for-bayesian-optim/
Chapter 43. Example: Thompson sampling for Bayesian Optimization with GPs
import argparse
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
import jax.random as random
from jax.scipy import linalg
import numpyro
import numpyro.distributions as dist
from numpyro.infer import SVI, Trace_ELBO
from numpyro.infer.autoguide import AutoDelta

numpyro.enable_x64()

# the function to be minimized. At y=0 to get a 1D cut at the origin
def ackley_1d(x, y=0):
    out = (-20 * jnp.exp(-0.2 * jnp.sqrt(0.5 * (x**2 + y**2)))
          - jnp.exp(0.5 * (jnp.cos(2 * jnp.pi * x) + jnp.cos(2 * jnp.pi * y)))
          + jnp.e
          + 20)
    return out

# matern kernel with nu = 5/2
def matern52_kernel(X, Z, var=1.0, length=0.5, jitter=1.0e-6):
    d = jnp.sqrt(0.5) * jnp.sqrt(jnp.power((X[:, None] - Z), 2.0)) / length
    k = var * (1 + d + (d**2) / 3) * jnp.exp(-d)
    if jitter:
        # we are assuming a noise free process, but add a small jitter for numerical stability
        k += jitter * jnp.eye(X.shape[0])
    return k

def model(X, Y, kernel=matern52_kernel):
    # set uninformative log-normal priors on our kernel hyperparameters
    var = numpyro.sample("var", dist.LogNormal(0.0, 1.0))
    length = numpyro.sample("length", dist.LogNormal(0.0, 1.0))

    # compute kernel
    k = kernel(X, X, var, length)

    # sample Y according to the standard gaussian process formula
    numpyro.sample("Y",
                  dist.MultivariateNormal(loc=jnp.zeros(X.shape[0]), covariance_matrix=k),
                  obs=Y,
                  (continues on next page)
class GP:
    def __init__(self, kernel=matern52_kernel):
        self.kernel = kernel
        self.kernel_params = None

    def fit(self, X, Y, rng_key, n_step):
        self.X_train = X

        # store moments of training y (to normalize)
        self.y_mean = jnp.mean(Y)
        self.y_std = jnp.std(Y)

        # normalize y
        Y = (Y - self.y_mean) / self.y_std

        # setup optimizer and SVI
        optim = numpyro.optim.Adam(step_size=0.005, b1=0.5)
        svi = SVI(
            model,
            guide=AutoDelta(model),
            optim=optim,
            loss=Trace_ELBO(),
            X=X,
            Y=Y,
        )

        params, _ = svi.run(rng_key, n_step)

        # get kernel parameters from guide with proper names
        self.kernel_params = svi.guide.median(params)

        # store cholesky factor of prior covariance
        self.L = linalg.cho_factor(self.kernel(X, X, **self.kernel_params))

        # store inverted prior covariance multiplied by y
        self.alpha = linalg.cho_solve(self.L, Y)

        return self.kernel_params

    # do GP prediction for a given set of hyperparameters. this makes use of the well-known
    # formula for gaussian process predictions
    def predict(self, X, return_std=False):
        # compute kernels between train and test data, etc.
        k_pp = self.kernel(X, X, **self.kernel_params)
        k_pX = self.kernel(X, self.X_train, **self.kernel_params, jitter=0.0)

        # compute posterior covariance
K = k_pp - k_pX @ linalg.cho_solve(self.L, k_pX.T)

# compute posterior mean
mean = k_pX @ self.alpha

# we return both the mean function and the standard deviation
if return_std:
    return (mean * self.y_std) + self.y_mean,
             jnp.sqrt(jnp.diag(K * self.y_std**2)),
else:
    return (mean * self.y_std) + self.y_mean, K * self.y_std**2

def sample_y(self, rng_key, X):
    # get posterior mean and covariance
    y_mean, y_cov = self.predict(X)
    # draw one sample
    return jax.random.multivariate_normal(rng_key, mean=y_mean, cov=y_cov)

class ThompsonSamplingGP:
    """Adapted to numpyro from https://gdmarmerola.github.io/ts-for-bayesian-optim/""

    # initialization
    def __init__(self, gp, n_random_draws, objective, x_bounds, grid_resolution=1000, seed=123):
        # Gaussian Process
        self gp = gp

        # number of random samples before starting the optimization
        self.n_random_draws = n_random_draws

        # the objective is the function we're trying to optimize
        self.objective = objective

        # the bounds tell us the interval of x we can work
        self.bounds = x_bounds

        # interval resolution is defined as how many points we will use to
        # represent the posterior sample
        # we also define the x grid
        self.grid_resolution = grid_resolution
        self.X_grid = np.linspace(self.bounds[0], self.bounds[1], self.grid_resolution)

        # also initializing our design matrix and target variable
        self.X = np.array([])
        self.y = np.array([])

        self.rng_key = random.PRNGKey(seed)
# fitting process

def fit(self, X, y, n_step):
    self.rng_key, subkey = random.split(self.rng_key)
    # fitting the GP
    self.gp.fit(X, y, rng_key=subkey, n_step=n_step)

    # return the fitted model
    return self.gp

# choose the next Thompson sample

def choose_next_sample(self, n_step=2_000):
    # if we do not have enough samples, sample randomly from bounds
    if self.X.shape[0] < self.n_random_draws:
        self.rng_key, subkey = random.split(self.rng_key)
        next_sample = random.uniform(subkey, minval=self.bounds[0], maxval=self.bounds[1], shape=(1,))

    # define dummy values for sample, mean and std to avoid errors when returning them
    posterior_sample = np.array([np.mean(self.y)] * self.grid_resolution)
    posterior_mean = np.array([np.mean(self.y)] * self.grid_resolution)
    posterior_std = np.array([0] * self.grid_resolution)

    # if we do, we fit the GP and choose the next point based on the posterior draw
    else:
        # 1. Fit the GP to the observations we have
        self.gp = self.fit(self.X, self.y, n_step=n_step)

        # 2. Draw one sample (a function) from the posterior
        self.rng_key, subkey = random.split(self.rng_key)
        posterior_sample = self.gp.sample_y(subkey, self.X_grid)

        # 3. Choose next point as the optimum of the sample
        which_min = np.argmin(posterior_sample)
        next_sample = self.X_grid[which_min]

        # let us also get the std from the posterior, for visualization purposes
        posterior_mean, posterior_std = self.gp.predict(self.X_grid, return_std=True)

        # let us observe the objective and append this new data to our X and y
        next_observation = self.objective(next_sample)
        self.X = np.append(self.X, next_sample)
        self.y = np.append(self.y, next_observation)

    # returning values of interest
    return (self.X,
def main(args):
    gp = GP(kernel=matern52_kernel)
    # do inference
    thompson = ThompsonSamplingGP(
        gp, n_random_draws=args.num_random, objective=ackley_1d, x_bounds=(-4, 4)
    )

    fig, axes = plt.subplots(
        args.num_samples - args.num_random, 1, figsize=(6, 12), sharex=True, sharey=True
    )
    for i in range(args.num_samples):
        (X, y, X_grid, posterior_sample, posterior_mean, posterior_std, ) = thompson.choose_next_sample(
            n_step=args.num_step,
        )

        if i >= args.num_random:
            ax = axes[i - args.num_random]
            # plot training data
            ax.scatter(X, y, color="blue", marker="o", label="samples")
            ax.axvline(
                X_grid[posterior_sample.argmin()],
                color="blue",
                linestyle="--",
                label="next sample",
            )
            ax.plot(X_grid, ackley_1d(X_grid), color="black", linestyle="--")
            ax.plot(
                X_grid,
                posterior_sample,
                color="red",
                linestyle="-",
                label="posterior sample",
            )
            # plot 90% confidence level of predictions
            ax.fill_between(
                X_grid,
                posterior_mean - posterior_std,
                )

(continues on next page)
posterior_mean + posterior_std,
color="red",
alpha=0.5,
)
ax.set_ylabel("Y")
if i == args.num_samples - 1:
    ax.set_xlabel("X")

plt.legend(
    loc="upper center",
bbox_to_anchor=(0.5, -0.15),
fancybox=True,
    shadow=True,
    ncol=3,
)

fig.suptitle("Thompson sampling")
fig.tight_layout()
plt.show()

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Thompson sampling example")
    parser.add_argument(
        "--num-random", nargs="?", default=2, type=int, help="number of random draws"
    )
    parser.add_argument(
        "--num-samples",
        nargs="?",
        default=10,
        type=int,
        help="number of Thompson samples",
    )
    parser.add_argument(
        "--num-step",
        nargs="?",
        default=2_000,
        type=int,
        help="number of steps for optimization",
    )
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu". ')
    args = parser.parse_args()

    numpyro.set_platform(args.device)

    main(args)
CHAPTER
FORTYFOUR

BAYESIAN HIERARCHICAL STACKING: WELL SWITCHING CASE STUDY

44.1 Table of Contents

- Intro
  - 1. Exploratory Data Analysis
- 2. Prepare 6 Different Models
  - 2.1 Feature Engineering
  - 2.2 Training
  - 2.3 Estimate leave-one-out cross-validated score for each training point
- 3. Bayesian Hierarchical Stacking
  - 3.1 Prepare stacking datasets
  - 3.2 Define stacking model
- 4. Evaluate on test set
  - 4.1 Stack predictions
  - 4.2 Compare methods
- Conclusion
- References

44.2 Intro

Suppose you have just fit 6 models to a dataset, and need to choose which one to use to make predictions on your test set. How do you choose which one to use? A couple of common tactics are: - choose the best model based on cross-validation; - average the models, using weights based on cross-validation scores.

In the paper Bayesian hierarchical stacking: Some models are (somewhere) useful, a new technique is introduced: average models based on weights which are allowed to vary across according to the input data, based on a hierarchical structure.

Here, we’ll implement the first case study from that paper - readers are nonetheless encouraged to look at the original paper to find other cases studies, as well as theoretical results. Code from the article (in R / Stan) can be found here.

[1]: !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro

[2]:

```python
import os
import arviz as az
from IPython.display import set_matplotlib_formats
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.interpolate import BSpline
import seaborn as sns

import jax
import jax.numpy as jnp
```

(continues on next page)
import numpyro
import numpyro.distributions as dist

plt.style.use("seaborn")
if "NUMPYRO_SPHINXBUILD" in os.environ:
    set_matplotlib_formats("svg")
numpyro.set_host_device_count(4)
assert numpyro.__version__.startswith("0.15.2")

matplotlib inline

## 1. Exploratory Data Analysis

The data we have to work with looks at households in Bangladesh, some of which were affected by high levels of arsenic in their water. Would affected households want to switch to a neighbour’s well?

We’ll split the data into a train and test set, and then we’ll train six different models to try to predict whether households would switch wells. Then, we’ll see how we can stack them when predicting on the test set!

But first, let’s load it in and visualise it! Each row represents a household, and the features we have available to us are:

- switch: whether a household switched to another well;
- arsenic: level of arsenic in drinking water;
- educ: level of education of “head of household”;
- dist100: distance to nearest safe-drinking well;
- assoc: whether the household participates in any community activities.

```python
wells = pd.read_csv("http://stat.columbia.edu/~gelman/arm/examples/arsenic/wells.dat", sep=" ")

wells.head()
```

<table>
<thead>
<tr>
<th></th>
<th>switch</th>
<th>arsenic</th>
<th>dist</th>
<th>assoc</th>
<th>educ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2.36</td>
<td>16.82</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.71</td>
<td>47.32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2.07</td>
<td>20.97</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1.15</td>
<td>21.49</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1.10</td>
<td>40.87</td>
<td>1</td>
<td>14</td>
</tr>
</tbody>
</table>

```python
fig, ax = plt.subplots(2, 2, figsize=(12, 6))
fig.suptitle("Target variable plotted against various predictors")
sns.scatterplot(data=wells, x="arsenic", y="switch", ax=ax[0][0])
sns.scatterplot(data=wells, x="dist", y="switch", ax=ax[0][1])
sns.barplot(data=wells.groupby("assoc")["switch"].mean().reset_index(), x="assoc", y="switch",
```
(continues on next page)
ax=ax[1][0],
)
ax[1][0].set_ylabel("Proportion switch")
sns.barplot(
    data=wells.groupby("educ"["switch"]).mean().reset_index(),
    x="educ",
    y="switch",
    ax=ax[1][1],
)
ax[1][1].set_ylabel("Proportion switch");

Next, we’ll choose 200 observations to be part of our train set, and 1500 to be part of our test set.

    np.random.seed(1)
    train_id = wells.sample(n=200).index
    test_id = wells.loc[~wells.index.isin(train_id)].sample(n=1500).index
    y_train = wells.loc[train_id, "switch"].to_numpy()
    y_test = wells.loc[test_id, "switch"].to_numpy()
44.4 2. Prepare 6 different candidate models

44.4.1 2.1 Feature Engineering

First, let's add a few new columns: - edu0: whether educ is 0, - edu1: whether educ is between 1 and 5, - edu2: whether educ is between 6 and 11, - edu3: whether educ is between 12 and 17, - logarsenic: natural logarithm of arsenic, - assoc_half: half of assoc, - as_square: natural logarithm of arsenic, squared, - as_third: natural logarithm of arsenic, cubed, - dist100: dist divided by 100, - intercept: just a columns of 1s.

We're going to start by fitting 6 different models to our train set:

- logistic regression using intercept, arsenic, assoc, edu1, edu2, and edu3;
- same as above, but with logarsenic instead of arsenic;
- same as the first one, but with square and cubic features as well;
- same as the first one, but with spline features derived from logarsenic as well;
- same as the first one, but with spline features derived from dist100 as well;
- same as the first one, but with educ instead of the binary edu variables.

```python
wells["edu0"] = wells["educ"].isin(np.arange(0, 1)).astype(int)
wells["edu1"] = wells["educ"].isin(np.arange(1, 6)).astype(int)
wells["edu2"] = wells["educ"].isin(np.arange(6, 12)).astype(int)
wells["edu3"] = wells["educ"].isin(np.arange(12, 18)).astype(int)
wells["logarsenic"] = np.log(wells["arsenic"])
wells["assoc_half"] = wells["assoc"]/2.0
wells["as_square"] = wells["logarsenic"] ** 2
wells["as_third"] = wells["logarsenic"] ** 3
wells["dist100"] = wells["dist"] / 100.0
wells["intercept"] = 1
```

```python
def bs(x, knots, degree):
    ""
    Generate the B-spline basis matrix for a polynomial spline.

    Parameters
    ----------
    x
        predictor variable.
    knots
        locations of internal breakpoints (not padded).
    degree
        degree of the piecewise polynomial.

    Returns
    -------
    pd.DataFrame
        Spline basis matrix.

    Notes
    -----  
    This mirrors `bs` from splines package in R.
    ""
```

(continues on next page)
padded_knots = np.hstack(
    [[x.min()] * (degree + 1), knots, [x.max()] * (degree + 1)]
)
return pd.DataFrame(
    BSpline(padded_knots, np.eye(len(padded_knots) - degree - 1), degree)(x)[:, 1:],
    index=x.index,
)

knots = np.quantile(wells.loc[train_id, "logarsenic"], np.linspace(0.1, 0.9, num=10))
spline_arsenic = bs(wells["logarsenic"], knots=knots, degree=3)
knots = np.quantile(wells.loc[train_id, "dist100"], np.linspace(0.1, 0.9, num=10))
spline_dist = bs(wells["dist100"], knots=knots, degree=3)

[10]: features_0 = ["intercept", "dist100", "arsenic", "assoc", "edu1", "edu2", "edu3"]
features_1 = ["intercept", "dist100", "logarsenic", "assoc", "edu1", "edu2", "edu3"]
features_2 = [
    "intercept",
    "dist100",
    "arsenic",
    "as_third",
    "as_square",
    "assoc",
    "edu1",
    "edu2",
    "edu3",
]
features_3 = ["intercept", "dist100", "assoc", "edu1", "edu2", "edu3"]
features_4 = ["intercept", "logarsenic", "assoc", "edu1", "edu2", "edu3"]
features_5 = ["intercept", "dist100", "logarsenic", "assoc", "educ"]

X0 = wells.loc[train_id, features_0].to_numpy()
X1 = wells.loc[train_id, features_1].to_numpy()
X2 = wells.loc[train_id, features_2].to_numpy()
X3 = (pd.concat([wells.loc[:, features_3], spline_arsenic], axis=1)
        .loc[train_id]
        .to_numpy()
)
X4 = pd.concat([wells.loc[:, features_4], spline_dist], axis=1).loc[train_id].to_numpy()
X5 = wells.loc[train_id, features_5].to_numpy()

X0_test = wells.loc[test_id, features_0].to_numpy()
X1_test = wells.loc[test_id, features_1].to_numpy()
X2_test = wells.loc[test_id, features_2].to_numpy()
X3_test = (pd.concat([wells.loc[:, features_3], spline_arsenic], axis=1)
            .loc[test_id]
            .to_numpy()
)
X4_test = (pd.concat([wells.loc[:, features_4], spline_dist], axis=1).loc[test_id].to_numpy()
X5_test = wells.loc[test_id, features_5].to_numpy()

train_x_list = [X0, X1, X2, X3, X4, X5]
test_x_list = [X0_test, X1_test, X2_test, X3_test, X4_test, X5_test]
K = len(train_x_list)

### 44.4.2 2.2 Training

Each model will be trained in the same way - with a Bernoulli likelihood and a logit link function.

```python
[12]: def logistic(x, y=None):
    beta = numpyro.sample("beta", dist.Normal(0, 3).expand([x.shape[1]]))
    logits = numpyro.deterministic("logits", jnp.matmul(x, beta))

    numpyro.sample("obs",
                   dist.Bernoulli(logits=logits),
                   obs=y,
    )
```

```python
[13]: fit_list = []
    for k in range(K):
        sampler = numpyro.infer.NUTS(logistic)
        mcmc = numpyro.infer.MCMC(
            sampler, num_chains=4, num_samples=1000, num_warmup=1000, progress_bar=False
        )
        rng_key = jax.random.fold_in(jax.random.PRNGKey(13), k)
        mcmc.run(rng_key, x=train_x_list[k], y=y_train)
        fit_list.append(mcmc)
```

### 44.4.3 2.3 Estimate leave-one-out cross-validated score for each training point

Rather than refitting each model 100 times, we will estimate the leave-one-out cross-validated score using LOO.

```python
[14]: def find_point_wise_loo_score(fit):
    return az.loo(az.from_numpyro(fit), pointwise=True, scale="log").loo_i.values

lpd_point = np.vstack([find_point_wise_loo_score(fit) for fit in fit_list]).T
exp_lpd_point = np.exp(lpd_point)
```

44.4. 2. Prepare 6 different candidate models

627
44.5 3. Bayesian Hierarchical Stacking

44.5.1 3.1 Prepare stacking datasets

To determine how the stacking weights should vary across training and test sets, we will need to create “stacking datasets” which include all the features which we want the stacking weights to depend on. How should such features be included? For discrete features, this is easy, we just one-hot-encode them. But for continuous features, we need a trick. In Equation (16), the authors recommend the following: if you have a continuous feature $f$, then replace it with the following two features:

- $f_l$: $f$ minus the median of $f$, clipped above at 0;
- $f_r$: $f$ minus the median of $f$, clipped below at 0;

```python
[15]: dist100_median = wells.loc[wells.index[train_id], "dist100"].median()
logarsenic_median = wells.loc[wells.index[train_id], "logarsenic"].median()
wells["dist100_l"] = (wells["dist100"] - dist100_median).clip(upper=0)
wells["dist100_r"] = (wells["dist100"] - dist100_median).clip(lower=0)
wells["logarsenic_l"] = (wells["logarsenic"] - logarsenic_median).clip(upper=0)
wells["logarsenic_r"] = (wells["logarsenic"] - logarsenic_median).clip(lower=0)
```

```python
stacking_features = ["edu0", "edu1", "edu2", "edu3", "assoc_half", "dist100_l", "dist100_r", "logarsenic_l", "logarsenic_r",]
X_stacking_train = wells.loc[train_id, stacking_features].to_numpy()
X_stacking_test = wells.loc[test_id, stacking_features].to_numpy()
```

44.5.2 3.2 Define stacking model

What we seek to find is a matrix of weights $W$ with which to multiply the models’ predictions. Let’s define a matrix $P_{red}$ such that $P_{red,i,k}$ represents the prediction made for point $i$ by model $k$. Then the final prediction for point $i$ will then be:

$$\sum_k W_{i,k} P_{red,i,k}$$

Such a matrix $W$ would be required to have each column sum to 1. Hence, we calculate each row $W_i$ of $W$ as:

$$W_i = \text{softmax}(X_{\text{stacking}_i} \cdot \beta),$$

where $\beta$ is a matrix whose values we seek to determine. For the discrete features, $\beta$ is given a hierarchical structure over the possible inputs. Continuous features, on the other hand, get no hierarchical structure in this case study and just vary according to the input values.

