

---

# NumPyro Documentation

Uber AI Labs

Jan 24, 2021



---

## Contents

---

<b>1</b>	<b>Getting Started with NumPyro</b>	<b>1</b>
<b>2</b>	<b>API Reference</b>	<b>11</b>
<b>3</b>	<b>Bayesian Regression Using NumPyro</b>	<b>139</b>
<b>4</b>	<b>Bayesian Hierarchical Linear Regression</b>	<b>163</b>
<b>5</b>	<b>Example: Baseball Batting Average</b>	<b>173</b>
<b>6</b>	<b>Example: Variational Autoencoder</b>	<b>177</b>
<b>7</b>	<b>Example: Neal’s Funnel</b>	<b>181</b>
<b>8</b>	<b>Example: Stochastic Volatility</b>	<b>185</b>
<b>9</b>	<b>Example: Bayesian Models of Annotation</b>	<b>189</b>
<b>10</b>	<b>Example: Enumerate Hidden Markov Model</b>	<b>195</b>
<b>11</b>	<b>Example: CJS Capture-Recapture Model for Ecological Data</b>	<b>203</b>
<b>12</b>	<b>Bayesian Imputation for Missing Values in Discrete Covariates</b>	<b>209</b>
<b>13</b>	<b>Time Series Forecasting</b>	<b>221</b>
<b>14</b>	<b>Ordinal Regression</b>	<b>229</b>
<b>15</b>	<b>Bayesian Imputation</b>	<b>233</b>
<b>16</b>	<b>Example: Gaussian Process</b>	<b>245</b>
<b>17</b>	<b>Example: Bayesian Neural Network</b>	<b>251</b>
<b>18</b>	<b>Example: Sparse Regression</b>	<b>257</b>
<b>19</b>	<b>Example: Proportion Test</b>	<b>265</b>
<b>20</b>	<b>Example: Generalized Linear Mixed Models</b>	<b>269</b>

<b>21 Example: Hidden Markov Model</b>	<b>273</b>
<b>22 Example: Predator-Prey Model</b>	<b>279</b>
<b>23 Example: Neural Transport</b>	<b>283</b>
<b>24 Indices and tables</b>	<b>289</b>
<b>Python Module Index</b>	<b>291</b>
<b>Index</b>	<b>293</b>

---

## 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 small probabilistic programming library that provides a NumPy backend for [Pyro](#). We rely on [JAX](#) for automatic differentiation and JIT compilation to GPU / CPU. This is an alpha release 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 currently supports Hamiltonian Monte Carlo, including an implementation of the No U-Turn Sampler. 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 for reparameterized distributions together with many flexible (auto)guides for Automatic Differentiation Variational Inference (ADVI).
- **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.

- **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

>>> 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 parameterization](#) for `tau` in our model. This is straightforward to do in NumPyro by using a [TransformedDistribution](#) instance together with a [reparameterization](#) effect handler. Let us rewrite the same model but instead of sampling `theta` from a `Normal(mu, tau)`, we will instead sample it from a base `Normal(0, 1)` distribution that is transformed using an [AffineTransform](#). Note that by doing so, NumPyro runs HMC by generating samples `theta_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
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

(continues on next page)



(continued from previous page)

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.
- [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 hyperparameters of a Gaussian Process.
- [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](#) folder.

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 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](#) if you want to use GPUs on Windows.

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

```
pip install numpyro
```

To use NumPyro on the GPU, you will need to first [install](#) `jax` and `jaxlib` with CUDA support.

To run NumPyro on Cloud TPUs, you can use pip to install NumPyro as above and setup the TPU backend as detailed [here](#).

**Default Platform:** In contrast to JAX, which uses GPU as the default platform, we use CPU as the default platform. You can use [set\\_platform](#) utility to switch to other platforms such as GPU or TPU at the beginning of your program.

You can also install NumPyro from source:

```
git clone https://github.com/pyro-ppl/numpyro.git
# install jax/jaxlib first for CUDA support
pip install -e .[dev] # contains additional dependencies for NumPyro development
```

## 1.5 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):
```

```
x = numpyro.sample('x', dist.Beta(1, 1)) # random.PRNGKey(0) is used
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 be 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 be 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](#).

## 1.6 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.

### ## 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},
  journal={arXiv preprint arXiv:1912.11554},
  year={2019}
}
```

as well as

```
@article{bingham2018pyro,
  author = {Bingham, Eli and Chen, Jonathan P. and Jankowiak, Martin and Obermeyer, Fritz and
    Pradhan, Neeraj and Karaletsos, Theofanis and Singh, Rohit and Szerlip, Paul and
    Horsfall, Paul and Goodman, Noah D.},
  title = {{Pyro: Deep Universal Probabilistic Programming}},
  journal = {arXiv preprint arXiv:1810.09538},
  year = {2018}
}
```



## 2.1 Modeling

### 2.1.1 Pyro Primitives

#### **param**

**param** (*name*, *init\_value=None*, *\*\*kwargs*)

Annotate the given site as an optimizable parameter for use with `jax.experimental.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.

## sample

**sample** (*name*, *fn*, *obs=None*, *rng\_key=None*, *sample\_shape=()*, *infer=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.

**Returns** sample from the stochastic *fn*.

## 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 leftmost available dim is allocated.



## plate\_stack

**plate\_stack** (*prefix, sizes, rightmost\_dim=-1*)

Create a contiguous stack of *plates* 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.

## subsample

**subsample** (*data, event\_dim*)

EXPERIMENTAL Subsampling statement to subsample data based on enclosing *plates*.

This is typically called on arguments to `model()` when subsampling is performed automatically by *plates* 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 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`

## 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.

## 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.

## 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.

## 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.

## flax\_module

**flax\_module** (*name*, *nn\_module*, \*, *input\_shape*=None, \*\**kwargs*)

Declare a `flax` 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\_module** (*flax.nn.Module*) – a `flax` Module which has `.init` and `.apply` methods
- **input\_shape** (*tuple*) – shape of the input taken by the neural network.
- **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.

## haiku\_module

**haiku\_module** (*name*, *nn\_module*, \*, *input\_shape*=None, \*\**kwargs*)

Declare a haiku 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\_module** (*haiku.Module*) – a *haiku* Module which has `.init` and `.apply` methods
- **input\_shape** (*tuple*) – shape of the input taken by the neural network.
- **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.

## random\_flax\_module

**random\_flax\_module** (*name*, *nn\_module*, *prior*, \*, *input\_shape*=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(nn.Module):
    def apply(self, x):
        return nn.Dense(x, 1, bias=False, name='dense')

class B(nn.Module):
    def apply(self, x):
        return A(x, name='inner')
```

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.nn.Module** – the module to be registered with NumPyro
- **prior** – 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.nn.Dense.partial(features=1),
                        prior={"bias": dist.Cauchy(), "kernel":
↪ dist.Normal()},
                        input_shape=(4,))
```

- **input\_shape** (*tuple*) – shape of the input taken by the neural network.
- **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 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):
...     def apply(self, x, n_units):
...         x = nn.Dense(x[...], None, features=n_units)
...         x = nn.relu(x)
...         x = nn.Dense(x, features=n_units)
...         x = nn.relu(x)
...         mean = nn.Dense(x, features=1)
...         rho = nn.Dense(x, features=1)
...         return mean.squeeze(), rho.squeeze()
>>>
>>> def generate_data(n_samples):
...     x = np.random.normal(size=n_samples)
...     y = np.cos(x * 3) + np.random.normal(size=n_samples) * np.abs(x) / 2
...     return x, y
>>>
>>> def model(x, y=None, batch_size=None):
...     module = Net.partial(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
>>> params, losses = svi.run(random.PRNGKey(0), n_iterations, x_train, y_train,
↳batch_size=256)
>>> 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

```

## random\_haiku\_module

**random\_haiku\_module** (*name*, *nn\_module*, *prior*, \*, *input\_shape*=None, \*\**kwargs*)

A primitive to place a prior over the parameters of the Haiku module *nn\_module*.

### Parameters

- **name** (*str*) – name of NumPyro module
- **haiku.Module** – the module to be registered with NumPyro
- **prior** – 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,))
```

- **input\_shape** (*tuple*) – shape of the input taken by the neural network.

**Returns** a sampled module

## 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  $O(\log(\text{length}))$ .

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

- **init sites:** those sites belong to the first *history* traces of  $f$ . Sites at the  $i$ -th trace will have name prefixed with `'_PREV_' * (2 * history - 1 - i)`.
- **scanned sites:** those sites collect the values of the remaining *scan* loop over  $f$ . An additional 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”.

## 2.1.2 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_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]
>>> 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

(continues on next page)

(continued from previous page)

```

    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

```

## block

**class block** (*fn=None, hide\_fn=None, hide=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.

### Example:

```

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

>>> def model():
...     a = numpyro.sample('a', dist.Normal(0., 1.))
...     return numpyro.sample('b', dist.Normal(a, 1.))

>>> model = seed(model, random.PRNGKey(0))
>>> block_all = block(model)
>>> block_a = block(model, lambda site: site['name'] == 'a')
>>> trace_block_all = trace(block_all).get_trace()
>>> assert not {'a', 'b'}.intersection(trace_block_all.keys())
>>> trace_block_a = trace(block_a).get_trace()
>>> assert 'a' not in trace_block_a
>>> assert 'b' in trace_block_a

```



`process_message(msg)`

## 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)`

## 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.))

>>> 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)`

## 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*)

**infer\_config**

**class infer\_config** (*fn=None, config\_fn=None*)

Bases: `numpyro.primitives.Messenger`

Given a callable *fn* that contains Pyro 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*)

**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*)

### 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*)

### 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*)

## replay

**class replay** (*fn=None, guide\_trace=None*)

Bases: `numpyro.primitives.Messenger`

Given a callable *fn* and an execution trace *guide\_trace*, return a callable which substitutes *sample* calls in *fn* with values from the corresponding site names in *guide\_trace*.

### Parameters

- **fn** – Python callable with NumPyro primitives.
- **guide\_trace** – an `OrderedDict` containing execution metadata.

### Example

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

>>> def model():
...     numpyro.sample('a', dist.Normal(0., 1.))

>>> exec_trace = trace(seed(model, random.PRNGKey(0))).get_trace()
>>> print(exec_trace['a']['value'])
-0.20584235
>>> replayed_trace = trace(replay(model, exec_trace)).get_trace()
>>> print(exec_trace['a']['value'])
-0.20584235
>>> assert replayed_trace['a']['value'] == exec_trace['a']['value']
```

**process\_message** (*msg*)

## 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*)

## scope

**class scope** (*fn=None, prefix="", divider='/'*)

Bases: `numpyro.primitives.Messenger`

This handler prepend a prefix followed by a divider to the name of sample sites.

Example:

```
.. doctest::
```

```

>>> 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 '/'

**process\_message** (*msg*)

## 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.))

```

(continues on next page)

(continued from previous page)

```
>>> # as function decorator (/modifier)
>>> y = handlers.seed(model, rng_seed=1) ()
>>> assert x == y
```

```
process_message(msg)
```

## 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.

### 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)
```

## 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
>>> import pprint as pp
```

(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()
>>> 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.

## 2.2 Distributions

### 2.2.1 Base Distribution

#### 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

**is\_discrete** = False

**reparametrized\_params** = []

**tree\_flatten**()

**classmethod tree\_unflatten**(*aux\_data, params*)

**static set\_default\_validate\_args**(*value*)

**batch\_shape**

Returns the shape over which the distribution parameters are batched.

**Returns** batch shape of the distribution.

**Return type** `tuple`

**event\_shape**

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

**Returns** event shape of the distribution.

**Return type** `tuple`

**event\_dim**

**Returns** Number of dimensions of individual events.

**Return type** `int`

**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*

**mean**

Mean of the distribution.

**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*

## ExpandedDistribution

**class** *ExpandedDistribution* (*base\_dist*, *batch\_shape=()*)

Bases: *numpyro.distributions.distribution.Distribution*

**arg\_constraints** = {}

**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.

**is\_discrete**

*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.

**has\_rsample**

**rsample** (*key*, *sample\_shape=()*)

**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.

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**tree\_flatten** ()

**classmethod tree\_unflatten** (`aux_data, params`)

## 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)))
...
...     # 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 = {}
```

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

```
tree_flatten()
```

### 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 = {}
```

```
support
```

**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.

**is\_discrete**

`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.

**reparameterized\_params****mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**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*)

**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** = {}

**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.

**is\_discrete**

`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.

**has\_rsample**

**rsample** (*key*, *sample\_shape*=())

**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.

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**tree\_flatten** ()

**classmethod tree\_unflatten** (*aux\_data*, *params*)

## 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** = {}

**has\_rsample**

**rsample** (*key*, *sample\_shape=()*)

**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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**tree\_flatten** ()

## 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': <numpyro.distributions.constraints.\_Real object>, 'v'

**reparameterized\_params** = ['v', 'log\_density']

**is\_discrete** = True

**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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**tree\_flatten** ()

**classmethod tree\_unflatten** (*aux\_data, params*)

## 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': <numpyro.distributions.constraints.\_Real object>}

**support** = <numpyro.distributions.constraints.\_Real object>

**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`

## 2.2.2 Continuous Distributions

### Beta

**class Beta** (*concentration1, concentration0, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'concentration0': `<numpyro.distributions.constraints._GreaterThan object>`

**reparametrized\_params** = ['concentration1', 'concentration0']

**support** = `<numpyro.distributions.constraints._Interval object>`

**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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

### Cauchy

**class Cauchy** (*loc=0.0, scale=1.0, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'loc': `<numpyro.distributions.constraints._Real object>`, 'scale': `<`

**support** = `<numpyro.distributions.constraints._Real object>`

**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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## Chi2

**class** `Chi2` (*df*, *validate\_args*=None)

Bases: `numpyro.distributions.continuous.Gamma`

**arg\_constraints** = {'df': <numpyro.distributions.constraints.\_GreaterThan object>}

**reparametrized\_params** = ['df']

## Dirichlet

**class** `Dirichlet` (*concentration*, *validate\_args*=None)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'concentration': <numpyro.distributions.constraints.\_GreaterThan ob

**reparametrized\_params** = ['concentration']

**support** = <numpyro.distributions.constraints.\_Simplex object>

**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*)

**mean**

Mean of the distribution.

**variance**  
Variance of the distribution.

## Exponential

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

    reparametrized_params = ['rate']

    arg_constraints = {'rate': <numpyro.distributions.constraints._GreaterThan object>}

    support = <numpyro.distributions.constraints._GreaterThan object>

    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)

mean
    Mean of the distribution.

variance
    Variance of the distribution.
```

## Gamma

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

    arg_constraints = {'concentration': <numpyro.distributions.constraints._GreaterThan ob

    support = <numpyro.distributions.constraints._GreaterThan object>

    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)
```

**mean**  
Mean of the distribution.

**variance**  
Variance of the distribution.

## Gumbel

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

    arg_constraints = {'loc': <numpyro.distributions.constraints._Real object>, 'scale': <
    support = <numpyro.distributions.constraints._Real object>
    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)

    mean
        Mean of the distribution.

    variance
        Variance of the distribution.
```

## GaussianRandomWalk

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

    arg_constraints = {'scale': <numpyro.distributions.constraints._GreaterThan object>}
    support = <numpyro.distributions.constraints._IndependentConstraint object>
    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)`

`mean`

Mean of the distribution.

`variance`

Variance of the distribution.

`tree_flatten ()`

`classmethod tree_unflatten (aux_data, params)`

## HalfCauchy

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

Bases: `numpyro.distributions.distribution.Distribution`

`reparametrized_params = ['scale']`

`support = <numpyro.distributions.constraints._GreaterThan object>`

`arg_constraints = {'scale': <numpyro.distributions.constraints._GreaterThan object>}`

`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)`

`mean`

Mean of the distribution.

`variance`

Variance of the distribution.

## HalfNormal

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

Bases: `numpyro.distributions.distribution.Distribution`

`reparametrized_params = ['scale']`

`support = <numpyro.distributions.constraints._GreaterThan object>`

`arg_constraints = {'scale': <numpyro.distributions.constraints._GreaterThan object>}`

`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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## 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](https://en.wikipedia.org/wiki/Inverse-gamma_distribution))

---

**arg\_constraints** = {'concentration': `<numpyro.distributions.constraints._GreaterThan ob`

**reparametrized\_params** = ['concentration', 'rate']

**support** = `<numpyro.distributions.constraints._GreaterThan object>`

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**tree\_flatten** ()

## Laplace

**class Laplace** (*loc=0.0, scale=1.0, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'loc': `<numpyro.distributions.constraints._Real object>`, 'scale': `<`

**support** = `<numpyro.distributions.constraints._Real object>`

**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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## 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  $\eta$  to make the probability of the correlation matrix  $M$  proportional to  $\det(M)^{\eta-1}$ . Because of that, when `concentration == 1`, we have a uniform distribution over correlation matrices.

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

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

### 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': `<numpyro.distributions.constraints._GreaterThan ob`

**reparametrized\_params** = ['concentration']

**support** = `<numpyro.distributions.constraints._CorrMatrix object>`

**mean**

Mean of the distribution.

**tree\_flatten** ()

**classmethod tree\_unflatten** (*aux\_data, params*)

## 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  $\eta$  to make the probability of the correlation matrix  $M$  generated from a Cholesky

factor proportional to  $\det(M)^{n-1}$ . 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 determinant). This is useful when we know a priori that the underlying variables are not correlated.

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

#### 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': <numpyro.distributions.constraints._GreaterThan ob
```

```
reparametrized_params = ['concentration']
```

```
support = <numpyro.distributions.constraints._CorrCholesky object>
```

```
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)
```

```
tree_flatten ()
```

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

## LogNormal

```
class LogNormal (loc=0.0, scale=1.0, validate_args=None)
```

Bases: `numpyro.distributions.distribution.TransformedDistribution`

```
arg_constraints = {'loc': <numpyro.distributions.constraints._Real object>, 'scale': <
```

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

**mean**

Mean of the distribution.



**variance**  
Variance of the distribution.

**tree\_flatten()**

## Logistic

**class Logistic** (*loc=0.0, scale=1.0, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'loc': `<numpyro.distributions.constraints._Real object>`, 'scale': `<`

**support** = `<numpyro.distributions.constraints._Real object>`

**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*)

**mean**  
Mean of the distribution.

**variance**  
Variance of the distribution.

## 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': `<numpyro.distributions.constraints._PositiveDe`

**support** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**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)

covariance_matrix

precision_matrix

mean
    Mean of the distribution.

variance
    Variance of the distribution.

tree_flatten ()

classmethod tree_unflatten (aux_data, params)
```

## LowRankMultivariateNormal

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

    arg_constraints = {'cov_diag': <numpyro.distributions.constraints._IndependentConstraint object>, 'cov_factor': <numpyro.distributions.constraints._IndependentConstraint object>, 'loc': <numpyro.distributions.constraints._Real object>}
    support = <numpyro.distributions.constraints._IndependentConstraint object>
    reparametrized_params = ['loc', 'cov_factor', 'cov_diag']

    mean
        Mean of the distribution.

    variance

    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)

    entropy ()
```

## Normal

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

    arg_constraints = {'loc': <numpyro.distributions.constraints._Real object>, 'scale': <numpyro.distributions.constraints._Positive object>}
```

**support** = `<numpyro.distributions.constraints._Real object>`

**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*)

**icdf** (*q*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## Pareto

**class Pareto** (*scale*, *alpha*, *validate\_args*=None)

Bases: `numpyro.distributions.distribution.TransformedDistribution`

**arg\_constraints** = `{'alpha': <numpyro.distributions.constraints._GreaterThan object>, 'scale': <numpyro.distributions.constraints._GreaterThan object>}`

**reparametrized\_params** = `['scale', 'alpha']`

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

**tree\_flatten** ()

## StudentT

**class StudentT** (*df*, *loc*=0.0, *scale*=1.0, *validate\_args*=None)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = `{'df': <numpyro.distributions.constraints._GreaterThan object>, 'loc': <numpyro.distributions.constraints._Real object>, 'scale': <numpyro.distributions.constraints._Real object>}`

**support** = `<numpyro.distributions.constraints._Real object>`

**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)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**TruncatedCauchy**

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

Bases: `numpyro.distributions.distribution.TransformedDistribution`

```
arg_constraints = {'loc': <numpyro.distributions.constraints._Real object>, 'low': <num
```

```
reparametrized_params = ['low', 'loc', 'scale']
```

**support**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

```
tree_flatten()
```

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

**TruncatedNormal**

```
class TruncatedNormal (low=0.0, loc=0.0, scale=1.0, validate_args=None)
```

Bases: `numpyro.distributions.distribution.TransformedDistribution`

```
arg_constraints = {'loc': <numpyro.distributions.constraints._Real object>, 'low': <num
```

```
reparametrized_params = ['low', 'loc', 'scale']
```

**support**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

```
tree_flatten()
```

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

## 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 = <numpyro.distributions.constraints._Interval object>
    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)
    tree_flatten ()
    classmethod tree_unflatten (aux_data, params)
```

## Uniform

```
class Uniform (low=0.0, high=1.0, validate_args=None)
    Bases: numpyro.distributions.distribution.TransformedDistribution

    arg_constraints = {'high': <numpyro.distributions.constraints._Dependent object>, 'low'
    reparametrized_params = ['low', 'high']
    support
    mean
        Mean of the distribution.
    variance
        Variance of the distribution.
    tree_flatten ()
    classmethod tree_unflatten (aux_data, params)
```

## 2.2.3 Discrete Distributions

### Bernoulli

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

## BernoulliLogits

```
class BernoulliLogits (logits=None, validate_args=None)
    Bases: numpyro.distributions.distribution.Distribution

    arg_constraints = {'logits': <numpyro.distributions.constraints._Real object>}
    support = <numpyro.distributions.constraints._Boolean object>
    has_enumerate_support = True
    is_discrete = 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)

    probs

    mean
        Mean of the distribution.

    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.
```

## BernoulliProbs

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

    arg_constraints = {'probs': <numpyro.distributions.constraints._Interval object>}
    support = <numpyro.distributions.constraints._Boolean object>
    has_enumerate_support = True
    is_discrete = 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*)

**logits**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**enumerate\_support** (*expand=True*)

Returns an array with shape  $len(support) \times batch\_shape$  containing all values in the support.

## 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': `<numpyro.distributions.constraints._GreaterThan of`

**has\_enumerate\_support** = `True`

**is\_discrete** = `True`

**enumerate\_support** (*expand=True*)

Returns an array with shape  $len(support) \times batch\_shape$  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*)

**mean**

Mean of the distribution.

**variance**  
Variance of the distribution.

**support**

## Binomial

**Binomial** (*total\_count=1, probs=None, logits=None, validate\_args=None*)

## BinomialLogits

**class BinomialLogits** (*logits, total\_count=1, validate\_args=None*)  
Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'logits': <numpyro.distributions.constraints.\_Real object>, 'total\_

**has\_enumerate\_support** = True

**is\_discrete** = 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*)

**probs**

**mean**  
Mean of the distribution.

**variance**  
Variance of the distribution.

**support**

## BinomialProbs

**class BinomialProbs** (*probs, total\_count=1, validate\_args=None*)  
Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'probs': <numpyro.distributions.constraints.\_Interval object>, 'tot

**has\_enumerate\_support** = True

**is\_discrete** = 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*)

**logits**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

**enumerate\_support** (*expand=True*)

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

## Categorical

**Categorical** (*probs=None*, *logits=None*, *validate\_args=None*)

## CategoricalLogits

**class CategoricalLogits** (*logits*, *validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'logits': `<numpyro.distributions.constraints._IndependentConstraint`

**has\_enumerate\_support** = `True`

**is\_discrete** = `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*)

**probs**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

**enumerate\_support** (*expand=True*)

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

## CategoricalProbs

**class CategoricalProbs** (*probs, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'probs': <numpyro.distributions.constraints.\_Simplex object>}

**has\_enumerate\_support** = True

**is\_discrete** = True

**sample** (*key, sample\_shape=()*)

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

### Parameters

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

**Returns** an array of shape  $\text{sample\_shape} + \text{batch\_shape} + \text{event\_shape}$

**Return type** `numpy.ndarray`

**log\_prob** (*\*args, \*\*kwargs*)

**logits**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

**enumerate\_support** (*expand=True*)

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

## 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': <numpyro.distributions.constraints.\_GreaterThan ob

**is\_discrete** = 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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

## 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': <numpyro.distributions.constraints.\_GreaterThan ob

**support** = <numpyro.distributions.constraints.\_IntegerGreaterThan object>

**is\_discrete** = 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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## Geometric

**Geometric** (*probs=None, logits=None, validate\_args=None*)

### GeometricLogits

**class GeometricLogits** (*logits, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'logits': `<numpyro.distributions.constraints._Real object>`}

**support** = `<numpyro.distributions.constraints._IntegerGreaterThan object>`

**is\_discrete** = `True`

**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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

### GeometricProbs

**class GeometricProbs** (*probs, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'probs': `<numpyro.distributions.constraints._Interval object>`}

**support** = `<numpyro.distributions.constraints._IntegerGreaterThan object>`

**is\_discrete** = `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*)

**logits**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## Multinomial

**Multinomial** (*total\_count*=1, *probs*=None, *logits*=None, *validate\_args*=None)

## MultinomialLogits

**class MultinomialLogits** (*logits*, *total\_count*=1, *validate\_args*=None)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'logits': `<numpyro.distributions.constraints._IndependentConstraint`

**is\_discrete** = 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*)

**probs**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

## MultinomialProbs

**class MultinomialProbs** (*probs, total\_count=1, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'probs': `<numpyro.distributions.constraints._Simplex object>`, 'total\_count': `<numpyro.distributions.constraints._GreaterThan object>`}

**is\_discrete** = 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*)

**logits**

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

**support**

## 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': `<numpyro.distributions.constraints._OrderedVector object>`, 'rate': `<numpyro.distributions.constraints._GreaterThan object>`}

## Poisson

**class Poisson** (*rate, validate\_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

**arg\_constraints** = {'rate': `<numpyro.distributions.constraints._GreaterThan object>`}

**support** = <numpyro.distributions.constraints.\_IntegerGreaterThan object>

**is\_discrete** = 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*)

**mean**

Mean of the distribution.

**variance**

Variance of the distribution.

## PRNGIdentity

**class** `PRNGIdentity`

Bases: `numpyro.distributions.distribution.Distribution`

Distribution over `PRNGKey()`. This can be used to draw a batch of `PRNGKey()` using the *seed* handler. Only *sample* method is supported.

**is\_discrete** = 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`

## ZeroInflatedPoisson

**class** `ZeroInflatedPoisson` (*gate*, *rate*=1.0, *validate\_args*=None)

Bases: `numpyro.distributions.distribution.Distribution`

A Zero Inflated Poisson distribution.

#### Parameters

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

```
arg_constraints = {'gate': <numpyro.distributions.constraints._Interval object>, 'rate'  
support = <numpyro.distributions.constraints._IntegerGreaterThan object>  
is_discrete = 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)
```

**mean**

**variance**

## 2.2.4 Directional Distributions

### VonMises

```
class VonMises (loc, concentration, validate_args=None)
```

Bases: *numpyro.distributions.distribution.Distribution*

```
arg_constraints = {'concentration': <numpyro.distributions.constraints._GreaterThan ob
```

```
reparametrized_params = ['loc']
```

```
support = <numpyro.distributions.constraints._Interval object>
```

```
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)
```

**mean**

Computes circular mean of distribution. NOTE: same as location when mapped to support  $[-\pi, \pi]$

**variance**

Computes circular variance of distribution

## 2.2.5 TensorFlow Distributions

Thin wrappers around TensorFlow Probability (TFP) distributions. For details on the TFP distribution interface, see [its Distribution docs](#).



## BijectorConstraint

**class** `BijectorConstraint` (*bijector*)

A constraint which is codomain of a TensorFlow bijector.

**Parameters** `bijector` (*Bijector*) – a TensorFlow bijector

## BijectorTransform

**class** `BijectorTransform` (*bijector*)

A wrapper for TensorFlow bijectors to make them compatible with NumPyro's transforms.

**Parameters** `bijector` (*Bijector*) – a TensorFlow bijector

## TFPDistributionMixin

**class** `TFPDistributionMixin` (*batch\_shape=()*, *event\_shape=()*, *validate\_args=None*)

A mixin layer to make TensorFlow Probability (TFP) distribution compatible with NumPyro internal.

## Autoregressive

**class** `Autoregressive` (*distribution\_fn*, *sample0=None*, *num\_steps=None*, *validate\_args=False*, *allow\_nan\_stats=True*, *name='Autoregressive'*)

Wraps `tensorflow_probability.substrates.jax.distributions.autoregressive.Autoregressive` with `TFPDistributionMixin`.

## BatchReshape

**class** `BatchReshape` (*distribution*, *batch\_shape*, *validate\_args=False*, *allow\_nan\_stats=True*, *name=None*)

Wraps `tensorflow_probability.substrates.jax.distributions.batch_reshape.BatchReshape` with `TFPDistributionMixin`.

## Bates

**class** `Bates` (*total\_count*, *low=0.0*, *high=1.0*, *validate\_args=False*, *allow\_nan\_stats=True*, *name='Bates'*)

Wraps `tensorflow_probability.substrates.jax.distributions.bates.Bates` with `TFPDistributionMixin`.

## Bernoulli

**class** `Bernoulli` (*logits=None*, *probs=None*, *dtype=<class 'jax.\_src.numpy.lax\_numpy.int32'>*, *validate\_args=False*, *allow\_nan\_stats=True*, *name='Bernoulli'*)

Wraps `tensorflow_probability.substrates.jax.distributions.bernoulli.Bernoulli` with `TFPDistributionMixin`.

## Beta

**class** `Beta` (*concentration1*, *concentration0*, *validate\_args=False*, *allow\_nan\_stats=True*, *name='Beta'*)

Wraps `tensorflow_probability.substrates.jax.distributions.beta.Beta` with `TFPDistributionMixin`.

## BetaBinomial

```
class BetaBinomial (total_count, concentration1, concentration0, validate_args=False, allow_nan_stats=True, name='BetaBinomial')
    Wraps tensorflow_probability.substrates.jax.distributions.beta_binomial.BetaBinomial with
    TFPDistributionMixin.
```

## Binomial

```
class Binomial (total_count, logits=None, probs=None, validate_args=False, allow_nan_stats=True, name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.binomial.Binomial with
    TFPDistributionMixin.
```

## Blockwise

```
class Blockwise (distributions, dtype_override=None, validate_args=False, allow_nan_stats=False, name='Blockwise')
    Wraps tensorflow_probability.substrates.jax.distributions.blockwise.Blockwise with
    TFPDistributionMixin.
```

## Categorical

```
class Categorical (logits=None, probs=None, dtype=<class 'jax._src.numpy.lax_numpy.int32'>, validate_args=False, allow_nan_stats=True, name='Categorical')
    Wraps tensorflow_probability.substrates.jax.distributions.categorical.Categorical with
    TFPDistributionMixin.
```

## Cauchy

```
class Cauchy (loc, scale, validate_args=False, allow_nan_stats=True, name='Cauchy')
    Wraps tensorflow_probability.substrates.jax.distributions.cauchy.Cauchy with TFPDistributionMixin.
```

## Chi

```
class Chi (df, validate_args=False, allow_nan_stats=True, name='Chi')
    Wraps tensorflow_probability.substrates.jax.distributions.chi.Chi with TFPDistributionMixin.
```

## Chi2

```
class Chi2 (df, validate_args=False, allow_nan_stats=True, name='Chi2')
    Wraps tensorflow_probability.substrates.jax.distributions.chi2.Chi2 with TFPDistributionMixin.
```

## CholeskyLKJ

```
class CholeskyLKJ (dimension, concentration, validate_args=False, allow_nan_stats=True, name='CholeskyLKJ')
    Wraps tensorflow_probability.substrates.jax.distributions.cholesky_lkj.CholeskyLKJ with
    TFPDistributionMixin.
```

## ContinuousBernoulli

```
class ContinuousBernoulli (logits=None, probs=None, lims=(0.499, 0.501), dtype=<class
                        'jax._src.numpy.lax_numpy.float32'>, validate_args=False, al-
                        low_nan_stats=True, name='ContinuousBernoulli')
    Wraps tensorflow_probability.substrates.jax.distributions.continuous_bernoulli.ContinuousBernoulli with
    TFPDistributionMixin.
```

## DeterminantalPointProcess

```
class DeterminantalPointProcess (eigenvalues, eigenvectors, validate_args=False, al-
                        low_nan_stats=False, name='DeterminantalPointProcess')
    Wraps tensorflow_probability.substrates.jax.distributions.dpp.DeterminantalPointProcess with
    TFPDistributionMixin.
```

## Deterministic

```
class Deterministic (loc, atol=None, rtol=None, validate_args=False, allow_nan_stats=True,
                    name='Deterministic')
    Wraps tensorflow_probability.substrates.jax.distributions.deterministic.Deterministic with
    TFPDistributionMixin.
```

## Dirichlet

```
class Dirichlet (concentration, validate_args=False, allow_nan_stats=True, name='Dirichlet')
    Wraps tensorflow_probability.substrates.jax.distributions.dirichlet.Dirichlet with
    TFPDistributionMixin.
```

## DirichletMultinomial

```
class DirichletMultinomial (total_count, concentration, validate_args=False, al-
                        low_nan_stats=True, name='DirichletMultinomial')
    Wraps tensorflow_probability.substrates.jax.distributions.dirichlet_multinomial.DirichletMultinomial with
    TFPDistributionMixin.
```

## DoublesidedMaxwell

```
class DoublesidedMaxwell (loc, scale, validate_args=False, allow_nan_stats=True,
                        name='doublesided_maxwell')
    Wraps tensorflow_probability.substrates.jax.distributions.doublesided_maxwell.DoublesidedMaxwell with
    TFPDistributionMixin.
```

## Empirical

```
class Empirical (samples, event_ndims=0, validate_args=False, allow_nan_stats=True,
                name='Empirical')
    Wraps tensorflow_probability.substrates.jax.distributions.empirical.Empirical with
    TFPDistributionMixin.
```

## ExpGamma

```
class ExpGamma (concentration,      rate=None,      log_rate=None,      validate_args=False,      al-  
                  low_nan_stats=True, name='ExpGamma')  
    Wraps      tensorflow_probability.substrates.jax.distributions.exp_gamma.ExpGamma      with  
    TFPDistributionMixin.
```

## ExpInverseGamma

```
class ExpInverseGamma (concentration, scale=None, log_scale=None, validate_args=False, al-  
                        low_nan_stats=True, name='ExpInverseGamma')  
    Wraps      tensorflow_probability.substrates.jax.distributions.exp_gamma.ExpInverseGamma      with  
    TFPDistributionMixin.
```

## ExpRelaxedOneHotCategorical

```
class ExpRelaxedOneHotCategorical (temperature,      logits=None,      probs=None,      vali-  
                                   idate_args=False,      allow_nan_stats=True,  
                                   name='ExpRelaxedOneHotCategorical')  
    Wraps tensorflow_probability.substrates.jax.distributions.relaxed_onehot_categorical.ExpRelaxedOneHotCategorical  
    with TFPDistributionMixin.
```

## Exponential

```
class Exponential (rate, force_probs_to_zero_outside_support=False, validate_args=False, al-  
                   low_nan_stats=True, name='Exponential')  
    Wraps      tensorflow_probability.substrates.jax.distributions.exponential.Exponential      with  
    TFPDistributionMixin.
```

## ExponentiallyModifiedGaussian

```
class ExponentiallyModifiedGaussian (loc,      scale,      rate,      vali-  
                                     date_args=False,      allow_nan_stats=True,  
                                     name='ExponentiallyModifiedGaussian')  
    Wraps tensorflow_probability.substrates.jax.distributions.exponentially_modified_gaussian.ExponentiallyModifiedGaussian  
    with TFPDistributionMixin.
```

## FiniteDiscrete

```
class FiniteDiscrete (outcomes, logits=None, probs=None, rtol=None, atol=None, vali-  
                      date_args=False, allow_nan_stats=True, name='FiniteDiscrete')  
    Wraps      tensorflow_probability.substrates.jax.distributions.finite_discrete.FiniteDiscrete      with  
    TFPDistributionMixin.
```

## Gamma

```
class Gamma (concentration, rate=None, log_rate=None, validate_args=False, allow_nan_stats=True,  
             name='Gamma')  
    Wraps tensorflow_probability.substrates.jax.distributions.gamma.Gamma with TFPDistributionMixin.
```

### GammaGamma

```
class GammaGamma (concentration, mixing_concentration, mixing_rate, validate_args=False, allow_nan_stats=True, name='GammaGamma')
```

Wraps `tensorflow_probability.substrates.jax.distributions.gamma_gamma.GammaGamma` with `TFPDistributionMixin`.

