NumPyro Documentation

Uber AI Labs

Sep 05, 2022

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH NUMPYRO

Probabilistic programming with NumPy powered by JAX for autograd and JIT compilation to GPU/TPU/CPU.

Docs and Examples | Forum

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).
- **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:

```
>>> 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.

```
>>> 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.

```
>>> from jax import random

>>> from numpyro.infer import MCMC, NUTS
```

(continues on next page)

(continued from previous page)

```
>>> 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.

```
>>> mcmc.print_summary()


                mean       std    median      5.0%     95.0%     n_eff     r_hat
        mu      4.14      3.18      3.87     -0.76      9.50    115.42      1.01
       tau      4.12      3.58      3.12      0.51      8.56     90.64      1.02
  theta[0]      6.40      6.22      5.36     -2.54     15.27    176.75      1.00
  theta[1]      4.96      5.04      4.49     -1.98     14.22    217.12      1.00
  theta[2]      3.65      5.41      3.31     -3.47     13.77    247.64      1.00
  theta[3]      4.47      5.29      4.00     -3.22     12.92    213.36      1.01
  theta[4]      3.22      4.61      3.28     -3.72     10.93    242.14      1.01
  theta[5]      3.89      4.99      3.71     -3.39     12.54    206.27      1.00
  theta[6]      6.55      5.72      5.66     -1.43     15.78    124.57      1.00
  theta[7]      4.81      5.95      4.19     -3.90     13.40    299.66      1.00


Number of divergences: 19


>>> 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_hat`) indicates that the chain has not fully converged. The low value for the effective sample size (`n_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 paramaterization 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_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!

```
>>> 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)


                mean       std    median      5.0%     95.0%     n_eff     r_hat
```

(continues on next page)

(continued from previous page)

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


Number of divergences: 0


>>> 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

```
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.

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


>>> # New School

... def new_school():

...     mu = numpyro.sample('mu', dist.Normal(0, 5))

...     tau = numpyro.sample('tau', dist.HalfCauchy(5))

...     return numpyro.sample('obs', dist.Normal(mu, tau))


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

>>> samples_predictive = predictive(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 implemement generalized 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.
- NestedSampler offers a wrapper for jaxns. See here for an example.

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 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.
- 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.

1.4.3 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:

```
pip install numpyro[cuda] -f https://storage.googleapis.com/jax-releases/jax_cuda_
↪releases.html
```

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

To run **NumPyro on Cloud TPUs**, 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

```
```

, or as a higher order function:

```
```python

def fn():

    x = numpyro.sample('x', dist.Beta(1, 1))

    y = numpyro.sample('y', dist.Bernoulli(x))

    return y



print(handlers.seed(fn, rng_seed=0)())

```
```

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 neural networks. See the *VAE* example 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:

```
@article{phan2019composable,

  title={Composable Effects for Flexible and Accelerated Probabilistic Programming in␣
↪NumPyro},

  author={Phan, Du and Pradhan, Neeraj and Jankowiak, Martin},
```

(continues on next page)

(continued from previous page)

```
  journal={arXiv preprint arXiv:1912.11554},
  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}
}
```

CHAPTER

TWO

PYRO PRIMITIVES

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** (*numpy.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** (*numpy.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** (*numpy.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:

```
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:

```
# 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** (*numpy.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** (*numpy.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** (*numpy.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.:

```
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 numpy.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*, *, *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:

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

or with dropout layers:

```
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
- **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*, *, *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:

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

or with dropout layers:

```
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
- **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*, ***, *input_shape=None*, *apply_rng=None*, *mutable=None*, ***kwargs*)

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

Note: Parameters of a Flax module are stored in a nested dict. For example, the module *B* defined as follows:

```
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:

```
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:

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

- **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

```
# NB: this example is ported from https://github.com/ctallec/pyvarinf/blob/master/
↪main_regression.ipynb
>>> import numpy as np; np.random.seed(0)
>>> 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()
...
```

(continues on next page)

(continued from previous page)

```
>>> 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)
...         batch_y = numpyro.subsample(y, event_dim=0) if y is not None else None
...         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
>>> assert np.sqrt(np.mean(np.square(y_test - y_pred))) < 1
```

2.14 random_haiku_module

random_haiku_module(*name*, *nn_module*, *prior*, *, *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:

```
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:

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

- **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:

```
>>> 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):  # generative
...     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

```
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

```
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.

Note: We can scan over discrete latent variables in *f*. The joint density is evaluated using parallel-scan (reference [1]) over time dimension, which reduces parallel complexity to *O(log(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 *'_PREV_' * (2 * history - 1 - i)*.
- scanned sites: those sites collect the values of the remaining scan loop over *f*. An addition time dimension *_time_foo* will be added to those sites, where *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 [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

1. *Temporal Parallelization of Bayesian Smoothers*, Simo Sarkka, Angel F. Garcia-Fernandez (https://arxiv.org/abs/1905.13002)
2. *Inference with Discrete Latent Variables* (http://pyro.ai/examples/enumeration.html#Dependencies-among-plates)

Parameters

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

Returns output of scan, quoted from `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".

2.16 cond

cond(*pred*, *true_fun*, *false_fun*, *operand*)

This primitive conditionally applies `true_fun` or `false_fun`. See `jax.lax.cond()` for more information.

Usage:

```
>>> 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, *sample* and *deterministic* sites within *true_fun* and *false_fun* are supported. If you notice that any effect handlers or distributions are unsupported, please file an issue.

Warning: The `cond` primitive does not currently support enumeration and can not be used inside a `numpyro.plate` context.

Note: All `sample` sites must belong to the same distribution class. For example the following is not supported

```
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

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

reparametrized_params = []

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 tuple

property event_shape
Returns the shape of a single sample from the distribution without batching.

Returns event shape of the distribution.

Return type tuple

property event_dim

Returns Number of dimensions of individual events.

Return type int

property has_rsample

rsample(*key*, *sample_shape=()*)

shape(*sample_shape=()*)
The tensor shape of samples from this distribution.

Samples are of shape:

```
d.shape(sample_shape) == sample_shape + d.batch_shape + d.event_shape
```

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

Returns shape of samples.

Return type 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=None*)
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.

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:

```
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.distributions import constraints
```

(continues on next page)

(continued from previous page)

```
>>> 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)
...     numpyro.sample("latent_fairness", dist.Beta(alpha_q, beta_q))


>>> data = jnp.concatenate([jnp.ones(5), jnp.zeros(5)])
>>> # select values equal to one
>>> masked_array = jnp.where(data == 1, True, False)
>>> optimizer = numpyro.optim.Adam(step_size=0.05)
>>> svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
>>> svi_result = svi.run(random.PRNGKey(0), 300, data, masked_array)
>>> params = svi_result.params
>>> # 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 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 is_discrete

3.1.2 ExpandedDistribution

class ExpandedDistribution(*base_dist*, *batch_shape=()*)

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {}

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.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.1.3 FoldedDistribution

class FoldedDistribution(*base_dist*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.TransformedDistribution*

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

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

support = GreaterThan(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

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.1.4 ImproperUniform

class ImproperUniform(*support*, *batch_shape*, *event_shape*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

Note: *sample* method is not implemented for this distribution. In autoguide and mcmc, initial parameters for improper sites are derived from *init_to_uniform* or *init_to_value* strategies.

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)))
```

(continues on next page)

(continued from previous page)

```
...
...     # 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.

arg_constraints = {}

support = Dependent()

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

tree_flatten()

3.1.5 Independent

class Independent(*base_dist*, *reinterpreted_batch_ndims*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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:

```
>>> 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

- **base_distribution** (*numpyro.distribution.Distribution*) – a distribution instance.
- **reinterpreted_batch_ndims** (*int*) – the number of batch dims to reinterpret as event dims.

arg_constraints = {}

property support

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 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*

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.1.6 MaskedDistribution

class MaskedDistribution(*base_dist*, *mask*)
Bases: *numpyro.distributions.distribution.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 = {}

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: *numpyro.distributions.distribution.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 = {}`

`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.

tree_flatten()

3.1.8 Delta

class Delta(*v=0.0*, *log_density=0.0*, *event_dim=0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.1.9 Unit

class Unit(*log_factor*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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: *numpyro.distributions.distribution.Distribution*

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

reparametrized_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 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*.

3.2.2 AsymmetricLaplaceQuantile

class AsymmetricLaplaceQuantile(*loc=0.0*, *scale=1.0*, *quantile=0.5*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

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

The *scale* parameter is also interpreted slightly differently than in AsymmetricLaplce. When *loc=0* and *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': GreaterThan(lower_bound=0.0)}

reparametrized_params = ['loc', 'scale', 'quantile']

support = Real()

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 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*.

3.2.3 Beta

class Beta(*concentration1*, *concentration0*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

reparametrized_params = ['concentration1', 'concentration0']

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

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 cummulative distribution function of this distribution.

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

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

3.2.4 BetaProportion

class BetaProportion(*mean*, *concentration*, *, *validate_args=None*)
Bases: *numpyro.distributions.continuous.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** Francisco Cribari-Neto. Journal of Applied Statistics 31.7 (2004): 799-815.

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

reparametrized_params = ['mean', 'concentration']

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

3.2.5 CAR

class CAR(*loc*, *correlation*, *conditional_precision*, *adj_matrix*, ***, *is_sparse=False*, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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

- **or ndarray loc** (*float*) – 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
- **or scipy.sparse.csr_matrix adj_matrix** (*ndarray*) – 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': GreaterThan(lower_bound=0.0), 'correlation': OpenInterval(lower_bound=-1, upper_bound=1), 'loc': IndependentConstraint(Real(), 1)}

support = IndependentConstraint(Real(), 1)

reparametrized_params = ['loc', 'correlation', 'conditional_precision', '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()

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

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

3.2.6 Cauchy

class Cauchy(*loc=0.0*, *scale=1.0*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'loc': Real(), 'scale': GreaterThan(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 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*.

3.2.7 Chi2

class Chi2(*df*, *, *validate_args=None*)
Bases: *numpyro.distributions.continuous.Gamma*

arg_constraints = {'df': GreaterThan(lower_bound=0.0)}

reparametrized_params = ['df']

3.2.8 Dirichlet

class Dirichlet(*concentration*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'concentration':
IndependentConstraint(GreaterThan(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

3.2.9 Exponential

class Exponential(*rate=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

reparametrized_params = ['rate']

arg_constraints = {'rate': GreaterThan(lower_bound=0.0)}

support = GreaterThan(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 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*.

3.2.10 Gamma

class Gamma(*concentration*, *rate=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

support = GreaterThan(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 cummulative distribution function of this distribution.

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

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

3.2.11 Gumbel

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

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'loc': Real(), 'scale': GreaterThan(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 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*.

3.2.12 GaussianRandomWalk

class GaussianRandomWalk(*scale=1.0*, *num_steps=1*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

support = IndependentConstraint(Real(), 1)

reparametrized_params = ['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.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.2.13 HalfCauchy

class HalfCauchy(*scale=1.0*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

reparametrized_params = ['scale']

support = GreaterThan(lower_bound=0.0)

arg_constraints = {'scale': GreaterThan(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

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.

3.2.14 HalfNormal

class HalfNormal(*scale=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

reparametrized_params = ['scale']

support = GreaterThan(lower_bound=0.0)

arg_constraints = {'scale': GreaterThan(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

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.

3.2.15 InverseGamma

class InverseGamma(*concentration*, *rate=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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)

---

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

reparametrized_params = ['concentration', 'rate']

support = GreaterThan(lower_bound=0.0)

property mean
Mean of the distribution.

property variance
Variance of the distribution.

tree_flatten()

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*.

3.2.16 Kumaraswamy

class Kumaraswamy(*concentration1*, *concentration0*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.TransformedDistribution*

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

reparametrized_params = ['concentration1', 'concentration0']

support = Interval(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.

tree_flatten()

3.2.17 Laplace

class Laplace(*loc=0.0*, *scale=1.0*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'loc': Real(), 'scale': GreaterThan(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 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*.

3.2.18 LKJ

class LKJ(*dimension*, *concentration=1.0*, *sample_method='onion'*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.TransformedDistribution*

LKJ distribution for correlation matrices. The distribution is controlled by `concentration` parameter η 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 large determinent. This is useful when we know a priori that the underlying variables are not correlated.

When `concentration < 1`, the distribution favors samples with small determinent. 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:

```
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
```

(continues on next page)

(continued from previous page)

```
    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(theta, theta) * 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

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

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0)}

reparametrized_params = ['concentration']

support = CorrMatrix()

property mean
Mean of the distribution.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.2.19 LKJCholesky

class LKJCholesky(*dimension*, *concentration=1.0*, *sample_method='onion'*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

When `concentration > 1`, the distribution favors samples with large diagonal entries (hence large determinent). This is useful when we know a priori that the underlying variables are not correlated.

When `concentration < 1`, the distribution favors samples with small diagonal entries (hence small determinent). 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:

```
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),
↪ 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

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

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0)}

reparametrized_params = ['concentration']

support = CorrCholesky()

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

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

3.2.20 LogNormal

class LogNormal(*loc=0.0*, *scale=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.TransformedDistribution*

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

support = GreaterThan(lower_bound=0.0)

reparametrized_params = ['loc', 'scale']

property mean
Mean of the distribution.

property variance
Variance of the distribution.

tree_flatten()

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*.

3.2.21 Logistic

class Logistic(*loc=0.0*, *scale=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'loc': Real(), 'scale': GreaterThan(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 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*.

3.2.22 MultivariateNormal

`class MultivariateNormal`(*loc=0.0*, *covariance_matrix=None*, *precision_matrix=None*, *scale_tril=None*, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

```
support = IndependentConstraint(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

`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.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

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__()`.

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

3.2.23 MultivariateStudentT

class MultivariateStudentT(*df*, *loc=0.0*, *scale_tril=None*, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'df': GreaterThan(lower_bound=0.0), 'loc': IndependentConstraint(Real(), 1), 'scale_tril': LowerCholesky()}

support = IndependentConstraint(Real(), 1)

reparametrized_params = ['df', 'loc', '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

`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

3.2.24 LowRankMultivariateNormal

`class LowRankMultivariateNormal`(*loc*, *cov_factor*, *cov_diag*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

`support = IndependentConstraint(Real(), 1)`

`reparametrized_params = ['loc', 'cov_factor', 'cov_diag']`

`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()

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

3.2.25 Normal

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

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'loc': Real(), 'scale': GreaterThan(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 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.

3.2.26 Pareto

class Pareto(*scale*, *alpha*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.TransformedDistribution*

arg_constraints = {'alpha': GreaterThan(lower_bound=0.0), 'scale': GreaterThan(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 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*.

tree_flatten()

3.2.27 RelaxedBernoulli

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

3.2.28 RelaxedBernoulliLogits

class RelaxedBernoulliLogits(*temperature*, *logits*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.TransformedDistribution*

arg_constraints = {'logits': Real(), 'temperature': GreaterThan(lower_bound=0.0)}

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

tree_flatten()

3.2.29 SoftLaplace

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

Bases: *numpyro.distributions.distribution.Distribution*

Smooth distribution with Laplace-like tail behavior.

This distribution corresponds to the log-convex density:

```
z = (value - loc) / scale
log_prob = log(2 / pi) - log(scale) - logaddexp(z, -z)
```

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': GreaterThan(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 *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

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.

3.2.30 StudentT

class StudentT(*df*, *loc=0.0*, *scale=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

support = Real()

reparametrized_params = ['df', '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 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*.

3.2.31 Uniform

class Uniform(*low=0.0*, *high=1.0*, *, *validate_args=None*)

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'high': Dependent(), 'low': Dependent()}

reparametrized_params = ['low', 'high']

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.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

static infer_shapes(*low=()*, *high=()*)

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

3.2.32 Weibull

class Weibull(*scale*, *concentration*, *, *validate_args=None*)

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'scale': GreaterThan(lower_bound=0.0)}

support = GreaterThan(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 cummulative distribution function of this distribution.

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

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

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: *numpyro.distributions.distribution.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.

3.3.3 BernoulliProbs

class BernoulliProbs(*probs*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'probs': Interval(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

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.

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

3.3.4 BetaBinomial

class BetaBinomial(*concentration1*, *concentration0*, *total_count=1*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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.

arg_constraints = {'concentration0': GreaterThan(lower_bound=0.0), 'concentration1': GreaterThan(lower_bound=0.0), 'total_count': IntegerGreaterThan(lower_bound=0)}

has_enumerate_support = True

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

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: *numpyro.distributions.distribution.Distribution*

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

has_enumerate_support = True

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

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: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'probs': Interval(lower_bound=0.0, upper_bound=1.0), 'total_count': IntegerGreaterThan(lower_bound=0)}

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.

3.3.8 Categorical

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

3.3.9 CategoricalLogits

class CategoricalLogits(*logits*, *, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'logits': IndependentConstraint(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.

3.3.10 CategoricalProbs

class CategoricalProbs(*probs*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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.

3.3.11 DirichletMultinomial

class DirichletMultinomial(*concentration*, *total_count=1*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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(GreaterThan(lower_bound=0.0), 1), 'total_count': IntegerGreaterThan(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

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

class DiscreteUniform(*low=0*, *high=1*, ***, *validate_args=None*)

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'high': Dependent(), 'low': Dependent()}

has_enumerate_support = True

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.

enumerate_support(*expand=True*)

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

3.3.13 GammaPoisson

class GammaPoisson(*concentration*, *rate=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.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': GreaterThan(lower_bound=0.0), 'rate': GreaterThan(lower_bound=0.0)}

support = IntegerGreaterThan(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

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 cummulative 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: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'logits': Real()}

support = IntegerGreaterThan(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.

3.3.16 GeometricProbs

class GeometricProbs(*probs*, ***, *validate_args=None*)

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'probs': Interval(lower_bound=0.0, upper_bound=1.0)}

support = IntegerGreaterThan(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.

3.3.17 Multinomial

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

3.3.18 MultinomialLogits

class MultinomialLogits(*logits*, *total_count=1*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'logits': IndependentConstraint(Real(), 1), 'total_count': IntegerGreaterThan(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

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*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'probs': Simplex(), 'total_count': IntegerGreaterThan(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.

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: *numpyro.distributions.discrete.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

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: *numpyro.distributions.conjugate.GammaPoisson*

arg_constraints = {'logits': Real(), 'total_count': GreaterThan(lower_bound=0.0)}

support = IntegerGreaterThan(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: *numpyro.distributions.conjugate.GammaPoisson*

arg_constraints = {'probs': Interval(lower_bound=0.0, upper_bound=1.0), 'total_count': GreaterThan(lower_bound=0.0)}

support = IntegerGreaterThan(lower_bound=0)

3.3.24 NegativeBinomial2

class NegativeBinomial2(*mean*, *concentration*, ***, *validate_args=None*)

Bases: *numpyro.distributions.conjugate.GammaPoisson*

Another parameterization of GammaPoisson with *rate* is replaced by *mean*.

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'mean': GreaterThan(lower_bound=0.0)}

support = IntegerGreaterThan(lower_bound=0)

3.3.25 Poisson

class Poisson(*rate*, ***, *is_sparse=False*, *validate_args=None*)

Bases: *numpyro.distributions.distribution.Distribution*

Creates a Poisson distribution parameterized by rate, the rate parameter.

Samples are nonnegative integers, with a pmf given by

$$\text{rate}^k \frac{e^{-\text{rate}}}{k!}$$

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.

arg_constraints = {'rate': GreaterThan(lower_bound=0.0)}

support = IntegerGreaterThan(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

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 cummulative distribution function evaluated at *value*.

3.3.26 ZeroInflatedDistribution

ZeroInflatedDistribution(*base_dist*, ***, *gate=None*, *gate_logits=None*, *validate_args=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=None*)
Bases: numpyro.distributions.discrete.ZeroInflatedProbs

A Zero Inflated Poisson distribution.

Parameters

- **gate** (*numpy.ndarray*) – probability of extra zeros.
- **rate** (*numpy.ndarray*) – rate of Poisson distribution.

arg_constraints = {'gate': Interval(lower_bound=0.0, upper_bound=1.0), 'rate': GreaterThan(lower_bound=0.0)}

support = IntegerGreaterThan(lower_bound=0)

3.3.28 ZeroInflatedNegativeBinomial2

ZeroInflatedNegativeBinomial2(*mean*, *concentration*, ***, *gate=None*, *gate_logits=None*, *validate_args=None*)

3.4 Mixture Distributions

3.4.1 Mixture

Mixture(*mixing_distribution*, *component_distributions*, ***, *validate_args=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

class MixtureSameFamily(*mixing_distribution*, *component_distribution*, *, *validate_args=None*)

Bases: `numpyro.distributions.mixtures._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

```
>>> 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
()
```

property component_distribution

Return the vectorized distribution of components being mixed.

Returns Component distribution

Return type *Distribution*

property support

property is_discrete

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

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*, *, *validate_args=None*)

Bases: numpyro.distributions.mixtures._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.

Example

```
>>> 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
()
```

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

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

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

class ProjectedNormal(*concentration*, ***, *validate_args=None*)

Bases: *numpyro.distributions.distribution.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.:

```
@handlers.reparam(config={"direction": ProjectedNormalReparam()})
def model():
    direction = numpyro.sample("direction",
                               ProjectedNormal(zeros(3)))
    ...
```

Note: This implements *log_prob()* only for dimensions {2,3}.

[1] D. Hernandez-Stumpfhauser, F.J. Breidt, M.J. van der Woerd (2017) "The General Projected Normal Distribution of Arbitrary Dimension: Modeling and Bayesian Inference" https://projecteuclid.org/euclid.ba/1453211962

arg_constraints = {'concentration': IndependentConstraint(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: *numpyro.distributions.distribution.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

$$C=(2\pi)^2\sum_{i=0}\binom{2i}{i}\left(\frac{\rho^2}{4\kappa_1\kappa_2}\right)^i I_i(\kappa_1)I_i(\kappa_2),$$

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: **

1. Probabilistic model for two dependent circular variables Singh, H., Hnizdo, V., and Demchuck, E. (2002)

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 weigthed_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': GreaterThan(lower_bound=0.0), 'phi_loc': Interval(lower_bound=-3.141592653589793, upper_bound=3.141592653589793), 'psi_concentration': GreaterThan(lower_bound=0.0), 'psi_loc': Interval(lower_bound=-3.141592653589793, upper_bound=3.141592653589793)}

support = IndependentConstraint(Interval(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: **

1. A New Unified Approach for the Simulation of a Wide Class of Directional Distributions John T. Kent, Asaad M. Ganeiber & Kanti V. Mardia (2018)

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:* numpyro.distributions.distribution.Distribution, *skewness*, *, *validate_args=None*)

Bases: *numpyro.distributions.distribution.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,:

```
@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('skew_phi', 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** Ameijeiras-Alonso, J., Ley, C. (2019)

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()}

support = IndependentConstraint(Interval(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: *numpyro.distributions.distribution.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* reparametrizer in the model, e.g.:

```
@handlers.reparam(config={"direction": CircularReparam()})
def model():
    direction = numpyro.sample("direction", VonMises(0.0, 4.0))
    ...
```

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'loc': Real()}

reparametrized_params = ['loc']

support = Interval(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: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'low': Real()}

reparametrized_params = ['low']

supported_types = (<class 'numpyro.distributions.continuous.Cauchy'>, <class 'numpyro.distributions.continuous.Laplace'>, <class 'numpyro.distributions.continuous.Logistic'>, <class 'numpyro.distributions.continuous.Normal'>, <class 'numpyro.distributions.continuous.SoftLaplace'>, <class 'numpyro.distributions.continuous.StudentT'>)

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

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

property mean
Mean of the distribution.

property var

3.6.2 RightTruncatedDistribution

class RightTruncatedDistribution(*base_dist*, *high=0.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'high': Real()}

reparametrized_params = ['high']

supported_types = (<class 'numpyro.distributions.continuous.Cauchy'>, <class 'numpyro.distributions.continuous.Laplace'>, <class 'numpyro.distributions.continuous.Logistic'>, <class 'numpyro.distributions.continuous.Normal'>, <class 'numpyro.distributions.continuous.SoftLaplace'>, <class 'numpyro.distributions.continuous.StudentT'>)

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

tree_flatten()

classmethod tree_unflatten(*aux_data*, *params*)

property mean
Mean of the distribution.

property var

3.6.3 TruncatedCauchy

class TruncatedCauchy(*loc=0.0*, *scale=1.0*, ***, *low=None*, *high=None*, *validate_args=None*)

3.6.4 TruncatedDistribution

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*)

3.6.6 TruncatedPolyaGamma

class TruncatedPolyaGamma(*batch_shape=()*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

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

`tree_flatten`()

`classmethod tree_unflatten`(*aux_data*, *params*)

3.6.7 TwoSidedTruncatedDistribution

`class TwoSidedTruncatedDistribution`(*base_dist*, *low=0.0*, *high=1.0*, ***, *validate_args=None*)
Bases: *numpyro.distributions.distribution.Distribution*

`arg_constraints = {'high': Dependent(), 'low': Dependent()}`

`reparametrized_params = ['low', 'high']`

`supported_types = (<class 'numpyro.distributions.continuous.Cauchy'>, <class 'numpyro.distributions.continuous.Laplace'>, <class 'numpyro.distributions.continuous.Logistic'>, <class 'numpyro.distributions.continuous.Normal'>, <class 'numpyro.distributions.continuous.SoftLaplace'>, <class 'numpyro.distributions.continuous.StudentT'>)`

`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

`tree_flatten`()

`classmethod tree_unflatten`(*aux_data*, *params*)

`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 = TFPDistribution[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.:

```
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*.

3.8.2 boolean

boolean = Boolean()

3.8.3 circular

circular = Interval(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 optimized.

3.8.15 nonnegative_integer

nonnegative_integer = IntegerGreaterThan(lower_bound=0)

3.8.16 ordered_vector

ordered_vector = OrderedVector()

3.8.17 positive

positive = GreaterThan(lower_bound=0.0)

3.8.18 positive_definite

positive_definite = PositiveDefinite()

3.8.19 positive_integer

positive_integer = IntegerGreaterThan(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 = IndependentConstraint(Real(), 1)
Wraps a constraint by aggregating over reinterpreted_batch_ndims-many dims in check(), so that an event is valid only if all its independent entries are valid.

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 = Interval(lower_bound=0.0, upper_bound=1.0)

3.9 Transforms

3.9.1 biject_to

biject_to(*constraint*)

3.9.2 Transform

class Transform
Bases: object

domain = Real()

codomain = Real()

property inv

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.

3.9.3 AbsTransform

class AbsTransform
Bases: *numpyro.distributions.transforms.Transform*

domain = Real()

codomain = GreaterThan(lower_bound=0.0)

3.9.4 AffineTransform

class AffineTransform(*loc*, *scale*, *domain=Real()*)
Bases: *numpyro.distributions.transforms.Transform*

Note: When *scale* is a JAX tracer, we always assume that *scale* > *0* when calculating *codomain*.

---

property codomain

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.

3.9.5 CholeskyTransform

class CholeskyTransform

Bases: *numpyro.distributions.transforms.Transform*

Transform via the mapping $y = cholesky(x)$, 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: *numpyro.distributions.transforms.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.

3.9.7 CorrCholeskyTransform

class CorrCholeskyTransform

Bases: *numpyro.distributions.transforms.Transform*

Transforms a uncontrained real vector x with length $D * (D-1)/2$ 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 *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 $s_i = StickBreakingTransform(z_i)$.

   d. Transforms back into signed domain: $y_i = (sign(r_i), 1) * \sqrt{s_i}$.

domain = IndependentConstraint(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

class CorrMatrixCholeskyTransform
Bases: *numpyro.distributions.transforms.CholeskyTransform*

Transform via the mapping $y = cholesky(x)$, where x is a correlation matrix.

domain = CorrMatrix()

codomain = CorrCholesky()

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.9 ExpTransform

class ExpTransform(*domain=Real()*)
Bases: *numpyro.distributions.transforms.Transform*

property codomain

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.10 IdentityTransform

class IdentityTransform
Bases: *numpyro.distributions.transforms.Transform*

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.11 L1BallTransform

class L1BallTransform
Bases: *numpyro.distributions.transforms.Transform*

Transforms a uncontrained real vector x into the unit L1 ball.

domain = IndependentConstraint(Real(), 1)

codomain = L1Ball()

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.12 LowerCholeskyAffine

class LowerCholeskyAffine(*loc*, *scale_tril*)

Bases: *numpyro.distributions.transforms.Transform*

Transform via the mapping $y = loc + scale\_tril \mathbin{@} x$.

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)
DeviceArray([0.3, 1.5], dtype=float32)
```

domain = IndependentConstraint(Real(), 1)

codomain = IndependentConstraint(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.

3.9.13 LowerCholeskyTransform

class LowerCholeskyTransform

Bases: *numpyro.distributions.transforms.Transform*

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 = IndependentConstraint(Real(), 1)

codomain = LowerCholesky()

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.

3.9.14 OrderedTransform

class OrderedTransform

Bases: *numpyro.distributions.transforms.Transform*

Transform a real vector to an ordered vector.

References:

1. *Stan Reference Manual v2.20, section 10.6*, Stan Development Team

Example

```
>>> 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 = IndependentConstraint(Real(), 1)

codomain = OrderedVector()

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.15 PermuteTransform

class PermuteTransform(*permutation*)

Bases: *numpyro.distributions.transforms.Transform*

domain = IndependentConstraint(Real(), 1)

codomain = IndependentConstraint(Real(), 1)

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.16 PowerTransform

class PowerTransform(*exponent*)

Bases: *numpyro.distributions.transforms.Transform*

domain = GreaterThan(lower_bound=0.0)

codomain = GreaterThan(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.

3.9.17 ScaledUnitLowerCholeskyTransform

class ScaledUnitLowerCholeskyTransform

Bases: *numpyro.distributions.transforms.LowerCholeskyTransform*

Like *LowerCholeskyTransform* this *Transform* transforms a real vector to a lower triangular cholesky factor. However it does so via a decomposition

$y = loc + unit\_scale\_tril \mathbin{@} scale\_diag \mathbin{@} x.$

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 = IndependentConstraint(Real(), 1)

codomain = ScaledUnitLowerCholesky()

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.18 SigmoidTransform

class SigmoidTransform

Bases: *numpyro.distributions.transforms.Transform*

codomain = Interval(lower_bound=0.0, upper_bound=1.0)

log_abs_det_jacobian(*x*, *y*, *intermediates=None*)

3.9.19 SimplexToOrderedTransform

class SimplexToOrderedTransform(*anchor_point=0.0*)

Bases: *numpyro.distributions.transforms.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:

1. *Ordinal Regression Case Study, section 2.2*, M. Betancourt, https://betanalpha.github.io/assets/case_studies/ordinal_regression.html

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*)

3.9.20 SoftplusLowerCholeskyTransform

class SoftplusLowerCholeskyTransform

Bases: *numpyro.distributions.transforms.Transform*

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 = IndependentConstraint(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.

3.9.21 SoftplusTransform

class SoftplusTransform

Bases: *numpyro.distributions.transforms.Transform*

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*)

3.9.22 StickBreakingTransform

class StickBreakingTransform

Bases: *numpyro.distributions.transforms.Transform*

domain = IndependentConstraint(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.

3.10 Flows

3.10.1 InverseAutoregressiveTransform

class InverseAutoregressiveTransform(*autoregressive_nn*, *log_scale_min_clip=- 5.0*, *log_scale_max_clip=3.0*)

Bases: *numpyro.distributions.transforms.Transform*

An implementation of Inverse Autoregressive Flow, using Eq (10) from Kingma et al., 2016,

$$\mathbf{y} = \mu_t + \sigma_t \odot \mathbf{x}$$

where $\mathbf{x}$ are the inputs, $\mathbf{y}$ are the outputs, μ_t, σ_t are calculated from an autoregressive network on $\mathbf{x}$, and $\sigma_t > 0$.

References

1. *Improving Variational Inference with Inverse Autoregressive Flow* [arXiv:1606.04934], Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, Max Welling

domain = IndependentConstraint(Real(), 1)

codomain = IndependentConstraint(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

3.10.2 BlockNeuralAutoregressiveTransform

class BlockNeuralAutoregressiveTransform(*bn_arn*)

Bases: *numpyro.distributions.transforms.Transform*

An implementation of Block Neural Autoregressive flow.

References

1. *Block Neural Autoregressive Flow*, Nicola De Cao, Ivan Titov, Wilker Aziz

domain = IndependentConstraint(Real(), 1)

codomain = IndependentConstraint(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

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 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.
- NestedSampler offers a wrapper for jaxns. See here for an example.

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.

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 launced 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*.

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*) – 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,

```
def do_mcmc(rng_key, n_vectorized=8):
    nuts_kernel = NUTS(model)
    mcmc = MCMC(
        nuts_kernel,
```

(continues on next page)

(continued from previous page)

```
        progress_bar=False,
        num_chains=n_vectorized,
        chain_method='vectorized'
    )
    mcmc.run(
        rng_key,
        extra_fields=("potential_energy",),
    )
    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,

```
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.

warmup(*rng_key*, **args*, *extra_fields=()*, *collect_warmup=False*, *init_params=None*, ***kwargs*)

Run the MCMC warmup adaptation phase. After this call, *self.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.
- **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*.

- **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 in to 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"*) 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.
- **init_params** – Initial parameters to begin sampling. The type must be consistent with the input type to *potential_fn*.
- **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 and 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*:

```
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.

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.

4.1.1 MCMC Kernels

MCMCKernel

class MCMCKernel

Bases: abc.ABC

Defines the interface for the Markov transition kernel that is used for *MCMC* inference.

Example:

```
>>> 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))
...         u_new = jnp.where(dist.Uniform().sample(key_accept) < accept_prob, u_
↪proposal, u)
...         return MHState(u_new, rng_key)

>>> 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).

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: *numpyro.infer.mcmc.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:

1. The Barker proposal: combining robustness and efficiency in gradient-based MCMC. Samuel Livingstone, Giacomo Zanella.

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 adapation. Defaults to `target_accept_prob=0.4`.
- **init_strategy** (*callable*) – a per-site initialization function. See *Initialization Strategies* section for available functions.

Example

```
>>> 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))
```

(continues on next page)

(continued from previous page)

```
>>>
>>> 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 *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*.

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: *numpyro.infer.mcmc.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 *init()* 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 2π.
- **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

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: *numpyro.infer.hmc.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: *numpyro.infer.mcmc.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

```
>>> 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]))
...
>>> 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}
```

(continues on next page)

(continued from previous page)

```
...
>>> 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_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*.

DiscreteHMCGibbs

class DiscreteHMCGibbs(*inner_kernel*, *, *random_walk=False*, *modified=False*)
Bases: *numpyro.infer.hmc_gibbs.HMCGibbs*

[EXPERIMENTAL INTERFACE]

A subclass of *HMCGibbs* which performs Metropolis updates for discrete latent sites.

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.

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.
- **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:

1. *Peskun's theorem and a modified discrete-state Gibbs sampler*, Liu, J. S. (1996)

Example

```
>>> 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*.
- **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: *numpyro.infer.hmc_gibbs.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

1. *Mixed Hamiltonian Monte Carlo for Mixed Discrete and Continuous Variables*, Guangyao Zhou (2020)
2. *Peskun's theorem and a modified discrete-state Gibbs sampler*, Liu, J. S. (1996)

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(gibbs_site | remaining sites)*, where *gibbs_site* is one of the discrete sample sites in the model. Otherwise, a sample will be drawn uniformly from the domain of *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

```
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import HMC, MCMC, MixedHMC
...
>>> 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 = MixedHMC(HMC(model, trajectory_length=1.2), num_discrete_updates=20)
>>> 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()
>>> assert "x" in samples and "c" in samples
>>> assert abs(jnp.mean(samples["x"]) - 1.3) < 0.1
>>> assert abs(jnp.var(samples["x"]) - 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*.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

Returns Next *state*.

HMCECS

class **HMCECS**(*inner_kernel*, *, *num_blocks=1*, *proxy=None*)
Bases: *numpyro.infer.hmc_gibbs.HMCGibbs*

[EXPERIMENTAL INTERFACE]

HMC with Energy Conserving Subsampling.

A subclass of *HMCGibbs* for performing HMC-within-Gibbs for models with subsample statements using the `plate` primitive. This implements Algorithm 1 of reference [1] but uses a naive estimation (without control variates) of log likelihood, hence might incur a high variance.

The function can divide subsample indices into blocks and update only one block at each MCMC step to improve the acceptance rate of proposed subsamples as detailed in [3].

Note: New subsample indices are proposed randomly with replacement at each MCMC step.

References:

1. *Hamiltonian Monte Carlo with energy conserving subsampling*, Dang, K. D., Quiroz, M., Kohn, R., Minh-Ngoc, T., & Villani, M. (2019)
2. *Speeding Up MCMC by Efficient Data Subsampling*, Quiroz, M., Kohn, R., Villani, M., & Tran, M. N. (2018)
3. *The Block Pseudo-Margional Sampler*, Tran, M.-N., Kohn, R., Quiroz, M. Villani, M. (2017)
4. *The Fundamental Incompatibility of Scalable Hamiltonian Monte Carlo and Naive Data Subsampling* Betancourt, M. (2015)

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

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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*.

static taylor_proxy(*reference_params*)
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: *numpyro.infer.mcmc.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:

1. *Sample Adaptive MCMC* (https://papers.nips.cc/paper/9107-sample-adaptive-mcmc), Michael Zhu

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.

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_kwargs*)
Run SA from the given *SAState* and return the resulting *SAState*.

Parameters

- **state** (*SAState*) – Represents the current state.
- **model_args** – Arguments provided to the model.
- **model_kwargs** – Keyword arguments provided to the model.

Returns Next *state* after running SA.

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
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

```
>>> 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,))
>>> 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(*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 2π.
- **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.
- **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.

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.

taylor_proxy(*reference_params*)

Control variate for unbiased log likelihood estimation using a Taylor expansion around a reference parameter. Suggest for subsampling in [1].

Parameters **reference_params** (*dict*) – Model parameterization at MLE or MAP-estimate.

References:

[1] Towards scaling up Markov chain Monte Carlo: an adaptive subsampling approach Bardenet., R., Doucet, A., Holmes, C. (2014)

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 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.

HMCGibbsState = <class 'numpyro.infer.hmc_gibbs.HMCGibbsState'>

- **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.

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

class TFPKernel(*model=None*, *potential_fn=None*, *init_strategy=<function init_to_uniform>*, ***kernel_kwargs*)

A thin wrapper for TensorFlow Probability (TFP) MCMC transition kernels. The argument *target_log_prob_fn* in TFP is replaced by either *model* or *potential_fn* (which is the negative of *target_log_prob_fn*).

This class can be used to convert a TFP kernel to a NumPyro-compatible one as follows:

```
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.hmc.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.hmc.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.langevin.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.random_walk_metropolis.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 namedtupe *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>*, *progbar=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, *progbar=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_fn*.
- **progbar** – 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:

1. *Bayes and big data: The consensus Monte Carlo algorithm*, Steven L. Scott, Alexander W. Blocker, Fernando V. Bonassi, Hugh A. Chipman, Edward I. George, Robert E. McCulloch

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 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.

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:

```
>>> 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 Predictive, SVI, Trace_ELBO

>>> def model(data):
...     f = numpyro.sample("latent_fairness", dist.Beta(10, 10))
```

(continues on next page)

(continued from previous page)

```
...     with numpyro.plate("N", data.shape[0]):
...         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)
>>> svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
>>> svi_result = svi.run(random.PRNGKey(0), 2000, data)
>>> params = svi_result.params
>>> inferred_mean = params["alpha_q"] / (params["alpha_q"] + params["beta_q"])
>>> # get posterior samples
>>> predictive = Predictive(guide, params=params, num_samples=1000)
>>> samples = predictive(random.PRNGKey(1), data)
```

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(*rng_key*, **args*, ***kwargs*)

Gets the initial SVI state.

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).
- **kwargs** – keyword arguments to the model / guide (these can possibly vary during the course of fitting).

Returns the initial `SVIState`

get_params(*svi_state*)

Gets values at *param* sites of the *model* and *guide*.

Parameters **svi_state** – current state of SVI.

Returns the corresponding parameters

update(*svi_state*, **args*, ***kwargs*)
Take a single step of SVI (possibly on a batch / minibatch of data), using the optimizer.

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 (these can possibly vary during the course of fitting).

Returns tuple of *(svi_state, loss)*.

stable_update(*svi_state*, **args*, ***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).
- **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*, *init_state=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.
- **init_state** (*SVIState*) – if not None, begin SVI from the final state of previous SVI run. Usage:

```
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
```

(continues on next page)

(continued from previous page)

```
# 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)
```

- **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*).

4.2.1 ELBO

class ELBO(*num_particles=1*)
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.

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 dictionay containing ELBO loss and the mutable state

4.2.2 Trace_ELBO

class Trace_ELBO(*num_particles=1*)

Bases: *numpyro.infer.elbo.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:

1. *Automated Variational Inference in Probabilistic Programming*, David Wingate, Theo Weber
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.

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 dictionay containing ELBO loss and the mutable state

4.2.3 TraceGraph_ELBO

class TraceGraph_ELBO(*num_particles=1*)
Bases: *numpyro.infer.elbo.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

[1] ***Gradient Estimation Using Stochastic Computation Graphs***, John Schulman, Nicolas Heess, Theophane Weber, Pieter Abbeel

[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.

4.2.4 TraceMeanField_ELBO

class TraceMeanField_ELBO(*num_particles=1*)
Bases: *numpyro.infer.elbo.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 dictionay containing ELBO loss and the mutable state

4.2.5 RenyiELBO

class RenyiELBO(*alpha=0*, *num_particles=2*)

Bases: *numpyro.infer.elbo.ELBO*

An implementation of Renyi's α-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].

Note: Setting $\alpha < 1$ gives a better bound than the usual ELBO.

Parameters

- **alpha** (*float*) – The order of α-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.

References:

1. *Renyi Divergence Variational Inference*, Yingzhen Li, Richard E. Turner
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.

4.3.1 AutoGuide

class AutoGuide(*model*, ***, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *create_plates=None*)
Bases: abc.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:

```
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 AutoContinuous

class AutoContinuous(*model*, *, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *create_plates=None*)
Bases: *numpyro.infer.autoguide.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

4.3.3 AutoBNAFNormal

class AutoBNAFNormal(*model, *, prefix='auto', init_loc_fn=<function init_to_uniform>, num_flows=1, hidden_factors=[8, 8]*)
Bases: *numpyro.infer.autoguide.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*.

4.3.4 AutoDiagonalNormal

class AutoDiagonalNormal(*model*, *, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *init_scale=0.1*)
Bases: *numpyro.infer.autoguide.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

4.3.5 AutoMultivariateNormal

class AutoMultivariateNormal(*model*, *, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *init_scale=0.1*)

Bases: *numpyro.infer.autoguide.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:

```
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 AutoIAFNormal

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: *numpyro.infer.autoguide.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:

```
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*.

4.3.7 AutoLaplaceApproximation

class AutoLaplaceApproximation(*model*, *, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *create_plates=None*, *hessian_fn=None*)

Bases: *numpyro.infer.autoguide.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 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:

```
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 `lambda f, x: jax.hessian(f)(x)`. Other alternatives can be `lambda f, x: jax.jacobian(jax.jacobian(f))(x)` or `lambda f, x: jax.hessian(f)(x) + 1e-3 * jnp.eye(x.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:

```
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

4.3.8 AutoLowRankMultivariateNormal

class AutoLowRankMultivariateNormal(*model*, ***, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *init_scale=0.1*, *rank=None*)

Bases: *numpyro.infer.autoguide.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:

```
guide = AutoLowRankMultivariateNormal(model, rank=2, ...)
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 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

4.3.9 AutoNormal

class AutoNormal(*model*, ***, *prefix='auto'*, *init_loc_fn=<function init_to_uniform>*, *init_scale=0.1*, *create_plates=None*)

Bases: *numpyro.infer.autoguide.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 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

4.3.10 AutoDelta

class AutoDelta(*model*, *, *prefix='auto'*, *init_loc_fn=<function init_to_median>*, *create_plates=None*)

Bases: *numpyro.infer.autoguide.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`.

Note: This class does MAP inference in constrained space.

Usage:

```
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*, *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

4.3.11 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: *numpyro.infer.autoguide.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 ().