Notice how, for the discrete features, a non-centered parametrisation is used. Also note that we only need to estimate K-1 columns of $\beta$, because the weights $W_{i,k}$ will have to sum to 1 for each $i$. 

628 Chapter 44. Bayesian Hierarchical Stacking: Well Switching Case Study
```python
[16]:

def stacking(
    X,
    d_discrete,
    X_test,
    lpd_point,
    tau_mu,
    tau_sigma,
    *,
    test,
):
    ""
    Get weights with which to stack candidate models' predictions.

    Parameters
    ----------
    X
      Training stacking matrix: features on which stacking weights should depend, for the training set.
    d_discrete
      Number of discrete features in `X` and `X_test`. The first `d_discrete` features from these matrices should be the discrete ones, with the continuous ones coming after them.
    X_test
      Test stacking matrix: features on which stacking weights should depend, for the testing set.
    lpd_point
      LOO score evaluated at each point in the training set, for each candidate model.
    tau_mu
      Hyperprior for mean of `beta`, for discrete features.
    tau_sigma
      Hyperprior for standard deviation of `beta`, for continuous features.
    test
      Whether to calculate stacking weights for test set.

    Notes
    -----
    Naming of variables mirrors what's used in the original paper.
    ""
    N = X.shape[0]
    d = X.shape[1]
    N_test = X_test.shape[0]
    K = lpd_point.shape[1]  # number of candidate models

    with numpyro.plate("Candidate models", K - 1, dim=-2):
        # mean effect of discrete features on stacking weights
        mu = numpyro.sample("mu", dist.Normal(0, tau_mu))
        # standard deviation effect of discrete features on stacking weights
        sigma = numpyro.sample("sigma", dist.HalfNormal(scale=tau_sigma))
        with numpyro.plate("Discrete features", d_discrete, dim=-1):
            # effect of discrete features on stacking weights
            tau = numpyro.sample("tau", dist.Normal(0, 1))
        with numpyro.plate("Continuous features", d - d_discrete, dim=-1):
            # effect of continuous features on stacking weights
            #... (continues on next page)
```

44.5. 3. Bayesian Hierarchical Stacking
# effect of continuous features on stacking weights
beta_con = numpyro.sample("beta_con", dist.Normal(0, 1))

# effects of features on stacking weights
beta = numpyro.deterministic(
    "beta", jnp.hstack([(sigma.squeeze() * tau.T + mu.squeeze()).T, beta_con])
)
assert beta.shape == (K - 1, d)

# stacking weights (in unconstrained space)
f = jnp.hstack([X @ beta.T, jnp.zeros((N, 1))])
assert f.shape == (N, K)

# log probability of LOO training scores weighted by stacking weights.
log_w = jax.nn.log_softmax(f, axis=1)

# stacking weights (constrained to sum to 1)
numpyro.deterministic("w", jnp.exp(log_w))
logp = jax.nn.logsumexp(lpd_point + log_w, axis=1)
numpyro.factor("logp", jnp.sum(logp))

if test:
    # test set stacking weights (in unconstrained space)
f_test = jnp.hstack([X_test @ beta.T, jnp.zeros((N_test, 1))])
    # test set stacking weights (constrained to sum to 1)
numpyro.deterministic("w_test", jnp.nn.softmax(f_test, axis=1))

[17]: sampler = numpyro.infer.NUTS(stacking)
mcmc = numpyro.infer.MCMC(
    sampler, num_chains=4, num_samples=1000, num_warmup=1000, progress_bar=False
)
mcmc.run(
    jax.random.PRNGKey(17),
    X=X_stacking_train,
    d_discrete=4,
    X_test=X_stacking_test,
    lpd_point=lpd_point,
    tau_mu=1.0,
    tau_sigma=0.5,
    test=True,
)
trace = mcmc.get_samples()

We can now extract the weights with which to weight the different models from the posterior, and then visualise how they vary across the training set.

Let’s compare them with what the weights would’ve been if we’d just used fixed stacking weights (computed using ArviZ - see their docs for details).

[18]: fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16, 6), sharey=True)
training_stacking_weights = trace["w"].mean(axis=0)
sns.scatterplot(data=pd.DataFrame(training_stacking_weights), ax=ax[0])

(continues on next page)
fixed_weights = (  
    az.compare(idx: fit for idx, fit in enumerate(fit_list)), method="stacking")  
    .sort_index()["weight"]  
    .to_nump()  
)  
fixed_weights_df = pd.DataFrame(  
    np.repeat(  
        fixed_weights[jnp.newaxis, :],  
        len(X_stacking_train),  
        axis=0,  
    )  
)  
sns.scatterplot(data=fixed_weights_df, ax=ax[1])  
ax[0].set_title("Training weights from Bayesian Hierarchical stacking")  
ax[1].set_title("Fixed weights stacking")  
ax[0].set_xlabel("Index")  
ax[1].set_xlabel("Index")  
ax[0].legend(ncol=6, loc="upper center")  
fig.suptitle(  
    "Bayesian Hierarchical Stacking weights can vary according to the input",  
    fontsize=18,  
)  
fig.tight_layout();

44.6 4. Evaluate on test set

44.6.1 4.1 Stack predictions

Now, for each model, let’s evaluate the log predictive density for each point in the test set. Once we have predictions for each model, we need to think about how to combine them, such that for each test point, we get a single prediction.

We decided we’d do this in three ways: - Bayesian Hierarchical Stacking (bhs_pred); - choosing the model with the best training set LOO score (model_selection_preds); - fixed-weights stacking (fixed_weights_preds).
# for each candidate model, extract the posterior predictive logits
train_preds = []
for k in range(K):
    predictive = numpyro.infer.Predictive(logistic, fit_list[k].get_samples())
    rng_key = jax.random.fold_in(jax.random.PRNGKey(19), k)
    train_pred = predictive(rng_key, x=train_x_list[k])['logits']
    train_preds.append(train_pred.mean(axis=0))

# reshape, so we have (N, K)
train_preds = np.vstack(train_preds).T

# same as previous cell, but for test set

test_preds = []
for k in range(K):
    predictive = numpyro.infer.Predictive(logistic, fit_list[k].get_samples())
    rng_key = jax.random.fold_in(jax.random.PRNGKey(20), k)
    test_pred = predictive(rng_key, x=test_x_list[k])['logits']
    test_preds.append(test_pred.mean(axis=0))

test_preds = np.vstack(test_preds).T

# get the stacking weights for the test set


test_stacking_weights = trace['w_test'].mean(axis=0)
# get predictions using the stacking weights
bhs_predictions = (test_stacking_weights * test_preds).sum(axis=1)
# get predictions using only the model with the best LOO score
model_selection_preds = test_preds[:, lpd_point.sum(axis=0).argmax()]
# get predictions using fixed stacking weights, dependent on the LOO score
fixed_weights_preds = (fixed_weights * test_preds).sum(axis=1)

44.6.2 4.2 Compare methods

Let's compare the negative log predictive density scores on the test set (note - lower is better):

fig, ax = plt.subplots(figsize=(12, 6))
neg_log_pred_densities = np.vstack([
    -dist.Bernoulli(logits=bhs_predictions).log_prob(y_test),
    -dist.Bernoulli(logits=model_selection_preds).log_prob(y_test),
    -dist.Bernoulli(logits=fixed_weights_preds).log_prob(y_test),
]).T

neg_log_pred_density = pd.DataFrame(neg_log_pred_densities,
    columns=[
        "Bayesian Hierarchical Stacking",
        "Model selection",
        "Fixed stacking weights",
    ],
)

sns.barplot(data=neg_log_pred_density.reindex(columns=neg_log_pred_density.mean(axis=0).sort_values(ascending=False).index)
(continues on next page)
So, in this dataset, with this particular train-test split, Bayesian Hierarchical Stacking does indeed bring a small gain compared with model selection and compared with fixed-weight stacking.

### 44.6.3 4.3 Does this prove that Bayesian Hierarchical Stacking works?

No, a single train-test split doesn't prove anything. Check the original paper for results with varying training set sizes, repeated with different train-test splits, in which they show that Bayesian Hierarchical Stacking consistently outperforms model selection and fixed-weight stacking.

The goal of this notebook was just to show how to implement this technique in NumPyro.

### 44.7 Conclusion

We've seen how Bayesian Hierarchical Stacking can help us average models with input-dependent weights, in a manner which doesn’t overfit. We only implemented the first case study from the paper, but readers are encouraged to check out the other two as well. Also check the paper for theoretical results and results from more experiments.
44.8 References

1. Yuling Yao, Gregor Pirš, Aki Vehtari, Andrew Gelman (2021). Bayesian hierarchical stacking: Some models are (somewhere) useful

2. Måns Magnusson, Michael Riis Andersen, Johan Jonasson, Aki Vehtari (2020). Leave-One-Out Cross-Validation for Bayesian Model Comparison in Large Data


4. Thomas Wiecki (2017). Why hierarchical models are awesome, tricky, and Bayesian
EXAMPLE: SINE-SKEWED SINE (BIVARIATE VON MISES) MIXTURE

This example models the dihedral angles that occur in the backbone of a protein as a mixture of skewed directional distributions. The backbone angle pairs, called $\phi$ and $\psi$, are a canonical representation for the fold of a protein. In this model, we fix the third dihedral angle (omega) as it usually only takes angles 0 and pi radian, with the latter being the most common. We model the angle pairs as a distribution on the torus using the sine distribution [1] and break point-wise (toroidal) symmetry using sine-skewing [2].

References:
import argparse
import math
from math import pi

import matplotlib.colors
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

from jax import numpy as jnp, random

import numpyro
from numpyro.distributions import (Beta, Categorical, Dirichlet, Gamma, Normal, SineBivariateVonMises, SineSkewed, Uniform, VonMises,
)

from numpyro.distributions.transforms import L1BallTransform

(continues on next page)
from numpyro.examples.datasets import NINE_MERS, load_dataset
from numpyro.infer import MCMC, NUTS, Predictive, init_to_value
from numpyro.infer.reparam import CircularReparam

AMINO_ACIDS = [
    "M",
    "N",
    "I",
    "F",
    "E",
    "L",
    "R",
    "D",
    "G",
    "K",
    "Y",
    "T",
    "H",
    "S",
    "P",
    "A",
    "V",
    "Q",
    "W",
    "C",
]

# The support of the von Mises is [-π, π) with a periodic boundary at ±π. However, the support of
# the implemented von Mises distribution is just the interval [-π, π) without the periodic boundary. If the
# loc is close to one of the boundaries (-π or π), the sampler must traverse the entire interval to cross the
# boundary. This produces a bias, especially if the concentration is high. The interval around
# zero will have a low probability, making the jump to the other boundary unlikely for the sampler.
# Using the `CircularReparam` introduces the periodic boundary by transforming the real line to [-π, π).
# The sampler can sample from the real line, thus crossing the periodic boundary without having to traverse the
# entire interval, which eliminates the bias.
@numpyro.handlers.reparam(config= {
    "phi_loc": CircularReparam(),
    "psi_loc": CircularReparam()
})
def ss_model(data, num_data, num_mix_comp=2):
    # Mixture prior
    mix_weights = numpyro.sample("mix_weights", Dirichlet(jnp.ones((num_mix_comp,))))
    # Hprior BVM
    # Bayesian Inference and Decision Theory by Kathryn Blackmond Laskey
beta_mean_phi = numpyro.sample("beta_mean_phi", Uniform(0.0, 1.0))
beta_count_phi = numpyro.sample(
    "beta_count_phi", Gamma(1.0, 1.0 / num_mix_comp)
) # shape, rate
halpha_phi = beta_mean_phi * beta_count_phi
beta_mean_psi = numpyro.sample("beta_mean_psi", Uniform(0, 1.0))
beta_count_psi = numpyro.sample(
    "beta_count_psi", Gamma(1.0, 1.0 / num_mix_comp)
) # shape, rate
halpha_psi = beta_mean_psi * beta_count_psi

with numpyro.plate("mixture", num_mix_comp):
    # BvM priors
    # Place gap in forbidden region of the Ramachandran plot (protein backbone, dihedral angle pairs)
    phi_loc = numpyro.sample("phi_loc", VonMises(pi, 2.0))
    psi_loc = numpyro.sample("psi_loc", VonMises(0.0, 0.1))

    phi_conc = numpyro.sample(
        "phi_conc", Beta(halpha_phi, beta_count_phi - halpha_phi)
    )
    psi_conc = numpyro.sample(
        "psi_conc", Beta(halpha_psi, beta_count_psi - halpha_psi)
    )
    corr_scale = numpyro.sample("corr_scale", Beta(2.0, 10.0))

    # Skewness prior
    ball_transform = L1BallTransform()
    skewness = numpyro.sample("skewness", Normal(0, 0.5).expand((2,)).to_event(1))
    skewness = ball_transform(skewness)

    with numpyro.plate("obs_plate", num_data, dim=-1):
        assign = numpyro.sample(
            "mix_comp", Categorical(mix_weights), infer={"enumerate": "parallel"}
        )
        sine = SineBivariateVonMises(
            phi_loc=phi_loc[assign],
            psi_loc=psi_loc[assign],
            # These concentrations are an order of magnitude lower than expected (550-1000)!
            phi_concentration=70 * phi_conc[assign],
            psi_concentration=70 * psi_conc[assign],
            weighted_correlation=corr_scale[assign],
        )
        return numpyro.sample("phi_psi", SineSkewed(sine, skewness[assign]), obs=data)

def run_hmc(rng_key, model, data, num_mix_comp, args, bvm_init_locs):
    kernel = NUTS(model, init_strategy=init_to_value(values=bvm_init_locs), max_tree_depth=7)
mcmc = MCMC(kernel, num_samples=args.num_samples, num_warmup=args.num_warmup)
mcmc.run(rng_key, data, len(data), num_mix_comp)
mcmc.print_summary()
post_samples = mcmc.get_samples()
return post_samples

def fetch_aa_dihedrals(aa):
    _, fetch = load_dataset(NINE_MERS, split=aa)
    return jnp.stack(fetch())

def num_mix_comps(amino_acid):
    num_mix = {"G": 10, "P": 7}
    return num_mix.get(amino_acid, 9)

def ramachandran_plot(data, pred_data, aas, file_name="ssbvm_mixture.pdf"):
    amino_acids = {"S": "Serine", "P": "Proline", "G": "Glycine"}
    fig, axss = plt.subplots(2, len(aas))
cdata = data
for i in range(len(axss)):
    if i == 1:
        cdata = pred_data
    for ax, aa in zip(axss[i], aas):
        aa_data = cdata[aa]
nbins = 50
        ax.hexbin(
            aa_data[..., 0].reshape(-1),
            aa_data[..., 1].reshape(-1),
            norm=matplotlib.colors.LogNorm(),
            bins=nbins,
            gridsize=100,
            cmap="Blues",
        )
        # label the contours
        ax.set_aspect("equal", "box")
        ax.set_xlim([-math.pi, math.pi])
        ax.set_ylim([-math.pi, math.pi])
        ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
        ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
        ax.xaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))
        ax.yaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
        ax.yaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
        ax.yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))
    if i == 0:
        axtop = ax.secondary_xaxis("top")
        axtop.set_xlabel(amino_acids[aa])
        axtop.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
        axtop.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
        axtop.xaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))
(continues on next page)
if i == 1:
    ax.set_xlabel(r"$\phi$"

for i in range(len(axss)):
    axss[i, 0].set_ylabel(r"$\psi$"
    axss[i, 0].yaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
    axss[i, 0].yaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
    axss[i, 0].yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))
    axright = axss[i, -1].secondary_yaxis("right")
    axright.set_ylabel("data" if i == 0 else "simulation")
    axright.yaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
    axright.yaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
    axright.yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))

for ax in axss[:, 1:].reshape(-1):
    ax.tick_params(labelleft=False)
    ax.tick_params(labelbottom=False)

for ax in axss[0, :].reshape(-1):
    ax.tick_params(labelbottom=False)
    ax.tick_params(labelleft=False)

if file_name:
    fig.tight_layout()
    plt.savefig(file_name, bbox_inches="tight")

def multiple_formatter(denominator=2, number=np.pi, latex=r"\pi"):
    def gcd(a, b):
        while b:
            a, b = b, a % b
        return a

    def _multiple_formatter(x, pos):
        den = denominator
        num = int(np.rint(den * x / number))
        com = gcd(num, den)
        (num, den) = (int(num / com), int(den / com))
        if den == 1:
            if num == 0:
                return r"$0$"
            if num == 1:
                return r"$%s$" % latex
            elif num == -1:
                return r"$-%s$" % latex
            else:
                return r"$%s%$" % (num, latex)
        else:
            if num == 1:
                return r"\frac{%s}{%s}$" % (latex, den)
            elif num == -1:
                return r"\frac{-%s}{%s}$" % (latex, den)
            else:
                return r"\frac{%s}{%s}$" % (latex, den)

    def gcd(a, b):
        while b:
            a, b = b, a % b
        return a

    def _multiple_formatter(x, pos):
        den = denominator
        num = int(np.rint(den * x / number))
        com = gcd(num, den)
        (num, den) = (int(num / com), int(den / com))
        if den == 1:
            if num == 0:
                return r"$0$"
            if num == 1:
                return r"$%s$" % latex
            elif num == -1:
                return r"$-%s$" % latex
            else:
                return r"$%s%$" % (num, latex)
        else:
            if num == 1:
                return r"\frac{%s}{%s}$" % (latex, den)
            elif num == -1:
                return r"\frac{-%s}{%s}$" % (latex, den)
            else:
                return r"\frac{%s}{%s}$" % (latex, den)
return r"$\frac{-%s}{%s}$" % (latex, den)

else:
    return r"$\frac{%s}{%s}$" % (num, latex, den)

return _multiple_formatter

def main(args):
    data = {}
    pred_datas = {}
    rng_key = random.PRNGKey(args.rng_seed)
    for aa in args.amino_acids:
        rng_key, inf_key, pred_key = random.split(rng_key, 3)
        data[aa] = fetch_aa_dihedrals(aa)
        num_mix_comp = num_mix_comps(aa)

        # Use kmeans to initialize the chain location.
        kmeans = KMeans(num_mix_comp)
        kmeans.fit(data[aa])
        means = {
            "phi_loc": kmeans.cluster_centers_[:, 0],
            "psi_loc": kmeans.cluster_centers_[:, 1],
        }

        posterior_samples = {
            "ss": run_hmc(inf_key, ss_model, data[aa], num_mix_comp, args, means)
        }
        predictive = Predictive(ss_model, posterior_samples["ss"], parallel=True)

        pred_datas[aa] = predictive(pred_key, None, 1, num_mix_comp)["phi_psi"].reshape(-1, 2)

    ramachandran_plot(data, pred_datas, args.amino_acids)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Sine-skewed sine (bivariate von mises) mixture model example"
    )
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=500, type=int)
    parser.add_argument("--amino-acids", nargs="+", default=["S", "P", "G"])
    parser.add_argument("--rng_seed", type=int, default=123)
    parser.add_argument("--device", default="gpu", type=str, help='use "cpu" or "gpu"."")

    args = parser.parse_args()
    assert all(
        aa in AMINO_ACIDS for aa in args.amino_acids
    ), f"\n    \n    list(filter(lambda aa: aa not in AMINO_ACIDS, args.amino_acids)) are not amino acids."
    main(args)
In this example we show how to use `jax.lax.scan` to avoid writing a (slow) Python for-loop. In this toy example, with `--num-data=1000`, the improvement is of almost almost 3x.