### GaussianProcess

```
class GaussianProcess (kernel, index_points=None, mean_fn=None, observation_noise_variance=0.0, jitter=1e-06, validate_args=False, allow_nan_stats=False, name='GaussianProcess')
```

Wraps `tensorflow_probability.substrates.jax.distributions.gaussian_process.GaussianProcess` with `TFPDistributionMixin`.

### GaussianProcessRegressionModel

```
class GaussianProcessRegressionModel (kernel, index_points=None, observation_index_points=None, observations=None, observation_noise_variance=0.0, predictive_noise_variance=None, mean_fn=None, jitter=1e-06, validate_args=False, allow_nan_stats=False, name='GaussianProcessRegressionModel')
```

Wraps `tensorflow_probability.substrates.jax.distributions.gaussian_process_regression_model.GaussianProcessRegressionModel` with `TFPDistributionMixin`.

### GeneralizedExtremeValue

```
class GeneralizedExtremeValue (loc, scale, concentration, validate_args=False, allow_nan_stats=True, name='GeneralizedExtremeValue')
```

Wraps `tensorflow_probability.substrates.jax.distributions.gev.GeneralizedExtremeValue` with `TFPDistributionMixin`.

### GeneralizedNormal

```
class GeneralizedNormal (loc, scale, power, validate_args=False, allow_nan_stats=True, name='GeneralizedNormal')
```

Wraps `tensorflow_probability.substrates.jax.distributions.generalized_normal.GeneralizedNormal` with `TFPDistributionMixin`.

### GeneralizedPareto

```
class GeneralizedPareto (loc, scale, concentration, validate_args=False, allow_nan_stats=True, name=None)
```

Wraps `tensorflow_probability.substrates.jax.distributions.generalized_pareto.GeneralizedPareto` with `TFPDistributionMixin`.

## Geometric

```
class Geometric (logits=None, probs=None, force_probs_to_zero_outside_support=False, validate_args=False, allow_nan_stats=True, name='Geometric')
    Wraps      tensorflow_probability.substrates.jax.distributions.geometric.Geometric      with
    TFPDistributionMixin.
```

## Gumbel

```
class Gumbel (loc, scale, validate_args=False, allow_nan_stats=True, name='Gumbel')
    Wraps tensorflow_probability.substrates.jax.distributions.gumbel.Gumbel with TFPDistributionMixin.
```

## HalfCauchy

```
class HalfCauchy (loc, scale, validate_args=False, allow_nan_stats=True, name='HalfCauchy')
    Wraps      tensorflow_probability.substrates.jax.distributions.half_cauchy.HalfCauchy      with
    TFPDistributionMixin.
```

## HalfNormal

```
class HalfNormal (scale, validate_args=False, allow_nan_stats=True, name='HalfNormal')
    Wraps      tensorflow_probability.substrates.jax.distributions.half_normal.HalfNormal      with
    TFPDistributionMixin.
```

## HalfStudentT

```
class HalfStudentT (df, loc, scale, validate_args=False, allow_nan_stats=True,
                    name='HalfStudentT')
    Wraps      tensorflow_probability.substrates.jax.distributions.half_student_t.HalfStudentT      with
    TFPDistributionMixin.
```

## HiddenMarkovModel

```
class HiddenMarkovModel (initial_distribution, transition_distribution, observation_distribution, num_steps, validate_args=False, allow_nan_stats=True,
                        time_varying_transition_distribution=False, time_varying_observation_distribution=False,
                        name='HiddenMarkovModel')
    Wraps tensorflow_probability.substrates.jax.distributions.hidden_markov_model.HiddenMarkovModel with
    TFPDistributionMixin.
```

## Horseshoe

```
class Horseshoe (scale, validate_args=False, allow_nan_stats=True, name='Horseshoe')
    Wraps      tensorflow_probability.substrates.jax.distributions.horseshoe.Horseshoe      with
    TFPDistributionMixin.
```

## Independent

```
class Independent (distribution, reinterpreted_batch_ndims=None, validate_args=False, experimental_use_kahan_sum=False, name=None)
    Wraps      tensorflow_probability.substrates.jax.distributions.independent.Independent      with
    TFPDistributionMixin.
```

## InverseGamma

```
class InverseGamma (concentration, scale=None, validate_args=False, allow_nan_stats=True, name='InverseGamma')
    Wraps      tensorflow_probability.substrates.jax.distributions.inverse_gamma.InverseGamma      with
    TFPDistributionMixin.
```

## InverseGaussian

```
class InverseGaussian (loc, concentration, validate_args=False, allow_nan_stats=True, name='InverseGaussian')
    Wraps      tensorflow_probability.substrates.jax.distributions.inverse_gaussian.InverseGaussian      with
    TFPDistributionMixin.
```

## JohnsonSU

```
class JohnsonSU (skewness, tailweight, loc, scale, validate_args=False, allow_nan_stats=True, name=None)
    Wraps      tensorflow_probability.substrates.jax.distributions.johnson_su.JohnsonSU      with
    TFPDistributionMixin.
```

## JointDistribution

```
class JointDistribution (dtype, reparameterization_type, validate_args, allow_nan_stats, parameters=None, graph_parents=None, name=None)
    Wraps      tensorflow_probability.substrates.jax.distributions.joint_distribution.JointDistribution      with
    TFPDistributionMixin.
```

## JointDistributionCoroutine

```
class JointDistributionCoroutine (model, sample_dtype=None, validate_args=False, name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.joint_distribution_coroutine.JointDistributionCoroutine
    with TFPDistributionMixin.
```

## JointDistributionCoroutineAutoBatched

```
class JointDistributionCoroutineAutoBatched (model, sample_dtype=None, batch_ndims=0, use_vectorized_map=True, validate_args=False, experimental_use_kahan_sum=False, name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.joint_distribution_auto_batched.JointDistributionCoroutineAutoBatched
    with TFPDistributionMixin.
```

### JointDistributionNamed

```
class JointDistributionNamed(model, validate_args=False, name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.joint_distribution_named.JointDistributionNamed
    with TFPDistributionMixin.
```

### JointDistributionNamedAutoBatched

```
class JointDistributionNamedAutoBatched(model,                                batch_ndims=0,
                                         use_vectorized_map=True,  validate_args=False,
                                         experimental_use_kahan_sum=False,
                                         name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.joint_distribution_auto_batched.JointDistributionNamedAutoBatched
    with TFPDistributionMixin.
```

### JointDistributionSequential

```
class JointDistributionSequential(model, validate_args=False, name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.joint_distribution_sequential.JointDistributionSequential
    with TFPDistributionMixin.
```

### JointDistributionSequentialAutoBatched

```
class JointDistributionSequentialAutoBatched(model,                                batch_ndims=0,
                                              use_vectorized_map=True,          val-
                                              idate_args=False,                  experimen-
                                              tal_use_kahan_sum=False, name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.joint_distribution_auto_batched.JointDistributionSequentialAutoBatched
    with TFPDistributionMixin.
```

### Kumaraswamy

```
class Kumaraswamy(concentration1=1.0,    concentration0=1.0,    validate_args=False,    al-
                  low_nan_stats=True, name='Kumaraswamy')
    Wraps tensorflow_probability.substrates.jax.distributions.kumaraswamy.Kumaraswamy with
    TFPDistributionMixin.
```

### LKJ

```
class LKJ(dimension,    concentration,    input_output_cholesky=False,    validate_args=False,    al-
          low_nan_stats=True, name='LKJ')
    Wraps tensorflow_probability.substrates.jax.distributions.lkj.LKJ with TFPDistributionMixin.
```

### Laplace

```
class Laplace(loc, scale, validate_args=False, allow_nan_stats=True, name='Laplace')
    Wraps tensorflow_probability.substrates.jax.distributions.laplace.Laplace with TFPDistributionMixin.
```



## LinearGaussianStateSpaceModel

```
class LinearGaussianStateSpaceModel (num_timesteps, transition_matrix, transition_noise, ob-
                                     servation_matrix, observation_noise, initial_state_prior,
                                     initial_step=0, experimental_parallelize=False,
                                     validate_args=False, allow_nan_stats=True,
                                     name='LinearGaussianStateSpaceModel')
    Wraps tensorflow_probability.substrates.jax.distributions.linear_gaussian_ssm.LinearGaussianStateSpaceModel
    with TFPDistributionMixin.
```

## LogLogistic

```
class LogLogistic (loc, scale, validate_args=False, allow_nan_stats=True, name='LogLogistic')
    Wraps tensorflow_probability.substrates.jax.distributions.loglogistic.LogLogistic with
    TFPDistributionMixin.
```

## LogNormal

```
class LogNormal (loc, scale, validate_args=False, allow_nan_stats=True, name='LogNormal')
    Wraps tensorflow_probability.substrates.jax.distributions.lognormal.LogNormal with
    TFPDistributionMixin.
```

## Logistic

```
class Logistic (loc, scale, validate_args=False, allow_nan_stats=True, name='Logistic')
    Wraps tensorflow_probability.substrates.jax.distributions.logistic.Logistic with TFPDistributionMixin.
```

## LogitNormal

```
class LogitNormal (loc, scale, validate_args=False, allow_nan_stats=True, name='LogitNormal')
    Wraps tensorflow_probability.substrates.jax.distributions.logitnormal.LogitNormal with
    TFPDistributionMixin.
```

## MixtureSameFamily

```
class MixtureSameFamily (mixture_distribution, components_distribution, reparameterize=False, val-
                           idate_args=False, allow_nan_stats=True, name='MixtureSameFamily')
    Wraps tensorflow_probability.substrates.jax.distributions.mixture_same_family.MixtureSameFamily with
    TFPDistributionMixin.
```

## Moyal

```
class Moyal (loc, scale, validate_args=False, allow_nan_stats=True, name='Moyal')
    Wraps tensorflow_probability.substrates.jax.distributions.moyal.Moyal with TFPDistributionMixin.
```

## Multinomial

```
class Multinomial (total_count, logits=None, probs=None, validate_args=False, allow_nan_stats=True, name='Multinomial')
    Wraps tensorflow_probability.substrates.jax.distributions.multinomial.Multinomial with
    TFPDistributionMixin.
```

## MultivariateNormalDiag

```
class MultivariateNormalDiag (loc=None, scale_diag=None, scale_identity_multiplier=None,
                                validate_args=False, allow_nan_stats=True, experimental_use_kahan_sum=False, name='MultivariateNormalDiag')
    Wraps tensorflow_probability.substrates.jax.distributions.mvn_diag.MultivariateNormalDiag with
    TFPDistributionMixin.
```

## MultivariateNormalDiagPlusLowRank

```
class MultivariateNormalDiagPlusLowRank (loc=None, scale_diag=None,
                                            scale_perturb_factor=None,
                                            scale_perturb_diag=None, validate_args=False,
                                            allow_nan_stats=True,
                                            name='MultivariateNormalDiagPlusLowRank')
    Wraps tensorflow_probability.substrates.jax.distributions.mvn_diag_plus_low_rank.MultivariateNormalDiagPlusLowRank
    with TFPDistributionMixin.
```

## MultivariateNormalFullCovariance

```
class MultivariateNormalFullCovariance (loc=None, covariance_matrix=None, validate_args=False,
                                           allow_nan_stats=True,
                                           name='MultivariateNormalFullCovariance')
    Wraps tensorflow_probability.substrates.jax.distributions.mvn_full_covariance.MultivariateNormalFullCovariance
    with TFPDistributionMixin.
```

## MultivariateNormalLinearOperator

```
class MultivariateNormalLinearOperator (loc=None, scale=None, validate_args=False,
                                           allow_nan_stats=True,
                                           experimental_use_kahan_sum=False,
                                           name='MultivariateNormalLinearOperator')
    Wraps tensorflow_probability.substrates.jax.distributions.mvn_linear_operator.MultivariateNormalLinearOperator
    with TFPDistributionMixin.
```

## MultivariateNormalTriL

```
class MultivariateNormalTriL (loc=None, scale_tril=None, validate_args=False,
                                allow_nan_stats=True, experimental_use_kahan_sum=False,
                                name='MultivariateNormalTriL')
    Wraps tensorflow_probability.substrates.jax.distributions.mvn_tril.MultivariateNormalTriL with
    TFPDistributionMixin.
```

## MultivariateStudentTLinearOperator

```
class MultivariateStudentTLinearOperator (df, loc, scale, validate_args=False, allow_nan_stats=True, name='MultivariateStudentTLinearOperator')
```

Wraps `tensorflow_probability.substrates.jax.distributions.multivariate_student_t.MultivariateStudentTLinearOperator` with `TFPDistributionMixin`.

## NegativeBinomial

```
class NegativeBinomial (total_count, logits=None, probs=None, validate_args=False, allow_nan_stats=True, name='NegativeBinomial')
```

Wraps `tensorflow_probability.substrates.jax.distributions.negative_binomial.NegativeBinomial` with `TFPDistributionMixin`.

## Normal

```
class Normal (loc, scale, validate_args=False, allow_nan_stats=True, name='Normal')
```

Wraps `tensorflow_probability.substrates.jax.distributions.normal.Normal` with `TFPDistributionMixin`.

## OneHotCategorical

```
class OneHotCategorical (logits=None, probs=None, dtype=<class 'jax._src.numpy.lax_numpy.int32'>, validate_args=False, allow_nan_stats=True, name='OneHotCategorical')
```

Wraps `tensorflow_probability.substrates.jax.distributions.onehot_categorical.OneHotCategorical` with `TFPDistributionMixin`.

## OrderedLogistic

```
class OrderedLogistic (cutpoints, loc, dtype=<class 'jax._src.numpy.lax_numpy.int32'>, validate_args=False, allow_nan_stats=True, name='OrderedLogistic')
```

Wraps `tensorflow_probability.substrates.jax.distributions.ordered_logistic.OrderedLogistic` with `TFPDistributionMixin`.

## PERT

```
class PERT (low, peak, high, temperature=4.0, validate_args=False, allow_nan_stats=False, name='PERT')
```

Wraps `tensorflow_probability.substrates.jax.distributions.pert.PERT` with `TFPDistributionMixin`.

## Pareto

```
class Pareto (concentration, scale=1.0, validate_args=False, allow_nan_stats=True, name='Pareto')
```

Wraps `tensorflow_probability.substrates.jax.distributions.pareto.Pareto` with `TFPDistributionMixin`.

### PlackettLuce

```
class PlackettLuce(scores, dtype=<class 'jax._src.numpy.lax_numpy.int32'>, validate_args=False,
                  allow_nan_stats=True, name='PlackettLuce')
    Wraps      tensorflow_probability.substrates.jax.distributions.plackett_luce.PlackettLuce      with
    TFPDistributionMixin.
```

### Poisson

```
class Poisson(rate=None, log_rate=None, force_probs_to_zero_outside_support=None, interpolate_nondiscrete=True,
              validate_args=False, allow_nan_stats=True, name='Poisson')
    Wraps tensorflow_probability.substrates.jax.distributions.poisson.Poisson with TFPDistributionMixin.
```

### PoissonLogNormalQuadratureCompound

```
class PoissonLogNormalQuadratureCompound(loc,          scale,          quadrature_size=8,
                                           quadrature_fn=<function          quadrature_scheme_lognormal_quantiles>,
                                           validate_args=False,          allow_nan_stats=True,
                                           name='PoissonLogNormalQuadratureCompound')
    Wraps tensorflow_probability.substrates.jax.distributions.poisson_lognormal.PoissonLogNormalQuadratureCompound
    with TFPDistributionMixin.
```

### PowerSpherical

```
class PowerSpherical(mean_direction, concentration, validate_args=False, allow_nan_stats=True,
                     name='PowerSpherical')
    Wraps      tensorflow_probability.substrates.jax.distributions.power_spherical.PowerSpherical      with
    TFPDistributionMixin.
```

### ProbitBernoulli

```
class ProbitBernoulli(probits=None,          probs=None,          dtype=<class
                      'jax._src.numpy.lax_numpy.int32'>,          validate_args=False,          al-
                      low_nan_stats=True, name='ProbitBernoulli')
    Wraps      tensorflow_probability.substrates.jax.distributions.probit_bernoulli.ProbitBernoulli      with
    TFPDistributionMixin.
```

### QuantizedDistribution

```
class QuantizedDistribution(distribution, low=None, high=None, validate_args=False,
                           name='QuantizedDistribution')
    Wraps tensorflow_probability.substrates.jax.distributions.quantized_distribution.QuantizedDistribution with
    TFPDistributionMixin.
```

### RelaxedBernoulli

```
class RelaxedBernoulli(temperature, logits=None, probs=None, validate_args=False, al-
                      low_nan_stats=True, name='RelaxedBernoulli')
    Wraps      tensorflow_probability.substrates.jax.distributions.relaxed_bernoulli.RelaxedBernoulli      with
    TFPDistributionMixin.
```

## RelaxedOneHotCategorical

```
class RelaxedOneHotCategorical (temperature, logits=None, probs=None, validate_args=False,
                                allow_nan_stats=True, name='RelaxedOneHotCategorical')
    Wraps tensorflow_probability.substrates.jax.distributions.relaxed_onehot_categorical.RelaxedOneHotCategorical
    with TFPDistributionMixin.
```

## Sample

```
class Sample (distribution, sample_shape=(), validate_args=False, experimental_use_kahan_sum=False,
               name=None)
    Wraps tensorflow_probability.substrates.jax.distributions.sample.Sample with TFPDistributionMixin.
```

## SinhArcsinh

```
class SinhArcsinh (loc, scale, skewness=None, tailweight=None, distribution=None, validate_args=False,
                    allow_nan_stats=True, name='SinhArcsinh')
    Wraps tensorflow_probability.substrates.jax.distributions.sinh_arcsinh.SinhArcsinh with
    TFPDistributionMixin.
```

## Skellam

```
class Skellam (rate1=None, rate2=None, log_rate1=None, log_rate2=None,
                force_probs_to_zero_outside_support=False, validate_args=False, allow_nan_stats=True,
                name='Skellam')
    Wraps tensorflow_probability.substrates.jax.distributions.skellam.Skellam with TFPDistributionMixin.
```

## SphericalUniform

```
class SphericalUniform (dimension, batch_shape=(), dtype=<class 'jax._src.numpy.lax_numpy.float32'>,
                         validate_args=False, allow_nan_stats=True, name='SphericalUniform')
    Wraps tensorflow_probability.substrates.jax.distributions.spherical_uniform.SphericalUniform with
    TFPDistributionMixin.
```

## StoppingRatioLogistic

```
class StoppingRatioLogistic (cutpoints, loc, dtype=<class 'jax._src.numpy.lax_numpy.int32'>,
                              validate_args=False, allow_nan_stats=True, name='StoppingRatioLogistic')
    Wraps tensorflow_probability.substrates.jax.distributions.stopping_ratio_logistic.StoppingRatioLogistic with
    TFPDistributionMixin.
```

## StudentT

```
class StudentT (df, loc, scale, validate_args=False, allow_nan_stats=True, name='StudentT')
    Wraps tensorflow_probability.substrates.jax.distributions.student_t.StudentT with
    TFPDistributionMixin.
```

## StudentTProcess

```
class StudentTProcess(df, kernel, index_points=None, mean_fn=None, jitter=1e-06, validate_args=False, allow_nan_stats=False, name='StudentTProcess')
    Wraps      tensorflow_probability.substrates.jax.distributions.student_t_process.StudentTProcess      with
    TFPDistributionMixin.
```

## TransformedDistribution

```
class TransformedDistribution(distribution, bijector, kwargs_split_fn=<function _default_kwargs_split_fn>, validate_args=False, parameters=None, name=None)
    Wraps      tensorflow_probability.substrates.jax.distributions.transformed_distribution.TransformedDistribution
    with TFPDistributionMixin.
```

## Triangular

```
class Triangular(low=0.0, high=1.0, peak=0.5, validate_args=False, allow_nan_stats=True, name='Triangular')
    Wraps      tensorflow_probability.substrates.jax.distributions.triangular.Triangular      with
    TFPDistributionMixin.
```

## TruncatedCauchy

```
class TruncatedCauchy(loc, scale, low, high, validate_args=False, allow_nan_stats=True, name='TruncatedCauchy')
    Wraps      tensorflow_probability.substrates.jax.distributions.truncated_cauchy.TruncatedCauchy      with
    TFPDistributionMixin.
```

## TruncatedNormal

```
class TruncatedNormal(loc, scale, low, high, validate_args=False, allow_nan_stats=True, name='TruncatedNormal')
    Wraps      tensorflow_probability.substrates.jax.distributions.truncated_normal.TruncatedNormal      with
    TFPDistributionMixin.
```

## Uniform

```
class Uniform(low=0.0, high=1.0, validate_args=False, allow_nan_stats=True, name='Uniform')
    Wraps tensorflow_probability.substrates.jax.distributions.uniform.Uniform with TFPDistributionMixin.
```

## VariationalGaussianProcess

```
class VariationalGaussianProcess(kernel, index_points, inducing_index_points,
    variational_inducing_observations_loc, variational_inducing_observations_scale, mean_fn=None,
    observation_noise_variance=None, predictive_noise_variance=None, jitter=1e-06, validate_args=False,
    allow_nan_stats=False, name='VariationalGaussianProcess')
    Wraps tensorflow_probability.substrates.jax.distributions.variational_gaussian_process.VariationalGaussianProcess
```

with *TFPDistributionMixin*.

### VectorDeterministic

```
class VectorDeterministic(loc, atol=None, rtol=None, validate_args=False, al-
                           low_nan_stats=True, name='VectorDeterministic')
Wraps      tensorflow_probability.substrates.jax.distributions.deterministic.VectorDeterministic with
TFPDistributionMixin.
```

### VectorExponentialDiag

```
class VectorExponentialDiag(loc=None, scale_diag=None, scale_identity_multiplier=None,
                             validate_args=False, allow_nan_stats=True,
                             name='VectorExponentialDiag')
Wraps tensorflow_probability.substrates.jax.distributions.vector_exponential_diag.VectorExponentialDiag with
TFPDistributionMixin.
```

### VonMises

```
class VonMises(loc, concentration, validate_args=False, allow_nan_stats=True, name='VonMises')
Wraps      tensorflow_probability.substrates.jax.distributions.von_mises.VonMises with
TFPDistributionMixin.
```

### VonMisesFisher

```
class VonMisesFisher(mean_direction, concentration, validate_args=False, allow_nan_stats=True,
                      name='VonMisesFisher')
Wraps      tensorflow_probability.substrates.jax.distributions.von_mises_fisher.VonMisesFisher with
TFPDistributionMixin.
```

### Weibull

```
class Weibull(concentration, scale, validate_args=False, allow_nan_stats=True, name='Weibull')
Wraps tensorflow_probability.substrates.jax.distributions.weibull.Weibull with TFPDistributionMixin.
```

### WishartLinearOperator

```
class WishartLinearOperator(df, scale, input_output_cholesky=False, validate_args=False, al-
                             low_nan_stats=True, name='WishartLinearOperator')
Wraps      tensorflow_probability.substrates.jax.distributions.wishart.WishartLinearOperator with
TFPDistributionMixin.
```

### WishartTriL

```
class WishartTriL(df, scale_tril=None, input_output_cholesky=False, validate_args=False, al-
                  low_nan_stats=True, name='WishartTriL')
Wraps      tensorflow_probability.substrates.jax.distributions.wishart.WishartTriL with
TFPDistributionMixin.
```

## 2.2.6 Constraints

### 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.

**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*.

### boolean

`boolean` = `<numpyro.distributions.constraints._Boolean object>`

### corr\_cholesky

`corr_cholesky` = `<numpyro.distributions.constraints._CorrCholesky object>`

### corr\_matrix

`corr_matrix` = `<numpyro.distributions.constraints._CorrMatrix object>`

### dependent

`dependent` = `<numpyro.distributions.constraints._Dependent object>`

### 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.

### 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.



## 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.

## 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.

## 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.

## lower\_cholesky

`lower_cholesky = <numpyro.distributions.constraints._LowerCholesky object>`

## 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.

## nonnegative\_integer

`nonnegative_integer = <numpyro.distributions.constraints._IntegerGreaterThan object>`

## ordered\_vector

`ordered_vector = <numpyro.distributions.constraints._OrderedVector object>`

## positive

`positive = <numpyro.distributions.constraints._GreaterThan object>`

### positive\_definite

```
positive_definite = <numpyro.distributions.constraints._PositiveDefinite object>
```

### positive\_integer

```
positive_integer = <numpyro.distributions.constraints._IntegerGreaterThan object>
```

### real

```
real = <numpyro.distributions.constraints._Real object>
```

### real\_vector

```
real_vector = <numpyro.distributions.constraints._IndependentConstraint object>
    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.
```

### simplex

```
simplex = <numpyro.distributions.constraints._Simplex object>
```

### unit\_interval

```
unit_interval = <numpyro.distributions.constraints._Interval object>
```

## 2.2.7 Transforms

### biject\_to

```
biject_to(constraint)
```

### Transform

```
class Transform
```

```
    Bases: object
```

```
    domain = <numpyro.distributions.constraints._Real object>
```

```
    codomain = <numpyro.distributions.constraints._Real object>
```

```
    event_dim
```

```
    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.

## AbsTransform

**class AbsTransform**

Bases: `numpyro.distributions.transforms.Transform`

**domain** = `<numpyro.distributions.constraints._Real object>`

**codomain** = `<numpyro.distributions.constraints._GreaterThan object>`

## AffineTransform

**class AffineTransform** (*loc, scale, domain=<numpyro.distributions.constraints.\_Real object>*)

Bases: `numpyro.distributions.transforms.Transform`

---

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

---

**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.

## ComposeTransform

**class ComposeTransform** (*parts*)

Bases: `numpyro.distributions.transforms.Transform`

**domain**

**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.

## CorrCholeskyTransform

**class CorrCholeskyTransform**

Bases: `numpyro.distributions.transforms.Transform`

Transforms a unconstrained 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  $s_i = \text{StickBreakingTransform}(z_i)$ .
- d. Transforms back into signed domain:  $y_i = (\text{sign}(r_i), 1) * \sqrt{s_i}$ .

**domain** = <numpyro.distributions.constraints.\_IndependentConstraint object>

**codomain** = <numpyro.distributions.constraints.\_CorrCholesky object>

**log\_abs\_det\_jacobian** ( $x, y, \text{intermediates}=\text{None}$ )

**forward\_shape** ( $\text{shape}$ )

Infers the shape of the forward computation, given the input shape. Defaults to preserving shape.

**inverse\_shape** ( $\text{shape}$ )

Infers the shapes of the inverse computation, given the output shape. Defaults to preserving shape.

## ExpTransform

**class** `ExpTransform` ( $\text{domain}=\text{<numpyro.distributions.constraints._Real object>}$ )

Bases: `numpyro.distributions.transforms.Transform`

**codomain**

**log\_abs\_det\_jacobian** ( $x, y, \text{intermediates}=\text{None}$ )

## IdentityTransform

**class** `IdentityTransform`

Bases: `numpyro.distributions.transforms.Transform`

**log\_abs\_det\_jacobian** ( $x, y, \text{intermediates}=\text{None}$ )

## InvCholeskyTransform

**class** `InvCholeskyTransform` ( $\text{domain}=\text{<numpyro.distributions.constraints._LowerCholesky object>}$ ) *ob-*

Bases: `numpyro.distributions.transforms.Transform`

Transform via the mapping  $y = x @ x.T$ , where  $x$  is a lower triangular matrix with positive diagonal.

**codomain**

**log\_abs\_det\_jacobian** ( $x, y, \text{intermediates}=\text{None}$ )

## LowerCholeskyAffine

**class LowerCholeskyAffine** (*loc, scale\_tril*)

Bases: `numpyro.distributions.transforms.Transform`

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

### Parameters

- **loc** – a real vector.
- **scale\_tril** – a lower triangular matrix with positive diagonal.

**domain** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**codomain** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**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.

## LowerCholeskyTransform

**class LowerCholeskyTransform**

Bases: `numpyro.distributions.transforms.Transform`

**domain** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**codomain** = `<numpyro.distributions.constraints._LowerCholesky object>`

**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.

## 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

**domain** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**codomain** = `<numpyro.distributions.constraints._OrderedVector object>`

**log\_abs\_det\_jacobian** (*x, y, intermediates=None*)

## PermuteTransform

```
class PermuteTransform(permutation)
    Bases: numpyro.distributions.transforms.Transform

    domain = <numpyro.distributions.constraints._IndependentConstraint object>
    codomain = <numpyro.distributions.constraints._IndependentConstraint object>
    log_abs_det_jacobian(x, y, intermediates=None)
```

## PowerTransform

```
class PowerTransform(exponent)
    Bases: numpyro.distributions.transforms.Transform

    domain = <numpyro.distributions.constraints._GreaterThan object>
    codomain = <numpyro.distributions.constraints._GreaterThan object>
    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.
```

## SigmoidTransform

```
class SigmoidTransform
    Bases: numpyro.distributions.transforms.Transform

    codomain = <numpyro.distributions.constraints._Interval object>
    log_abs_det_jacobian(x, y, intermediates=None)
```

## StickBreakingTransform

```
class StickBreakingTransform
    Bases: numpyro.distributions.transforms.Transform

    domain = <numpyro.distributions.constraints._IndependentConstraint object>
    codomain = <numpyro.distributions.constraints._Simplex object>
    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.
```

## 2.2.8 Flows

### 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,  $\mu_t, \sigma_t$  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** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**codomain** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**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

### 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** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**codomain** = `<numpyro.distributions.constraints._IndependentConstraint object>`

**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

## 2.3 Inference

### 2.3.1 Markov Chain Monte Carlo (MCMC)

**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.

---

#### 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 Number of MCMC chains to run. By default, chains will be run in parallel using `jax.pmap()`, failing which, 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.

#### `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), 100, 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()
```

**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*) – Extra fields (aside from *z*, *diverging*) from `numpyro.infer.mcmc.HMCState` to collect during the MCMC run.
- **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](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).

**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.

## MCMC Kernels

**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
... 
```

(continues on next page)

(continued from previous page)

```

...     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.]))
>>> 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.

**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*.

**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()`.

**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.

```
class HMC(model=None, potential_fn=None, kinetic_fn=None, step_size=1.0, adapt_step_size=True,
          adapt_mass_matrix=True, dense_mass=False, target_accept_prob=0.8, trajectory_length=6.283185307179586,
          init_strategy=<function init_to_uniform>, find_heuristic_step_size=False, forward_mode_differentiation=False)
```

Bases: `numpyro.infer.mcmc.MCMCKernel`

Hamiltonian Monte Carlo inference, using fixed trajectory length, with provision for step size and mass matrix adaptation.

**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.
- **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*) – A flag to decide if mass matrix is dense or diagonal (default when `dense_mass=False`)
- **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. Default to 0.8.
- **trajectory\_length** (*float*) – Length of a MCMC trajectory for HMC. Default value is  $2\pi$ .
- **init\_strategy** (*callable*) – a per-site initialization function. See *Initialization Strategies* section for available functions.

- **find\_heuristic\_step\_size** (*bool*) – whether 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.

**model**

**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()`.

**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.

```
class NUTS(model=None, potential_fn=None, kinetic_fn=None, step_size=1.0, 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)
```

Bases: `numpyro.infer.hmc.HMC`

Hamiltonian Monte Carlo inference, using the No U-Turn Sampler (NUTS) with adaptive path length and mass matrix adaptation.

#### 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.
- **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*) – A flag to decide if mass matrix is dense or diagonal (default when *dense\_mass=False*)
- **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. Default 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.
- **init\_strategy** (*callable*) – a per-site initialization function. See [Initialization Strategies](#) section for available functions.
- **find\_heuristic\_step\_size** (*bool*) – whether to a heuristic function to adjust the step size at the beginning of each adaptation window. Defaults to False.
- **forward\_mode\_differentiation** (*bool*) – whether to use forward-mode differentiation or reverse-mode differentiation. By default, we use reverse mode but the forward mode can be useful in some cases to improve the performance. In addition, some control

flow utility on JAX such as `jax.lax.while_loop` or `jax.lax.fori_loop` only supports forward-mode differentiation. See [JAX's The Autodiff Cookbook](#) for more information.