Returns a dict containing samples drawn the this guide.

Return type dict

4.3.12 AutoSemiDAIS

class AutoSemiDAIS(*model*, *local_model*, *base_guide*, ***, *prefix='auto'*, *K=4*, *eta_init=0.01*, *eta_max=0.1*, *gamma_init=0.9*, *init_scale=0.1*)

Bases: *numpyro.infer.autoguide.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())
base_guide = AutoNormal(global_model)
guide = AutoSemiDAIS(model, local_model, base_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.
- **base_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.
- **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.

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

4.3.13 AutoSurrogateLikelihoodDAIS

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: *numpyro.infer.autoguide.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.

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:

```
# 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.
- **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.

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.ABC

Base class for reparameterizers.

4.4.1 Loc-Scale Decentering

class LocScaleReparam(*centered=None*, *shape_params=()*)

Bases: *numpyro.infer.reparam.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:

1. *Automatic Reparameterisation of Probabilistic Programs*, Maria I. Gorinova, Dave Moore, Matthew D. Hoffman (2019)

Parameters

- **centered** (*float*) – optional centered parameter. If None (default) learn a per-site per-element centering parameter in `[0,1]`. 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 (`new_fn`, `value`).

4.4.2 Neural Transport

class NeuTraReparam(*guide*, *params*)

Bases: *numpyro.infer.reparam.Reparam*

Neural Transport reparameterizer [1] of multiple latent variables.

This uses a trained *AutoContinuous* guide to alter the geometry of a model, typically for use e.g. in MCMC. Example usage:

```
# 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 = netra.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 *NeuTraReparam* instance, and that the model must have static structure.

[1] Hoffman, M. et al. (2019) "NeuTra-lizing Bad Geometry in Hamiltonian Monte Carlo Using Neural Transport" https://arxiv.org/abs/1903.03704

Parameters

- **guide** (AutoContinuous) – A guide.
- **params** – trained parameters of the guide.

reparam(*fn=None*)

__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`).

transform_sample(*latent*)

Given latent samples from the warped posterior (with possible batch dimensions), return a *dict* of samples from the latent sites in the model.

Parameters **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: *numpyro.infer.reparam.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: *numpyro.infer.reparam.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: *numpyro.infer.reparam.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).

4.5 Funsor-based NumPyro

See the GitHub repo for more information about Funsor.

4.5.1 Effect handlers

class enum(*fn=None*, *first_available_dim=None*)
Bases: numpyro.contrib.funsor.enum_messenger.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(*fn=None*, *config_fn=None*)
Bases: numpyro.primitives.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(*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: numpyro.contrib.funsor.enum_messenger.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: 0>*)
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: 0>*)
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*.

Return type funsor.terms.Funsor

class trace(*fn=None*)
Bases: *numpyro.handlers.trace*

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()*.

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:

```
model = config_enumerate(model)
```

or as a decorator:

```
@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:

```
@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))):
        states.append(numpyro.sample("states_{}".format(t),
                                     dist.Categorical(transition[states[-1]])))
        numpyro.sample("obs_{}".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:

```
def model(*args, **kwargs):
    ...

log_joint = log_density(enum(config_enumerate(model)), args, kwargs,␣
↪params)
```

- **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:

```
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, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params

Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]

Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._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: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params

Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]

Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._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:

A Method for Stochastic Optimization, Diederik P. Kingma, Jimmy Ba https://arxiv.org/abs/1412.6980

eval_and_stable_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params

Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._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:

```
>>> 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)  # get guide's parameters
>>> quantiles = guide.quantiles(params, 0.5)  # get means of posterior samples
>>> assert_allclose(quantiles["a"], 2., atol=1e-3)
>>> assert_allclose(quantiles["b"], 3., atol=1e-3)
```

eval_and_stable_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params
Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]
Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._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, numpyro.optim._OptState]*)
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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)
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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params
Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]
Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._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

class RMSProp(**args*, ***kwargs*)
Wrapper class for the JAX optimizer: `rmsprop()`

eval_and_stable_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)
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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)
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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params
Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]
Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._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, numpyro.optim._OptState]*)
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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)
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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params
Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]
Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._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, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params

Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]

Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._OptState]

Gradient update for the optimizer.

Parameters

- **g** – gradient information for parameters.
- **state** – current optimizer state.

Returns new optimizer state after the update.

4.6.9 SM3

class SM3(**args*, ***kwargs*)

Wrapper class for the JAX optimizer: `sm3()`

eval_and_stable_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

eval_and_update(*fn: Callable[[Any], Tuple]*, *state: Tuple[int, numpyro.optim._OptState]*)

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.

Returns a pair of the output of objective function and the new optimizer state.

get_params(*state: Tuple[int, numpyro.optim._OptState]*) → numpyro.optim._Params

Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init(*params: numpyro.optim._Params*) → Tuple[int, numpyro.optim._OptState]

Initialize the optimizer with parameters designated to be optimized.

Parameters **params** – a collection of numpy arrays.

Returns initial optimizer state.

update(*g: numpyro.optim._Params*, *state: Tuple[int, numpyro.optim._OptState]*) → Tuple[int, numpyro.optim._OptState]

Gradient update for the optimizer.

Parameters

- **g** – gradient information for parameters.
- **state** – current optimizer state.

Returns new optimizer state after the update.

4.6.10 Optax support

optax_to_numpyro(*transformation*) → numpyro.optim._NumPyroOptim

This function produces a `numpyro.optim._NumPyroOptim` instance from an `optax.GradientTransformation` so that it can be used with `numpyro.infer.svi.SVI`. It is a lightweight wrapper that recreates the `(init_fn, update_fn, get_params_fn)` interface defined by `jax.example_libraries.optimizers`.

Parameters **transformation** – An `optax.GradientTransformation` instance to wrap.

Returns An instance of `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

autocorrelation(*x*, *axis=0*)

Computes the autocorrelation of samples at dimension `axis`.

Parameters

- **x** (*numpy.ndarray*) – the input array.
- **axis** (*int*) – the dimension to calculate autocorrelation.

Returns autocorrelation of `x`.

Return type 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
2. *Stan Reference Manual version 2.18*, Stan Development Team

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.

Note: 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.

Note: This utility only takes effect at the beginning of your program. Under the hood, this sets the environment variable *XLA_FLAGS=–xla_force_host_platform_device_count=[num_devices]*, where *[num_device]* is the desired number of CPU devices *n*.

Warning: Our understanding of the side effects of using the *xla_force_host_platform_device_count* 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.

Parameters **n** (*int*) – number of CPU devices to use.

4.9 Inference Utilities

4.9.1 Predictive

class Predictive(*model: Callable, posterior_samples: Optional[Dict] = None, *, guide: Optional[Callable] = None, params: Optional[Dict] = None, num_samples: Optional[int] = None, return_sites: Optional[List[str]] = None, infer_discrete: bool = False, parallel: bool = False, batch_ndims: Optional[int] = None*)
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*.

Warning: The interface for the *Predictive* class is experimental, and might change in the future.

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 ...)*

Returns dict of samples from the predictive distribution.

Example:

Given a model:

```
def model(X, y=None):
    ...
    return numpyro.sample("obs", likelihood, obs=y)
```

you can sample from the prior predictive:

```
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:

```
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 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

- **transforms** – Dictionary of transforms keyed by names. Names in *transforms* and *params* should align.
- **params** – Dictionary of arrays keyed by names.
- **invert** – Whether to apply the inverse of the transforms.

Returns *dict* of transformed params.

4.9.4 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.

4.9.5 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.

4.9.6 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.

4.9.7 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.

4.9.8 Initialization Strategies

init_to_feasible

init_to_feasible(*site=None*)

Initialize to an arbitrary feasible point, ignoring distribution parameters.

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

init_to_uniform(*site=None*, *radius=2*)
Initialize to a random point in the area *(-radius, radius)* of unconstrained domain.

Parameters **radius** (*float*) – specifies the range to draw an initial point in the unconstrained domain.

init_to_value

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.

4.9.9 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*

```
xij = Vindex(x)[i, :, j]

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:

```
old_batch_shape = x.shape[:-3]
old_event_shape = x.shape[-3:]

xij = Vindex(x)[..., i, :, j]   # The ... denotes unknown batch shape.

new_batch_shape = broadcast_shape(old_batch_shape, i.shape, j.shape)
```

(continues on next page)

(continued from previous page)

```
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:

```
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

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.

4.9.10 Model Inspection

get_dependencies

get_dependencies(*model: Callable*, *model_args: Optional[tuple] = None*, *model_kwargs: Optional[dict] = None*) → Dict[str, object]

Infers dependency structure about a conditioned model.

This returns a nested dictionary with structure like:

```
{
    "prior_dependencies": {
        "variable1": {"variable1": set()},
        "variable2": {"variable1": set(), "variable2": set()},
        ...
    },
    "posterior_dependencies": {
        "variable1": {"variable1": {"plate1"}, "variable2": 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:

```
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": {"a": set(), "b": set()},
    },
    "posterior_dependencies": {
        "a": {"a": set(), "b": 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": {"p"}, "b": set()},
    },
}
```

[1] S.Webb, A.Goliński, R.Zinkov, N.Siddharth, T.Rainforth, Y.W.Teh, F.Wood (2018) "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_kwargs** (*dict*) – Optional dict of model kwargs.

Returns A dictionary of metadata (see above).

Return type dict

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:

```
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:

```
{'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

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** (*str*) – File to save rendered model in.
- **render_distributions** (*bool*) – Whether to include RV distribution annotations in the plot.
- **render_params** (*bool*) – Whether to show params in the plot.

4.10.2 Trace Inspection

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** (*dict*) – 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** (*str*) – Title for the table of shapes.
- **last_site** (*str*) – Name of a site in the model. If supplied, subsequent sites are not displayed in the table.

Usage:

```
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))
```

CHAPTER

FIVE

EFFECT HANDLERS

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.

```
>>> 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␣
↪size 5.83e-01. acc. prob=0.85]
```

(continues on next page)

(continued from previous page)

```
>>> mcmc.print_summary()


                mean       sd       5.5%      94.5%      n_eff       Rhat
    coefs[0]       0.96       0.07       0.85       1.07     455.35       1.01
    coefs[1]       2.05       0.09       1.91       2.20     332.00       1.01
    coefs[2]       3.18       0.13       2.96       3.37     320.27       1.00
   intercept      -0.03       0.02      -0.06       0.00     402.53       1.00

>>> 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'])

>>> 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))

>>> 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*)

Bases: numpyro.primitives.Messenger

Given a callable *fn*, return another callable that selectively hides primitive sites where *hide_fn* returns True from other effect handlers on the stack.

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']*.

Example:

```
>>> from jax import random
>>> import numpyro
>>> from numpyro.handlers import block, seed, trace
>>> import numpyro.distributions as dist

>>> def model():
```

(continues on next page)

(continued from previous page)

```
...     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
```

process_message(*msg*)

5.2 collapse

class collapse(**args*, ***kwargs*)

Bases: *numpyro.handlers.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: `numpyro.primitives.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.))
```

(continues on next page)

(continued from previous page)

```
>>> 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']
```

process_message(*msg*)

5.4 do

class do(*fn=None*, *data=None*)

Bases: numpyro.primitives.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 = numpyro.sample("z", ...)* with *z = 1.* and introducing a fresh sample site *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:

```
>>> 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: numpyro.primitives.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: numpyro.primitives.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:

```
>>> 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: numpyro.primitives.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: numpyro.primitives.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.

[1] Maria I. Gorinova, Dave Moore, Matthew D. Hoffman (2019) "Automatic Reparameterisation of Probabilistic Programs" https://arxiv.org/pdf/1906.03028.pdf

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: numpyro.primitives.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
```

(continues on next page)

(continued from previous page)

```
>>> 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: numpyro.primitives.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='/'*, ***, *hide_types=None*)

Bases: numpyro.primitives.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))
...
>>> 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*)

Bases: numpyro.primitives.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.

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).

Example:

```
>>> 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: numpyro.primitives.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.

Note: This handler is mainly used for internal algorithms. For conditioning a generative model on observed data, please use the *condition* handler.

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:

```
>>> from jax import random
>>> import numpyro
>>> from numpyro.handlers import 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(substitute(model, {'a': -1})).get_trace()
>>> assert exec_trace['a']['value'] == -1
```

process_message(*msg*)

5.14 trace

class trace(*fn=None*)

Bases: numpyro.primitives.Messenger

Returns a handler that records the inputs and outputs at primitive calls inside *fn*.

Example:

```
>>> from jax import random
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.handlers import seed, trace
```

(continues on next page)

(continued from previous page)

```
>>> 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': DeviceArray([0, 0], dtype=uint32)},
               'name': 'a',
               'type': 'sample',
               'value': DeviceArray(-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

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.

Note: To enumerate over a discrete latent variable, you can add the keyword *infer={"enumerate": "parallel"}* to the corresponding *sample* statement.

Note: To improve the performance, please consider enabling x64 mode at the beginning of your NumPyro program `numpyro.enable_x64()`.

References

1. *JAXNS: a high-performance nested sampling package based on JAX*, Joshua G. Albert (https://arxiv.org/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.

Example

```
>>> 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.])
```

(continues on next page)

(continued from previous page)

```
>>> 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 a 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 ELBO-within-Stein 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. This property means Stein mixtures significantly reduce the number of particles needed to represent high dimensional models.

`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*
- *PrecondMatrixKernel*
- *HessianPrecondMatrix*
- *GraphicalKernel*

For example, usage see:

- The Bayesian neural network example

References

1. *Stein's Method Meets Statistics: A Review of Some Recent Developments* (2021) Andreas Anastasiou, Alessandro Barp, François-Xavier Briol, Bruno Ebner, Robert E. Gaunt, Fatemeh Ghaderinezhad, Jackson Gorham, Arthur Gretton, Christophe Ley, Qiang Liu, Lester Mackey, Chris. J. Oates, Gesine Reinert, Yvik Swan. https://arxiv.org/abs/2105.03481

2. *Stein variational gradient descent: A general-purpose Bayesian inference algorithm* (2016) Qiang Liu, Dilin Wang. NeurIPS

3. *Nonlinear Stein Variational Gradient Descent for Learning Diversified Mixture Models* (2019) Dilin Wang, Qiang Liu. PMLR

6.2.1 SteinVI Interface

class SteinVI(*model, guide, optim, loss, kernel_fn: numpyro.contrib.einstein.kernels.SteinKernel, num_particles: int = 10, loss_temperature: float = 1.0, repulsion_temperature: float = 1.0, classic_guide_params_fn: Callable[[str], bool] = <function SteinVI.<lambda>>, enum=True, **static_kwargs*)

Stein variational inference for stein mixtures.

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`.
- **loss** – ELBO loss, i.e. negative Evidence Lower Bound, to minimize.
- **kernel_fn** – Function that produces a logarithm of the statistical kernel to use with Stein inference
- **num_particles** – number of particles for Stein inference. (More particles capture more of the posterior distribution)
- **loss_temperature** – scaling of loss factor
- **repulsion_temperature** – scaling of repulsive forces (Non-linear Stein)
- **enum** – whether to apply automatic marginalization of discrete variables
- **classic_guide_param_fn** – 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 fitting.

6.2.2 SteinVI Kernels

class RBFKernel(*mode='norm', matrix_mode='norm_diag', bandwidth_factor: 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: 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[numpyro.contrib.einstein.kernels.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

class PrecondMatrixKernel(*precond_matrix_fn: numpyro.contrib.einstein.kernels.PrecondMatrix, inner_kernel_fn: numpyro.contrib.einstein.kernels.SteinKernel, precond_mode='anchor_points'*)

Calculates the const preconditioned kernel $k(x, y) = Q^{-\frac{1}{2}} k(Q^{\frac{1}{2}} x, Q^{\frac{1}{2}} y) Q^{-\frac{1}{2}}$, or anchor point preconditioned kernel $k(x, y) = \sum_{l=1}^{m} k_{Q_l}(x, y) w_l(x) w_l(y)$ both from [1].

References:

1. "Stein Variational Gradient Descent with Matrix-Valued Kernels" by Wang, Tang, Bajaj and Liu

Parameters

- **precond_matrix_fn** – The constant preconditioning matrix
- **inner_kernel_fn** – The inner kernel function
- **precond_mode** – How to use the precondition matrix, either constant ('const') or as mixture with anchor points ('anchor_points')

class GraphicalKernel(*mode='matrix', local_kernel_fns: Optional[Dict[str, numpyro.contrib.einstein.kernels.SteinKernel]] = None, default_kernel_fn: numpyro.contrib.einstein.kernels.SteinKernel = <numpyro.contrib.einstein.kernels.RBFKernel object>*)

Calculates graphical kernel $k(x, y) = diag(K_l(x_l, y_l))$ for local kernels K_l from [1][2].

References:

1. *Stein Variational Message Passing for Continuous Graphical Models* by Wang, Zheng, and Liu

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

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]:
```
!pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
```

[2]:
```
import os

from IPython.display import set_matplotlib_formats
import jax.numpy as jnp
from jax import random, vmap
from jax.scipy.special import logsumexp
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

import numpyro
from numpyro.diagnostics import hpdi
import numpyro.distributions as dist
from numpyro import handlers
from numpyro.infer import MCMC, NUTS

plt.style.use("bmh")
if "NUMPYRO_SPHINXBUILD" in os.environ:
    set_matplotlib_formats("svg")

assert numpyro.__version__.startswith("0.10.1")
```

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=";")
dset
```

[3]:

	Location	Loc	...	Population1860	PropSlaves1860
0	Alabama	AL	...	964201	0.450000
1	Alaska	AK	...	0	0.000000
2	Arizona	AZ	...	0	0.000000
3	Arkansas	AR	...	435450	0.260000
4	California	CA	...	379994	0.000000
5	Colorado	CO	...	34277	0.000000
6	Connecticut	CT	...	460147	0.000000
7	Delaware	DE	...	112216	0.016000
8	District of Columbia	DC	...	75080	0.000000
9	Florida	FL	...	140424	0.440000
10	Georgia	GA	...	1057286	0.440000
11	Hawaii	HI	...	0	0.000000
12	Idaho	ID	...	0	0.000000
13	Illinois	IL	...	1711951	0.000000
14	Indiana	IN	...	1350428	0.000000
15	Iowa	IA	...	674913	0.000000

(continues on next page)

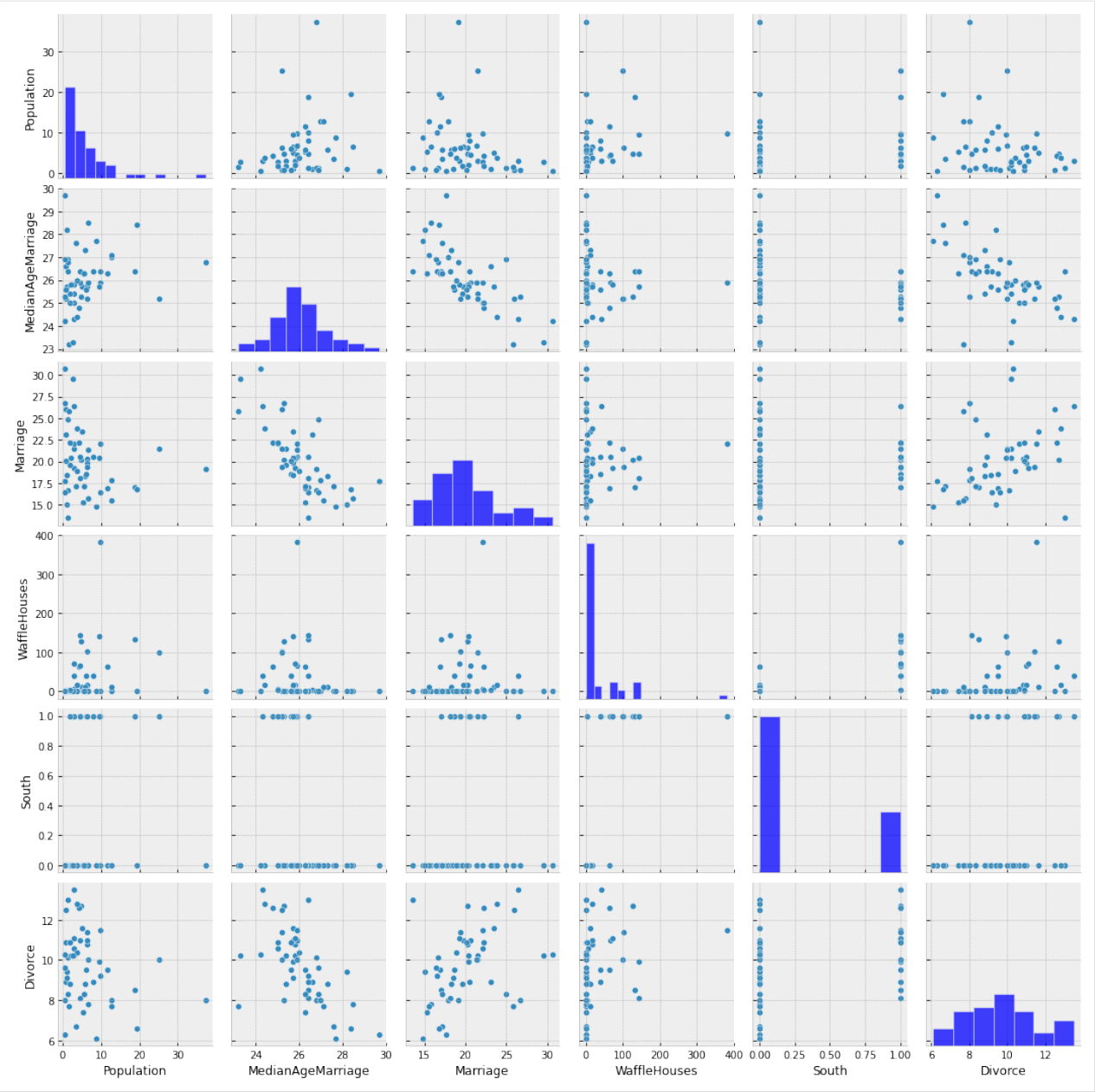
(continued from previous page)

```
16                Kansas  KS  ...          107206        0.000019
17              Kentucky  KY  ...         1155684        0.000000
18             Louisiana  LA  ...          708002        0.470000
19                 Maine  ME  ...          628279        0.000000
20              Maryland  MD  ...          687049        0.130000
21         Massachusetts  MA  ...         1231066        0.000000
22              Michigan  MI  ...          749113        0.000000
23             Minnesota  MN  ...          172023        0.000000
24           Mississippi  MS  ...          791305        0.550000
25              Missouri  MO  ...         1182012        0.097000
26               Montana  MT  ...               0        0.000000
27              Nebraska  NE  ...           28841        0.000520
28         New Hampshire  NH  ...          326073        0.000000
29            New Jersey  NJ  ...          672035        0.000027
30            New Mexico  NM  ...           93516        0.000000
31              New York  NY  ...         3880735        0.000000
32        North Carolina  NC  ...          992622        0.330000
33          North Dakota  ND  ...               0        0.000000
34                  Ohio  OH  ...         2339511        0.000000
35              Oklahoma  OK  ...               0        0.000000
36                Oregon  OR  ...           52465        0.000000
37          Pennsylvania  PA  ...         2906215        0.000000
38          Rhode Island  RI  ...          174620        0.000000
39        South Carolina  SC  ...          703708        0.570000
40          South Dakota  SD  ...            4837        0.000000
41             Tennessee  TN  ...         1109801        0.200000
42                 Texas  TX  ...          604215        0.300000
43                  Utah  UT  ...           40273        0.000000
44               Vermont  VT  ...          315098        0.000000
45              Virginia  VA  ...         1219630        0.400000
46            Washington  WA  ...           11594        0.000000
47         West Virginia  WV  ...          376688        0.049000
48             Wisconsin  WI  ...          775881        0.000000
49               Wyoming  WY  ...               0        0.000000

[50 rows x 13 columns]
```

Let us plot the pair-wise relationship amongst the main variables in the dataset, using `seaborn.pairplot`.

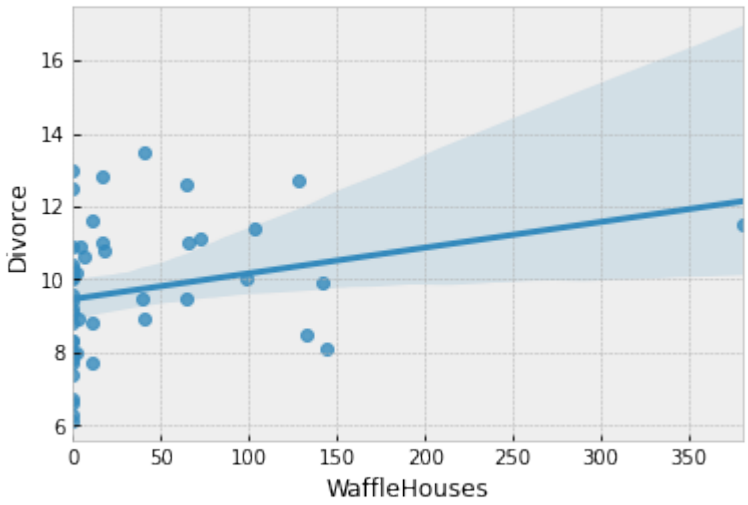
[4]:
```
vars = [
    "Population",
    "MedianAgeMarriage",
    "Marriage",
    "WaffleHouses",
    "South",
    "Divorce",
]
sns.pairplot(dset, x_vars=vars, y_vars=vars, palette="husl");
```



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 regressionn 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]:
```
standardize = lambda x: (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.
- 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.

[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.

[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
mcmc = MCMC(kernel, num_warmup=1000, num_samples=num_samples)
mcmc.run(
```

(continues on next page)

(continued from previous page)

```
    rng_key_, marriage=dset.MarriageScaled.values, divorce=dset.DivorceScaled.values
)
mcmc.print_summary()
samples_1 = mcmc.get_samples()
```

```
sample: 100%|| 3000/3000 [00:04<00:00, 748.14it/s, 7 steps of size 7.41e-01. acc. prob=0.92]

                mean       std    median      5.0%     95.0%     n_eff     r_hat
         a      0.00      0.11      0.00     -0.16      0.20   1510.96      1.00
        bM      0.35      0.13      0.35      0.14      0.57   2043.12      1.00
     sigma      0.95      0.10      0.94      0.78      1.10   1565.40      1.00

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 hpdi 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.

[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
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 6))
    ax.plot(marriage, mean)
```

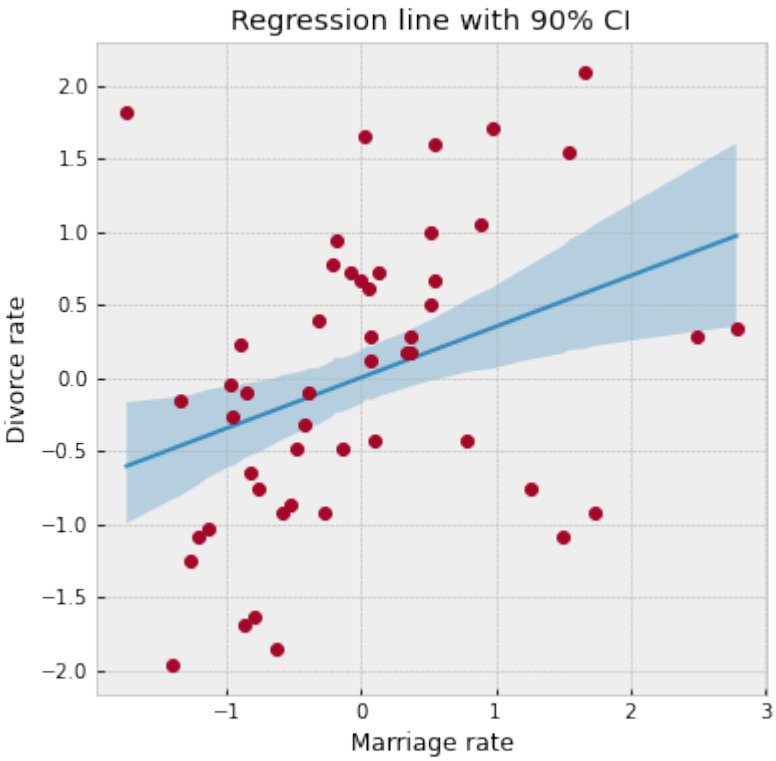
(continues on next page)

(continued from previous page)

```
    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.

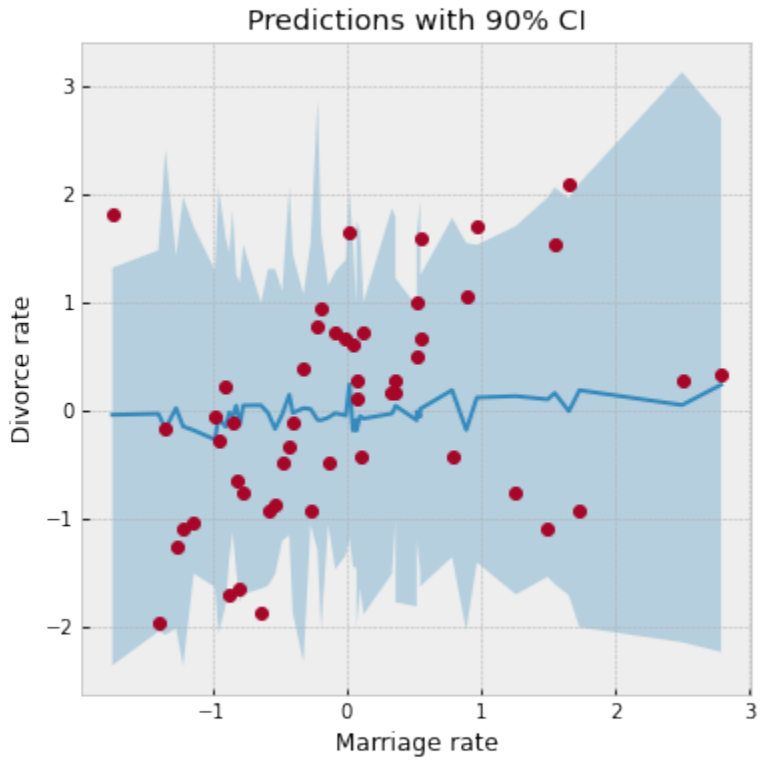
Prior Predictive Distribution

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.

[10]:
```
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.

[11]:
```
rng_key, rng_key_ = random.split(rng_key)
predictive = Predictive(model, samples_1)
```

(continues on next page)

(continued from previous page)

```
predictions = predictive(rng_key_, marriage=dset.MarriageScaled.values)["obs"]
df = dset.filter(["Location"])
df["Mean Predictions"] = jnp.mean(predictions, axis=0)
df.head()
```

[11]:

	Location	Mean Predictions
0	Alabama	0.016434
1	Alaska	0.501293
2	Arizona	0.025105
3	Arkansas	0.600058
4	California	-0.082887

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.

[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 sample sites 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.

[13]:
```
# Using the same key as we used for Predictive - note that the results are identical.
```

(continues on next page)

(continued from previous page)

```
predictions_1 = predict_fn(random.split(rng_key_, num_samples), samples_1)

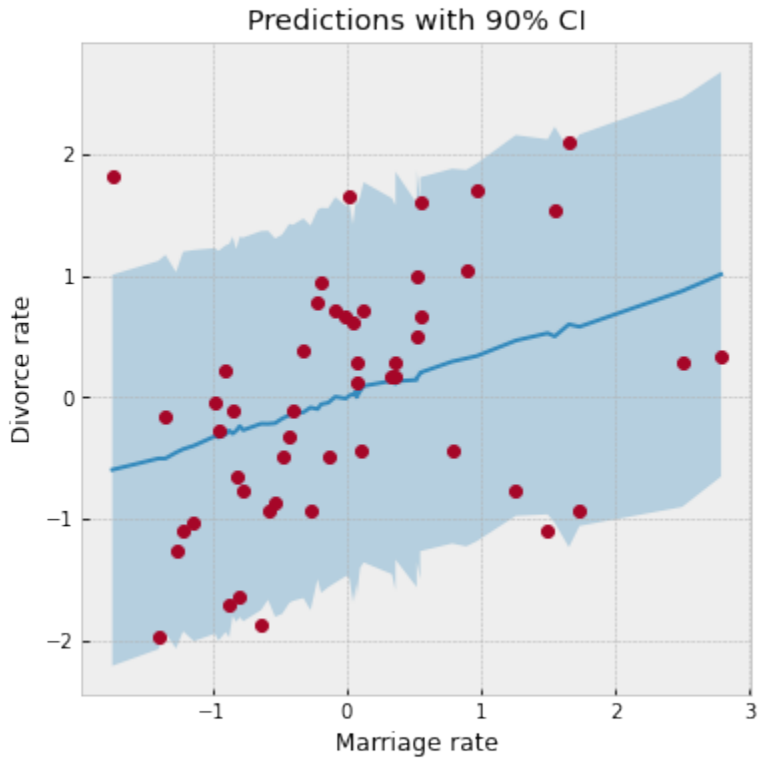
mean_pred = jnp.mean(predictions_1, axis=0)
df = dset.filter(["Location"])
df["Mean Predictions"] = mean_pred
df.head()
```

[13]:

	Location	Mean Predictions
0	Alabama	0.016434
1	Alaska	0.501293
2	Arizona	0.025105
3	Arkansas	0.600058
4	California	-0.082887

[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

$$log \prod_{i=1}^{n} \int p(y_i|\theta) p_{post}(\theta) d\theta \approx \sum_{i=1}^{n} log \frac{\sum_s p(\theta^s)}{S}$$
$$= \sum_{i=1}^{n} (log \sum_s p(\theta^s) - log(S))$$

.

Here, i indexes the observed data points y and s indexes the posterior samples over the latent parameters θ. 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.

[17]:
```
rng_key, rng_key_ = random.split(rng_key)

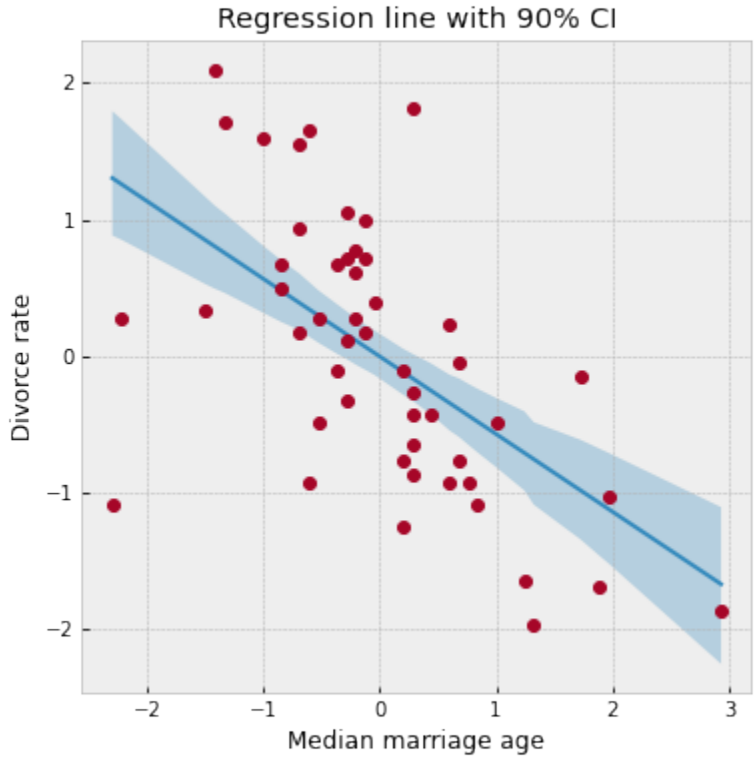
mcmc.run(rng_key_, age=dset.AgeScaled.values, divorce=dset.DivorceScaled.values)
mcmc.print_summary()
samples_2 = mcmc.get_samples()
```

```
sample: 100%|██████████| 3000/3000 [00:04<00:00, 722.23it/s, 7 steps of size 7.58e-01. acc. prob=0.92]

                mean       std    median      5.0%     95.0%     n_eff     r_hat
         a     -0.00      0.10     -0.00     -0.17      0.16   1942.82      1.00
        bA     -0.57      0.12     -0.57     -0.75     -0.38   1995.70      1.00
     sigma      0.82      0.08      0.82      0.69      0.96   1865.82      1.00

Number of divergences: 0
```

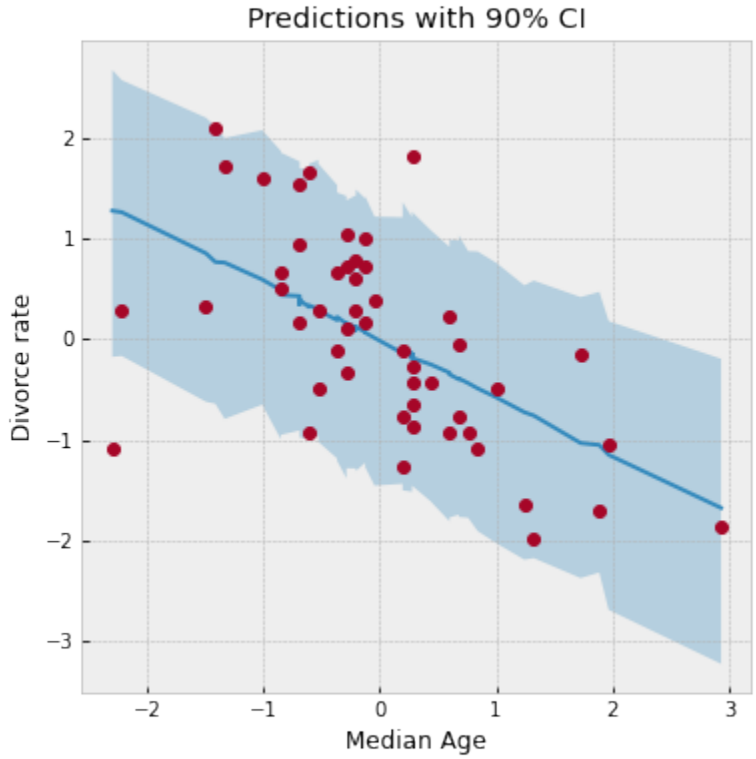
[18]:
```
posterior_mu = (
    jnp.expand_dims(samples_2["a"], -1)
    + jnp.expand_dims(samples_2["bA"], -1) * dset.AgeScaled.values
)
mean_mu = jnp.mean(posterior_mu, axis=0)
hpdi_mu = hpdi(posterior_mu, 0.9)
ax = plot_regression(dset.AgeScaled.values, mean_mu, hpdi_mu)
ax.set(
    xlabel="Median marriage age",
    ylabel="Divorce rate",
    title="Regression line with 90% CI",
);
```



[19]:
```
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");
```



[20]:
```
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
```

7.3.3 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.

[21]:
```
rng_key, rng_key_ = random.split(rng_key)

mcmc.run(
    rng_key_,
    marriage=dset.MarriageScaled.values,
```

(continues on next page)

(continued from previous page)

```
    age=dset.AgeScaled.values,
    divorce=dset.DivorceScaled.values,
)
mcmc.print_summary()
samples_3 = mcmc.get_samples()
```

```
sample: 100%|| 3000/3000 [00:04<00:00, 644.48it/s, 7 steps of size 4.65e-01. acc. prob=0.94]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
         a      0.00      0.10      0.00     -0.17      0.16   2007.41      1.00
        bA     -0.61      0.16     -0.61     -0.89     -0.37   1225.02      1.00
        bM     -0.07      0.16     -0.07     -0.34      0.19   1275.37      1.00
     sigma      0.83      0.08      0.82      0.69      0.96   1820.77      1.00

Number of divergences: 0
```

[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.

[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)

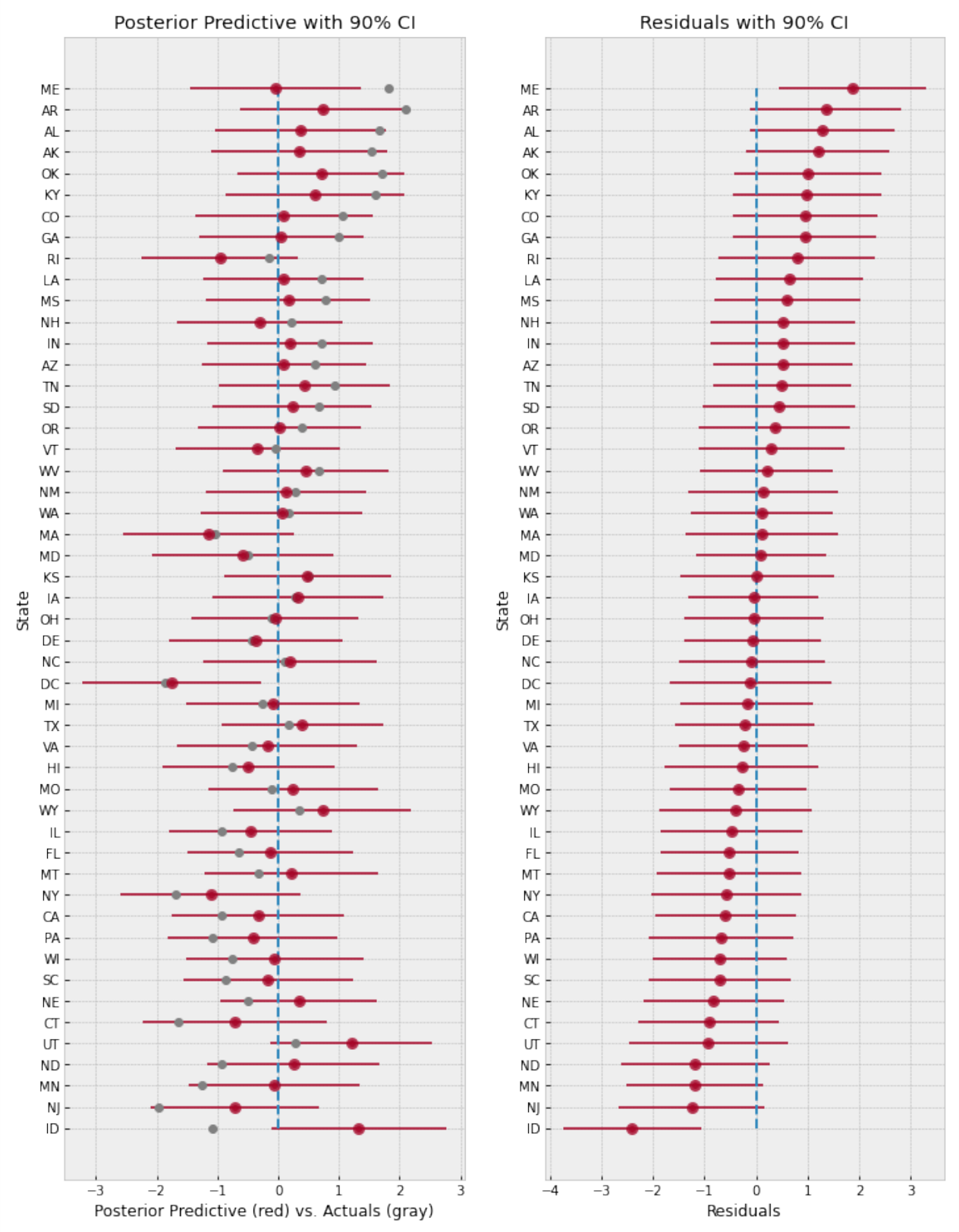
(continued from previous page)

```
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).