To demonstrate, we will be implementing an AR2 process. The idea is that we have some times series

\[ y_0, y_1, \ldots, y_T \]

and we seek parameters \( c, \alpha_1, \) and \( \alpha_2 \) such that for each \( t \) between 2 and \( T \), we have

\[ y_t = c + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \epsilon_t \]

where \( \epsilon_t \) is an error term.
import argparse
import os
import time
import matplotlib.pyplot as plt

import jax
from jax import random
import jax.numpy as jnp

import numpyro
from numpyro.contrib.control_flow import scan
import numpyro.distributions as dist

def ar2_scan(y):
    alpha_1 = numpyro.sample("alpha_1", dist.Normal(0, 1))
    alpha_2 = numpyro.sample("alpha_2", dist.Normal(0, 1))
    const = numpyro.sample("const", dist.Normal(0, 1))
    sigma = numpyro.sample("sigma", dist.HalfNormal(1))

    def transition(carry, _):
        y_prev, y_prev_prev = carry
        m_t = const + alpha_1 * y_prev + alpha_2 * y_prev_prev
        y_t = numpyro.sample("y", dist.Normal(m_t, sigma))
        carry = (y_t, y_prev)
        return carry, m_t

    timesteps = jnp.arange(y.shape[0] - 2)
    init = (y[1], y[0])

    with numpyro.handlers.condition(data={"y": y[2:]})�
        _, mu = scan(transition, init, timesteps)
    numpyro.deterministic("mu", mu)

def ar2_for_loop(y):
    alpha_1 = numpyro.sample("alpha_1", dist.Normal(0, 1))
    alpha_2 = numpyro.sample("alpha_2", dist.Normal(0, 1))
    const = numpyro.sample("const", dist.Normal(0, 1))
    sigma = numpyro.sample("sigma", dist.HalfNormal(1))

    y_prev = y[1]
    y_prev_prev = y[0]
    mu = []

    for i in range(2, len(y)):
        m_t = const + alpha_1 * y_prev + alpha_2 * y_prev_prev
        mu.append(m_t)
        y_t = numpyro.sample("y_{i}".format(i), dist.Normal(m_t, sigma), obs=y[i])
        y_prev_prev = y_prev
        y_prev = y_t
```python
numpyro.deterministic("mu", jnp.array(mu))

def run_inference(model, args, rng_key, y):
    start = time.time()
    sampler = numpyro.infer.NUTS(model)
    mcmc = numpyro.infer.MCMC(
        sampler,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False
    )
    mcmc.run(rng_key, y=y)
    mcmc.print_summary()
    print("MCMC elapsed time:", time.time() - start)
    return mcmc.get_samples()

def main(args):
    # generate artificial dataset
    num_data = args.num_data
    rng_key = jax.random.PRNGKey(0)
    t = jnp.arange(0, num_data)
    y = jnp.sin(t) + random.normal(rng_key, (num_data,)) * 0.1
    # do inference
    if args.unroll_loop:
        # slower
        model = ar2_for_loop
    else:
        # faster
        model = ar2_scan
    samples = run_inference(model, args, rng_key, y)
    # do prediction
    mean_prediction = samples["mu"].mean(axis=0)
    # make plots
    fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)
    # plot training data
    ax.plot(t, y, color="blue", label="True values")
    # plot mean prediction
    # note that we can't make predictions for the first two points,
    # because they don't have lagged values to use for prediction.
    ax.plot(t[2:], mean_prediction, color="orange", label="Mean predictions")
    ax.set(xlabel="time", ylabel="y", title="AR2 process")
    ax.legend()
    plt.savefig("ar2.png")
```

```python
if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="AR2 example")
    parser.add_argument("--num-data", nargs="?", default=142, type=int)
    parser.add_argument("--n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--device", default="cpu", help='use "cpu" or "gpu".')
    parser.add_argument("--unroll-loop",
                        action="store_true",
                        help="whether to unroll for-loop (note: slower)",
                        )
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```
EXAMPLE: HOLT-WINTERS EXPONENTIAL SMOOTHING

In this example we show how to implement Exponential Smoothing. This is intended to be a simple counter-part to the Time Series Forecasting notebook.

The idea is that we have some times series

\[ y_1, \ldots, y_T, y_{T+1}, \ldots, y_{T+H} \]

where we train on \( y_1, \ldots, y_T \) and predict for \( y_{T+1}, \ldots, y_{T+H} \), where \( T \) is the maximum training timestamp and \( H \) is the maximum number of future timesteps for which we want to forecast.

We will be using the update equations from the excellent book Forecasting Principles and Practice:

\[
\begin{align*}
\hat{y}_{t+h|t} &= l_t + hb_t + s_{t+h-m(k+1)} \\
l_t &= \alpha(y_t - s_{t-m}) + (1 - \alpha)(l_{t-1} + b_{t-1}) \\
b_t &= \beta^*(l_t - l_{t-1}) + (1 - \beta^*)b_{t-1} \\
s_t &= \gamma(y_t - l_{t-1} - b_{t-1}) + (1 - \gamma)s_{t-m}
\end{align*}
\]

where

\* \( \hat{y}_t \) is the forecast at time \( t \);
\* \( h \) is the number of time steps into the future which we want to predict for;
\* \( l_t \) is the level, \( b_t \) is the trend, and \( s_t \) is the seasonality,
\* \( \alpha \) is the level smoothing, \( \beta^* \) is the trend smoothing, and \( \gamma \) is the seasonality smoothing.
\* \( k \) is the integer part of \( (h - 1)/m \) (this looks more complicated than it is, it just takes the latest seasonality estimate for this time point).
import argparse
import os
import time

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import jax
from jax import random
import jax.numpy as jnp
import numpyro
from numpyro.contrib.control_flow import scan
from numpyro.diagnostics import hpdi
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, Predictive

matplotlib.use("Agg")

N_POINTS_PER_UNIT = 10  # number of points to plot for each unit interval
```python
def holt_winters(y, n_seasons, future=0):
    T = y.shape[0]
    level_smoothing = numpyro.sample("level_smoothing", dist.Beta(1, 1))
    trend_smoothing = numpyro.sample("trend_smoothing", dist.Beta(1, 1))
    seasonality_smoothing = numpyro.sample("seasonality_smoothing", dist.Beta(1, 1))
    adj_seasonality_smoothing = seasonality_smoothing * (1 - level_smoothing)
    noise = numpyro.sample("noise", dist.HalfNormal(1))
    level_init = numpyro.sample("level_init", dist.Normal(0, 1))
    trend_init = numpyro.sample("trend_init", dist.Normal(0, 1))
    with numpyro.plate("n_seasons", n_seasons):
        seasonality_init = numpyro.sample("seasonality_init", dist.Normal(0, 1))

    def transition_fn(carry, t):
        previous_level, previous_trend, previous_seasonality = carry
        level = jnp.where(
            t < T,
            level_smoothing * (y[t] - previous_seasonality[0])
            + (1 - level_smoothing) * (previous_level + previous_trend),
            previous_level,
        )
        trend = jnp.where(
            t < T,
            trend_smoothing * (level - previous_level)
            + (1 - trend_smoothing) * previous_trend,
            previous_trend,
        )
        new_season = jnp.where(
            t < T,
            adj_seasonality_smoothing * (y[t] - (previous_level + previous_trend))
            + (1 - adj_seasonality_smoothing) * previous_seasonality[0],
            previous_seasonality[0],
        )
        step = jnp.where(t < T, 1, t - T + 1)
        mu = previous_level + step * previous_trend + previous_seasonality[0]
        pred = numpyro.sample("pred", dist.Normal(mu, noise))
        seasonality = jnp.concatenate(
            [previous_seasonality[1:], new_season[None]], axis=0
        )
        return (level, trend, seasonality), pred

    with numpyro.handlers.condition(data={"pred": y}):
        _, preds = scan(
            transition_fn,
            (level_init, trend_init, seasonality_init),
            jnp.arange(T + future),
        )

    if future > 0:
        numpyro.deterministic("y_forecast", preds[-future:])
```

(continues on next page)
def run_inference(model, args, rng_key, y, n_seasons):
    start = time.time()
    sampler = NUTS(model)
    mcmc = MCMC(
        sampler,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, y=y, n_seasons=n_seasons)
    mcmc.print_summary()
    print("\nMCMC elapsed time: ", time.time() - start)
    return mcmc.get_samples()

def predict(model, args, samples, rng_key, y, n_seasons):
    predictive = Predictive(model, samples, return_sites=["y_forecast")
    return predictive(
        rng_key, y=y, n_seasons=n_seasons, future=args.future * N_POINTS_PER_UNIT
    )

def main(args):
    # generate artificial dataset
    rng_key, _ = random.split(random.PRNGKey(0))
    T = args.T
    t = jnp.linspace(0, T + args.future, (T + args.future) * N_POINTS_PER_UNIT)
    y = jnp.sin(2 * np.pi * t) + 0.3 * t + jax.random.normal(rng_key, t.shape) * 0.1
    n_seasons = N_POINTS_PER_UNIT
    y_train = y[: -args.future * N_POINTS_PER_UNIT]
    t_test = t[-args.future * N_POINTS_PER_UNIT :]

    # do inference
    rng_key, _ = random.split(random.PRNGKey(1))
    samples = run_inference(holt_winters, args, rng_key, y_train, n_seasons)

    # do prediction
    rng_key, _ = random.split(random.PRNGKey(2))
    preds = predict(holt_winters, args, samples, rng_key, y_train, n_seasons)
    mean_preds = preds.mean(axis=0)
    hpdi_preds = hpdi(preds)

    # make plots
    fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)
    # plot true data and predictions
    ax.plot(t, y, color="blue", label="True values")
    ax.plot(t_test, mean_preds, color="orange", label="Mean predictions")
    ax.fill_between(t_test, *hpdi_preds, color="orange", alpha=0.2, label="90% CI")
    ax.set(xlabel="time", ylabel="y", title="Holt-Winters Exponential Smoothing")
    ax.legend()
plt.savefig("holt_winters_plot.pdf")

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="Holt-Winters")
    parser.add_argument("--T", nargs="?", default=6, type=int)
    parser.add_argument("--future", nargs="?", default=1, type=int)
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu"."")
    args = parser.parse_args()
    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)
    main(args)
This example is adapted from [1]. The model in the paper estimates death rates for 6791 small areas in England for 19 age groups (0, 1-4, 5-9, 10-14, ..., 80-84, 85+ years) from 2002-19.

When modelling mortality at high spatial resolutions, the number of deaths in each age group, spatial unit and year is small, meaning that death rates calculated from observed data have an apparent variability which is larger than the true differences in the risk of dying. A Bayesian multilevel modelling framework can overcome small number issues by sharing information across ages, space and time to obtain smoothed death rates and capture the uncertainty in the estimate.

As well as a global intercept ($\alpha_0$) and slope ($\beta_0$), the model includes the following effects:

- **Age ($\alpha_2a$, $\beta_2a$).** Each age group has a different intercept and slope with a random-walk structure over age groups to allow for non-linear age associations.

- **Space ($\alpha_1s$).** Each spatial unit has an intercept. The spatial effects are defined by a nested hierarchy of random effects following the administrative hierarchy of local government. The spatial term at the lower level unit is centred on the spatial term of the higher level unit (e.g., $\alpha_{1s_1}$) containing that lower level unit.

The model also has a random walk effect over time ($\pi_t$).

Death rates are linked to the death and population data using a binomial likelihood. The full generative model of death rates is written as

$$
\logit(m_{ast}) = \alpha_{1s_1} + \alpha_{2a} + \beta_{2a}t + \pi_t
$$

with the hyperpriors

$$
\alpha_0 \sim \text{Normal}(0, 10),
\beta_0 \sim \text{Normal}(0, 10),
\sigma_i \sim \text{Half-Normal}(1)
$$

Further detail about the model terms can be found in [1].

The NumPyro implementation below uses \texttt{plate} notation to declare the batch dimensions of the age, space and time variables. This allows us to efficiently broadcast arrays in the likelihood.

As written above, the model includes a lot of centred random effects. The NUTS algorithm benefits from a non-centred reparametrisation to overcome difficult posterior geometries [2]. Rather than manually writing out the non-centred parametrisation, we make use of the NumPyro’s automatic reparametrisation in \texttt{LocScaleReparam}.
Death data at the spatial resolution in [1] are identifiable, so in this example we are using simulated data. Compared to [1], the simulated data have fewer spatial units and a two-tier (rather than three-tier) spatial hierarchy. There are still 19 age groups and 18 years as in the original study. The data here have (event) dimensions of \((19, 113, 18)\) (age, space, time).

The original implementation in nimble is at [3].

**References**


```python
import argparse
import os
import numpy as np
from jax import random
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import MORTALITY, load_dataset
from numpyro.infer import MCMC, NUTS
from numpyro.infer.reparam import LocScaleReparam

def create_lookup(s1, s2):
    """
    Create a map between s1 indices and unique s2 indices
    """
    lookup = np.column_stack([s1, s2])
    lookup = np.unique(lookup, axis=0)
    lookup = lookup[lookup[:, 1].argsort()]
    return lookup[:, 0]

reparam_config = {
    k: LocScaleReparam(0)
    for k in ["alpha_s1", "alpha_s2", "alpha_age_drift", "beta_age_drift", "pi_drift"]
}

@numpyro.handlers.reparam(config=reparam_config)
def model(age, space, time, lookup, population, deaths=None):
    (continues on next page)```
N_s1 = len(np.unique(lookup))
N_s2 = len(np.unique(space))
N_age = len(np.unique(age))
N_t = len(np.unique(time))
N = len(population)

# plates
age_plate = numpyro.plate("age_groups", N_age, dim=-3)
space_plate = numpyro.plate("space", N_s2, dim=-2)
year_plate = numpyro.plate("year", N_t - 1, dim=-1)

# hyperparameters
sigma_alpha_s1 = numpyro.sample("sigma_alpha_s1", dist.HalfNormal(1.0))
sigma_alpha_s2 = numpyro.sample("sigma_alpha_s2", dist.HalfNormal(1.0))
sigma_alpha_age = numpyro.sample("sigma_alpha_age", dist.HalfNormal(1.0))
sigma_beta_age = numpyro.sample("sigma_beta_age", dist.HalfNormal(1.0))
sigma_pi = numpyro.sample("sigma_pi", dist.HalfNormal(1.0))

# spatial hierarchy
with numpyro.plate("s1", N_s1, dim=-2):
    alpha_s1 = numpyro.sample("alpha_s1", dist.Normal(0, sigma_alpha_s1))
    with space_plate:
        alpha_s2 = numpyro.sample("alpha_s2", dist.Normal(alpha_s1[lookup], sigma_alpha_s2))

# age
with age_plate:
    alpha_age_drift_scale = jnp.pad(  
        jnp.broadcast_to(sigma_alpha_age, N_age - 1),  
        (1, 0),  
        constant_values=10.0,  
    )[:, jnp.newaxis, jnp.newaxis]
    alpha_age_drift = numpyro.sample("alpha_age_drift", dist.Normal(0, alpha_age_drift_scale))
    alpha_age = jnp.cumsum(alpha_age_drift, -3)

    beta_age_drift_scale = jnp.pad(  
        jnp.broadcast_to(sigma_beta_age, N_age - 1), (1, 0), constant_values=10.0  
    )[:, jnp.newaxis, jnp.newaxis]
    beta_age_drift = numpyro.sample("beta_age_drift", dist.Normal(0, beta_age_drift_scale))
    beta_age = jnp.cumsum(beta_age_drift, -3)
    beta_age_cum = jnp.outer(beta_age, jnp.arange(N_t))[:, jnp.newaxis, :]

# random walk over time
with year_plate:
    pi_drift = numpyro.sample("pi_drift", dist.Normal(0, sigma_pi))
    pi = jnp.pad(jnp.cumsum(pi_drift, -1), (1, 0))
# likelihood
latent_rate = alpha_age + beta_age_cum + alpha_s2 + pi
with numpyro.plate("N", N):
    mu_logit = latent_rate[age, space, time]
numpyro.sample("deaths", dist.Binomial(population, logits=mu_logit), obs=deaths)

def print_model_shape(model, age, space, time, lookup, population):
    with numpyro.handlers.seed(rng_seed=1):
        trace = numpyro.handlers.trace(model).get_trace(
            age=age,
            space=space,
            time=time,
            lookup=lookup,
            population=population,
        )
        print(numpyro.util.format_shapes(trace))

def run_inference(model, age, space, time, lookup, population, deaths, rng_key, args):
    kernel = NUTS(model)
mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
mcmc.run(rng_key, age, space, time, lookup, population, deaths)
mcmc.print_summary()
return mcmc.get_samples()

def main(args):
    print("Fetching simulated data...")
    _, fetch = load_dataset(MORTALITY, shuffle=False)
a, s1, s2, t, deaths, population = fetch()

lookups = create_lookup(s1, s2)

print("Model shape:")
print_model_shape(model, a, s2, t, lookup, population)

print("Starting inference...")
rng_key = random.PRNGKey(args.rng_seed)
run_inference(model, a, s2, t, lookup, population, deaths, rng_key, args)

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argp.parse.ArgumentParser(description="Mortality regression model")
parser.add_argument("-n", "--num-samples", nargs="?", default=500, type=int)
parser.add_argument("--num-warmup", nargs="?", default=200, type=int)
parser.add_argument("--num-chains", nargs="?", default=1, type=int)
parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu"."
parser.add_argument("--rng_seed", default=21, type=int, help="random number generator seed"

args = parser.parse_args()
numpyro.set_platform(args.device)
numpyro.enable_x64()

main(args)
Chapter 48. Example: Modelling mortality over space and time
EXAMPLE: ZERO-INFLATED POISSON REGRESSION MODEL

In this example, we model and predict how many fish are caught by visitors to a state park. Many groups of visitors catch zero fish, either because they did not fish at all or because they were unlucky. We would like to explicitly model this bimodal behavior (zero versus non-zero) and ascertain which variables contribute to each behavior.

We answer this question by fitting a zero-inflated poisson regression model. We use MAP, VI and MCMC as estimation methods. Finally, from the MCMC samples, we identify the variables that contribute to the zero and non-zero components of the zero-inflated poisson likelihood.

```python
import argparse
import os
import random
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error

import jax.numpy as jnp
from jax.random import PRNGKey
import jax.scipy as jsp

import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS, SVI, Predictive, Trace_ELBO, autoguide

matplotlib.use("Agg")  # noqa: E402

def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)

def model(X, Y):
    D_X = X.shape[1]
    b1 = jnp.random.sample("b1", dist.Normal(0.0, 1.0).expand([D_X]).to_event(1))
    b2 = jnp.random.sample("b2", dist.Normal(0.0, 1.0).expand([D_X]).to_event(1))

    q = jsp.special.expit(jnp.dot(X, b1[:, None])).reshape(-1)
    lam = jnp.exp(jnp.dot(X, b2[:, None])).reshape(-1)
```

(continues on next page)
with numpyro.plate("obs", X.shape[0]):
    numpyro.sample("Y", dist.ZeroInflatedPoisson(gate=q, rate=lam), obs=Y)

def run_mcmc(model, args, X, Y):
    kernel = NUTS(model)
    mcmc = MCMC(
        kernel,
        num_warmup=args.num_warmup,
        num_samples=args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(PRNGKey(1), X, Y)
    mcmc.print_summary()
    return mcmc.get_samples()

def run_svi(model, guide_family, args, X, Y):
    if guide_family == "AutoDelta":
        guide = autoguide.AutoDelta(model)
    elif guide_family == "AutoDiagonalNormal":
        guide = autoguide.AutoDiagonalNormal(model)
    optimizer = numpyro.optim.Adam(0.001)
    svi = SVI(model, guide, optimizer, Trace_ELBO())
    svi_results = svi.run(PRNGKey(1), args.maxiter, X=X, Y=Y)
    params = svi_results.params
    return params, guide

def main(args):
    set_seed(args.seed)

    # prepare dataset
    df = pd.read_stata("http://www.stata-press.com/data/r11/fish.dta")
    df["intercept"] = 1
    cols = ["livebait", "camper", "persons", "child", "intercept"]

    mask = np.random.randn(len(df)) < args.train_size
    df_train = df[mask]
    df_test = df[-mask]
    X_train = jnp.asarray(df_train[cols].values)
    y_train = jnp.asarray(df_train["count"].values)
    X_test = jnp.asarray(df_test[cols].values)
    y_test = jnp.asarray(df_test["count"].values)

    print("run MAP.")
    map_params, map_guide = run_svi(model, "AutoDelta", args, X_train, y_train)
print("run VI.")
vi_params, vi_guide = run_svi(model, "AutoDiagonalNormal", args, X_train, y_train)
print("run MCMC.")
posterior_samples = run_mcmc(model, args, X_train, y_train)

# evaluation

def svi_predict(model, guide, params, args, X):
    predictive = Predictive(
        model=model, guide=guide, params=params, num_samples=args.num_samples
    )
    predictions = predictive(PRNGKey(1), X=X, Y=None)
    svi_predictions = jnp.rint(predictions["Y"].mean(0))
    return svi_predictions

map_predictions = svi_predict(model, map_guide, map_params, args, X_test)
vi_predictions = svi_predict(model, vi_guide, vi_params, args, X_test)
predictive = Predictive(model, posterior_samples=posterior_samples)
predictions = predictive(PRNGKey(1), X=X_test, Y=None)
mcmc_predictions = jnp.rint(predictions["Y"].mean(0))

print(
    "MAP RMSE: ",
    mean_squared_error(y_test.to_py(), map_predictions.to_py(), squared=False),
)
print(
    "VI RMSE: ",
    mean_squared_error(y_test.to_py(), vi_predictions.to_py(), squared=False),
)
print(
    "MCMC RMSE: ",
    mean_squared_error(y_test.to_py(), mcmc_predictions.to_py(), squared=False),
)

# make plot
fig, axes = plt.subplots(2, 1, figsize=(6, 6), constrained_layout=True)

def add_fig(var_name, title, ax):
    ax.set_title(title)
    ax.violinplot(
        [posterior_samples[var_name][:, i].to_py() for i in range(len(cols))]
    )
    ax.set_xticks(np.arange(1, len(cols) + 1))
    ax.set_xticklabels(cols, rotation=45, fontsize=10)

    add_fig("b1", "Coefficients for probability of catching fish", axes[0])
    add_fig("b2", "Coefficients for the number of fish caught", axes[1])

plt.savefig("zip_fish.png")
if __name__ == "__main__":
    parser = argparse.ArgumentParser("Zero-Inflated Poisson Regression")
    parser.add_argument("--seed", nargs="?", default=42, type=int)
    parser.add_argument("-n", "--num-samples", nargs="?", default=2000, type=int)
    parser.add_argument("--num-warmup", nargs="?", default=1000, type=int)
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument("--num-data", nargs="?", default=100, type=int)
    parser.add_argument("--maxiter", nargs="?", default=5000, type=int)
    parser.add_argument("--train-size", nargs="?", default=0.8, type=float)
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
EXAMPLE: CONDITIONAL VARIATIONAL AUTOENCODER IN FLAX

This example trains a *Conditional Variational Autoencoder* (CVAE) [1] on the MNIST data using Flax’ neural network API. The implementation can be found here: https://github.com/pyro-ppl/numpyro/tree/master/examples/cvae-flax

The model is a port of Pyro’s excellent CVAE example which describes the model as well as the data in detail: https://pyro.ai/examples/cvae.html

The model first trains a baseline to predict an entire MNIST image from a single quadrant of it (i.e., input is one quadrant of an image, output is the entire image (not the other three quadrants)). Then, in a second model, the generation/prior/recognition nets of the CVAE are trained while keeping the model parameters of the baseline fixed/frozen. We use Optax’ *multi_transform* to apply different gradient transformations to the trainable parameters and the frozen parameters.