**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}
...
>>> hmc_kernel = NUTS(model)
>>> kernel = HMCGibbs(hmc_kernel, gibbs_fn=gibbs_fn, gibbs_sites=['x'])
>>> mcmc = MCMC(kernel, 100, 100, progress_bar=False)
>>> mcmc.run(random.PRNGKey(0))
>>> mcmc.print_summary()
```

**sample\_field** = 'z'

**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*.

**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*.
- **discrete\_sites** (*list*) – a list of site names for the discrete latent variables that are covered by the Gibbs sampler.



- **random\_walk** (*bool*) – If False, Gibbs sampling will be used to draw a sample from the conditional  $p(\text{gibbs\_site} \mid \text{remaining\_sites})$ . Otherwise, a sample will be drawn uniformly from the domain of *gibbs\_site*.
- **modified** (*bool*) – whether to use a modified proposal, as suggested in reference [1], which always proposes a new state for the current Gibbs site. The modified scheme appears in the literature under the name “modified Gibbs sampler” or “Metropolis 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, 1000, 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*.

**class** **HMCECS** (*inner\_kernel*, \*, *num\_blocks*=1)  
 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-Marginal Sampler*, Tran, M.-N., Kohn, R., Quiroz, M. Villani, M. (2017)

#### Parameters

- **inner\_kernel** – One of `HMC` or `NUTS`.
- **num\_blocks** (*int*) – Number of blocks to partition subsample into.

#### 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
>>> kernel = HMCECS(NUTS(model), num_blocks=10)
>>> mcmc = MCMC(kernel, 1000, 1000)
>>> mcmc.run(random.PRNGKey(0), data)
>>> samples = mcmc.get_samples()["x"]
>>> assert abs(jnp.mean(samples) - 1.) < 0.1
```

**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*.

```
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*.

#### 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()*.

#### 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 *SASState* and return the resulting *SASState*.

#### Parameters

- **state** (*SASState*) – 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 `numpyro.infer.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')
```

(continues on next page)

(continued from previous page)

```

>>> 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*, *trajectory\_length=6.283185307179586*, *max\_tree\_depth=10*, *find\_heuristic\_step\_size=False*, *forward\_mode\_differentiation=False*, *model\_args=()*, *model\_kwargs=None*, *rng\_key=DeviceArray([0, 0], dtype=uint32)*)

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*) – 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.
- **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*) – A flag to decide if mass matrix is dense or diagonal (default when *dense\_mass=False*)
- **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. Default to 0.8.
- **trajectory\_length** (*float*) – Length of a MCMC trajectory for HMC. Default value is  $2\pi$ .
- **max\_tree\_depth** (*int*) – Max depth of the binary tree created during the doubling scheme of NUTS sampler. Defaults to 10.
- **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.
- **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.

**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.
- **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 `hmc_state`
- **rng\_key** - random key for the current step

**SASState** = <class 'numpyro.infer.sa.SASState'>

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.

## TensorFlow Kernels

Thin wrappers around TensorFlow Probability (TFP) distributions. For details on the TFP distribution 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`.

## 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)
(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.

**Returns** a namedtuple *ModelInfo* which contains the fields (*param\_info*, *potential\_fn*, *postprocess\_fn*, *model\_trace*), where *param\_info* is a namedtuple *ParamInfo* containing values from the prior used to initiate MCMC, their corresponding potential energy, and their gradients; *postprocess\_fn* is a callable that uses inverse transforms to convert unconstrained HMC samples to constrained values that lie within the site’s support, in addition to returning values at *deterministic* sites in the model.

**fori\_collect** (*lower*, *upper*, *body\_fun*, *init\_val*, *transform*=<function identity>, *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\_fun*.
- **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  $(\text{upper} - \text{lower}) // \text{thinning}$ . If the size is larger than  $(\text{upper} - \text{lower}) // \text{thinning}$ , only the top  $(\text{upper} - \text{lower}) // \text{thinning}$  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.

## 2.3.2 Stochastic Variational Inference (SVI)

**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](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 SVI, Trace_ELBO

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

```

### 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.
- **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*).

**run** (*rng\_key*, *num\_steps*, \**args*, *progress\_bar*=*True*, \*\**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*.
- **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*).

## ELBO

**class** **ELBO** (*num\_particles*=1)

Bases: `numpyro.infer.elbo.Trace_ELBO`

## Trace\_ELBO

**class** `Trace_ELBO` (*num\_particles=1*)

Bases: `object`

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](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** (*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.

## TraceMeanField\_ELBO

**class** `TraceMeanField_ELBO` (*num\_particles=1*)

Bases: `numpyro.infer.elbo.Trace_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** (*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.

## RenyiELBO

**class** **RenyiELBO** (*alpha=0*, *num\_particles=2*)

Bases: `numpyro.infer.elbo.Trace_ELBO`

An implementation of Renyi's  $\alpha$ -divergence variational inference following reference [1]. In order for the objective to be a strict lower bound, we require  $\alpha \geq 0$ . Note, however, that according to reference [1], depending on the dataset  $\alpha < 0$  might give better results. In the special case  $\alpha = 0$ , the objective function is that of the important weighted autoencoder derived in reference [2].

---

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

---

#### Parameters

- **alpha** (*float*) – The order of  $\alpha$ -divergence. Here  $\alpha \neq 1$ . Default is 0.
- **num\_particles** – The number of particles/samples used to form the objective (gradient) estimator. Default is 2.

#### 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 Renyi 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 Renyi Evidence Lower Bound (ELBO) to be minimized.

## 2.3.3 Automatic Guide Generation

### 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=())`

Get samples from the learned posterior.

#### 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*) – batch shape of each latent sample, 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 tensor.

**Return type** *dict*

**quantiles** (*params, quantiles*)

Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(opt_state, [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 a list of quantile values.

**Return type** *dict*

## AutoBNAFNormal

```
class AutoBNAFNormal(model, *, prefix='auto', init_loc_fn=<function init_to_uniform>,
                      num_flows=1, hidden_factors=[8, 8], init_strategy=None)
```

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 3.

- **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*.

## AutoDiagonalNormal

**class AutoDiagonalNormal** (*model*, \*, *prefix*='auto', *init\_loc\_fn*=<function init\_to\_uniform>, *init\_scale*=0.1, *init\_strategy*=None)

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, ...)
```

**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 tensor.

**Return type** *dict*

**quantiles** (*params*, *quantiles*)

Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(opt_state, [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 a list of quantile values.

**Return type** `dict`

## AutoMultivariateNormal

**class** `AutoMultivariateNormal` (*model*, \*, *prefix*='auto', *init\_loc\_fn*=<function *init\_to\_uniform*>, *init\_scale*=0.1, *init\_strategy*=None)

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, ...)
```

**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 tensor.

**Return type** `dict`

**quantiles**(*params*, *quantiles*)

Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(opt_state, [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 a list of quantile values.

**Return type** `dict`

## 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>>),
                    init_strategy=None)
```

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.experimental.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`.

## AutoLaplaceApproximation

```
class AutoLaplaceApproximation (model, *, prefix='auto', init_loc_fn=<function
                                init_to_uniform>, create_plates=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, ...)
```

**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*=())

Get samples from the learned posterior.

**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*) – batch shape of each latent sample, 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 tensor.

**Return type** *dict*

**quantiles** (*params*, *quantiles*)

Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(opt_state, [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 a list of quantile values.

**Return type** *dict*

## AutoLowRankMultivariateNormal

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

**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 tensor.

**Return type** *dict*

**quantiles(params, quantiles)**

Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles(opt_state, [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 a list of quantile values.

**Return type** *dict*

## 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 :class: *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 inputting 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*) – PRNG seed.
- **params** – Current parameters of model and autoguide.
- **sample\_shape** – (keyword argument) shape of samples to be drawn.

**Returns** batch of samples from the approximate posterior.

**median** (*params*)

**quantiles** (*params*, *quantiles*)

## 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 inputting 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*) – PRNG seed.
- **params** – Current parameters of model and autoguide.
- **sample\_shape** – (keyword argument) shape of samples to be drawn.

**Returns** batch of samples from the approximate posterior.

**median** (*params*)

Returns the posterior median value of each latent variable.

**Returns** A dict mapping sample site name to median tensor.

**Return type** `dict`

## 2.3.4 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. `AutoNormal` guides and MCMC.

**class Reparam**

Bases: `abc.ABC`

Base class for reparameterizers.

### 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).

## 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 = neutra.reparam(model)
nuts = NUTS(model)
# ...now use the model in HMC or NUTS...
```

This reparameterization works only for latent variables, not likelihoods. Note that all sites must share a single common `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*

## 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`).

## 2.3.5 Funsor-based NumPyro

### 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 Pyro 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.functor.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 leftmost 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 `Functor`.

#### Parameters

- **x** (*Functor*) – A functor 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-functor equivalent to *x*.

**to\_functor** (*x, output=None, dim\_to\_name=None, dim\_type=<DimType.LOCAL: 0>*)

A primitive to convert a Python object to a `Functor`.

**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*)

**Inference Utilities**

**config\_enumerate** (*fn, 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”.

**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)
```

## 2.3.6 Optimizers

Optimizer classes defined here are light wrappers over the corresponding optimizers sourced from `jax.experimental.optimizers` with an interface that is better suited for working with NumPyro inference algorithms.

**Adam**

**class Adam** (\*args, \*\*kwargs)

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

**eval\_and\_update** (*fn: Callable, state: Tuple[int, \_OptState]*) → *Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (*state: Tuple[int, \_OptState]*) → *\_Params*

Get current parameter values.

**Parameters** **state** – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params*: *\_Params*) → *Tuple*[*int*, *\_OptState*]

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g*: *\_Params*, *state*: *Tuple*[*int*, *\_OptState*]) → *Tuple*[*int*, *\_OptState*]

Gradient update for the optimizer.

**Parameters**

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

**Returns** new optimizer state after the update.

## Adagrad

**class** *Adagrad* (\**args*, \*\**kwargs*)

Wrapper class for the JAX optimizer: *adagrad*()

**eval\_and\_update** (*fn*: *Callable*, *state*: *Tuple*[*int*, *\_OptState*]) → *Tuple*[*int*, *\_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* – objective function.
- *state* – current optimizer state.

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

**get\_params** (*state*: *Tuple*[*int*, *\_OptState*]) → *\_Params*

Get current parameter values.

**Parameters** *state* – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params*: *\_Params*) → *Tuple*[*int*, *\_OptState*]

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g*: *\_Params*, *state*: *Tuple*[*int*, *\_OptState*]) → *Tuple*[*int*, *\_OptState*]

Gradient update for the optimizer.

**Parameters**

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

**Returns** new optimizer state after the update.

## 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\_update** (*fn: Callable, state: Tuple[int, \_OptState]*) → *Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (*state: Tuple[int, \_OptState]*) → *\_Params*

Get current parameter values.

**Parameters** **state** – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params: \_Params*) → *Tuple[int, \_OptState]*

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g, state*)

Gradient update for the optimizer.

### Parameters

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

**Returns** new optimizer state after the update.

## 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
```

(continues on next page)



(continued from previous page)

```

>>> 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\_update** (*fn*: Callable, *state*: Tuple[int, \_OptState]) → Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (*state*: Tuple[int, \_OptState]) → \_Params

Get current parameter values.

**Parameters** **state** – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params*: \_Params) → Tuple[int, \_OptState]

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g*: \_Params, *state*: Tuple[int, \_OptState]) → Tuple[int, \_OptState]

Gradient update for the optimizer.

#### Parameters

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

**Returns** new optimizer state after the update.

## Momentum

**class Momentum** (\*args, \*\*kwargs)

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

**eval\_and\_update** (fn: Callable, state: Tuple[int, \_OptState]) → Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (state: Tuple[int, \_OptState]) → \_Params

Get current parameter values.

**Parameters** **state** – current optimizer state.

**Returns** collection with current value for parameters.

**init** (params: \_Params) → Tuple[int, \_OptState]

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (g: \_Params, state: Tuple[int, \_OptState]) → Tuple[int, \_OptState]

Gradient update for the optimizer.

**Parameters**

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

**Returns** new optimizer state after the update.

## RMSProp

**class RMSProp** (\*args, \*\*kwargs)

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

**eval\_and\_update** (fn: Callable, state: Tuple[int, \_OptState]) → Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (*state: Tuple[int, \_OptState]*) → *\_Params*

Get current parameter values.

**Parameters** *state* – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params: \_Params*) → *Tuple[int, \_OptState]*

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g: \_Params, state: Tuple[int, \_OptState]*) → *Tuple[int, \_OptState]*

Gradient update for the optimizer.

**Parameters**

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

**Returns** new optimizer state after the update.

## RMSPropMomentum

**class RMSPropMomentum** (*\*args, \*\*kwargs*)

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

**eval\_and\_update** (*fn: Callable, state: Tuple[int, \_OptState]*) → *Tuple[int, \_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* – objective function.
- *state* – current optimizer state.

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

**get\_params** (*state: Tuple[int, \_OptState]*) → *\_Params*

Get current parameter values.

**Parameters** *state* – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params: \_Params*) → *Tuple[int, \_OptState]*

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g: \_Params, state: Tuple[int, \_OptState]*) → *Tuple[int, \_OptState]*

Gradient update for the optimizer.

**Parameters**

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

**Returns** new optimizer state after the update.

## SGD

**class** `SGD` (\*args, \*\*kwargs)

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

**eval\_and\_update** (*fn*: Callable, *state*: Tuple[int, \_OptState]) → Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (*state*: Tuple[int, \_OptState]) → \_Params

Get current parameter values.

**Parameters** **state** – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params*: \_Params) → Tuple[int, \_OptState]

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g*: \_Params, *state*: Tuple[int, \_OptState]) → Tuple[int, \_OptState]

Gradient update for the optimizer.

### Parameters

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

**Returns** new optimizer state after the update.

## SM3

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

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

**eval\_and\_update** (*fn*: Callable, *state*: Tuple[int, \_OptState]) → Tuple[int, \_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** – objective function.
- **state** – current optimizer state.

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

**get\_params** (*state*: *Tuple[int, \_OptState]*) → *\_Params*

Get current parameter values.

**Parameters** *state* – current optimizer state.

**Returns** collection with current value for parameters.

**init** (*params*: *\_Params*) → *Tuple[int, \_OptState]*

Initialize the optimizer with parameters designated to be optimized.

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

**Returns** initial optimizer state.

**update** (*g*: *\_Params*, *state*: *Tuple[int, \_OptState]*) → *Tuple[int, \_OptState]*

Gradient update for the optimizer.

**Parameters**

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

**Returns** new optimizer state after the update.

## 2.3.7 Diagnostics

This provides a small set of utilities in NumPyro that are used to diagnose posterior samples.

### 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*

### 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*

## 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`

## 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`

## 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`

## 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  $(1 - \text{prob}) / 2$  and  $(1 + \text{prob}) / 2$ .

**Return type** `numpy.ndarray`

## 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).

## 2.3.8 Runtime Utilities

### 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.

### 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.

### 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.

### 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’.

### 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.

## 2.3.9 Inference Utilities

### Predictive

**class Predictive** (*model, posterior\_samples=None, guide=None, params=None, num\_samples=None, return\_sites=None, parallel=False, batch\_ndims=1*)

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.
- **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. Some usages:
  - 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 ...)
  - 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*.

**Returns** dict of samples from the predictive distribution.

## 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

## 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.

## 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.

## potential\_energy

**potential\_energy** (*model*, *model\_args*, *model\_kwargs*, *params*, *enum=False*)

(EXPERIMENTAL INTERFACE) Computes potential energy of a model given unconstrained params. The *inv\_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*) – unconstrained parameters of *model*.
- **enum** (*bool*) – whether to enumerate over discrete latent sites.

**Returns** potential energy given unconstrained parameters.

## 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.

### find\_valid\_initial\_params

**find\_valid\_initial\_params** (`rng_key`, `model`, `init_strategy=<function init_to_uniform>`,  
`enum=False`, `model_args=()`, `model_kwargs=None`, `proto-`  
`type_params=None`, `forward_mode_differentiation=False`)

(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.

**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.

### 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.

## 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)
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.



---

## 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.

### 3.1 Tutorial Outline:

1. *Dataset*
2. *Regression Model to Predict Divorce Rate*
  - *Model-1: Predictor-Marriage Rate*
  - *Posterior Distribution over the Regression Parameters*
  - *Posterior Predictive Distribution*
  - *Predictive Utility With Effect Handlers*
  - *Model Predictive Density*
  - *Model-2: Predictor-Median Age of Marriage*
  - *Model-3: Predictor-Marriage Rate and Median Age of Marriage*
  - *Divorce Rate Residuals by State*

### 3. Regression Model with Measurement Error

- *Effect of Incorporating Measurement Noise on Residuals*

### 4. References

```
[ ]: %reset -s -f

[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('0.5.0')
```

## 3.2 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	Population	MedianAgeMarriage	Marriage	\
0	Alabama	AL	4.78	25.3	20.2	
1	Alaska	AK	0.71	25.2	26.0	
2	Arizona	AZ	6.33	25.8	20.3	
3	Arkansas	AR	2.92	24.3	26.4	
4	California	CA	37.25	26.8	19.1	
5	Colorado	CO	5.03	25.7	23.5	
6	Connecticut	CT	3.57	27.6	17.1	
7	Delaware	DE	0.90	26.6	23.1	
8	District of Columbia	DC	0.60	29.7	17.7	
9	Florida	FL	18.80	26.4	17.0	
10	Georgia	GA	9.69	25.9	22.1	
11	Hawaii	HI	1.36	26.9	24.9	
12	Idaho	ID	1.57	23.2	25.8	

(continues on next page)



(continued from previous page)

13	Illinois	IL	12.83	27.0	17.9
14	Indiana	IN	6.48	25.7	19.8
15	Iowa	IA	3.05	25.4	21.5
16	Kansas	KS	2.85	25.0	22.1
17	Kentucky	KY	4.34	24.8	22.2
18	Louisiana	LA	4.53	25.9	20.6
19	Maine	ME	1.33	26.4	13.5
20	Maryland	MD	5.77	27.3	18.3
21	Massachusetts	MA	6.55	28.5	15.8
22	Michigan	MI	9.88	26.4	16.5
23	Minnesota	MN	5.30	26.3	15.3
24	Mississippi	MS	2.97	25.8	19.3
25	Missouri	MO	5.99	25.6	18.6
26	Montana	MT	0.99	25.7	18.5
27	Nebraska	NE	1.83	25.4	19.6
28	New Hampshire	NH	1.32	26.8	16.7
29	New Jersey	NJ	8.79	27.7	14.8
30	New Mexico	NM	2.06	25.8	20.4
31	New York	NY	19.38	28.4	16.8
32	North Carolina	NC	9.54	25.7	20.4
33	North Dakota	ND	0.67	25.3	26.7
34	Ohio	OH	11.54	26.3	16.9
35	Oklahoma	OK	3.75	24.4	23.8
36	Oregon	OR	3.83	26.0	18.9
37	Pennsylvania	PA	12.70	27.1	15.5
38	Rhode Island	RI	1.05	28.2	15.0
39	South Carolina	SC	4.63	26.4	18.1
40	South Dakota	SD	0.81	25.6	20.1
41	Tennessee	TN	6.35	25.2	19.4
42	Texas	TX	25.15	25.2	21.5
43	Utah	UT	2.76	23.3	29.6
44	Vermont	VT	0.63	26.9	16.4
45	Virginia	VA	8.00	26.4	20.5
46	Washington	WA	6.72	25.9	21.4
47	West Virginia	WV	1.85	25.0	22.2
48	Wisconsin	WI	5.69	26.3	17.2
49	Wyoming	WY	0.56	24.2	30.7

	Marriage SE	Divorce	Divorce SE	WaffleHouses	South	Slaves1860	\
0	1.27	12.7	0.79	128	1	435080	
1	2.93	12.5	2.05	0	0	0	
2	0.98	10.8	0.74	18	0	0	
3	1.70	13.5	1.22	41	1	111115	
4	0.39	8.0	0.24	0	0	0	
5	1.24	11.6	0.94	11	0	0	
6	1.06	6.7	0.77	0	0	0	
7	2.89	8.9	1.39	3	0	1798	
8	2.53	6.3	1.89	0	0	0	
9	0.58	8.5	0.32	133	1	61745	
10	0.81	11.5	0.58	381	1	462198	
11	2.54	8.3	1.27	0	0	0	
12	1.84	7.7	1.05	0	0	0	
13	0.58	8.0	0.45	2	0	0	
14	0.81	11.0	0.63	17	0	0	
15	1.46	10.2	0.91	0	0	0	
16	1.48	10.6	1.09	6	0	2	
17	1.11	12.6	0.75	64	1	225483	

(continues on next page)

(continued from previous page)

18	1.19	11.0	0.89	66	1	331726
19	1.40	13.0	1.48	0	0	0
20	1.02	8.8	0.69	11	0	87189
21	0.70	7.8	0.52	0	0	0
22	0.69	9.2	0.53	0	0	0
23	0.77	7.4	0.60	0	0	0
24	1.54	11.1	1.01	72	1	436631
25	0.81	9.5	0.67	39	1	114931
26	2.31	9.1	1.71	0	0	0
27	1.44	8.8	0.94	0	0	15
28	1.76	10.1	1.61	0	0	0
29	0.59	6.1	0.46	0	0	18
30	1.90	10.2	1.11	2	0	0
31	0.47	6.6	0.31	0	0	0
32	0.98	9.9	0.48	142	1	331059
33	2.93	8.0	1.44	0	0	0
34	0.61	9.5	0.45	64	0	0
35	1.29	12.8	1.01	16	0	0
36	1.10	10.4	0.80	0	0	0
37	0.48	7.7	0.43	11	0	0
38	2.11	9.4	1.79	0	0	0
39	1.18	8.1	0.70	144	1	402406
40	2.64	10.9	2.50	0	0	0
41	0.85	11.4	0.75	103	1	275719
42	0.61	10.0	0.35	99	1	182566
43	1.77	10.2	0.93	0	0	0
44	2.40	9.6	1.87	0	0	0
45	0.83	8.9	0.52	40	1	490865
46	1.00	10.0	0.65	0	0	0
47	1.69	10.9	1.34	4	1	18371
48	0.79	8.3	0.57	0	0	0
49	3.92	10.3	1.90	0	0	0
Population1860 PropSlaves1860						
0	964201	0.450000				
1	0	0.000000				
2	0	0.000000				
3	435450	0.260000				
4	379994	0.000000				
5	34277	0.000000				
6	460147	0.000000				
7	112216	0.016000				
8	75080	0.000000				
9	140424	0.440000				
10	1057286	0.440000				
11	0	0.000000				
12	0	0.000000				
13	1711951	0.000000				
14	1350428	0.000000				
15	674913	0.000000				
16	107206	0.000019				
17	1155684	0.000000				
18	708002	0.470000				
19	628279	0.000000				
20	687049	0.130000				
21	1231066	0.000000				
22	749113	0.000000				

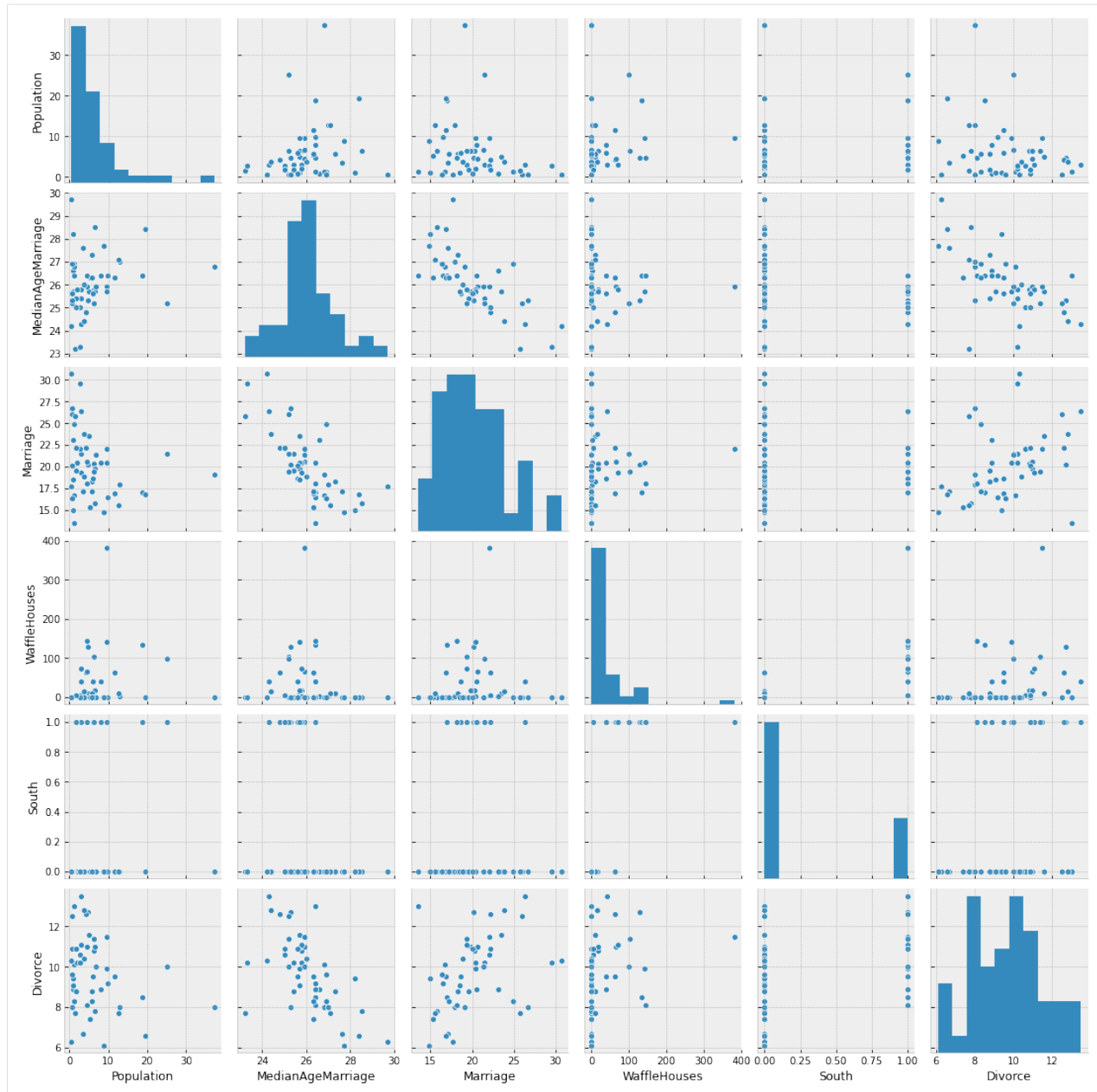
(continues on next page)

(continued from previous page)

23	172023	0.000000
24	791305	0.550000
25	1182012	0.097000
26	0	0.000000
27	28841	0.000520
28	326073	0.000000
29	672035	0.000027
30	93516	0.000000
31	3880735	0.000000
32	992622	0.330000
33	0	0.000000
34	2339511	0.000000
35	0	0.000000
36	52465	0.000000
37	2906215	0.000000
38	174620	0.000000
39	703708	0.570000
40	4837	0.000000
41	1109801	0.200000
42	604215	0.300000
43	40273	0.000000
44	315098	0.000000
45	1219630	0.400000
46	11594	0.000000
47	376688	0.049000
48	775881	0.000000
49	0	0.000000

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

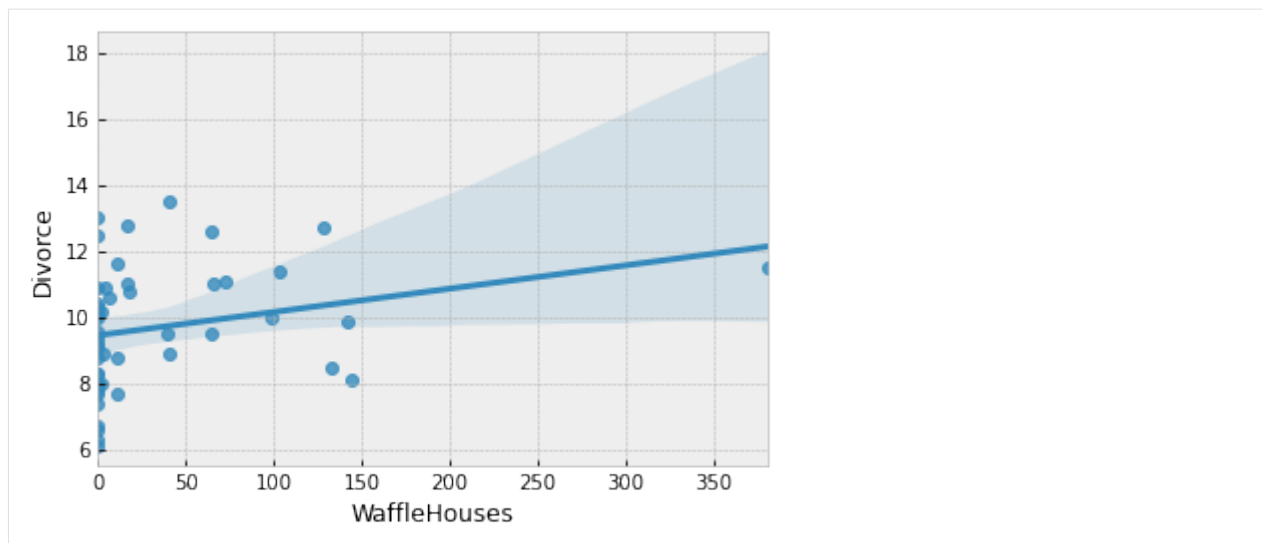
```
[4]: vars = ['Population', 'MedianAgeMarriage', 'Marriage', 'WaffleHouses', 'South',
↪ 'Divorce']
sns.pairplot(dset, x_vars=vars, y_vars=vars, palette='husl');
```



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('WaffleHouses', 'Divorce', 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.

### 3.3 Regression Model to Predict Divorce Rate

Let us now write a regression model in *NumPyro* to predict the divorce rate as a linear function of marriage rate and median age of marriage in each of the states.

First, note that our predictor variables have somewhat different scales. It is a good practice to standardize our predictors and response variables to mean 0 and standard deviation 1, which should result in [faster inference](#).

```
[6]: 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., 0.2))
    M, A = 0., 0.
    if marriage is not None:
        bM = numpyro.sample('bM', dist.Normal(0., 0.5))
        M = bM * marriage
    if age is not None:
        bA = numpyro.sample('bA', dist.Normal(0., 0.5))
        A = bA * age
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    mu = a + M + A
    numpyro.sample('obs', dist.Normal(mu, sigma), obs=divorce)
```

### 3.3.1 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)

num_warmup, num_samples = 1000, 2000

# Run NUTS.
kernel = NUTS(model)
mcmc = MCMC(kernel, num_warmup, num_samples)
mcmc.run(rng_key_, marriage=dset.MarriageScaled.values, divorce=dset.DivorceScaled.
    ↪ values)
```

(continues on next page)

(continued from previous page)

```
mcmc.print_summary()
samples_1 = mcmc.get_samples()

sample: 100%|████████████████████| 3000/3000 [00:06<00:00, 429.13it/s, 3 steps of
↳size 7.48e-01. acc. prob=0.91]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
a	-0.00	0.11	-0.00	-0.17	0.18	1804.63	1.00
bM	0.35	0.13	0.35	0.13	0.56	1556.49	1.00
sigma	0.94	0.10	0.94	0.78	1.08	1916.15	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)
    ax.plot(marriage, divorce, 'o')
    ax.fill_between(marriage, hpdi[0], hpdi[1], alpha=0.3, interpolate=True)
    return ax
```

(continues on next page)

(continued from previous page)

```
# 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.

## 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. NumPyro provides a handy `Predictive` utility for this purpose. 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.

```
[10]: from numpyro.infer import Predictive

rng_key, rng_key_ = random.split(rng_key)
predictive = Predictive(model, samples_1)
predictions = predictive(rng_key_, marriage=dset.MarriageScaled.values)['obs']
df = dset.filter(['Location'])
df['Mean Predictions'] = jnp.mean(predictions, axis=0)
df.head()
```



```
[10]:
```

	Location	Mean Predictions
0	Alabama	0.003563
1	Alaska	0.559501
2	Arizona	0.007107
3	Arkansas	0.539909
4	California	-0.077508

## 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.

```
[11]: 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.

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

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()
```

```
[12]:
```

	Location	Mean Predictions
0	Alabama	0.003563
1	Alaska	0.559501
2	Arizona	0.007107

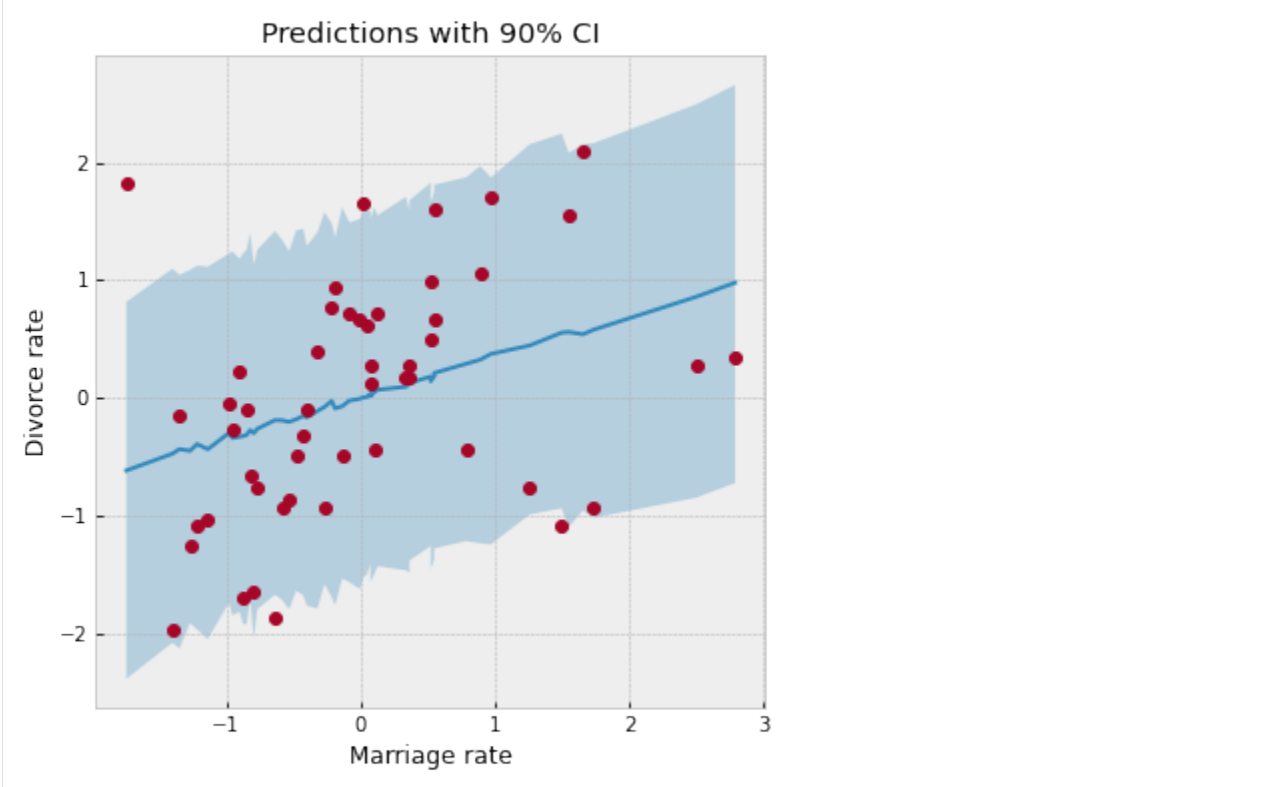
(continues on next page)

(continued from previous page)

3	Arkansas	0.539909
4	California	-0.077508

```
[13]: 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

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

Here,  $i$  indexes the observed data points  $y$  and  $s$  indexes the posterior samples over the latent parameters  $\theta$ . If the posterior predictive density for a model has a comparatively high value, it indicates that the observed data-points have higher probability under the given model.

```
[14]: 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.