[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]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
a	-0.06	0.10	-0.06	-0.20	0.11	3203.01	1.00
bA	-0.61	0.16	-0.61	-0.87	-0.35	2156.51	1.00
bM	0.06	0.17	0.06	-0.21	0.33	1943.15	1.00
divorce_rate[0]	1.16	0.36	1.15	0.53	1.72	2488.98	1.00
divorce_rate[1]	0.69	0.55	0.68	-0.15	1.65	4832.63	1.00
divorce_rate[2]	0.42	0.34	0.42	-0.16	0.96	4419.13	1.00
divorce_rate[3]	1.41	0.46	1.40	0.63	2.11	4782.86	1.00
divorce_rate[4]	-0.90	0.13	-0.90	-1.12	-0.71	4269.33	1.00
divorce_rate[5]	0.65	0.39	0.65	0.01	1.31	4139.51	1.00
divorce_rate[6]	-1.36	0.35	-1.36	-1.96	-0.82	5180.21	1.00
divorce_rate[7]	-0.33	0.49	-0.33	-1.15	0.45	4089.39	1.00
divorce_rate[8]	-1.88	0.59	-1.88	-2.89	-0.93	3305.68	1.00
divorce_rate[9]	-0.62	0.17	-0.61	-0.90	-0.34	4936.95	1.00
divorce_rate[10]	0.76	0.29	0.76	0.28	1.24	3627.89	1.00
divorce_rate[11]	-0.55	0.50	-0.55	-1.38	0.26	3822.80	1.00
divorce_rate[12]	0.20	0.53	0.20	-0.74	0.99	1476.70	1.00
divorce_rate[13]	-0.86	0.23	-0.87	-1.24	-0.48	5333.10	1.00
divorce_rate[14]	0.55	0.30	0.55	0.09	1.05	5533.56	1.00
divorce_rate[15]	0.28	0.38	0.28	-0.35	0.92	5179.68	1.00
divorce_rate[16]	0.49	0.43	0.49	-0.23	1.16	5134.56	1.00
divorce_rate[17]	1.25	0.35	1.24	0.69	1.84	4571.21	1.00
divorce_rate[18]	0.42	0.38	0.41	-0.15	1.10	4946.86	1.00
divorce_rate[19]	0.38	0.55	0.36	-0.50	1.29	2145.11	1.00
divorce_rate[20]	-0.56	0.34	-0.56	-1.12	-0.02	5219.59	1.00
divorce_rate[21]	-1.11	0.27	-1.11	-1.53	-0.65	3778.88	1.00
divorce_rate[22]	-0.28	0.26	-0.28	-0.71	0.13	5751.65	1.00
divorce_rate[23]	-0.99	0.30	-0.99	-1.46	-0.49	4385.57	1.00
divorce_rate[24]	0.43	0.41	0.42	-0.26	1.08	3868.84	1.00
divorce_rate[25]	-0.03	0.32	-0.03	-0.57	0.48	5927.41	1.00
divorce_rate[26]	-0.01	0.49	-0.01	-0.79	0.81	4581.29	1.00
divorce_rate[27]	-0.16	0.39	-0.15	-0.79	0.49	4522.45	1.00
divorce_rate[28]	-0.27	0.50	-0.29	-1.08	0.53	3824.97	1.00
divorce_rate[29]	-1.79	0.24	-1.78	-2.18	-1.39	5134.14	1.00
divorce_rate[30]	0.17	0.42	0.16	-0.55	0.82	5978.21	1.00
divorce_rate[31]	-1.66	0.16	-1.66	-1.93	-1.41	5976.18	1.00
divorce_rate[32]	0.12	0.25	0.12	-0.27	0.52	5759.99	1.00
divorce_rate[33]	-0.04	0.52	-0.04	-0.91	0.82	2926.68	1.00
divorce_rate[34]	-0.13	0.22	-0.13	-0.50	0.23	4390.05	1.00
divorce_rate[35]	1.27	0.43	1.27	0.53	1.94	4659.54	1.00
divorce_rate[36]	0.22	0.36	0.22	-0.36	0.84	3758.16	1.00
divorce_rate[37]	-1.02	0.23	-1.02	-1.38	-0.64	5954.84	1.00
divorce_rate[38]	-0.93	0.54	-0.94	-1.84	-0.06	3289.66	1.00
divorce_rate[39]	-0.67	0.33	-0.67	-1.18	-0.09	4787.55	1.00
divorce_rate[40]	0.25	0.55	0.24	-0.67	1.16	4526.98	1.00
divorce_rate[41]	0.73	0.34	0.73	0.17	1.29	4237.28	1.00
divorce_rate[42]	0.20	0.18	0.20	-0.10	0.48	5156.91	1.00
divorce_rate[43]	0.81	0.43	0.81	0.14	1.50	2067.24	1.00
divorce_rate[44]	-0.42	0.51	-0.43	-1.23	0.45	3844.29	1.00
divorce_rate[45]	-0.39	0.25	-0.39	-0.78	0.04	4611.94	1.00
divorce_rate[46]	0.13	0.31	0.13	-0.36	0.64	5879.70	1.00
divorce_rate[47]	0.56	0.47	0.56	-0.15	1.37	4319.38	1.00

(continues on next page)

(continued from previous page)

```
divorce_rate[48]     -0.63      0.28     -0.63     -1.11     -0.18   5820.05      1.00
divorce_rate[49]      0.86      0.59      0.88     -0.10      1.79   2460.53      1.00
           sigma      0.58      0.11      0.57      0.40      0.76    735.02      1.00

Number of divergences: 0
```

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.

[27]:
```
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"]
```

[28]:
```
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
)

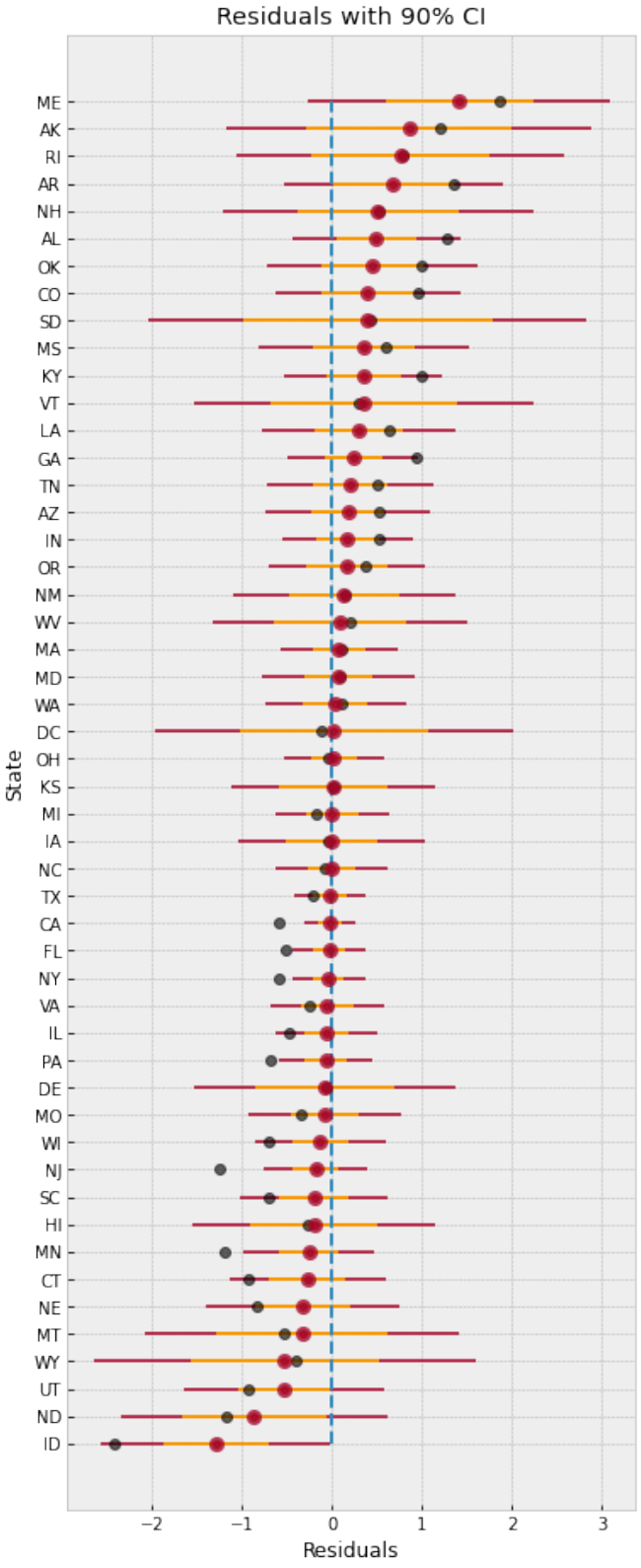
# Plot SD
ax.errorbar(residuals_mean[idx], y, xerr=sd[idx], ls="none", color="orange", alpha=0.9)

# 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,
)
```

(continues on next page)

(continued from previous page)

```
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 \nshows residuals from the previous model (Model 3). "
    "Measurement \nerror is indicated by orange bar.",
);
```



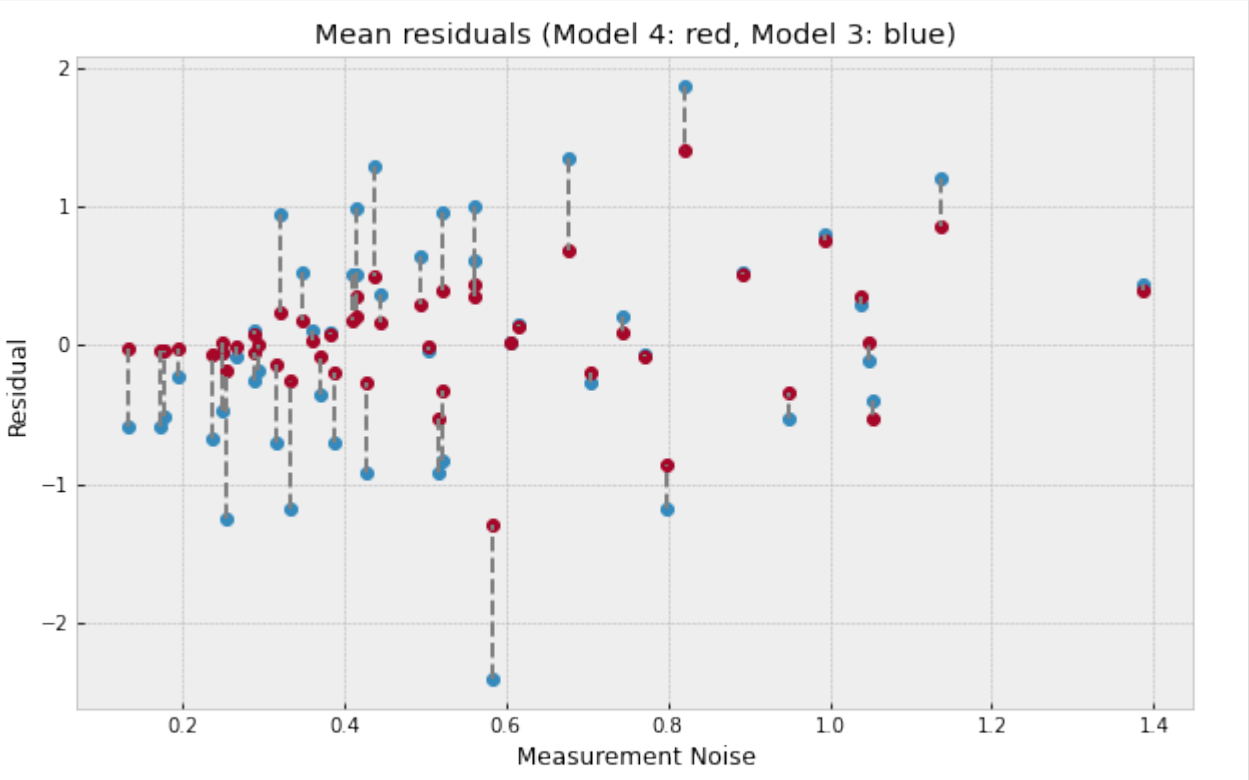
Residuals (with error-bars) from current model (in red). Black marker shows residuals from the previous model (Model 3). Measurement error is indicated by orange bar.

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.

[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 happend 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

1. McElreath, R. (2016). Statistical Rethinking: A Bayesian Course with Examples in R and Stan CRC Press.
2. Stan Development Team. Stan User's Guide
3. Goodman, N.D., and StuhlMueller, A. (2014). The Design and Implementation of Probabilistic Programming Languages
4. Pyro Development Team. Poutine: A Guide to Programming with Effect Handlers in Pyro
5. Hoffman, M.D., Gelman, A. (2011). The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo.
6. Betancourt, M. (2017). A Conceptual Introduction to Hamiltonian Monte Carlo.
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

CHAPTER

EIGHT

BAYESIAN HIERARCHICAL LINEAR REGRESSION

Author: Carlos Souza

Updated by: Chris Stoafer

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:

[1]:
```
!pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro arviz
```

[2]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

[3]:
```
train = pd.read_csv(
    "https://gist.githubusercontent.com/ucals/"
    "2cf9d101992cb1b78c2cdd6e3bac6a4b/raw/"
    "43034c39052dcf97d4b894d2ec1bc3f90f3623d9/"
    "osic_pulmonary_fibrosis.csv"
)
train.head()
```

[3]:

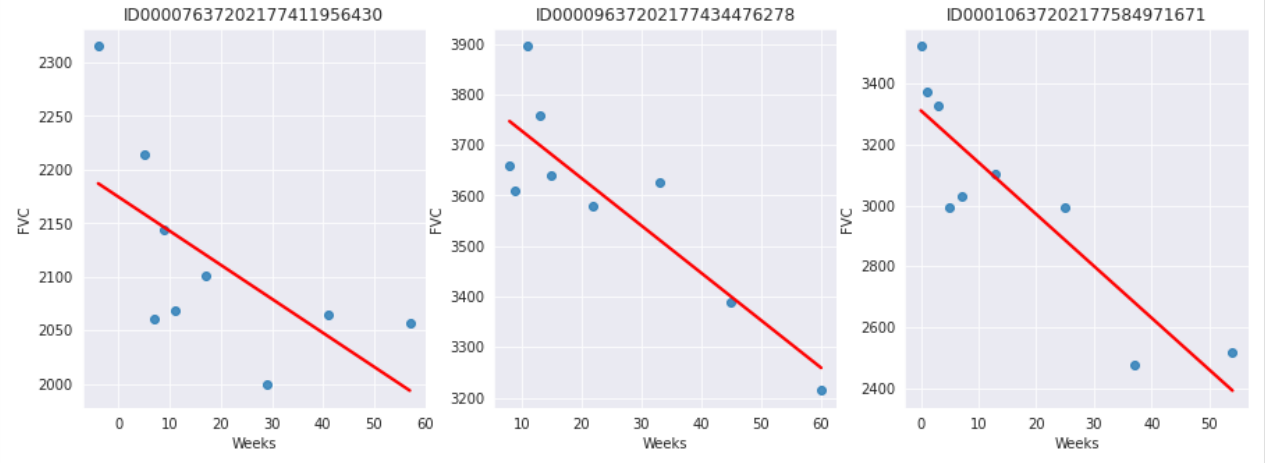
	Patient	Weeks	FVC	Percent	Age	Sex	SmokingStatus
0	ID00007637202177411956430	-4	2315	58.253649	79	Male	Ex-smoker
1	ID00007637202177411956430	5	2214	55.712129	79	Male	Ex-smoker
2	ID00007637202177411956430	7	2061	51.862104	79	Male	Ex-smoker
3	ID00007637202177411956430	9	2144	53.950679	79	Male	Ex-smoker
4	ID00007637202177411956430	11	2069	52.063412	79	Male	Ex-smoker

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:

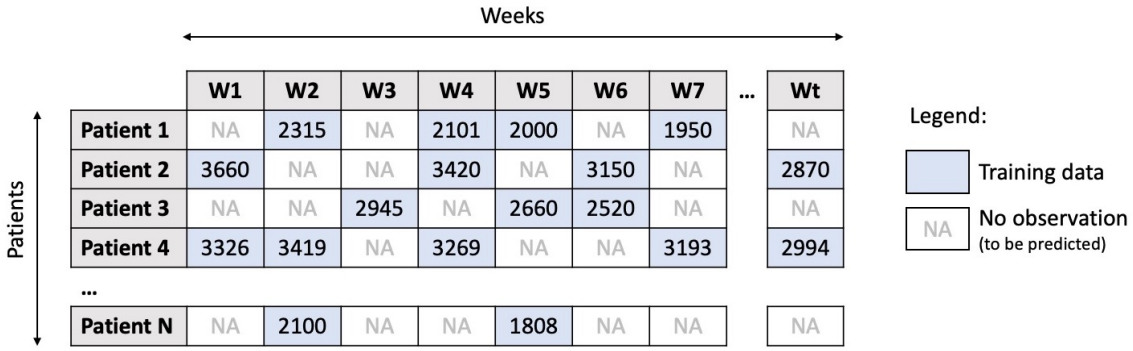
[4]:
```
def chart_patient(patient_id, ax):
    data = train[train["Patient"] == patient_id]
    x = data["Weeks"]
    y = data["FVC"]
    ax.set_title(patient_id)
    sns.regplot(x=x, y=y, ax=ax, ci=None, line_kws={"color": "red"})


f, axes = plt.subplots(1, 3, figsize=(15, 5))
chart_patient("ID00007637202177411956430", axes[0])
chart_patient("ID00009637202177434476278", axes[1])
chart_patient("ID00010637202177584971671", axes[2])
```



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:

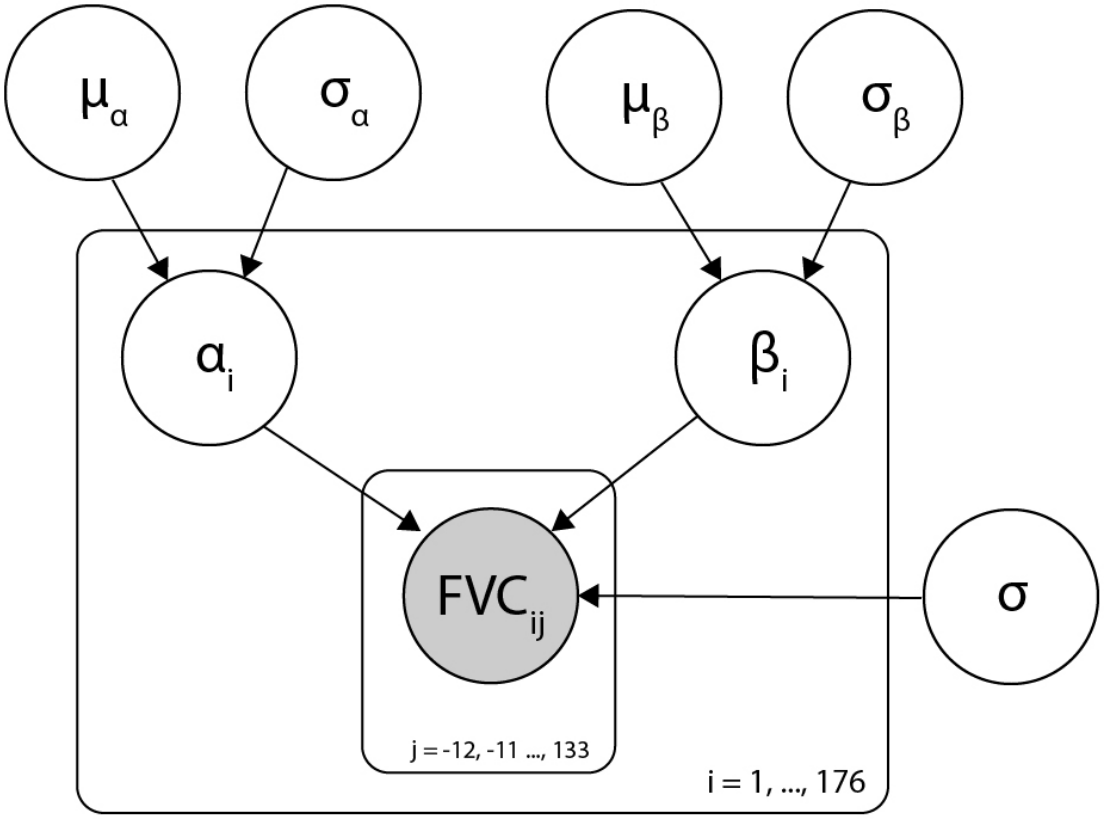


The task was perfect to apply Bayesian inference. However, the vast majority of solutions shared by Kaggle community used discriminative machine learning models, disconsidering the fact that most discriminative methods are very poor at providing realistic uncertainty estimates. Because they are typically 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 α and β. 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 α's and β'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:

$$\mu_\alpha \sim \mathcal{N}(0, 500) \tag{8.1}$$
$$\sigma_\alpha \sim |\mathcal{N}(0, 100)| \tag{8.2}$$
$$\mu_\beta \sim \mathcal{N}(0, 3) \tag{8.3}$$
$$\sigma_\beta \sim |\mathcal{N}(0, 3)| \tag{8.4}$$
$$\alpha_i \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha) \tag{8.5}$$
$$\beta_i \sim \mathcal{N}(\mu_\beta, \sigma_\beta) \tag{8.6}$$
$$\sigma \sim \mathcal{N}(0, 100) \tag{8.7}$$
$$FVC_{ij} \sim \mathcal{N}(\alpha_i + t\beta_i, \sigma) \tag{8.8}$$

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:

[5]:
```
import numpyro
from numpyro.infer import MCMC, NUTS, Predictive
import numpyro.distributions as dist
from jax import random

assert numpyro.__version__.startswith("0.10.1")
```

[6]:
```
def model(patient_code, Weeks, FVC_obs=None):
    μ_α = numpyro.sample("μ_α", dist.Normal(0.0, 500.0))
```

(continues on next page)

(continued from previous page)

```
    σ_α = numpyro.sample("σ_α", dist.HalfNormal(100.0))
    μ_β = numpyro.sample("μ_β", dist.Normal(0.0, 3.0))
    σ_β = numpyro.sample("σ_β", dist.HalfNormal(3.0))

    n_patients = len(np.unique(patient_code))

    with numpyro.plate("plate_i", n_patients):
        α = numpyro.sample("α", dist.Normal(μ_α, σ_α))
        β = numpyro.sample("β", dist.Normal(μ_β, σ_β))

    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)
```

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:

[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:

[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()
```

```
sample: 100
↪%|████████████████████████████████████████| 4000/
↪4000 [00:51<00:00, 77.93it/s, 255 steps of size 1.48e-02. acc. prob=0.92]
```

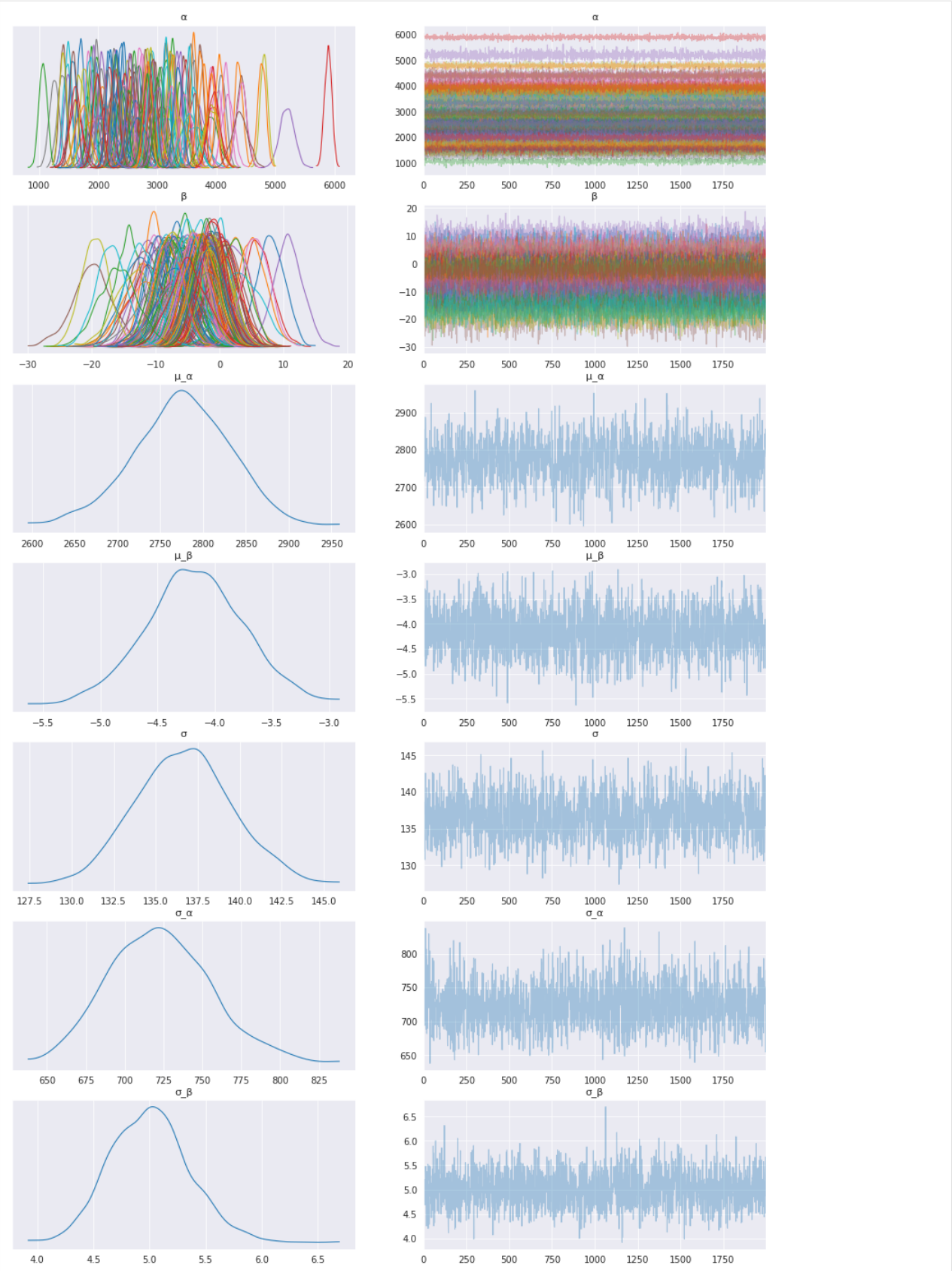
8.4 4. Checking the model

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:

[9]:
```
import arviz as az

data = az.from_numpyro(mcmc)
az.plot_trace(data, compact=True, figsize=(15, 25));
```


α
β
μ_α
μ_β
σ
σ_α
σ_β

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:

[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)
```

[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:

[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:

[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()
```

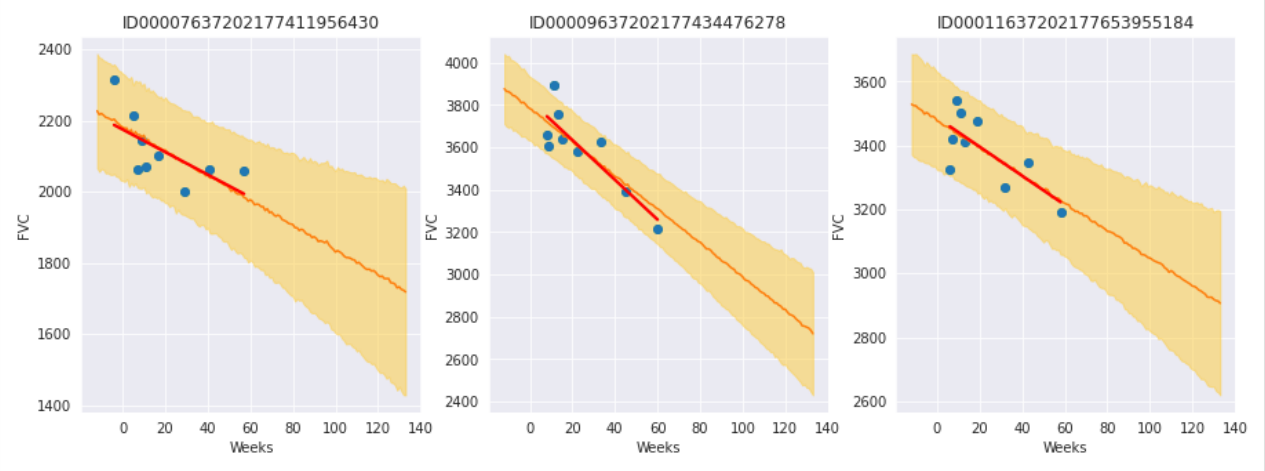
[13]:

	Patient	patient_code	Weeks	FVC_pred	sigma	FVC_inf	FVC_sup	FVC_true
0	ID00007637202177411956430	0	-12	2226.545166	160.158493	2066.386719	2386.703613	NaN
1	ID00007637202177411956430	0	-11	2216.172852	160.390778	2055.781982	2376.563721	NaN
2	ID00007637202177411956430	0	-10	2219.136963	155.339615	2063.797363	2374.476562	NaN
3	ID00007637202177411956430	0	-9	2214.727051	153.333313	2061.393799	2368.060303	NaN
4	ID00007637202177411956430	0	-8	2208.758545	157.368637	2051.389893	2366.127197	NaN

Finally, let's see our predictions for 3 patients:

[14]:
```
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 σ). The

metric was computed as:

$$\sigma_{clipped} = max(\sigma, 70) \tag{8.9}$$

$$\delta = min(|FVC_{true} - FVC_{pred}|, 1000) \tag{8.10}$$

$$metric = -\frac{\sqrt{2}\delta}{\sigma_{clipped}} - \ln(\sqrt{2}\sigma_{clipped}) \tag{8.11}$$

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:

[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

[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.

[21]:
```
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
)
```

[22]:
```
nuts_kernel = NUTS(reparam_model_smoking_hierarchy, target_accept_prob=0.97)

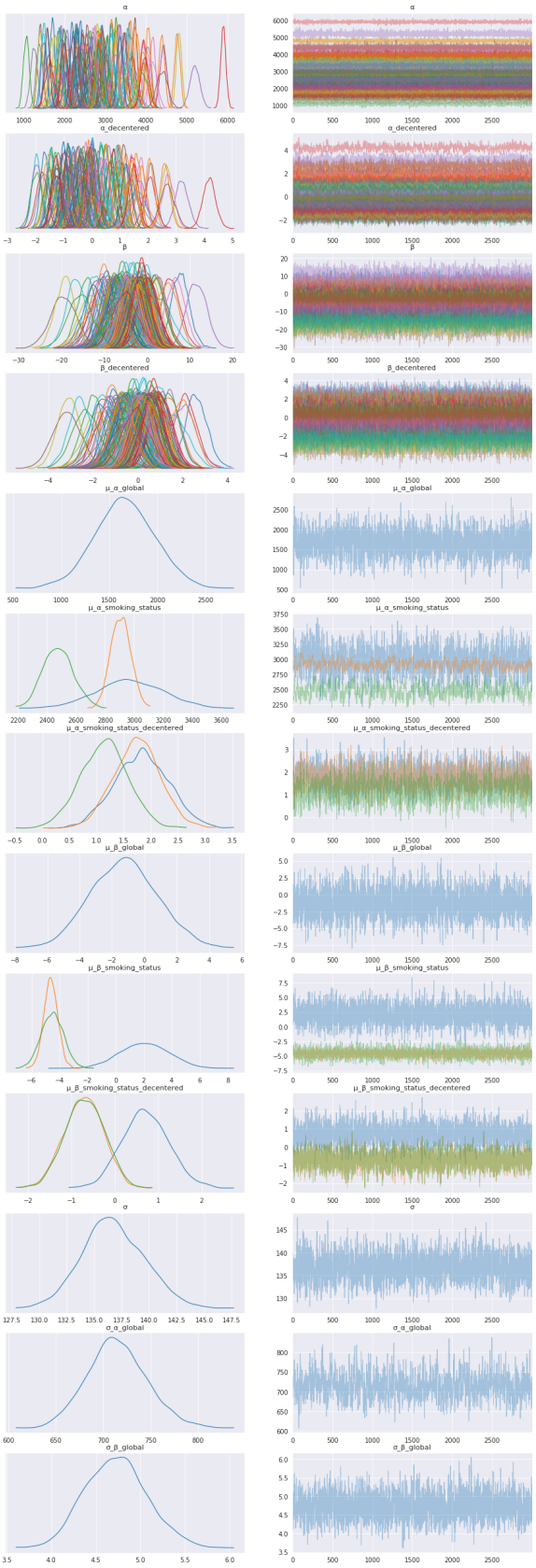
mcmc = MCMC(nuts_kernel, num_samples=3000, num_warmup=5000)
rng_key = random.PRNGKey(0)
mcmc.run(rng_key, patient_code, Weeks, map_patient_to_smoking_status, FVC_obs=FVC_obs)

posterior_samples = mcmc.get_samples()
```

```
sample: 100
↪%|██████████████████████████████████████████████████████████████████████| 8000/
↪8000 [03:55<00:00, 33.99it/s, 1023 steps of size 5.68e-03. acc. prob=0.97]
```

8.5.2 5.1. Inspect the learned parameters

[23]:
```
data = az.from_numpyro(mcmc)
az.plot_trace(data, compact=True, figsize=(15, 45));
```


α
α_decentered
β
β_decentered
μ_α_global
μ_α_smoking_status
μ_α_smoking_status_decentered
μ_β_global
μ_β_smoking_status
μ_β_smoking_status_decentered
σ
σ_α_global
σ_β_global

Smoking Status distributions

Adding a legend for the smoking status distributions to help interpret the model results for that level.

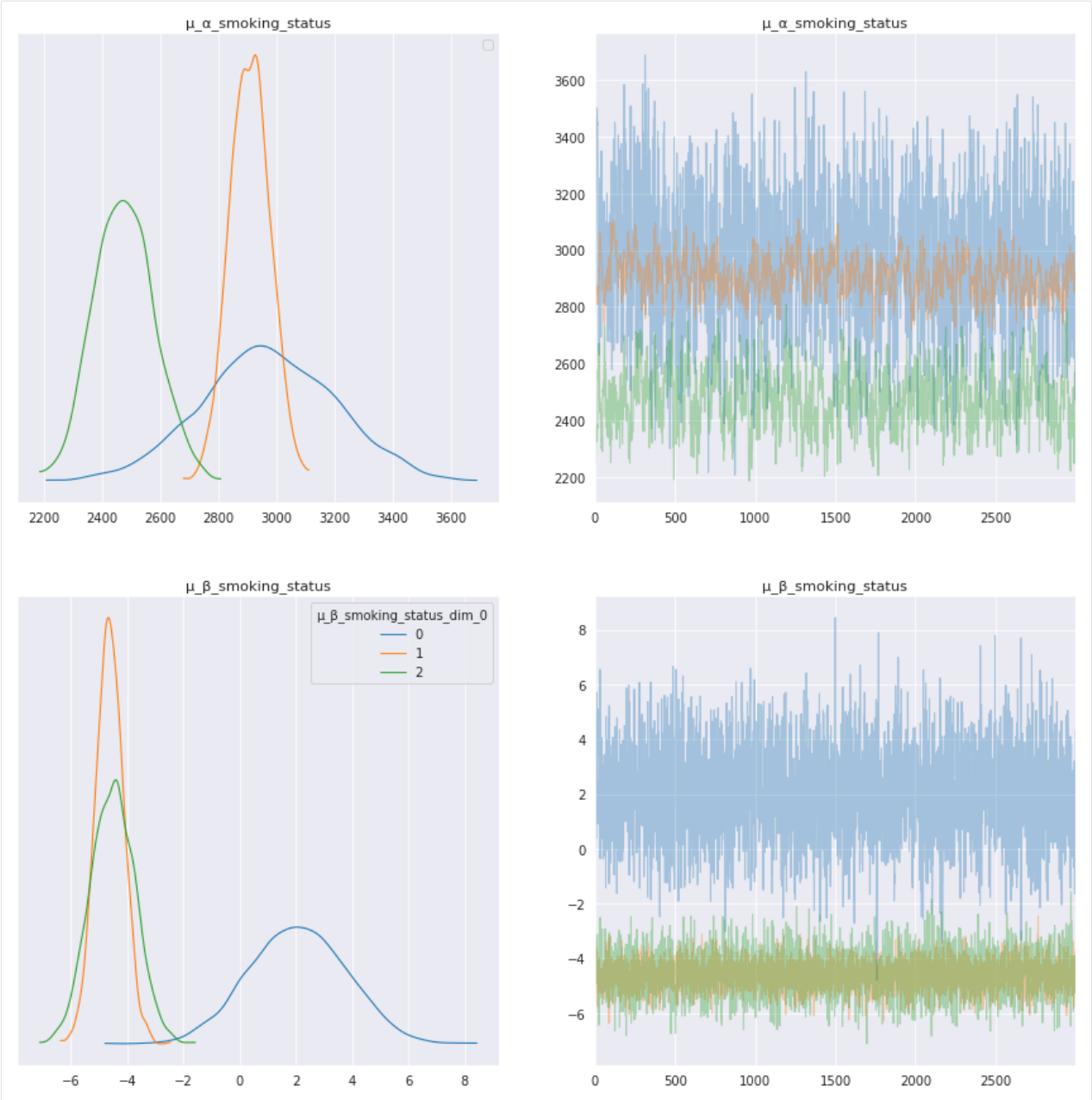
Smoking Status	Code
Currently smokes	0
Ex-smoker	1
Never smoked	2

[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, μ_β_smoking_status. In the trace plots above and summary table below the trend for current smokers, μ_β_smoking_status[0], has a positive mean, whereas the trend for patients that are ex-smokers and those that have never smoked are negative, μ_β_smoking_status[1] and μ_β_smoking_status[2].

[26]:
```
trace = az.from_numpyro(mcmc)
az.summary(
    trace,
    var_names=["μ_α_global", "μ_β_global", "μ_α_smoking_status", "μ_β_smoking_status"],
```

(continues on next page)

(continued from previous page)

```
)
```

```
Shape validation failed: input_shape: (1, 3000), minimum_shape: (chains=2, draws=4)
```

[26]:

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
μ_α_global	1660.172	309.657	1118.038	2274.933	6.589	4.660	2203.0	2086.0	NaN
μ_β_global	-1.252	2.062	-5.014	2.678	0.037	0.035	3040.0	2041.0	NaN
μ_α_smoking_status[0]	2970.486	227.761	2572.943	3429.343	7.674	5.452	878.0	1416.0	NaN
μ_α_smoking_status[1]	2907.950	68.011	2782.993	3035.172	5.209	3.698	171.0	281.0	NaN
μ_α_smoking_status[2]	2475.281	102.948	2286.072	2671.298	6.181	4.381	278.0	566.0	NaN
μ_β_smoking_status[0]	2.061	1.713	-1.278	5.072	0.032	0.024	2797.0	2268.0	NaN
μ_β_smoking_status[1]	-4.625	0.498	-5.566	-3.721	0.010	0.007	2309.0	2346.0	NaN
μ_β_smoking_status[2]	-4.513	0.789	-6.011	-3.056	0.016	0.011	2466.0	2494.0	NaN

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

[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()
```

[28]:

	Patient	patient_code	Weeks	FVC_pred	sigma
0	ID00007637202177411956430	0	-12	2229.098877	157.880753
1	ID00007637202177411956430	0	-11	2225.022461	157.358429
2	ID00007637202177411956430	0	-10	2224.487549	155.416016

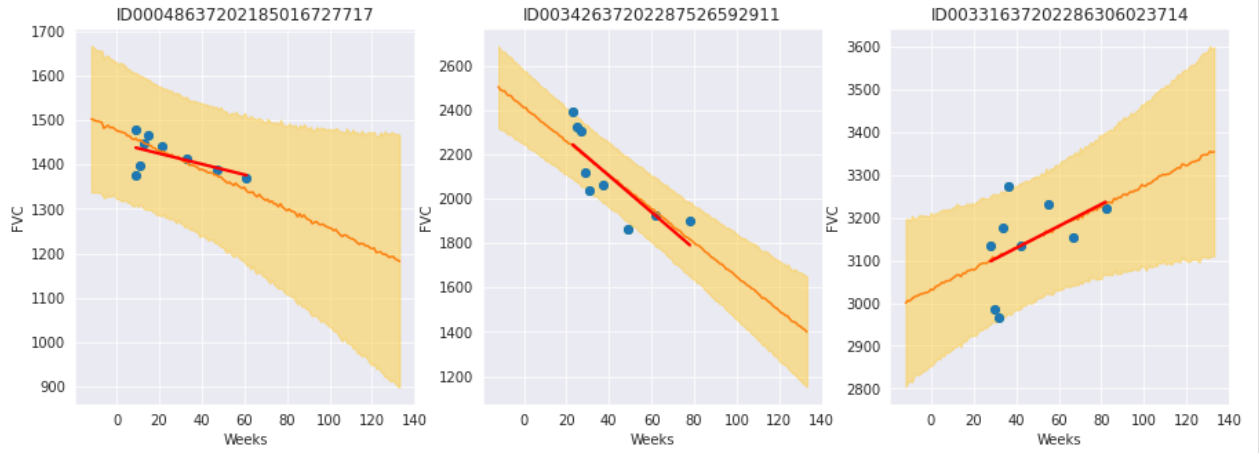
(continues on next page)

(continued from previous page)

```
3  ID00007637202177411956430              0     -9  2212.780518  154.162155
4  ID00007637202177411956430              0     -8  2219.202393  154.729507

       FVC_inf      FVC_sup  FVC_true
0  2071.218018  2386.979736       NaN
1  2067.664062  2382.380859       NaN
2  2069.071533  2379.903564       NaN
3  2058.618408  2366.942627       NaN
4  2064.472900  2373.931885       NaN
```

[29]:
```
f, axes = plt.subplots(1, 3, figsize=(15, 5))
chart_patient_with_predictions("ID00048637202185016727717", axes[0])  # Never smoked
chart_patient_with_predictions("ID00342637202287526592911", axes[1])  # Ex-smoker
chart_patient_with_predictions("ID00331637202286306023714", axes[2])  # Currently smokes
```



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:

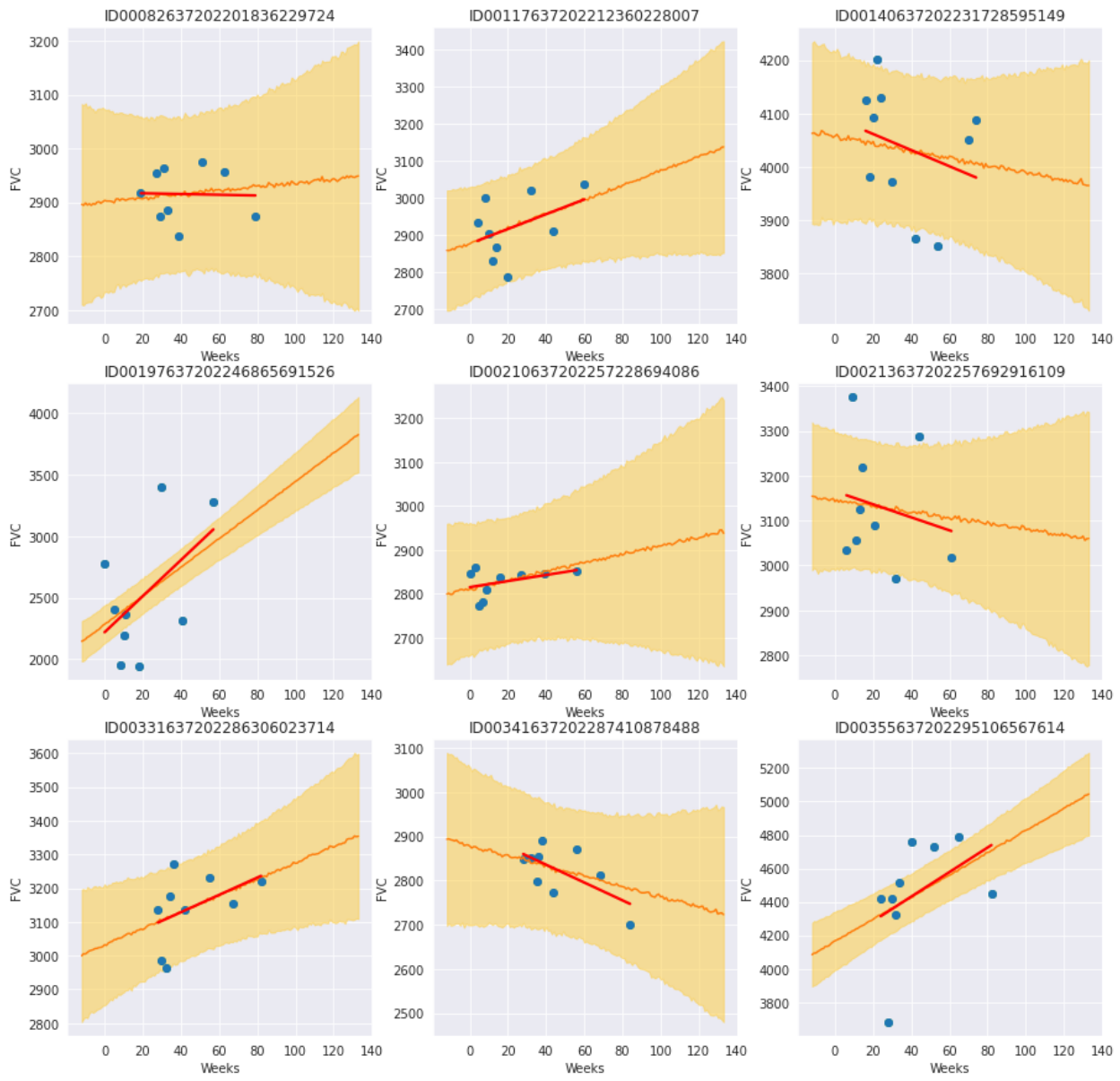
- If we just wanted to get an understanding of different attributes as they relate to patient's FVC over time, we could stop here with an understanding that current smokers might have an increase in FVC over time when being monitored for Pulmonary Fibrosis. We might hypothesize causes for this observation 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)

(continued from previous page)

```
):
    chart_patient_with_predictions(patient, axes.flatten()[i])
```



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.

[31]:
```
y = df.dropna()
rmse = ((y["FVC_pred"] - y["FVC_true"]) ** 2).mean() ** (1 / 2)
print(f"RMSE: {rmse:.1f} ml")
```

(continues on next page)

(continued from previous page)

```
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: 122.6 ml
Laplace Log Likelihood: -6.1420
```

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

1. Ghahramani, Z. Probabilistic machine learning and artificial intelligence. Nature 521, 452–459 (2015). https://doi.org/10.1038/nature14541
2. Rainforth, Thomas William Gamlen. Automating Inference, Learning, and Design Using Probabilistic Programming. University of Oxford, 2017.

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:

1. Carpenter B. (2016), "Hierarchical Partial Pooling for Repeated Binary Trials".
2. Efron B., Morris C. (1975), "Data analysis using Stein's estimator and its generalizations", J. Amer. Statist. Assoc., 70, 311-319.
3. Neal, R. (2012), "MCMC using Hamiltonian Dynamics", (https://arxiv.org/pdf/1206.1901.pdf)
4. Hoffman, M. D. and Gelman, A. (2014), "The No-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo", (https://arxiv.org/abs/1111.4246)
5. Michael Zhu (2019), "Sample Adaptive MCMC", (https://papers.nips.cc/paper/9107-sample-adaptive-mcmc)


```
import argparse
import os

import jax.numpy as jnp
import jax.random as random
from jax.scipy.special import logsumexp
```

(continues on next page)

(continued from previous page)

```
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
    $c_1 = m * kappa$, $c_2 = (1 - m) * kappa$, $m ~ Uniform(0, 1)$,
    and $kappa ~ Pareto(1, 1.5)$.