![Image of MNIST examples](image.png)

References:

TEXT-BASED IDEAL POINTS USING NUMPYRO

51.1 Szymon Sacher & Keyon Vafa This notebook replicates Text-Based Ideal Point model (Vafa, Naidu & Blei, 2020)

This notebook is designed to run on Google Colab.

**IMPORTANT:** To save this code and your results, make sure you copy to your personal Google Drive. Under “File”, select “Save a copy in Drive”.

Use this Colab notebook to run a NumPyro implementation of the text-based ideal point model (TBIP) on a corpus of political text. The Github repository is more complete.

See also Tensorflow implementation on which this notebook is based.

The TBIP is an unsupervised probabilistic topic model that analyzes texts to quantify the political positions of its authors. The model does not use political parties or votes, nor does it require any text labelled by ideology. Given a corpus of political text and the authors of each document, the TBIP estimates the latent political positions of the authors of texts and how per-topic word choice changes as a function of the political position of the author (“ideological topics”). Refer to the paper for more information.

51.2 Getting started

First, make sure you are running this Colab using a GPU. Go to the “Runtime” menu, and click “Change runtime type”. If the “Hardware accelerator” is listed as “None” or “TPU”, change to “GPU”. Click “Save” and you’re ready to go. Also, as described in the first cell, make sure this code is copied to your personal Google Drive.

51.3 Install NumPyro

NumPyro is a probabilistic programming framework powered by JAX for autograd and JIT compilation to GPU/TPU/CPU.

```
[1]: %capture
%pip install numpyro==0.10.1
%pip install optax
```
51.4 Clone TBIP repository

Below we clone the Github repo for the TBIP. This is where the data resides.

```
[2]: ! git clone https://github.com/keyonvafa/tbip
fatal: destination path 'tbip' already exists and is not an empty directory.
```

51.5 Hyperparameters and Initialization

We start setting some hyperparameters. We fix the number of topics $K = 50$. We also set a random seed for reproducibility.

```
[3]: from jax import random
num_topics = 50
rng_seed = random.PRNGKey(0)
```

The next cell provides the data directory. The directory in the cell below links to speeches from the 114th Senate session from the tbip repo.

To use your own corpus, upload the following four files to the Colab working directory:

- **counts.npz**: a $[\text{num_documents}, \text{num_words}]$ sparse CSR matrix containing the word counts for each document.
- **author_indices.npy**: a $[\text{num_documents}]$ vector where each entry is an integer in the set $\{0, 1, \ldots, \text{num_authors} - 1\}$, indicating the author of the corresponding document in counts.npz.
- **vocabulary.txt**: a $[\text{num_words}]$-length file where each line denotes the corresponding word in the vocabulary.
- **author_map.txt**: a $[\text{num_authors}]$-length file where each line denotes the name of an author in the corpus.

See Senate speech clean data for an example of what the four files look like for Senate speeches. Our setup script contains example code for creating the four files from unprocessed data for Senate speeches.

**IMPORTANT**: If you are using your own corpus, change the following line to `data_dir = '.'` after uploading the four files to the Colab working directory.

```
[4]: import numpy as np
from scipy import sparse
import jax
import jax.numpy as jnp

dataPath = "tbip/data/senate-speeches-114/clean/

# Load data
author_indices = jax.device_put(jnp.load(dataPath + "author_indices.npz"), jax.device('gpu')[0])

counts = sparse.load_npz(dataPath + "counts.npz")
```

(continues on next page)
with open(dataPath + "vocabulary.txt", "r") as f:
    vocabulary = f.readlines()

with open(dataPath + "author_map.txt", "r") as f:
    author_map = f.readlines()

author_map = np.array(author_map)

num_authors = int(author_indices.max() + 1)
num_documents, num_words = counts.shape

In the paper, the parameters are pre-initialized with Poisson factorization. Most of the time, we find this doesn’t make a big difference for the learned ideal points, but it helps to interpret the ideological topics.

Below, we initialize with Scikit-Learn’s non-negative matrix factorization (NMF) implementation. Although we find that Poisson factorization learns more interpretable topics, we use Scikit-Learn’s NMF implementation here because it is faster. To use Poisson factorization, see the code in the Github repo.

If you would like to skip this pre-initialization step, set pre_initialize_parameters = False in the cell below. (Pre-initialization is recommended.)

[5]: pre_initialize_parameters = True

If you are pre-initializing parameters, the following cell might take a minute or so.

[6]: # Fit NMF to be used as initialization for TBIP
from sklearn.decomposition import NMF

if pre_initialize_parameters:
    nmf_model = NMF(
        n_components=num_topics, init="random", random_state=0, max_iter=500
    )
    # Define initialization arrays
    initial_document_loc = jnp.log(
        jnp.array(np.float32(nmf_model.fit_transform(counts) + 1e-2))
    )
    initial_objective_topic_loc = jnp.log(
        jnp.array(np.float32(nmf_model.components_ + 1e-2))
    )
else:
    rng1, rng2 = random.split(rng_seed, 2)
    initial_document_loc = random.normal(rng1, shape=(num_documents, num_topics))
    initial_objective_topic_loc = random.normal(rng2, shape=(num_topics, num_words))
51.6 Perform Inference

We perform inference using variational inference with reparameterization gradients. We provide a brief summary below, but encourage readers to refer to the original paper for a more complete overview.

It is intractable to evaluate the posterior distribution \( p(\theta, \beta, \eta, x | y) \), so we approximate the posterior with a distribution \( q_{\phi}(\theta, \beta, \eta, x) \), parameterized by \( \phi \). How do we set the values \( \phi \)? We want to minimize the KL-Divergence between \( q \) and the posterior, which is equivalent to maximizing the ELBO:

\[
E_q[\log p(y, \theta, \beta, \eta, x) - \log q_{\phi}(\theta, \beta, \eta, x)].
\]

We set the variational family to be the mean-field family, meaning the latent variables factorize over documents \( d \), topics \( k \), and authors \( s \):

\[
q_{\phi}(\theta, \beta, \eta, x) = \prod_{d,k,s} q(\theta_d)q(\beta_k)q(\eta_k)q(x_s).
\]

We use lognormal factors for the positive variables and Gaussian factors for the real variables:

\[
q(\theta_{dk}) = \text{LogNormal}(\mu_{\theta_{dk}}, \sigma_{\theta_{dk}}^2)
\]

\[
q(\beta_{kv}) = \text{LogNormal}(\mu_{\beta_{kv}}, \sigma_{\beta_{kv}}^2)
\]

\[
q(\eta_{kv}) = \text{Normal}(\mu_{\eta_{kv}}, \sigma_{\eta_{kv}}^2)
\]

\[
q(x_s) = \text{Normal}(\mu_x, \sigma_x^2).
\]

Thus, our goal is to maximize the ELBO with respect to \( \phi = \{ \mu_\theta, \sigma_\theta, \mu_\beta, \sigma_\beta, \mu_\eta, \sigma_\eta, \mu_x, \sigma_x \} \).

In the cell below, we define the model and the variational family (guide).

```python
from numpyro import param, plate, sample
import numpyro.distributions as dist
from numpyro.distributions import constraints

# Define the model and variational family

class TBIP:
    def __init__(self, N, D, K, V, batch_size, init_mu_theta=None, init_mu_beta=None):
        self.N = N  # number of people
        self.D = D  # number of documents
        self.K = K  # number of topics
        self.V = V  # number of words in vocabulary
        self.batch_size = batch_size  # number of documents in a batch

        if init_mu_theta is None:
            init_mu_theta = jnp.zeros([D, K])
        else:
            self.init_mu_theta = init_mu_theta

        if init_mu_beta is None:
            init_mu_beta = jnp.zeros([K, V])
        else:
            self.init_mu_beta = init_mu_beta
```

(continues on next page)
def model(self, Y_batch, d_batch, i_batch):
    with plate("i", self.N):
        # Sample the per-unit latent variables (ideal points)
        x = sample("x", dist.Normal())

    with plate("k", size=self.K, dim=-2):
        with plate("k_v", size=self.V, dim=-1):
            beta = sample("beta", dist.Gamma(0.3, 0.3))
            eta = sample("eta", dist.Normal())

    with plate("d", size=self.D, subsample_size=self.batch_size, dim=-2):
        with plate("d_k", size=self.K, dim=-1):
            # Sample document-level latent variables (topic intensities)
            theta = sample("theta", dist.Gamma(0.3, 0.3))

            # Compute Poisson rates for each word
            P = jnp.sum(
                jnp.expand_dims(theta, 2)
                * jnp.expand_dims(beta, 0)
                * jnp.exp(
                    jnp.expand_dims(x[i_batch], (1, 2)) * jnp.expand_dims(eta, 0)
                ),
                1,
            )

    with plate("v", size=self.V, dim=-1):
        # Sample observed words
        sample("Y_batch", dist.Poisson(P), obs=Y_batch)

def guide(self, Y_batch, d_batch, i_batch):
    # This defines variational family. Notice that each of the latent variables
    # defined in the sample statements in the model above has a corresponding
    # sample statement in the guide. The guide is responsible for providing
    # variational parameters for each of these latent variables.

    # Also notice it is required that model and the guide have the same call.

    mu_x = param(
        "mu_x", init_value=-1 + 2 * random.uniform(random.PRNGKey(1), (self.N,))
    )
    sigma_x = param(
        "sigma_y", init_value=jnp.ones([self.N]), constraint=constraints.positive
    )

    mu_eta = param(
        "mu_eta", init_value=random.normal(random.PRNGKey(2), (self.K, self.V))
    )
    sigma_eta = param(
        "sigma_eta",
        init_value=jnp.ones([self.K, self.V]),
        constraint=constraints.positive,
        (continues on next page)
num_theta = param("mu_theta", init_value=self.init_mu_theta)
sigma_theta = param(
    "sigma_theta",
    init_value=jnp.ones([self.D, self.K]),
    constraint=constraints.positive,
)

mu_beta = param("mu_beta", init_value=self.init_mu_beta)
sigma_beta = param(
    "sigma_beta",
    init_value=jnp.ones([self.K, self.V]),
    constraint=constraints.positive,
)

with plate("i", self.N):
    sample("x", dist.Normal(mu_x, sigma_x))

with plate("k", size=self.K, dim=-2):
    with plate("k_v", size=self.V, dim=-1):
        sample("beta", dist.LogNormal(mu_beta, sigma_beta))
        sample("eta", dist.Normal(mu_eta, sigma_eta))

with plate("d", size=self.D, subsample_size=self.batch_size, dim=-2):
    with plate("d_k", size=self.K, dim=-1):
        sample("theta", dist.LogNormal(mu_theta[d_batch], sigma_theta[d_batch]))

def get_batch(self, rng, Y, author_indices):
    # Helper functions to obtain a batch of data, convert from scipy.sparse
    # to jax.numpy.array and move to gpu

    D_batch = random.choice(rng, jnp.arange(self.D), shape=(self.batch_size,))
    Y_batch = jax.device_put(jnp.array(Y[D_batch].toarray()), jax.devices("gpu")[0])
    D_batch = jax.device_put(D_batch, jax.devices("gpu")[0])
    I_batch = author_indices[D_batch]
    return Y_batch, I_batch, D_batch

51.7 Initialization

Below we initialize an instance of the TBIP model, and associated SVI object. The latter is used to compute the Evidence Lower Bound (ELBO) given current value of guide’s parameters and current batch of data.

We optimize the model using Adam with exponential decay of learning rate.

[8]: 
# Initialize the model
from jax import jit
from optax import adam, exponential_decay

from numpyro.infer import SVI, TraceMeanField_ELBO
(continues on next page)
num_steps = 50000
batch_size = 512  # Large batches are recommended
learning_rate = 0.01
decay_rate = 0.01

tbip = TBIP(
    N=num_authors,
    D=num_documents,
    K=num_topics,
    V=num_words,
    batch_size=batch_size,
    init_mu_theta=initial_document_loc,
    init_mu_beta=initial_objective_topic_loc,
)

svi_batch = SVI(
    model=tbip.model,
    guide=tbip.guide,
    optim=adam(exponential_decay(learning_rate, num_steps, decay_rate)),
    loss=TraceMeanField_ELBO(),
)

# Compile update function for faster training
svi_batch_update = jit(svi_batch.update)

# Get initial batch. This informs the dimension of arrays and ensures they are
# consistent with dimensions (N, D, K, V) defined above.
Y_batch, I_batch, D_batch = tbip.get_batch(random.PRNGKey(1), counts, author_indices)

# Initialize the parameters using initial batch
svi_state = svi_batch.init(
    random.PRNGKey(0), Y_batch=Y_batch, d_batch=D_batch, i_batch=I_batch
)

[9]: # @title Run this cell to create helper function for printing topics

def get_topics(
    neutral_mean, negative_mean, positive_mean, vocabulary, print_to_terminal=True
):
    num_topics, num_words = neutral_mean.shape
    words_per_topic = 10
    top_neutral_words = np.argsort(-neutral_mean, axis=1)
    top_negative_words = np.argsort(-negative_mean, axis=1)
    top_positive_words = np.argsort(-positive_mean, axis=1)
    topic_strings = []
    for topic_idx in range(num_topics):
        neutral_start_string = "Neutral  {}":format(topic_idx)
        neutral_row = [
            vocabulary[word] for word in top_neutral_words[topic_idx, :words_per_topic]
        ]
numeral_row_string = ",\".join(numeral_row)
numeral_string = ",\".join([numeral_start_string, numeral_row_string])

positive_start_string = "Positive {}:\".format(topic_idx)
positive_row = [
  vocabulary[word] for word in top_positive_words[topic_idx, :words_per_topic]
]
positive_row_string = ",\".join(positive_row)
positive_string = ",\".join([positive_start_string, positive_row_string])

negative_start_string = "Negative {}:\".format(topic_idx)
negative_row = [
  vocabulary[word] for word in top_negative_words[topic_idx, :words_per_topic]
]
negative_row_string = ",\".join(negative_row)
negative_string = ",\".join([negative_start_string, negative_row_string])

if print_to_terminal:
    topic_strings.append(negative_string)
    topic_strings.append(numeral_string)
    topic_strings.append(positive_string)
    topic_strings.append("==========")
else:
    topic_strings.append(
        
        \n        ,\".join([negative_string, numeral_string, positive_string])
    )

if print_to_terminal:
    all_topics = 
        \n        .format(np.array(topic_strings))
else:
    all_topics = np.array(topic_strings)
return all_topics

51.8 Execute Training

The code above was creating the model; below we actually run training. You can adjust the number of steps to train (num_steps, defined above) and the frequency at which to print the ELBO (print_steps, defined below).

Here, we run our training loop. Topic summaries and ordered ideal points will print every 2500 steps. Typically in our experiments it takes 15,000 steps or so to begin seeing sensible results, but of course this depends on the corpus. These sensible results should be reached within a half hour. For the default corpus of Senate speeches, it should take less than 2 hours to complete 30,000 training steps (which is the default num_steps).

[10]: # Run SVI
   import pandas as pd
   from tqdm import tqdm

   print_steps = 100
   print_intermediate_results = False

(continues on next page)
rngs = random.split(random.PRNGKey(2), num_steps)
losses = []
pbar = tqdm(range(num_steps))

for step in pbar:
    Y_batch, I_batch, D_batch = tbip.get_batch(rngs[step], counts, author_indices)
    svi_state, loss = svi_batch_update(svi_state, Y_batch=Y_batch, d_batch=D_batch, i_batch=I_batch)
    loss = loss / counts.shape[0]
    losses.append(loss)

    if step % print_steps == 0 or step == num_steps - 1:
        pbar.set_description(
            "Init loss: "
            + ":10.4f"".format(jnp.array(losses[0]))
            + ":10.4f"".format(jnp.array(losses[-100:]).mean())
        )

    if (step + 1) % 2500 == 0 or step == num_steps - 1:
        # Save intermediate results
        estimated_params = svi_batch.get_params(svi_state)

        neutral_mean = (estimated_params["mu_beta"] + estimated_params["sigma_beta"] ** 2 / 2)

        positive_mean = (estimated_params["mu_beta"]
                         + estimated_params["mu_eta"]
                         + (estimated_params["sigma_beta"] ** 2 + estimated_params["sigma_eta"] ** 2) / 2)

        negative_mean = (estimated_params["mu_beta"]
                         - estimated_params["mu_eta"]
                         + (estimated_params["sigma_beta"] ** 2 + estimated_params["sigma_eta"] ** 2) / 2)

        np.save("neutral_topic_mean.npy", neutral_mean)
        np.save("negative_topic_mean.npy", positive_mean)
        np.save("positive_topic_mean.npy", negative_mean)

        topics = get_topics(neutral_mean, positive_mean, negative_mean, vocabulary)

        with open("topics.txt", "w") as f:
            print(topics, file=f)

(continues on next page)
authors = pd.DataFrame(
    {
        "name": author_map, 
        "ideal_point": np.array(estimated_params["mu_x"])
    }
)
authors.to_csv("authors.csv")

if print_intermediate_results:
    print(f"Results after {step} steps.
    print(topics)
    sorted_authors = "Authors sorted by their ideal points: " + ",".join(
        list(authors.sort_values("ideal_point")["name"])
    )
    print(sorted_authors.replace("\n", " "))

Init loss: 14323.9385; Avg loss (last 100 iter): 953.5815: 5%|\2499/
˓50000 [03:47<1:11:33, 11.06it/s]/usr/local/lib/python3.7/dist-packages/jax/_src/numpy/
˓lax_numpy.py:3327: UserWarning: 'kind' argument to argsort is ignored; only 'stable'
˓sorts are supported.
˓warnings.warn("'kind' argument to argsort is ignored; only 'stable' sorts 
Init loss: 14323.9385; Avg loss (last 100 iter): 634.0942: 100
˓%|\50000/50000 [1:17:57<00:00, 10.69it/s]

[11]: import matplotlib.pyplot as plt
import seaborn as sns
neutral_topic_mean = np.load("neutral_topic_mean.npy")
negative_topic_mean = np.load("negative_topic_mean.npy")
positive_topic_mean = np.load("positive_topic_mean.npy")
authors = pd.read_csv("authors.csv")
authors["name"] = authors["name"].str.replace("\n", "")

For example, here is a graph of the learned ideal points. We don’t label each point because there are too many to plot. Below we select some authors to label.