```
[15]: 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: -66.65252685546875
```

### 3.3.2 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.

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

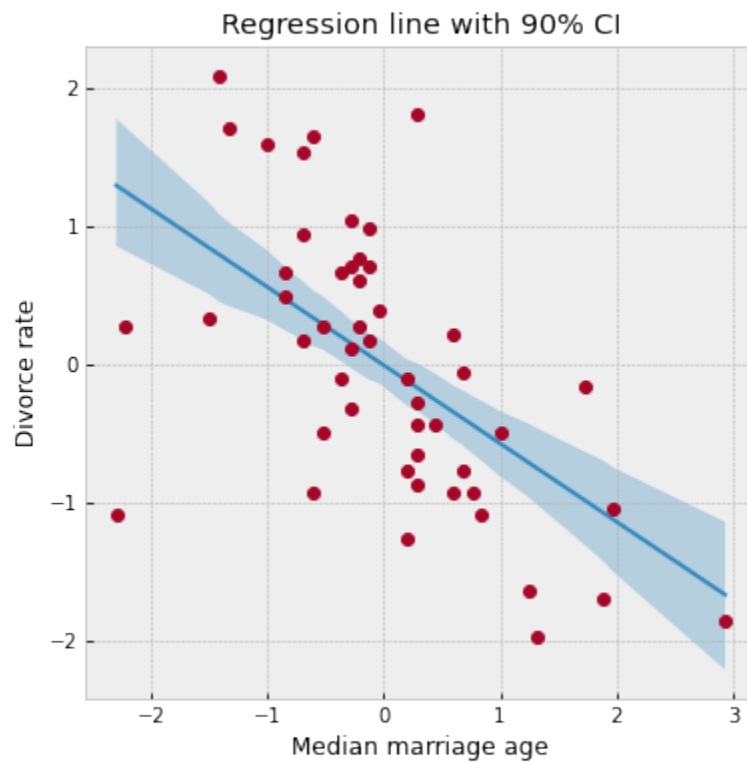
mcmc.run(rng_key_, age=dset.AgeScaled.values, divorce=dset.DivorceScaled.values)
mcmc.print_summary()
samples_2 = mcmc.get_samples()

sample: 100%|██████████| 3000/3000 [00:07<00:00, 425.30it/s, 3 steps of
    ↪size 7.68e-01. acc. prob=0.92]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
a	-0.00	0.10	-0.00	-0.16	0.16	1862.10	1.00
bA	-0.57	0.11	-0.57	-0.75	-0.39	1962.83	1.00
sigma	0.82	0.08	0.81	0.68	0.95	1544.86	1.00

Number of divergences: 0

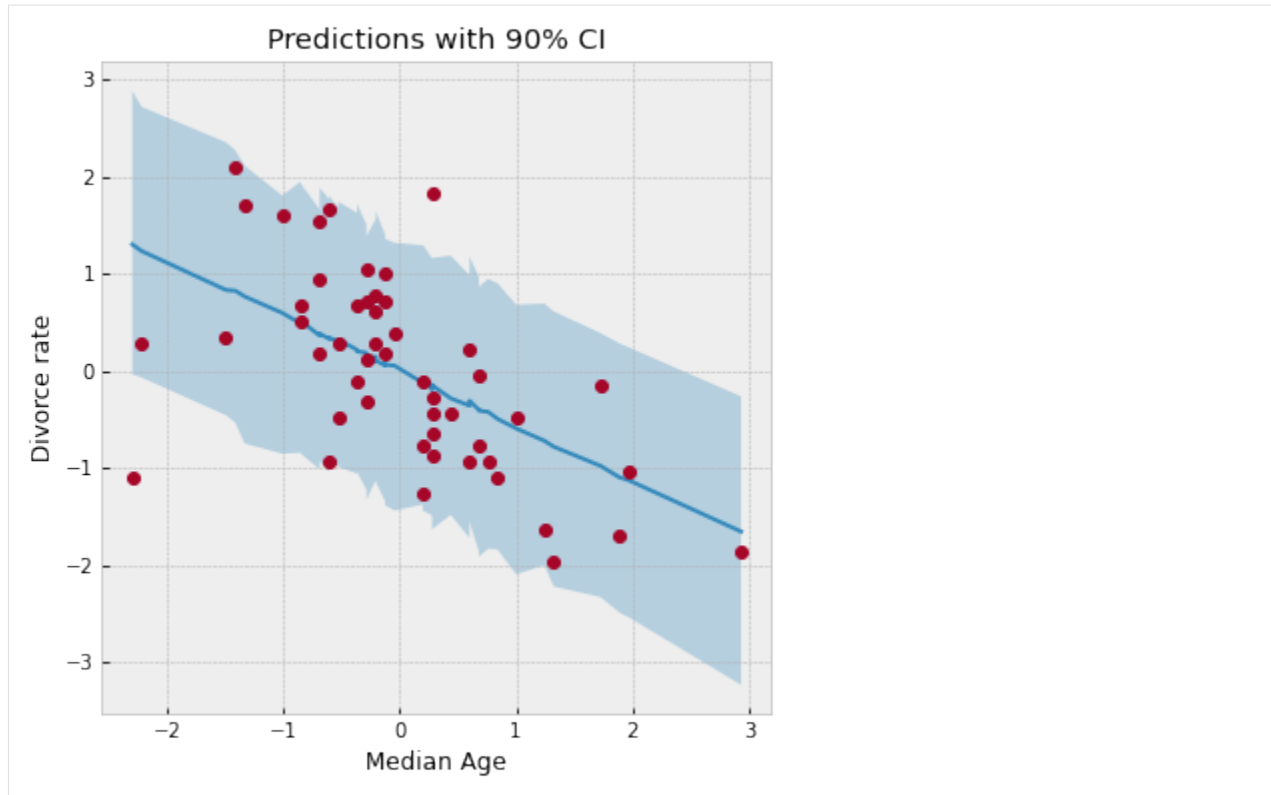
```
[17]: posterior_mu = jnp.expand_dims(samples_2['a'], -1) + \
      jnp.expand_dims(samples_2['bA'], -1) * dset.AgeScaled.values
mean_mu = jnp.mean(posterior_mu, axis=0)
hpdi_mu = hpdi(posterior_mu, 0.9)
ax = plot_regression(dset.AgeScaled.values, mean_mu, hpdi_mu)
ax.set(xlabel='Median marriage age',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



```
[18]: 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');
```



```
[19]: 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: -59.238067626953125
```

### 3.3.3 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.

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

mcmc.run(rng_key_, marriage=dset.MarriageScaled.values,
         age=dset.AgeScaled.values, divorce=dset.DivorceScaled.values)
mcmc.print_summary()
samples_3 = mcmc.get_samples()

sample: 100%|████████████████████| 3000/3000 [00:07<00:00, 389.02it/s, 7 steps of_
↪size 5.15e-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	1731.49	1.00

(continues on next page)

(continued from previous page)

bA	-0.61	0.16	-0.61	-0.88	-0.36	1446.05	1.00
bM	-0.06	0.16	-0.07	-0.32	0.20	1432.76	1.00
sigma	0.82	0.08	0.82	0.69	0.96	1654.31	1.00

Number of divergences: 0

```
[21]: 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: -59.06075668334961

### 3.3.4 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.

```
[22]: # 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)
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);

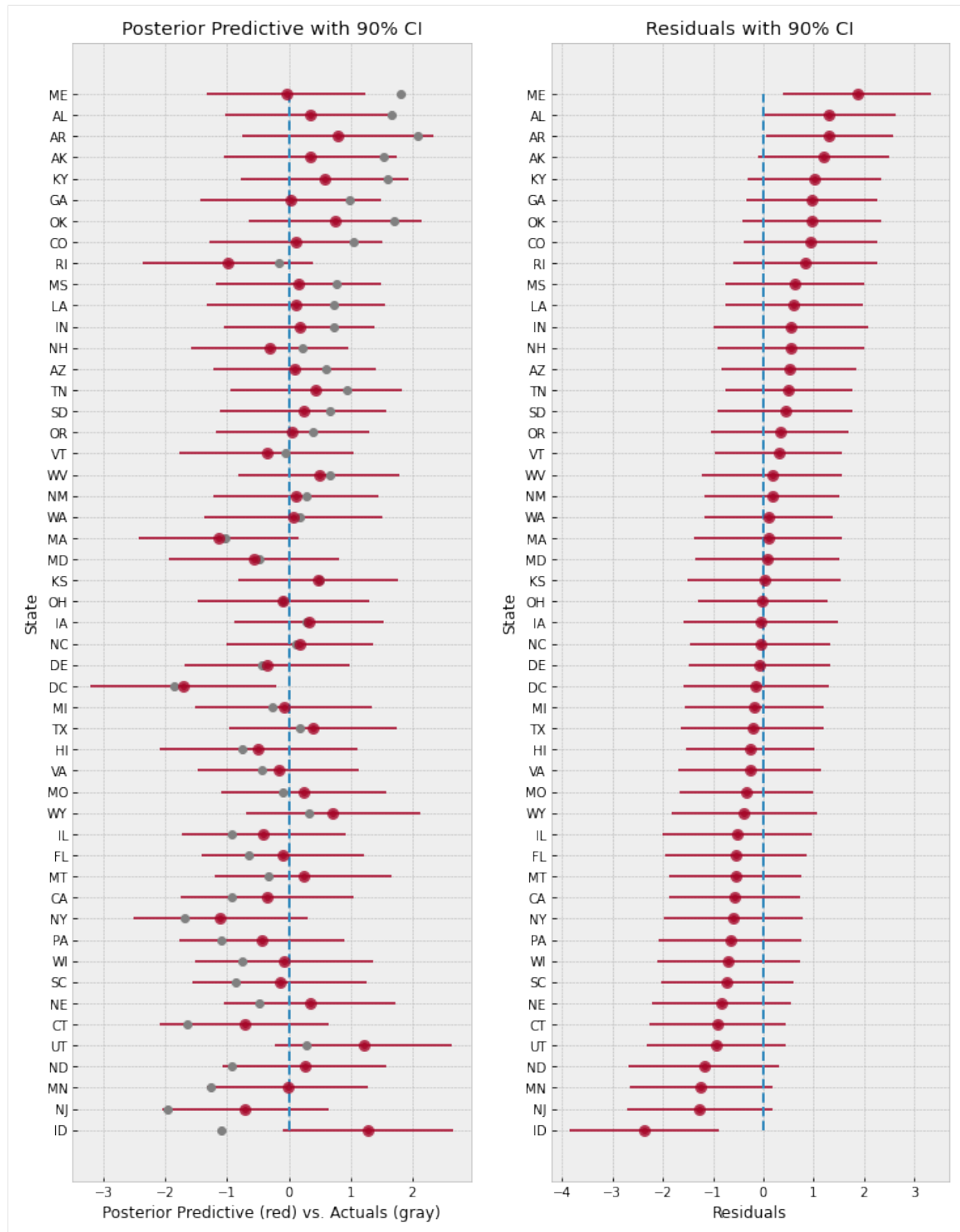
# Plot residuals
residuals_3 = dset.DivorceScaled.values - predictions_3
residuals_mean = jnp.mean(residuals_3, axis=0)
```

(continues on next page)

(continued from previous page)

```
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 lie 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.

## 3.4 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 [7].

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).

```
[23]: def model_se(marriage, age, divorce_sd, divorce=None):
      a = numpyro.sample('a', dist.Normal(0., 0.2))
      bM = numpyro.sample('bM', dist.Normal(0., 0.5))
      M = bM * marriage
      bA = numpyro.sample('bA', dist.Normal(0., 0.5))
      A = bA * age
      sigma = numpyro.sample('sigma', dist.Exponential(1.))
      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)
```

```
[24]: # Standardize
      dset['DivorceScaledSD'] = dset['Divorce SE'] / jnp.std(dset.Divorce.values)
```

```
[25]: 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 [00:09<00:00, 408.70it/s, 15 steps of
↳ size 3.00e-01. acc. prob=0.91]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
a	-0.06	0.09	-0.06	-0.21	0.10	3381.24	1.00
bA	-0.61	0.16	-0.61	-0.86	-0.32	2457.71	1.00
bM	0.06	0.17	0.06	-0.19	0.35	2384.97	1.00
divorce_rate[0]	1.14	0.37	1.14	0.54	1.74	4196.29	1.00

(continues on next page)

(continued from previous page)

divorce_rate[1]	0.69	0.55	0.68	-0.24	1.54	4457.97	1.00
divorce_rate[2]	0.43	0.34	0.43	-0.12	0.97	4692.82	1.00
divorce_rate[3]	1.41	0.47	1.41	0.71	2.21	5338.40	1.00
divorce_rate[4]	-0.90	0.13	-0.90	-1.12	-0.69	6373.86	1.00
divorce_rate[5]	0.65	0.39	0.64	0.00	1.29	5890.75	1.00
divorce_rate[6]	-1.36	0.35	-1.35	-1.90	-0.74	6029.70	1.00
divorce_rate[7]	-0.33	0.48	-0.32	-1.13	0.42	6226.93	1.00
divorce_rate[8]	-1.88	0.61	-1.89	-2.84	-0.86	3797.97	1.00
divorce_rate[9]	-0.62	0.16	-0.62	-0.88	-0.33	7313.05	1.00
divorce_rate[10]	0.76	0.28	0.75	0.31	1.24	5122.44	1.00
divorce_rate[11]	-0.54	0.47	-0.55	-1.32	0.22	4301.37	1.00
divorce_rate[12]	0.20	0.51	0.21	-0.60	1.07	2646.56	1.00
divorce_rate[13]	-0.87	0.23	-0.86	-1.24	-0.49	6676.11	1.00
divorce_rate[14]	0.55	0.31	0.55	0.04	1.03	5340.67	1.00
divorce_rate[15]	0.29	0.38	0.29	-0.35	0.89	7546.96	1.00
divorce_rate[16]	0.51	0.43	0.50	-0.23	1.17	4965.65	1.00
divorce_rate[17]	1.24	0.34	1.25	0.69	1.82	4936.42	1.00
divorce_rate[18]	0.42	0.38	0.42	-0.15	1.11	5954.93	1.00
divorce_rate[19]	0.38	0.56	0.37	-0.53	1.28	2854.65	1.00
divorce_rate[20]	-0.56	0.32	-0.55	-1.09	-0.05	5861.17	1.00
divorce_rate[21]	-1.10	0.26	-1.10	-1.53	-0.66	5929.05	1.00
divorce_rate[22]	-0.28	0.27	-0.28	-0.70	0.16	6215.17	1.00
divorce_rate[23]	-0.99	0.30	-1.00	-1.44	-0.45	4756.94	1.00
divorce_rate[24]	0.42	0.42	0.41	-0.25	1.10	6284.63	1.00
divorce_rate[25]	-0.03	0.32	-0.03	-0.53	0.53	6587.60	1.00
divorce_rate[26]	-0.02	0.50	-0.02	-0.88	0.75	5119.31	1.00
divorce_rate[27]	-0.15	0.38	-0.14	-0.78	0.44	4868.48	1.00
divorce_rate[28]	-0.26	0.50	-0.27	-1.06	0.61	4898.44	1.00
divorce_rate[29]	-1.79	0.24	-1.80	-2.19	-1.40	5940.81	1.00
divorce_rate[30]	0.18	0.43	0.17	-0.60	0.80	6070.03	1.00
divorce_rate[31]	-1.66	0.16	-1.66	-1.92	-1.38	7048.38	1.00
divorce_rate[32]	0.12	0.24	0.11	-0.24	0.52	6550.93	1.00
divorce_rate[33]	-0.03	0.53	-0.01	-0.92	0.81	4108.95	1.00
divorce_rate[34]	-0.13	0.22	-0.13	-0.51	0.21	7752.43	1.00
divorce_rate[35]	1.27	0.40	1.27	0.66	1.99	5621.12	1.00
divorce_rate[36]	0.23	0.36	0.22	-0.36	0.81	7705.96	1.00
divorce_rate[37]	-1.02	0.22	-1.02	-1.40	-0.68	5115.13	1.00
divorce_rate[38]	-0.93	0.54	-0.93	-1.84	-0.10	3426.71	1.00
divorce_rate[39]	-0.67	0.32	-0.67	-1.23	-0.17	6310.21	1.00
divorce_rate[40]	0.25	0.54	0.25	-0.60	1.15	4844.06	1.00
divorce_rate[41]	0.73	0.35	0.74	0.12	1.27	5814.62	1.00
divorce_rate[42]	0.20	0.18	0.20	-0.12	0.48	8983.24	1.00
divorce_rate[43]	0.82	0.42	0.84	0.13	1.52	3795.81	1.00
divorce_rate[44]	-0.42	0.52	-0.43	-1.25	0.48	4979.32	1.00
divorce_rate[45]	-0.38	0.25	-0.38	-0.80	0.02	6811.34	1.00
divorce_rate[46]	0.13	0.31	0.14	-0.38	0.64	6527.01	1.00
divorce_rate[47]	0.57	0.48	0.56	-0.22	1.33	6760.92	1.00
divorce_rate[48]	-0.63	0.27	-0.63	-1.08	-0.20	6560.02	1.00
divorce_rate[49]	0.86	0.59	0.87	-0.13	1.78	4066.94	1.00
sigma	0.58	0.11	0.57	0.41	0.76	1067.58	1.00

Number of divergences: 0

### 3.4.1 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.

```
[26]: 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']
```

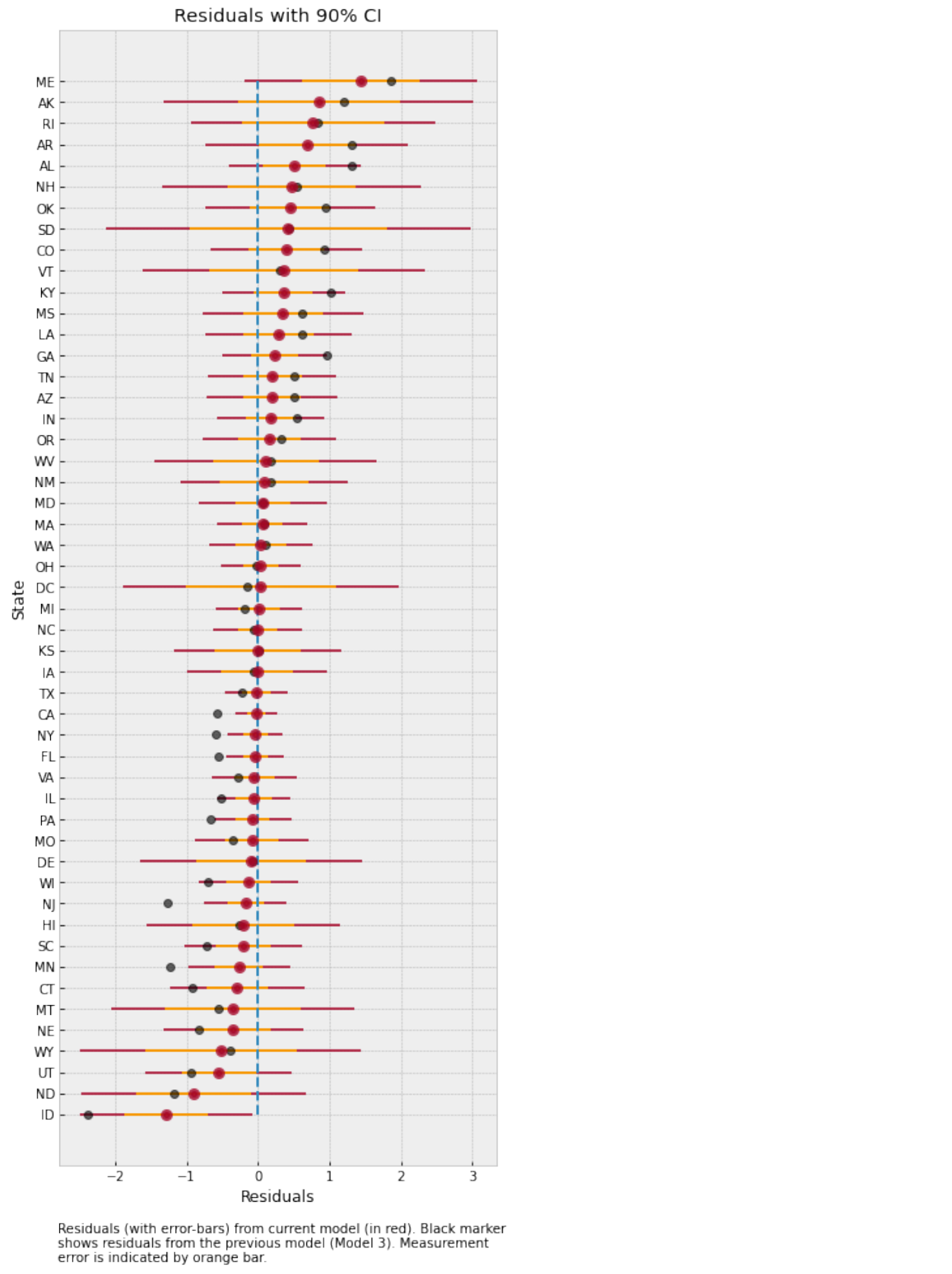
```
[27]: sd = dset.DivorceScaledSD.values
      residuals_4 = dset.DivorceScaled.values - predictions_4
      residuals_mean = jnp.mean(residuals_4, axis=0)
      residuals_hpdi = hpdi(residuals_4, 0.9)
      err = residuals_hpdi[1] - residuals_mean
      idx = jnp.argsort(residuals_mean)
      y = 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)

      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.');
```



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.

```
[28]: fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10, 6))
x = dset.DivorceScaledSD.values
y1 = jnp.mean(residuals_3, 0)
y2 = jnp.mean(residuals_4, 0)
ax.plot(x, y1, ls='none', marker='o')
ax.plot(x, y2, ls='none', marker='o')
for i, (j, k) in enumerate(zip(y1, y2)):
    ax.plot([x[i], x[i]], [j, k], '--', color='gray');

ax.set(xlabel='Measurement Noise', ylabel='Residual', title='Mean residuals (Model 4: ↵
↵red, Model 3: blue)');
```



The plot above shows what has happened in more detail - the regression line itself has moved to ensure a better fit for observations with low measurement noise (left of the plot) where the residuals have shrunk very close to 0. That is to say that data points with low measurement error have a concomitantly higher contribution in determining the regression line. On the other hand, for states with high measurement error (right of the plot), incorporating measurement noise allows us to move our posterior distribution mass closer to the observations resulting in a shrinkage of residuals as well.

## 3.5 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 Stuhlmüller, 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*

---

## Bayesian Hierarchical Linear Regression

---

Author: Carlos Souza

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.

### 4.1 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]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

[2]: train = pd.read_csv('https://gist.githubusercontent.com/ucals/'
                        '2cf9d101992cb1b78c2cdd6e3bac6a4b/raw/'
                        '43034c39052dcf97d4b894d2ec1bc3f90f3623d9/')
```

(continues on next page)

(continued from previous page)

```

train.head()
[2]:

```

	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.

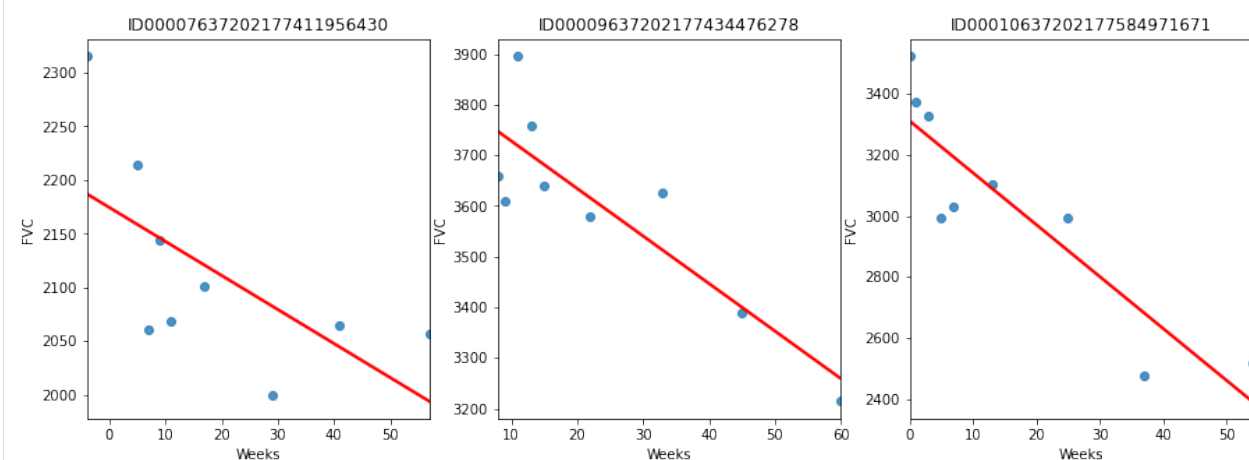
Since this is real medical data, the relative timing of FVC measurements varies widely, as shown in the 3 sample patients below:

```

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

f, axes = plt.subplots(1, 3, figsize=(15, 5))
chart('ID00007637202177411956430', axes[0])
chart('ID00009637202177434476278', axes[1])
chart('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 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:



	W1	W2	W3	W4	W5	W6	W7	...	Wt
Patient 1	NA	2315	NA	2101	2000	NA	1950		NA
Patient 2	3660	NA	NA	3420	NA	3150	NA		2870
Patient 3	NA	NA	2945	NA	2660	2520	NA		NA
Patient 4	3326	3419	NA	3269	NA	NA	3193		2994
...									
Patient N	NA	2100	NA	NA	1808	NA	NA		NA

Legend:

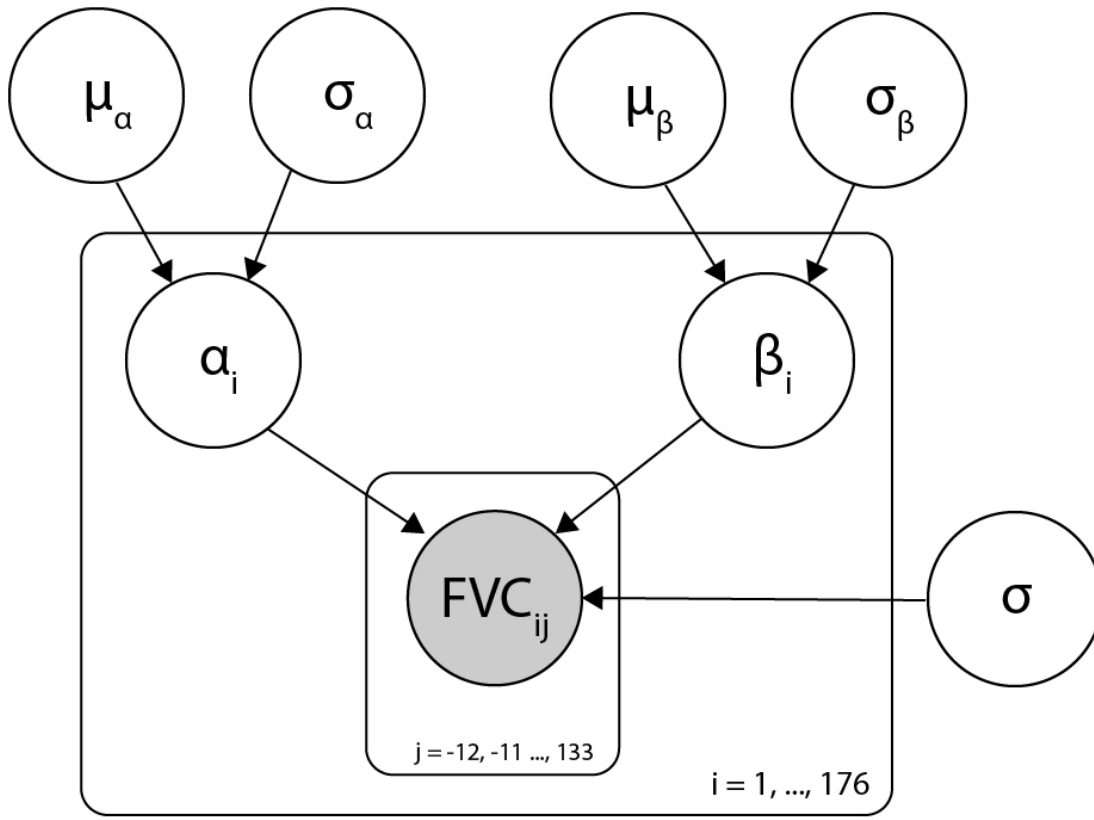
- Training data
- NA No observation (to be predicted)

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].

## 4.2 2. Modelling: Bayesian Hierarchical Linear Regression with Partial Pooling

The simplest possible linear regression, not hierarchical, would assume all FVC decline curves have the same  $\alpha$  and  $\beta$ . That's the **pooled model**. In the other extreme, we could assume a model where each patient has a personalized FVC decline curve, and **these curves are completely unrelated**. That's the **unpooled model**, where each patient has completely separate regressions.

Here, I'll use the middle ground: **Partial pooling**. Specifically, I'll assume that while  $\alpha$ 's and  $\beta$ 's are different for each patient as in the unpooled case, **the coefficients all share similarity**. We can model this by assuming that each individual coefficient comes from a common group distribution. The image below represents this model graphically:



Mathematically, the model is described by the following equations:

$$\mu_\alpha \sim \mathcal{N}(0, 100) \quad (4.1)$$

$$\sigma_\alpha \sim |\mathcal{N}(0, 100)| \quad (4.2)$$

$$\mu_\beta \sim \mathcal{N}(0, 100) \quad (4.3)$$

$$\sigma_\beta \sim |\mathcal{N}(0, 100)| \quad (4.4)$$

$$\alpha_i \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha) \quad (4.5)$$

$$\beta_i \sim \mathcal{N}(\mu_\beta, \sigma_\beta) \quad (4.6)$$

$$\sigma \sim \mathcal{N}(0, 100) \quad (4.7)$$

$$FVC_{ij} \sim \mathcal{N}(\alpha_i + t\beta_i, \sigma) \quad (4.8)$$

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:

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

```
[5]: def model(PatientID, Weeks, FVC_obs=None):
    μ_α = numpyro.sample("μ_α", dist.Normal(0., 100.))
    σ_α = numpyro.sample("σ_α", dist.HalfNormal(100.))
    μ_β = numpyro.sample("μ_β", dist.Normal(0., 100.))
```

(continues on next page)

(continued from previous page)

```

 $\sigma_\beta$  = numpyro.sample(" $\sigma_\beta$ ", dist.HalfNormal(100.))

unique_patient_IDs = np.unique(PatientID)
n_patients = len(unique_patient_IDs)

with numpyro.plate("plate_i", n_patients):
     $\alpha$  = numpyro.sample(" $\alpha$ ", dist.Normal( $\mu_\alpha$ ,  $\sigma_\alpha$ ))
     $\beta$  = numpyro.sample(" $\beta$ ", dist.Normal( $\mu_\beta$ ,  $\sigma_\beta$ ))

 $\sigma$  = numpyro.sample(" $\sigma$ ", dist.HalfNormal(100.))
FVC_est =  $\alpha$ [PatientID] +  $\beta$ [PatientID] * Weeks

with numpyro.plate("data", len(PatientID)):
    numpyro.sample("obs", dist.Normal(FVC_est,  $\sigma$ ), obs=FVC_obs)

```

That's all for modelling!

## 4.3 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:

```

[6]: from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
train['PatientID'] = le.fit_transform(train['Patient'].values)

FVC_obs = train['FVC'].values
Weeks = train['Weeks'].values
PatientID = train['PatientID'].values

```

Now, calling NumPyro's inference engine:

```

[7]: nuts_kernel = NUTS(model)

mcmc = MCMC(nuts_kernel, num_samples=2000, num_warmup=2000)
rng_key = random.PRNGKey(0)
mcmc.run(rng_key, PatientID, Weeks, FVC_obs=FVC_obs)

posterior_samples = mcmc.get_samples()

sample: 100%|████████████████████| 4000/4000 [00:20<00:00, 195.69it/s, 63 steps of_
↪ size 1.06e-01. acc. prob=0.89]

```

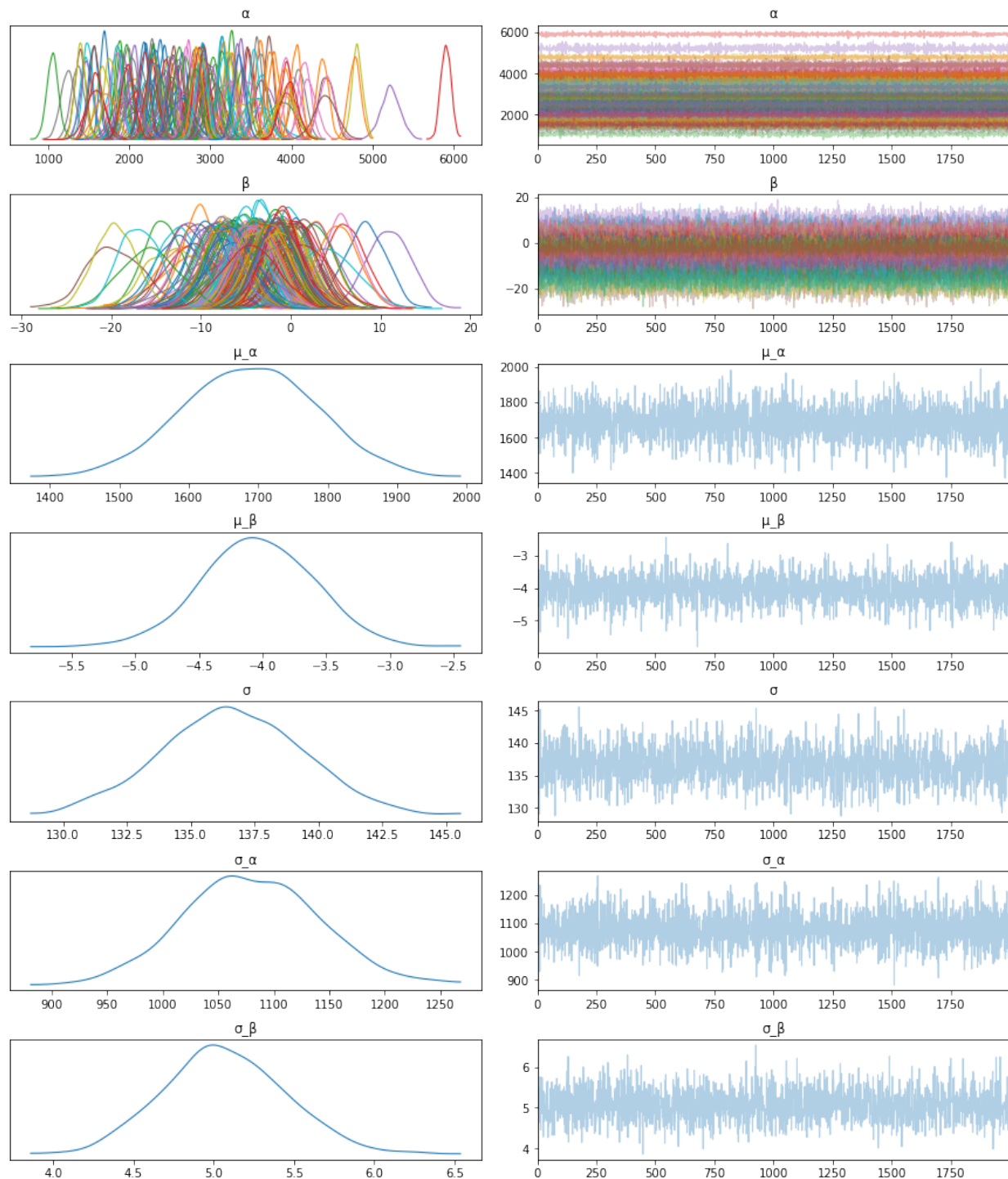
## 4.4 4. Checking the model

### 4.4.1 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:

```
[8]: import arviz as az

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



Looks like our model learned personalized alphas and betas for each patient!