    :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))
```

(continues on next page)

(continued from previous page)

```
    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
    )
```

(continues on next page)

(continued from previous page)

```
    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(
            "\nLog 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("0.10.1")
    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)
```

(continues on next page)

(continued from previous page)

```
    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)
```

CHAPTER

TEN

EXAMPLE: VARIATIONAL AUTOENCODER

```
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,
```

(continues on next page)

(continued from previous page)

```
        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)

(continued from previous 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={}.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={}.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()
```

(continues on next page)

(continued from previous page)

```
        test_loss = eval_test(svi_state, rng_key_test, test_idx)
        reconstruct_img(i, rng_key_reconstruct)
        print(
            "Epoch {}: loss = {} ({:.2f} s.)".format(
                i, test_loss, time.time() - t_start
            )
        )


if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    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)
```

CHAPTER

ELEVEN

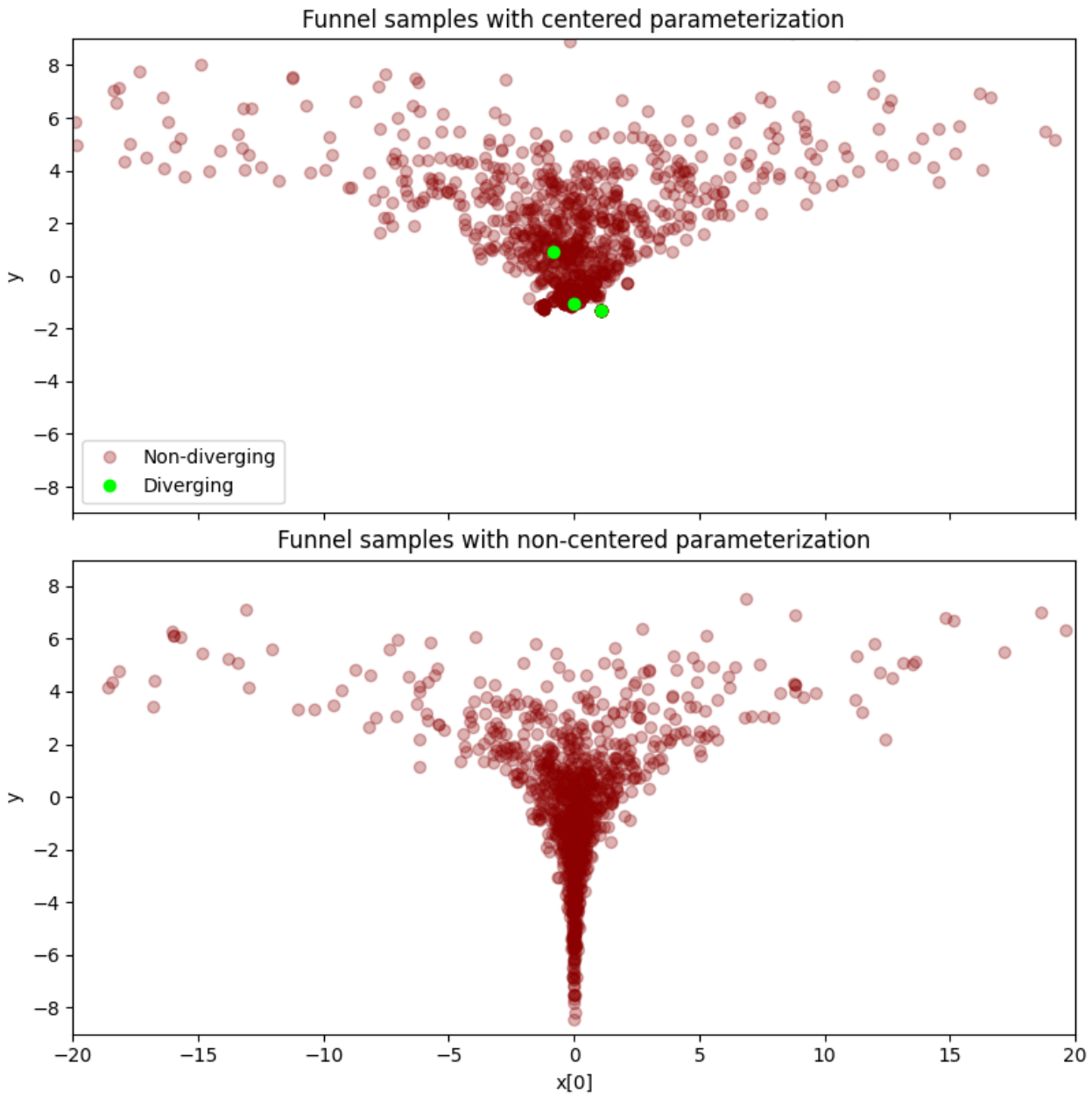
EXAMPLE: NEAL'S FUNNEL

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:

1. *Stan User's Guide*, https://mc-stan.org/docs/2_19/stan-users-guide/reparameterization-section.html
2. Maria I. Gorinova, Dave Moore, Matthew D. Hoffman (2019), "Automatic Reparameterisation of Probabilistic Programs", (https://arxiv.org/abs/1906.03028)




```
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)

(continued from previous page)

```
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========================== 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)

(continued from previous 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("0.10.1")
    parser = argparse.ArgumentParser(
```

(continues on next page)

(continued from previous page)

```
        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)
```

CHAPTER

TWELVE

EXAMPLE: STOCHASTIC VOLATILITY

Generative model:

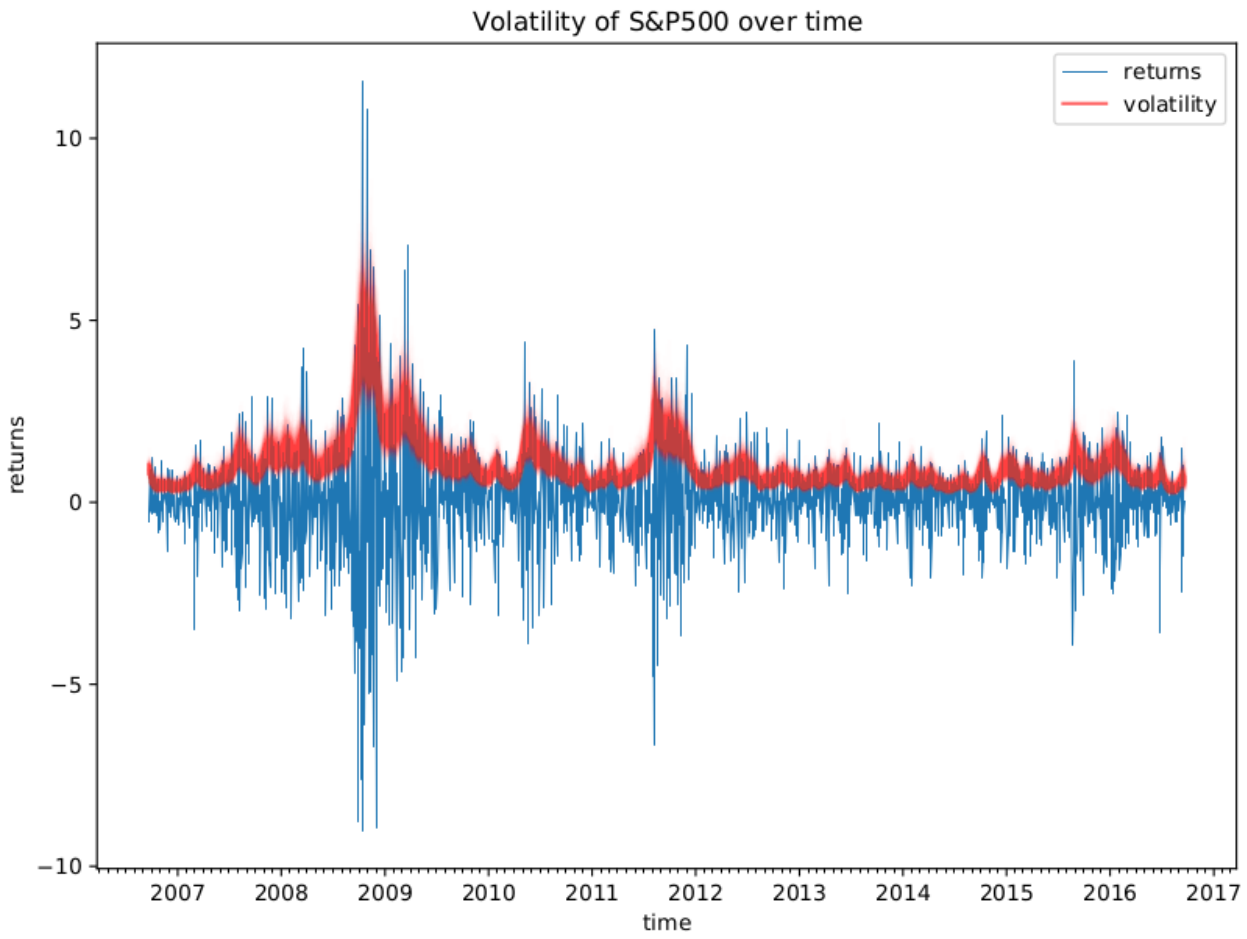
$$\sigma \sim \text{Exponential}(50) \tag{12.1}$$
$$\nu \sim \text{Exponential}(.1) \tag{12.2}$$
$$s_i \sim \text{Normal}(s_{i-1}, \sigma^{-2}) \tag{12.3}$$
$$r_i \sim \text{StudentT}(\nu, 0, \exp(s_i)) \tag{12.4}$$

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 *hmc*. However, we recommend readers to use *MCMC* class as in other examples because it is more stable and has more features supported.

References:

1. *Stochastic Volatility Model*, https://docs.pymc.io/notebooks/stochastic_volatility.html
2. *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*, https://arxiv.org/pdf/1111.4246.pdf
3. Pyro forum discussion, https://forum.pyro.ai/t/problems-transforming-a-pymc3-model-to-pyro-mcmc/208/14




```
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)

(continued from previous 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
```

(continues on next page)

(continued from previous page)

```
    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.legendHandles[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("0.10.1")
    parser = argparse.ArgumentParser(description="Stochastic Volatility Model")
    parser.add_argument("-n", "--num-samples", nargs="?", default=600, type=int)
    parser.add_argument("--num-warmup", nargs="?", 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)
```

CHAPTER

THIRTEEN

EXAMPLE: PRODLDA WITH FLAX AND HAIKU

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:

1. http://pyro.ai/examples/prodlda.html
2. Akash Srivastava, & Charles Sutton. (2017). Autoencoding Variational Inference For Topic Models.
3. Federico Bianchi, Silvia Terragni, and Dirk Hovy (2021), "Pre-training is a Hot Topic: Contextualized Document Embeddings Improve Topic Coherence" (https://arxiv.org/abs/2004.03974)


```
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)

(continued from previous 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):
```

(continues on next page)

(continued from previous page)

```
        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"]),
```

(continues on next page)

(continued from previous page)

```
            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(
```

(continues on next page)

(continued from previous page)

```
                    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=args.num_topics,
        hidden=args.hidden,
        dropout_rate=args.dropout_rate,
        batch_size=args.batch_size,
```

(continues on next page)

(continued from previous page)

```
    )

    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)
    wc = 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()
```

(continues on next page)

(continued from previous page)

```
    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("0.10.1")
    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", nargs="?", default=32, type=int)
    parser.add_argument("--learning-rate", nargs="?", default=1e-3, type=float)
    parser.add_argument("--hidden", nargs="?", default=200, type=int)
    parser.add_argument("--dropout-rate", nargs="?", 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)
```

CHAPTER

FOURTEEN

AUTOMATIC RENDERING OF NUMPYRO MODELS

In this tutorial we will demonstrate how to create beautiful visualizations of your probabilistic graphical models using `numpyro.render_model() <https://num.pyro.ai/en/stable/utilities.html#render-model>`__.

[1]:
```
!pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
```

[1]:
```
import flax
import flax.linen as flax_nn
from numpyro.contrib.module import flax_module
import numpy as np
from jax import nn
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
import numpyro.distributions.constraints as constraints

assert numpyro.__version__.startswith("0.10.1")
```

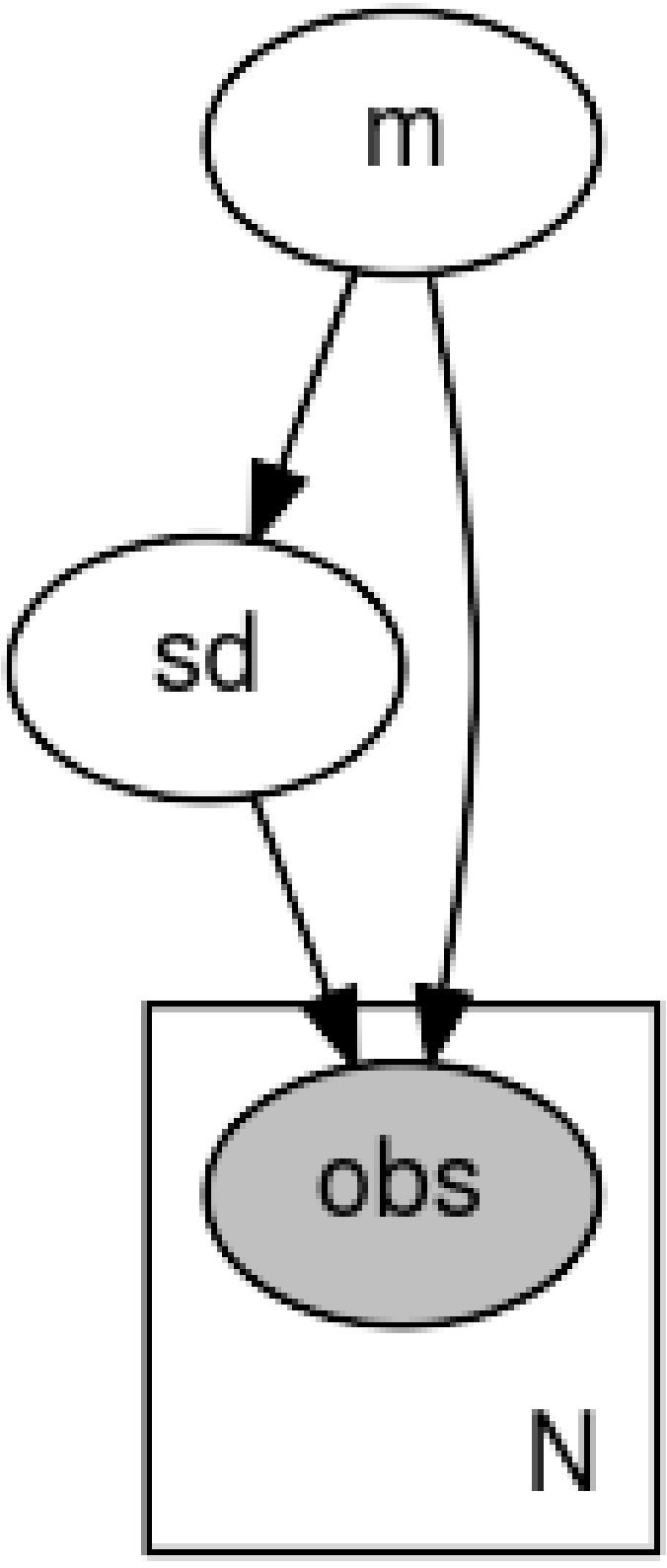
14.1 A Simple Example

The visualization interface can be readily used with your models:

[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)
```

[3]:
```
data = jnp.ones(10)
numpyro.render_model(model, model_args=(data,))
```

[3]:



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`.

[4]:
```
graph = numpyro.render_model(model, model_args=(data,), filename="model.pdf")
```

14.2 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.

[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)


positions = np.array([1, 1, 1, 2, 3, 4, 5])
# fmt: off
annotations = np.array([
    [1, 3, 1, 2, 2, 2, 1, 3, 2, 2, 4, 2, 1, 2, 1,
     1, 1, 1, 2, 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, 2, 1, 1, 3, 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, 1, 2, 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, 3, 2, 2, 4, 3, 1, 3, 1,
     2, 1, 1, 2, 1, 2, 2, 3, 2, 1, 1, 2, 1, 1, 1,
```

(continues on next page)

(continued from previous page)

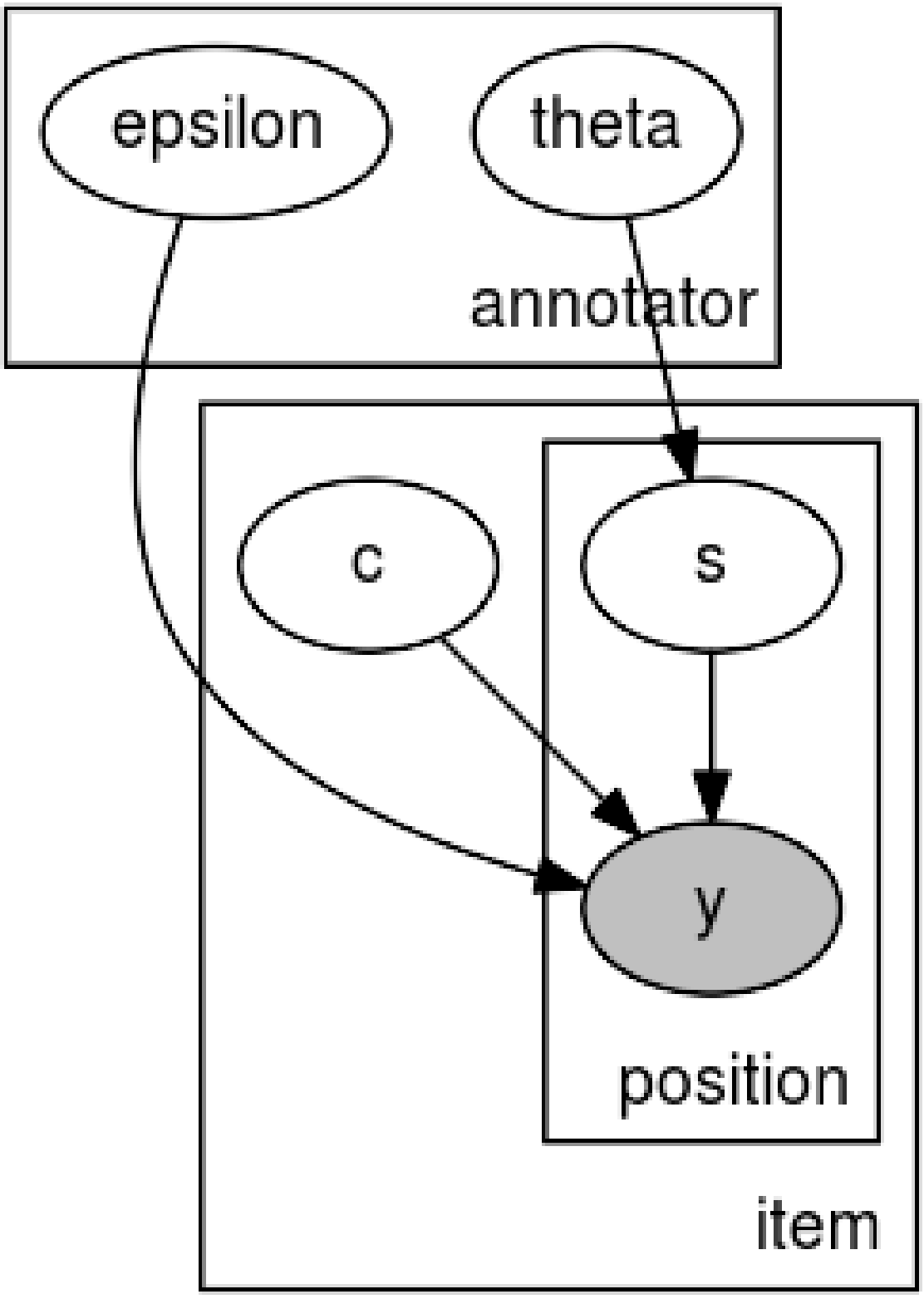
```
     1, 3, 1, 2, 3, 4, 2, 3, 3, 1, 1, 2, 2, 1, 2],
    [1, 3, 1, 1, 2, 3, 1, 4, 2, 2, 4, 3, 1, 2, 1,
     1, 1, 1, 2, 3, 2, 2, 2, 2, 1, 1, 2, 1, 1, 1,
     1, 2, 1, 2, 2, 3, 2, 2, 4, 1, 1, 1, 2, 1, 2],
    [1, 3, 2, 2, 2, 2, 1, 3, 2, 2, 4, 4, 1, 1, 1,
     1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1, 2,
     1, 3, 1, 2, 3, 4, 3, 3, 3, 1, 1, 1, 2, 1, 2],
    [1, 4, 2, 1, 2, 2, 1, 3, 3, 3, 4, 3, 1, 2, 1,
     1, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 1, 1, 1,
     1, 3, 1, 2, 2, 3, 2, 3, 2, 1, 1, 1, 2, 1, 2],
]).T
# fmt: on

# 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))
```

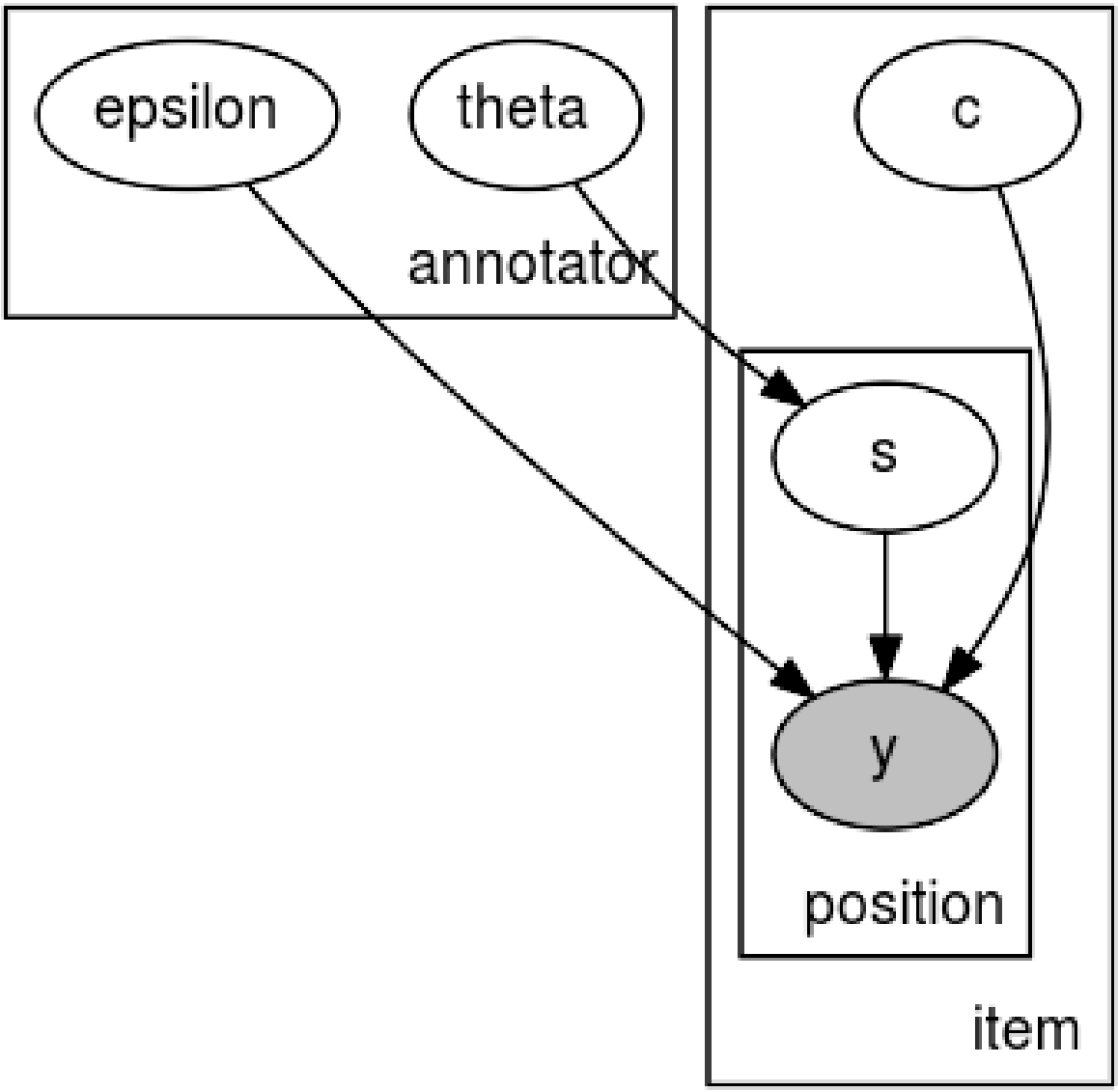
[6]:
```
# default layout
mace_graph
```

[6]:



[7]:
```
# layout after processing the layout with unflatten
mace_graph.unflatten(stagger=2)
```

[7]:



14.3 Rendering the parameters

We can render the parameters defined as `numpyro.param` by setting `render_params=True` in `numpyro.render_model`.

[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))
```

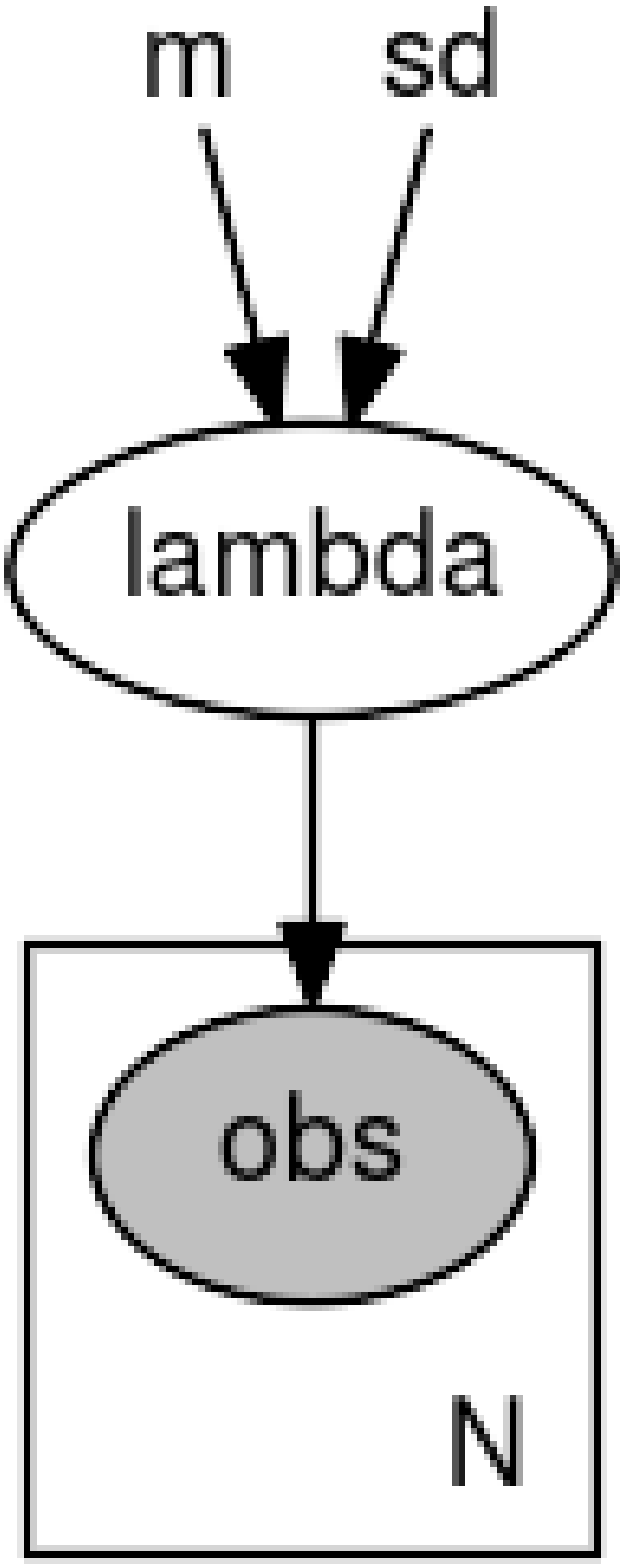
(continues on next page)

(continued from previous page)

```
    with numpyro.plate("N", len(data)):
        numpyro.sample("obs", dist.Exponential(lambd), obs=data)
```

[9]:
```
data = jnp.ones(10)
numpyro.render_model(model, model_args=(data,), render_params=True)
```

[9]:

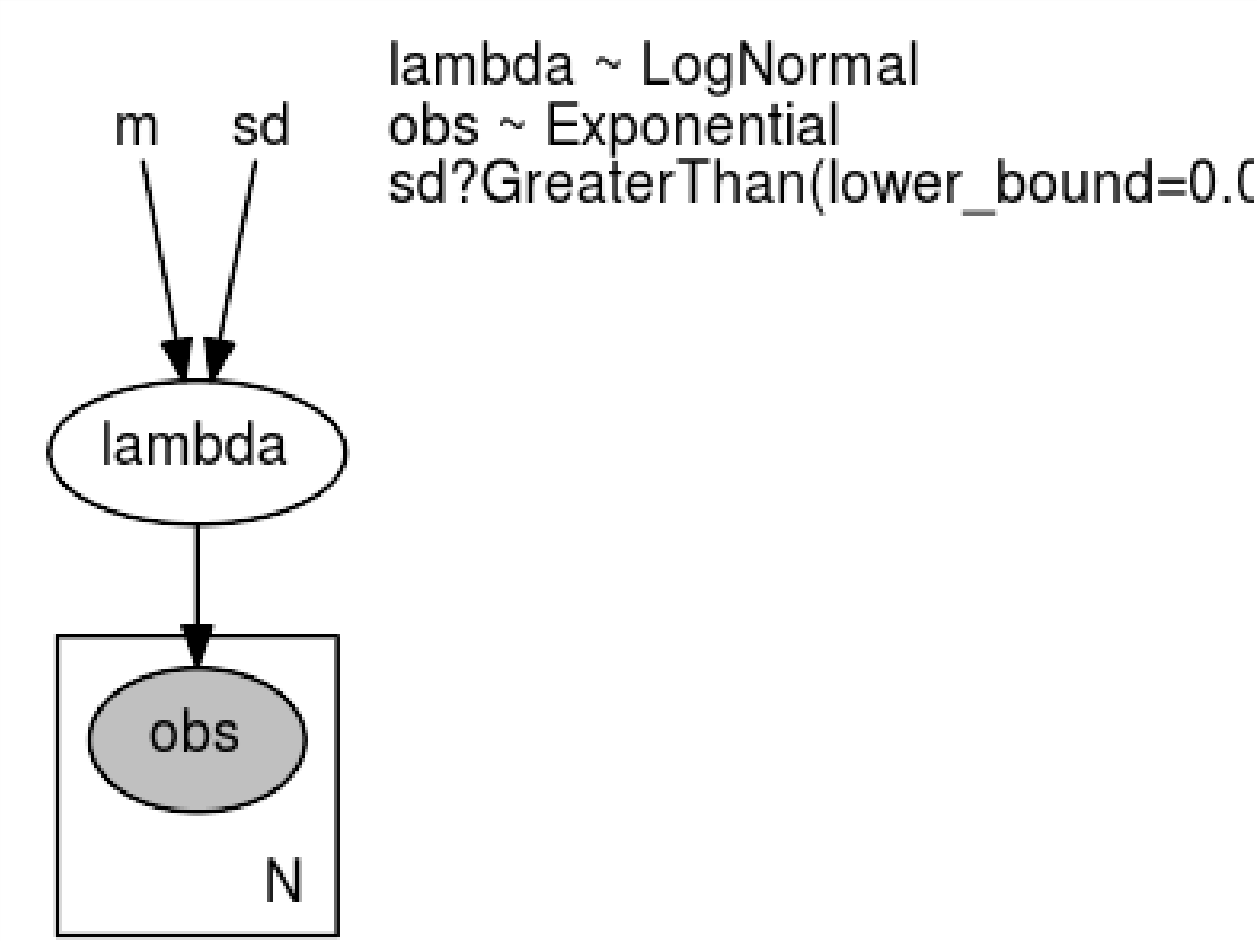


14.4 Distribution and Constraint annotations

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
)
```

[10]:



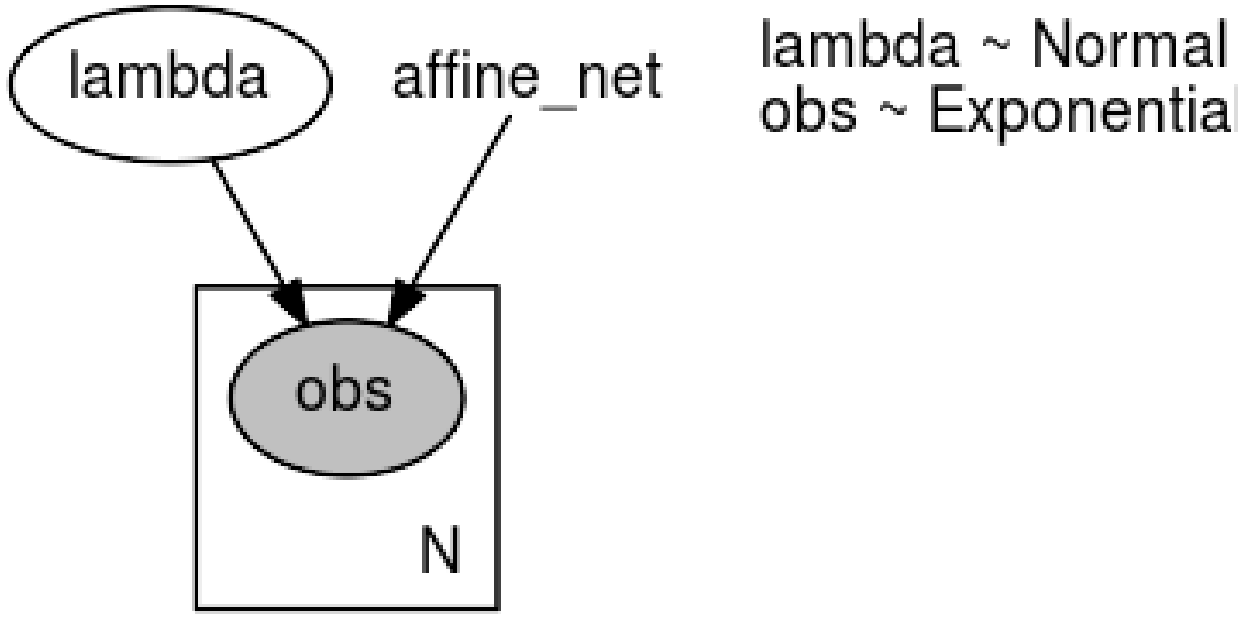
In the above plot **'~'** denotes the distribution of RV and **':math:`in`'** denotes the constraint of parameter.

14.5 Rendering neural network's parameters

[11]:
```
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)
```

[12]:
```
numpyro.render_model(
    model, model_args=(data,), render_distributions=True, render_params=True
)
```

[12]:



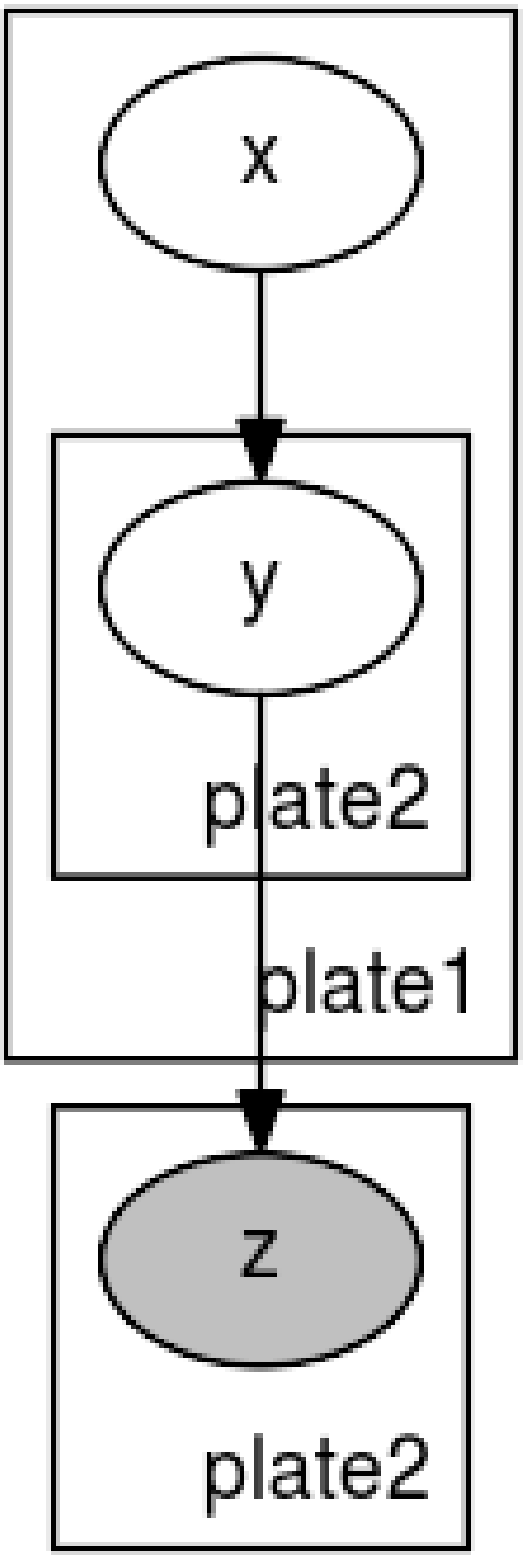
14.6 Overlapping non-nested plates

Note that overlapping non-nested plates may be drawn as multiple rectangles.

[13]:
```
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))
```

[14]:
```
numpyro.render_model(model)
```

[14]:



CHAPTER

FIFTEEN

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.

[1]:
```
!pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
```

[1]:
```
from functools import partial

import numpy as np

import jax.numpy as jnp
from jax import random

import numpyro
import numpyro.distributions as dist
from numpyro.diagnostics import summary

from numpyro.infer import MCMC, NUTS

assert numpyro.__version__.startswith("0.10.1")

# 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.

[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)
```

(continues on next page)

(continued from previous page)

```
    mcmc = MCMC(
        kernel,
        num_warmup=num_warmup,
        num_samples=num_samples,
        num_chains=1,
        progress_bar=False,
    )
    mcmc.run(random.PRNGKey(0))
    summary_dict = summary(mcmc.get_samples(), group_by_chain=False)

    # print the largest r_hat for each variable
    for k, v in summary_dict.items():
        spaces = " " * max(12 - len(k), 0)
        print("[{}] {} \t max r_hat: {:.4f}".format(k, spaces, np.max(v["r_hat"])))
```

15.1 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_hat` diagnostics returned by `mcmc.print_summary()`. If we see values of `r_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_hat`s for some of the variables (e.g. `r_hat = 1.15`) then HMC is likely struggling with the posterior geometry. In the following we will use `r_hat` as our primary diagnostic metric.

15.2 Model reparameterization

15.2.1 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].

[3]:
```
# In this unreparameterized model some of the parameters of the distributions
# explicitly depend on other parameters (in particular beta depends on lambdas and tau).
# This kind of coordinate system can be a challenge for HMC.
def _unrep_hs_model(X, Y):
    lambdas = numpyro.sample("lambdas", dist.HalfCauchy(jnp.ones(X.shape[1])))
    tau = numpyro.sample("tau", dist.HalfCauchy(jnp.ones(1)))
    betas = numpyro.sample("betas", dist.Normal(scale=tau * lambdas))
    mean_function = jnp.dot(X, betas)
    numpyro.sample("Y", dist.Normal(mean_function, 0.05), obs=Y)
```

To deal with the bad geometry that results form 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 β is now defined *deterministically* in terms of λ, τ, and β'. 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 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.
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 numpyro.infer.reparam. There are at least two ways to do this. First let's use LocScaleReparam.