[12]: selected_authors = np.array(

    ["Dean Heller (R)",
     "Bernard Sanders (I)",
     "Elizabeth Warren (D)",
     "Charles Schumer (D)",
     "Susan Collins (R)",
     "Marco Rubio (R)",
     "John McCain (R)",
     "Ted Cruz (R)",
    ]
)
sns.set(style=\"whitegrid\")
fig = plt.figure(figsize=(12, 1))
ax = plt.axes([0, 0, 1, 1], frameon=False)
for index in range(authors.shape[0]):
    ax.scatter(authors["ideal_point"][index], 0, c="black", s=20)
    if authors["name"][index] in selected_authors:
    continue
    ax.text(authors["ideal_point"][index], 0, authors["name"][index], ha=\"center\", va=\"bottom\")

(continues on next page)
51.9 Automatic guide generation

Above, for pedagogical reasons, we defined the guide (i.e. Variational Family) manually, making sure it is identical to that described in the original paper.

However, manually defining the guide is often unnecessary as NumPyro contains a module that enables automatic guide generation based on the model provided.

In our case it turns out that AutoNormal creates a guide that is identical to the one we manually defined above. Specifically it first transform variables to belong to unrestricted space of real numbers. For example, by applying the log transformation to those variables that are restricted to be non-negative (document locations $\theta_d$ and objective topic locations $\beta_j$). Then it uses independent Normal distribution as a guide for each transformed distribution.

In the cell below we verify that the guide generated by AutoNormal is in fact identical to the guide manually defined above as a part of the TBIP class.

[13]: from numpyro.infer.autoguide import AutoNormal

```python
def create_svi_object(guide):
    SVI(
        model=tbip.model,
        guide=guide,
        optim=adam(exponential_decay(learning_rate, num_steps, decay_rate)),
        loss=TraceMeanField_ELBO(),
    )

    Y_batch, I_batch, D_batch = tbip.get_batch(
        random.PRNGKey(1), counts, author_indices
    )

    svi_state = svi_batch.init(
        random.PRNGKey(0), Y_batch=Y_batch, d_batch=D_batch, i_batch=I_batch
    )
```
return svi_state

# This state uses the guide defined manually above
svi_state_manualguide = create_svi_object(guide=tbip.guide)

# Now let's create this object but using AutoNormal guide. We just need to ensure that
# parameters are initialized as above.
autoguide = AutoNormal(
    model=tbip.model,
    init_loc_fn={'beta': initial_objective_topic_loc, 'theta': initial_document_loc},
)
svi_state_autoguide = create_svi_object(guide=autoguide)

# Assert that the keys in the optimizer states are identical
assert svi_state_manualguide[0][1][0].keys() == svi_state_autoguide[0][1][0].keys()

# Assert that the values in the optimizer states are identical
for key in svi_state_manualguide[0][1][0].keys():
    assert jnp.all(
        svi_state_manualguide[0][1][0][key] == svi_state_autoguide[0][1][0][key]
    )
EXAMPLE: MCMC METHODS FOR TALL DATA

This example illustrates the usages of various MCMC methods which are suitable for tall data:

- algo="SA" uses the sample adaptive MCMC method in [1]
- algo="HMCECS" uses the energy conserving subsampling method in [2]
- algo="FlowHMCECS" utilizes a normalizing flow to neutralize the posterior geometry into a Gaussian-like one. Then HMCECS is used to draw the posterior samples. Currently, this method gives the best mixing rate among those methods.

References:

2. Hamiltonian Monte Carlo with energy conserving subsampling, Dang, K. D., Quiroz, M., Kohn, R., Minh-Ngoc, T., & Villani, M. (2019)

```python
import argparse
import time
import matplotlib.pyplot as plt
from jax import random
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import COVTYPE, load_dataset
from numpyro.infer import HMC, HMCECS, MCMC, NUTS, SA, SVI, init_to_value
from numpyro.infer.autoguide import AutoBNAFNormal
from numpyro.infer.reparam import NeuTraReparam

def _load_dataset():
    _, fetch = load_dataset(COVTYPE, shuffle=False)
    features, labels = fetch()

    # normalize features and add intercept
    features = (features - features.mean(0)) / features.std(0)
    features = jnp.hstack([features, jnp.ones((features.shape[0], 1))])

    # make binary feature
```

(continues on next page)
_, counts = jnp.unique(labels, return_counts=True)
specific_category = jnp.argmax(counts)
labels = labels == specific_category

N, dim = features.shape
print("Data shape:", features.shape)
print(
    "Label distribution: {} has label 1, {} has label 0".format(
        labels.sum(), N - labels.sum()
    )
)
return features, labels

def model(data, labels, subsample_size=None):
dim = data.shape[1]
coefs = numpyro.sample("coefs", dist.Normal(jnp.zeros(dim), jnp.ones(dim)))
with numpyro.plate("N", data.shape[0], subsample_size=subsample_size) as idx:
    logits = jnp.dot(data[idx], coefs)
return numpyro.sample("obs", dist.Bernoulli(logits=logits), obs=labels[idx])

def benchmark_hmc(args, features, labels):
    rng_key = random.PRNGKey(1)
    start = time.time()
    # a MAP estimate at the following source
    # https://github.com/google/edward2/blob/master/examples/no_u_turn_sampler/logistic_regression.py#L117
    ref_params = {
        "coefs": jnp.array(
            [+2.03420663e00,
             -3.53567265e-02,
             -1.49223924e-01,
             -3.07049364e-01,
             -1.00028366e-01,
             -1.46827862e-01,
             -1.64167881e-01,
             -4.20344204e-01,
             +9.47479829e-02,
             -1.12681836e-02,
             +2.64442056e-01,
             -1.22087866e-01,
             -6.00568838e-02,
             -3.79419506e-01,
             -1.06668741e-01,
             -2.97053963e-01,
             -2.05253899e-01,
             -4.69537191e-02,
             -2.78072730e-02,
             -1.43250525e-01,
             -6.77954629e-02,]...)
-4.34899796e-03,
+5.90927452e-02,
+7.2313609e-02,
+1.3852691e-02,
-1.24497896e-01,
-1.50733739e-02,
-2.6872194e-02,
-1.80925727e-02,
+3.47936489e-02,
+4.03552800e-02,
-9.8773426e-03,
+6.20188080e-02,
+1.1500271e-01,
+1.32145107e-01,
+2.69109547e-01,
+2.45785132e-01,
+1.19035013e-01,
-2.5974357e-02,
+9.42795154e-04,
+3.3926285e-02,
-1.44057125e-02,
-6.95222765e-02,
-7.5201308e-02,
+1.21171586e-01,
+2.29205526e-02,
+1.47308692e-01,
-8.34354162e-02,
-9.34122875e-02,
-2.97472421e-02,
-3.3937674e-01,
-1.70958012e-01,
-1.59496680e-01,
-1.88516974e-01,
-1.20889175e00,

})

if args.algo == "HMC":
    step_size = jnp.sqrt(0.5 / features.shape[0])
    trajectory_length = step_size * args.num_steps
    kernel = HMC(
        model,
        step_size=step_size,
        trajectory_length=trajectory_length,
        adapt_step_size=False,
        dense_mass=args.dense_mass,
    )
    subsample_size = None
elif args.algo == "NUTS":
    kernel = NUTS(model, dense_mass=args.dense_mass)
    subsample_size = None
elif args.algo == "HMCECS":
(continues on next page)
```
subsample_size = 1000
inner_kernel = NUTS(
    model,
    init_strategy=init_to_value(values=ref_params),
    dense_mass=args.dense_mass,
)
# note: if num_blocks=100, we'll update 10 index at each MCMC step
# so it took 50000 MCMC steps to iterative the whole dataset
kernel = HMCECS(
    inner_kernel, num_blocks=100, proxy=HMCECS.taylor_proxy(ref_params)
)

elif args.algo == "SA":
    # NB: this kernel requires large num_warmup and num_samples
    # and running on GPU is much faster than on CPU
    kernel = SA(
        model, adapt_state_size=1000, init_strategy=init_to_value(values=ref_params)
    )
subsample_size = None

elif args.algo == "FlowHMCECS":
    subsample_size = 1000
    guide = AutoBNAFNormal(model, num_flows=1, hidden_factors=[8])
    svi = SVI(model, guide, numpyro.optim.Adam(0.01), Trace_ELBO())
    svi_result = svi.run(random.PRNGKey(2), 2000, features, labels)
    params, losses = svi_result.params, svi_result.losses
    plt.plot(losses)
    plt.show()

    neutra = NeuTraReparam(guide, params)
    neutra_model = neutra.reparam(model)
    neutra_ref_params = {"auto_shared_latent": jnp.zeros(55)}
    # no need to adapt mass matrix if the flow does a good job
    inner_kernel = NUTS(
        neutra_model,
        init_strategy=init_to_value(values=neutra_ref_params),
        adapt_mass_matrix=False,
    )
    kernel = HMCECS(
        inner_kernel, num_blocks=100, proxy=HMCECS.taylor_proxy(neutra_ref_params)
    )
else:
    raise ValueError("Invalid algorithm, either 'HMC', 'NUTS', or 'HMCECS'.")

mcmc = MCMC(kernel, num_warmup=args.num_warmup, num_samples=args.num_samples)

mcmc.run(rng_key, features, labels, subsample_size, extra_fields=("accept_prob"),)
print("Mean accept prob:", jnp.mean(mcmc.get_extra_fields()["accept_prob"]))
mcmc.print_summary(exclude_deterministic=False)
print("\nMCMC elapsed time:", time.time() - start)
```

---

```
def main(args):
    features, labels = _load_dataset()
    benchmark_hmc(args, features, labels)
```

---
if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.15.2")
    parser = argparse.ArgumentParser(description="parse args")
    parser.add_argument(
        "-n", "--num-samples", default=1000, type=int, help="number of samples"
    )
    parser.add_argument(
        "--num-warmup", default=1000, type=int, help="number of warmup steps"
    )
    parser.add_argument(
        "--num-steps", default=10, type=int, help='number of steps (for "HMC")'
    )
    parser.add_argument("--num-chains", nargs="?", default=1, type=int)
    parser.add_argument(
        "--algo",
        default="HMCECS",
        type=str,
        help='whether to run "HMC", "NUTS", "HMCECS", "SA" or "FlowHMCECS"'
    )
    parser.add_argument("--dense-mass", action="store_true")
    parser.add_argument("--x64", action="store_true")
    parser.add_argument("--device", default="cpu", type=str, help='use "cpu" or "gpu".')
    args = parser.parse_args()
    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)
    if args.x64:
        numpyro.enable_x64()
    main(args)
EXAMPLE: HAMILTONIAN MONTE CARLO WITH ENERGY CONSERVING SUBSAMPLING

This example illustrates the use of data subsampling in HMC using Energy Conserving Subsampling. Data subsampling is applicable when the likelihood factorizes as a product of N terms.

References:

1. Hamiltonian Monte Carlo with energy conserving subsampling, Dang, K. D., Quiroz, M., Kohn, R., Minh-Ngoc, T., & Villani, M. (2019)

```
import argparse
import time
```

(continues on next page)
import matplotlib.pyplot as plt
import numpy as np
from jax import random
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
from numpyro.examples.datasets import HIGGS, load_dataset
from numpyro.infer import HMC, HMCECS, MCMC, NUTS, SVI, Trace_ELBO, autoguide

def model(data, obs, subsample_size):
    n, m = data.shape
    theta = numpyro.sample("theta", dist.Normal(jnp.zeros(m), 0.5 * jnp.ones(m)))
    with numpyro.plate("N", n, subsample_size=subsample_size):
        batch_feats = numpyro.subsample(data, event_dim=1)
        batch_obs = numpyro.subsample(obs, event_dim=0)
        numpyro.sample("obs", dist.Bernoulli(logits=theta @ batch_feats.T), obs=batch_obs)

def run_hmcecs(hmcecs_key, args, data, obs, inner_kernel):
    svi_key, mcmc_key = random.split(hmcecs_key)
    # find reference parameters for second order taylor expansion to estimate likelihood
    # (taylor_proxy)
    optimizer = numpyro.optim.Adam(step_size=1e-3)
    guide = autoguide.AutoDelta(model)
    svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
    svi_result = svi.run(svi_key, args.num_svi_steps, data, obs, args.subsample_size)
    params, losses = svi_result.params, svi_result.losses
    ref_params = {"theta": params["theta_auto_loc"]}

    # taylor proxy estimates log likelihood (ll) by
    # taylor_expansion(ll, theta_curr) +
    # sum_{i in subsample} ll_i(theta_curr) - taylor_expansion(ll_i, theta_curr)
    # around ref_params
    proxy = HMCECS.taylor_proxy(ref_params)
    kernel = HMCECS(inner_kernel, num_blocks=2, proxy=proxy)
    mcmc = MCMC(kernel, num_warmup=args.num_warmup, num_samples=args.num_samples)
    mcmc.run(mcmc_key, data, obs, args.subsample_size)
    mcmc.print_summary()
    return losses, mcmc.get_samples()

def run_hmc(mcmc_key, args, data, obs, kernel):
    mcmc = MCMC(kernel, num_warmup=args.num_warmup, num_samples=args.num_samples)
mcmc.run(mcmc_key, data, obs, None)
mcmc.print_summary()
return mcmc.get_samples()

def main(args):
    assert (11_000_000 >= args.num_datapoints), "11,000,000 data points in the Higgs dataset"
# full dataset takes hours for plain hmc!
if args.dataset == "higgs":
    _, fetch = load_dataset(HIGGS, shuffle=False, num_datapoints=args.num_datapoints)
    data, obs = fetch()
else:
    data, obs = (np.random.normal(size=(10, 28)), np.ones(10))

hmcecs_key, hmc_key = random.split(random.PRNGKey(args.rng_seed))

# choose inner_kernel
if args.inner_kernel == "hmc":
    inner_kernel = HMC(model)
else:
    inner_kernel = NUTS(model)

start = time.time()
losses, hmcecs_samples = run_hmcecs(hmcecs_key, args, data, obs, inner_kernel)
hmcecs_runtime = time.time() - start

start = time.time()
hmc_samples = run_hmc(hmc_key, args, data, obs, inner_kernel)
hmc_runtime = time.time() - start

summary_plot(losses, hmc_samples, hmcecs_samples, hmc_runtime, hmcecs_runtime)

def summary_plot(losses, hmc_samples, hmcecs_samples, hmc_runtime, hmcecs_runtime):
    fig, ax = plt.subplots(2, 2)
    ax[0, 0].plot(losses, "r")
    ax[0, 0].set_title("SVI losses")
    ax[0, 0].set_ylabel("ELBO")
    if hmc_runtime > hmcecs_runtime:
        ax[0, 1].bar([0], hmc_runtime, label="hmc", color="b")
        ax[0, 1].bar([0], hmcecs_runtime, label="hmcecs", color="r")
    else:
        ax[0, 1].bar([0], hmcecs_runtime, label="hmcecs", color="r")
        ax[0, 1].bar([0], hmc_runtime, label="hmc", color="b")
    ax[0, 1].set_title("Runtime")
    ax[0, 1].set_ylabel("Seconds")
    ax[0, 1].legend()
ax[0, 1].set_xticks([])

ax[1, 0].plot(jnp.sort(hmc_samples["theta"] .mean(0)), "or")
ax[1, 0].plot(jnp.sort(hmcecs_samples["theta"] .mean(0)), "b")
ax[1, 0].set_title(r"$\mathbb{E}[^{\theta}]$")

ax[1, 1].plot(jnp.sort(hmc_samples["theta"] .var(0)), "or")
ax[1, 1].plot(jnp.sort(hmcecs_samples["theta"] .var(0)), "b")
ax[1, 1].set_title(r"Var[^{\theta}]$")

for a in ax[1, :]:
a.set_xticks([])

fig.tight_layout()
fig.savefig("hmcecs_plot.pdf", bbox_inches="tight")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        "Hamiltonian Monte Carlo with Energy Conserving Subsampling"
    )
    parser.add_argument("--subsample_size", type=int, default=1300)
    parser.add_argument("--num_svi_steps", type=int, default=5000)
    parser.add_argument("--num_blocks", type=int, default=100)
    parser.add_argument("--num_warmup", type=int, default=500)
    parser.add_argument("--num_samples", type=int, default=500)
    parser.add_argument("--num_datapoints", type=int, default=1_500_000)
    parser.add_argument(
        "--dataset", type=str, choices=["higgs", "mock"], default="higgs"
    )
    parser.add_argument(
        "--inner_kernel", type=str, choices=["nuts", "hmc"], default="nuts"
    )
    parser.add_argument("--device", default="cpu", type=str, choices=["cpu", "gpu"])
    parser.add_argument(
        "--rng_seed", default=37, type=int, help="random number generator seed"
    )
    args = parser.parse_args()

    numpyro.set_platform(args.device)

    main(args)
EXAMPLE: BAYESIAN NEURAL NETWORK WITH STEINVI

We demonstrate how to use SteinVI to predict housing prices using a BNN for the Boston Housing prices dataset from the UCI regression benchmarks.

```python
import argpase
from collections import namedtuple
import datetime
from functools import partial
from time import time

from matplotlib.collections import LineCollection
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split

import jax
from jax import random
import jax.numpy as jnp

import numpyro
from numpyro import deterministic
from numpyro.contrib.einstein import IMQKernel, SteinVI
from numpyro.contrib.einstein.mixture_guide_predictive import MixtureGuidePredictive
from numpyro.distributions import Gamma, Normal
from numpyro.examples.datasets import BOSTON_HOUSING, load_dataset
from numpyro.infer import init_to_uniform
from numpyro.infer.autoguide import AutoNormal
from numpyro.optim import Adagrad

DataState = namedtuple("data", ["xtr", "xte", "ytr", "yte"])

def load_data() -> DataState:
    _, fetch = load_dataset(BOSTON_HOUSING, shuffle=False)
    x, y = fetch()
    xtr, xte, ytr, yte = train_test_split(x, y, train_size=0.90, random_state=1)

    return DataState(*map(jnp.array, (xtr, xte, ytr, yte)))

def normalize(val, mean=None, std=None):
    return (val - mean) / std
```

(continues on next page)
if mean is None and std is None:
    # Only use training data to estimate mean and std.
    std = jnp.std(val, 0, keepdims=True)
    std = jnp.where(std == 0, 1.0, std)
    mean = jnp.mean(val, 0, keepdims=True)
return (val - mean) / std, mean, std

def model(x, y=None, hidden_dim=50, subsample_size=100):
    """BNN described in section 5 of [1].
    