## 4.4.2 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:

```
[9]: pred_template = []
for i in range(train['Patient'].nunique()):
    df = pd.DataFrame(columns=['PatientID', 'Weeks'])
    df['Weeks'] = np.arange(-12, 134)
    df['PatientID'] = i
    pred_template.append(df)
pred_template = pd.concat(pred_template, ignore_index=True)
```

Predicting the missing values in the FVC table and confidence (sigma) for each value becomes really easy:

```
[10]: PatientID = pred_template['PatientID'].values
Weeks = pred_template['Weeks'].values
predictive = Predictive(model, posterior_samples,
                        return_sites=[' $\sigma$ ', 'obs'])
samples_predictive = predictive(random.PRNGKey(0),
                               PatientID, Weeks, None)
```

Let's now put the predictions together with the true values, to visualize them:

```
[11]: df = pd.DataFrame(columns=['Patient', 'Weeks', 'FVC_pred', 'sigma'])
df['Patient'] = le.inverse_transform(pred_template['PatientID'])
df['Weeks'] = pred_template['Weeks']
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()
```

```
[11]:
```

	Patient	Weeks	FVC_pred	sigma	FVC_inf	\
0	ID00007637202177411956430	-12	2219.361084	159.272430	2060.088623	
1	ID00007637202177411956430	-11	2209.278076	157.698868	2051.579102	
2	ID00007637202177411956430	-10	2212.443115	154.503906	2057.939209	
3	ID00007637202177411956430	-9	2208.173096	153.068268	2055.104736	
4	ID00007637202177411956430	-8	2202.373047	157.185608	2045.187500	

	FVC_sup	FVC_true
0	2378.633545	NaN
1	2366.977051	NaN
2	2366.947021	NaN
3	2361.241455	NaN
4	2359.558594	NaN

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

```
[12]: def chart(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'])
```

(continues on next page)

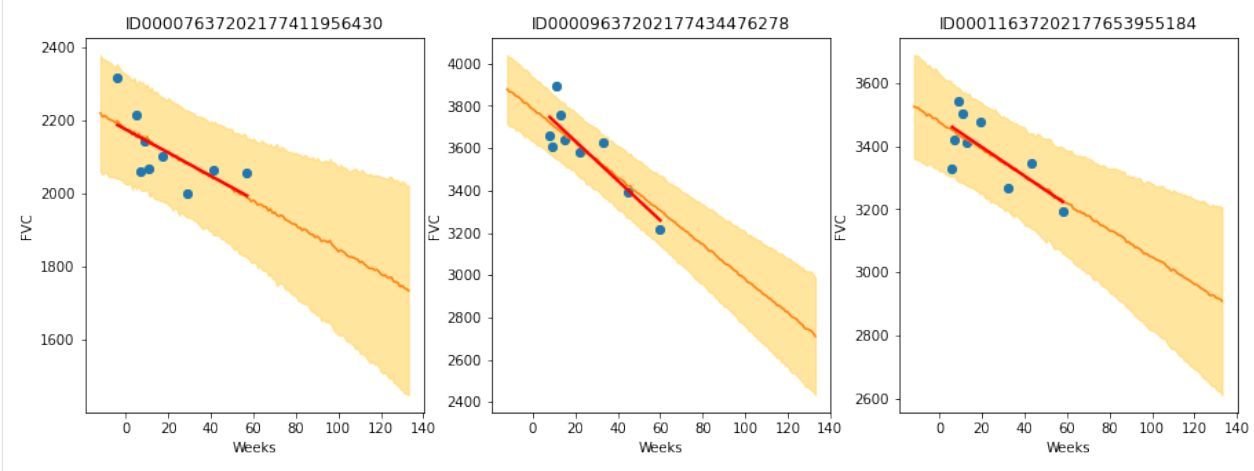
(continued from previous page)

```

ax = sns.regplot(x, 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('ID00007637202177411956430', axes[0])
chart('ID00009637202177434476278', axes[1])
chart('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!

#### 4.4.3 4.3. Computing the modified Laplace Log Likelihood and RMSE

As mentioned earlier, the competition was evaluated on a modified version of the Laplace Log Likelihood. In medical applications, it is useful to evaluate a model's confidence in its decisions. Accordingly, the metric is designed to reflect both the accuracy and certainty of each prediction.

For each true FVC measurement, we predicted both an FVC and a confidence measure (standard deviation  $\sigma$ ). The metric was computed as:

$$\sigma_{clipped} = \max(\sigma, 70) \quad (4.9)$$

$$\delta = \min(|FVC_{true} - FVC_{pred}|, 1000) \quad (4.10)$$

$$metric = -\frac{\sqrt{2}\delta}{\sigma_{clipped}} - \ln(\sqrt{2}\sigma_{clipped}) \quad (4.11)$$

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:

```
[13]: 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
l1l = - np.sqrt(2) * delta / sigma_c - np.log(np.sqrt(2) * sigma_c)
print(f'Laplace Log Likelihood: {l1l.mean():.4f}')
```

RMSE: 122.1 ml  
Laplace Log Likelihood: -6.1376

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.**”

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.

## 4.5 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.





---

## 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

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):
    """
    Number of hits in  $K$  at bats for each player has a Binomial
    distribution with a common probability of success,  $\phi$ .

    :param (jnp.DeviceArray) at_bats: Number of at bats for each player.
    :param (jnp.DeviceArray) 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):
    """
    Number of hits in  $K$  at bats for each player has a Binomial
    distribution with independent probability of success,  $\phi_i$ .

    :param (jnp.DeviceArray) at_bats: Number of at bats for each player.
    :param (jnp.DeviceArray) 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):
    """
    Number of hits has a Binomial distribution with independent
    probability of success,  $\phi_i$ . Each  $\phi_i$  follows a Beta
    distribution with concentration parameters  $\zeta_1$  and  $\zeta_2$ , where
 $\zeta_1 = m * \kappa$ ,  $\zeta_2 = (1 - m) * \kappa$ ,  $m \sim \text{Uniform}(0, 1)$ ,
    and  $\kappa \sim \text{Pareto}(1, 1.5)$ .

    :param (jnp.DeviceArray) at_bats: Number of at bats for each player.
    :param (jnp.DeviceArray) hits: Number of hits for the given at bats.
    :return: Number of hits predicted by the model.

```

(continues on next page)

(continued from previous page)

```

"""
m = numpyro.sample("m", dist.Uniform(0, 1))
kappa = numpyro.sample("kappa", dist.Pareto(1, 1.5))
num_players = at_bats.shape[0]
with numpyro.plate("num_players", num_players):
    phi_prior = dist.Beta(m * kappa, (1 - m) * kappa)
    phi = numpyro.sample("phi", phi_prior)
    return numpyro.sample("obs", dist.Binomial(at_bats, probs=phi), obs=hits)

def partially_pooled_with_logit(at_bats, hits=None):
    """
    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.DeviceArray) at_bats: Number of at bats for each player.
    :param (jnp.DeviceArray) 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, args.num_warmup, args.num_samples, num_chains=args.num_chains,
                progress_bar=False if (
                    "NUMPYRO_SPHINXBUILD" in os.environ or args.disable_progbar) else
    True)
    mcmc.run(rng_key, at_bats, hits)
    return mcmc.get_samples()

def predict(model, at_bats, hits, z, rng_key, player_names, train=True):
    header = model.__name__ + (' - TRAIN' if train else ' - TEST')
    predictions = Predictive(model, posterior_samples=z)(rng_key, at_bats)['obs']
    print_results('=' * 30 + header + '=' * 30,
                  predictions,
                  player_names,
                  at_bats,
                  hits)

    if not train:
        post_loglik = log_likelihood(model, z, at_bats, hits)['obs']
        # computes expected log predictive density at each data point
        exp_log_density = logsumexp(post_loglik, axis=0) - jnp.log(jnp.shape(post_
    loglik)[0])
        # reports log predictive density of all test points

```

(continues on next page)

(continued from previous page)

```

        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('0.5.0')
    parser = argparse.ArgumentParser(description="Baseball batting average using MCMC
↪")
    parser.add_argument("-n", "--num-samples", nargs="?", default=3000, type=int)
    parser.add_argument("--num-warmup", nargs='?', default=1500, type=int)
    parser.add_argument("--num-chains", nargs='?', default=1, type=int)
    parser.add_argument('--algo', default='NUTS', type=str,
                        help='whether to run "HMC", "NUTS", or "SA"')
    parser.add_argument('-dp', '--disable-progbar', action="store_true",
↪ default=False,
                        help="whether to disable progress bar")
    parser.add_argument('--device', default='cpu', type=str, help='use "cpu" or "gpu".
↪')
    args = parser.parse_args()

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

    main(args)

```

Example: Variational Autoencoder

---

```
import argparse
import inspect
import os
import time

import matplotlib.pyplot as plt

from jax import jit, lax, random
from jax.experimental 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):
```

(continues on next page)

(continued from previous page)

```

    return stax.serial(
        stax.Dense(hidden_dim, W_init=stax.randn()), stax.Softplus,
        stax.Dense(out_dim, W_init=stax.randn()), stax.Sigmoid,
    )

def model(batch, hidden_dim=400, z_dim=100):
    batch = jnp.reshape(batch, (batch.shape[0], -1))
    batch_dim, out_dim = jnp.shape(batch)
    decode = numpyro.module('decoder', decoder(hidden_dim, out_dim), (batch_dim, z_
    ↪dim))
    z = numpyro.sample('z', dist.Normal(jnp.zeros((z_dim,)), jnp.ones((z_dim,))))
    img_loc = decode(z)
    return numpyro.sample('obs', dist.Bernoulli(img_loc), 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)
    z = numpyro.sample('z', dist.Normal(z_loc, z_std))
    return z

@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):
        def body_fn(i, val):
            loss_sum, svi_state = val
            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

```

(continues on next page)

(continued from previous page)

```

    return lax.fori_loop(0, num_train, body_fn, (0., svi_state))

@jit
def eval_test(svi_state, rng_key):
    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.)
    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)
        rng_key, rng_key_test, rng_key_reconstruct = random.split(rng_key, 3)
        num_test, test_idx = test_init()
        test_loss = eval_test(svi_state, rng_key_test)
        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('0.5.0')
    parser = argparse.ArgumentParser(description="parse args")
    parser.add_argument('-n', '--num-epochs', default=15, type=int, help='number of
    ↪training epochs')
    parser.add_argument('-lr', '--learning-rate', default=1.0e-3, type=float, help=
    ↪'learning rate')
    parser.add_argument('-batch-size', default=128, type=int, help='batch size')
    parser.add_argument('-z-dim', default=50, type=int, help='size of latent')
    parser.add_argument('-hidden-dim', default=400, type=int, help='size of hidden
    ↪layer in encoder/decoder networks')
    args = parser.parse_args()
    main(args)

```





---

## Example: Neal's Funnel

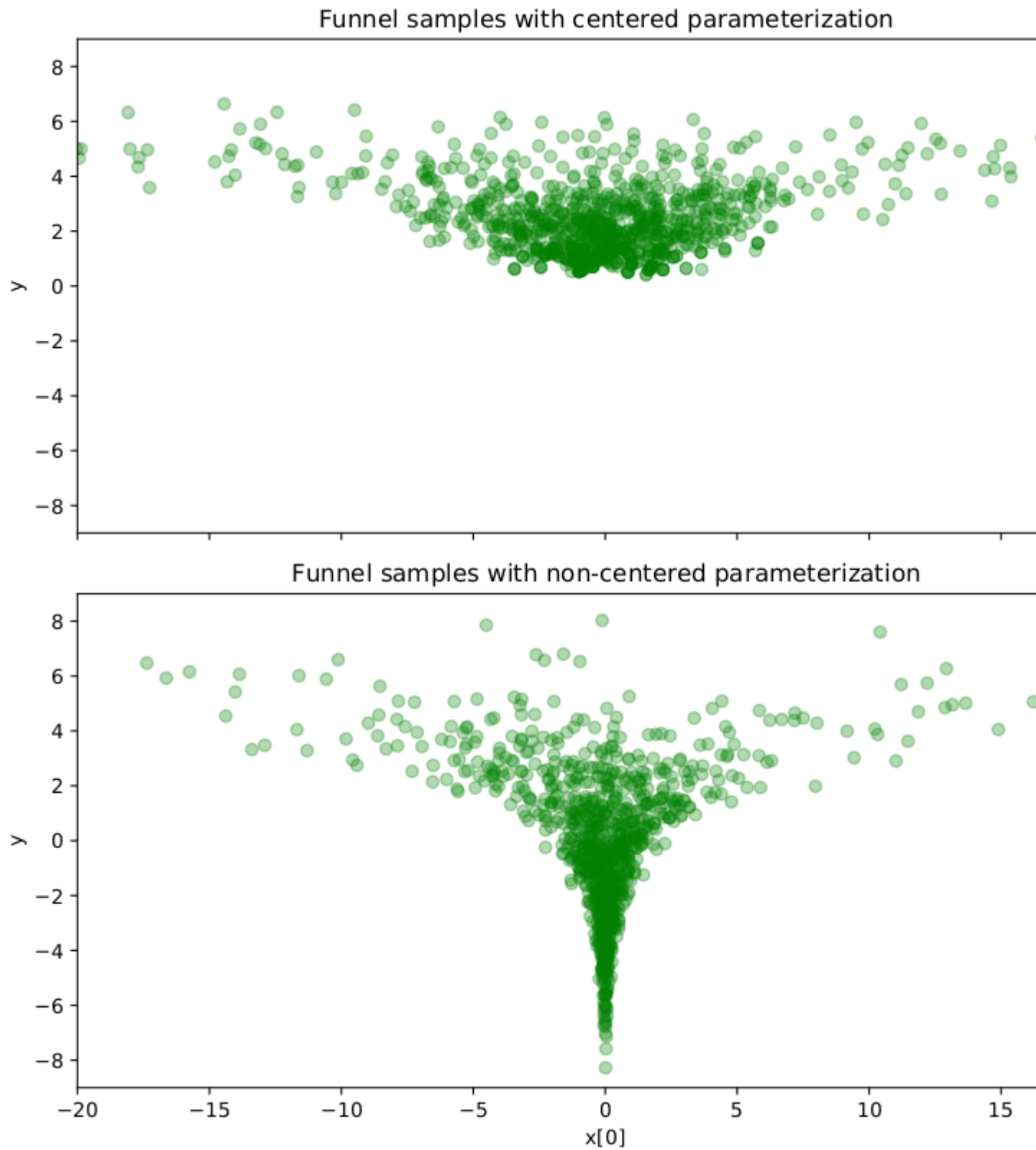
---

This example, which is adapted from [1], illustrates how to leverage non-centered parameterization using the `reparam` handler. We will examine the difference between two types of parameterizations on the 10-dimensional Neal's funnel distribution. As we will see, HMC gets trouble at the neck of the funnel if centered parameterization is used. On the contrary, the problem can be solved by using non-centered parameterization.

Using non-centered parameterization through `LocScaleReparam` or `TransformReparam` in NumPyro has the same effect as the automatic reparameterisation technique introduced in [2].

### References:

1. *Stan User's Guide*, [https://mc-stan.org/docs/2\\_19/stan-users-guide/reparameterization-section.html](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
```

(continues on next page)

(continued from previous page)

```

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

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, args.num_warmup, 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.get_samples()

def main(args):
    rng_key = random.PRNGKey(0)

    # do inference with centered parameterization
    print("===== Centered Parameterization_")
    ↪=====")
    samples = run_inference(model, args, rng_key)

    # do inference with non-centered parameterization
    print("\n===== Non-centered Parameterization_")
    ↪=====")
    reparam_samples = run_inference(reparam_model, args, rng_key)
    # 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)

    ax1.plot(samples['x'][:, 0], samples['y'], "go", alpha=0.3)
    ax1.set(xlim=(-20, 20), ylim=(-9, 9), ylabel='y',
            title='Funnel samples with centered parameterization')

    ax2.plot(reparam_samples['x'][:, 0], reparam_samples['y'], "go", alpha=0.3)
    ax2.set(xlim=(-20, 20), ylim=(-9, 9), xlabel='x[0]', ylabel='y',

```

(continues on next page)

(continued from previous page)

```
        title='Funnel samples with non-centered parameterization')

plt.savefig('funnel_plot.pdf')

if __name__ == "__main__":
    assert numpyro.__version__.startswith('0.5.0')
    parser = argparse.ArgumentParser(description="Non-centered reparameterization_
↪example")
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs='?', default=1000, type=int)
    parser.add_argument("--num-chains", nargs='?', default=1, type=int)
    parser.add_argument("--device", default='cpu', type=str, help='use "cpu" or "gpu".
↪')
    args = parser.parse_args()

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

    main(args)
```

---

## Example: Stochastic Volatility

---

Generative model:

$$\sigma \sim \text{Exponential}(50) \quad (8.1)$$

$$\nu \sim \text{Exponential}(.1) \quad (8.2)$$

$$s_i \sim \text{Normal}(s_{i-1}, \sigma^{-2}) \quad (8.3)$$

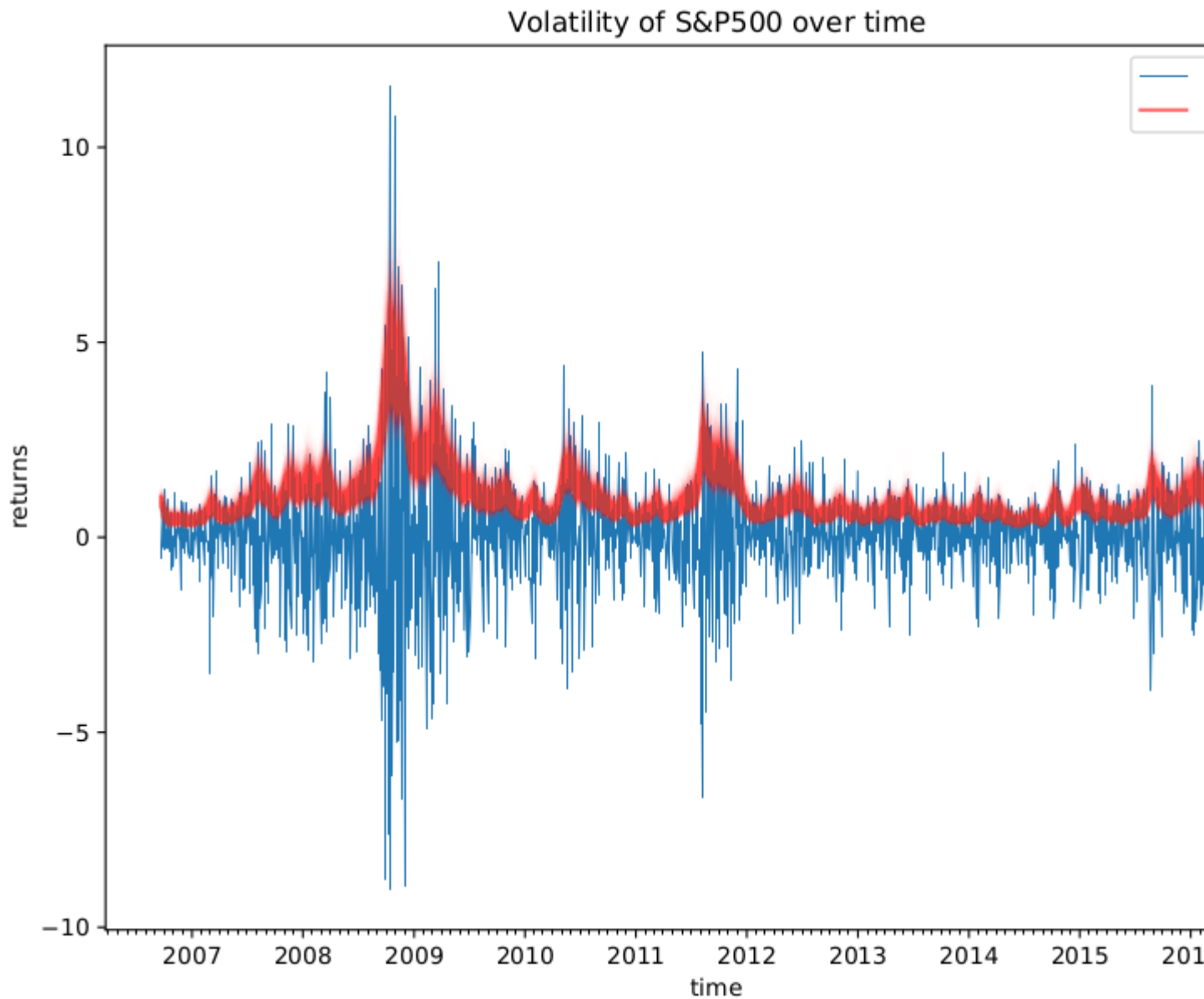
$$r_i \sim \text{StudentT}(\nu, 0, \exp(s_i)) \quad (8.4)$$

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](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
```

(continues on next page)

(continued from previous page)

```

matplotlib.use('Agg') # noqa: E402

def model(returns):
    step_size = numpyro.sample('sigma', dist.Exponential(50.))
    s = numpyro.sample('s', dist.GaussianRandomWalk(scale=step_size, num_steps=jnp.
↳shape(returns)[0]))
    nu = numpyro.sample('nu', dist.Exponential(.1))
    return numpyro.sample('r', dist.StudentT(df=nu, loc=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(q * 100) for q in quantiles]
        q_values = jnp.quantile(values, quantiles, axis=0)
        print(header_format.format('', *columns))
        print(row_format.format(row_name, *q_values))
        print('\n')

    print('=' * 20, 'sigma', '=' * 20)
    _print_row(posterior['sigma'])
    print('=' * 20, 'nu', '=' * 20)
    _print_row(posterior['nu'])
    print('=' * 20, 'volatility', '=' * 20)
    for i in range(0, len(dates), 180):
        _print_row(jnp.exp(posterior['s'][:, i]), dates[i])

def main(args):
    _, fetch = load_dataset(SP500, shuffle=False)
    dates, returns = fetch()
    init_rng_key, sample_rng_key = random.split(random.PRNGKey(args.rng_seed))
    model_info = initialize_model(init_rng_key, model, model_args=(returns,))
    init_kernel, sample_kernel = hmc(model_info.potential_fn, algo='NUTS')
    hmc_state = init_kernel(model_info.param_info, args.num_warmup, rng_key=sample_
↳rng_key)
    hmc_states = fori_collect(args.num_warmup, args.num_warmup + args.num_samples,
↳sample_kernel, hmc_state,
                                transform=lambda hmc_state: model_info.postprocess_
↳fn(hmc_state.z),
                                progbar=False if "NUMPYRO_SPHINXBUILD" in os.environ_
↳else True)
    print_results(hmc_states, dates)

    fig, ax = plt.subplots(figsize=(8, 6), constrained_layout=True)
    dates = mdates.num2date(mdates.datestr2num(dates))
    ax.plot(dates, returns, lw=0.5)
    # format the ticks
    ax.xaxis.set_major_locator(mdates.YearLocator())
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
    ax.xaxis.set_minor_locator(mdates.MonthLocator())

```

(continues on next page)

(continued from previous page)

```
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('0.5.0')
    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)
```



---

## Example: Bayesian Models of Annotation

---

In this example, we run MCMC for various crowdsourced annotation models in [1].

All models have discrete latent variables. Under the hood, we enumerate over (marginalize out) those discrete latent sites in inference. Those models have different complexity so they are great references for those who are new to Pyro/NumPyro enumeration mechanism. We recommend readers compare the implementations with the corresponding plate diagrams in [1] to see how concise a Pyro/NumPyro program is.

The interested readers can also refer to [3] for more explanation about enumeration.

The data is taken from Table 1 of reference [2].

Currently, this example does not include postprocessing steps to deal with “Label Switching” issue (mentioned in section 6.2 of [1]).

### References:

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
import jax.numpy as jnp

import numpyro
from numpyro import handlers
from numpyro.contrib.indexing import Vindex
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS
```

(continues on next page)

(continued from previous page)

```

from numpyro.infer.reparam import LocScaleReparam

def get_data():
    """
    :return: a tuple of annotator indices and class indices. The first term has shape
    `num_positions` whose entries take values from `0` to `num_annotators - 1`.
    The second term has shape `num_items x num_positions` whose entries take
    ↪ values
    from `0` to `num_classes - 1`.
    """
    # NB: the first annotator assessed each item 3 times
    positions = np.array([1, 1, 1, 2, 3, 4, 5])
    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, 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,
         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
    # 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))

        with numpyro.plate("position", num_positions):
            numpyro.sample("y", dist.Categorical(zeta[c]), obs=annotations)

```

(continues on next page)

(continued from previous page)

```

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
    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))

        # 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):
        # NB: using constant logits for discrete uniform prior
        # (NumPyro does not have DiscreteUniform distribution yet)
        c = numpyro.sample("c", dist.Categorical(logits=jnp.zeros(num_classes)))

        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)

def hierarchical_dawid_skene(positions, annotations):
    """
    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

```

(continues on next page)

(continued from previous page)

```

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))

    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)))

    with numpyro.plate("item", num_items, dim=-2):
        c = numpyro.sample("c", dist.Categorical(pi))

        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):
    """

```

(continues on next page)

(continued from previous page)

```

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))

    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 = {
    "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),
        args.num_warmup,
        args.num_samples,
        num_chains=args.num_chains,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )

```

(continues on next page)

(continued from previous page)

```
mcmc.run(random.PRNGKey(0), *data)
mcmc.print_summary()

if __name__ == "__main__":
    assert numpyro.__version__.startswith("0.5.0")
    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)
```

---

## Example: Enumerate Hidden Markov Model

---

This example is ported from [1], which shows how to marginalize out discrete model variables in Pyro.

This combines MCMC with a variable elimination algorithm, where we use enumeration to exactly marginalize out some variables from the joint density.

To marginalize out discrete variables  $x$ :

1. Verify that the variable dependency structure in your model admits tractable inference, i.e. the dependency graph among enumerated variables should have narrow treewidth.
2. Ensure your model can handle broadcasting of the sample values of those variables.

Note that difference from [1], which uses Python loop, here we use `scan()` to reduce compilation times (only one step needs to be compiled) of the model. Under the hood, `scan` stacks all the priors' parameters and values into an additional time dimension. This allows us computing the joint density in parallel. In addition, the stacked form allows us to use the parallel-scan algorithm in [2], which reduces parallel complexity from  $O(\text{length})$  to  $O(\log(\text{length}))$ .

Data are taken from [3]. However, the original source of the data seems to be the Institut fuer Algorithmen und Kognitive Systeme at Universitaet Karlsruhe.

### References:

1. *Pyro's Hidden Markov Model example*, (<https://pyro.ai/examples/hmm.html>)
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
```

(continues on next page)

(continued from previous page)

```

from jax import random
import jax.numpy as jnp

import numpyro
from numpyro.contrib.control_flow import scan
from numpyro.contrib.indexing import Vindex
import numpyro.distributions as dist
from numpyro.examples.datasets import JSB_CHORALES, load_dataset
from numpyro.handlers import mask
from numpyro.infer import HMC, MCMC, NUTS

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]))
                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]))
            # 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))

        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))

```

(continues on next page)

(continued from previous page)

```

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]))
            x = numpyro.sample("x", dist.Categorical(probs_x[x_prev]))
            # 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))
        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]))
                x = numpyro.sample("x", dist.Categorical(Vindex(probs_x)[w, x_prev]))
                with numpyro.plate("tones", data_dim, dim=-1) as tones:
                    numpyro.sample("y",

```

(continues on next page)



```

models = {name[len('model_'):] : model
           for name, model in globals().items()
           if name.startswith('model_')}

def main(args):

    model = models[args.model]

    _, fetch = load_dataset(JSB_CHORALES, split='train', shuffle=False)
    lengths, sequences = fetch()
    if args.num_sequences:
        sequences = sequences[0:args.num_sequences]
        lengths = lengths[0:args.num_sequences]

    logger.info('-' * 40)
    logger.info('Training {} on {} sequences'.format(
        model.__name__, len(sequences)))

    # find all the notes that are present at least once in the training set
    present_notes = ((sequences == 1).sum(0).sum(0) > 0)
    # remove notes that are never played (we remove 37/88 notes with default args)
    sequences = sequences[..., present_notes]

    if args.truncate:
        lengths = lengths.clip(0, args.truncate)
        sequences = sequences[:, :args.truncate]

    logger.info('Each sequence has shape {}'.format(sequences[0].shape))
    logger.info('Starting inference...')
    rng_key = random.PRNGKey(2)
    start = time.time()
    kernel = {'nuts': NUTS, 'hmc': HMC}[args.kernel](model)
    mcmc = MCMC(kernel, args.num_warmup, args.num_samples, 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)

```

(continues on next page)

(continued from previous page)

```
numpyro.set_host_device_count(args.num_chains)

main(args)
```



---

## Example: CJS Capture-Recapture Model for Ecological Data

---

This example is ported from [8].

We show how to implement several variants of the Cormack-Jolly-Seber (CJS) [4, 5, 6] model used in ecology to analyze animal capture-recapture data. For a discussion of these models see reference [1].

We make use of two datasets:

- the European Dipper (*Cinclus cinclus*) data from reference [2] (this is Norway's national bird).
- the meadow voles data from reference [3].

Compare to the Stan implementations in [7].

### References

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](http://pyro.ai/examples/capture_recapture.html)

```
import argparse
import os

from jax import random
```

(continues on next page)

(continued from previous page)

```
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
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)))
                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):
        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))
```

(continues on next page)



(continued from previous page)

```

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)))
        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  $\phi_i$  for T-1 of the T time periods of the capture data (just like in model\_2), but here each  $\phi_i$  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))
        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)))
                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

```

(continues on next page)

(continued from previous page)

```
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)))
                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  $\phi$ :  $\text{logit}(\phi_t) = \beta_{\text{group}} + \gamma_t$ . We need to take care that the model is not overparameterized; to do this we effectively let a single scalar  $\beta$  encode the difference in male and female survival probabilities.

```
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):
```

(continues on next page)

(continued from previous page)

```

        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)))
            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))

```

## 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, args.num_warmup, 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 "
                             "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(

```

(continues on next page)

(continued from previous page)

```

        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)

```

---

## 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.

```
[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.

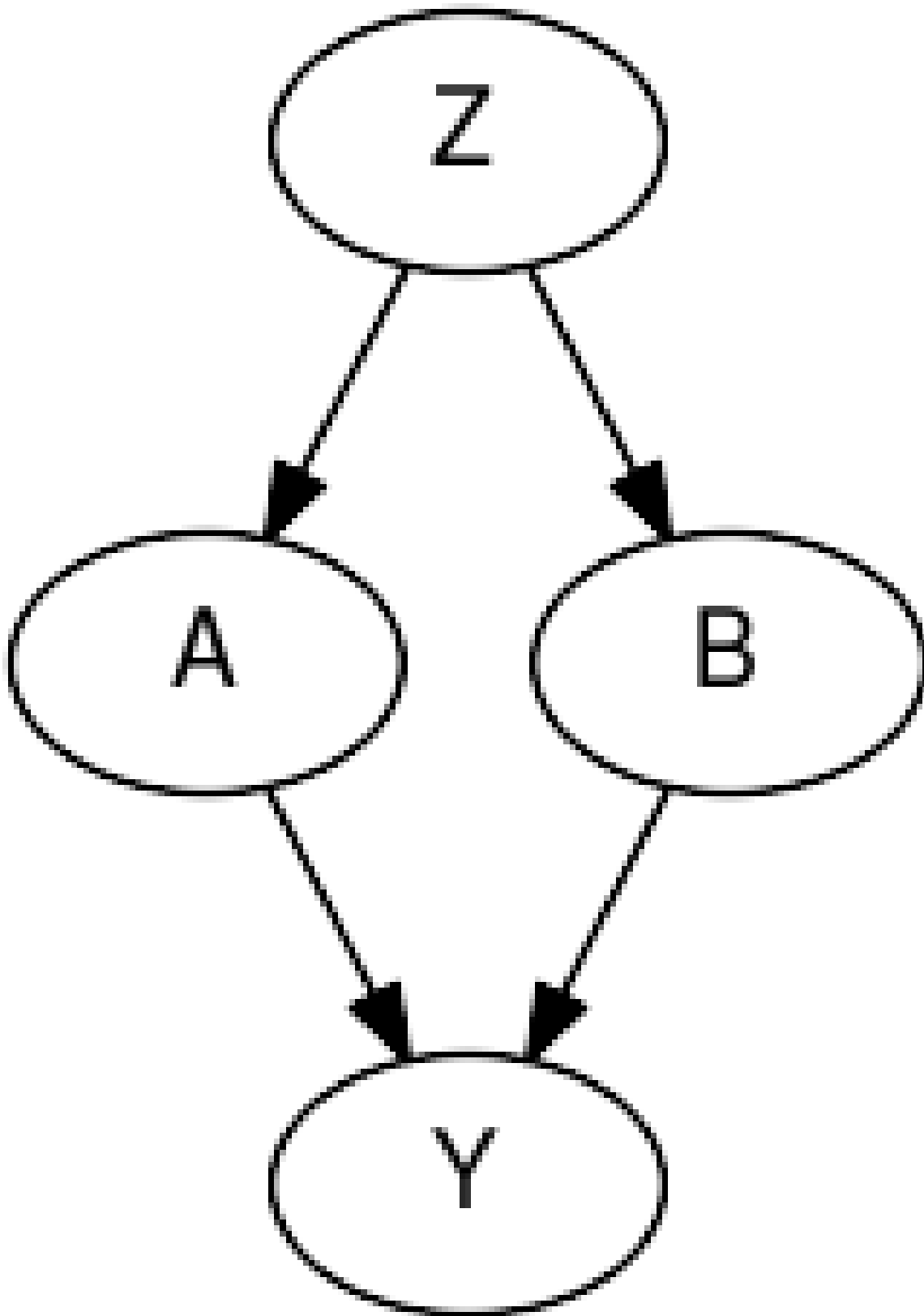
```
[2]: dot = Digraph()
    dot.node('A')
    dot.node('B')
    dot.node('Z')
    dot.node('Y')
```

(continues on next page)

(continued from previous page)