[5]:
```
from numpyro.infer.reparam import LocScaleReparam

# 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 numpyro.infer.reparam library let's do the reparameterization using TransformReparam instead.

[6]:
```
from numpyro.distributions.transforms import AffineTransform
from numpyro.infer.reparam import TransformReparam

# 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)))
```

(continues on next page)

(continued from previous page)

```
    # instruct NumPyro to do the reparameterization automatically.
    reparam_config = {"betas": TransformReparam()}
    with numpyro.handlers.reparam(config=reparam_config):
        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
```

(continues on next page)

(continued from previous page)

```
reparameterized model with TransformReparam (good r_hats)
[betas]                  max r_hat: 1.0087
[betas_base]            max r_hat: 1.0080
[lambdas]                max r_hat: 1.0114
[tau]                    max r_hat: 1.0029
```

15.2.2 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:

```
scaled_betas = tau * lambdas * unscaled_betas
```

without invoking the `deterministic` primitive.

15.3 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.

15.3.1 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`.

[9]:
```
# 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)
```

```
dense_mass = False (bad r_hat)
[x]                      max r_hat: 1.3810
dense_mass = True (good r_hat)
[x]                      max r_hat: 0.9992
```

15.3.2 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.

[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)
    )
    y = 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`.

[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
#   x1 | * * 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
[x2]                    max r_hat: 1.0738
```

(continues on next page)

(continued from previous page)

```
[y]                     max r_hat: 1.2746

structured mass matrix (good r_hats)
[x1]                    max r_hat: 1.0023
[x2]                    max r_hat: 1.0024
[y]                     max r_hat: 1.0030
```

15.4 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`).

15.4.1 Example #4

[12]:
```
# 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():
    x = 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)
[x]                     max r_hat: 1.1159
max_tree_depth = 10 (good r_hat)
[x]                     max r_hat: 1.0166
```

15.5 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].

15.6 References

[1] "Hamiltonian Monte Carlo for Hierarchical Models," M. J. Betancourt, Mark Girolami.

[2] "NeuTra-lizing Bad Geometry in Hamiltonian Monte Carlo Using Neural Transport," Matthew Hoffman, Pavel Sountsov, Joshua V. Dillon, Ian Langmore, Dustin Tran, Srinivas Vasudevan.

[3] "Reparameterization" in the Stan user's manual. https://mc-stan.org/docs/2_27/stan-users-guide/reparameterization-section.html

CHAPTER

SIXTEEN

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.

16.1 Table of contents

16.2 Setup

To run this notebook, we are going to need the following imports

[ ]:
```
!pip install -q git+https://github.com/pyro-ppl/numpyro.git
```

[2]:
```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import numpy as np
import numpyro
import numpyro.distributions as dist
from jax import lax, random
from jax.scipy.special import ndtr, ndtri
from jax.scipy.stats import poisson, norm
```

(continues on next page)

(continued from previous page)

```
from numpyro.distributions import (
    constraints,
    Distribution,
    FoldedDistribution,
    SoftLaplace,
    StudentT,
    TruncatedDistribution,
    TruncatedNormal,
)
from numpyro.distributions.util import promote_shapes
from numpyro.infer import DiscreteHMCGibbs, MCMC, NUTS, Predictive
from scipy.stats import poisson as sp_poisson

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",
)
```

16.3 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

$$p_Z(z) \propto \begin{cases} p_Y(z) & \text{if } z \text{ is in } T \\ 0 & \text{if } z \text{ is outside } T \end{cases} \tag{16.1}$$

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:

$$p_Z(z) = \begin{cases} \frac{1}{M} p_Y(z) & \text{if } z \text{ is in } T \\ 0 & \text{if } z \text{ is outside } T \end{cases} \tag{16.2}$$

where $M = \int_T p_Y(y) \mathrm{d}y$.

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 $N(1, 1)$.

```
<img src="https://i.ibb.co/6vHyFfq/truncated-normal.png" alt="truncated" width="900"/>
```

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.

16.4 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:

$$p_Z(z) = \begin{cases} p_Y(z) + p_Y(-z) & \text{if } z \geq 0 \\ 0 & \text{if } z0 \end{cases} \tag{16.3}$$

The figure below illustrates a folded normal distribution $N(1, 1)$.

```
<img src="https://i.ibb.co/3d2xJbc/folded-normal.png" alt="folded" width="900"/>
```

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 3 and 4*

If the original distribution is symmetric around zero, then folding and truncating at zero have the same effect.

16.5 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:

```
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:

```
samples = np.random.normal(size=1000)
folded_samples = np.abs(samples)
```

16.6 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:

[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

[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.

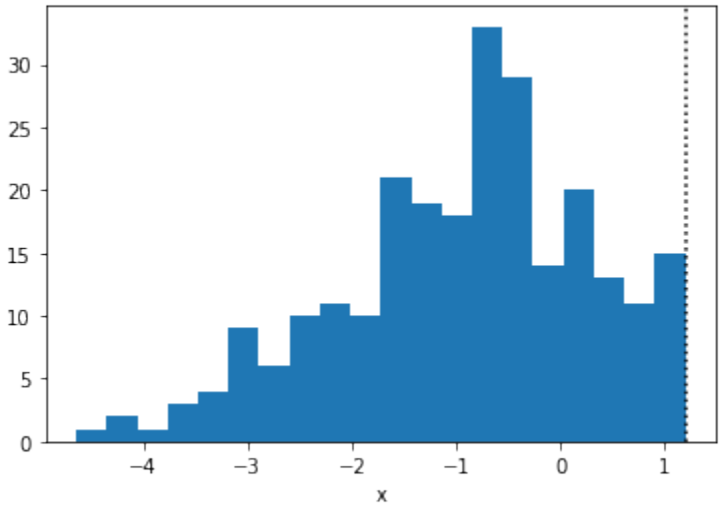
[5]:
```
# -- select an arbitrary prior sample as true data
true_idx = 0
true_loc = prior_samples["loc"][true_idx]
true_scale = prior_samples["scale"][true_idx]
true_x = prior_samples["x"][true_idx]
```

[6]:
```
plt.hist(true_x.copy(), bins=20)
plt.axvline(high, linestyle=":", color="k")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("x")
plt.show()
```



[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:3.2}")
print(f"True scale: {true_scale:3.2}")
```

```
sample: 100
↪%|
↪4000/4000 [00:02<00:00, 1909.24it/s, 1 steps of size 5.65e-01. acc. prob=0.93]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 10214.14it/s, 3 steps of size 5.16e-01. acc. prob=0.95]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 15102.95it/s, 1 steps of size 6.42e-01. acc. prob=0.90]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 16522.03it/s, 3 steps of size 6.39e-01. acc. prob=0.90]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       loc     -0.58      0.15     -0.59     -0.82     -0.35   2883.69      1.00
     scale      1.49      0.11      1.48      1.32      1.66   3037.78      1.00

Number of divergences: 0
True loc  : -0.56
True scale: 1.4
```

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 `high` to infinity.

[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.

[9]:
```
# thin the samples to not saturate matplotlib
samples_thinned = pred_samples["x"].ravel()[::1000]
```

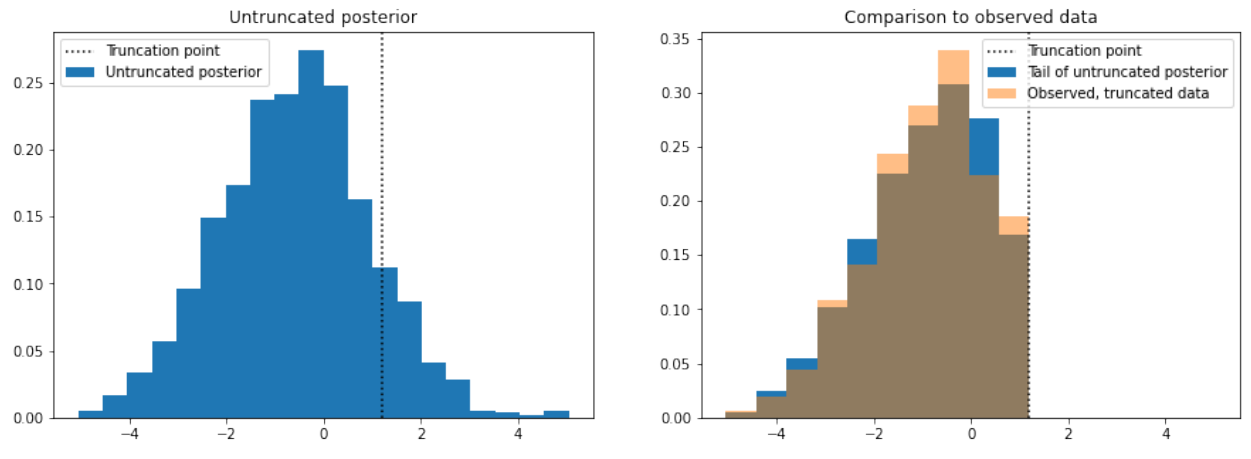
[10]:
```
f, axes = plt.subplots(1, 2, figsize=(15, 5), sharex=True)

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")
    ax.legend()

plt.show()
```



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:

[11]:
```
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,
    )
```

[12]:
```
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:

[13]:
```
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:3.2}")
print(f"True scale: {true_scale:3.2}")
```

```
sample: 100
↪%|
↪4000/4000 [00:02<00:00, 1745.70it/s, 1 steps of size 6.78e-01. acc. prob=0.93]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 9294.56it/s, 1 steps of size 7.02e-01. acc. prob=0.93]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 10412.30it/s, 1 steps of size 7.20e-01. acc. prob=0.92]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 10583.85it/s, 3 steps of size 7.01e-01. acc. prob=0.93]

                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.

[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:

[16]:
```
# --- prior sampling
num_observations = 500
num_prior_samples = 100
```

(continues on next page)

(continued from previous page)

```
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 diagostics
mcmc.print_summary()

# --- Compare to ground truth:
print(f"True df   : {true_df:3.2f}")
print(f"True loc  : {true_loc:3.2f}")
print(f"True scale: {true_scale:3.2f}")
```

```
sample: 100
↪%|
↪4000/4000 [00:02<00:00, 1343.54it/s, 7 steps of size 3.51e-01. acc. prob=0.75]
sample: 100
↪%|
↪4000/4000 [00:01<00:00, 3644.99it/s, 7 steps of size 3.56e-01. acc. prob=0.73]
sample: 100
↪%|
↪4000/4000 [00:01<00:00, 3137.13it/s, 7 steps of size 2.62e-01. acc. prob=0.91]
sample: 100
↪%|
↪4000/4000 [00:01<00:00, 3028.93it/s, 7 steps of size 1.85e-01. acc. prob=0.96]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
        df      3.12      0.52      3.07      2.30      3.97   2057.60      1.00
       loc     -0.02      0.88     -0.03     -1.28      1.34    925.84      1.01
     scale      2.23      0.21      2.25      1.89      2.57   1677.38      1.00

Number of divergences: 33
True df   : 3.01
True loc  : 0.37
True scale: 2.41
```

16.7 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.

16.7.1 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 auto-guides, 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

```
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
```

16.7.2 5.2 Example: Right-truncated normal

We are going to modify a normal distribution so that its new support is of the form `(-inf, high)`, with `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.

```
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
reparametrized_params = ["loc", "scale"]
```

The `reparametrized_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 `reparametrized_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:

```
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:

```
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.

```
# ...
    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

$$\log p_Z(z) = \begin{cases} \log p_Y(z) - \log M & \text{if } z \text{ is in } T \\ -\infty & \text{if } z \text{ is outside } T \end{cases} \tag{16.4}$$

where, again, p_Z is the density of the truncated distribution, p_Y is the density before truncation, and $M = \int_T p_Y(y)\mathrm{d}y$. For the specific case of truncating the normal distribution to the interval `(-inf, high)`, the constant M is equal to the cumulative density evaluated at the truncation point. We can easily implement this log-density method because `jax.scipy.stats` already has a `norm` module that we can use.

```
# ...
    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 `ndtri` function that lives inside `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 $X \sim Normal(0, 1)$ and $Y = \mu + \sigma X$, then $Y \sim Normal(\mu, \sigma)$. Then if Z is the truncated Y, its cumulative density is given by:

$$F_Z(y) = \int_{-\infty}^{y} p_Z(r)dr \qquad = \frac{1}{M}\int_{-\infty}^{y} p_Y(s)ds \quad \text{if } y < high = \frac{1}{M}F_Y(y) \tag{16.5}$$

And so its inverse is

$$F_Z^{-1}(u) = \left(\frac{1}{M}F_Y\right)^{-1}(u) = F_Y^{-1}(Mu) = F_{\mu+\sigma X}^{-1}(Mu) = \mu + \sigma F_X^{-1}(Mu) \tag{16.6}$$

The translation of the above math into code is

```
# ...
    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.

[17]:
```
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()


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,
```

(continues on next page)

(continued from previous page)

```
        "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)
```

Let's try it out!

[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)
```

[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:

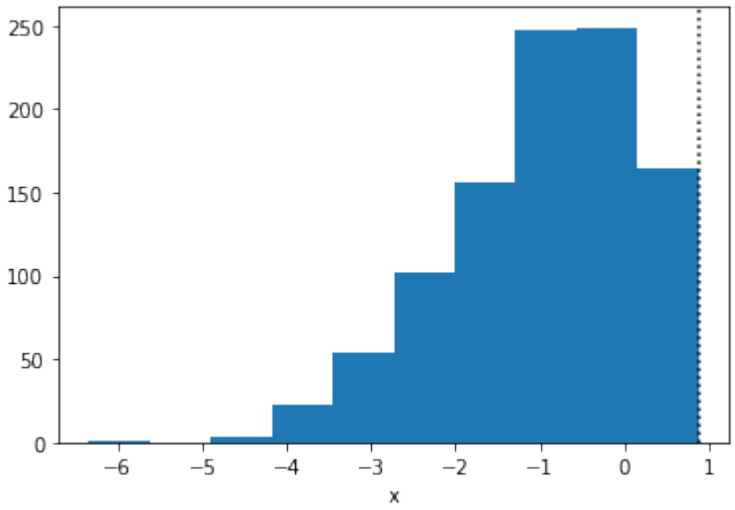
[20]:
```
true_idx = 0
true_loc = prior_samples["loc"][true_idx]
true_scale = prior_samples["scale"][true_idx]
```

(continues on next page)

(continued from previous page)

```
true_high = prior_samples["high"][true_idx]
true_x = prior_samples["x"][true_idx]
```

[21]:
```
plt.hist(true_x.copy())
plt.axvline(true_high, linestyle=":", color="k")
plt.xlabel("x")
plt.show()
```



Run MCMC and check the estimates:

[22]:
```
mcmc = MCMC(NUTS(truncated_normal_model), **MCMC_KWARGS)
mcmc.run(MCMC_RNG, num_observations, true_x)
mcmc.print_summary()
```

```
sample: 100
↪%|
↪4000/4000 [00:02<00:00, 1850.91it/s, 15 steps of size 8.88e-02. acc. prob=0.88]
sample: 100
↪%|
↪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]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
      high      0.88      0.01      0.88      0.88      0.89    590.13      1.01
       loc     -0.58      0.07     -0.58     -0.70     -0.46    671.04      1.01
     scale      1.40      0.05      1.40      1.32      1.48    678.30      1.01

Number of divergences: 6310
```

Compare estimates against the ground truth:

[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:

[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)
```

[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
↪%|
↪4000/4000 [00:00<00:00, 8802.95it/s, 7 steps of size 5.19e-01. acc. prob=0.92]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 8975.35it/s, 3 steps of size 5.72e-01. acc. prob=0.89]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 8471.94it/s, 15 steps of size 3.76e-01. acc. prob=0.96]

                mean       std    median      5.0%     95.0%     n_eff     r_hat
     delta       0.01      0.01      0.00      0.00      0.01   6104.22      1.00
      high       0.88      0.01      0.88      0.88      0.89   6104.22      1.00
       loc      -0.58      0.08     -0.58     -0.71     -0.46   3319.65      1.00
     scale       1.40      0.06      1.40      1.31      1.49   3377.38      1.00

Number of divergences: 0
```

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:

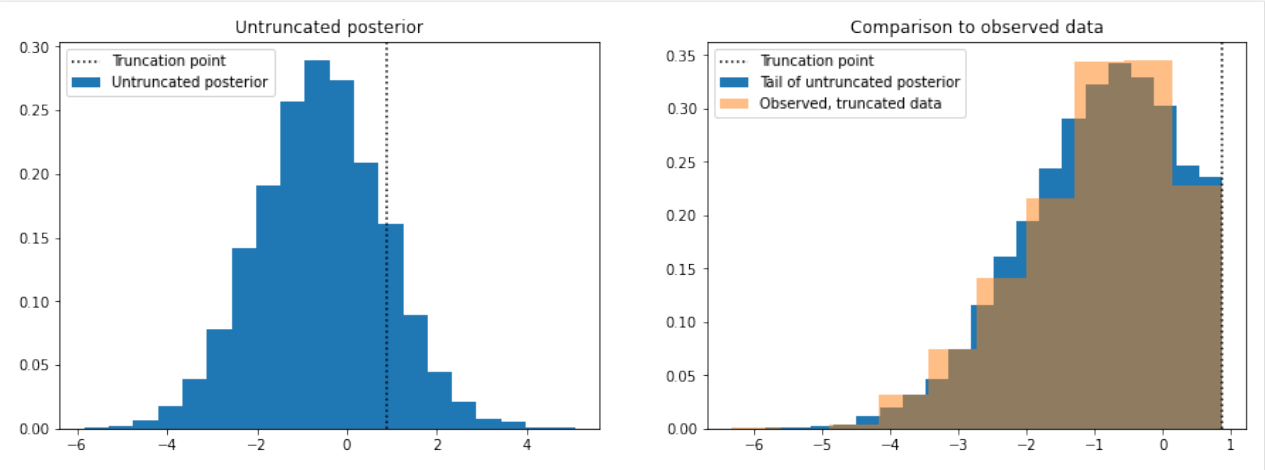
[26]:
```
model_without_truncation = numpyro.handlers.condition(
    truncated_normal_model,
    {"high": float("inf")},
)
estimates = mcmc.get_samples().copy()
estimates.pop("high")  # Drop to make sure these are not used
pred = Predictive(
    model_without_truncation,
    posterior_samples=estimates,
)
pred_samples = pred(PRED_RNG, num_observations=1000)
```

[27]:
```
# thin the samples for a faster histogram
samples_thinned = pred_samples["x"].ravel()[::1000]
```

[28]:
```
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",
    bins=20,
    density=True,
)
axes[1].hist(true_x.copy(), label="Observed, truncated data", density=True, alpha=0.5)
axes[1].axvline(true_high, linestyle=":", color="k", label="Truncation point")
axes[1].set_title("Comparison to observed data")
axes[1].legend()
plt.show()
```



16.7.3 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.

```
# ...
def __init__(self, rate=1.0, low=0, validate_args=None):
```

(continues on next page)

(continued from previous page)

```
    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:

$$M = \sum_{n=L}^{\infty} p_Y(n) = 1 - \sum_{n=0}^{L-1} p_Y(n) = 1 - F_Y(L-1) \tag{16.7}$$

For the code, we can rely on the `poisson` module that lives inside `jax.scipy.stats`.

```
# ...
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:

$$F^{-1}(u) = \max\{n \in \mathbb{N} | F(n) u\} \tag{16.8}$$

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:

$$F_Z(z) = \sum_{n=0}^{z} p_z(n) \quad = \frac{1}{M} \sum_{n=L}^{z} p_Y(n) \quad \text{assuming } z >= L = \frac{1}{M} \left(\sum_{n=0}^{z} p_Y(n) - \sum_{n=0}^{L-1} p_Y(n) \right) \quad = \frac{1}{M} \left(F_Y(z) - F_Y(L-1) \right) \tag{16.9}$$

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)

```
# ...
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)
```

(continues on next page)

(continued from previous page)

```
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 `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 `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:

```
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 `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* icdf as:

$$F_Z^{-1}(u) = F_Y^{-1}(Mu + F_Y(L-1)) \tag{16.10}$$

And in python:

```
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
```

(continues on next page)

(continued from previous page)

```
    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,
    }

    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)
```

(continues on next page)

(continued from previous page)

```
    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))

    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)
```

(continues on next page)

(continued from previous page)

```
    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):
    low = numpyro.sample("low", dist.Categorical(0.2 * jnp.ones((5,))))
    rate = numpyro.sample("rate", dist.LogNormal(1, 1))
    with numpyro.plate("observations", num_observations):
        numpyro.sample("x", LeftTruncatedPoisson(rate, low), obs=x)
```

Prior samples

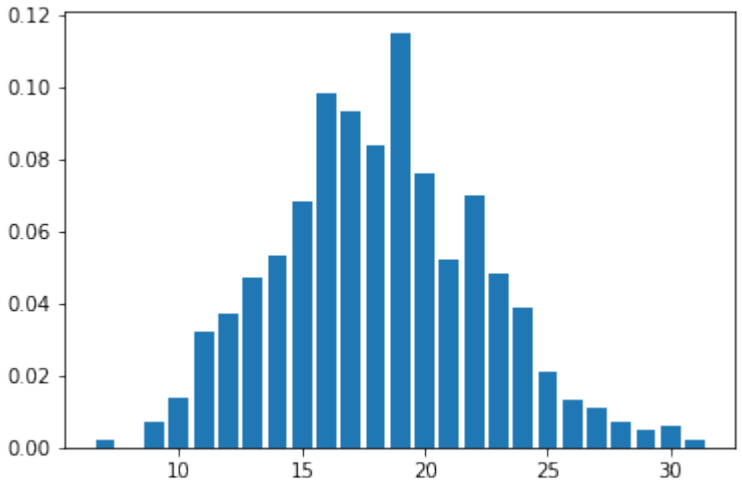
[32]:
```
# -- prior samples
num_observations = 1000
num_prior_samples = 100
prior = Predictive(truncated_poisson_model, num_samples=num_prior_samples)
prior_samples = prior(PRIOR_RNG, num_observations)
```

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)
```

[34]:
```
# Take any prior sample as the true process.
true_idx = 6
true_low = prior_samples["low"][true_idx]
true_rate = prior_samples["rate"][true_idx]
true_x = prior_samples["x"][true_idx]
discrete_distplot(true_x.copy());
```



To do inference, we set `k = x.min() + 1`. Note also the use of DiscreteHMCGibbs:

[35]:
```
mcmc = MCMC(DiscreteHMCGibbs(NUTS(truncated_poisson_model)), **MCMC_KWARGS)
mcmc.run(MCMC_RNG, num_observations, true_x, k=true_x.min() + 1)
mcmc.print_summary()
```

```
sample: 100
↪%|
↪4000/4000 [00:04<00:00, 808.70it/s, 3 steps of size 9.58e-01. acc. prob=0.93]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 5916.30it/s, 3 steps of size 9.14e-01. acc. prob=0.93]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 5082.16it/s, 3 steps of size 9.91e-01. acc. prob=0.92]
sample: 100
↪%|
↪4000/4000 [00:00<00:00, 6511.68it/s, 3 steps of size 8.66e-01. acc. prob=0.94]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
low	4.13	2.43	4.00	0.00	7.00	7433.79	1.00
rate	18.16	0.14	18.16	17.96	18.40	3074.46	1.00

[36]:
```
true_rate
```

[36]:
```
DeviceArray(18.2091848, dtype=float64)
```

As before, one needs to be extra careful when estimating the truncation point. If the truncation point is known is best to provide it.

[37]:
```
model_with_known_low = numpyro.handlers.condition(
    truncated_poisson_model, {"low": true_low}
)
```

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()
```

```
sample: 100
↪%|████████████████████████████████████████████████████████████████████████████████
↪4000/4000 [00:03<00:00, 1185.13it/s, 1 steps of size 9.18e-01. acc. prob=0.93]
sample: 100
↪%|████████████████████████████████████████████████████████████████████████████████
↪4000/4000 [00:00<00:00, 5786.32it/s, 3 steps of size 1.00e+00. acc. prob=0.92]
sample: 100
↪%|████████████████████████████████████████████████████████████████████████████████
↪4000/4000 [00:00<00:00, 5919.13it/s, 1 steps of size 8.62e-01. acc. prob=0.94]
sample: 100
↪%|████████████████████████████████████████████████████████████████████████████████
↪4000/4000 [00:00<00:00, 7562.36it/s, 3 steps of size 9.01e-01. acc. prob=0.93]
```

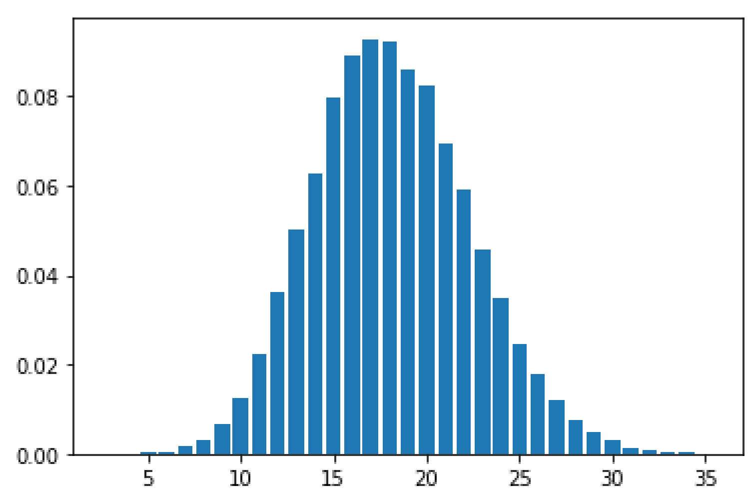
```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
      rate     18.17      0.13     18.17     17.95     18.39   3406.81      1.00

Number of divergences: 0
```

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());
```



16.8 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

SEVENTEEN

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 refererences 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:

1. Paun, S., Carpenter, B., Chamberlain, J., Hovy, D., Kruschwitz, U., and Poesio, M. (2018). "Comparing bayesian models of annotation" (https://www.aclweb.org/anthology/Q18-1040/)
2. Dawid, A. P., and Skene, A. M. (1979). "Maximum likelihood estimation of observer error-rates using the EM algorithm"
3. "Inference with Discrete Latent Variables" (http://pyro.ai/examples/enumeration.html)


```
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`.
```

(continues on next page)

(continued from previous page)

```
        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, 1], [2, 2, 2, 3, 2, 2, 2], [2, 2, 2, 3, 3, 2, 2],
         [1, 2, 2, 2, 1, 1, 1], [3, 3, 3, 3, 4, 3, 3], [2, 2, 2, 2, 2, 2, 3],
         [2, 3, 2, 2, 2, 2, 3], [4, 4, 4, 4, 4, 4, 4], [2, 2, 2, 3, 3, 4, 3],
         [1, 1, 1, 1, 1, 1, 1], [2, 2, 2, 3, 2, 1, 2], [1, 2, 1, 1, 1, 1, 1],
         [1, 1, 1, 2, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1],
         [2, 2, 2, 2, 2, 2, 1], [2, 2, 2, 1, 3, 2, 2], [2, 2, 2, 2, 2, 2, 2],
         [2, 2, 2, 2, 2, 2, 1], [2, 2, 2, 3, 2, 2, 2], [2, 2, 1, 2, 2, 2, 2],
         [1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1], [2, 3, 2, 2, 2, 2, 2],
         [1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1], [1, 1, 2, 1, 1, 2, 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, 3, 2, 3, 2], [4, 3, 3, 4, 3, 4, 3],
         [2, 2, 1, 2, 2, 3, 2], [2, 3, 2, 3, 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, 2, 1, 2, 1, 1, 1],
         [2, 3, 2, 2, 2, 2, 2], [1, 2, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2, 2]])
    # fmt: on
    # we minus 1 because in Python, the first index is 0
    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
```

(continues on next page)

(continued from previous page)

```
    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`
        # ref: http://num.pyro.ai/en/latest/utilities.html#numpyro.contrib.indexing.
↪vindex
        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)


def hierarchical_dawid_skene(positions, annotations):
```

(continues on next page)

(continued from previous page)

```
    """
    This model corresponds to the plate diagram in Figure 4 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):
        # 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)


def item_difficulty(annotations):
    """
    This model corresponds to the plate diagram in Figure 5 of reference [1].
    """
    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)

(continued from previous 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)

(continued from previous page)

```
    "mn": multinomial,
    "ds": dawid_skene,
    "mace": mace,
    "hds": hierarchical_dawid_skene,
    "id": item_difficulty,
    "lre": logistic_random_effects,
}


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:

```
chain_discrete_samples = jax.tree_util.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)*.

```
if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    parser = argparse.ArgumentParser(description="Bayesian Models of Annotation")
    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(
        "--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

EIGHTEEN

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(length) to O(log(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)
2. *Temporal Parallelization of Bayesian Smoothers*, Simo Sarkka, Angel F. Garcia-Fernandez (https://arxiv.org/abs/1905.13002)
3. *Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription*, Boulanger-Lewandowski, N., Bengio, Y. and Vincent, P.
4. *Tensor Variable Elimination for Plated Factor Graphs*, Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Justin Chiu, Neeraj Pradhan, Alexander Rush, Noah Goodman (https://arxiv.org/abs/1902.03210)


```
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)

(continued from previous 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.

```
#     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):
            with mask(mask=(t < lengths)[..., None]):
                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)
    # NB swapaxes: we move time dimension of `sequences` to the front to scan over it
    scan(transition_fn, (x_init, 0), jnp.swapaxes(sequences, 0, 1))
```

Next let's add a dependency of y[t] on y[t-1].

```
#     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)

(continued from previous 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)
```

(continues on next page)

(continued from previous page)

```
        )
        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␣
↪conveniently
                # 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.

```
#     w[t-1] ----> w[t] ---> w[t+1]
#        |  \        |  \        |   \
#        | x[t-1] ----> x[t] ----> x[t+1]
#        |   /       |   /       |   /
#        V  /        V  /        V  /
#     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)

(continued from previous 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)[..., None]):
                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          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)

(continued from previous 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)))
```

(continues on next page)

(continued from previous page)

```
    # 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()
    kernel = {"nuts": NUTS, "hmc": HMC}[args.kernel](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, sequences, lengths, args=args)
    mcmc.print_summary()
    logger.info("\nMCMC elapsed time: {}".format(time.time() - start))


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="HMC for HMMs")
    parser.add_argument(
        "-m",
        "--model",
        default="1",
        type=str,
        help="one of: {}".format(", ".join(sorted(models.keys()))),
    )
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("-d", "--hidden-dim", default=16, type=int)
    parser.add_argument("-t", "--truncate", type=int)
    parser.add_argument("--num-sequences", type=int)
    parser.add_argument("--kernel", default="nuts", type=str)
    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)
```

CHAPTER

NINETEEN

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

1. Kery, M., & Schaub, M. (2011). Bayesian population analysis using WinBUGS: a hierarchical perspective. Academic Press.
2. Lebreton, J.D., Burnham, K.P., Clobert, J., & Anderson, D.R. (1992). Modeling survival and testing biological hypotheses using marked animals: a unified approach with case studies. Ecological monographs, 62(1), 67-118.
3. Nichols, Pollock, Hines (1984) The use of a robust capture-recapture design in small mammal population studies: A field example with Microtus pennsylvanicus. Acta Theriologica 29:357-365.
4. Cormack, R.M., 1964. Estimates of survival from the sighting of marked animals. Biometrika 51, 429-438.
5. Jolly, G.M., 1965. Explicit estimates from capture-recapture data with both death and immigration-stochastic model. Biometrika 52, 225-247.
6. Seber, G.A.F., 1965. A note on the multiple recapture census. Biometrika 52, 249-259.
7. https://github.com/stan-dev/example-models/tree/master/BPA/Ch.07
8. http://pyro.ai/examples/capture_recapture.html


```
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)

(continued from previous page)

```
from numpyro.examples.datasets import DIPPER_VOLE, load_dataset
from numpyro.infer import HMC, MCMC, NUTS
from numpyro.infer.reparam import LocScaleReparam
```

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.

```
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)
                # 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
                )

        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.

```
def model_2(capture_history, sex):
    N, T = capture_history.shape
    rho = numpyro.sample("rho", dist.Uniform(0.0, 1.0))  # recapture probability

    def transition_fn(carry, y):
```

(continues on next page)

(continued from previous 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)
                # 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
                )

        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 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)

(continued from previous page)

```
            )
        phi_t = expit(phi_logit_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
                )

        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 fourth model variant we include group-level fixed effects for sex (male, female).

```
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)),
```

(continues on next page)

(continued from previous page)

```
                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 final model variant we include both fixed group effects and fixed time effects for the survival probability phi: logit(phi_t) = beta_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.

```
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)

(continued from previous page)

```
        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)

(continued from previous 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.".
↪format(
            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", 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(
        "--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)
```

CHAPTER

TWENTY

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
2. dynesty's Gaussian Shells demo: https://github.com/joshspeagle/dynesty/blob/master/demos/Examples%20–%20Gaussian%20Sh

```
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()
        )  # a dummy sample to initialize the samplers

    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)

(continued from previous 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(
```

(continues on next page)

(continued from previous page)

```
        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("0.10.1")
    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)
```

CHAPTER

TWENTYONE

BAYESIAN IMPUTATION FOR 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]:
```
import numpyro
from jax import numpy as jnp, random, ops
from jax.scipy.special import expit
from numpyro import distributions as dist, sample
from numpyro.infer.mcmc import MCMC
from numpyro.infer.hmc import NUTS
from math import inf
from graphviz import Digraph

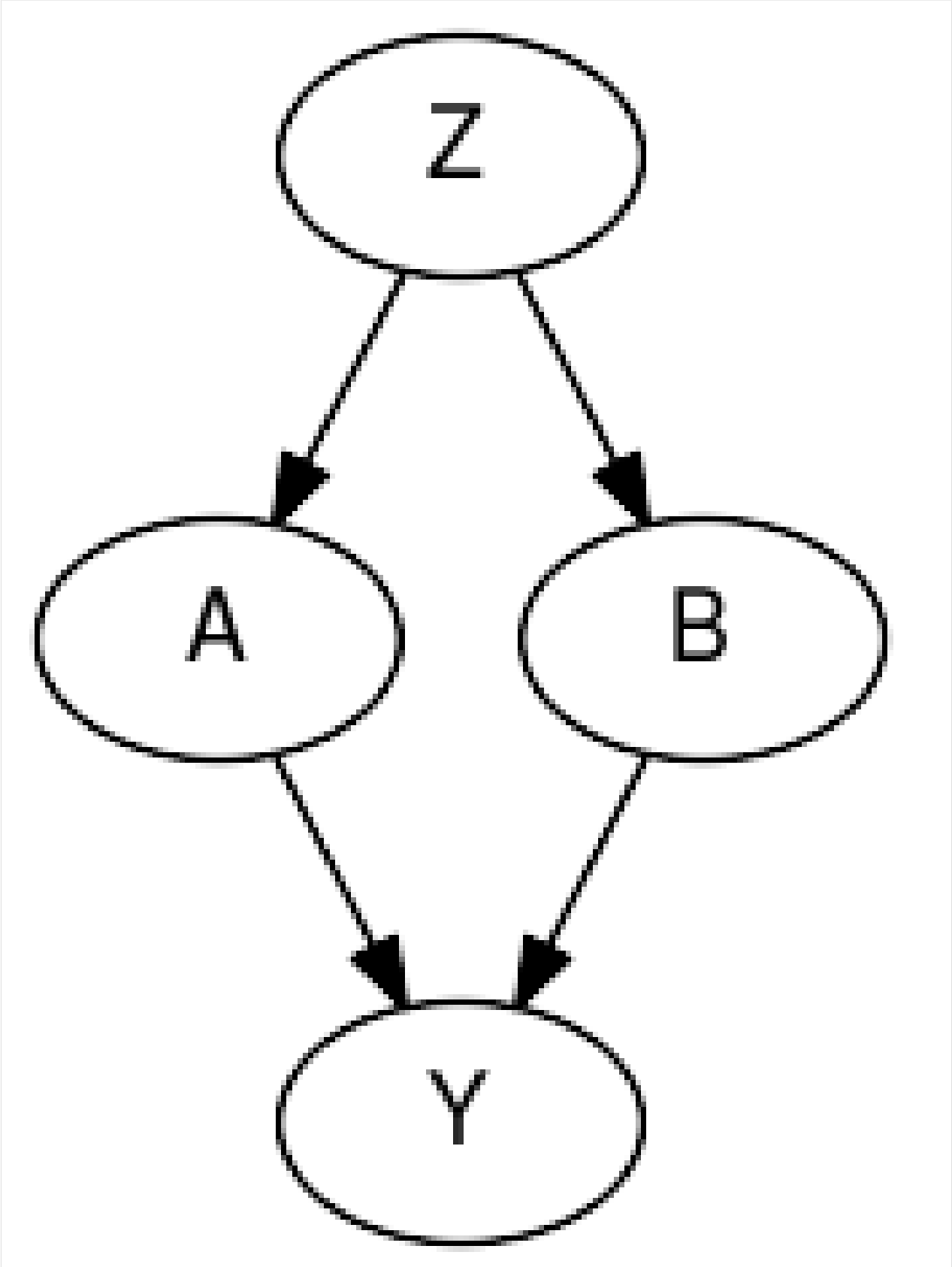
simkeys = random.split(random.PRNGKey(0), 10)
nsim = 5000
mcmc_key = random.PRNGKey(1)
```

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")
dot.edges(["ZA", "ZB", "AY", "BY"])
dot
```

[2]:



[3]:
```
b_A = 0.25
b_B = 0.25
s_Y = 0.25
Z = random.normal(simkeys[0], (nsim,))
A = random.bernoulli(simkeys[1], expit(Z))
B = random.bernoulli(simkeys[2], expit(Z))
Y = A * b_A + B * b_B + s_Y * random.normal(simkeys[3], (nsim,))
```

21.1 MAR conditional on outcome

According to Rubin's classic definitions there are 3 distinct of missing data mechanisms:

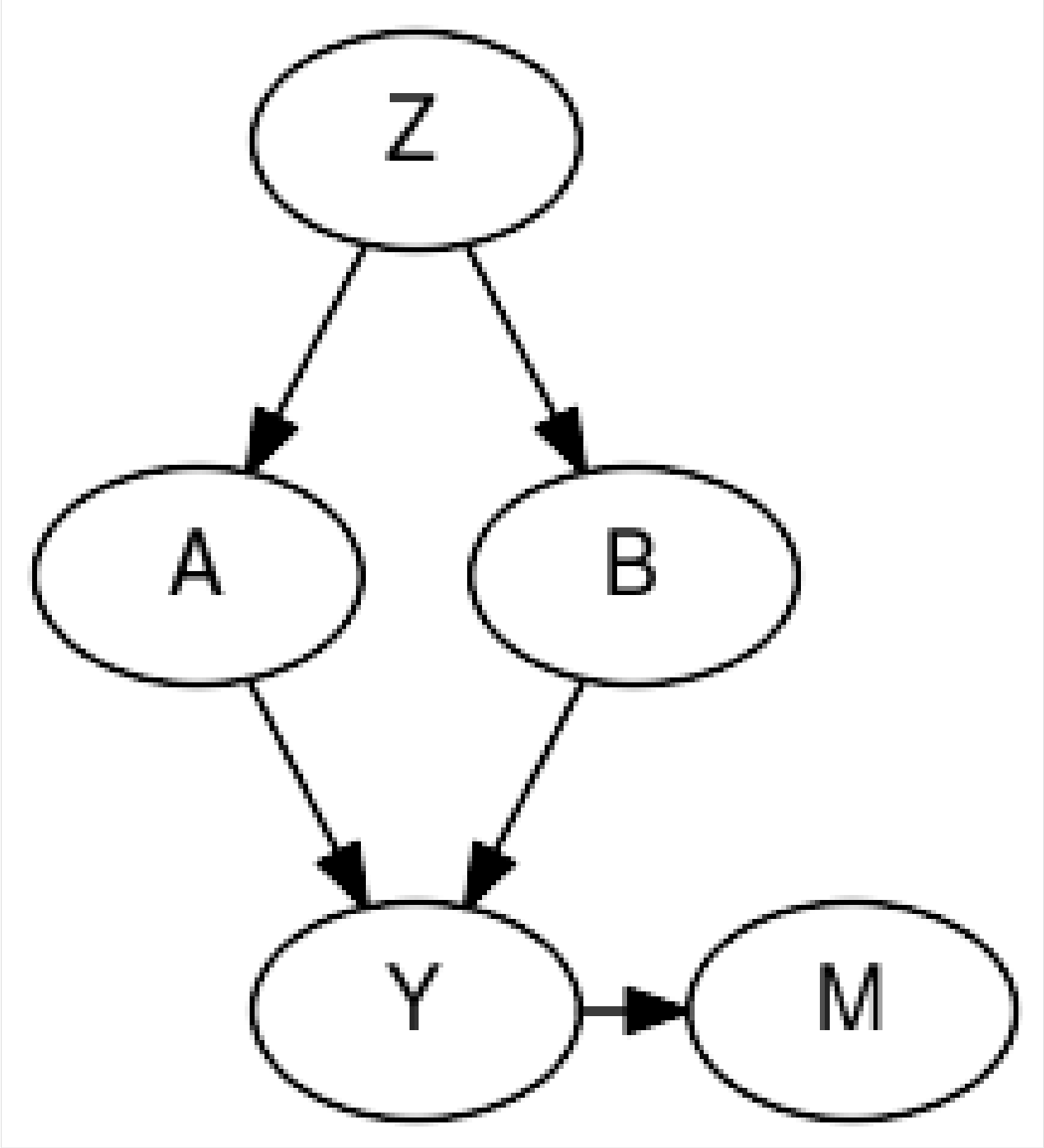
1. missing completely at random (MCAR)
2. missing at random, conditional on observed data (MAR)
3. missing not at random, even after conditioning on observed data (MNAR)

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).

[4]:
```
dot_mnar_y = Digraph()
with dot_mnar_y.subgraph() as s:
    s.attr(rank="same")
    s.node("Y")
    s.node("M")
dot_mnar_y.node("A")
dot_mnar_y.node("B")
dot_mnar_y.node("Z")
dot_mnar_y.node("M")
dot_mnar_y.edges(["YM", "ZA", "ZB", "AY", "BY"])
dot_mnar_y
```

[4]:



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 availalbe only after they have visited the clinic. This makes the missingness in A (history of heart disease), dependent on the outcome (visiting the clinic).