    **References:**
    1. "Stein variational gradient descent: A general purpose bayesian inference algorithm"  
       Qiang Liu and Dilin Wang (2016).
    ""

    prec_nn = numpyro.sample("prec_nn", Gamma(1.0, 0.1))
    )  # hyper prior for precision of nn weights and biases

    n, m = x.shape

    with numpyro.plate("l1_hidden", hidden_dim, dim=-1):
        # prior l1 bias term
        b1 = numpyro.sample("nn_b1", Normal(0.0, 1.0 / jnp.sqrt(prec_nn)),
            ),
        )
        assert b1.shape == (hidden_dim,)

    with numpyro.plate("l1_feat", m, dim=-2):
        w1 = numpyro.sample("nn_w1", Normal(0.0, 1.0 / jnp.sqrt(prec_nn))
            )  # prior on l1 weights
        assert w1.shape == (m, hidden_dim)

    with numpyro.plate("l2_hidden", hidden_dim, dim=-1):
        w2 = numpyro.sample("nn_w2", Normal(0.0, 1.0 / jnp.sqrt(prec_nn))
            )  # prior on output weights

    b2 = numpyro.sample("nn_b2", Normal(0.0, 1.0 / jnp.sqrt(prec_nn))
        )  # prior on output bias term

    # precision prior on observations
    prec_obs = numpyro.sample("prec_obs", Gamma(1.0, 0.1))
(continues on next page)
with numpyro.plate(
    "data",
    x.shape[0],
    subsample_size=subsample_size,
    dim=-1,
):  
    batch_x = numpyro.subsample(x, event_dim=1)
    if y is not None:
        batch_y = numpyro.subsample(y, event_dim=0)
    else:
        batch_y = y

    loc_y = deterministic("y_pred", jnp.maximum(batch_x @ w1 + b1, 0) @ w2 + b2)

    numpyro.sample(
        "y",
        Normal(
            loc_y, 1.0 / jnp.sqrt(prec_obs)
        ),
        # 1 hidden layer with ReLU activation
        obs=batch_y,
    )

def main(args):
    data = load_data()

    inf_key, pred_key, data_key = random.split(random.PRNGKey(args.rng_key), 3)
    # normalize data and labels to zero mean unit variance!
    x, xtr_mean, xtr_std = normalize(data.xtr)
    y, ytr_mean, ytr_std = normalize(data.ytr)

    rng_key, inf_key = random.split(inf_key)

    guide = AutoNormal(model, init_loc_fn=partial(init_to_uniform, radius=0.1))

    stein = SteinVI(
        model,
        guide,
        Adagrad(0.05),
        IMQKernel(),
        # ProbabilityProductKernel(guide=guide, scale=1.),
        repulsion_temperature=args.repulsion,
        num_stein_particles=args.num_stein_particles,
        num_elbo_particles=args.num_elbo_particles,
    )
    start = time()

    # use keyword params for static (shape etc.)!
    result = stein.run(
        rng_key,
        args.max_iter,
        x,
    )
y,
    hidden_dim=args.hidden_dim,
    subsample_size=args.subsample_size,
    progress_bar=args.progress_bar,
)
time_taken = time() - start

pred = MixtureGuidePredictive(
    model,
    guide=stein.guide,
    params=stein.get_params(result.state),
    num_samples=100,
    guide_sites=stein.guide_sites,
)

t, _, _ = normalize(
    data.xte, xtr_mean, xtr_std
)
# use train data statistics when accessing generalization
preds = pred(
    pred_key, t, subsample_size=t.shape[0], hidden_dim=args.hidden_dim
)["y_pred"]

y_pred = preds * ytr_std + ytr_mean
rmse = jnp.sqrt(jnp.mean((y_pred.mean(0) - data.yte) ** 2))

print(rf"Time taken: {datetime.timedelta(seconds=int(time_taken))}")
print(rf"RMSE: {rmse:.2f}")

# compute mean prediction and confidence interval around median
mean_prediction = y_pred.mean(0)

ran = np.arange(mean_prediction.shape[0])
percentiles = np.percentile(preds * ytr_std + ytr_mean, [5.0, 95.0], axis=0)

# make plots
fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)
ax.add_collection(LineCollection(
    zip(zip(ran, percentiles[0]), zip(ran, percentiles[1])), colors="lightblue"
)
)
ax.plot(data.yte, "kx", label="y true")
ax.plot(mean_prediction, "ko", label="y pred")
ax.set_xlabel("example")
ax.set_ylabel("y")
ax.set_title("Mean predictions with 90% CI")
ax.legend()
fig.savefig("stein_bnn.pdf")

if __name__ == "__main__":
    jax.config.update("jax_debug_nans", True)

    parser = argparse.ArgumentParser()
    parser.add_argument("--subsample-size", type=int, default=100)
parser.add_argument("--max-iter", type=int, default=1000)
parser.add_argument("--repulsion", type=float, default=1.0)
parser.add_argument("--verbose", type=bool, default=True)
parser.add_argument("--num-elbo-particles", type=int, default=50)
parser.add_argument("--num-stein-particles", type=int, default=5)
parser.add_argument("--progress-bar", type=bool, default=True)
parser.add_argument("--rng-key", type=int, default=142)
parser.add_argument("--device", default="cpu", choices=['gpu', 'cpu'])
parser.add_argument("--hidden-dim", default=50, type=int)

args = parser.parse_args()
numpyro.set_platform(args.device)

main(args)
EXAMPLE: DEEP MARKOV MODEL INFERRED USING STEINVI

In this example we infer a deep Markov model (DMM) using SteinVI for generating music (chorales by Johan Sebastian Bach).

The model DMM based on reference [1][2] and the Pyro DMM example: https://pyro.ai/examples/dmm.html.

Reference:


2.Structured Inference Networks for Nonlinear State Space Models [arXiv:1609.09869]
   Rahul G. Krishnan, Uri Shalit and David Sontag (2016)

```python
import argparse
import numpy as np
```

(continues on next page)
import jax
from jax import nn, numpy as jnp, random
from optax import adam, chain

import numpyro
from numpyro.contrib.einstein import SteinVI
from numpyro.contrib.einstein.mixture_guide_predictive import MixtureGuidePredictive
from numpyro.contrib.einstein.stein_kernels import RBFKernel
import numpyro.distributions as dist
from numpyro.examples.datasets import JSB_CHORALES, load_dataset
from numpyro.optim import optax_to_numpyro

def _reverse_padded(padded, lengths):
    def _reverse_single(p, length):
        new = jnp.zeros_like(p)
        reverse = jnp.roll(p[::-1], length, axis=0)
        return new.at[0].set(reverse)

    return jax.vmap(_reverse_single)(padded, lengths)

def load_data(split="train"):
    _, fetch = load_dataset(JSB_CHORALES, split=split)
    lengths, seqs = fetch(0)
    return (seqs, _reverse_padded(seqs, lengths), lengths)

def emitter(x, params):
    """Parameterizes the bernoulli observation likelihood \(p(x_t | z_t)\)""

    l1 = nn.relu(jnp.matmul(x, params["l1"])
    l2 = nn.relu(jnp.matmul(l1, params["l2"])
    return jnp.matmul(l2, params["l3"])

def transition(x, params):
    """Parameterizes the gaussian latent transition probability \(p(z_t | z_{t-1})\)"
    See section 5 in [1].

    **Reference:**
      Rahul G. Krishnan, Uri Shalit and David Sontag (2016)

    def _gate(x, params):
        l1 = nn.relu(jnp.matmul(x, params["l1"])
        return nn.sigmoid(jnp.matmul(l1, params["l2"])

    def _shared(x, params):
        l1 = nn.relu(jnp.matmul(x, params["l1"])
        return jnp.matmul(l1, params["l2"])

(continued from previous page)
def _mean(x, params):
    return jnp.matmul(x, params["l1"])  

def _std(x, params):
    l1 = jnp.matmul(nn.relu(x), params["l1"])
    return nn.softplus(l1)  

gt = _gate(x, params["gate"])
ht = _shared(x, params["shared"])
loc = (1 - gt) * _mean(x, params["mean"]) + gt * ht
std = _std(ht, params["std"])  
return loc, std  

def combiner(x, params):
    mean = jnp.matmul(x, params["mean"])  
    std = nn.softplus(jnp.matmul(x, params["std"]))  
    return mean, std  

def gru(xs, lengths, init_hidden, params):
    """RNN with GRU. Based on https://github.com/google/jax/pull/2298""
    def apply_fun_single(state, inputs):
        i, x = inputs
        inp_update = jnp.matmul(x, params["update_in"])  
        hidden_update = jnp.dot(state, params["update_weight"])  
        update_gate = nn.sigmoid(inp_update + hidden_update)
        reset_gate = nn.sigmoid(
            jnp.matmul(x, params["reset_in"]) + jnp.dot(state, params["reset_weight"])  
        )  
        output_gate = update_gate * state + (1 - update_gate) * jnp.tanh(
            jnp.matmul(x, params["out_in"])  
            + jnp.dot(reset_gate * state, params["out_weight"])  
        )  
        hidden = jnp.where((i < lengths)[::, None], output_gate, jnp.zeros_like(state))  
        return hidden, hidden  

    init_hidden = jnp.broadcast_to(init_hidden, (xs.shape[1], init_hidden.shape[1]))
    return jax.lax.scan(apply_fun_single, init_hidden, (jnp.arange(xs.shape[0]), xs))  

def _normal_init("shape"):  
    return lambda rng_key: dist.Normal(scale=0.1).sample(rng_key, shape)  

def model(
    seqs,
    seqs_rev,
    lengths,
    *,
    (continues on next page)
subsample_size=77,
latent_dim=32,
emission_dim=100,
transition_dim=200,
data_dim=88,
gru_dim=150,
annealing_factor=1.0,
predict=False,
):
    max_seq_length = seqs.shape[1]

    emitter_params = {
        "l1": numpyro.param("emitter_l1", _normal_init(latent_dim, emission_dim)),
        "l2": numpyro.param("emitter_l2", _normal_init(emission_dim, emission_dim)),
        "l3": numpyro.param("emitter_l3", _normal_init(emission_dim, data_dim)),
    }

    trans_params = {
        "gate": {
            "l1": numpyro.param("gate_l1", _normal_init(latent_dim, transition_dim)),
            "l2": numpyro.param("gate_l2", _normal_init(transition_dim, latent_dim)),
        },
        "shared": {
            "l1": numpyro.param("shared_l1", _normal_init(latent_dim, transition_dim)),
            "l2": numpyro.param("shared_l2", _normal_init(transition_dim, latent_dim)),
        },
        "mean": {"l1": numpyro.param("mean_l1", _normal_init(latent_dim, latent_dim))},
        "std": {"l1": numpyro.param("std_l1", _normal_init(latent_dim, latent_dim))},
    }

    z0 = numpyro.param("z0",
        lambda rng_key: dist.Normal(0, 1.0).sample(rng_key, (latent_dim,))
    )

    z0 = jnp.broadcast_to(z0, (subsample_size, 1, latent_dim))

    with numpyro.plate("data", seqs.shape[0], subsample_size=subsample_size, dim=-1)
        as idx:
            if subsample_size == seqs.shape[0]:
                seqs_batch = seqs
                lengths_batch = lengths
            else:
                seqs_batch = seqs[idx]
                lengths_batch = lengths[idx]

    masks = jnp.repeat(
        jnp.expand_dims(jnp.arange(max_seq_length), axis=0), subsample_size, axis=0
    ) < jnp.expand_dims(lengths_batch, axis=-1)
    # NB: Mask is to avoid scoring 'z' using distribution at this point
    z = numpyro.sample("z",
        dist.Normal(0.0, jnp.ones((max_seq_length, latent_dim)))
            .mask(False)
    )

    (continues on next page)
.to_event(2),
)

z_shift = jnp.concatenate([[z0, z[:, :, -1, :]], axis=-2]
z_loc, z_scale = transition(z_shift, params=trans_params)

with numpyro.handlers.scale(scale=annealing_factor):
    # Actually score 'z'
    numpyro.sample(
        "z_aux",
        dist.Normal(z_loc, z_scale)
        .mask(jnp.expand_dims(masks, axis=-1))
        .to_event(2),
        obs=z,
    )

emission_probs = emitter(z, params=emitter_params)
if predict:
    tunes = None
else:
    tunes = seqs_batch
numpyro.sample(
    "tunes",
    dist.Bernoulli(logits=emission_probs)
    .mask(jnp.expand_dims(masks, axis=-1))
    .to_event(2),
    obs=tunes,
)

def guide(
    seqs,
    seqs_rev,
    lengths,
    *,
    subsample_size=77,
    latent_dim=32,
    emission_dim=100,
    transition_dim=200,
    data_dim=88,
    gru_dim=150,
    annealing_factor=1.0,
    predict=False,
):
    max_seq_length = seqs.shape[1]
    seqs_rev = jnp.transpose(seqs_rev, axes=(1, 0, 2))

    combiner_params = {
        "mean": numpyro.param("combiner_mean", _normal_init(gru_dim, latent_dim)),
        "std": numpyro.param("combiner_std", _normal_init(gru_dim, latent_dim)),
    }
gru_params = {
    "update_in": numpyro.param("update_in", _normal_init(data_dim, gru_dim)),
    "update_weight": numpyro.param("update_weight", _normal_init(gru_dim, gru_dim)),
    "reset_in": numpyro.param("reset_in", _normal_init(data_dim, gru_dim)),
    "reset_weight": numpyro.param("reset_weight", _normal_init(gru_dim, gru_dim)),
    "out_in": numpyro.param("out_in", _normal_init(data_dim, gru_dim)),
    "out_weight": numpyro.param("out_weight", _normal_init(gru_dim, gru_dim)),
}

with numpyro.plate(
    "data", seqs.shape[0], subsample_size=subsample_size, dim=-1
) as idx:
    if subsample_size == seqs.shape[0]:
        seqs_rev_batch = seqs_rev
        lengths_batch = lengths
    else:
        seqs_rev_batch = seqs_rev[:, idx, :]
        lengths_batch = lengths[idx]

masks = jnp.repeat(
    jnp.expand_dims(jnp.arange(max_seq_length), axis=0), subsample_size, axis=0
) < jnp.expand_dims(lengths_batch, axis=-1)

h0 = numpyro.param(
    "h0",
    lambda rng_key: dist.Normal(0.0, 1).sample(rng_key, (1, gru_dim)),
)

_, hs = gru(seqs_rev_batch, lengths_batch, h0, gru_params)
hs = _reverse_padded(jnp.transpose(hs, axes=(1, 0, 2)), lengths_batch)

with numpyro.handlers.scale(scale=annealing_factor):
    numpyro.sample(
        "z",
        dist.Normal(*combiner(hs, combiner_params)).mask(jnp.expand_dims(masks, axis=-1)).to_event(2),
    )

def vis_tune(i, tunes, lengths, name="stein_dmm.pdf"):
    tune = tunes[i, : lengths[i]]
    try:
        from music21.chord import Chord
        from music21.pitch import Pitch
        from music21.stream import Stream

        stream = Stream()
        for chord in tune:
            stream.append(Chord(list(Pitch(pitch) for pitch in (np.arange(88) + 21)[chord > 0])))
        plot = stream.plot(doneAction=None)
        plot.write(name)
except ModuleNotFoundError:
    import matplotlib.pyplot as plt

    plt.imshow(tune.T, cmap="Greys")
    plt.ylabel("Pitch")
    plt.xlabel("Offset")
    plt.savefig(name)

def main(args):
    inf_key, pred_key = random.split(random.PRNGKey(seed=args.rng_seed), 2)

    steinvi = SteinVI(
        model,
        guide,
        optax_to_numpyro(chain(adam(1e-2))),
        RBFKernel(),
        num_elbo_particles=args.num_elbo_particles,
        num_stein_particles=args.num_stein_particles,
    )

    seqs, rev_seqs, lengths = load_data()
    results = steinvi.run(
        inf_key,
        args.max_iter,
        seqs,
        rev_seqs,
        lengths,
        gru_dim=args.gru_dim,
        subsample_size=args.subsample_size,
    )

    pred = MixtureGuidePredictive(
        model,
        guide,
        params=results.params,
        num_samples=1,
        guide_sites=steinvi.guide_sites,
    )

    seqs, rev_seqs, lengths = load_data("valid")
    pred_notes = pred(
        pred_key, seqs, rev_seqs, lengths, subsample_size=seqs.shape[0], predict=True
    )["tunes"]

    vis_tune(0, pred_notes[0], lengths)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--subsample-size", type=int, default=10)
    parser.add_argument("--max-iter", type=int, default=100)
    parser.add_argument("--repulsion", type=float, default=1.0)
    parser.add_argument("--verbose", type=bool, default=True)
parser.add_argument("--num-stein-particles", type=int, default=5)
parser.add_argument("--num-elbo-particles", type=int, default=5)
parser.add_argument("--progress-bar", type=bool, default=True)
parser.add_argument("--gru-dim", type=int, default=150)
parser.add_argument("--rng-key", type=int, default=142)
parser.add_argument("--device", default="cpu", choices=["gpu", "cpu"])
parser.add_argument("--rng-seed", default=142, type=int)

args = parser.parse_args()

numpyro.set_platform(args.device)

main(args)
INDICES AND TABLES

- genindex
- modindex
- search
n
numpyro.contrib.funsor, 201
numpyro.contrib.tfp.distributions, 116
numpyro.contrib.tfp.mcmc, 165
numpyro.diagnostics, 216
numpyro.handlers, 233
numpyro.infer.autoguide, 180
numpyro.infer.reparam, 197
numpyro.infer.util, 220
numpyro.ops.indexing, 226
numpyro.optim, 205
numpyro.primitives, 13
numpyro.util, 219
Symbols

__call__ () (CircularReparam method), 200
__call__ () (ExplicitReparam method), 200
__call__ () (LocScaleReparam method), 197
__call__ () (NeuTraReparam method), 198
__call__ () (ProjectedNormalReparam method), 199
__call__ () (TransformReparam method), 199
arg_constraints

A
AbsTransform (class in numpyro.distributions.transforms), 121
Adagrad (class in numpyro.optim), 206
Adam (class in numpyro.optim), 205
AffineTransform (class in numpyro.distributions.transforms), 121
AIES (class in numpyro.infer.ensemble), 156
AIESState (in module numpyro.infer.ensemble), 164
append() (AutoGuideList method), 181
arg_constraints (AsymmetricLaplace attribute), 41
arg_constraints (AsymmetricLaplaceQuantile attribute), 42
arg_constraints (BernoulliLogits attribute), 83
arg_constraints (BernoulliProbs attribute), 84
arg_constraints (Beta attribute), 43
arg_constraints (BetaBinomial attribute), 85
arg_constraints (BetaProportion attribute), 44
arg_constraints (BinomialLogits attribute), 86
arg_constraints (BinomialProbs attribute), 87
arg_constraints (CAR attribute), 45
arg_constraints (CategoricalLogits attribute), 88
arg_constraints (CategoricalProbs attribute), 89
arg_constraints (Cauchy attribute), 46
arg_constraints (Chi2 attribute), 47
arg_constraints (Delta attribute), 39
arg_constraints (Dirichlet attribute), 48
arg_constraints (DirichletMultinomial attribute), 90
arg_constraints (DiscreteUniform attribute), 92
arg_constraints (Distribution attribute), 27
arg_constraints (EulerMaruyama attribute), 49
arg_constraints (ExpandedDistribution attribute), 32
arg_constraints (Exponential attribute), 50
arg_constraints (Gamma attribute), 51
arg_constraints (GammaPoisson attribute), 93
arg_constraints (GaussianCopula attribute), 52
arg_constraints (GaussianCopulaBeta attribute), 53
arg_constraints (GaussianRandomWalk attribute), 54
arg_constraints (GeometricLogits attribute), 94
arg_constraints (GeometricProbs attribute), 95
arg_constraints (Gompertz attribute), 55
arg_constraints (Gumbel attribute), 56
arg_constraints (HalfCauchy attribute), 57
arg_constraints (HalfNormal attribute), 58
arg_constraints (ImproperCauchy attribute), 34
arg_constraints (Independent attribute), 35
arg_constraints (InverseGamma attribute), 59
arg_constraints (Kumaraswamy attribute), 60
arg_constraints (Laplace attribute), 61
arg_constraints (LeftTruncatedDistribution attribute), 111
arg_constraints (LKJ attribute), 63
arg_constraints (LKJCholesky attribute), 64
arg_constraints (Logistic attribute), 66
arg_constraints (LogNormal attribute), 65
arg_constraints (LogUniform attribute), 65
arg_constraints (LowRankMultivariateNormal attribute), 67
arg_constraints (MaskedDistribution attribute), 37
arg_constraints (MatrixNormal attribute), 68
arg_constraints (MultinomialLogits attribute), 96
arg_constraints (MultinomialProbs attribute), 97
arg_constraints (MultivariateNormal attribute), 69
arg_constraints (MultivariateStudentT attribute), 71
arg_constraints (NegativeBinomial2 attribute), 100
arg_constraints (NegativeBinomialLogits attribute), 100
arg_constraints (NegativeBinomialProbs attribute), 100
arg_constraints (Normal attribute), 72
arg_constraints (OrderedLogistic attribute), 99
arg_constraints (Pareto attribute), 73
arg_constraints (Poisson attribute), 101
arg_constraints (ProjectedNormal attribute), 106
arg_constraints (RelaxedBernoulliLogits attribute), 74

INDEX
arg_constraints (RightTruncatedDistribution attribute), 112
arg_constraints (SineBivariateVonMises attribute), 108
arg_constraints (SineSkewed attribute), 110
arg_constraints (SoftLaplace attribute), 74
arg_constraints (StudentT attribute), 75
arg_constraints (TransformedDistribution attribute), 38
arg_constraints (TruncatedPolyaGamma attribute), 114
arg_constraints (TwoSidedTruncatedDistribution attribute), 115
arg_constraints (Uniform attribute), 77
arg_constraints (Unit attribute), 40
arg_constraints (VonMises attribute), 110
arg_constraints (Weibull attribute), 78
arg_constraints (Wishart attribute), 79
arg_constraints (WishartCholesky attribute), 80
arg_constraints (ZeroInflatedPoisson attribute), 102
arg_constraints (ZeroSumNormal attribute), 83
AsymmetricLaplace (class in numpyro.distributions.continuous), 41
AsymmetricLaplaceQuantile (class in numpyro.distributions.continuous), 42
AutoBNAFNormal (class in numpyro.infer.autoguide), 184
AutoContinuous (class in numpyro.infer.autoguide), 182
autocorrelation() (in module numpyro.diagnostics), 216
autocovariance() (in module numpyro.diagnostics), 217
AutoDAIS (class in numpyro.infer.autoguide), 193
AutoDelta (class in numpyro.infer.autoguide), 191
AutoDiagonalNormal (class in numpyro.infer.autoguide), 184
AutoGuide (class in numpyro.infer.autoguide), 180
AutoGuideList (class in numpyro.infer.autoguide), 181
AutoIAFNormal (class in numpyro.infer.autoguide), 187
AutoLaplaceApproximation (class in numpyro.infer.autoguide), 187
AutoLowRankMultivariateNormal (class in numpyro.infer.autoguide), 189
AutoMultivariateNormal (class in numpyro.infer.autoguide), 185
AutoNormal (class in numpyro.infer.autoguide), 190
AutoSemiDAIS (class in numpyro.infer.autoguide), 194
AutoSurrogateLikelihoodDAIS (class in numpyro.infer.autoguide), 195