```
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,))
```

## 12.1 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)

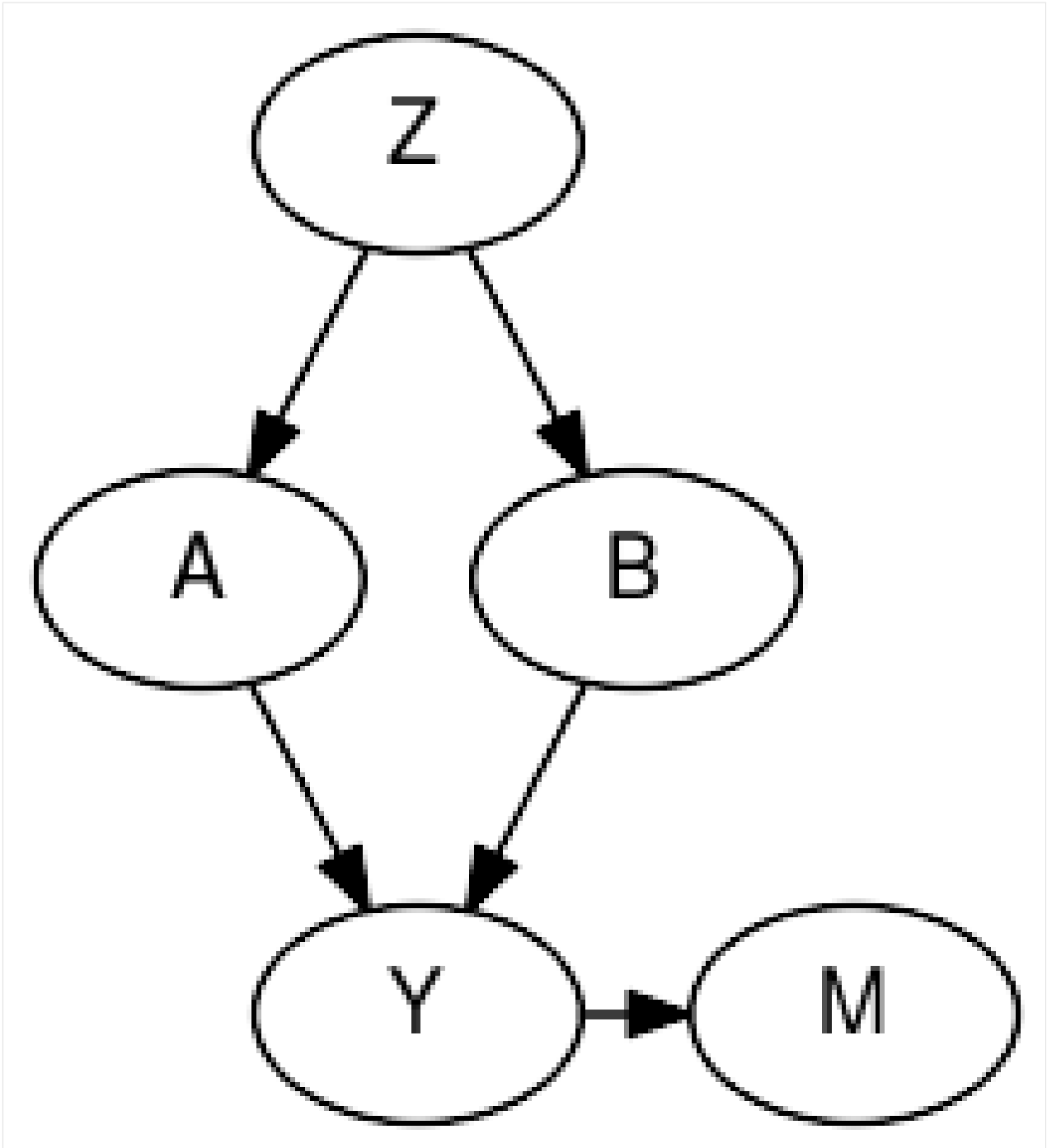
Missing data mechanisms 1. and 2. are ‘easy’ to handle as they depend on observed data only. Mechanism 3. (MNAR) is trickier as it depends on data that is not observed, but may still be relevant to the outcome you are modeling (see below for a concrete example).

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

```
[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 unnecessary. The sampled imputations for A will condition on Y indirectly as the likelihood of Y is conditional on A. So values of A that give high likelihood to Y will be sampled more often than other values.

```
[6]: def ccmodel(A, B, Y):
    ntotal = A.shape[0]
    # get parameters of outcome model
    b_A = sample('b_A', dist.Normal(0, 2.5))
    b_B = sample('b_B', dist.Normal(0, 2.5))
    s_Y = sample('s_Y', dist.HalfCauchy(2.5))

    with numpyro.plate('obs', ntotal):
        ### outcome model
        eta_Y = b_A * A + b_B * B
        sample("obs_Y", dist.Normal(eta_Y, s_Y), obs=Y)
```

```
[7]: cckernel = NUTS(ccmodel)
ccmcmc = MCMC(cckernel, num_warmup=250, num_samples=750)
ccmcmc.run(mcmc_key, Acc, Bcc, Ycc)
ccmcmc.print_summary()
```

```
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

    # 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))
```

(continues on next page)

(continued from previous page)

```

# 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))

    # '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).

## 12.2 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).

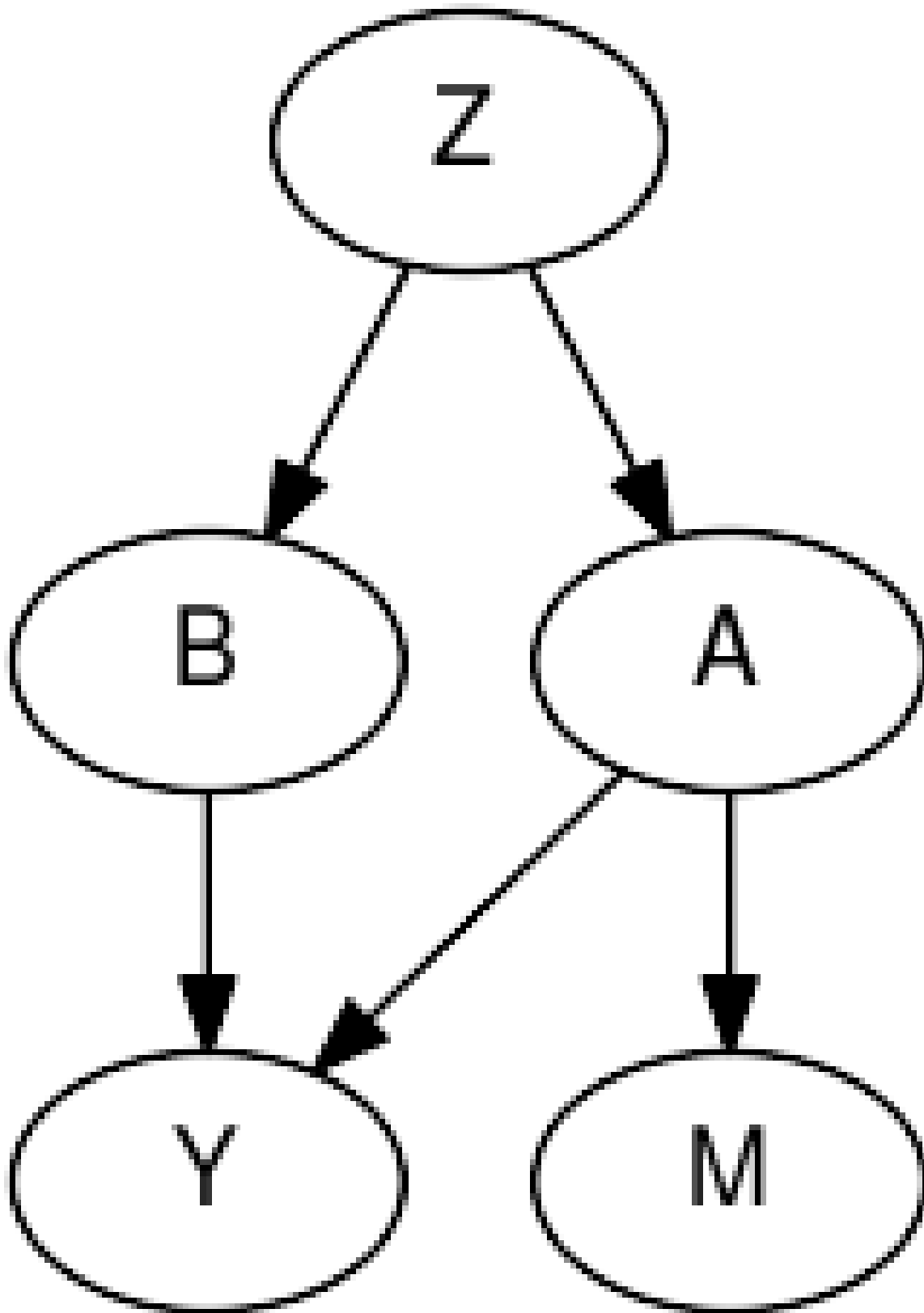
As an example consider patients who have cancer:

- 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
```

(continues on next page)

(continued from previous page)

```

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))

    # '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]: impmissskernel = NUTS(impmisssmodel)
impmisssmcmc = MCMC(impmissskernel, num_warmup=250, num_samples=750)
impmisssmcmc.run(mcmc_key, Aobs, B, Y)
impmisssmcmc.print_summary()

```

```

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

```

	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 \(doi:10.1214/17-STS630\)](#)



---

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 [Rlg: 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.

```
[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")

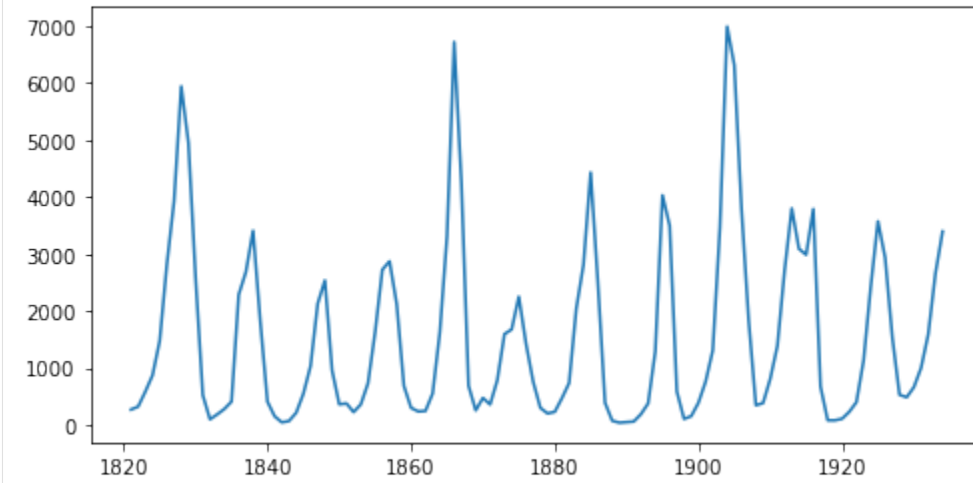
numpyro.set_host_device_count(4)
assert numpyro.__version__.startswith("0.5.0")
```

## 13.1 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))
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:]
```

## 13.2 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:

$$\text{exp-val}_t = \text{level}_{t-1} + \text{coef-trend} \times \text{level}_{t-1}^{\text{pow-trend}} + s_t \times \text{level}_{t-1}^{\text{pow-season}}, \quad (13.1)$$

$$\sigma_t = \sigma \times \text{exp-val}_t^{\text{pow-x}} + \text{offset}, \quad (13.2)$$

$$y_t \sim \text{StudentT}(\nu, \text{exp-val}_t, \sigma_t) \quad (13.3)$$

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

$$\text{level-p} = \begin{cases} y_t - s_t \times \text{level}_{t-1}^{\text{pow-season}} & \text{if } t \leq \text{seasonality}, \\ \text{Average}[y(t - \text{seasonality} + 1), \dots, y(t)] & \text{otherwise,} \end{cases} \quad (13.4)$$

$$\text{level}_t = \text{level-sm} \times \text{level-p} + (1 - \text{level-sm}) \times \text{level}_{t-1}, \quad (13.5)$$

$$s_{t+\text{seasonality}} = s\text{-sm} \times \frac{y_t - \text{level}_t}{\text{level}_{t-1}^{\text{pow-trend}}} + (1 - s\text{-sm}) \times s_t. \quad (13.6)$$

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  $\sigma_t$  will be nearly constant while when `powx` is near 1, the error will be proportional to the expected value.
- There are several varieties of SGT. In this tutorial, we use generalized seasonality and seasonal average method.

We are ready to specify the model using *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.

## 13.3 Inference

First, we want to choose a good value for `seasonality`. Following [the demo in Rlg](#), 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
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

(continues on next page)

(continued from previous page)

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

## 13.4 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"
]
```

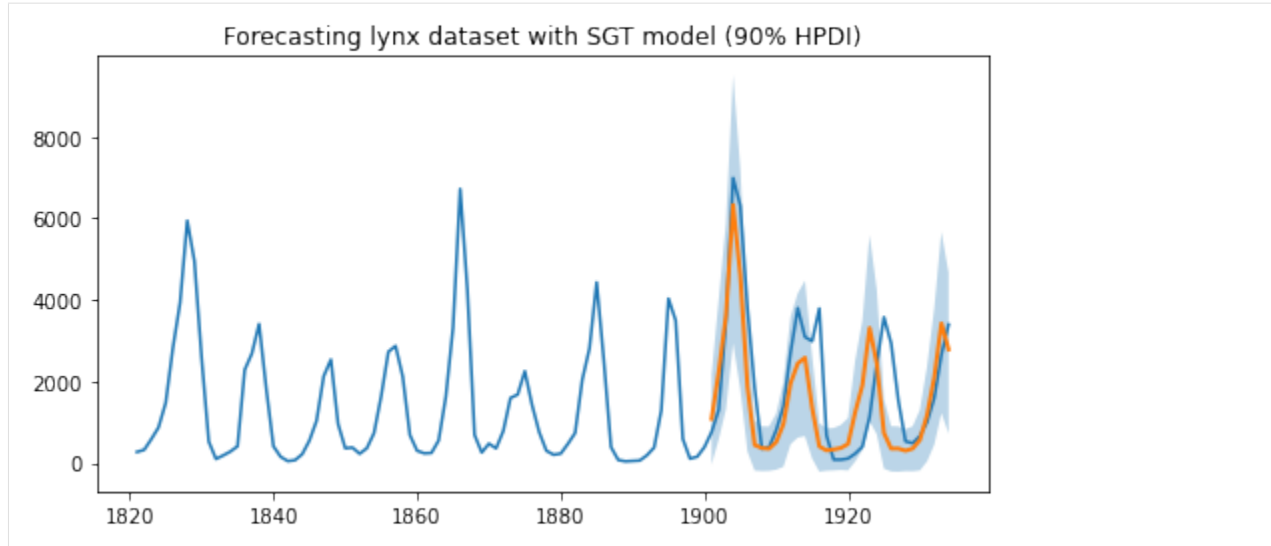
Let's get `sMAPE`, root mean square error of the prediction, and visualize the result with the mean prediction and the 90% highest posterior density interval (HPDI).

```
[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.

## 13.5 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.

## 13.6 References

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





## Ordinal Regression

Some data are discrete but intrinsically **ordered**, these are called **ordinal** data. One example is the [likert scale](#) for questionnaires (“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 approaches for modeling ordered data. Here we will show how to use the `OrderedLogistic` distribution using cutpoints that are sampled from a Normal distribution with as additional constrain that the cutpoints they are ordered. For a more in-depth discussion of Bayesian modeling of ordinal data, see e.g. [Michael Betancour’s blog](#)

```
[1]: from jax import numpy as np, random
import numpyro
from numpyro import sample
from numpyro.distributions import (Categorical, ImproperUniform, Normal,
↳OrderedLogistic,
                                   TransformedDistribution, constraints, transforms)
from numpyro.infer import MCMC, NUTS
import pandas as pd
import seaborn as sns
assert numpyro.__version__.startswith('0.5.0')
```

First, generate some data with ordinal structure

```
[2]: 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:")
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame({'X': X, 'Y': Y})
print(df.Y.value_counts())

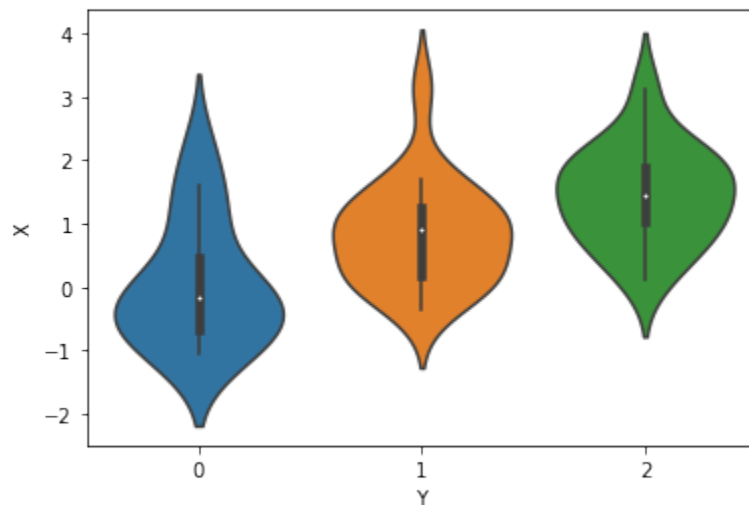
for i in range(nclasses):
    print(f"mean(X) for Y == {i}: {X[np.where(Y==i)].mean():.3f}")
```

value counts of Y:

Y	count
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

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



We will model the outcomes  $Y$  as coming from an `OrderedLogistic` distribution, conditional on  $X$ . The `OrderedLogistic` distribution in `numpyro` requires ordered cutpoints. We can use the `ImproperUniform` distribution to introduce a parameter with an arbitrary support that is otherwise completely uninformative, and then add an `ordered_vector` constraint.

```
[4]: 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 [00:07<00:00, 126.55it/s, 7 steps of\_  
↪size 4.34e-01. acc. prob=0.95]

	mean	std	median	5.0%	95.0%	n_eff	r_hat
b_X_eta	1.43	0.35	1.42	0.82	1.96	352.72	1.00
c_y[0]	-0.11	0.41	-0.11	-0.78	0.55	505.85	1.00
c_y[1]	2.18	0.52	2.15	1.35	2.95	415.22	1.00

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

```
[5]: def model2(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,)))

    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, 315.02it/s, 7 steps of
↳ size 4.80e-01. acc. prob=0.94]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
b_X_eta	1.23	0.30	1.23	0.69	1.65	535.41	1.00
c_y[0]	-0.25	0.33	-0.25	-0.82	0.27	461.96	1.00
c_y[1]	1.76	0.38	1.75	1.10	2.33	588.10	1.00

Number of divergences: 0

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.

```
[6]: 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)
```

(continues on next page)

(continued from previous page)

```
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, 282.20it/s, 7 steps of
↳size 4.84e-01. acc. prob=0.94]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
b_X_eta	1.41	0.34	1.40	0.88	2.00	444.42	1.00
c_y[0]	-0.05	0.36	-0.05	-0.58	0.54	591.60	1.00
c_y[1]	2.08	0.47	2.07	1.37	2.87	429.27	1.00

```
Number of divergences: 0
```

---

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.

```
[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 ops, 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:
```

(continues on next page)

(continued from previous page)

```
set_matplotlib_formats("svg")

assert numpyro.__version__.startswith("0.5.0")
```

## 15.1 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	\
0	1	0	3	
1	2	1	1	
2	3	1	3	
3	4	1	1	
4	5	0	3	

	Name	Sex	Age	SibSp	\
0	Braund, Mr. Owen Harris	male	22.0	1	
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	
2	Heikkinen, Miss. Laina	female	26.0	0	
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	
4	Allen, Mr. William Henry	male	35.0	0	

	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
1	0	PC 17599	71.2833	C85	C
2	0	STON/O2. 3101282	7.9250	NaN	S

(continues on next page)

(continued from previous page)

3	0	113803	53.1000	C123	S
4	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
5        5
3        5
4        4
6         1
Name: Parch, dtype: int64

S      644
C      168
Q       77
Name: Embarked, dtype: int64
```

## 15.2 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
Major.          2
Col.            2
Ms.             1
Don.            1
the             1
Sir.            1
Lady.           1
Capt.          1
Jonkheer.       1
Mme.            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
```

(continues on next page)



(continued from previous page)

```
# compute the age mean for each title
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)])
```

## 15.3 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]
    if bayesian_impute:
```

(continues on next page)

(continued from previous page)

```

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 = ops.index_update(age, age_nanidx, 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 = ops.index_update(age, age_nanidx, 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`.

## 15.4 Sampling

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

```

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

```

```

sample: 100%|████████████████████| 2000/2000 [00:18<00:00, 110.91it/s, 63 steps of_
↪size 6.48e-02. acc. prob=0.94]

```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
a	0.18	0.80	0.20	-1.14	1.50	1078.98	1.00
age_impute[0]	0.23	0.82	0.27	-1.09	1.54	2412.70	1.00
age_impute[1]	-0.09	0.83	-0.10	-1.34	1.39	1956.25	1.00
age_impute[2]	0.36	0.80	0.33	-1.01	1.60	1558.35	1.00
age_impute[3]	0.23	0.90	0.24	-1.23	1.70	2134.81	1.00
age_impute[4]	-0.65	0.89	-0.61	-2.06	0.82	2027.61	1.00
age_impute[5]	0.23	0.87	0.23	-1.26	1.50	1777.25	1.00
age_impute[6]	0.45	0.78	0.45	-0.78	1.66	1428.74	1.00
age_impute[7]	-0.66	0.91	-0.64	-2.01	0.91	2021.78	1.00
age_impute[8]	-0.09	0.87	-0.08	-1.41	1.42	1630.14	1.00
age_impute[9]	0.22	0.89	0.24	-1.42	1.56	1404.25	1.00
age_impute[10]	0.20	0.87	0.20	-1.16	1.66	2489.31	1.00
age_impute[11]	0.17	0.85	0.17	-1.28	1.51	2063.14	1.00
age_impute[12]	-0.66	0.89	-0.62	-2.18	0.72	1632.65	1.00
age_impute[13]	0.19	0.91	0.17	-1.34	1.64	2394.35	1.00
age_impute[14]	-0.02	0.85	-0.01	-1.38	1.33	1809.29	1.00

(continues on next page)

(continued from previous page)

age_impute[15]	0.37	0.84	0.40	-1.04	1.66	1443.69	1.00
age_impute[16]	-1.73	0.25	-1.73	-2.11	-1.30	2062.38	1.00
age_impute[17]	0.22	0.86	0.22	-1.23	1.55	1565.91	1.00
age_impute[18]	0.23	0.90	0.23	-1.28	1.72	1483.35	1.00
age_impute[19]	-0.68	0.86	-0.66	-2.13	0.67	2069.45	1.00
age_impute[20]	0.21	0.89	0.27	-1.39	1.59	1675.83	1.00
age_impute[21]	0.19	0.89	0.22	-1.28	1.56	1984.23	1.00
age_impute[22]	0.19	0.92	0.20	-1.37	1.57	1608.50	1.00
age_impute[23]	-0.14	0.86	-0.13	-1.59	1.26	1890.99	1.00
age_impute[24]	-0.67	0.88	-0.66	-2.14	0.74	1530.94	1.00
age_impute[25]	0.17	0.89	0.19	-1.34	1.55	1767.90	1.00
age_impute[26]	0.19	0.84	0.22	-1.11	1.56	1617.58	1.00
age_impute[27]	-0.70	0.89	-0.68	-2.02	0.89	1761.35	1.00
age_impute[28]	0.60	0.76	0.62	-0.72	1.75	1645.46	1.00
age_impute[29]	0.24	0.85	0.24	-1.13	1.56	2126.36	1.00
age_impute[30]	0.22	0.84	0.23	-1.03	1.69	2796.85	1.00
age_impute[31]	-1.72	0.27	-1.72	-2.15	-1.28	2433.69	1.00
age_impute[32]	0.43	0.86	0.43	-0.93	1.81	1458.56	1.00
age_impute[33]	0.32	0.87	0.33	-1.08	1.70	1583.13	1.00
age_impute[34]	-1.72	0.28	-1.72	-2.19	-1.28	2429.56	1.00
age_impute[35]	-0.43	0.86	-0.42	-1.88	0.90	1451.65	1.00
age_impute[36]	0.30	0.87	0.29	-1.09	1.72	2084.66	1.00
age_impute[37]	0.30	0.83	0.33	-0.98	1.74	1924.58	1.00
age_impute[38]	0.35	0.79	0.33	-0.82	1.73	1980.63	1.00
age_impute[39]	0.19	0.93	0.19	-1.33	1.75	1463.15	1.00
age_impute[40]	-0.65	0.90	-0.67	-2.04	0.85	1548.78	1.00
age_impute[41]	0.20	0.86	0.21	-1.19	1.56	1889.65	1.00
age_impute[42]	0.22	0.88	0.22	-1.39	1.54	1785.76	1.00
age_impute[43]	0.21	0.82	0.23	-1.35	1.36	1663.29	1.00
age_impute[44]	-0.40	0.92	-0.40	-1.96	1.15	1745.40	1.00
age_impute[45]	-0.34	0.86	-0.33	-1.75	1.05	1380.72	1.00
age_impute[46]	-0.30	0.90	-0.30	-1.87	1.10	1237.61	1.00
age_impute[47]	-0.73	0.91	-0.73	-2.13	0.78	1467.82	1.00
age_impute[48]	0.22	0.89	0.22	-1.15	1.84	2169.66	1.00
age_impute[49]	0.43	0.77	0.43	-0.79	1.65	1839.23	1.00
age_impute[50]	0.23	0.86	0.24	-1.30	1.49	1579.39	1.00
age_impute[51]	-0.29	0.88	-0.35	-1.58	1.23	2247.95	1.00
age_impute[52]	0.36	0.82	0.38	-1.13	1.57	1606.92	1.00
age_impute[53]	-0.67	0.94	-0.65	-2.08	0.94	1587.69	1.00
age_impute[54]	0.25	0.90	0.25	-1.07	1.77	2455.98	1.00
age_impute[55]	0.33	0.88	0.33	-0.94	1.88	1593.76	1.00
age_impute[56]	0.39	0.78	0.39	-0.81	1.65	1101.71	1.00
age_impute[57]	-0.01	0.82	-0.03	-1.40	1.25	1532.71	1.00
age_impute[58]	-0.69	0.90	-0.68	-2.09	0.77	1614.68	1.00
age_impute[59]	-0.12	0.88	-0.13	-1.52	1.33	1384.61	1.00
age_impute[60]	-0.62	0.89	-0.61	-2.08	0.77	2116.31	1.00
age_impute[61]	0.22	0.83	0.24	-1.11	1.58	1581.98	1.00
age_impute[62]	-0.59	0.93	-0.60	-2.17	0.88	1528.18	1.00
age_impute[63]	0.20	0.87	0.20	-1.35	1.48	1677.96	1.00
age_impute[64]	-0.69	0.89	-0.67	-2.22	0.70	1877.18	1.00
age_impute[65]	0.41	0.75	0.42	-0.80	1.67	1501.98	1.00
age_impute[66]	0.24	0.96	0.25	-1.25	1.86	2558.45	1.00
age_impute[67]	0.32	0.73	0.34	-0.82	1.55	1678.01	1.00
age_impute[68]	0.34	0.88	0.34	-1.05	1.83	1712.65	1.00
age_impute[69]	0.24	0.92	0.24	-1.15	1.88	1170.47	1.00
age_impute[70]	-0.65	0.89	-0.64	-2.13	0.76	2018.98	1.00
age_impute[71]	-0.67	0.86	-0.69	-2.12	0.64	1625.09	1.00

(continues on next page)

(continued from previous page)

age_impute[72]	0.18	0.84	0.17	-1.14	1.53	1929.52	1.00
age_impute[73]	0.38	0.77	0.37	-0.92	1.63	1696.03	1.00
age_impute[74]	-0.67	0.90	-0.66	-2.14	0.81	1683.08	1.00
age_impute[75]	0.44	0.84	0.40	-0.84	1.88	1215.51	1.00
age_impute[76]	0.20	0.86	0.22	-1.24	1.61	1899.12	1.00
age_impute[77]	0.18	0.86	0.16	-1.38	1.44	1809.40	1.00
age_impute[78]	-0.42	0.87	-0.44	-1.79	1.05	2265.80	1.00
age_impute[79]	0.20	0.84	0.20	-1.27	1.50	2009.37	1.00
age_impute[80]	0.25	0.84	0.23	-1.15	1.64	1561.96	1.00
age_impute[81]	0.26	0.88	0.29	-1.20	1.68	2251.52	1.00
age_impute[82]	0.62	0.83	0.62	-0.70	1.98	1502.80	1.00
age_impute[83]	0.21	0.86	0.20	-1.07	1.63	2044.48	1.00
age_impute[84]	0.20	0.86	0.18	-1.30	1.54	1497.56	1.00
age_impute[85]	0.25	0.88	0.24	-1.23	1.72	2379.66	1.00
age_impute[86]	0.31	0.75	0.32	-1.07	1.43	1543.45	1.00
age_impute[87]	-0.13	0.90	-0.10	-1.61	1.41	2132.30	1.00
age_impute[88]	0.21	0.90	0.25	-1.40	1.53	1824.99	1.00
age_impute[89]	0.23	0.93	0.23	-1.29	1.70	2859.52	1.00
age_impute[90]	0.40	0.82	0.39	-1.00	1.73	1416.46	1.00
age_impute[91]	0.24	0.92	0.24	-1.17	1.84	1843.64	1.00
age_impute[92]	0.20	0.83	0.19	-1.18	1.47	1970.14	1.00
age_impute[93]	0.24	0.89	0.26	-1.24	1.66	1831.49	1.00
age_impute[94]	0.21	0.90	0.17	-1.24	1.69	1975.06	1.00
age_impute[95]	0.21	0.88	0.22	-1.22	1.67	1904.83	1.00
age_impute[96]	0.34	0.90	0.33	-1.05	1.81	2090.87	1.00
age_impute[97]	0.27	0.88	0.24	-1.26	1.61	1768.14	1.00
age_impute[98]	-0.40	0.92	-0.40	-1.83	1.12	1787.86	1.00
age_impute[99]	0.16	0.91	0.14	-1.22	1.70	1518.86	1.00
age_impute[100]	0.24	0.86	0.22	-1.21	1.61	1856.56	1.00
age_impute[101]	0.20	0.88	0.21	-1.14	1.76	1967.24	1.00
age_impute[102]	-0.30	0.89	-0.28	-1.76	1.12	1795.47	1.00
age_impute[103]	0.01	0.86	0.01	-1.37	1.38	1416.70	1.00
age_impute[104]	0.25	0.92	0.26	-1.13	1.87	1643.05	1.00
age_impute[105]	0.25	0.85	0.29	-1.24	1.55	1831.86	1.00
age_impute[106]	0.25	0.87	0.24	-1.17	1.72	2185.41	1.00
age_impute[107]	0.21	0.87	0.21	-1.07	1.70	1945.87	1.00
age_impute[108]	0.34	0.86	0.35	-1.03	1.78	1666.87	1.00
age_impute[109]	0.27	0.88	0.22	-0.95	1.93	1470.86	1.00
age_impute[110]	0.32	0.76	0.35	-0.96	1.41	1737.45	1.00
age_impute[111]	0.22	0.86	0.22	-1.25	1.59	2439.65	1.00
age_impute[112]	-0.03	0.86	-0.03	-1.41	1.44	1737.27	1.00
age_impute[113]	0.22	0.87	0.23	-1.32	1.56	1211.30	1.00
age_impute[114]	0.38	0.79	0.36	-0.90	1.65	1416.21	1.00
age_impute[115]	0.20	0.88	0.19	-1.12	1.78	1437.63	1.00
age_impute[116]	0.25	0.86	0.20	-1.26	1.53	1336.80	1.00
age_impute[117]	-0.35	0.91	-0.36	-1.91	1.10	1737.09	1.00
age_impute[118]	0.23	0.94	0.23	-1.18	1.95	2347.76	1.00
age_impute[119]	-0.63	0.93	-0.66	-2.09	0.94	1681.58	1.00
age_impute[120]	0.60	0.80	0.59	-0.54	2.06	1437.89	1.00
age_impute[121]	0.21	0.85	0.23	-1.06	1.69	2152.47	1.00
age_impute[122]	0.22	0.82	0.20	-1.07	1.65	2296.29	1.00
age_impute[123]	-0.38	0.94	-0.39	-2.01	1.03	1557.61	1.00
age_impute[124]	-0.61	0.92	-0.63	-2.13	0.86	1450.41	1.00
age_impute[125]	0.24	0.93	0.23	-1.16	1.74	2397.23	1.00
age_impute[126]	0.21	0.84	0.21	-1.06	1.72	2248.42	1.00
age_impute[127]	0.36	0.87	0.36	-0.96	1.81	1663.20	1.00
age_impute[128]	0.24	0.90	0.25	-1.34	1.62	1790.40	1.00

(continues on next page)

(continued from previous page)

age_impute[129]	-0.71	0.88	-0.67	-2.07	0.84	2155.52	1.00
age_impute[130]	0.19	0.85	0.17	-1.29	1.50	1766.80	1.00
age_impute[131]	0.25	0.88	0.25	-1.12	1.76	1891.41	1.00
age_impute[132]	0.32	0.88	0.32	-1.21	1.70	2209.15	1.00
age_impute[133]	0.22	0.89	0.20	-1.21	1.66	1656.09	1.00
age_impute[134]	-0.10	0.91	-0.14	-1.51	1.46	2255.33	1.00
age_impute[135]	0.22	0.85	0.23	-1.04	1.66	1534.46	1.00
age_impute[136]	0.18	0.84	0.17	-1.21	1.55	2292.26	1.00
age_impute[137]	-0.69	0.88	-0.68	-2.29	0.63	2473.01	1.00
age_impute[138]	0.19	0.93	0.18	-1.36	1.65	2256.20	1.00
age_impute[139]	0.20	0.85	0.19	-1.16	1.59	1589.73	1.00
age_impute[140]	0.40	0.79	0.41	-0.90	1.66	2200.96	1.00
age_impute[141]	0.24	0.90	0.22	-1.14	1.73	1805.79	1.00
age_impute[142]	-0.32	0.92	-0.32	-1.82	1.19	1755.92	1.00
age_impute[143]	-0.15	0.86	-0.14	-1.58	1.22	1850.64	1.00
age_impute[144]	-0.67	0.94	-0.66	-2.21	0.80	1812.97	1.00
age_impute[145]	-1.75	0.25	-1.75	-2.17	-1.36	1786.83	1.00
age_impute[146]	0.35	0.84	0.34	-1.02	1.66	2006.44	1.00
age_impute[147]	0.26	0.89	0.27	-1.27	1.61	1800.77	1.00
age_impute[148]	-0.67	0.88	-0.65	-2.13	0.68	1832.48	1.00
age_impute[149]	0.29	0.83	0.29	-1.07	1.59	2181.78	1.00
age_impute[150]	0.22	0.87	0.23	-1.20	1.63	1788.63	1.00
age_impute[151]	0.20	0.87	0.17	-1.19	1.62	1561.13	1.00
age_impute[152]	0.01	0.83	-0.01	-1.35	1.38	2966.42	1.00
age_impute[153]	0.19	0.91	0.22	-1.44	1.55	2302.81	1.00
age_impute[154]	1.06	0.96	1.07	-0.53	2.59	1612.33	1.00
age_impute[155]	0.22	0.81	0.22	-1.08	1.51	1460.31	1.00
age_impute[156]	0.27	0.94	0.25	-1.40	1.75	1409.05	1.00
age_impute[157]	0.22	0.92	0.21	-1.26	1.70	1705.55	1.00
age_impute[158]	0.22	0.87	0.22	-1.08	1.72	1561.10	1.00
age_impute[159]	0.21	0.90	0.22	-1.30	1.56	1366.89	1.00
age_impute[160]	0.18	0.84	0.14	-1.15	1.46	1437.73	1.00
age_impute[161]	-0.49	0.92	-0.52	-1.91	1.09	1357.29	1.00
age_impute[162]	0.37	0.84	0.37	-0.98	1.72	1971.90	1.00
age_impute[163]	0.31	0.84	0.28	-1.05	1.71	1541.32	1.00
age_impute[164]	0.22	0.85	0.21	-1.23	1.60	2056.93	1.00
age_impute[165]	0.23	0.88	0.22	-1.12	1.65	2037.07	1.00
age_impute[166]	-0.11	0.87	-0.11	-1.49	1.33	1851.82	1.00
age_impute[167]	0.21	0.89	0.21	-1.10	1.73	1777.15	1.00
age_impute[168]	0.20	0.83	0.21	-1.12	1.56	1793.28	1.00
age_impute[169]	0.02	0.88	0.01	-1.40	1.43	2410.48	1.00
age_impute[170]	0.16	0.88	0.16	-1.29	1.60	2230.68	1.00
age_impute[171]	0.41	0.83	0.40	-0.96	1.75	1846.52	1.00
age_impute[172]	0.24	0.89	0.22	-1.32	1.54	1852.66	1.00
age_impute[173]	-0.44	0.87	-0.45	-1.92	0.92	2089.83	1.00
age_impute[174]	0.22	0.82	0.23	-1.12	1.55	2427.19	1.00
age_impute[175]	0.22	0.96	0.26	-1.45	1.79	2380.77	1.00
age_impute[176]	-0.43	0.89	-0.41	-1.84	1.06	1803.21	1.00
age_mu[0]	0.19	0.04	0.19	0.12	0.27	1509.52	1.00
age_mu[1]	-0.55	0.08	-0.55	-0.67	-0.43	1224.77	1.00
age_mu[2]	0.42	0.08	0.42	0.30	0.56	1212.38	1.00
age_mu[3]	-1.73	0.04	-1.73	-1.79	-1.64	1274.40	1.00
age_mu[4]	0.85	0.19	0.85	0.55	1.16	1387.85	1.00
age_sigma[0]	0.88	0.03	0.88	0.83	0.93	646.28	1.00
age_sigma[1]	0.90	0.05	0.90	0.82	0.98	1120.20	1.00
age_sigma[2]	0.79	0.05	0.79	0.71	0.88	1113.02	1.00
age_sigma[3]	0.26	0.03	0.25	0.20	0.31	1443.42	1.00

(continues on next page)

(continued from previous page)

age_sigma[4]	0.94	0.13	0.92	0.74	1.16	1106.42	1.00
b_Age	-0.44	0.13	-0.44	-0.66	-0.24	832.14	1.00
b_Embarked[0]	-0.30	0.53	-0.30	-1.14	0.59	525.77	1.00
b_Embarked[1]	0.27	0.54	0.28	-0.65	1.08	572.29	1.00
b_Embarked[2]	0.02	0.55	0.02	-1.00	0.78	524.41	1.00
b_Parch[0]	0.44	0.57	0.46	-0.52	1.35	484.44	1.00
b_Parch[1]	0.10	0.57	0.10	-0.76	1.11	494.90	1.00
b_Parch[2]	-0.49	0.56	-0.48	-1.42	0.38	498.59	1.00
b_Pclass[0]	1.16	0.56	1.17	0.32	2.15	475.02	1.00
b_Pclass[1]	0.02	0.55	0.04	-0.87	0.94	496.89	1.00
b_Pclass[2]	-1.23	0.56	-1.20	-2.20	-0.37	477.51	1.00
b_Sex[0]	1.18	0.70	1.16	-0.05	2.24	691.88	1.00
b_Sex[1]	-1.00	0.69	-1.01	-2.12	0.11	802.26	1.00
b_SibSp[0]	0.26	0.63	0.28	-0.78	1.25	779.73	1.00
b_SibSp[1]	-0.19	0.64	-0.18	-1.15	0.91	775.18	1.00
b_Title[0]	-0.96	0.57	-0.96	-1.85	0.02	521.30	1.00
b_Title[1]	-0.34	0.62	-0.33	-1.40	0.62	695.62	1.00
b_Title[2]	0.54	0.63	0.54	-0.42	1.58	672.71	1.00
b_Title[3]	1.46	0.64	1.46	0.32	2.39	708.52	1.00
b_Title[4]	-0.66	0.62	-0.68	-1.61	0.49	635.26	1.00

Number of divergences: 0

To double check that the assumption “age is correlated with title” is reasonable, let’s look at the inferred age by title. Recall that we performed 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,
'Misc.': 42.081936}
```

The inferred result confirms our assumption that Age is correlated with Title:

- those with `Master.` title has pretty small age (in other words, they are children in the ship) comparing to the other groups,
- those with `Mrs.` title have larger age than those with `Miss.` title (in average).

We can also see that the result is similar to the actual statistical mean of Age given Title in our training dataset:

```
[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.