[5]:
```
A_isobs = random.bernoulli(simkeys[4], expit(3 * (Y - Y.mean())))
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]
```

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 unneccesary. 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()
```

```
sample: 100%|██████████| 1000/1000 [00:02<00:00, 348.50it/s, 3 steps of␣
↪size 4.27e-01. acc. prob=0.94]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       b_A      0.30      0.01      0.30      0.29      0.31    500.83      1.00
       b_B      0.28      0.01      0.28      0.27      0.29    546.34      1.00
       s_Y      0.25      0.00      0.25      0.24      0.25    559.55      1.00

Number of divergences: 0
```

[8]:
```
def impmodel(A, B, Y):
    ntotal = A.shape[0]
    A_isobs = A >= 0
```

(continues on next page)

(continued from previous 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(impmodel)
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]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       b_A      0.25      0.01      0.25      0.24      0.27    447.79      1.01
       b_B      0.25      0.01      0.25      0.24      0.26    570.66      1.01
     b_B_A      0.74      0.08      0.74      0.60      0.86    316.36      1.00
      mu_A     -0.39      0.06     -0.39     -0.48     -0.29    290.86      1.00
       s_Y      0.25      0.00      0.25      0.25      0.25    527.97      1.00

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).

21.2 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).

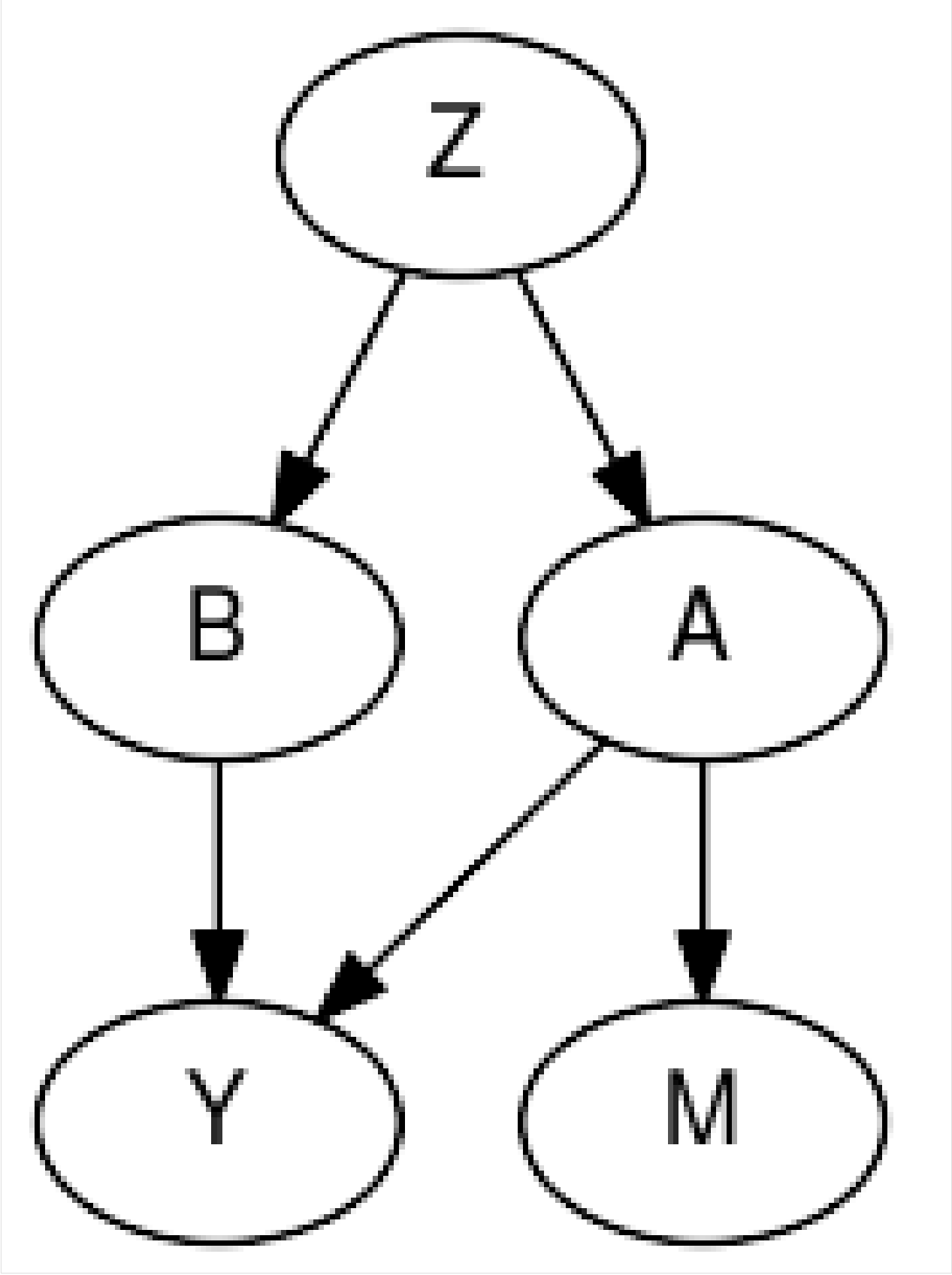
As an example consider patients who have cancer:

- 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.

[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
```

[10]:



[11]:
```
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]
```

[12]:
```
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
```

[13]:
```
impkernel = NUTS(impmodel)
impmcmc = MCMC(impkernel, num_warmup=250, num_samples=750)
impmcmc.run(mcmc_key, Aobs, B, Y)
impmcmc.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.

[14]:
```
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)
```

[15]:
```
impmisskernel = NUTS(impmissmodel)
impmissmcmc = MCMC(impmisskernel, num_warmup=250, num_samples=750)
impmissmcmc.run(mcmc_key, Aobs, B, Y)
impmissmcmc.print_summary()
```

```
sample: 100%|██████████| 1000/1000 [00:09<00:00, 106.81it/s, 7 steps of␣
↪size 2.86e-01. acc. prob=0.91]
```

(continues on next page)

(continued from previous page)

```
              mean       std    median      5.0%     95.0%     n_eff     r_hat
       b_A      0.26      0.01      0.26      0.24      0.27    267.57      1.00
       b_B      0.25      0.01      0.25      0.24      0.26    537.10      1.00
     b_B_A      0.74      0.07      0.74      0.62      0.84    421.54      1.00
      mu_A     -0.45      0.08     -0.45     -0.58     -0.31    241.11      1.00
 p_Aobs[0]      0.10      0.01      0.10      0.09      0.11    451.90      1.00
 p_Aobs[1]      0.86      0.03      0.86      0.82      0.91    244.47      1.00
       s_Y      0.25      0.00      0.25      0.24      0.25    375.51      1.00

Number of divergences: 0
```

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)

CHAPTER

TWENTYTWO

TIME SERIES FORECASTING

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.

[ ]:
```
!pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
```

[1]:
```
import os

import matplotlib.pyplot as plt
import pandas as pd
from IPython.display import set_matplotlib_formats

import jax.numpy as jnp
from jax import random

import numpyro
import numpyro.distributions as dist
from numpyro.contrib.control_flow import scan
from numpyro.diagnostics import autocorrelation, hpdi
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("0.10.1")
```

22.1 Data

First, lets import and take a look at the dataset.

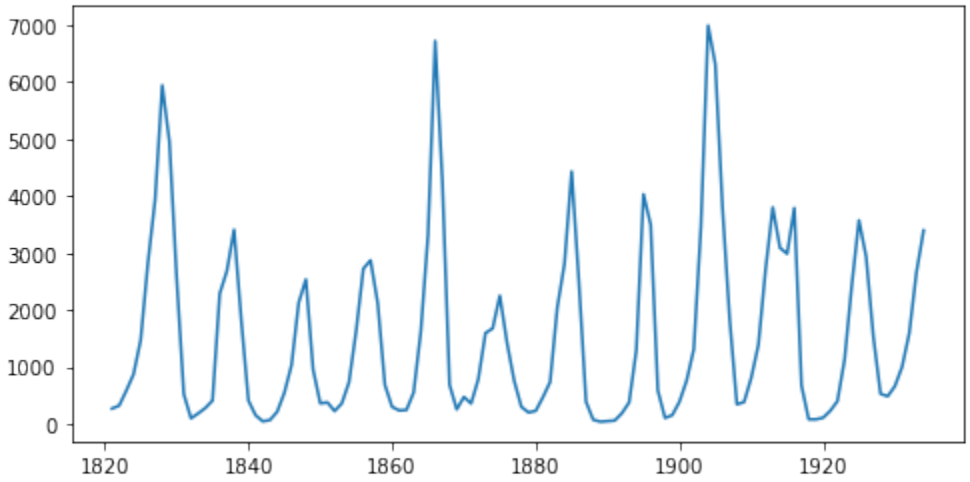
[2]:
```
URL = "https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/
↪datasets/lynx.csv"
lynx = pd.read_csv(URL, index_col=0)
data = lynx["value"].values
print("Length of time series:", data.shape[0])
plt.figure(figsize=(8, 4))
```

(continues on next page)

(continued from previous 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.

[3]:
```
y_train, y_test = jnp.array(data[:80], dtype=jnp.float32), data[80:]
```

22.2 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:

$$\text{exp-val}_t = \text{level}_{t-1} + \text{coef-trend} \times \text{level}_{t-1}^{\text{pow-trend}} + \text{s}_t \times \text{level}_{t-1}^{\text{pow-season}}, \tag{22.1}$$

$$\sigma_t = \sigma \times \text{exp-val}_t^{\text{powx}} + \text{offset}, \tag{22.2}$$

$$y_t \sim \text{StudentT}(\nu, \text{exp-val}_t, \sigma_t) \tag{22.3}$$

, where `level` and `s` follows the following recursion rules:

$$\text{level-p} = \begin{cases} y_t - \text{s}_t \times \text{level}_{t-1}^{\text{pow-season}} & \text{if } t \le \text{seasonality}, \\ \text{Average}\left[y(t - \text{seasonality} + 1), \ldots, y(t)\right] & \text{otherwise}, \end{cases} \tag{22.4}$$

$$\text{level}_t = \text{level-sm} \times \text{level-p} + (1 - \text{level-sm}) \times \text{level}_{t-1}, \tag{22.5}$$

$$\text{s}_{t+\text{seasonality}} = \text{s-sm} \times \frac{y_t - \text{level}_t}{\text{level}_{t-1}^{\text{pow-trend}}} + (1 - \text{s-sm}) \times \text{s}_t. \tag{22.6}$$

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 `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 `coef_trend`, and b is `pow_trend`. Note that when $b \sim 0$, the trend is linear with a is the slope, and when $b \sim 1$, 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 `level_sm` and `s_sm` are used to make the transition smoothly.
- When `powx` is near 0, the error σ_t will be nearly constant while when `powx` is near 1, the error will be propotional 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 *NumPyro* primitives. In NumPyro, we use the primitive `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.

[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
    # https://github.com/cbergmeir/Rlgt/blob/master/Rlgt/R/rlgtcontrol.R
    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, a_min=0)
        # use expected vale 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)
        )
```

(continues on next page)

(continued from previous page)

```
        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, a_min=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.

22.3 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:\n")
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 13  6 25 74 53 42 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()
```

```
                  mean       std    median      5.0%     95.0%     n_eff     r_hat
     coef_trend     32.33    123.99     12.07    -91.43    157.37   1307.74      1.00
      init_s[0]     84.35    105.16     61.08    -59.71    232.37   4053.35      1.00
      init_s[1]    -21.48     72.05    -26.11   -130.13     94.34   1038.51      1.01
      init_s[2]     26.08     92.13     13.57   -114.83    156.16   1559.02      1.00
      init_s[3]    122.52    123.56    102.67    -59.39    305.43   4317.17      1.00
      init_s[4]    443.91    254.12    395.89     69.08    789.27   3090.34      1.00
      init_s[5]   1163.56    491.37   1079.23    481.92   1861.90   1562.40      1.00
      init_s[6]   1968.70    649.68   1860.04    902.00   2910.49   1974.42      1.00
      init_s[7]   3652.34   1107.27   3505.37   1967.67   5383.26   1669.91      1.00
      init_s[8]   2593.04    831.42   2452.27   1317.67   3858.55   1805.87      1.00
      init_s[9]    947.28    422.29    885.72    311.39   1589.56   3355.27      1.00
     init_s[10]     44.09    102.92     28.38   -105.25    203.73   1367.99      1.00
     init_s[11]     -2.25     52.92     -2.71    -86.51     72.90    611.35      1.01
     init_s[12]    -12.22     64.98    -13.67   -110.07     85.65    892.13      1.01
     init_s[13]     74.43    106.48     53.13    -79.73    225.92    658.08      1.01
     init_s[14]    332.98    255.28    281.72    -11.18    697.96   3685.55      1.00
     init_s[15]    965.80    389.00    893.29    373.98   1521.59   2575.80      1.00
     init_s[16]   1261.12    469.99   1191.83    557.98   1937.38   2300.48      1.00
     init_s[17]   1372.34    559.14   1274.21    483.96   2151.94   2007.79      1.00
     init_s[18]    611.20    313.13    546.56    167.97   1087.74   2854.06      1.00
     init_s[19]     17.81     87.79      8.93   -118.64    143.96   5689.95      1.00
     init_s[20]    -31.84     66.70    -25.15   -146.89     58.97   3083.09      1.00
     init_s[21]    -14.01     44.74     -5.80    -86.03     42.99   2118.09      1.00
     init_s[22]     -2.26     42.99     -2.39    -61.40     66.34   3022.51      1.00
     init_s[23]     43.53     90.60     29.14    -82.56    167.89   3230.17      1.00
     init_s[24]    509.69    326.73    453.22     44.04    975.15   2087.02      1.00
     init_s[25]    919.23    431.15    837.03    284.54   1563.05   3257.27      1.00
     init_s[26]   1783.39    697.15   1660.09    720.83   2811.83   1730.70      1.00
     init_s[27]   1247.60    461.26   1172.88    544.44   1922.68   1573.09      1.00
     init_s[28]    217.92    169.08    191.38    -29.78    456.65   4899.06      1.00
     init_s[29]     -7.43     82.23    -12.99   -133.20    118.31   7588.25      1.00
     init_s[30]     -6.69     86.99    -17.03   -130.99    125.43   1687.37      1.00
     init_s[31]    -35.24     71.31    -35.75   -148.09     76.96   5462.22      1.00
     init_s[32]     -8.63     80.39    -14.95   -138.34    113.89   6626.25      1.00
```

(continues on next page)

(continued from previous page)

```
      init_s[33]    117.38    148.71     91.69    -78.12    316.69   2424.57      1.00
      init_s[34]    502.79    297.08    448.55     87.88    909.45   1863.99      1.00
      init_s[35]   1064.57    445.88    984.10    391.61   1710.35   2584.45      1.00
      init_s[36]   1849.48    632.44   1763.31    861.63   2800.25   1866.47      1.00
      init_s[37]   1452.62    546.57   1382.62    635.28   2257.04   2343.09      1.00
        level_sm      0.00      0.00      0.00      0.00      0.00   7829.05      1.00
              nu     12.17      4.73     12.31      5.49     19.99   4979.84      1.00
     offset_sigma    31.82     31.84     22.43      0.01     73.13   1442.32      1.00
       pow_season     0.09      0.04      0.09      0.01      0.15   1091.99      1.00
   pow_trend_beta     0.26      0.18      0.24      0.00      0.52    199.20      1.01
             powx     0.62      0.13      0.62      0.40      0.84   2476.16      1.00
             s_sm     0.08      0.09      0.05      0.00      0.18   5866.57      1.00
            sigma     9.67      9.87      6.61      0.35     20.60   2376.07      1.00

Number of divergences: 4568
CPU times: user 1min 17s, sys: 108 ms, total: 1min 18s
Wall time: 41.2 s
```

22.4 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.

[7]:
```
predictive = Predictive(sgt, samples, return_sites=["y_forecast"])
forecast_marginal = predictive(random.PRNGKey(1), y_train, seasonality=38, future=34)[
    "y_forecast"
]
```

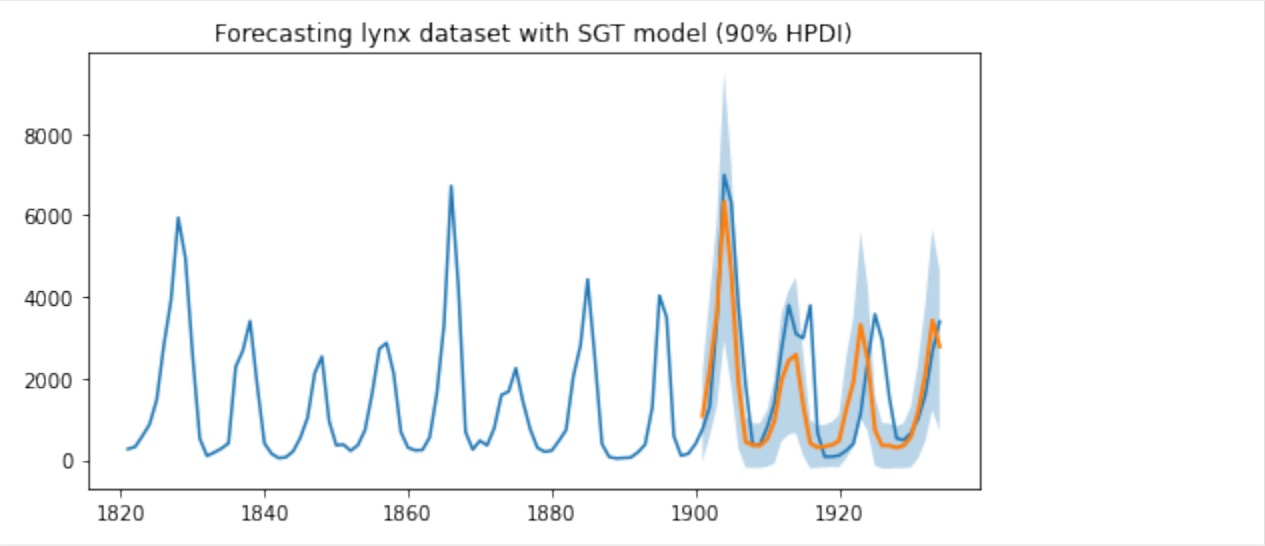
Let's get sMAPE, root mean square error of the prediction, and visualize the result with the mean prediction and the 90% highest posterior density interval (HPDI).

[8]:
```
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("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.

[9]:
```
plt.figure(figsize=(8, 4))
plt.plot(lynx["time"], data)
t_future = lynx["time"][80:]
hpd_low, hpd_high = hpdi(forecast_marginal)
plt.plot(t_future, y_pred, lw=2)
plt.fill_between(t_future, hpd_low, hpd_high, alpha=0.3)
plt.title("Forecasting lynx dataset with SGT model (90% HPDI)")
plt.show()
```



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.

22.5 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.

22.6 References

[1] `Rlgt: Bayesian Exponential Smoothing Models with Trend Modifications`, Slawek Smyl, Christoph Bergmeir, Erwin Wibowo, To Wang Ng, Trustees of Columbia University

CHAPTER

TWENTYTHREE

ORDINAL REGRESSION

Some data are discrete but intrinsically **ordered**, these are called **ordinal** data. One example is the likert 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

References: 1. Betancourt, M. (2019), "Ordinal Regression", (https://betanalpha.github.io/assets/case_studies/ordinal_regression.html)

[1]:
```
# !pip install -q numpyro@git+https://github.com/pyro-ppl/numpyro
```

[2]:
```
from jax import numpy as np, random
import numpyro
from numpyro import sample, handlers
from numpyro.distributions import (
    Categorical,
    Dirichlet,
    ImproperUniform,
    Normal,
    OrderedLogistic,
    TransformedDistribution,
    constraints,
    transforms,
)
from numpyro.infer import MCMC, NUTS
from numpyro.infer.reparam import TransformReparam

import pandas as pd
import seaborn as sns

assert numpyro.__version__.startswith("0.10.1")
```

23.1 Data Generation

First, generate some data with ordinal structure

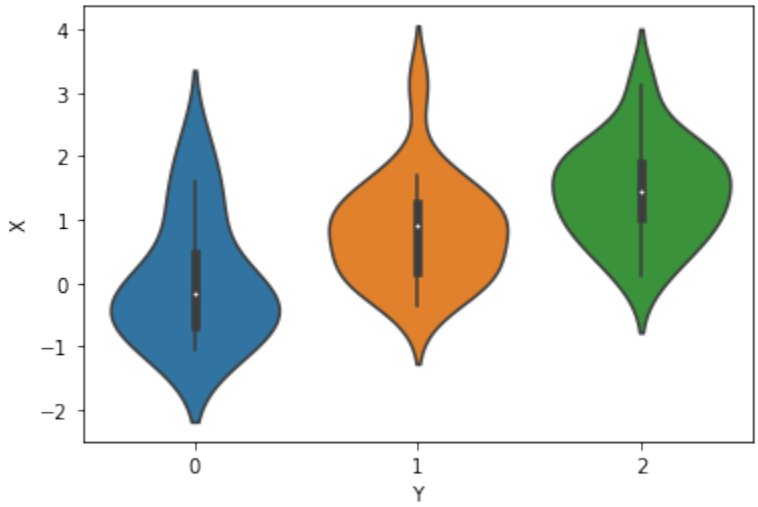
[3]:
```
simkeys = random.split(random.PRNGKey(1), 2)
nsim = 50
nclasses = 3
Y = Categorical(logits=np.zeros(nclasses)).sample(simkeys[0], sample_shape=(nsim,))
X = Normal().sample(simkeys[1], sample_shape=(nsim,))
X += Y

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:
1    19
2    16
0    15
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
```

[4]:
```
sns.violinplot(x="Y", y="X", data=df);
```



23.2 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 `ImproperUnifrom` distribution to introduce a parameter with an arbitrary support that is otherwise completely uninformative, and then add an `ordered_vector` constraint.

[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)


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()
```

```
sample: 100%|| 1000/1000 [00:03<00:00, 258.56it/s, 7 steps of␣
↪size 5.02e-01. acc. prob=0.94]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
   b_X_eta      1.44      0.37      1.42      0.83      2.05    349.38      1.01
    c_y[0]     -0.10      0.38     -0.10     -0.71      0.51    365.63      1.00
    c_y[1]      2.15      0.49      2.13      1.38      2.99    376.45      1.01

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

[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)

(continued from previous 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()
```

```
sample: 100%|██████████| 1000/1000 [00:03<00:00, 256.41it/s, 7 steps of␣
↪size 5.31e-01. acc. prob=0.92]
```

```
               mean       std    median      5.0%     95.0%     n_eff     r_hat
   b_X_eta      1.23      0.31      1.23      0.64      1.68    501.31      1.01
    c_y[0]     -0.24      0.34     -0.23     -0.76      0.38    492.91      1.00
    c_y[1]      1.77      0.40      1.76      1.11      2.42    628.46      1.00

Number of divergences: 0
```

23.3 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.

[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()
```

```
sample: 100%|██████████| 1000/1000 [00:04<00:00, 244.78it/s, 7 steps of␣
↪size 5.54e-01. acc. prob=0.93]
```

```
              mean       std    median      5.0%     95.0%     n_eff     r_hat
   b_X_eta      1.40      0.34      1.41      0.86      1.98    300.30      1.03
    c_y[0]     -0.03      0.35     -0.03     -0.57      0.54    395.98      1.00
    c_y[1]      2.06      0.47      2.04      1.26      2.83    475.16      1.01

Number of divergences: 0
```

23.4 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 SimplexToOrderedTransform. This approach should be advantageous when there is a need for strong prior knowledge to be added to our Ordinal model, eg, 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)
```

(continues on next page)

(continued from previous page)

```
# with exclude_deterministic=False, we will also show the ordinal probabilities sampled␣
↪from Dirichlet (vis. `c_y_base`)
mcmc.print_summary(exclude_deterministic=False)
```

```
sample: 100%|| 1000/1000 [00:05<00:00, 193.88it/s, 7 steps of␣
↪size 7.00e-01. acc. prob=0.93]
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
    b_X_eta      1.01      0.26      1.01      0.59      1.42    388.46      1.00
     c_y[0]     -0.42      0.26     -0.42     -0.88     -0.05    491.73      1.00
     c_y[1]      1.34      0.29      1.34      0.86      1.80    617.53      1.00
c_y_base[0]      0.40      0.06      0.40      0.29      0.49    488.71      1.00
c_y_base[1]      0.39      0.06      0.39      0.29      0.48    523.65      1.00
c_y_base[2]      0.21      0.05      0.21      0.13      0.29    610.33      1.00

Number of divergences: 0
```

CHAPTER

TWENTYFOUR

BAYESIAN IMPUTATION

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.

[ ]:
```
!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
from jax import random
from jax.scipy.special import expit

import numpyro
from numpyro import distributions as dist
from numpyro.distributions import constraints
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("0.10.1")
```

24.1 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_df = pd.read_csv(
    "https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv"
)
train_df.info()
train_df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
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	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

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:

[3]:
```
for col in ["Survived", "Pclass", "Sex", "SibSp", "Parch", "Embarked"]:
    print(train_df[col].value_counts(), end="\n\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
```

24.2 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.

[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:

[5]:
```
train_df.Name.str.split(", ").str.get(1).str.split(" ").str.get(0).value_counts()
```

[5]:
```
Mr.          517
Miss.        182
Mrs.         125
Master.       40
Dr.            7
Rev.           6
Mlle.          2
Col.           2
Major.         2
Lady.          1
Sir.           1
the            1
Ms.            1
Capt.          1
Mme.           1
Jonkheer.      1
Don.           1
Name: Name, dtype: int64
```

We will make a new column `Title`, where rare titles are merged into one group `Misc.`.

[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.

[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)

(continued from previous 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)])
```

24.3 Modelling

First, we want to note that in NumPyro, the following models

```
def model1a():
    x = numpyro.sample("x", dist.Normal(0, 1).expand([10]))
```

and

```
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 `dist.Normal(0, 1)` prior,
- and the same log densities `dist.Normal(0, 1).log_prob(x)`.

Now, assume that we observed the last 6 values of `x` (non-observed entries take value `NaN`), the typical model will be

```
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`,

```
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.

[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)

(continued from previous 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`.

24.4 Sampling

We will use MCMC with NUTS kernel to sample both regression coefficients and imputed values.

[9]:
```
mcmc = MCMC(NUTS(model), num_warmup=1000, num_samples=1000)
mcmc.run(random.PRNGKey(0), **data, survived=survived)
mcmc.print_summary()
```

```
sample: 100%|██████████| 2000/2000 [00:15<00:00, 132.15it/s, 63 steps of␣
↪size 5.68e-02. acc. prob=0.95]

                   mean       std    median      5.0%     95.0%     n_eff     r_hat
              a      0.12      0.82      0.11     -1.21      1.49    887.50      1.00
  age_impute[0]      0.20      0.84      0.18     -1.22      1.53   1346.09      1.00
  age_impute[1]     -0.06      0.86     -0.08     -1.41      1.26   1057.70      1.00
  age_impute[2]      0.38      0.73      0.39     -0.80      1.58   1570.36      1.00
  age_impute[3]      0.25      0.84      0.23     -0.99      1.86   1027.43      1.00
  age_impute[4]     -0.63      0.91     -0.59     -1.99      0.87   1183.66      1.00
  age_impute[5]      0.21      0.89      0.19     -1.02      1.97   1456.79      1.00
  age_impute[6]      0.45      0.82      0.46     -0.90      1.73   1239.22      1.00
  age_impute[7]     -0.62      0.86     -0.62     -2.13      0.72   1406.09      1.00
  age_impute[8]     -0.13      0.90     -0.14     -1.64      1.38   1905.07      1.00
  age_impute[9]      0.24      0.84      0.26     -1.06      1.77   1471.12      1.00
```

(continues on next page)

(continued from previous page)

age_impute[10]	0.20	0.89	0.21	-1.26	1.65	1588.79	1.00
age_impute[11]	0.17	0.91	0.19	-1.59	1.48	1446.52	1.00
age_impute[12]	-0.65	0.89	-0.68	-2.12	0.77	1457.47	1.00
age_impute[13]	0.21	0.85	0.18	-1.24	1.53	1057.77	1.00
age_impute[14]	0.05	0.92	0.05	-1.40	1.65	1207.08	1.00
age_impute[15]	0.37	0.94	0.37	-1.02	1.98	1326.55	1.00
age_impute[16]	-1.74	0.26	-1.74	-2.13	-1.32	1320.08	1.00
age_impute[17]	0.21	0.89	0.22	-1.30	1.60	1545.73	1.00
age_impute[18]	0.18	0.90	0.18	-1.26	1.58	2013.12	1.00
age_impute[19]	-0.67	0.86	-0.66	-1.97	0.85	1499.50	1.00
age_impute[20]	0.23	0.89	0.27	-1.19	1.71	1712.24	1.00
age_impute[21]	0.21	0.87	0.20	-1.11	1.68	1400.55	1.00
age_impute[22]	0.19	0.90	0.18	-1.26	1.63	1400.37	1.00
age_impute[23]	-0.15	0.85	-0.15	-1.57	1.24	1205.10	1.00
age_impute[24]	-0.71	0.89	-0.73	-2.05	0.82	1085.52	1.00
age_impute[25]	0.20	0.85	0.19	-1.20	1.62	1708.01	1.00
age_impute[26]	0.21	0.88	0.21	-1.20	1.68	1363.75	1.00
age_impute[27]	-0.69	0.91	-0.73	-2.20	0.77	1224.06	1.00
age_impute[28]	0.60	0.77	0.60	-0.61	1.95	1312.44	1.00
age_impute[29]	0.20	0.89	0.17	-1.23	1.71	938.19	1.00
age_impute[30]	0.24	0.87	0.23	-1.14	1.60	1324.50	1.00
age_impute[31]	-1.72	0.26	-1.72	-2.11	-1.28	1425.46	1.00
age_impute[32]	0.44	0.77	0.43	-0.83	1.58	1587.41	1.00
age_impute[33]	0.34	0.89	0.32	-1.14	1.73	1375.14	1.00
age_impute[34]	-1.72	0.26	-1.71	-2.11	-1.26	1007.71	1.00
age_impute[35]	-0.45	0.90	-0.47	-2.06	0.92	1329.44	1.00
age_impute[36]	0.30	0.84	0.30	-1.03	1.73	1080.80	1.00
age_impute[37]	0.33	0.88	0.32	-1.10	1.81	1033.30	1.00
age_impute[38]	0.33	0.76	0.35	-0.94	1.56	1550.68	1.00
age_impute[39]	0.19	0.93	0.21	-1.32	1.82	1203.79	1.00
age_impute[40]	-0.67	0.88	-0.69	-1.94	0.88	1382.98	1.00
age_impute[41]	0.17	0.89	0.14	-1.30	1.43	1438.18	1.00
age_impute[42]	0.23	0.82	0.25	-1.12	1.48	1499.59	1.00
age_impute[43]	0.22	0.82	0.21	-1.19	1.45	1236.67	1.00
age_impute[44]	-0.41	0.85	-0.42	-1.96	0.78	812.53	1.00
age_impute[45]	-0.36	0.89	-0.35	-2.01	0.94	1488.83	1.00
age_impute[46]	-0.33	0.91	-0.32	-1.76	1.27	1628.61	1.00
age_impute[47]	-0.71	0.85	-0.69	-2.12	0.64	1363.89	1.00
age_impute[48]	0.21	0.85	0.24	-1.21	1.64	1552.65	1.00
age_impute[49]	0.42	0.82	0.41	-0.83	1.77	754.08	1.00
age_impute[50]	0.26	0.86	0.24	-1.18	1.63	1155.49	1.00
age_impute[51]	-0.29	0.91	-0.30	-1.83	1.15	1212.08	1.00
age_impute[52]	0.36	0.85	0.34	-1.12	1.68	1190.99	1.00
age_impute[53]	-0.68	0.89	-0.65	-2.09	0.75	1104.75	1.00
age_impute[54]	0.27	0.90	0.25	-1.24	1.68	1331.19	1.00
age_impute[55]	0.36	0.89	0.36	-0.96	1.86	1917.52	1.00
age_impute[56]	0.38	0.86	0.40	-1.00	1.75	1862.00	1.00
age_impute[57]	0.01	0.91	0.03	-1.33	1.56	1285.43	1.00
age_impute[58]	-0.69	0.91	-0.66	-2.13	0.78	1438.41	1.00
age_impute[59]	-0.14	0.85	-0.16	-1.44	1.37	1135.79	1.00
age_impute[60]	-0.59	0.94	-0.61	-2.19	0.93	1222.88	1.00
age_impute[61]	0.24	0.92	0.25	-1.35	1.65	1341.95	1.00

(continues on next page)

(continued from previous page)

age_impute[62]	-0.55	0.91	-0.57	-2.01	0.96	753.85	1.00
age_impute[63]	0.21	0.90	0.19	-1.42	1.60	1238.50	1.00
age_impute[64]	-0.66	0.88	-0.68	-2.04	0.73	1214.85	1.00
age_impute[65]	0.44	0.78	0.48	-0.93	1.57	1174.41	1.00
age_impute[66]	0.22	0.94	0.20	-1.35	1.69	1910.00	1.00
age_impute[67]	0.33	0.76	0.34	-0.85	1.63	1210.24	1.00
age_impute[68]	0.31	0.84	0.33	-1.08	1.60	1756.60	1.00
age_impute[69]	0.26	0.91	0.25	-1.29	1.75	1155.87	1.00
age_impute[70]	-0.67	0.86	-0.70	-2.02	0.70	1186.22	1.00
age_impute[71]	-0.70	0.90	-0.69	-2.21	0.75	1469.35	1.00
age_impute[72]	0.24	0.86	0.24	-1.07	1.66	1604.16	1.00
age_impute[73]	0.34	0.72	0.35	-0.77	1.55	1144.55	1.00
age_impute[74]	-0.64	0.85	-0.64	-2.10	0.77	1513.79	1.00
age_impute[75]	0.41	0.78	0.42	-0.96	1.60	796.47	1.00
age_impute[76]	0.18	0.89	0.21	-1.19	1.74	755.44	1.00
age_impute[77]	0.21	0.84	0.22	-1.22	1.63	1371.73	1.00
age_impute[78]	-0.36	0.87	-0.33	-1.81	1.01	1017.23	1.00
age_impute[79]	0.20	0.84	0.19	-1.35	1.37	1677.57	1.00
age_impute[80]	0.23	0.84	0.24	-1.09	1.61	1545.61	1.00
age_impute[81]	0.28	0.90	0.32	-1.08	1.83	1735.91	1.00
age_impute[82]	0.61	0.80	0.60	-0.61	2.03	1353.67	1.00
age_impute[83]	0.24	0.89	0.26	-1.22	1.66	1165.03	1.00
age_impute[84]	0.21	0.91	0.21	-1.35	1.65	1584.00	1.00
age_impute[85]	0.24	0.92	0.21	-1.33	1.63	1271.37	1.00
age_impute[86]	0.31	0.81	0.30	-0.86	1.76	1198.70	1.00
age_impute[87]	-0.11	0.84	-0.10	-1.42	1.23	1248.38	1.00
age_impute[88]	0.21	0.94	0.22	-1.31	1.77	1082.82	1.00
age_impute[89]	0.24	0.86	0.23	-1.08	1.67	2141.98	1.00
age_impute[90]	0.41	0.84	0.45	-0.88	1.90	1518.73	1.00
age_impute[91]	0.21	0.86	0.20	-1.21	1.58	1723.50	1.00
age_impute[92]	0.21	0.84	0.20	-1.21	1.57	1742.44	1.00
age_impute[93]	0.22	0.87	0.23	-1.29	1.50	1359.74	1.00
age_impute[94]	0.22	0.87	0.18	-1.09	1.70	906.55	1.00
age_impute[95]	0.22	0.87	0.23	-1.16	1.65	1112.58	1.00
age_impute[96]	0.30	0.84	0.26	-1.18	1.57	1680.70	1.00
age_impute[97]	0.23	0.87	0.25	-1.22	1.63	1408.40	1.00
age_impute[98]	-0.36	0.91	-0.37	-1.96	1.03	1083.67	1.00
age_impute[99]	0.15	0.87	0.14	-1.22	1.61	1644.46	1.00
age_impute[100]	0.27	0.85	0.30	-1.27	1.45	1266.96	1.00
age_impute[101]	0.25	0.87	0.25	-1.19	1.57	1220.96	1.00
age_impute[102]	-0.29	0.85	-0.28	-1.70	1.10	1392.91	1.00
age_impute[103]	0.01	0.89	0.01	-1.46	1.39	1137.34	1.00
age_impute[104]	0.21	0.86	0.24	-1.16	1.64	1018.70	1.00
age_impute[105]	0.24	0.93	0.21	-1.14	1.90	1479.67	1.00
age_impute[106]	0.21	0.83	0.21	-1.09	1.55	1471.11	1.00
age_impute[107]	0.22	0.85	0.22	-1.09	1.64	1941.83	1.00
age_impute[108]	0.31	0.88	0.30	-1.10	1.76	1342.10	1.00
age_impute[109]	0.22	0.86	0.23	-1.25	1.56	1198.01	1.00
age_impute[110]	0.33	0.78	0.35	-0.95	1.62	1267.01	1.00
age_impute[111]	0.22	0.88	0.21	-1.11	1.71	1404.51	1.00
age_impute[112]	-0.03	0.90	-0.02	-1.38	1.55	1625.35	1.00
age_impute[113]	0.24	0.85	0.23	-1.17	1.62	1361.84	1.00

(continues on next page)

(continued from previous 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]	0.21	0.87	0.22	-1.20	1.64	1315.42	1.00
age_impute[119]	-0.69	0.88	-0.74	-2.00	0.90	1536.44	1.00
age_impute[120]	0.63	0.81	0.66	-0.65	1.89	899.61	1.00
age_impute[121]	0.27	0.90	0.26	-1.16	1.74	1744.32	1.00
age_impute[122]	0.18	0.87	0.18	-1.23	1.60	1625.58	1.00
age_impute[123]	-0.39	0.88	-0.38	-1.71	1.12	1266.58	1.00
age_impute[124]	-0.62	0.95	-0.63	-2.03	1.01	1600.28	1.00
age_impute[125]	0.23	0.88	0.23	-1.15	1.71	1604.27	1.00
age_impute[126]	0.18	0.91	0.18	-1.24	1.63	1527.38	1.00
age_impute[127]	0.32	0.85	0.36	-1.08	1.73	1074.98	1.00
age_impute[128]	0.25	0.88	0.25	-1.10	1.69	1486.79	1.00
age_impute[129]	-0.70	0.87	-0.68	-2.20	0.56	1506.55	1.00
age_impute[130]	0.21	0.88	0.20	-1.16	1.68	1451.63	1.00
age_impute[131]	0.22	0.87	0.23	-1.22	1.61	905.86	1.00
age_impute[132]	0.33	0.83	0.33	-1.01	1.66	1517.67	1.00
age_impute[133]	0.18	0.86	0.18	-1.19	1.59	1050.00	1.00
age_impute[134]	-0.14	0.92	-0.15	-1.77	1.24	1386.20	1.00
age_impute[135]	0.19	0.85	0.18	-1.22	1.53	1290.94	1.00
age_impute[136]	0.16	0.92	0.16	-1.35	1.74	1767.36	1.00
age_impute[137]	-0.71	0.90	-0.68	-2.24	0.82	1154.14	1.00
age_impute[138]	0.18	0.91	0.16	-1.30	1.67	1160.90	1.00
age_impute[139]	0.24	0.90	0.24	-1.15	1.76	1289.37	1.00
age_impute[140]	0.41	0.80	0.39	-1.05	1.53	1532.92	1.00
age_impute[141]	0.27	0.83	0.29	-1.04	1.60	1310.29	1.00
age_impute[142]	-0.28	0.89	-0.29	-1.68	1.22	1088.65	1.00
age_impute[143]	-0.12	0.91	-0.11	-1.56	1.40	1324.74	1.00
age_impute[144]	-0.65	0.87	-0.63	-1.91	0.93	1672.31	1.00
age_impute[145]	-1.73	0.26	-1.74	-2.11	-1.26	1502.96	1.00
age_impute[146]	0.40	0.85	0.40	-0.85	1.84	1443.81	1.00
age_impute[147]	0.23	0.87	0.20	-1.37	1.49	1220.62	1.00
age_impute[148]	-0.70	0.88	-0.70	-2.08	0.87	1846.67	1.00
age_impute[149]	0.27	0.87	0.29	-1.11	1.76	1451.79	1.00
age_impute[150]	0.21	0.90	0.20	-1.10	1.78	1409.94	1.00
age_impute[151]	0.25	0.87	0.26	-1.21	1.63	1224.08	1.00
age_impute[152]	0.05	0.85	0.05	-1.42	1.39	1164.23	1.00
age_impute[153]	0.18	0.90	0.15	-1.19	1.72	1697.92	1.00
age_impute[154]	1.05	0.93	1.04	-0.24	2.84	1212.82	1.00
age_impute[155]	0.20	0.84	0.18	-1.18	1.54	1398.45	1.00
age_impute[156]	0.23	0.95	0.19	-1.19	1.87	1773.79	1.00
age_impute[157]	0.19	0.85	0.22	-1.13	1.64	1123.21	1.00
age_impute[158]	0.22	0.86	0.22	-1.18	1.60	1307.64	1.00
age_impute[159]	0.18	0.84	0.18	-1.09	1.59	1499.97	1.00
age_impute[160]	0.24	0.89	0.28	-1.23	1.65	1100.08	1.00
age_impute[161]	-0.45	0.88	-0.45	-1.86	1.05	1414.97	1.00
age_impute[162]	0.39	0.89	0.40	-1.00	1.87	1525.80	1.00
age_impute[163]	0.34	0.89	0.35	-1.14	1.75	1600.03	1.00
age_impute[164]	0.21	0.94	0.19	-1.13	1.91	1090.05	1.00
age_impute[165]	0.22	0.85	0.20	-1.11	1.60	1330.87	1.00

(continues on next page)

(continued from previous page)

```
age_impute[166]     -0.13      0.91     -0.15     -1.69      1.28   1284.90      1.00
age_impute[167]      0.22      0.89      0.24     -1.15      1.76   1261.93      1.00
age_impute[168]      0.20      0.90      0.18     -1.18      1.83   1217.16      1.00
age_impute[169]      0.07      0.89      0.05     -1.29      1.60   2007.16      1.00
age_impute[170]      0.23      0.90      0.24     -1.25      1.67    937.57      1.00
age_impute[171]      0.41      0.80      0.42     -0.82      1.82   1404.02      1.00
age_impute[172]      0.23      0.87      0.20     -1.33      1.51   2032.72      1.00
age_impute[173]     -0.44      0.88     -0.44     -1.81      1.08   1006.62      1.00
age_impute[174]      0.19      0.84      0.19     -1.11      1.63   1495.21      1.00
age_impute[175]      0.20      0.85      0.20     -1.17      1.63   1551.22      1.00
age_impute[176]     -0.43      0.92     -0.44     -1.83      1.21   1477.58      1.00
      age_mu[0]      0.19      0.04      0.19      0.12      0.26    749.16      1.00
      age_mu[1]     -0.54      0.07     -0.54     -0.66     -0.42    786.30      1.00
      age_mu[2]      0.43      0.08      0.42      0.31      0.55   1134.72      1.00
      age_mu[3]     -1.73      0.04     -1.73     -1.79     -1.65   1194.53      1.00
      age_mu[4]      0.85      0.17      0.85      0.58      1.13   1111.96      1.00
   age_sigma[0]      0.88      0.03      0.88      0.82      0.93    766.67      1.00
   age_sigma[1]      0.90      0.06      0.90      0.81      0.99    992.72      1.00
   age_sigma[2]      0.79      0.05      0.78      0.71      0.87    708.34      1.00
   age_sigma[3]      0.26      0.03      0.25      0.20      0.31    959.62      1.00
   age_sigma[4]      0.93      0.13      0.93      0.74      1.15   1092.88      1.00
          b_Age     -0.45      0.14     -0.44     -0.66     -0.22    744.95      1.00
  b_Embarked[0]     -0.28      0.58     -0.30     -1.28      0.64    496.51      1.00
  b_Embarked[1]      0.30      0.60      0.29     -0.74      1.20    495.25      1.00
  b_Embarked[2]      0.04      0.61      0.03     -0.93      1.02    482.67      1.00
     b_Parch[0]      0.45      0.57      0.47     -0.45      1.42    336.02      1.02
     b_Parch[1]      0.12      0.58      0.14     -0.91      1.00    377.61      1.02
     b_Parch[2]     -0.49      0.58     -0.45     -1.48      0.41    358.61      1.01
    b_Pclass[0]      1.22      0.57      1.24      0.33      2.17    371.15      1.00
    b_Pclass[1]      0.06      0.57      0.07     -0.84      1.03    369.58      1.00
    b_Pclass[2]     -1.18      0.57     -1.16     -2.18     -0.31    373.55      1.00
       b_Sex[0]      1.15      0.74      1.18     -0.03      2.31    568.65      1.00
       b_Sex[1]     -1.05      0.74     -1.02     -2.18      0.21    709.29      1.00
     b_SibSp[0]      0.28      0.66      0.26     -0.86      1.25    585.03      1.00
     b_SibSp[1]     -0.17      0.67     -0.18     -1.28      0.87    596.44      1.00
     b_Title[0]     -0.94      0.54     -0.96     -1.86     -0.11    437.32      1.00
     b_Title[1]     -0.33      0.61     -0.33     -1.32      0.60    570.32      1.00
     b_Title[2]      0.53      0.62      0.53     -0.52      1.46    452.87      1.00
     b_Title[3]      1.48      0.59      1.48      0.60      2.48    562.71      1.00
     b_Title[4]     -0.68      0.58     -0.66     -1.71      0.15    472.57      1.00

Number of divergences: 0
```

To double check that the assumption "age is correlated with title" is reasonable, let's look at the infered age by title. Recall that we performed standarization on `age`, so here we need to scale back to original domain.