BarkerMH (class in numpyro.infer.barker), 140
BarkerMHState (in module numpyro.infer.barker), 163

batch_shape (Distribution property), 28
Bernoulli() (in module numpyro.distributions.discrete), 83
BernoulliLogits (class in numpyro.distributions.discrete), 83
BernoulliProbs (class in numpyro.distributions.discrete), 84
Beta (class in numpyro.distributions.continuous), 43
BetaBinomial (class in numpyro.distributions.conjugate), 85
BetaProportion (class in numpyro.distributions.continuous), 44
biject_to() (in module numpyro.distributions.transforms), 121
BijectorConstraint (class in numpyro.infer.tfp.distributions), 116
BijectorTransform (class in numpyro.infer.tfp.distributions), 116
Binomial() (class in numpyro.distributions.discrete), 86
BinomialLogits (class in numpyro.distributions.discrete), 86
BinomialProbs (class in numpyro.distributions.discrete), 87
block (class in numpyro.handlers), 234
BlockNeuralAutoregressiveTransform (class in numpyro.distributions.flows), 131
boolean (module numpyro.distributions.constraints), 117

call_with_intermediates() (BlockNeuralAutoregressiveTransform method), 131
call_with_intermediates() (ComposeTransform method), 122
call_with_intermediates() (InverseAutoregressiveTransform method), 131
call_with_intermediates() (Transform method), 121
can_infer_discrete (ELBO attribute), 174
can_infer_discrete (TraceEnum_ELBO attribute), 176

can_infer_discrete (TraceGraph_ELBO attribute), 177
CAR (class in numpyro.distributions.continuous), 45
Categorical() (in module numpyro.distributions.discrete), 88
CategoricalLogits (class in numpyro.distributions.discrete), 88
CategoricalProbs (class in numpyro.distributions.discrete), 89
Cauchy (class in numpyro.distributions.continuous), 46
cdf() (AsymmetricLaplaceQuantile method), 43
D
DCC (class in numpyro.contrib.stochastic_support.dcc), 249
default_fields (HMC property), 144
default_fields (MCMCKernel property), 140
default_fields (SA property), 154
Delta (class in numpyro.distributions.distribution), 39
DEMove() (AIES static method), 157
dependent (in module
numpyro.distributions.constrains), 117
deterministic() (in module numpyro.primitives), 16
diag_spectral_density_matern() (in module
numpyro.contrib.hsgp.spectral_densities), 258
diag_spectral_density_periodic() (in module
numpyro.contrib.hsgp.spectral_densities), 259
diag_spectral_density_squared_exponential() (in module
numpyro.contrib.hsgp.spectral_densities), 258
diagnostics() (NestedSampler method), 244
DifferentialMove() (ESS static method), 48
Dirichlet (class in numpyro.distributions.continuous), 48
DirichletMultinomial (class in
numpyro.distributions.conjugate), 90
DiscreteHMC Gibbs (class in numpyro.infer.hmc_gibbs), 148
DiscreteUniform (class in
numpyro.distributions.discrete), 92
Distribution (class in
numpyro.distributions.distribution), 27
do (class in numpyro.handlers), 236
domain (BlockNeuralAutoregressiveTransform attribute), 121
domain (CholeskyTransform attribute), 122
domain (ComposeTransform property), 122
domain (CorrCholeskyTransform attribute), 123
domain (CorrMatrixCholeskyTransform attribute), 123
domain (InverseAutoregressiveTransform attribute), 131
domain (L1BallTransform attribute), 124
domain (LowerCholeskyAffine attribute), 124
domain (LowerCholeskyTransform attribute), 125
domain (OrderedTransform attribute), 125
domain (PermuteTransform attribute), 125
domain (PowerTransform attribute), 126
domain (RealFastFourierTransform property), 126
domain (RecursiveLinearTransform attribute), 127
domain (ScaledUnitLowerCholeskyTransform attribute), 128
domain (SimplexToOrderedTransform attribute), 128
domain (SoftplusLowerCholeskyTransform attribute), 129
domain (SoftplusTransform attribute), 129
domain (StickBreakingTransform attribute), 130
domain (Transform attribute), 121
domain (ZeroSumTransform property), 130

E
effective_sample_size() (in module
numpyro.diagnostics), 217
eigenfunctions() (in module
numpyro.contrib.hsgp.laplacian), 255
eigenfunctions_periodic() (in module
numpyro.contrib.hsgp.laplacian), 256
eigenindices() (in module
numpyro.contrib.hsgp.laplacian), 254
ELBO (class in numpyro.infer.elbo), 174
enable_validation() (in module
numpyro.distributions.distribution), 219
enable_x64() (in module numpyro.util), 219
EnsembleSampler (class in numpyro.infer.ensemble), 155
EnsembleSamplerState (in module
numpyro.infer.ensemble), 164
entropy() (BernoulliLogits method), 84
entropy() (BernoulliProbs method), 85
entropy() (Beta method), 44
entropy() (CategoricalLogits method), 89
entropy() (CategoricalProbs method), 90
entropy() (Cauchy method), 47
entropy() (Delta method), 40
entropy() (Dirichlet method), 49
entropy() (DiscreteUniform method), 93
entropy() (Distribution method), 29
entropy() (ExpandedDistribution method), 33
entropy() (Exponential method), 51
entropy() (Gamma method), 52
entropy() (GeometricLogits method), 95
entropy() (GeometricProbs method), 95
entropy() (Independent method), 36
entropy() (InverseGamma method), 59
entropy() (Laplace method), 62
entropy() (Logistic method), 67
entropy() (LogNormal method), 65
entropy() (LogUniform method), 65
entropy() (LowRankMultivariateNormal method), 68
entropy() (MultivariateNormal method), 70
entropy() (Normal method), 73
entropy() (OrderedLogistic method), 99
entropy() (Pareto method), 73
entropy() (StudentT method), 76
entropy() (Uniform method), 78
entropy() (Weibull method), 79
entropy() (Wishart method), 80
enum (class in numpyro.contrib.funsor.enum_messenger), 201
enumerate_support() (BernoulliLogits method), 84
enumerate_support() (BernoulliProbs method), 85
Index 711

init() (Adam method), 205
init() (BarkerMH method), 141
init() (ClippedAdam method), 208
init() (DiscreteHMCGibbs method), 149
init() (EnsembleSampler method), 155
init() (HMC method), 144
init() (HMCECS method), 152
init() (HMCGibbs method), 147
init() (MCMCKernel method), 139
init() (Minimize method), 210
init() (MixedHMC method), 151
init() (Momentum method), 211
init() (RMSProp method), 212
init() (RMSPropMomentum method), 213
init() (SA method), 154
init() (SGD method), 214
init() (SM3 method), 215
init() (SVI method), 171
init inner state() (AIES method), 157
init inner state() (EnsembleSampler method), 155
init inner state() (ESS method), 159
init kernel() (in module numpyro.infer.hmc.hmc), 161
init to feasible() (in module numpyro.infer.initialization), 225
init to mean() (in module numpyro.infer.initialization), 225
init to median() (in module numpyro.infer.initialization), 225
init to sample() (in module numpyro.infer.initialization), 225
init to uniform() (in module numpyro.infer.initialization), 226
init to value() (in module numpyro.infer.initialization), 226
initialize model() (in module numpyro.infer.util), 167
integer greater than() (in module numpyro.distributions.constraints), 118
integer interval() (in module numpyro.distributions.constraints), 118
interval() (in module numpyro.distributions.constraints), 118
inv (Transform property), 121
inverse shape() (AffineTransform method), 122
inverse shape() (ComposeTransform method), 122
inverse shape() (CorrCholeskyTransform method), 123
inverse shape() (LowerCholeskyAffine method), 124
inverse shape() (LowerCholeskyTransform method), 125
inverse shape() (PowerTransform method), 126
inverse shape() (RealFastFourierTransform method), 126
inverse_shape() (SimplexToOrderedTransform method), 129
inverse_shape() (SoftplusLowerCholeskyTransform method), 129
inverse_shape() (StickBreakingTransform method), 130
inverse_shape() (Transform method), 121
inverse_shape() (ZeroSumTransform method), 130
InverseAutoregressiveTransform (class in numpyro.distributions.flows), 131
InverseGamma (class in numpyro.distributions.continuous), 131
is_discrete (Constraint attribute), 116
is_discrete (Distribution property), 31
is_discrete (MixtureGeneral property), 105
is_discrete (MixtureSameFamily property), 103
is_ensemble_kernel (EnsembleSampler property), 136
is_ensemble_kernel (MCMCKernel property), 140
K
KDEMove() (ESS static method), 159
KL_KUMARASWAMY_BETA_TAYLOR_ORDER (Kumaraswamy attribute), 60
Kumaraswamy (class in numpyro.distributions.continuous), 60
L
l1_ball() (in module numpyro.distributions.constraints), 118
L1BallTransform (class in numpyro.distributions.transforms), 124
Laplace (class in numpyro.distributions.continuous), 61
last_state (MCMC property), 136
left_scale() (AsymmetricLaplace method), 41
LeftTruncatedDistribution (class in numpyro.distributions.truncated), 111
less_than() (in module numpyro.distributions.constraints), 118
lift (class in numpyro.handlers), 237
LinearKernel (class in numpyro.contrib.einstein.stein_kernels), 247
LKJ (class in numpyro.distributions.continuous), 62
LKJCholesky (class in numpyro.distributions.continuous), 63
LocScaleReparam (class in numpyro.infer.reparam), 197
log_abs_det_jacobian() (AffineTransform method), 121
log_abs_det_jacobian() (BlockNeuralAutoregressiveTransform method), 131
log_abs_det_jacobian() (CholeskyTransform method), 122
log_abs_det_jacobian() (ComposeTransform method), 122
log_abs_det_jacobian() (CorrCholeskyTransform method), 123
log_abs_det_jacobian() (CorrMatrixCholeskyTransform method), 123
log_abs_det_jacobian() (ExpTransform method), 123
log_abs_det_jacobian() (IdentityTransform method), 124
log_abs_det_jacobian() (InverseAutoregressiveTransform method), 131
log_abs_det_jacobian() (L1BallTransform method), 124
log_abs_det_jacobian() (LowerCholeskyAffine method), 124
log_abs_det_jacobian() (LowerCholeskyTransform method), 125
log_abs_det_jacobian() (OrderedTransform method), 125
log_abs_det_jacobian() (PermuteTransform method), 125
log_abs_det_jacobian() (PowerTransform method), 126
log_abs_det_jacobian() (RealFastFourierTransform method), 126
log_abs_det_jacobian() (RecursiveLinearTransform method), 127
log_abs_det_jacobian() (ScaledUnitLowerCholeskyTransform method), 128
log_abs_det_jacobian() (SigmoidTransform method), 128
log_abs_det_jacobian() (SimplexToOrderedTransform method), 129
log_abs_det_jacobian() (SoftplusCholeskyTransform method), 129
log_abs_det_jacobian() (SoftplusTransform method), 129
log_abs_det_jacobian() (StickBreakingTransform method), 130
log_abs_det_jacobian() (Transform method), 121
log_abs_det_jacobian() (ZeroSumTransform method), 130
log_cdf() (Normal method), 73
log_density() (in module numpyro.contrib.funsor.infer_util), 204
log_density() (in module numpyro.infer.util), 222
log_likelihood() (in module numpyro.infer.util), 224
log_prob() (AsymmetricLaplace method), 41
log_prob() (AsymmetricLaplaceQuantile method), 42
log_prob() (BernoulliLogits method), 84
log_prob() (BernoulliProbs method), 85
log_prob() (Beta method), 43
log_prob() (BetaBinomial method), 86
mean (CategoricalLogits property), 89
mean (CategoricalProbs property), 90
mean (Cauchy property), 47
mean (Delta property), 40
mean (Dirichlet property), 48
mean (DirichletMultinomial property), 91
mean (DiscreteUniform property), 92
mean (Distribution property), 29
mean (ExpandedDistribution property), 33
mean (Exponential property), 50
mean (Gamma property), 52
mean (GammaPoisson property), 93
mean (GaussianCopula property), 53
mean (GaussianRandomWalk property), 54
mean (GeometricLogits property), 94
mean (GeometricProbs property), 95
mean (Gompertz property), 55
mean (Gumbel property), 56
mean (HalfCauchy property), 58
mean (HalfNormal property), 59
mean (Independent property), 35
mean (InverseGamma property), 59
mean (Kumaraswamy property), 60
mean (Laplace property), 61
mean (LeftTruncatedDistribution property), 112
mean (LKJ property), 63
mean (Logistic property), 66
mean (LogNormal property), 65
mean (LogUniform property), 65
mean (LowRankMultivariateNormal property), 67
mean (MaskedDistribution property), 37
mean (MatrixNormal property), 69
mean (MultinomialLogits property), 96
mean (MultinomialProbs property), 98
mean (MultivariateNormal property), 70
mean (MultivariateStudentT property), 71
mean (Normal property), 73
mean (Pareto property), 73
mean (Poisson property), 101
mean (ProjectedNormal property), 106
mean (RightTruncatedDistribution property), 113
mean (SineBivariateVonMises property), 108
mean (SineSkewed property), 110
mean (SoftLaplace property), 75
mean (StudentT property), 76
mean (TransformedDistribution property), 39
mean (TwoSidedTruncatedDistribution property), 115
mean (Uniform property), 77
mean (VonMises property), 111
mean (Weibull property), 79
mean (ZeroSumNormal property), 83
mean () (Wishart method), 80
mean () (WishartCholesky method), 81
median () (AutoDelta method), 192
median () (AutoDiagonalNormal method), 185
median () (AutoGuide method), 180
median () (AutoGuideList method), 182
median () (AutoLaplaceApproximation method), 188
median () (AutoLowRankMultivariateNormal method), 189
median () (AutoMultivariateNormal method), 186
median () (AutoNormal method), 191
MetropolisAdjustedLangevinAlgorithm (class in numpyro.contrib.tfp.mcmc), 166
Minimize (class in numpyro.optim), 208
MixedHMC (class in numpyro.infer.mixed_hmc), 150
Mixture () (in module numpyro.distributions.mixtures), 102
MixtureGeneral (class in numpyro.distributions.mixtures), 104
MixtureKernel (class in numpyro.contrib.einstein.stein_kernels), 247
MixtureSameFamily (class in numpyro.distributions.mixtures), 103
mode (ProjectedNormal property), 106
model (BarkerMH property), 141
model (EnsembleSampler property), 155
model (HMC property), 143
model (HMCGibbs property), 147
model (SA property), 154
module
numpyro.contrib.funsor, 201
numpyro.contrib.tfp.distributions, 116
numpyro.contrib.tfp.mcmc, 165
numpyro.diagnostics, 216
numpyro.handlers, 233
numpyro.infer.auto_guide, 180
numpyro.infer.reparam, 197
numpyro.infer.util, 220
numpyro.ops.indexing, 226
numpyro.optim, 205
numpyro.primitives, 13
numpyro.util, 219
module () (in module numpyro.primitives), 17
Momentum (class in numpyro.optim), 210
multinomial () (in module numpyro.distributions.constraints), 119
Multinomial () (in module numpyro.distributions.discrete), 96
MultinomialLogits (class in numpyro.distributions.discrete), 96
MultinomialProbs (class in numpyro.distributions.discrete), 97
MultivariateNormal (class in numpyro.distributions.continuous), 69
MultivariateStudentT (class in numpyro.distributions.continuous), 71
Index 715
NumPyro Documentation