## 15.5 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)
```

## 15.6 References

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



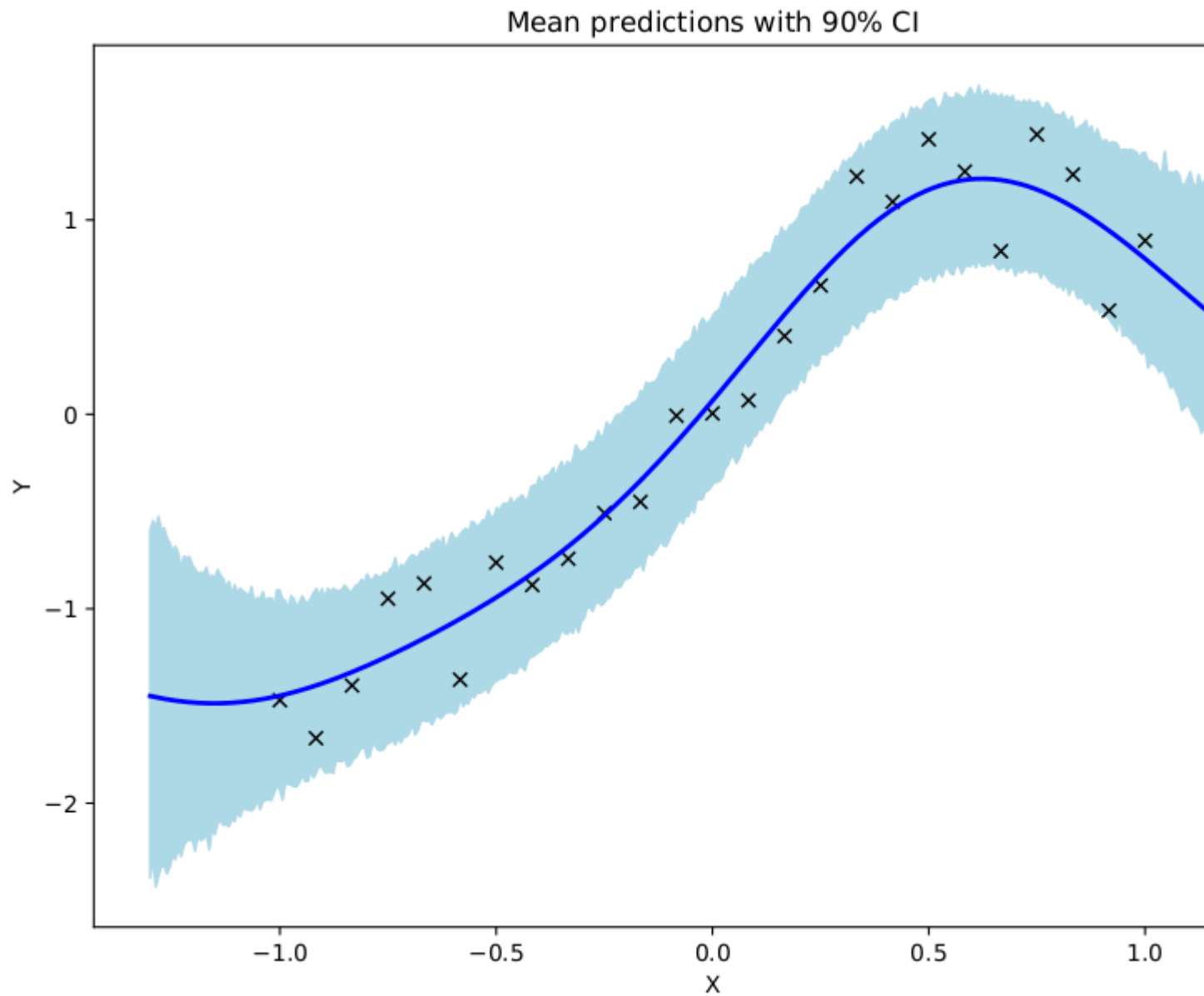
## CHAPTER 16

---

### 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

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
```

(continues on next page)

(continued from previous page)

```

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(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, args.num_warmup, 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):

```

(continues on next page)

(continued from previous page)

```

    # 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.)) * 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)

    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')

```

(continues on next page)

(continued from previous page)

```
# 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('0.5.0')
    parser = argparse.ArgumentParser(description="Gaussian Process example")
    parser.add_argument("-n", "--num-samples", nargs="?", default=1000, type=int)
    parser.add_argument("--num-warmup", nargs='?', default=1000, type=int)
    parser.add_argument("--num-chains", nargs='?', default=1, type=int)
    parser.add_argument("--thinning", nargs='?', default=2, type=int)
    parser.add_argument("--num-data", nargs='?', default=25, type=int)
    parser.add_argument("--device", default='cpu', type=str, help='use "cpu" or "gpu".
→')
    parser.add_argument("--init-strategy", default='median', type=str,
                        choices=['median', 'feasible', 'value', 'uniform', 'sample'])
    args = parser.parse_args()

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

    main(args)
```

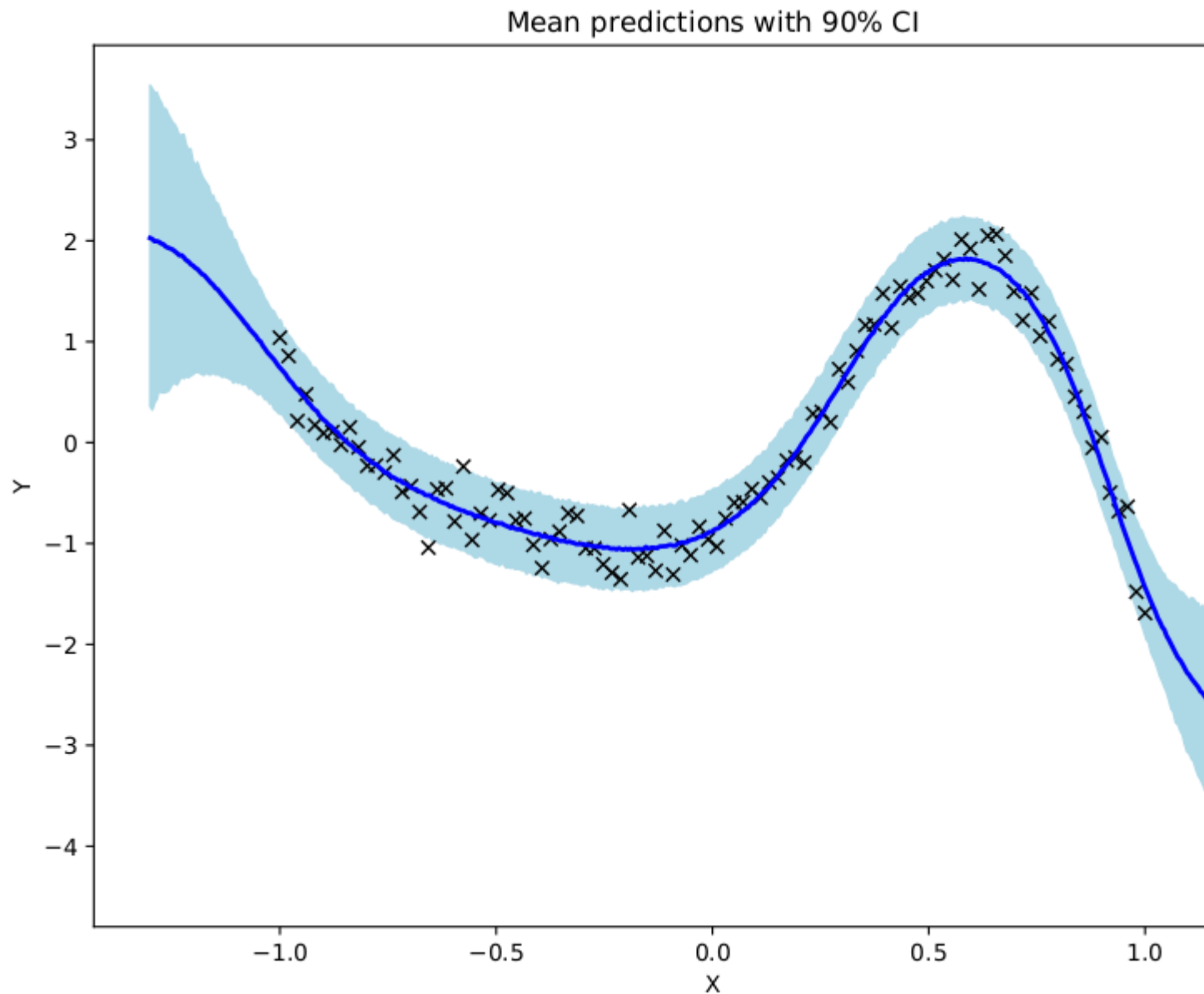


---

### Example: Bayesian Neural Network

---

We demonstrate how to use NUTS to do inference on a simple (small) Bayesian neural network with two hidden layers.



```
import argparse
import os
import time

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

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
```

(continues on next page)



(continued from previous page)

```

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  $D_X \Rightarrow D_H \Rightarrow D_Y$  where  $D_H$  is the number of
# hidden units. (note we indicate tensor dimensions in the comments)
def model(X, Y, D_H):

    D_X, D_Y = X.shape[1], 1

    # 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)))) #  $D_X D_H$ 
    z1 = nonlin(jnp.matmul(X, w1)) #  $N D_H \leftarrow$  first layer of activations

    # sample second layer
    w2 = numpyro.sample("w2", dist.Normal(jnp.zeros((D_H, D_H)), jnp.ones((D_H, D_
↪H)))) #  $D_H D_H$ 
    z2 = nonlin(jnp.matmul(z1, w2)) #  $N D_H \leftarrow$  second layer of activations

    # 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)))) #  $D_H D_Y$ 
    z3 = jnp.matmul(z2, w3) #  $N D_Y \leftarrow$  output of the neural network

    # 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
    numpyro.sample("Y", dist.Normal(z3, sigma_obs), obs=Y)

# 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, args.num_warmup, 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']

```

(continues on next page)

(continued from previous page)

```

# 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)

    # 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')

```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    assert numpyro.__version__.startswith('0.5.0')
    parser = argparse.ArgumentParser(description="Bayesian neural network example")
    parser.add_argument("-n", "--num-samples", nargs="?", default=2000, type=int)
    parser.add_argument("--num-warmup", nargs='?', default=1000, type=int)
    parser.add_argument("--num-chains", nargs='?', default=1, type=int)
    parser.add_argument("--num-data", nargs='?', default=100, type=int)
    parser.add_argument("--num-hidden", nargs='?', default=5, type=int)
    parser.add_argument("--device", default='cpu', type=str, help='use "cpu" or "gpu".
→')
    args = parser.parse_args()

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

    main(args)
```



---

## Example: 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

import jax
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.
```

(continues on next page)

(continued from previous page)

```

def kernel(X, Z, eta1, eta2, c, jitter=1.0e-4):
    etalsq = 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 = (etalsq - eta2sq) * dot(X, Z)
    k4 = jnp.square(c) - 0.5 * eta2sq
    if X.shape == Z.shape:
        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 = jax.ops.index_update(probe, jax.ops.index[:, dimension], 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)

```

(continues on next page)

(continued from previous page)

```

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([0.50, -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 = jax.ops.index_update(probe, jax.ops.index[:, dim1], jnp.array([1.0, 1.0, -
↪1.0, -1.0]))
    probe = jax.ops.index_update(probe, jax.ops.index[:, dim2], 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

```

(continues on next page)

(continued from previous page)

```

num_coefficients = P + M * (M - 1) // 2

probe = jnp.zeros((2 * P + 2 * M * (M - 1), P))
vec = jnp.zeros((num_coefficients, 2 * P + 2 * M * (M - 1)))
start1 = 0
start2 = 0

for dim in range(P):
    probe = jax.ops.index_update(probe, jax.ops.index[start1:start1 + 2, dim],
    ↪jnp.array([1.0, -1.0]))
    vec = jax.ops.index_update(vec, jax.ops.index[start2, start1:start1 + 2], 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 = jax.ops.index_update(probe, jax.ops.index[start1:start1 + 4,
    ↪dim1],
                                jnp.array([1.0, 1.0, -1.0, -1.0]))
        probe = jax.ops.index_update(probe, jax.ops.index[start1:start1 + 4,
    ↪dim2],
                                jnp.array([1.0, -1.0, 1.0, -1.0]))
        vec = jax.ops.index_update(vec, jax.ops.index[start2, start1:start1 + 4],
                                jnp.array([0.25, -0.25, -0.25, 0.25]))
        start1 += 4
        start2 += 1

eta2 = jnp.square(eta1) * jnp.sqrt(xisq) / msq
kappa = jnp.sqrt(msq) * lam / jnp.sqrt(msq + jnp.square(eta1 * lam))

kX = kappa * X
kprobe = kappa * probe

k_xx = kernel(kX, kX, eta1, eta2, c) + sigma ** 2 * jnp.eye(N)
L = cho_factor(k_xx, lower=True)[0]
k_probeX = kernel(kprobe, kX, eta1, eta2, c)
k_prbprb = kernel(kprobe, kprobe, eta1, eta2, c)

mu = jnp.matmul(k_probeX, cho_solve((L, True), Y))
mu = jnp.sum(mu * vec, axis=-1)

Linv_k_probeX = solve_triangular(L, jnp.transpose(k_probeX), lower=True)
covar = k_prbprb - jnp.matmul(jnp.transpose(Linv_k_probeX), Linv_k_probeX)
covar = jnp.matmul(vec, jnp.matmul(covar, jnp.transpose(vec)))

# sample from N(mu, covar)
L = jnp.linalg.cholesky(covar)
sample = mu + jnp.matmul(L, np.random.randn(num_coefficients))

return sample

# Helper function for doing HMC inference
def run_inference(model, args, rng_key, X, Y, hypers):

```

(continues on next page)



(continued from previous page)

```

start = time.time()
kernel = NUTS(model)
mcmc = MCMC(kernel, args.num_warmup, 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 = (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:

```

(continues on next page)

(continued from previous page)

```

        compute_pairwise_mean_variance(X, Y, dim1, dim2, msq, lam,
                                       etal, 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
    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)

```

(continues on next page)

(continued from previous page)

```

    for dim_pair, mean, std in zip(dim_pairs, means, stds):
        dim1, dim2 = dim_pair
        if dim1 >= dim2:
            continue
        lower, upper = mean - 3.0 * std, mean + 3.0 * std
        if not (lower < 0.0 and upper > 0.0):
            format_str = "Identified pairwise interaction between dimensions %d_
↪and %d: %.2e +- %.2e"
            print(format_str % (dim1 + 1, dim2 + 1, mean, std))

        # Draw a single sample of coefficients theta from the posterior, where we_
↪return all singleton
        # coefficients theta_i and pairwise coefficients theta_ij for i, j active_
↪dimensions. We use the
        # final MCMC sample obtained from the HMC sampler.
        thetas = sample_theta_space(X, Y, active_dimensions, samples['msq'][-1],_
↪samples['lambda'][-1],
                                samples['eta1'][-1], samples['xisq'][-1], hypers[
↪'c'], samples['sigma'][-1])
        print("Single posterior sample theta:\n", thetas)

if __name__ == "__main__":
    assert numpyro.__version__.startswith('0.5.0')
    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)

```



---

## 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,))
```

(continues on next page)

(continued from previous page)

```

    made_purchase_no_calls = dist.Bernoulli(0.061).sample(key2, sample_shape=(num_no_
↳calls,))

    made_purchase = jnp.concatenate([made_purchase_got_called, made_purchase_no_
↳calls])

    is_female = dist.Bernoulli(0.5).sample(key3, sample_shape=(num_calls + num_no_
↳calls,))
    got_called = jnp.concatenate([jnp.ones(num_calls), jnp.zeros(num_no_calls)])
    design_matrix = jnp.hstack([jnp.ones((num_no_calls + num_calls, 1)),
                                got_called.reshape(-1, 1),
                                is_female.reshape(-1, 1)])

    return design_matrix, made_purchase

def model(design_matrix: jnp.ndarray, outcome: jnp.ndarray = None) -> None:
    """
    Model definition: Log odds of making a purchase is a linear combination
    of covariates. Specify a Normal prior over regression coefficients.
    :param design_matrix: Covariates. All categorical variables have been one-hot
    encoded.
    :param outcome: Binary response variable. In this case, whether or not the
    customer made a purchase.
    """

    beta = numpyro.sample('coefficients', dist.MultivariateNormal(loc=0.,
                                                                    covariance_
↳matrix=jnp.eye(design_matrix.shape[1])))
    logits = design_matrix.dot(beta)

    with numpyro.plate('data', design_matrix.shape[0]):
        numpyro.sample('obs', dist.Bernoulli(logits=logits), obs=outcome)

def print_results(coef: jnp.ndarray, interval_size: float = 0.95) -> None:
    """
    Print the confidence interval for the effect size with interval_size
    probability mass.
    """

    baseline_response = expit(coef[:, 0])
    response_with_calls = expit(coef[:, 0] + coef[:, 1])

    impact_on_probability = hpdi(response_with_calls - baseline_response,
↳prob=interval_size)

    effect_of_gender = hpdi(coef[:, 2], prob=interval_size)

    print(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 ")

```

(continues on next page)

(continued from previous page)

```

        f"lies in the interval ({effect_of_gender[0]:.2f}, {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_samples, num_chains,
                progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True)
    mcmc.run(rng_key, design_matrix, outcome)

    # 0th column is intercept (not getting called)
    # 1st column is effect of getting called
    # 2nd column is effect of gender (should be none since assigned at random)
    coef = mcmc.get_samples()['coefficients']
    print_results(coef, interval_size)

def main(args):
    rng_key, _ = random.split(random.PRNGKey(3))
    design_matrix, response = make_dataset(rng_key)
    run_inference(design_matrix, response, rng_key,
                  args.num_warmup,
                  args.num_samples,
                  args.num_chains,
                  args.interval_size)

if __name__ == '__main__':
    assert numpyro.__version__.startswith('0.5.0')
    parser = argparse.ArgumentParser(description='Testing whether ')
    parser.add_argument('-n', '--num-samples', nargs='?', default=500, type=int)
    parser.add_argument('--num-warmup', nargs='?', default=1500, type=int)
    parser.add_argument('--num-chains', nargs='?', default=1, type=int)
    parser.add_argument('--interval-size', nargs='?', default=0.95, type=float)
    ↪parser.add_argument('--device', default='cpu', type=str, help='use "cpu" or "gpu".
    ↪')
    args = parser.parse_args()

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

    main(args)

```





---

## Example: Generalized Linear Mixed Models

---

The UCBadmit data is sourced from the study [1] of gender biased in graduate admissions at UC Berkeley in Fall 1973:

Table 1: UCBadmit dataset

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 accross departments. This is a form of Generalized Linear Mixed Models for binomial regression problem, which models

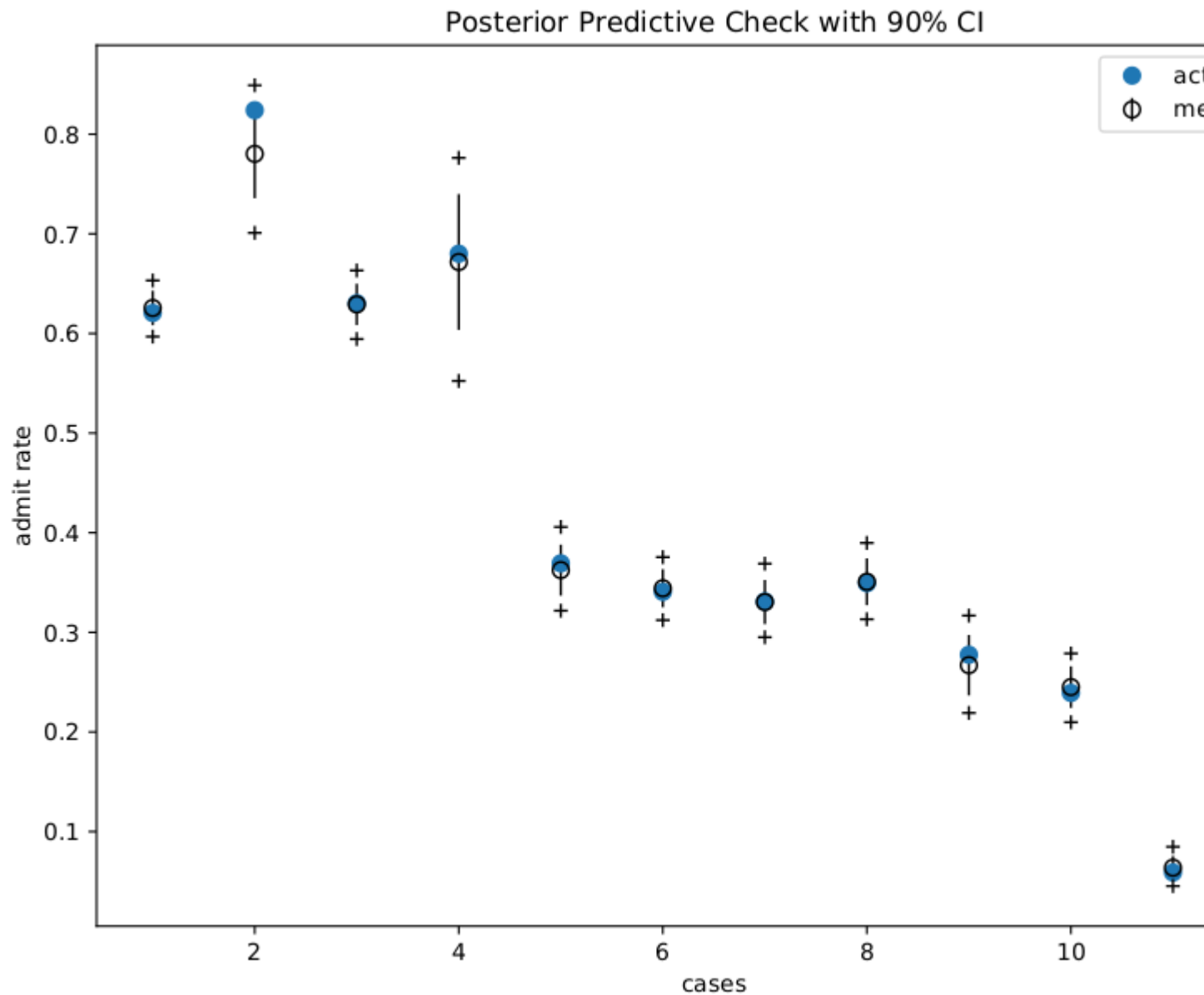
- varying intercepts accross departments,
- varying slopes (or the effects of being male) accross departments,
- correlation between intercepts and slopes,

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
```

(continues on next page)

(continued from previous page)

```

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., 1.])))

    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
    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, args.num_warmup, args.num_samples, 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)

```

(continues on next page)

(continued from previous page)

```

ax.plot(range(1, 13), admit / applications, "o", ms=7, label="actual rate")
ax.errorbar(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('0.5.0')
    parser = argparse.ArgumentParser(description='UCBadmit gender discrimination_
↪using HMC')
    parser.add_argument('-n', '--num-samples', nargs='?', default=2000, type=int)
    parser.add_argument('--num-warmup', nargs='?', default=500, type=int)
    parser.add_argument('--num-chains', nargs='?', default=1, type=int)
    parser.add_argument('--device', default='cpu', type=str, help='use "cpu" or "gpu".
↪')
    args = parser.parse_args()

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

    main(args)

```

---

## Example: Hidden Markov Model

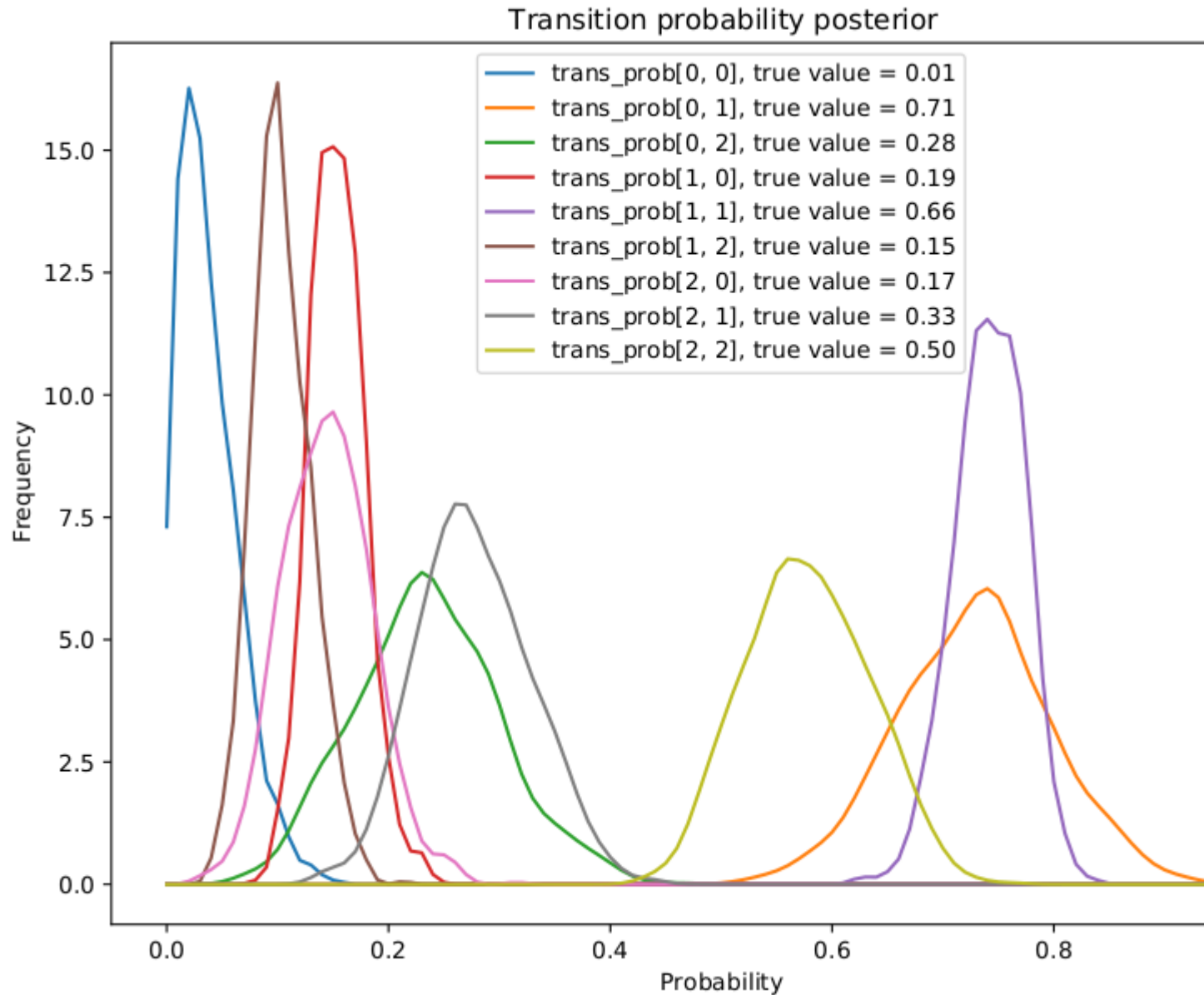
---

In this example, we will follow [1] to construct a semi-supervised Hidden Markov Model for a generative model with observations 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](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](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
```

(continues on next page)

(continued from previous page)

```

def simulate_data(rng_key, num_categories, num_words, num_supervised_data, num_
↳unsupervised_data):
    rng_key, rng_key_transition, rng_key_emission = random.split(rng_key, 3)

    transition_prior = jnp.ones(num_categories)
    emission_prior = jnp.repeat(0.1, num_words)

    transition_prob = dist.Dirichlet(transition_prior).sample(key=rng_key_transition,
                                                             sample_shape=(num_
↳categories,))
    emission_prob = dist.Dirichlet(emission_prior).sample(key=rng_key_emission,
                                                         sample_shape=(num_
↳categories,))

    start_prob = jnp.repeat(1. / num_categories, num_categories)
    categories, words = [], []
    for t in range(num_supervised_data + num_unsupervised_data):
        rng_key, rng_key_transition, rng_key_emission = random.split(rng_key, 3)
        if t == 0 or t == num_supervised_data:
            category = dist.Categorical(start_prob).sample(key=rng_key_transition)
        else:
            category = dist.Categorical(transition_prob[category]).sample(key=rng_key_
↳transition)
        word = dist.Categorical(emission_prob[category]).sample(key=rng_key_emission)
        categories.append(category)
        words.append(word)

    # split into supervised data and unsupervised data
    categories, words = jnp.stack(categories), jnp.stack(words)
    supervised_categories = categories[:num_supervised_data]
    supervised_words = words[:num_supervised_data]
    unsupervised_words = words[num_supervised_data:]
    return (transition_prior, emission_prior, transition_prob, emission_prob,
            supervised_categories, supervised_words, unsupervised_words)

def forward_one_step(prev_log_prob, curr_word, transition_log_prob, emission_log_
↳prob):
    log_prob_tmp = jnp.expand_dims(prev_log_prob, axis=1) + transition_log_prob
    log_prob = log_prob_tmp + emission_log_prob[:, curr_word]
    return logsumexp(log_prob, axis=0)

def forward_log_prob(init_log_prob, words, transition_log_prob, emission_log_prob,
↳unroll_loop=False):
    # Note: The following naive implementation will make it very slow to compile
    # and do inference. So we use lax.scan instead.
    #
    # >>> log_prob = init_log_prob
    # >>> for word in words:
    # ...     log_prob = forward_one_step(log_prob, word, transition_log_prob,
↳emission_log_prob)
    def scan_fn(log_prob, word):
        return forward_one_step(log_prob, word, transition_log_prob, emission_log_
↳prob), jnp.zeros((0,))

```

(continues on next page)

(continued from previous page)

```

    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]),
                   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)

```

(continues on next page)



(continued from previous page)

```

    for i in range(emission_prob.shape[0]):
        for j in range(emission_prob.shape[1]):
            idx = 'emission[{}, {}]'.format(i, j)
            print(row_format.format(idx, emission_prob[i, j], *quantiles[:, i, j]),
                  '\n')

def main(args):
    print('Simulating data...')
    (transition_prior, emission_prior, transition_prob, emission_prob,
     supervised_categories, supervised_words, unsupervised_words) = simulate_data(
        random.PRNGKey(1),
        num_categories=args.num_categories,
        num_words=args.num_words,
        num_supervised_data=args.num_supervised,
        num_unsupervised_data=args.num_unsupervised,
    )
    print('Starting inference...')
    rng_key = random.PRNGKey(2)
    start = time.time()
    kernel = NUTS(semi_supervised_hmm)
    mcmc = MCMC(kernel, args.num_warmup, 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()

    plt.savefig("hmm_plot.pdf")

if __name__ == '__main__':
    assert numpyro.__version__.startswith('0.5.0')
    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')

```

(continues on next page)

(continued from previous page)