[10]:
```
age_by_title = age_mean + age_std * mcmc.get_samples()["age_mu"].mean(axis=0)
dict(zip(title_cat.categories, age_by_title))
```

[10]:
```
{'Mr.': 32.434227,
 'Miss.': 21.763992,
 'Mrs.': 35.852997,
 'Master.': 4.6297398,
```

(continues on next page)

(continued from previous page)

```
 'Misc.': 42.081936}
```

The infered 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:

[11]:
```
train_df.groupby("Title")["Age"].mean()
```

[11]:
```
Title
Master.     4.574167
Misc.      42.384615
Miss.      21.773973
Mr.        32.368090
Mrs.       35.898148
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.

24.5 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.

[12]:
```
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 / confusion_matrix.sum(axis=1)
```

```
Accuracy: 0.8271605
```

[12]:
```
predict         0         1
actual
0        0.876138  0.198830
1        0.156648  0.748538
```

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.

[13]:
```
mcmc.run(random.PRNGKey(2), **data, survived=survived, bayesian_impute=False)
posterior_1 = mcmc.get_samples()
survived_pred_1 = Predictive(model, posterior_1)(random.PRNGKey(2), **data)["survived"]
survived_pred_1 = (survived_pred_1.mean(axis=0) >= 0.5).astype(jnp.uint8)
print("Accuracy:", (survived_pred_1 == survived).sum() / survived.shape[0])
confusion_matrix = pd.crosstab(
    pd.Series(survived, name="actual"), pd.Series(survived_pred_1, name="predict")
)
confusion_matrix / confusion_matrix.sum(axis=1)
confusion_matrix = pd.crosstab(
    pd.Series(survived, name="actual"), pd.Series(survived_pred_1, name="predict")
)
confusion_matrix / confusion_matrix.sum(axis=1)
```

```
sample: 100%|| 2000/2000 [00:11<00:00, 166.79it/s, 63 steps of size 7.18e-02. acc. prob=0.93]
```

```
Accuracy: 0.82042646
```

[13]:
```
predict         0         1
actual
0        0.872495  0.204678
1        0.163934  0.736842
```

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:

```
posterior.pop("age_impute")
survived_pred = Predictive(model, posterior)(random.PRNGKey(3), **new_data)
```

24.6 References

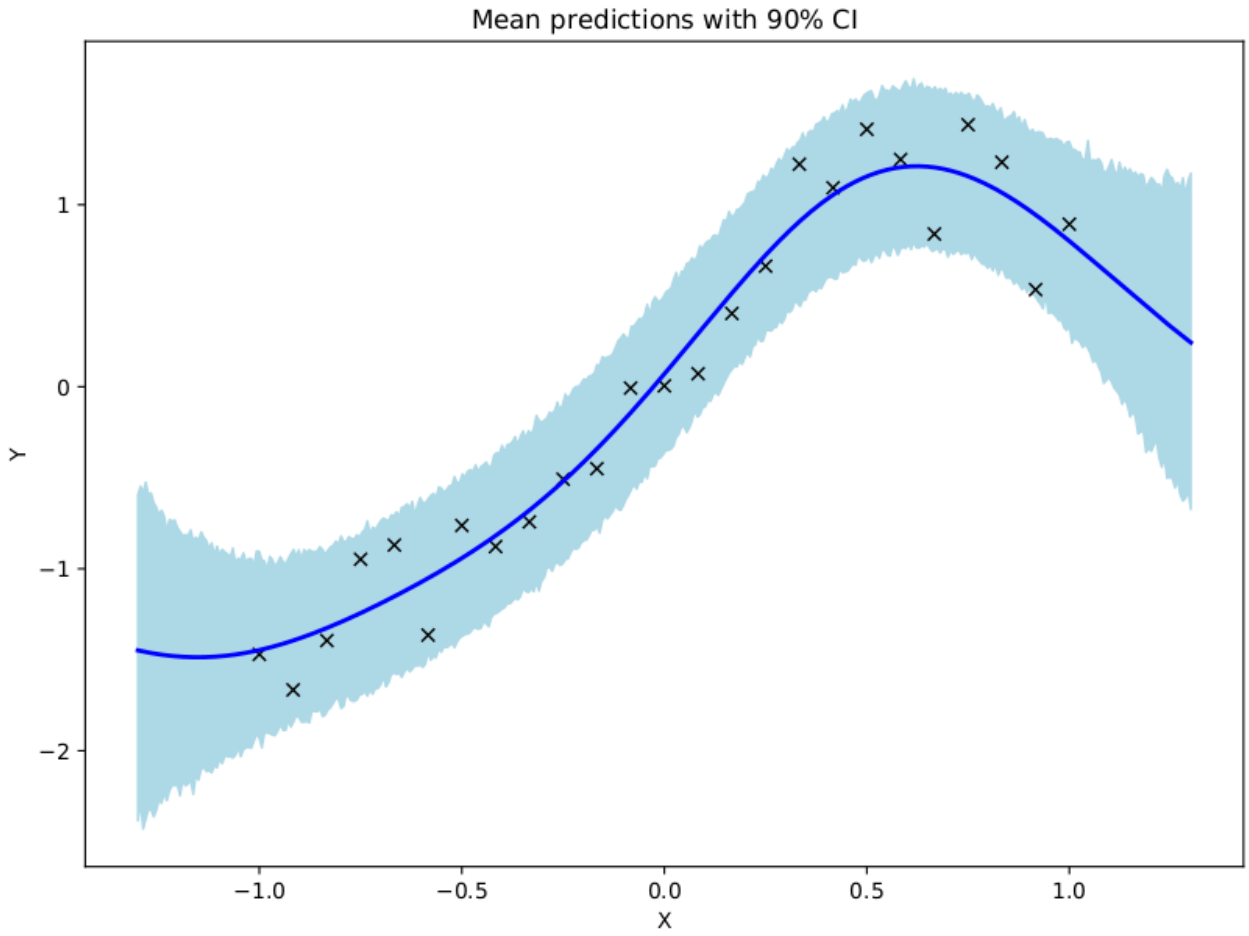
1. McElreath, R. (2016). Statistical Rethinking: A Bayesian Course with Examples in R and Stan.
2. Kaggle competition: Titanic: Machine Learning from Disaster

CHAPTER

TWENTYFIVE

EXAMPLE: GAUSSIAN PROCESS

In this example we show how to use NUTS to sample from the posterior over the hyperparameters of a gaussian process.




```
import argparse
import os
import time

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous 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 = init_to_value(
```

(continues on next page)

(continued from previous page)

```
            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):
    # 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)
    K_xx_inv = jnp.linalg.inv(k_XX)
    K = k_pp - jnp.matmul(k_pX, jnp.matmul(K_xx_inv, jnp.transpose(k_pX)))
    sigma_noise = jnp.sqrt(jnp.clip(jnp.diag(K), a_min=0.0)) * jax.random.normal(
        rng_key, X_test.shape[:1]
    )
    mean = jnp.matmul(k_pX, jnp.matmul(K_xx_inv, Y))
    # we return both the mean function and a sample from the posterior predictive for the
    # given set of hyperparameters
    return mean, mean + sigma_noise


# 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)
```

(continues on next page)

(continued from previous page)

```
    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
        )
    )(*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("0.10.1")
    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)
```

(continues on next page)

(continued from previous page)

```
    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"],
    )
    args = parser.parse_args()

    numpyro.set_platform(args.device)
    numpyro.set_host_device_count(args.num_chains)

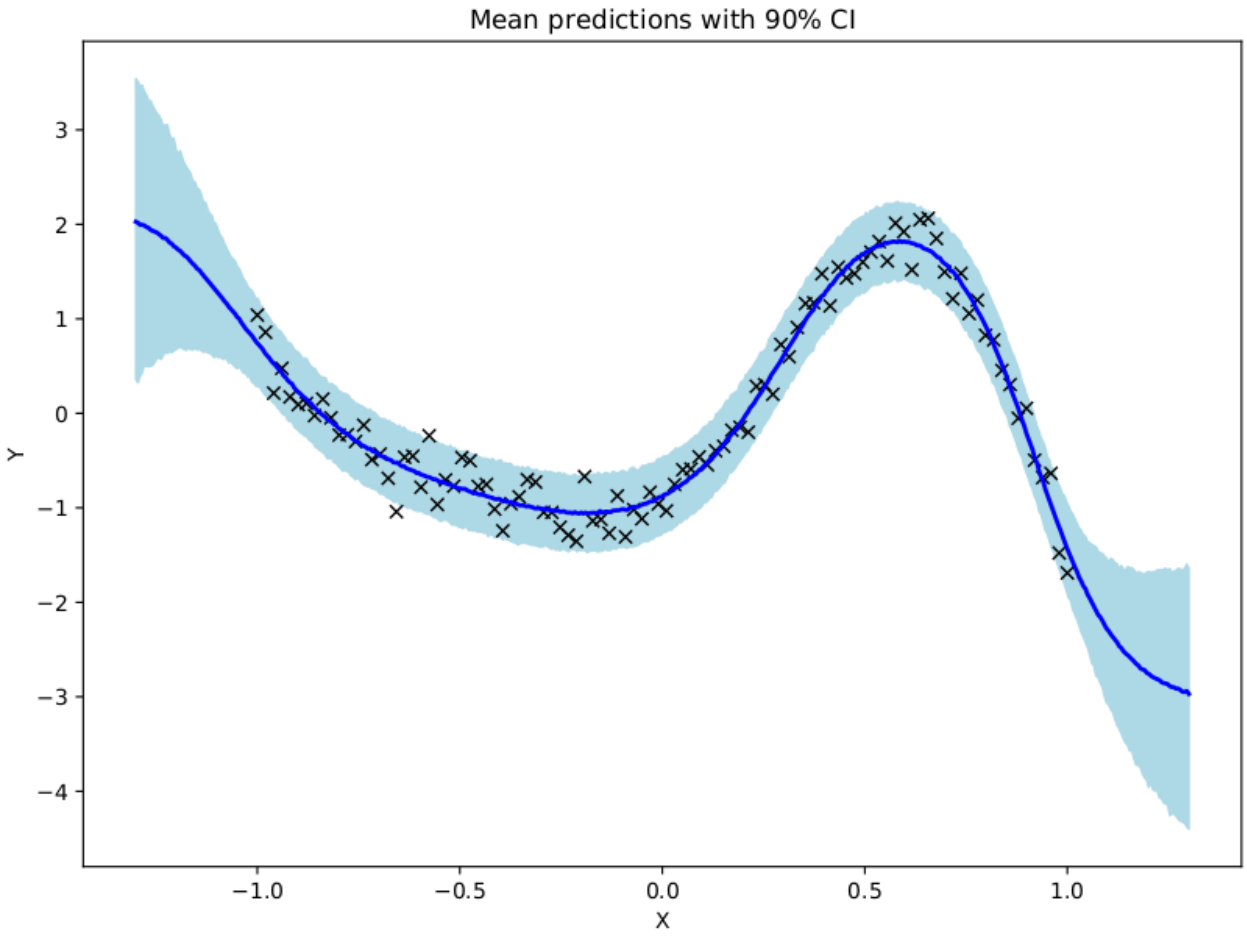
    main(args)
```

CHAPTER

TWENTYSIX

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)

(continued from previous 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))))
    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))))
    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))))
    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))
    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)
```

(continues on next page)

(continued from previous page)

```
# helper function for HMC inference
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("\nMCMC elapsed time:", time.time() - start)
    return mcmc.get_samples()


# helper function for prediction
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
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 = Y[:, np.newaxis]
    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


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)
```

(continues on next page)

(continued from previous page)

```
    # 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("0.10.1")
    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)
```

CHAPTER

TWENTYSEVEN

EXAMPLE: AUTODAIS

AutoDAIS constructs a guide that combines elements of Hamiltonian Monte Carlo, Annealed Importance Sampling, and Variational Inference.

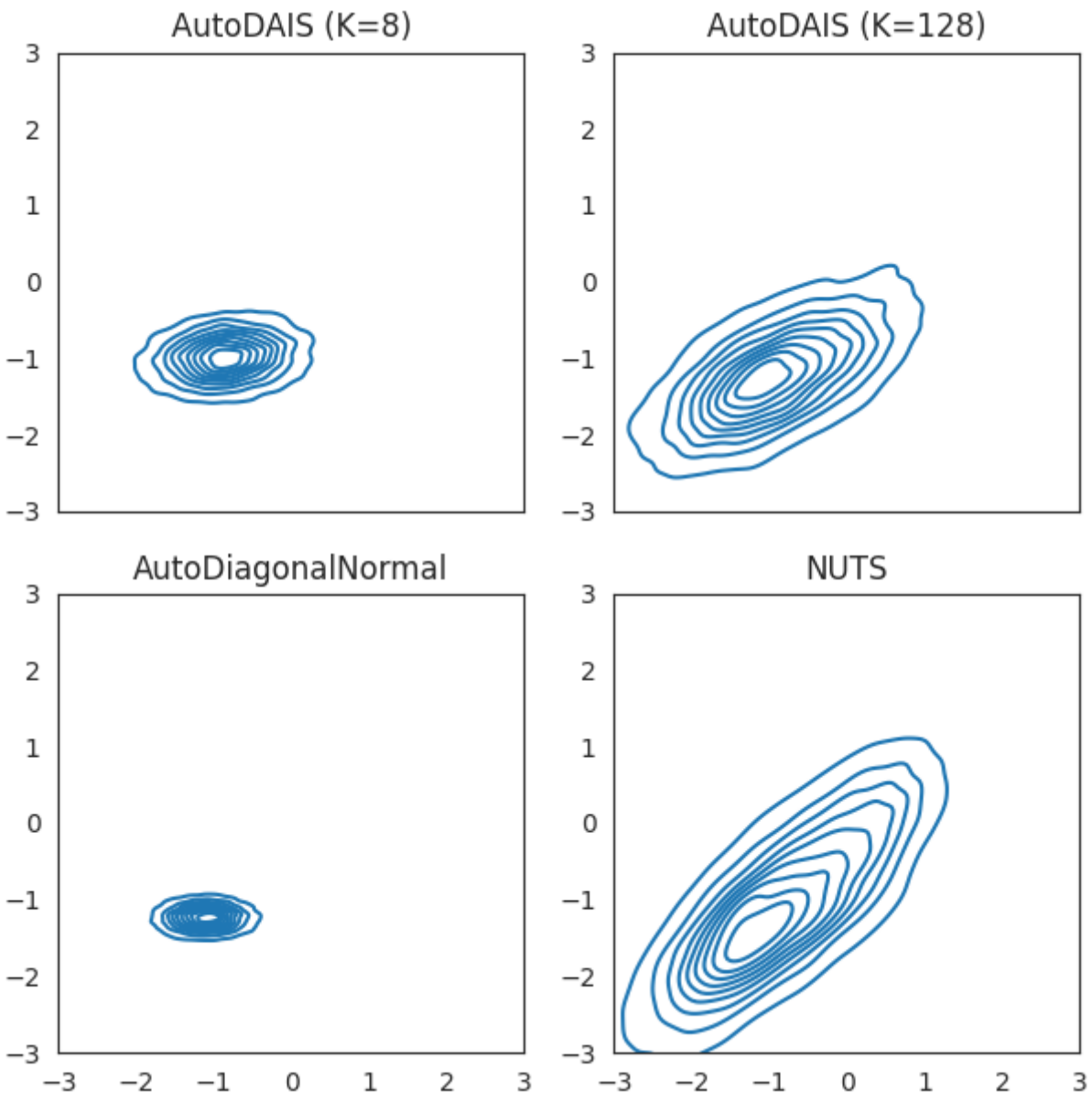
In this demo script we construct a somewhat artificial example involving a gaussian process binary classifier. We aim to demonstrate that:

- 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.

[2] "Differentiable Annealed Importance Sampling and the Perils of Gradient Noise," 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
```

(continues on next page)

(continued from previous 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())
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
        ylim = (-3, 3)

        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")


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)
```

CHAPTER

TWENTYEIGHT

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:

1. Raj Agrawal, Jonathan H. Huggins, Brian Trippe, Tamara Broderick (2019), "The Kernel Interaction Trick: Fast Bayesian Discovery of Pairwise Interactions in High Dimensions", (https://arxiv.org/abs/1905.06501)


```
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):
    eta1sq = 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 = (eta1sq - eta2sq) * dot(X, Z)
    k4 = jnp.square(c) - 0.5 * eta2sq
    if X.shape == Z.shape:
```

(continues on next page)

(continued from previous 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)
    eta1 = numpyro.sample("eta1", 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(eta1) * jnp.sqrt(xisq) / msq

    lam = numpyro.sample("lambda", dist.HalfCauchy(jnp.ones(P)))
    kappa = jnp.sqrt(msq) * lam / jnp.sqrt(msq + jnp.square(eta1 * lam))

    # compute kernel
    kX = kappa * X
    k = kernel(kX, kX, eta1, 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 (eta1, xisq, ...).
# Compare to theorem 5.1 in reference [1].
def compute_singleton_mean_variance(X, Y, dimension, msq, lam, eta1, 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(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)
```

(continues on next page)

(continued from previous page)

```
    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)
    # the total number of coefficients we return
    num_coefficients = P + M * (M - 1) // 2
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
# 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("\nMCMC 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[:, 0: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)

(continued from previous 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)

(continued from previous 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 [mean - 3 * std, mean + 3 *␣
↪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 +- %.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 != 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
```

(continues on next page)

(continued from previous page)

```
            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("0.10.1")
    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=500, 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-dimensions", nargs="?", default=20, type=int)
    parser.add_argument("--active-dimensions", nargs="?", default=3, 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)
```

CHAPTER

TWENTYNINE

EXAMPLE: HORSESHOE REGRESSION

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.

```
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
    mean_function = jnp.dot(X, scaled_betas)
```

(continues on next page)

(continued from previous page)

```
    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


# create artificial regression dataset with 3 non-zero regression coefficients
```

(continues on next page)

(continued from previous 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)

    # 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])

    print("[Experiment with binary-valued responses]")
    # next generate and analyze data 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
```

(continues on next page)

(continued from previous page)

```
    # 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("0.10.1")
    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)
```

CHAPTER

THIRTY

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.

```
import argparse
import os
from typing import Tuple

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])
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
    print(
        f"There is a {interval_size * 100}% probability that calling customers "
        "increases the chance they'll make a purchase by "
        f"{(100 * impact_on_probability[0]):.2} to {(100 * impact_on_probability[1]):.2}␣
↪percentage points."
    )

    print(
        f"There is a {interval_size * 100}% probability the effect of gender on the log␣
↪odds of conversion "
        f"lies in the interval ({effect_of_gender[0]:.2}, {effect_of_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,
```

(continues on next page)

(continued from previous page)

```
        rng_key,
        args.num_warmup,
        args.num_samples,
        args.num_chains,
        args.interval_size,
    )


if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    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)
```

CHAPTER

THIRTYONE

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

dept	male	applications	admit
0	1	825	512
0	0	108	89
1	1	560	353
1	0	25	17
2	1	325	120
2	0	593	202
3	1	417	138
3	0	375	131
4	1	191	53
4	0	393	94
5	1	373	22
5	0	341	24

This example replicates the multilevel model *m_glmm5* 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

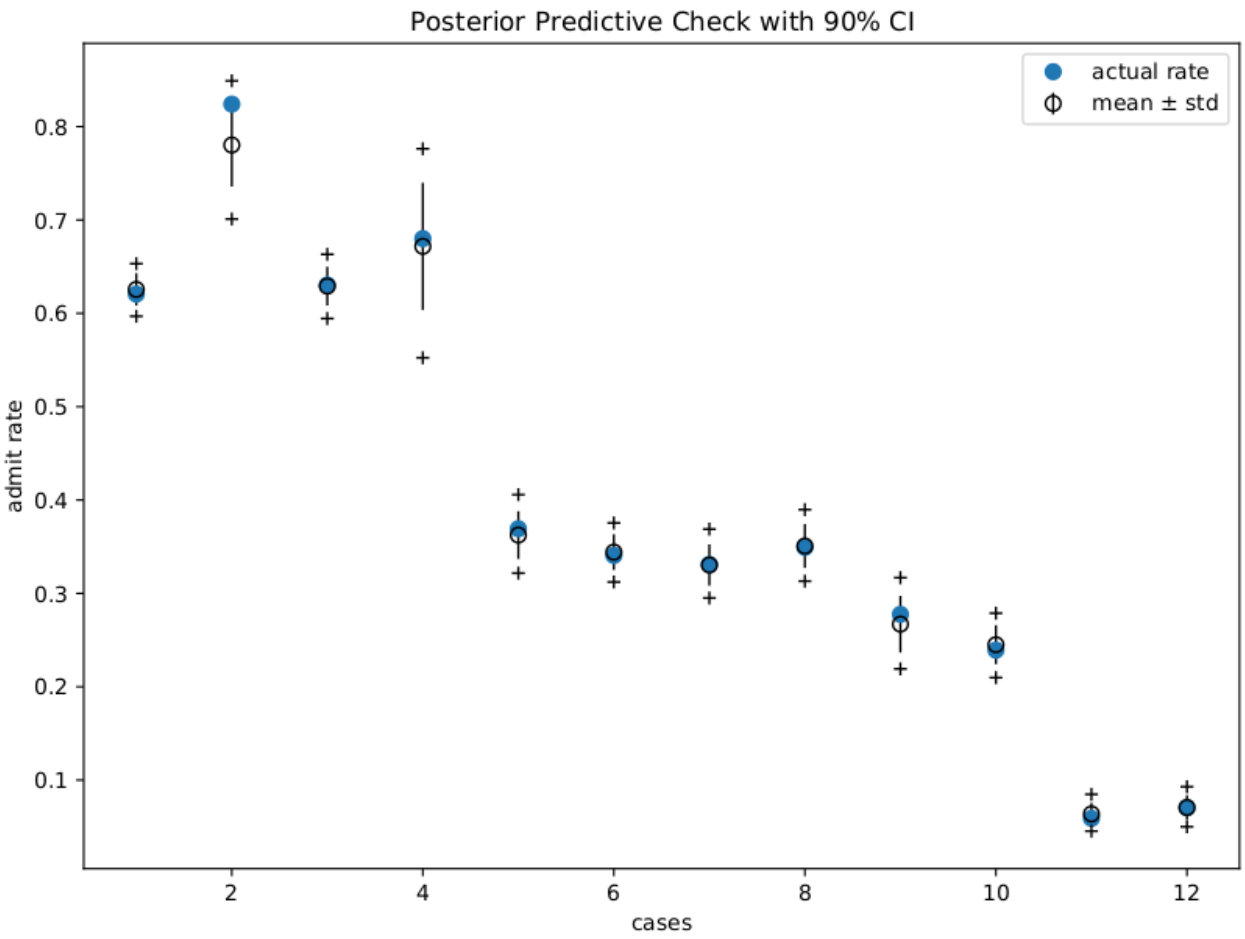
- varying intercepts across departments,
- varying slopes (or the effects of being male) across departments,
- correlation between intercepts and slopes,

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:

1. Bickel, P. J., Hammel, E. A., and O'Connell, J. W. (1975), "Sex Bias in Graduate Admissions: Data from Berkeley", Science, 187(4175), 398-404.
2. McElreath, R. (2018), "Statistical Rethinking: A Bayesian Course with Examples in R and Stan", Chapman and Hall/CRC.
3. https://github.com/rmcelreath/rethinking/tree/Experimental#multilevel-model-formulas




```
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)

(continued from previous 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).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)

(continued from previous 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("0.10.1")
    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)
```

CHAPTER

THIRTYTWO

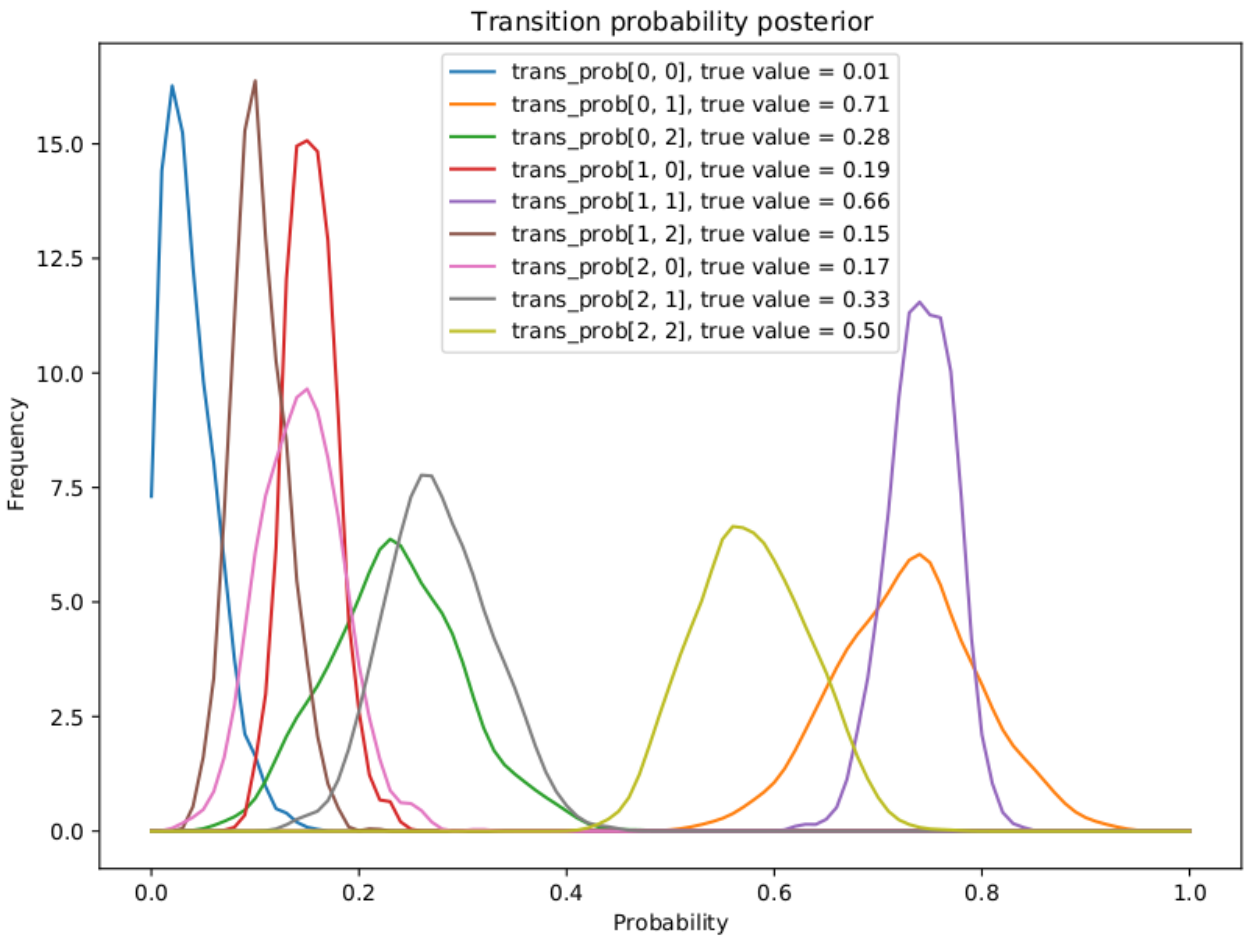
EXAMPLE: HIDDEN MARKOV MODEL

In this example, we will follow [1] to construct a semi-supervised Hidden Markov Model for a generative model with observations are words and latent variables are categories. Instead of automatically marginalizing all discrete latent variables (as in [2]), we will use the "forward algorithm" (which exploits the conditional independent 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 works 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:

1. https://mc-stan.org/docs/2_19/stan-users-guide/hmms-section.html
2. http://pyro.ai/examples/hmm.html
3. https://en.wikipedia.org/wiki/Forward_algorithm
4. https://discourse.pymc.io/t/how-to-marginalized-markov-chain-with-categorical/2230




```
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)
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
    # >>> for word in words:
    # ...     log_prob = forward_one_step(log_prob, word, transition_log_prob, emission_
↪log_prob)
    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)

(continued from previous 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,
```

(continues on next page)

(continued from previous page)

```
        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("\nMCMC 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()
```

(continues on next page)

(continued from previous page)

```
    plt.savefig("hmm_plot.pdf")


if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    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)
```

CHAPTER

THIRTYTHREE

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).

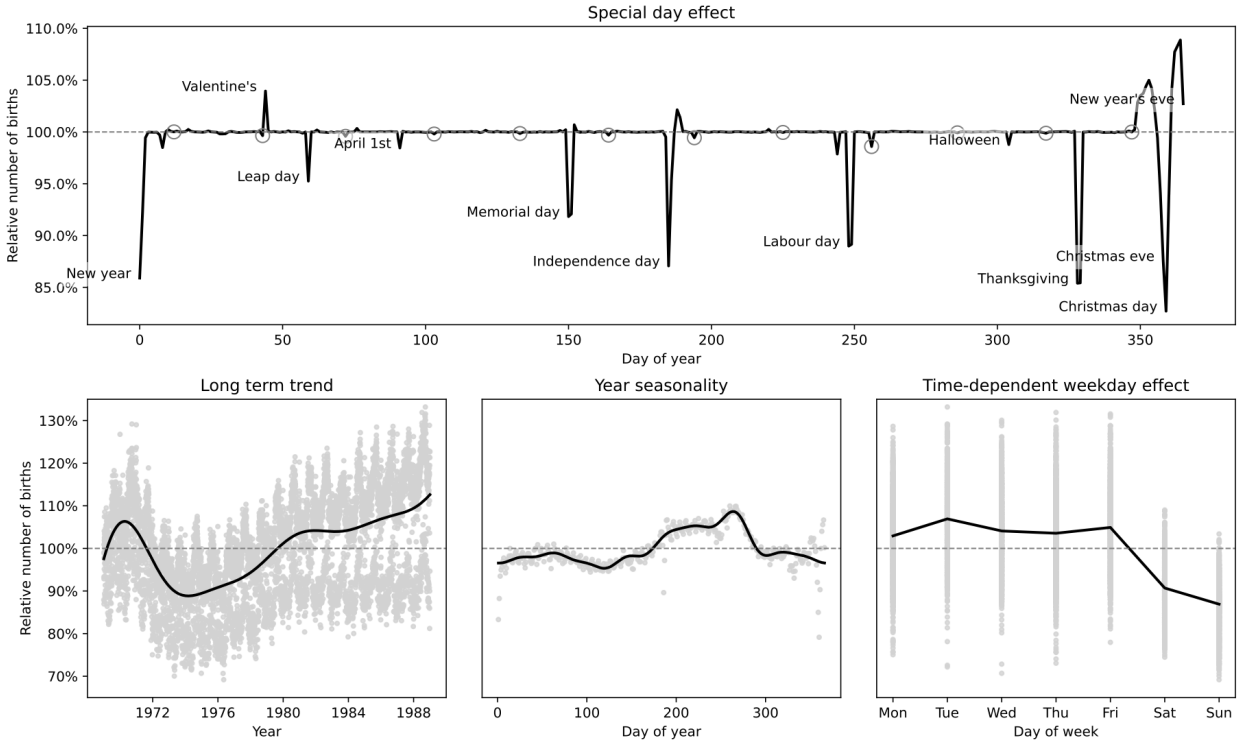
The different components of the model are isolated into separate functions so that they can easily be reused in different contexts. To combine the multiple components into a single birthdays model, here we make use of Numpyro's *scope* handler which modifies the site names of the components by adding 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.

The periodic kernel approximation requires tensorflow-probability on a jax backend. See <https://www.tensorflow.org/probability/examples/TensorFlow_Probability_on_JAX> for installation instructions.

References:

1. Gelman, Vehtari, Simpson, et al (2020), *"Bayesian workflow book - Birthdays" <https://avehtari.github.io/casestudies/Birthdays/birthdays.html>*.
2. Riutort-Mayol G, Bürkner PC, Andersen MR, et al (2020), "Practical hilbert space approximate bayesian gaussian processes for probabilistic programming".




```
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
import 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)
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
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))
```

(continues on next page)

(continued from previous page)

```
    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):
    """
```

(continues on next page)

(continued from previous page)

```
    Low rank approximation for the periodic squared
    exponential kernel in the non-centered parametrisation.
    """
    q2 = diag_spectral_density_periodic(alpha, length, M)
    cosines, sines = eigenfunctions_periodic(x, w0, M)

    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 --- #
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


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


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]


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)
```

(continues on next page)

(continued from previous page)

```
    )  # Original uses half-t with 100df
    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
    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]


def special_effect(indicator):
    beta = sample("beta", dist.Normal(0, 1))
    return beta * indicator


# --- Model --- #
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)

(continued from previous 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)

(continued from previous 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()]
```

(continues on next page)

(continued from previous page)

```
    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 = (
```

(continues on next page)

(continued from previous page)

```
        plt.subplot(grid[0, :]),
        plt.subplot(grid[1, 0]),
        plt.subplot(grid[1, 1]),
        plt.subplot(grid[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)

(continued from previous page)

```
    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)
```

CHAPTER

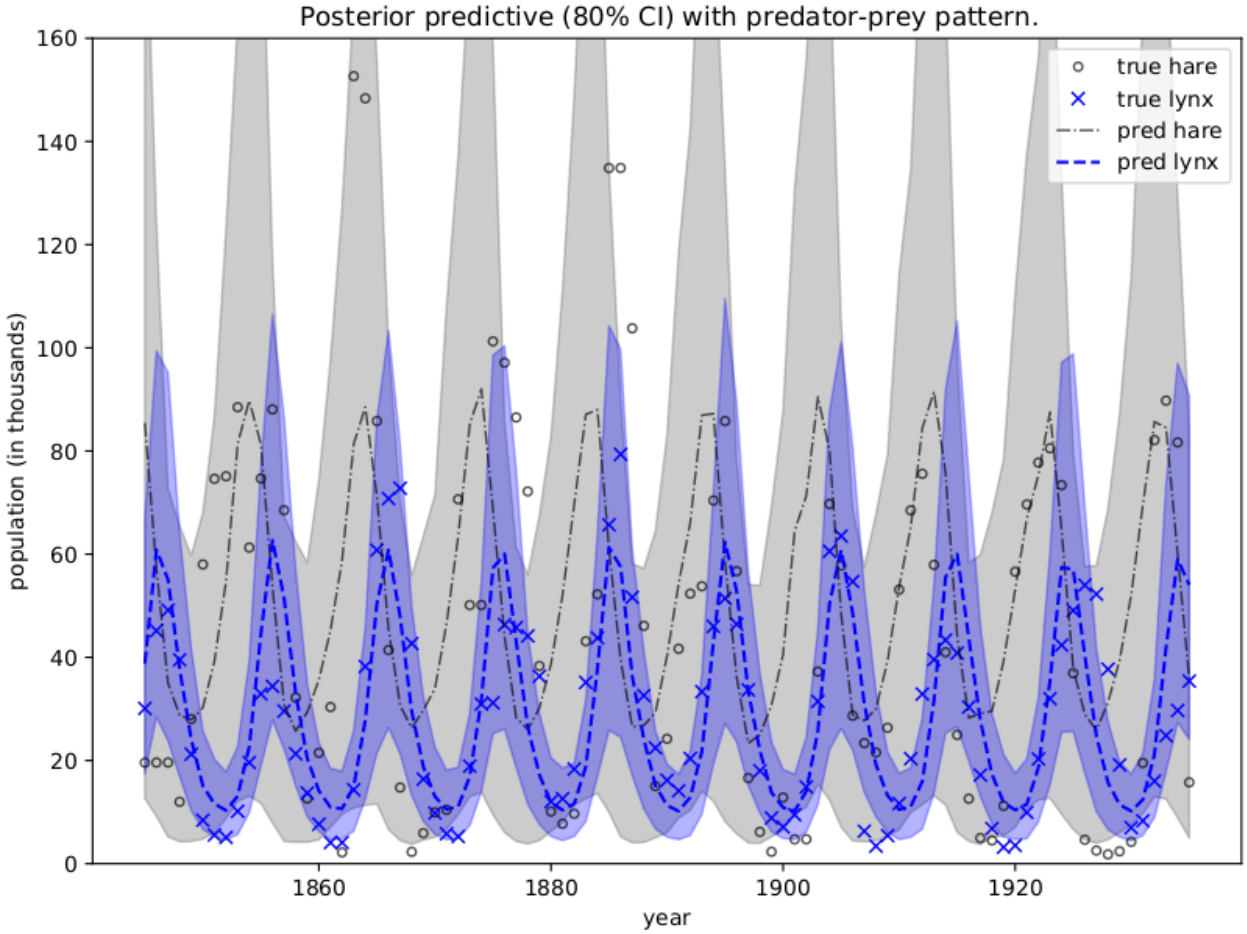
THIRTYFOUR

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:

1. Bob Carpenter (2018), "Predator-Prey Population Dynamics: the Lotka-Volterra model in Stan".
2. https://en.wikipedia.org/wiki/Lotka-Volterra_equations
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",
        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)

(continued from previous page)

```
    )
    # integrate dz/dt, the result will have shape N x 2
    z = odeint(dz_dt, z_init, ts, theta, rtol=1e-6, atol=1e-5, mxstep=1000)
    # measurement errors
    sigma = numpyro.sample("sigma", dist.LogNormal(-1, 1).expand([2]))
    # measured populations
    numpyro.sample("y", dist.LogNormal(jnp.log(z), sigma), 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("0.10.1")
    parser = argparse.ArgumentParser(description="Predator-Prey Model")
    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)
```

(continues on next page)

(continued from previous page)

```
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```

CHAPTER

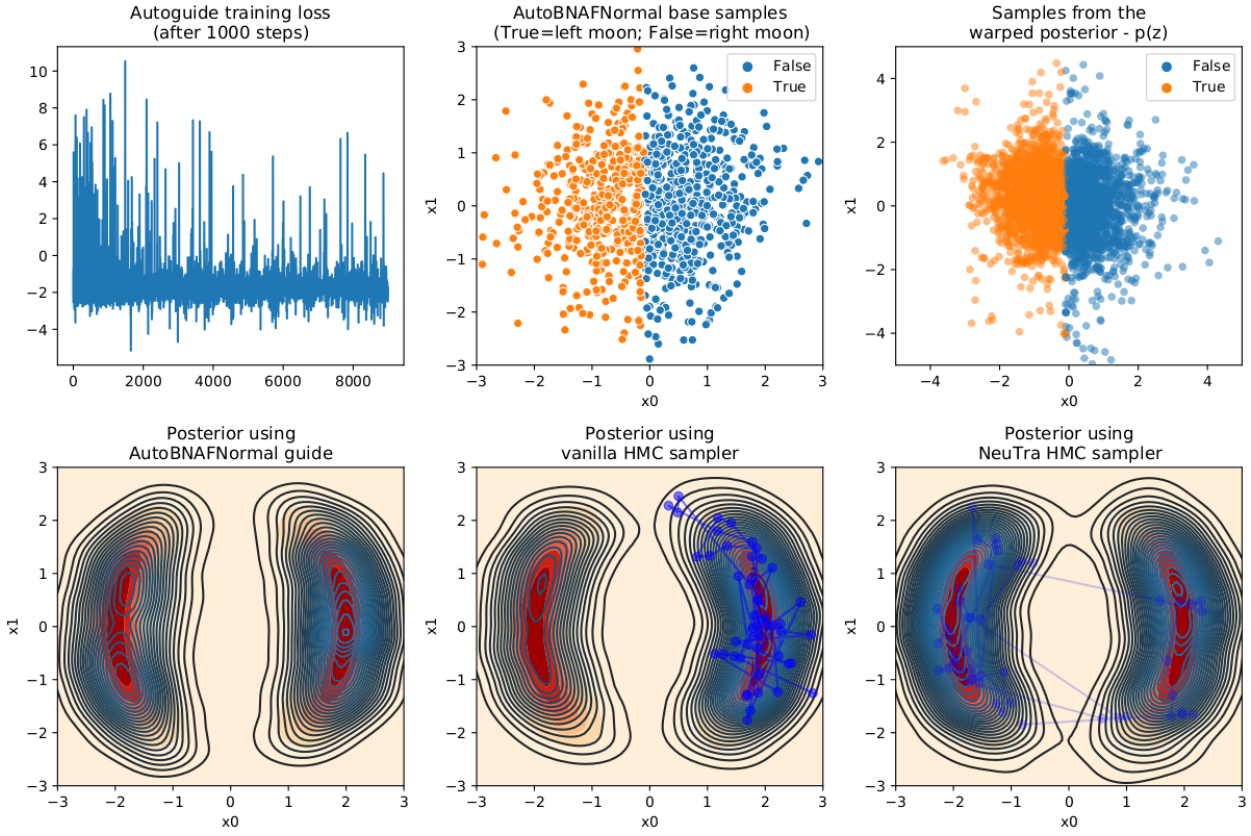
THIRTYFIVE

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:

1. Hoffman, M. et al. (2019), "NeuTra-lizing Bad Geometry in Hamiltonian Monte Carlo Using Neural Transport", (https://arxiv.org/abs/1903.03704)




```
import argparse
import os

from matplotlib.gridspec import GridSpec
import matplotlib.pyplot as plt
import seaborn as sns
```

(continues on next page)

(continued from previous 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)

(continued from previous 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...")
    neutra = NeuTraReparam(guide, svi_result.params)
    neutra_model = neutra.reparam(dual_moon_model)
    nuts_kernel = NUTS(neutra_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)

(continued from previous page)

```
    ax1.set_title("Autoguide training loss\n(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\nAutoBNAFNormal 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\n(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)
    ax4.plot(vanilla_samples[-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\nvanilla 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\nwarped posterior - p(z)",
```

(continues on next page)

(continued from previous page)

```
    )

    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\nNeuTra HMC sampler",
    )

    plt.savefig("neutra.pdf")


if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    parser = argparse.ArgumentParser(description="NeuTra HMC")
    parser.add_argument("-n", "--num-samples", nargs="?", default=4000, 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("--hidden-factor", nargs="?", default=8, type=int)
    parser.add_argument("--num-iters", nargs="?", 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)

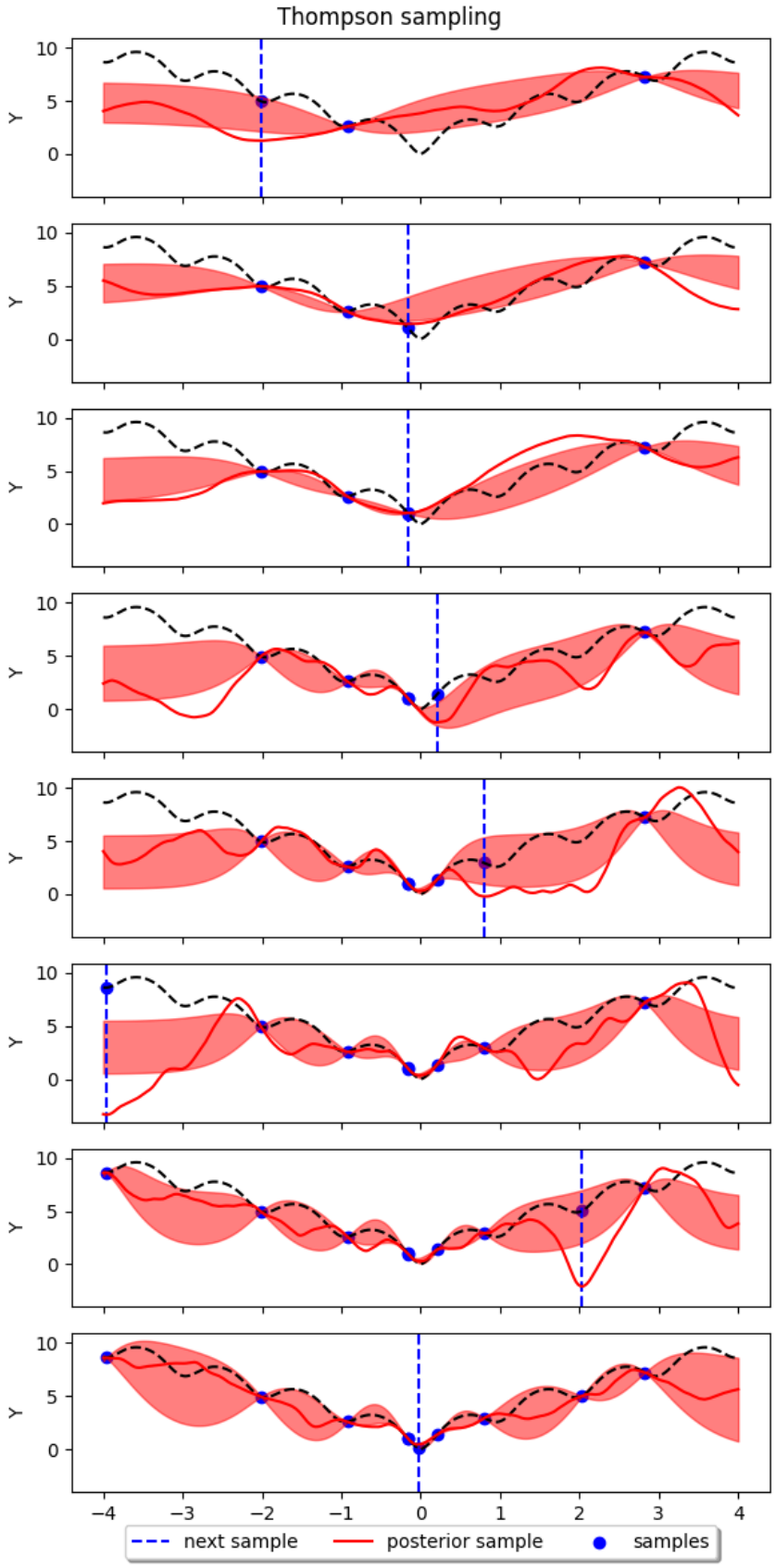
    main(args)
```

CHAPTER

THIRTYSIX

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/


Thompson sampling
Y
10
5
0
−4
−3
−2
−1
0
1
2
3
4
next sample
posterior sample
samples

```
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)

(continued from previous 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
```

(continues on next page)

(continued from previous page)

```
        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)


# our TS-GP optimizer
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)
```

(continues on next page)

(continued from previous page)

```
    # 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␣
↪minimum
        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
```

(continues on next page)

(continued from previous page)

```
        return (
            self.X,
            self.y,
            self.X_grid,
            posterior_sample,
            posterior_mean,
            posterior_std,
        )


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(
```

(continues on next page)

(continued from previous page)

```
                X_grid,
                posterior_mean - posterior_std,
                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("0.10.1")
    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

THIRTYSEVEN

BAYESIAN HIERARCHICAL STACKING: WELL SWITCHING CASE STUDY

Photo by Belinda Fewings, https://unsplash.com/photos/6p-KtXCBGNw.

37.1 Table of Contents

37.2 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]:
```
import os

from IPython.display import set_matplotlib_formats
import arviz as az
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

import numpyro
import numpyro.distributions as dist
```

(continues on next page)

(continued from previous page)

```
plt.style.use("seaborn")
if "NUMPYRO_SPHINXBUILD" in os.environ:
    set_matplotlib_formats("svg")

numpyro.set_host_device_count(4)
assert numpyro.__version__.startswith("0.10.1")
```

[3]:
```
%matplotlib inline
```

37.3 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.

[4]:
```
wells = pd.read_csv(
    "http://stat.columbia.edu/~gelman/arm/examples/arsenic/wells.dat", sep=" "
)
```

[5]:
```
wells.head()
```

[5]:

	switch	arsenic	dist	assoc	educ
1	1	2.36	16.826000	0	0
2	1	0.71	47.321999	0	0
3	0	2.07	20.966999	0	10
4	1	1.15	21.486000	0	12
5	1	1.10	40.874001	1	14

[6]:
```
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",
    ax=ax[1][0],
)
```

(continues on next page)

(continued from previous page)

```
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.