248
probs() (BernoulliLogits method, 84
probs() (BinomialLogits method, 87
probs() (CategoricalLogits method, 89
probs() (GeometricLogits method, 94
probs() (MultinomialLogits method, 96
process_message() (block method, 235
process_message() (collapse method, 235
process_message() (condition method, 236
process_message() (do method, 237
process_message() (enum method, 201
process_message() (infer_config method, 201, 237
process_message() (lift method, 237
process_message() (mask method, 238
process_message() (plate method, 202
process_message() (reparam method, 238
process_message() (replay method, 239
process_message() (scale method, 239
process_message() (scope method, 240
process_message() (seed method, 241
process_message() (substitute method, 242
ProjectedNormal (class
in numpyro.distributions.directional), 105
ProjectedNormalReparam (class
in numpyro.infer.reparam), 199
pytree_aux_fields (CAR attribute), 45
pytree_aux_fields (DirichletMultinomial attribute), 90
pytree_aux_fields (Distribution attribute), 27
pytree_aux_fields (EulerMaruyama attribute), 49
pytree_aux_fields (ExpandedDistribution attribute), 32
pytree_data_fields (GammaPoisson attribute), 93
pytree_data_fields (GaussianCopula attribute), 52
pytree_data_fields (GaussianCopulaBeta attribute), 53
pytree_data_fields (HalfCauchy attribute), 57
pytree_data_fields (HalfNormal attribute), 58
pytree_data_fields (ImproperUniform attribute), 34
pytree_data_fields (Independent attribute), 35
pytree_data_fields (LeftTruncatedDistribution attribute), 111
pytree_data_fields (LKJCholesky attribute), 64
pytree_data_fields (LogUniform attribute), 65
pytree_data_fields (LowRankMultivariateNormal attribute), 67
pytree_data_fields (MaskedDistribution attribute), 37
pytree_data_fields (MixtureGeneral attribute), 104
pytree_data_fields (MixtureSameFamily attribute), 103
pytree_data_fields (MultinomialLogits attribute), 96
pytree_data_fields (MultinomialProbs attribute), 97
pytree_data_fields (MultivariateStudentT attribute), 71
pytree_data_fields (NegativeBinomial2 attribute), 100
pytree_data_fields (RightTruncatedDistribution attribute), 112
pytree_data_fields (SineSkewed attribute), 110
pytree_data_fields (StudentT attribute), 75
pytree_data_fields (TransformedDistribution attribute), 38
pytree_data_fields (TwoSidedTruncatedDistribution attribute), 115
pytree_data_fields (Uniform attribute), 77
pytree_data_fields (ZeroInflatedPoisson attribute), 102
Q
quantiles() (AutoDiagonalNormal method), 185
quantiles() (AutoGuide method), 180
quantiles() (AutoGuideList method), 182
quantiles() (AutoLaplaceApproximation method), 188
quantiles() (AutoLowRankMultivariateNormal method), 190
quantiles() (AutoMultivariateNormal method), 186
quantiles() (AutoNormal method), 191
R
random_flax_module() (in module numpyro.contrib.module), 18
random_haiku_module() (in module numpyro.contrib.module), 21
<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomFeatureKernel</td>
<td>numpyro.contrib.einstein.stein_kernels, 247</td>
</tr>
<tr>
<td>RandomMove()</td>
<td>(ESS static method), 159</td>
</tr>
<tr>
<td>RandomWalkMetropolis</td>
<td>(class in numpyro.contrib.tfp.mcmc), 166</td>
</tr>
<tr>
<td>rate_matrix()</td>
<td>(Wishart method), 79</td>
</tr>
<tr>
<td>rate_matrix()</td>
<td>(WishartCholesky method), 81</td>
</tr>
<tr>
<td>RBFFKernel</td>
<td>(class in numpyro.contrib.einstein.stein_kernels), 247</td>
</tr>
<tr>
<td>real</td>
<td>(in module numpyro.distributions.constraints), 120</td>
</tr>
<tr>
<td>real_vector</td>
<td>(in module numpyro.distributions.constraints), 120</td>
</tr>
<tr>
<td>RealFastFourierTransform</td>
<td>(class in numpyro.distributions.transforms), 126</td>
</tr>
<tr>
<td>RecursiveLinearTransform</td>
<td>(class in numpyro.distributions.transforms), 127</td>
</tr>
<tr>
<td>RelaxedBernoulli()</td>
<td>(in module numpyro.distributions.continuous), 74</td>
</tr>
<tr>
<td>RelaxedBernoulliLogits</td>
<td>(class in numpyro.distributions.continuous), 74</td>
</tr>
<tr>
<td>render_model()</td>
<td>(in module numpyro.infer.inspect), 231</td>
</tr>
<tr>
<td>RenyiELBO</td>
<td>(class in numpyro.infer.elbo), 178</td>
</tr>
<tr>
<td>reparam</td>
<td>(class in numpyro.handlers), 238</td>
</tr>
<tr>
<td>Reparam</td>
<td>(class in numpyro.infer.reparam), 197</td>
</tr>
<tr>
<td>Reparam()</td>
<td>(NeuTraReparam method), 198</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(AsymmetricLaplace attribute), 41</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(AsymmetricLaplaceQuantile attribute), 42</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Beta attribute), 43</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(BetaProportion attribute), 44</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(CAR attribute), 45</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Cauchy attribute), 46</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Chi2 attribute), 47</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Delta attribute), 39</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Dirichlet attribute), 48</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Distribution attribute), 27</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Exponential attribute), 50</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Gamma attribute), 51</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(GaussianCopula attribute), 52</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(GaussianRandomWalk attribute), 54</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Gompertz attribute), 55</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Gumbel attribute), 56</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(HalfCauchy attribute), 57</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(HalfNormal attribute), 58</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Independent property), 35</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(InverseGamma attribute), 59</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Kumaraswamy attribute), 60</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Laplace attribute), 61</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(LeftTruncatedDistribution attribute), 111</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(LKJ attribute), 63</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(LKJCholesky attribute), 64</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Logistic attribute), 66</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(LogNormal attribute), 65</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(LogUniform attribute), 65</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(LowRankMultivariateNormal attribute), 67</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(MatrixNormal attribute), 68</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(MultivariateNormal attribute), 69</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(MultivariateStudentT attribute), 71</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Normal attribute), 72</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Pareto attribute), 73</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(ProjectedNormal attribute), 106</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(RightTruncatedDistribution attribute), 112</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(SoftLaplace attribute), 74</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(StudentT attribute), 75</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(TwoSidedTruncatedDistribution attribute), 115</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Uniform attribute), 77</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(VonMises attribute), 111</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Weibull attribute), 78</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(Wishart attribute), 79</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(WishartCholesky attribute), 81</td>
</tr>
<tr>
<td>reparametrized_params</td>
<td>(ZeroSumNormal attribute), 83</td>
</tr>
<tr>
<td>replay</td>
<td>(class in numpyro.handlers), 238</td>
</tr>
<tr>
<td>ReplicaExchangeMC</td>
<td>(class in numpyro.contrib.tfp.mcmc), 166</td>
</tr>
<tr>
<td>right_scale()</td>
<td>(AsymmetricLaplace method), 41</td>
</tr>
<tr>
<td>RightTruncatedDistribution</td>
<td>(class in numpyro.distributions.truncated), 112</td>
</tr>
<tr>
<td>RMSProp</td>
<td>(class in numpyro.optim), 211</td>
</tr>
<tr>
<td>RMSPropMomentum</td>
<td>(class in numpyro.optim), 212</td>
</tr>
<tr>
<td>rsample()</td>
<td>(Distribution method), 28</td>
</tr>
<tr>
<td>rsample()</td>
<td>(ExpandedDistribution method), 32</td>
</tr>
<tr>
<td>rsample()</td>
<td>(Independent method), 36</td>
</tr>
<tr>
<td>rsample()</td>
<td>(MaskedDistribution method), 37</td>
</tr>
<tr>
<td>rsample()</td>
<td>(TransformedDistribution method), 38</td>
</tr>
<tr>
<td>run()</td>
<td>(MCMC method), 136</td>
</tr>
<tr>
<td>run()</td>
<td>(NestedSampler method), 244</td>
</tr>
<tr>
<td>run()</td>
<td>(StochasticSupportInference method), 249</td>
</tr>
<tr>
<td>run()</td>
<td>(SVI method), 172</td>
</tr>
<tr>
<td>S</td>
<td>(class in numpyro.infer.sa), 153</td>
</tr>
<tr>
<td>sample()</td>
<td>(AsymmetricLaplace method), 41</td>
</tr>
<tr>
<td>sample()</td>
<td>(AsymmetricLaplaceQuantile method), 42</td>
</tr>
</tbody>
</table>
sample() (BarkerMH method), 142  
sample() (BernoulliLogits method), 83  
sample() (BernoulliProbs method), 84  
sample() (Beta method), 43  
sample() (BetaBinomial method), 86  
sample() (BinomialLogits method), 87  
sample() (BinomialProbs method), 87  
sample() (CAR method), 45  
sample() (CategoricalLogits method), 88  
sample() (CategoricalProbs method), 89  
sample() (Cauchy method), 46  
sample() (Delta method), 39  
sample() (Dirichlet method), 48  
sample() (DirichletMultinomial method), 90  
sample() (DiscreteHMCGibbs method), 149  
sample() (DiscreteUniform method), 92  
sample() (Distribution method), 28  
sample() (EnsembleSampler method), 156  
sample() (EulerMaruyama method), 49  
sample() (ExpandedDistribution method), 32  
sample() (Exponential method), 50  
sample() (Gamma method), 51  
sample() (GammaPoisson method), 93  
sample() (GaussianCopula method), 52  
sample() (GaussianRandomWalk method), 54  
sample() (GeometricLogits method), 94  
sample() (GeometricProbs method), 95  
sample() (Gompertz method), 55  
sample() (Gumbel method), 56  
sample() (HalfCauchy method), 57  
sample() (HalfNormal method), 58  
sample() (HMC method), 144  
sample() (HMCECS method), 153  
sample() (HMCgibbs method), 148  
sample() (in module numpyro.distributions.constraints), 13  
sample() (Independent method), 36  
sample() (Kumaraswamy method), 60  
sample() (Laplace method), 61  
sample() (LeftTruncatedDistribution method), 111  
sample() (LKJCholesky method), 64  
sample() (Logistic method), 66  
sample() (LowRankMultivariateNormal method), 67  
sample() (MaskedDistribution method), 37  
sample() (MatrixNormal method), 69  
sample() (MCMCKernel method), 139  
sample() (MixedHMC method), 151  
sample() (MultinomialLogits method), 96  
sample() (MultinomialProbs method), 97  
sample() (MultivariateNormal method), 69  
sample() (MultivariateStudentT method), 71  
sample() (Normal method), 72  
sample() (Poisson method), 101  
sample() (ProjectedNormal method), 106  
sample() (RightTruncatedDistribution method), 112  
sample() (SA method), 154  
sample() (SineBivariatevonMises method), 108  
sample() (SineSkewed method), 110  
sample() (SoftLaplace method), 74  
sample() (StudentT method), 75  
sample() (TransformedDistribution method), 38  
sample() (TruncatedPolyaGamma method), 114  
sample() (TwoSidedTruncatedDistribution method), 115  
sample() (Uniform method), 77  
sample() (Unit method), 40  
sample() (VonMises method), 111  
sample() (Weibull method), 78  
sample() (WishartCholesky method), 81  
sample_field (BarkerMH property), 141  
sample_field (EnsembleSampler property), 155  
sample_field (HMC property), 144  
sample_field (HMCgibbs attribute), 147  
sample_field (MCMCKernel property), 139  
sample_field (SA property), 154  
sample_kernel() (in module numpyro.infer.hmc.hmc), 162  
sample_posterior() (AutoContinuous method), 183  
sample_posterior() (AutoDAIS method), 193  
sample_posterior() (AutoDelta method), 192  
sample_posterior() (AutoGuide method), 180  
sample_posterior() (AutoGuideList method), 181  
sample_posterior() (AutoLaplaceApproximation method), 188  
sample_posterior() (AutoNormal method), 190  
sample_posterior() (AutoSemiDAIS method), 195  
sample_with_intermediates() (Distribution method), 29  
sample_with_intermediates() (ExpandedDistribution method), 32  
sample_with_intermediates() (TransformedDistribution method), 38  
SAState (in module numpyro.infer.sa), 164  
scale (class in numpyro.handlers), 239  
scale_constraint (AutoDiagonalNormal attribute), 184  
scale_constraint (AutoLowRankMultivariateNormal attribute), 189  
scale_constraint (AutoNormal attribute), 190  
scale_matrix() (Wishart method), 79  
scale_matrix() (WishartCholesky method), 81  
scale_tril() (LowRankMultivariateNormal method), 67  
scale_tril() (Wishart method), 80  
scale_tril_constraint (AutoMultivariateNormal attribute), 186  
scaled_unit_lower_cholesky (in module numpyro.distributions.constraints), 120  
ScaledUnitLowerCholeskyTransform (class in numpyro.distributions.transforms), 128
INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

index (class in numpyro.primitives), 297

NumPyro Documentation

scan() (in module numpyro.contrib.control_flow), 21 scope (class in numpyro.handlers), 239 SDVI (class in numpyro.contrib.stochastic_support.sdvi), 250 seed (class in numpyro.handlers), 240 set_default_validate_args() (Distribution static method), 28 set_host_device_count() (in module numpyro.util), 220 set_platform() (in module numpyro.util), 220 SGD (class in numpyro.optim), 214 shape() (Distribution method), 28 SigmoidTransform (class in numpyro.distributions.transforms), 128 simplex (in module numpyro.distributions.transforms), 128 SimplexToOrderedTransform (class in numpyro.distributions.transforms), 128 SineBivariateVonMises (class in numpyro.distributions.directional), 107 SineSkewed (class in numpyro.distributions.directional), 108 SliceSampler (class in numpyro.contrib.tfp.mcmc), 166 SM3 (class in numpyro.optim), 215 SoftLaplace (class in numpyro.distributions.continuous), 74 softplus_lower_cholesky (in module numpyro.distributions.constraints), 120 softplus_positive (in module numpyro.distributions.constraints), 120 SoftplusLowerCholeskyTransform (class in numpyro.distributions.transforms), 129 SoftplusTransform (class in numpyro.distributions.transforms), 129 spectral_density_matern() (in module numpyro.contrib.hsgp.spectral_densities), 257 spectral_density_squared_exponential() (in module numpyro.contrib.hsgp.spectral_densities), 257 sphere (in module numpyro.distributions.constraints), 120 split_gelman_rubin() (in module numpyro.diagnostics), 218 sqrt_eigenvalues() (in module numpyro.contrib.hsgp.laplacian), 255 stable_update() (SVI method), 172 SteinVI (class in numpyro.contrib.einstein.steinvi), 246 StickBreakingTransform (class in numpyro.distributions.transforms), 130 StochasticSupportInference (class in numpyro.contrib.stochastic_support.dcc), 248 StretchMove() (AIES static method), 157

StudentT (class in numpyro.distributions.continuous), 75

subsample() (in module numpyro.primitives), 15

substitute (class in numpyro.handlers), 241

summary() (in module numpyro.diagnostics), 218

support (AsymmetricLaplace attribute), 41

support (AsymmetricLaplaceQuantile attribute), 42

support (BernoulliLogits attribute), 83

support (BernoulliProbs attribute), 84

support (Beta attribute), 43

support (BetaBinomial property), 86

support (BetaProportion attribute), 44

support (BinomialLogits property), 87

support (BinomialProbs property), 88

support (CAR attribute), 45

support (CategoricalLogits property), 89

support (CategoricalProbs property), 90

support (Cauchy attribute), 46

support (Delta property), 39

support (Dirichlet attribute), 48

support (DirichletMultinomial property), 91

support (DiscreteUniform property), 92

support (Distribution attribute), 27

support (EulerMaruyama property), 49

support (ExpandedDistribution property), 32

support (Exponential attribute), 50

support (FoldedDistribution attribute), 33

support (Gamma attribute), 51

support (GammaPoisson attribute), 93

support (GaussianCopula property), 53

support (GaussianCopulaBeta attribute), 53

support (GaussianRandomWalk attribute), 54

support (GeometricLogits attribute), 94

support (GeometricProbs attribute), 95

support (Gompertz attribute), 55

support (Gumbel attribute), 56

support (HalfCauchy attribute), 57

support (HalfNormal attribute), 58

support (ImproperUniform attribute), 34

support (Independent property), 35

support (InverseGamma attribute), 59

support (Kumaraswamy attribute), 60

support (Laplace attribute), 61

support (LeftTruncatedDistribution property), 111

support (LKJ attribute), 63

support (LKJCholesky attribute), 64

support (Logistic attribute), 66

support (LogNormal attribute), 65

support (LogUniform property), 65

support (LowRankMultivariateNormal attribute), 67

support (MaskedDistribution property), 37

support (MatrixNormal attribute), 68

support (MixtureGeneral property), 105

support (MixtureSameFamily property), 103
support (MultinomialLogits property), 97
support (MultinomialProbs property), 98
support (MultivariateNormal attribute), 69
support (MultivariateStudentT attribute), 71
support (NegativeBinomial2 attribute), 100
support (NegativeBinomialLogits attribute), 100
support (NegativeBinomialProbs attribute), 100
support (Normal attribute), 72
support (Pareto property), 73
support (Poisson attribute), 101
support (ProjectedNormal attribute), 106
support (RelaxedBernoulliLogits attribute), 74
support (RightTruncatedDistribution property), 112
support (SineBivariateVonMises attribute), 108
support (SineSkewed attribute), 110
support (SoftLaplace attribute), 74
support (StudentT attribute), 75
support (TransformedDistribution property), 38
support (TruncatedPolyaGamma attribute), 114
support (TwoSidedTruncatedDistribution property), 115
support (Uniform property), 77
support (Unit attribute), 40
support (VonMises attribute), 111
support (Weibull attribute), 78
support (Wishart attribute), 79
support (WishartCholesky attribute), 81
support (ZeroInflatedPoisson attribute), 102
support (ZeroSumNormal property), 83
supported_types (LeftTruncatedDistribution attribute), 111
supported_types (RightTruncatedDistribution attribute), 112
supported_types (TwoSidedTruncatedDistribution attribute), 115
SVI (class in numpyro.infer.svi), 170
SVIRunResult (in module numpyro.infer.svi), 173
SVIState (in module numpyro.infer.svi), 173

T
taylor_proxy() (HMCECS static method), 153
taylor_proxy() (in module numpyro.infer.hmc_gibbs), 162
TFPDistribution (class in numpyro.contrib.tfp.distributions), 116
TFPKernel (class in numpyro.contrib.tfp.mcmc), 165
to_data() (in module numpyro.contrib.funsor.enum_messenger), 202
to_event() (Distribution method), 29
to_funsor() (in module numpyro.contrib.funsor.enum_messenger), 202
trace (class in numpyro.contrib.funsor.enum_messenger), 203

trace (class in numpyro.handlers), 242
Trace_ELBO (class in numpyro.infer.elbo), 175
TraceEnum_ELBO (class in numpyro.infer.elbo), 176
TraceGraph_ELBO (class in numpyro.infer.elbo), 177
TraceMeanField_ELBO (class in numpyro.infer.elbo), 177
transfer_states_to_host() (MCMC method), 138
Transform (class in numpyro.distributions.transforms), 121
transform_fn() (in module numpyro.infer.util), 222
transform_sample() (NeuTraReparam method), 198
TransformedDistribution (class in numpyro.distributions.distribution), 38
TransformReparam (class in numpyro.infer.reparam), 199
tree_flatten() (AffineTransform method), 122
tree_flatten() (BlockNeuralAutoregressiveTransform method), 131
tree_flatten() (CAR method), 46
tree_flatten() (ComposeTransform method), 122
tree_flatten() (Distribution method), 28
tree_flatten() (ExpTransform method), 123
tree_flatten() (InverseAutoregressiveTransform method), 131
tree_flatten() (LowerCholeskyAffine method), 124
tree_flatten() (MaskedDistribution method), 38
tree_flatten() (PermuteTransform method), 126
tree_flatten() (PowerTransform method), 126
tree_flatten() (RealFastFourierTransform method), 126
tree_flatten() (RecursiveLinearTransform method), 127
tree_flatten() (SimplexToOrderedTransform method), 129
tree_flatten() (ZeroSumTransform method), 130
tree_unflatten() (CAR class method), 46
tree_unflatten() (Constraint class method), 117
tree_unflatten() (Distribution class method), 28
tree_unflatten() (MaskedDistribution class method), 38
tree_unflatten() (Transform class method), 121
TruncatedCauchy (class in numpyro.distributions.truncated), 113
TruncatedDistribution (class in module numpyro.distributions.truncated), 113
TruncatedNormal (class in numpyro.distributions.truncated), 114
TruncatedPolyaGamma (class in numpyro.distributions.truncated), 114
truncation_point (TruncatedPolyaGamma attribute), 114
TwoSidedTruncatedDistribution (class in numpyro.distributions.truncated), 115
UncalibratedHamiltonianMonteCarlo (class in numpyro.contrib.tfp.mcmc), 167
UncalibratedLangevin (class in numpyro.contrib.tfp.mcmc), 167
UncalibratedRandomWalk (class in numpyro.contrib.tfp.mcmc), 167
unconstrain_fn() (in module numpyro.infer.util), 223
Uniform (class in numpyro.distributions.continuous), 77
Unit (class in numpyro.distributions.distribution), 40
unit_interval (in module numpyro.distributions.distribution), 120
update() (Adagrad method), 207
update() (Adam method), 206
update() (ClippedAdam method), 208
update() (Minimize method), 210
update() (Momentum method), 211
update() (RMSProp method), 212
update() (RMSPropMomentum method), 213
update() (SGD method), 214
update() (SM3 method), 216
update() (SVI method), 171
update_active_chains() (AIES method), 157
update_active_chains() (EnsembleSampler method), 155
update_active_chains() (ESS method), 159
validation_enabled() (in module numpyro.distributions.distribution), 219
var (LeftTruncatedDistribution property), 112
var (RightTruncatedDistribution property), 113
var (TwoSidedTruncatedDistribution property), 115
variance (AsymmetricLaplace property), 41
variance (AsymmetricLaplaceQuantile property), 43
variance (BernoulliLogits property), 84
variance (BernoulliProbs property), 85
variance (Beta property), 44
variance (BetaBinomial property), 86
variance (BinomialLogits property), 87
variance (BinomialProbs property), 88
variance (CategoricalLogits property), 89
variance (CategoricalProbs property), 90
variance (Cauchy property), 47
variance (Delta property), 40
variance (Dirichlet property), 48
variance (DirichletMultinomial property), 91
variance (DiscreteUniform property), 92
variance (Distribution property), 29
variance (ExpandedDistribution property), 33
variance (Exponential property), 50
variance (Gamma property), 52
variance (GammaPoisson property), 94
variance (GaussianCopula property), 53
variance (GaussianRandomWalk property), 54
variance (GeometricLogits property), 95
variance (GeometricProbs property), 95
variance (Gumbel property), 56
variance (HalfCauchy property), 58
variance (HalfNormal property), 59
variance (Independent property), 36
variance (InverseGamma property), 59
variance (Kumaraswamy property), 60
variance (Laplace property), 61
variance (Logistic property), 66
variance (LogNormal property), 65
variance (LogUniform property), 65
variance (MaskedDistribution property), 38
variance (MultinomialLogits property), 97
variance (MultinomialProbs property), 98
variance (MultivariateNormal property), 70
variance (MultivariateStudentT property), 71
variance (Normal property), 73
variance (Pareto property), 73
variance (Poisson property), 101
variance (SoftLaplace property), 75
variance (StudentT property), 76
variance (TransformedDistribution property), 39
variance (Uniform property), 77
variance (VonMises property), 111
variance (Weibull property), 79
variance (ZeroSumNormal property), 83
variance() (LowRankMultivariateNormal method), 67
variance() (Wishart method), 80
variance() (WishartCholesky method), 81
Vindex (class in numpyro.ops.indexing), 227
vindex() (in module numpyro.ops.indexing), 226
VonMises (class in numpyro.distributions.distribution), 80
Warmup (MCMC method), 136
Weibull (class in numpyro.distributions.continuous), 78
Wishart (class in numpyro.distributions.continuous), 79
WishartCholesky (class in numpyro.distributions.continuous), 80
ZeroInflatedDistribution() (in module numpyro.distributions.discrete), 102
ZeroInflatedNegativeBinomial2() (in module numpyro.distributions.conjugate), 102
ZeroInflatedPoisson (class in numpyro.distributions.discrete), 102
ZeroSumNormal (class in numpyro.distributions.continuous), 82
ZeroSumTransform (class in numpyro.distributions.transforms), 130