[7]:
```
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()
```

37.4 2. Prepare 6 different candidate models

37.4.1 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.

[8]:
```
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
```

[9]:
```
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.
    """
    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))
```

(continues on next page)

(continued from previous page)

```
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()
```

[11]:
```
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)
```

37.4.2 2.2 Training

Each model will be trained in the same way - with a Bernoulli likelihood and a logit link function.

[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,
    )
```

[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)
```

37.4.3 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.

[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)
```

37.5 3. Bayesian Hierarchical Stacking

37.5.1 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;

[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)

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()
```

37.5.2 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 $Pred$ such that $Pred_{i,k}$ represents the prediction made for point i by model k. Then the final prediction for point i will then be:

$$\sum_k W_{i,k} Pred_{i,k}$$

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 β is a matrix whose values we seek to determine. For the discrete features, β 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 β, because the weights `W_{i, k}` will have to sum to 1 for each `i`.

[16]:
```
def stacking(
    X,
    d_discrete,
    X_test,
    exp_lpd_point,
    tau_mu,
    tau_sigma,
    *,
    test,
):
    """
    Get weights with which to stack candidate models' predictions.
```

(continues on next page)

(continued from previous page)

```
    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.
    exp_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
            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))])
```

(continues on next page)

(continued from previous page)

```
    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)
        w_test = numpyro.deterministic("w_test", jax.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,
    exp_lpd_point=exp_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.

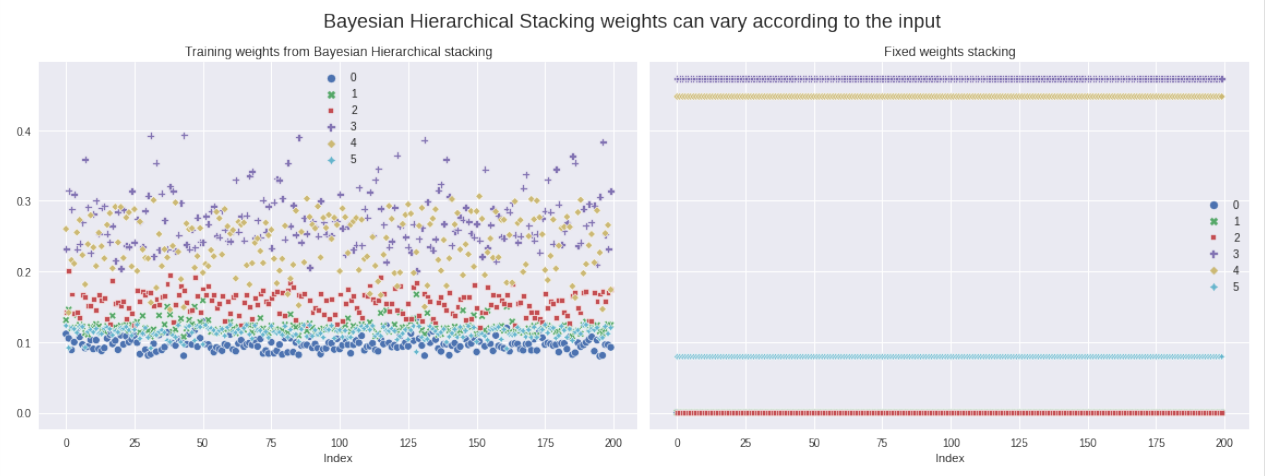
Let's compare them with what the weights would've been if we'd just used fixed stacking weights (computed using ArviZ - see their docs for details).

[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])
fixed_weights = (
    az.compare({idx: fit for idx, fit in enumerate(fit_list)}, method="stacking")
    .sort_index()["weight"]
    .to_numpy()
)
fixed_weights_df = pd.DataFrame(
    np.repeat(
        fixed_weights[jnp.newaxis, :],
        len(X_stacking_train),
        axis=0,
    )
)
```

(continues on next page)

(continued from previous page)

```
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")
fig.suptitle(
    "Bayesian Hierarchical Stacking weights can vary according to the input",
    fontsize=18,
)
fig.tight_layout();
```



37.6 4. Evaluate on test set

37.6.1 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`).

[19]:
```
# 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
```

[20]:
```
# 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())
```

(continues on next page)

(continued from previous page)

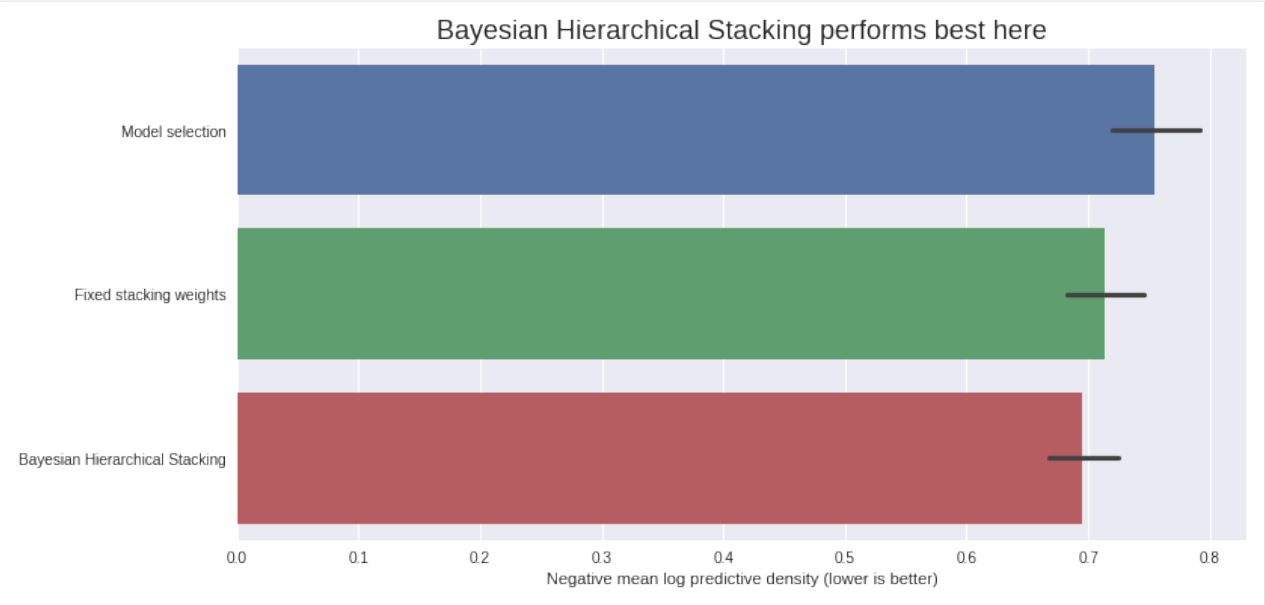
```
    rng_key = jax.random.fold_in(jax.random.PRNGKey(20), k)
    test_pred = predictive(rng_key, x=test_x_list[k])["logits"]
    test_preds.append(test_pred.mean(axis=0))
test_preds = np.vstack(test_preds).T
```

[21]:
```
# 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)
```

37.6.2 4.2 Compare methods

Let's compare the negative log predictive density scores on the test set (note - lower is better):

[22]:
```
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
    ),
    orient="h",
    ax=ax,
)
ax.set_title(
    "Bayesian Hierarchical Stacking performs best here", fontdict={"fontsize": 18}
)
ax.set_xlabel("Negative mean log predictive density (lower is better)");
```



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.

37.6.3 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.

37.7 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.

37.8 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
3. https://github.com/yao-yl/hierarchical-stacking-code.
4. Thomas Wiecki (2017). Why hierarchical models are awesome, tricky, and Bayesian

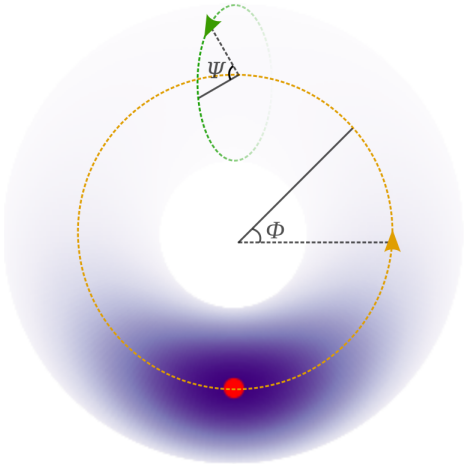
[ ]:

CHAPTER

THIRTYEIGHT

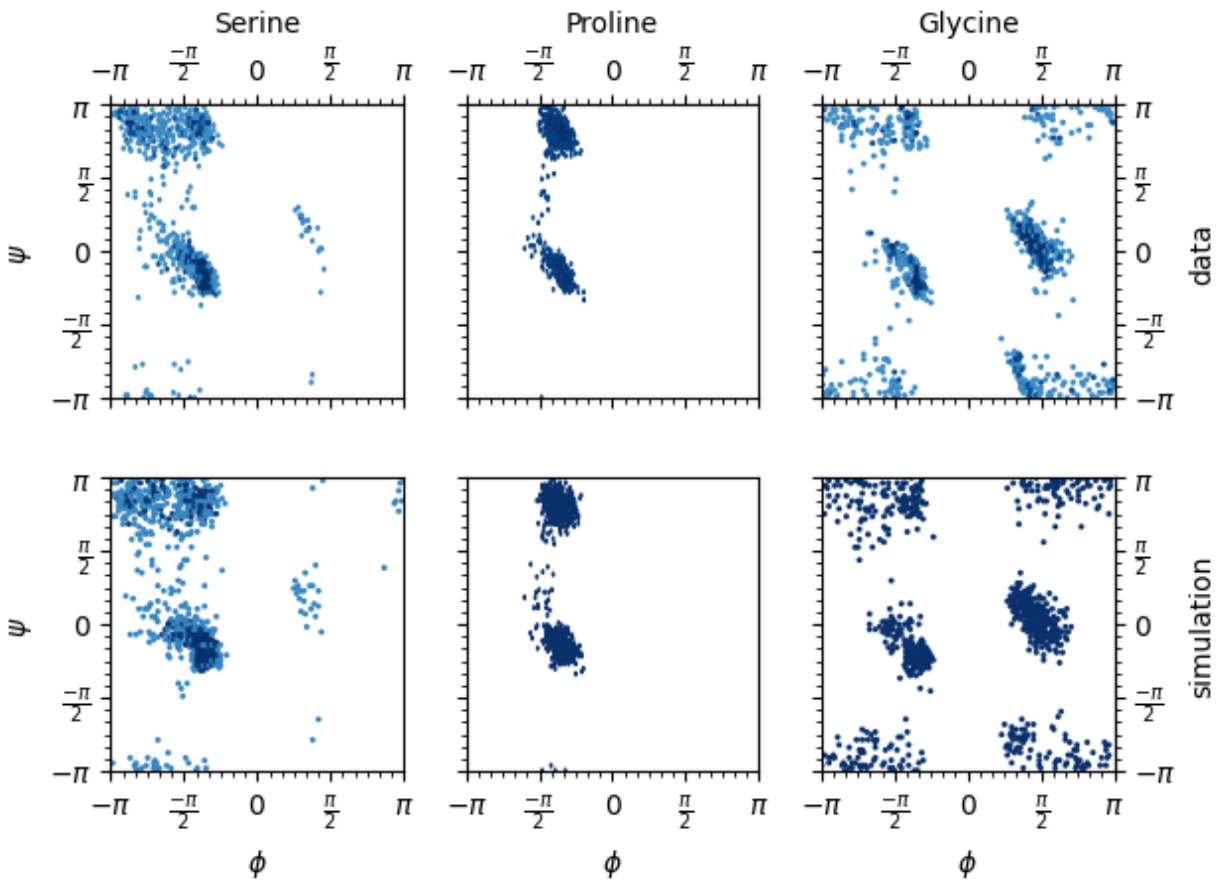
EXAMPLE: SINE-SKEWED SINE (BIVARIATE VON MISES) MIXTURE

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 ϕ and ψ, 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:

1. Singh et al. (2002). Probabilistic model for two dependent circular variables. Biometrika.
2. Jose Ameijeiras-Alonso and Christophe Ley (2021). Sine-skewed toroidal distributions and their application in protein bioinformatics. Biostatistics.




```
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)

(continued from previous 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
# 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
```

(continues on next page)

(continued from previous page)

```
    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
    )
```

(continues on next page)

(continued from previous page)

```
    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)

(continued from previous 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(labelleft=False)

    for ax in axss[0, :].reshape(-1):
        ax.tick_params(labelbottom=False)
        ax.tick_params(labelbottom=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"$%s%s$" % (num, latex)
        else:
            if num == 1:
                return r"$\frac{%s}{%s}$" % (latex, den)
            elif num == -1:
```

(continues on next page)

(continued from previous page)

```
                return r"$\frac{-%s}{%s}$" % (latex, den)
            else:
                return r"$\frac{%s%s}{%s}$" % (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"{list(filter(lambda aa: aa not in AMINO_ACIDS, args.amino_acids))} are not␣
↪amino acids."
    main(args)
```

CHAPTER

THIRTYNINE

EXAMPLE: AR2 PROCESS

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.

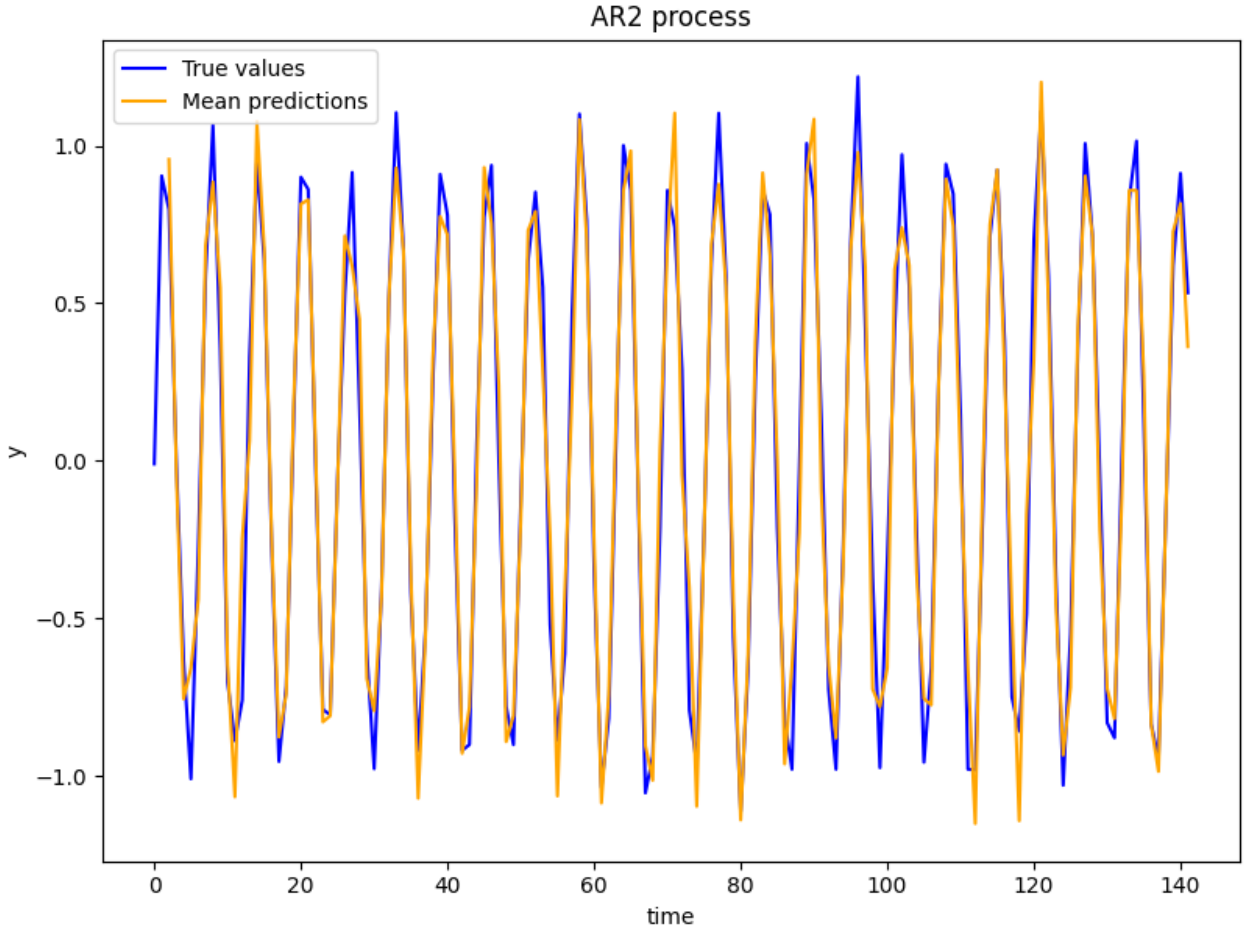
To demonstrate, we will be implementing an AR2 process. The idea is that we have some times series

$$y_0, y_1, ..., y_T$$

and we seek parameters c, α_1, and α_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 ϵ_t is an error term.



```
import argparse
import os
import time

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, None

    timesteps = jnp.arange(y.shape[0] - 2)
    init = (y[1], y[0])

    with numpyro.handlers.condition(data={"y": y[2:]}):
        scan(transition, init, timesteps)


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]

    for i in range(2, len(y)):
        m_t = const + alpha_1 * y_prev + alpha_2 * y_prev_prev
        y_t = numpyro.sample("y_{}".format(i), dist.Normal(m_t, sigma), obs=y[i])
        y_prev_prev = y_prev
        y_prev = y_t


def run_inference(model, args, rng_key, y):
    start = time.time()
    sampler = numpyro.infer.NUTS(model)
    mcmc = numpyro.infer.MCMC(
```

(continues on next page)

(continued from previous page)

```
        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)
    mcmc.print_summary()
    print("\nMCMC elapsed time:", time.time() - start)
    return mcmc.get_samples()


def main(args):
    # generate artifical 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

    run_inference(model, args, rng_key, y)


if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    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", type=str, 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)
```

CHAPTER

FORTY

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

$$y_1, ..., y_T, y_{T+1}, ..., y_{T+H}$$

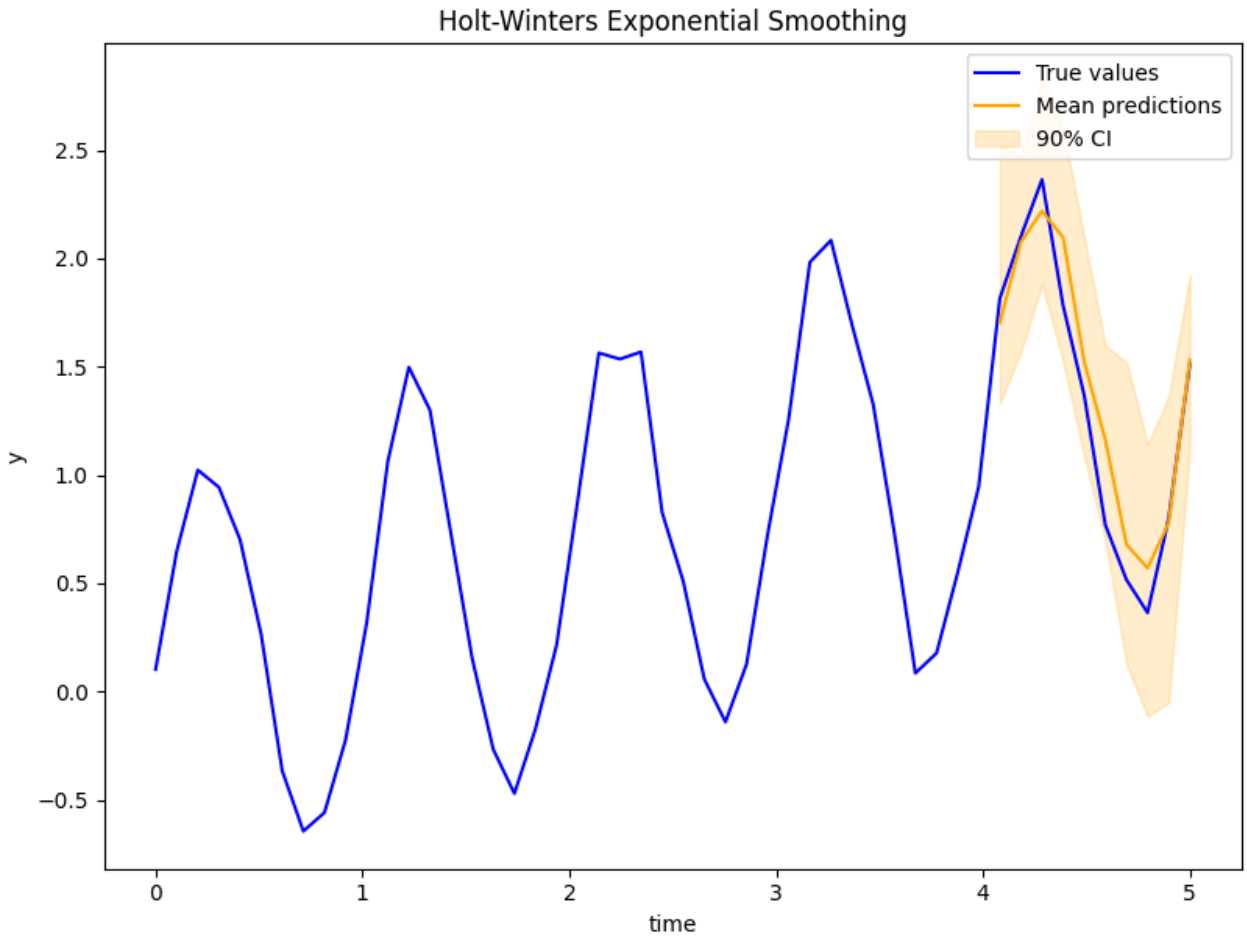
where we train on $y_1, ..., y_T$ and predict for $y_{T+1}, ..., y_{T+H}$, where T is the maximum training timestamp and H is the maximum number of future timesteps for which we want to forecast.

We will be using the update equations from the excellent book Forecasting Principles and Practice:

$$\begin{aligned}
\hat{y}_{t+h|t} &= l_t + hb_t + s_{t+h-m(k+1)} \\
l_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1}) \\
b_t &= \beta^*(l_t - l_{t-1}) + (1-\beta^*)b_{t-1} \\
s_t &= \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}
\end{aligned}$$

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,
- α is the level smoothing, β^* is the trend smoothing, and γ 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
```

(continues on next page)

(continued from previous page)

```
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)

(continued from previous 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
    )["y_forecast"]


def main(args):
    # generate artifical 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()
```

(continues on next page)

(continued from previous page)

```
    plt.savefig("holt_winters_plot.pdf")


if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    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)
```

CHAPTER

FORTYONE

EXAMPLE: MODELLING MORTALITY OVER SPACE AND TIME

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 (α_0) and slope (β_0), the model includes the following effects:

- Age (α_{2a}, β_{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 (α_{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., α_{1s_1}) containing that lower level unit.

The model also has a random walk effect over time (π_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

$$\alpha_{1s_1} \sim \mathcal{N}(0, \sigma^2_{\alpha_{s_1}}) \tag{41.1}$$

$$\alpha_{1s} \sim \mathcal{N}(\alpha_{1s_1(s_2)}, \sigma^2_{\alpha_{s_2}}) \tag{41.2}$$

$$\alpha_{2a} \sim \mathcal{N}(\alpha_{2,a-1}, \sigma^2_{\alpha_a}) \quad \alpha_{2,0} = \alpha_0 \tag{41.3}$$

$$\beta_{2a} \sim \mathcal{N}(\beta_{2,a-1}, \sigma^2_{\beta_a}) \quad \beta_{2,0} = \beta_0 \tag{41.4}$$

$$\pi_t \sim \mathcal{N}(\pi_{t-1}, \sigma^2_{\pi}), \quad \pi_0 = 0 \tag{41.5}$$

$$\text{logit}(m_{ast}) = \alpha_{1s} + \alpha_{2a} + \beta_{2a} t + \pi_t \tag{41.6}$$

with the hyperpriors

$$\alpha_0 \sim \mathcal{N}(0, 10), \tag{41.7}$$

$$\beta_0 \sim \mathcal{N}(0, 10), \tag{41.8}$$

$$\sigma_i \sim \mathcal{N}^+(1) \tag{41.9}$$

Further detail about the model terms can be found in [1].

The NumPyro implementation below uses `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 alogrithm benefits from a non-centred reparamatrisation 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 `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

1. Rashid, T., Bennett, J.E. et al. (2021). Life expectancy and risk of death in 6791 communities in England from 2002 to 2019: high-resolution spatiotemporal analysis of civil registration data. The Lancet Public Health, 6, e805 - e816.
2. Stan User's Guide. https://mc-stan.org/docs/2_28/stan-users-guide/reparameterization.html
3. Mortality using Bayesian hierarchical models. https://github.com/theorashid/mortality-statsmodel


```
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)

(continued from previous 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,  # pad so first term is alpha0, prior N(0, 10)
        )[:, 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))
```

(continues on next page)

(continued from previous page)

```
    # 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()

    lookup = 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("0.10.1")

    parser = argparse.ArgumentParser(description="Mortality regression model")
```

(continues on next page)

(continued from previous page)

```
    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

FORTYTWO

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.

```
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 = numpyro.sample("b1", dist.Normal(0.0, 1.0).expand([D_X]).to_event(1))
    b2 = numpyro.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)

(continued from previous 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)
```

(continues on next page)

(continued from previous page)

```
    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")
```

(continues on next page)

(continued from previous page)

```
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)
```

CHAPTER

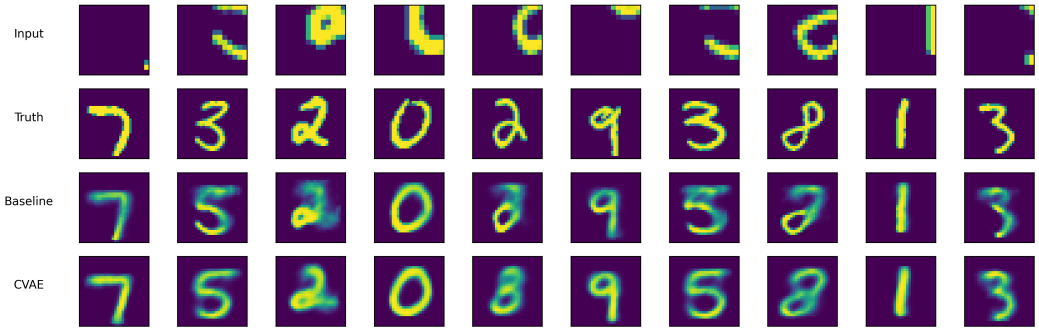
FORTYTHREE

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.



References:

1. Kihyuk Sohn, Xinchen Yan, Honglak Lee (2015), "Learning Structured Output Representation using Deep Conditional Generative Models (https://papers.nips.cc/paper/5775-learning-structured-output-representation-using-deep-conditional-generative-models)

CHAPTER

FORTYFOUR

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:

1. *Sample Adaptive MCMC*, Michael Zhu (2019)
2. *Hamiltonian Monte Carlo with energy conserving subsampling*, Dang, K. D., Quiroz, M., Kohn, R., Minh-Ngoc, T., & Villani, M. (2019)
3. *NeuTra-lizing Bad Geometry in Hamiltonian Monte Carlo Using Neural Transport*, Hoffman, M. et al. (2019)


```
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, Trace_ELBO, 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)

(continued from previous 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,
```

(continues on next page)

(continued from previous page)

```
                -4.34899796e-03,
                +5.90927452e-02,
                +7.23133609e-02,
                +1.38526391e-02,
                -1.24497898e-01,
                -1.50733739e-02,
                -2.68872194e-02,
                -1.80925727e-02,
                +3.47936489e-02,
                +4.03552800e-02,
                -9.98773426e-03,
                +6.20188080e-02,
                +1.15002751e-01,
                +1.32145107e-01,
                +2.69109547e-01,
                +2.45785132e-01,
                +1.19035013e-01,
                -2.59744357e-02,
                +9.94279515e-04,
                +3.39266285e-02,
                -1.44057125e-02,
                -6.95222765e-02,
                -7.52013028e-02,
                +1.21171586e-01,
                +2.29205526e-02,
                +1.47308692e-01,
                -8.34354162e-02,
                -9.34122875e-02,
                -2.97472421e-02,
                -3.03937674e-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)

(continued from previous 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)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.10.1")
    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)
```

CHAPTER

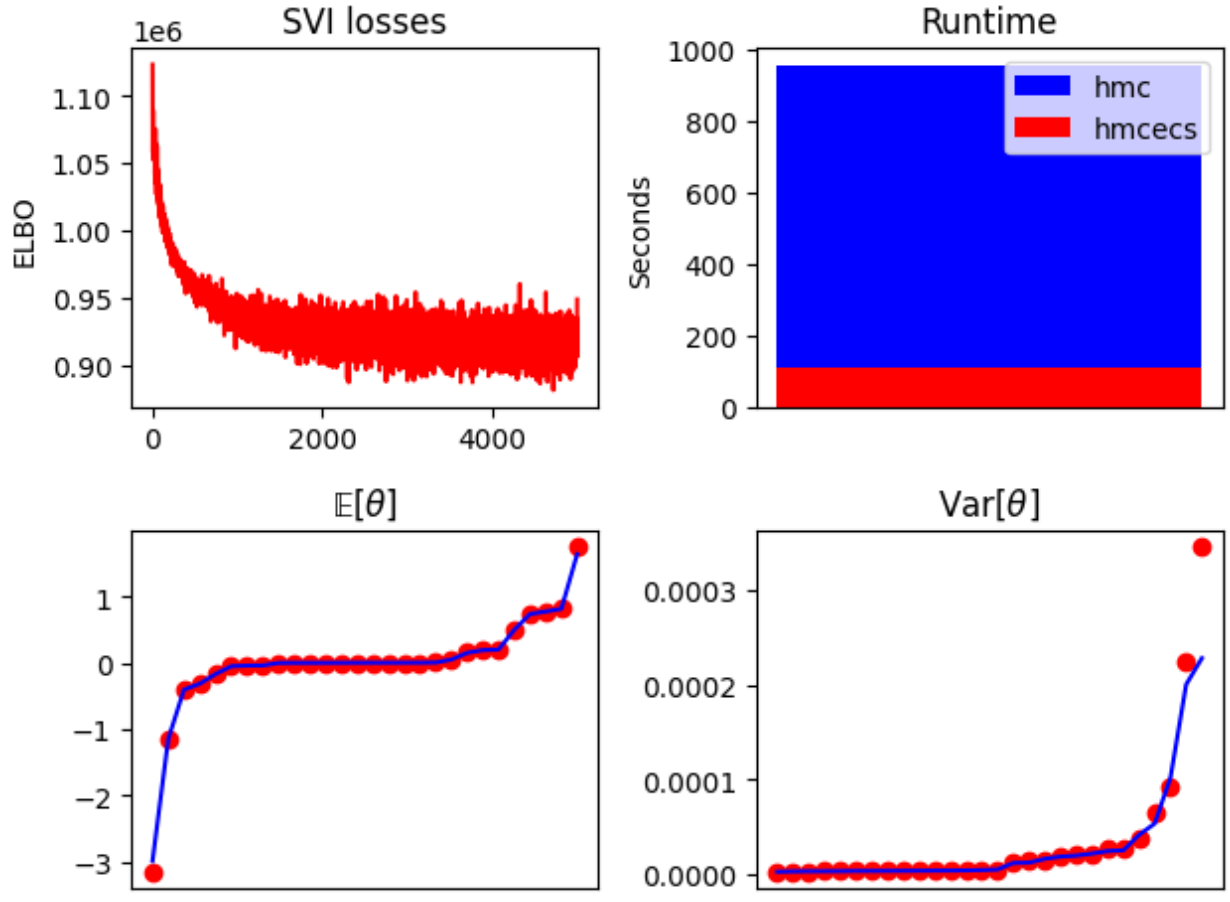
FORTYFIVE

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)

(continued from previous 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=args.num_blocks, 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)
```

(continues on next page)

(continued from previous page)

```
    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()
```

(continues on next page)

(continued from previous page)

```
    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"$\mathrm{\mathbb{E}}[\theta]$")

    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)
```

CHAPTER

FORTYSIX

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.

```
import argparse
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

from jax import random
import jax.numpy as jnp

import numpyro
from numpyro.contrib.einstein import RBFKernel, SteinVI
from numpyro.distributions import Gamma, Normal
from numpyro.examples.datasets import BOSTON_HOUSING, load_dataset
from numpyro.infer import Predictive, Trace_ELBO, init_to_uniform
from numpyro.infer.autoguide import AutoDelta
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)

    return DataState(*map(partial(jnp.array, dtype=float), (xtr, xte, ytr, yte)))


def normalize(val, mean=None, std=None):
    """Normalize data to zero mean, unit variance"""
    if mean is None and std is None:
        # Only use training data to estimate mean and std.
```

(continues on next page)

(continued from previous page)

```
        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))
    with numpyro.plate(
        "data",
        x.shape[0],
```

(continues on next page)

(continued from previous page)

```
        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

        numpyro.sample(
            "y",
            Normal(
                jnp.maximum(batch_x @ w1 + b1, 0) @ w2 + b2, 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)

    stein = SteinVI(
        model,
        AutoDelta(model, init_loc_fn=partial(init_to_uniform, radius=0.1)),
        Adagrad(0.05),
        Trace_ELBO(20),  # estimate elbo with 20 particles (not stein particles!)
        RBFKernel(),
        repulsion_temperature=args.repulsion,
        num_particles=args.num_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 = Predictive(
```

(continues on next page)

(continued from previous page)

```
        model,
        guide=stein.guide,
        params=stein.get_params(result.state),
        num_samples=200,
        batch_ndims=1,  # stein particle dimension
    )
    xte, _, _ = normalize(
        data.xte, xtr_mean, xtr_std
    )  # use train data statistics when accessing generalization
    preds = pred(
        pred_key, xte, subsample_size=xte.shape[0], hidden_dim=args.hidden_dim
    )["y"]

    y_pred = jnp.mean(preds, 1) * 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 = jnp.mean(y_pred, 0)

    ran = np.arange(mean_prediction.shape[0])
    percentiles = np.percentile(
        preds.reshape(-1, xte.shape[0]) * 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", ylabel="y", title="Mean predictions with 90% CI")
    ax.legend()
    fig.savefig("stein_bnn.pdf")


if __name__ == "__main__":
    from jax.config import config

    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-particles", type=int, default=100)
```

(continues on next page)

(continued from previous page)

```
    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)
```

CHAPTER

FORTYSEVEN

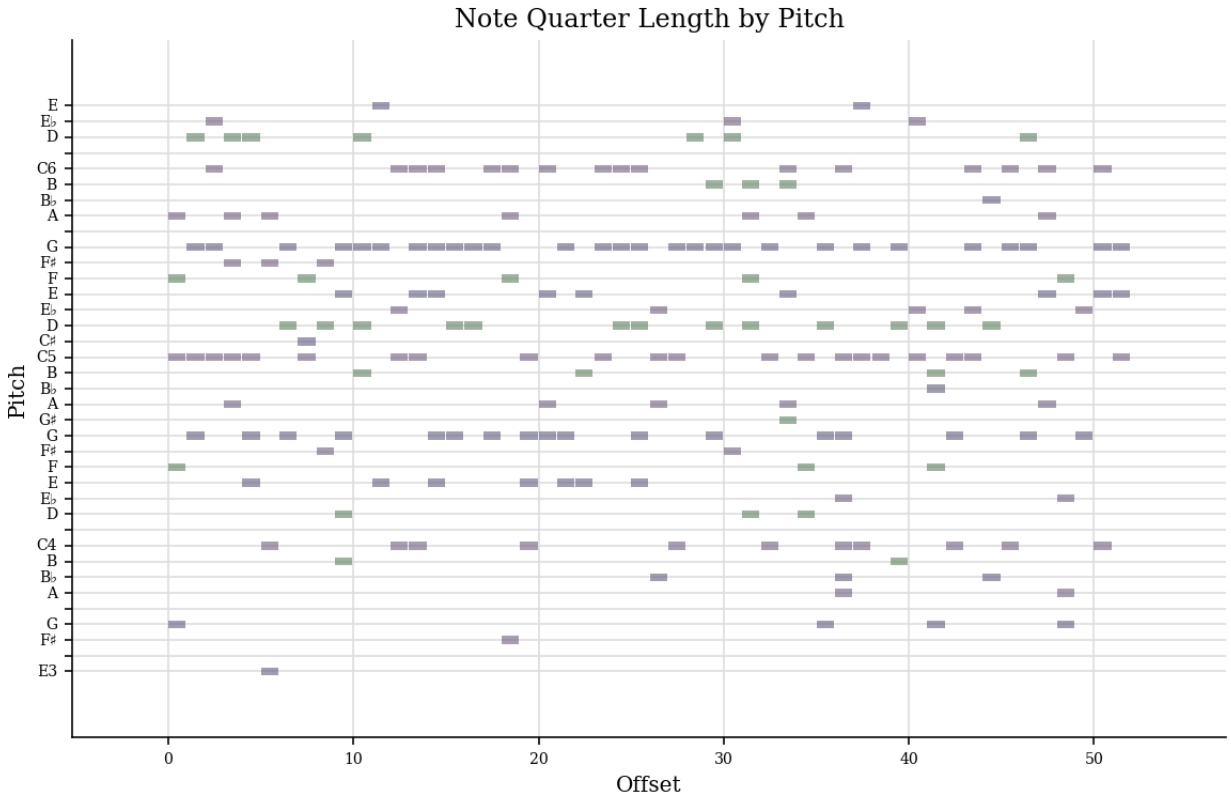
EXAMPLE: DEEP MARKOV MODEL INFERRED USING STEINVI

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:

1. Pathwise Derivatives for Multivariate Distributions Martin Jankowiak and Theofanis Karaletsos (2019)
2. **Structured Inference Networks for Nonlinear State Space Models [arXiv:1609.09869]** Rahul G. Krishnan, Uri Shalit and David Sontag (2016)




```
import argparse

import numpy as np
```

(continues on next page)

(continued from previous 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.kernels import RBFKernel
import numpyro.distributions as dist
from numpyro.examples.datasets import JSB_CHORALES, load_dataset
from numpyro.infer import Predictive, Trace_ELBO
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[:].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:**
        1. Structured Inference Networks for Nonlinear State Space Models [arXiv:1609.
↪09869]
        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"])
```

(continues on next page)

(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)

(continued from previous 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)

(continued from previous 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)),
    }
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
    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)

    vi = SteinVI(
        model,
        guide,
        optax_to_numpyro(chain(adam(1e-2))),
        Trace_ELBO(),
        RBFKernel(),
        num_particles=args.num_particles,
    )

    seqs, rev_seqs, lengths = load_data()
    results = vi.run(
        inf_key,
        args.max_iter,
        seqs,
        rev_seqs,
        lengths,
        gru_dim=args.gru_dim,
        subsample_size=args.subsample_size,
    )
    pred = Predictive(
        model,
        params=results.params,
        num_samples=1,
        batch_ndims=1,
    )
    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=77)
    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-particles", type=int, default=5)
```

(continues on next page)

(continued from previous page)

```
    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)
```

CHAPTER

FORTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z