```
parser.add_argument('--device', default='cpu', type=str, help='use "cpu" or "gpu".  
→')  
args = parser.parse_args()  
  
numpyro.set_platform(args.device)  
numpyro.set_host_device_count(args.num_chains)  
  
main(args)
```

---

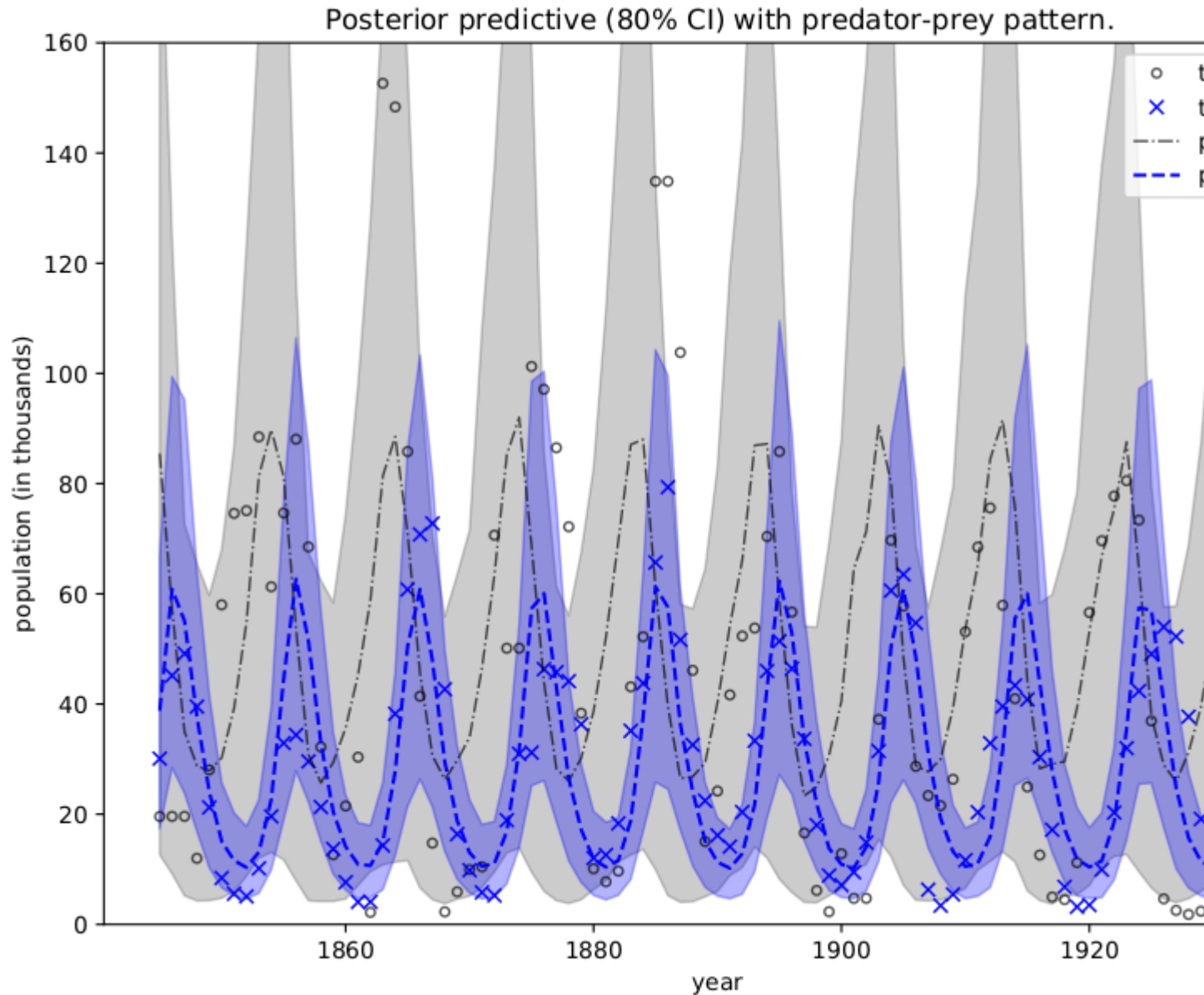
### Example: Predator-Prey Model

---

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](https://en.wikipedia.org/wiki/Lotka-Volterra_equations)
3. <http://people.whitman.edu/~hundlejr/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
```

(continues on next page)

(continued from previous page)

```

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., loc=jnp.array([1.0, 0.05, 1.0, 0.05]),
                           scale=jnp.array([0.5, 0.05, 0.5, 0.05])))
    # 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),
                args.num_warmup, 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, pi = jnp.mean(pop_pred, 0), jnp.percentile(pop_pred, (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)

```

(continues on next page)

(continued from previous page)

```
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('0.5.0')
    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)
    numpyro.set_host_device_count(args.num_chains)

    main(args)
```

---

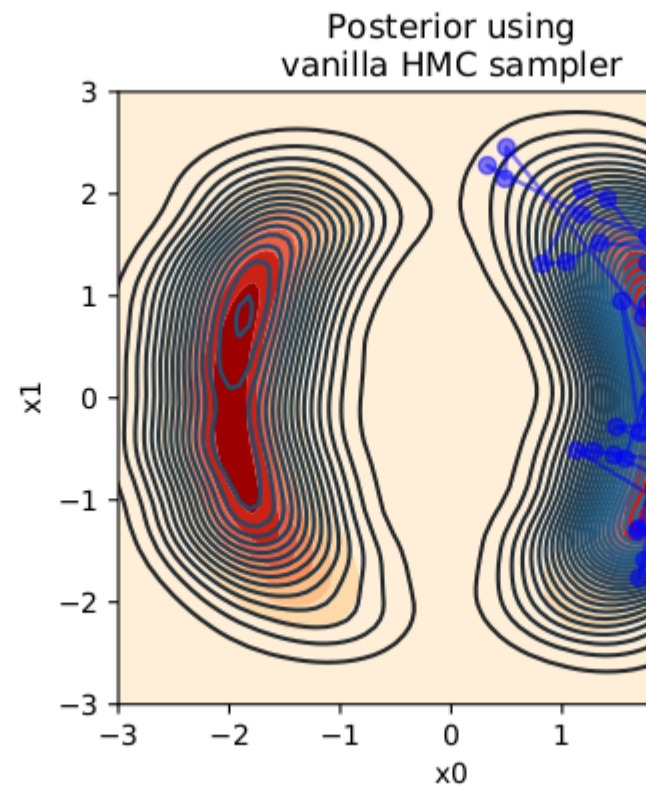
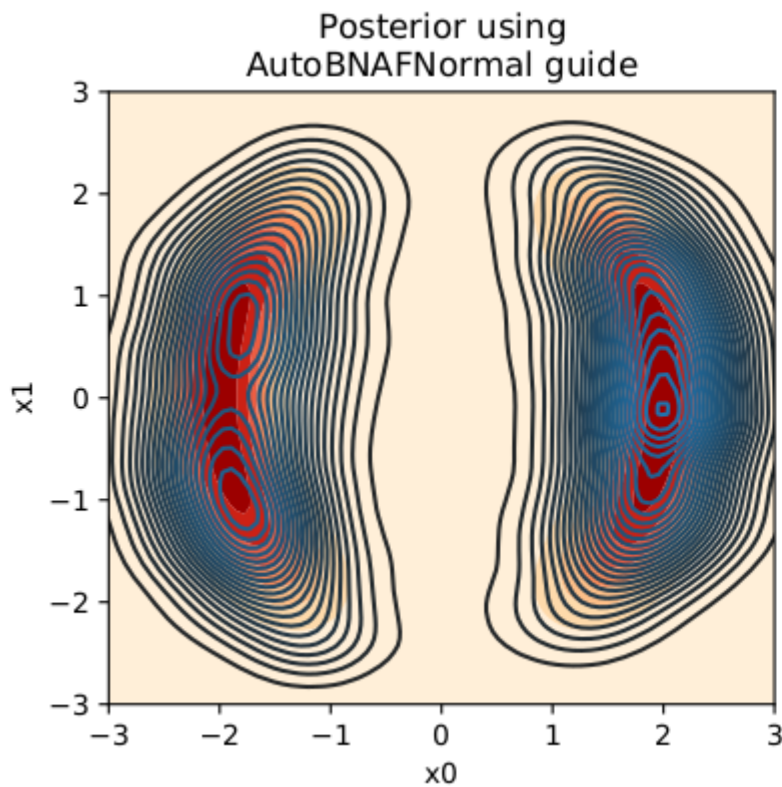
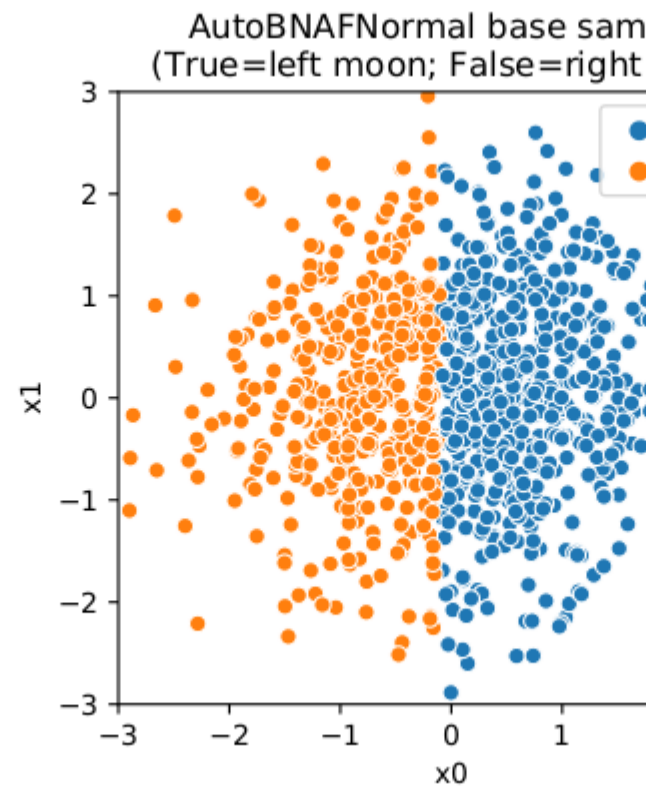
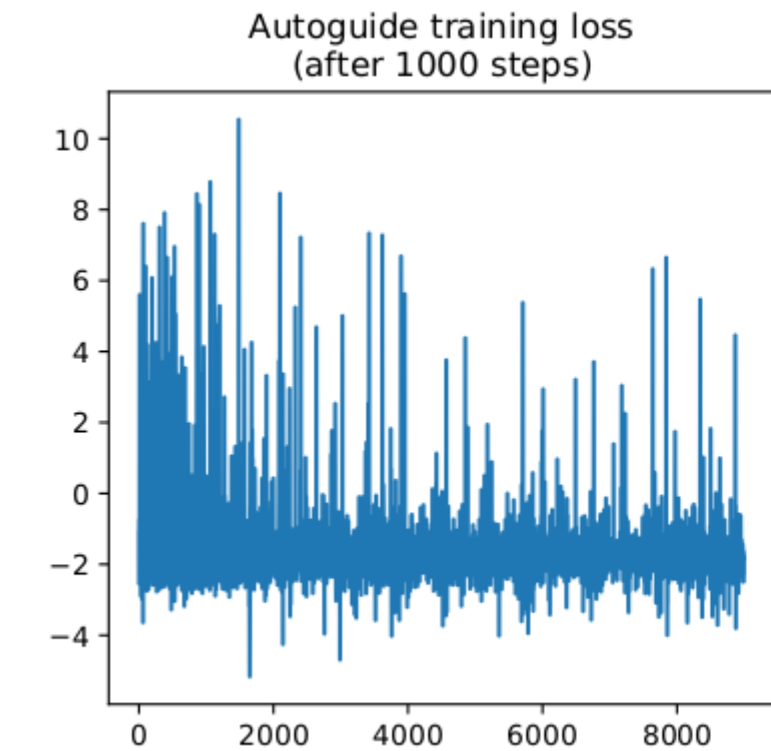
### Example: Neural Transport

---

This example illustrates how to use a trained AutoBNFNormal 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
```

(continues on next page)



(continued from previous page)

```

from matplotlib.gridspec import GridSpec
import matplotlib.pyplot as plt
import seaborn as sns

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., 2.])) / 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, args.num_warmup, 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])
    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...")

```

(continues on next page)

(continued from previous page)

```

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, args.num_warmup, 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.).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:])
ax1.set_title('Autoguide training loss\n(after 1000 steps)')

ax2.contourf(X1, X2, P, cmap='OrRd')
sns.kdeplot(guide_samples[:, 0], 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(guide_base_samples[:, 0], guide_base_samples[:, 1], ax=ax3,
                hue=guide_trans_samples[:, 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(vanilla_samples[:, 0], vanilla_samples[:, 1], n_levels=30, ax=ax4)

```

(continues on next page)

(continued from previous page)

```

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(zs[:, 0], zs[:, 1], ax=ax5, hue=samples[:, 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)
↪')

ax6.contourf(X1, X2, P, cmap='OrRd')
sns.kdeplot(samples[:, 0], 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('0.5.0')
    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 24

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### n

- `numpyro.contrib.functor`, [119](#)
- `numpyro.contrib.indexing`, [136](#)
- `numpyro.contrib.tfp.distributions`, [60](#)
- `numpyro.contrib.tfp.mcmc`, [100](#)
- `numpyro.diagnostics`, [129](#)
- `numpyro.handlers`, [19](#)
- `numpyro.infer.autoguide`, [109](#)
- `numpyro.infer.reparam`, [117](#)
- `numpyro.infer.util`, [132](#)
- `numpyro.optim`, [122](#)
- `numpyro.primitives`, [11](#)
- `numpyro.util`, [131](#)





## Symbols

`__call__()` (*LocScaleReparam* method), 118  
`__call__()` (*NeuTraReparam* method), 118  
`__call__()` (*TransformReparam* method), 119

## A

`AbsTransform` (class in `numpyro.distributions.transforms`), 79  
`Adagrad` (class in `numpyro.optim`), 123  
`Adam` (class in `numpyro.optim`), 122  
`AffineTransform` (class in `numpyro.distributions.transforms`), 79  
`arg_constraints` (*BernoulliLogits* attribute), 50  
`arg_constraints` (*BernoulliProbs* attribute), 50  
`arg_constraints` (*Beta* attribute), 37  
`arg_constraints` (*BetaBinomial* attribute), 51  
`arg_constraints` (*BinomialLogits* attribute), 52  
`arg_constraints` (*BinomialProbs* attribute), 52  
`arg_constraints` (*CategoricalLogits* attribute), 53  
`arg_constraints` (*CategoricalProbs* attribute), 54  
`arg_constraints` (*Cauchy* attribute), 37  
`arg_constraints` (*Chi2* attribute), 38  
`arg_constraints` (*Delta* attribute), 36  
`arg_constraints` (*Dirichlet* attribute), 38  
`arg_constraints` (*DirichletMultinomial* attribute), 55  
`arg_constraints` (*Distribution* attribute), 28  
`arg_constraints` (*ExpandedDistribution* attribute), 30  
`arg_constraints` (*Exponential* attribute), 39  
`arg_constraints` (*Gamma* attribute), 39  
`arg_constraints` (*GammaPoisson* attribute), 55  
`arg_constraints` (*GaussianRandomWalk* attribute), 40  
`arg_constraints` (*GeometricLogits* attribute), 56  
`arg_constraints` (*GeometricProbs* attribute), 56  
`arg_constraints` (*Gumbel* attribute), 40  
`arg_constraints` (*HalfCauchy* attribute), 41  
`arg_constraints` (*HalfNormal* attribute), 41

`arg_constraints` (*ImproperUniform* attribute), 32  
`arg_constraints` (*Independent* attribute), 32  
`arg_constraints` (*InverseGamma* attribute), 42  
`arg_constraints` (*Laplace* attribute), 42  
`arg_constraints` (*LKJ* attribute), 43  
`arg_constraints` (*LKJCholesky* attribute), 44  
`arg_constraints` (*Logistic* attribute), 45  
`arg_constraints` (*LogNormal* attribute), 44  
`arg_constraints` (*LowRankMultivariateNormal* attribute), 46  
`arg_constraints` (*MaskedDistribution* attribute), 34  
`arg_constraints` (*MultinomialLogits* attribute), 57  
`arg_constraints` (*MultinomialProbs* attribute), 58  
`arg_constraints` (*MultivariateNormal* attribute), 45  
`arg_constraints` (*Normal* attribute), 46  
`arg_constraints` (*OrderedLogistic* attribute), 58  
`arg_constraints` (*Pareto* attribute), 47  
`arg_constraints` (*Poisson* attribute), 58  
`arg_constraints` (*StudentT* attribute), 47  
`arg_constraints` (*TransformedDistribution* attribute), 35  
`arg_constraints` (*TruncatedCauchy* attribute), 48  
`arg_constraints` (*TruncatedNormal* attribute), 48  
`arg_constraints` (*TruncatedPolyaGamma* attribute), 49  
`arg_constraints` (*Uniform* attribute), 49  
`arg_constraints` (*Unit* attribute), 36  
`arg_constraints` (*VonMises* attribute), 60  
`arg_constraints` (*ZeroInflatedPoisson* attribute), 60  
`AutoBNFNormal` (class in `numpyro.infer.autoguide`), 110  
`AutoContinuous` (class in `numpyro.infer.autoguide`), 109  
`autocorrelation()` (in module `numpyro.diagnostics`), 129  
`autocovariance()` (in module `numpyro.diagnostics`), 129  
`AutoDelta` (class in `numpyro.infer.autoguide`), 116  
`AutoDiagonalNormal` (class in `numpyro.infer.autoguide`), 111

AutoIAFNormal (class in `numpyro.infer.autoguide`), 113  
 AutoLaplaceApproximation (class in `numpyro.infer.autoguide`), 113  
 AutoLowRankMultivariateNormal (class in `numpyro.infer.autoguide`), 114  
 AutoMultivariateNormal (class in `numpyro.infer.autoguide`), 112  
 AutoNormal (class in `numpyro.infer.autoguide`), 115  
 Autoregressive (class in `numpyro.contrib.tfp.distributions`), 61

## B

batch\_shape (Distribution attribute), 28  
 BatchReshape (class in `numpyro.contrib.tfp.distributions`), 61  
 Bates (class in `numpyro.contrib.tfp.distributions`), 61  
 Bernoulli (class in `numpyro.contrib.tfp.distributions`), 61  
 Bernoulli() (in module `numpyro.distributions.discrete`), 49  
 BernoulliLogits (class in `numpyro.distributions.discrete`), 50  
 BernoulliProbs (class in `numpyro.distributions.discrete`), 50  
 Beta (class in `numpyro.contrib.tfp.distributions`), 61  
 Beta (class in `numpyro.distributions.continuous`), 37  
 BetaBinomial (class in `numpyro.contrib.tfp.distributions`), 62  
 BetaBinomial (class in `numpyro.distributions.conjugate`), 51  
 biject\_to() (in module `numpyro.distributions.transforms`), 78  
 BijectorConstraint (class in `numpyro.contrib.tfp.distributions`), 61  
 BijectorTransform (class in `numpyro.contrib.tfp.distributions`), 61  
 Binomial (class in `numpyro.contrib.tfp.distributions`), 62  
 Binomial() (in module `numpyro.distributions.discrete`), 52  
 BinomialLogits (class in `numpyro.distributions.discrete`), 52  
 BinomialProbs (class in `numpyro.distributions.discrete`), 52  
 block (class in `numpyro.handlers`), 20  
 BlockNeuralAutoregressiveTransform (class in `numpyro.distributions.flows`), 83  
 Blockwise (class in `numpyro.contrib.tfp.distributions`), 62  
 boolean (in module `numpyro.distributions.constraints`), 76

## C

call\_with\_intermediates() (BlockNeuralAutoregressiveTransform method), 83  
 call\_with\_intermediates() (ComposeTransform method), 79  
 call\_with\_intermediates() (InverseAutoregressiveTransform method), 83  
 call\_with\_intermediates() (Transform method), 78  
 Categorical (class in `numpyro.contrib.tfp.distributions`), 62  
 Categorical() (in module `numpyro.distributions.discrete`), 53  
 CategoricalLogits (class in `numpyro.distributions.discrete`), 53  
 CategoricalProbs (class in `numpyro.distributions.discrete`), 54  
 Cauchy (class in `numpyro.contrib.tfp.distributions`), 62  
 Cauchy (class in `numpyro.distributions.continuous`), 37  
 check() (Constraint method), 76  
 Chi (class in `numpyro.contrib.tfp.distributions`), 62  
 Chi2 (class in `numpyro.contrib.tfp.distributions`), 62  
 Chi2 (class in `numpyro.distributions.continuous`), 38  
 CholeskyLKJ (class in `numpyro.contrib.tfp.distributions`), 62  
 ClippedAdam (class in `numpyro.optim`), 124  
 codomain (AbsTransform attribute), 79  
 codomain (AffineTransform attribute), 79  
 codomain (BlockNeuralAutoregressiveTransform attribute), 83  
 codomain (ComposeTransform attribute), 79  
 codomain (CorrCholeskyTransform attribute), 80  
 codomain (ExpTransform attribute), 80  
 codomain (InvCholeskyTransform attribute), 80  
 codomain (InverseAutoregressiveTransform attribute), 83  
 codomain (LowerCholeskyAffine attribute), 81  
 codomain (LowerCholeskyTransform attribute), 81  
 codomain (OrderedTransform attribute), 81  
 codomain (PermuteTransform attribute), 82  
 codomain (PowerTransform attribute), 82  
 codomain (SigmoidTransform attribute), 82  
 codomain (StickBreakingTransform attribute), 82  
 codomain (Transform attribute), 78  
 collapse (class in `numpyro.handlers`), 21  
 ComposeTransform (class in `numpyro.distributions.transforms`), 79  
 condition (class in `numpyro.handlers`), 21  
 config\_enumerate() (in module `numpyro.contrib.functor.infer_util`), 121  
 consensus() (in module `numpyro.infer.hmc_util`), 103  
 constrain\_fn() (in module `numpyro.infer.util`), 134  
 Constraint (class in `numpyro.distributions.constraints`), 76

ContinuousBernoulli (class in *numpyro.contrib.tfp.distributions*), 63  
 corr\_cholesky (in module *numpyro.distributions.constraints*), 76  
 corr\_matrix (in module *numpyro.distributions.constraints*), 76  
 CorrCholeskyTransform (class in *numpyro.distributions.transforms*), 79  
 covariance\_matrix (*LowRankMultivariateNormal* attribute), 46  
 covariance\_matrix (*MultivariateNormal* attribute), 46

## D

default\_fields (*HMC* attribute), 89  
 default\_fields (*MCMCKernel* attribute), 88  
 default\_fields (*SA* attribute), 96  
 Delta (class in *numpyro.distributions.distribution*), 36  
 dependent (in module *numpyro.distributions.constraints*), 76  
 DeterminantalPointProcess (class in *numpyro.contrib.tfp.distributions*), 63  
 Deterministic (class in *numpyro.contrib.tfp.distributions*), 63  
 deterministic() (in module *numpyro.primitives*), 13  
 Dirichlet (class in *numpyro.contrib.tfp.distributions*), 63  
 Dirichlet (class in *numpyro.distributions.continuous*), 38  
 DirichletMultinomial (class in *numpyro.contrib.tfp.distributions*), 63  
 DirichletMultinomial (class in *numpyro.distributions.conjugate*), 54  
 DiscreteHMC Gibbs (class in *numpyro.infer.hmc\_gibbs*), 92  
 Distribution (class in *numpyro.distributions.distribution*), 27  
 do (class in *numpyro.handlers*), 21  
 domain (*AbsTransform* attribute), 79  
 domain (*BlockNeuralAutoregressiveTransform* attribute), 83  
 domain (*ComposeTransform* attribute), 79  
 domain (*CorrCholeskyTransform* attribute), 80  
 domain (*InverseAutoregressiveTransform* attribute), 83  
 domain (*LowerCholeskyAffine* attribute), 81  
 domain (*LowerCholeskyTransform* attribute), 81  
 domain (*OrderedTransform* attribute), 81  
 domain (*PermuteTransform* attribute), 82  
 domain (*PowerTransform* attribute), 82  
 domain (*StickBreakingTransform* attribute), 82  
 domain (*Transform* attribute), 78  
 DoublesidedMaxwell (class in *numpyro.contrib.tfp.distributions*), 63

## E

effective\_sample\_size() (in module *numpyro.diagnostics*), 130  
 ELBO (class in *numpyro.infer.elbo*), 106  
 Empirical (class in *numpyro.contrib.tfp.distributions*), 63  
 enable\_validation() (in module *numpyro.distributions.distribution*), 131  
 enable\_x64() (in module *numpyro.util*), 132  
 entropy() (*LowRankMultivariateNormal* method), 46  
 enum (class in *numpyro.contrib.functor.enum\_messenger*), 119  
 enumerate\_support() (*BernoulliLogits* method), 50  
 enumerate\_support() (*BernoulliProbs* method), 51  
 enumerate\_support() (*BetaBinomial* method), 51  
 enumerate\_support() (*BinomialLogits* method), 52  
 enumerate\_support() (*BinomialProbs* method), 53  
 enumerate\_support() (*CategoricalLogits* method), 54  
 enumerate\_support() (*CategoricalProbs* method), 54  
 enumerate\_support() (*Distribution* method), 29  
 enumerate\_support() (*ExpandedDistribution* method), 31  
 enumerate\_support() (*MaskedDistribution* method), 34  
 eval\_and\_update() (*Adagrad* method), 123  
 eval\_and\_update() (*Adam* method), 122  
 eval\_and\_update() (*ClippedAdam* method), 124  
 eval\_and\_update() (*Minimize* method), 125  
 eval\_and\_update() (*Momentum* method), 126  
 eval\_and\_update() (*RMSProp* method), 126  
 eval\_and\_update() (*RMSPropMomentum* method), 127  
 eval\_and\_update() (*SGD* method), 128  
 eval\_and\_update() (*SM3* method), 128  
 evaluate() (*SVI* method), 106  
 event\_dim (*Constraint* attribute), 76  
 event\_dim (*Distribution* attribute), 28  
 event\_dim (*Transform* attribute), 78  
 event\_shape (*Distribution* attribute), 28  
 expand() (*Distribution* method), 29  
 expand() (*Independent* method), 33  
 expand\_by() (*Distribution* method), 29  
 ExpandedDistribution (class in *numpyro.distributions.distribution*), 30  
 ExpGamma (class in *numpyro.contrib.tfp.distributions*), 64  
 ExpInverseGamma (class in *numpyro.contrib.tfp.distributions*), 64

- Exponential (class in `numpyro.contrib.tfp.distributions`), 64
- Exponential (class in `numpyro.distributions.continuous`), 39
- ExponentiallyModifiedGaussian (class in `numpyro.contrib.tfp.distributions`), 64
- ExpRelaxedOneHotCategorical (class in `numpyro.contrib.tfp.distributions`), 64
- ExpTransform (class in `numpyro.distributions.transforms`), 80
- ## F
- `factor()` (in module `numpyro.primitives`), 14
- `feasible_like()` (Constraint method), 76
- `find_valid_initial_params()` (in module `numpyro.infer.util`), 135
- FiniteDiscrete (class in `numpyro.contrib.tfp.distributions`), 64
- `flax_module()` (in module `numpyro.contrib.module`), 14
- `fori_collect()` (in module `numpyro.util`), 103
- `forward_shape()` (AffineTransform method), 79
- `forward_shape()` (ComposeTransform method), 79
- `forward_shape()` (CorrCholeskyTransform method), 80
- `forward_shape()` (LowerCholeskyAffine method), 81
- `forward_shape()` (LowerCholeskyTransform method), 81
- `forward_shape()` (PowerTransform method), 82
- `forward_shape()` (StickBreakingTransform method), 82
- `forward_shape()` (Transform method), 78
- ## G
- Gamma (class in `numpyro.contrib.tfp.distributions`), 64
- Gamma (class in `numpyro.distributions.continuous`), 39
- GammaGamma (class in `numpyro.contrib.tfp.distributions`), 65
- GammaPoisson (class in `numpyro.distributions.conjugate`), 55
- GaussianProcess (class in `numpyro.contrib.tfp.distributions`), 65
- GaussianProcessRegressionModel (class in `numpyro.contrib.tfp.distributions`), 65
- GaussianRandomWalk (class in `numpyro.distributions.continuous`), 40
- `gelman_rubin()` (in module `numpyro.diagnostics`), 130
- GeneralizedExtremeValue (class in `numpyro.contrib.tfp.distributions`), 65
- GeneralizedNormal (class in `numpyro.contrib.tfp.distributions`), 65
- GeneralizedPareto (class in `numpyro.contrib.tfp.distributions`), 65
- Geometric (class in `numpyro.contrib.tfp.distributions`), 66
- `Geometric()` (in module `numpyro.distributions.discrete`), 56
- GeometricLogits (class in `numpyro.distributions.discrete`), 56
- GeometricProbs (class in `numpyro.distributions.discrete`), 56
- `get_base_dist()` (AutoBNAFNormal method), 111
- `get_base_dist()` (AutoContinuous method), 109
- `get_base_dist()` (AutoDiagonalNormal method), 111
- `get_base_dist()` (AutoIAFNormal method), 113
- `get_base_dist()` (AutoLaplaceApproximation method), 113
- `get_base_dist()` (AutoLowRankMultivariateNormal method), 115
- `get_base_dist()` (AutoMultivariateNormal method), 112
- `get_diagnostics_str()` (HMC method), 89
- `get_diagnostics_str()` (HMC Gibbs method), 91
- `get_diagnostics_str()` (MCMCKernel method), 88
- `get_diagnostics_str()` (SA method), 96
- `get_extra_fields()` (MCMC method), 86
- `get_params()` (Adagrad method), 123
- `get_params()` (Adam method), 122
- `get_params()` (ClippedAdam method), 124
- `get_params()` (Minimize method), 125
- `get_params()` (Momentum method), 126
- `get_params()` (RMSPProp method), 126
- `get_params()` (RMSPPropMomentum method), 127
- `get_params()` (SGD method), 128
- `get_params()` (SM3 method), 129
- `get_params()` (SVI method), 105
- `get_posterior()` (AutoContinuous method), 109
- `get_posterior()` (AutoDiagonalNormal method), 111
- `get_posterior()` (AutoLaplaceApproximation method), 114
- `get_posterior()` (AutoLowRankMultivariateNormal method), 115
- `get_posterior()` (AutoMultivariateNormal method), 112
- `get_samples()` (MCMC method), 86
- `get_trace()` (trace method), 27
- `get_transform()` (AutoContinuous method), 109
- `get_transform()` (AutoDiagonalNormal method), 111
- `get_transform()` (AutoLaplaceApproximation method), 113



- [get\\_transform\(\)](#) (*AutoLowRankMultivariateNormal method*), 115  
[get\\_transform\(\)](#) (*AutoMultivariateNormal method*), 112  
[greater\\_than\(\)](#) (in module *numpyro.distributions.constraints*), 76  
[Gumbel](#) (class in *numpyro.contrib.tfp.distributions*), 66  
[Gumbel](#) (class in *numpyro.distributions.continuous*), 40
- ## H
- [haiku\\_module\(\)](#) (in module *numpyro.contrib.module*), 15  
[HalfCauchy](#) (class in *numpyro.contrib.tfp.distributions*), 66  
[HalfCauchy](#) (class in *numpyro.distributions.continuous*), 41  
[HalfNormal](#) (class in *numpyro.contrib.tfp.distributions*), 66  
[HalfNormal](#) (class in *numpyro.distributions.continuous*), 41  
[HalfStudentT](#) (class in *numpyro.contrib.tfp.distributions*), 66  
[HamiltonianMonteCarlo](#) (class in *numpyro.contrib.tfp.mcmc*), 101  
[has\\_enumerate\\_support](#) (*BernoulliLogits attribute*), 50  
[has\\_enumerate\\_support](#) (*BernoulliProbs attribute*), 50  
[has\\_enumerate\\_support](#) (*BetaBinomial attribute*), 51  
[has\\_enumerate\\_support](#) (*BinomialLogits attribute*), 52  
[has\\_enumerate\\_support](#) (*BinomialProbs attribute*), 52  
[has\\_enumerate\\_support](#) (*CategoricalLogits attribute*), 53  
[has\\_enumerate\\_support](#) (*CategoricalProbs attribute*), 54  
[has\\_enumerate\\_support](#) (*Distribution attribute*), 28  
[has\\_enumerate\\_support](#) (*ExpandedDistribution attribute*), 30  
[has\\_enumerate\\_support](#) (*Independent attribute*), 32  
[has\\_enumerate\\_support](#) (*MaskedDistribution attribute*), 34  
[has\\_rsample](#) (*Distribution attribute*), 28  
[has\\_rsample](#) (*ExpandedDistribution attribute*), 30  
[has\\_rsample](#) (*Independent attribute*), 33  
[has\\_rsample](#) (*MaskedDistribution attribute*), 34  
[has\\_rsample](#) (*TransformedDistribution attribute*), 35  
[HiddenMarkovModel](#) (class in *numpyro.contrib.tfp.distributions*), 66  
[HMC](#) (class in *numpyro.infer.hmc*), 88  
[hmc\(\)](#) (in module *numpyro.infer.hmc*), 96  
[HMCECS](#) (class in *numpyro.infer.hmc\_gibbs*), 94  
[HMC Gibbs](#) (class in *numpyro.infer.hmc\_gibbs*), 91  
[HMC Gibbs State](#) (in module *numpyro.infer.hmc\_gibbs*), 99  
[HMC State](#) (in module *numpyro.infer.hmc*), 99  
[Horseshoe](#) (class in *numpyro.contrib.tfp.distributions*), 66  
[hpdi\(\)](#) (in module *numpyro.diagnostics*), 130
- ## I
- [icdf\(\)](#) (*Normal method*), 47  
[IdentityTransform](#) (class in *numpyro.distributions.transforms*), 80  
[ImproperUniform](#) (class in *numpyro.distributions.distribution*), 31  
[Independent](#) (class in *numpyro.contrib.tfp.distributions*), 67  
[Independent](#) (class in *numpyro.distributions.distribution*), 32  
[infer\\_config](#) (class in *numpyro.contrib.functor.enum\_messenger*), 119  
[infer\\_config](#) (class in *numpyro.handlers*), 22  
[init\(\)](#) (*Adagrad method*), 123  
[init\(\)](#) (*Adam method*), 123  
[init\(\)](#) (*ClippedAdam method*), 124  
[init\(\)](#) (*DiscreteHMC Gibbs method*), 93  
[init\(\)](#) (*HMC method*), 89  
[init\(\)](#) (*HMCECS method*), 94  
[init\(\)](#) (*HMC Gibbs method*), 92  
[init\(\)](#) (*MCMCKernel method*), 87  
[init\(\)](#) (*Minimize method*), 125  
[init\(\)](#) (*Momentum method*), 126  
[init\(\)](#) (*RMSPProp method*), 127  
[init\(\)](#) (*RMSPPropMomentum method*), 127  
[init\(\)](#) (*SA method*), 96  
[init\(\)](#) (*SGD method*), 128  
[init\(\)](#) (*SM3 method*), 129  
[init\(\)](#) (*SVI method*), 105  
[init\\_kernel\(\)](#) (in module *numpyro.infer.hmc.hmc*), 98  
[init\\_to\\_feasible\(\)](#) (in module *numpyro.infer.initialization*), 135  
[init\\_to\\_median\(\)](#) (in module *numpyro.infer.initialization*), 135  
[init\\_to\\_sample\(\)](#) (in module *numpyro.infer.initialization*), 136  
[init\\_to\\_uniform\(\)](#) (in module *numpyro.infer.initialization*), 136  
[init\\_to\\_value\(\)](#) (in module *numpyro.infer.initialization*), 136  
[initialize\\_model\(\)](#) (in module *numpyro.infer.util*), 102

[integer\\_greater\\_than\(\)](#) (in module [numpyro.distributions.constraints](#)), 77  
[integer\\_interval\(\)](#) (in module [numpyro.distributions.constraints](#)), 76  
[interval\(\)](#) (in module [numpyro.distributions.constraints](#)), 77  
[inv](#) (*Transform attribute*), 78  
[InvCholeskyTransform](#) (class in [numpyro.distributions.transforms](#)), 80  
[inverse\\_shape\(\)](#) (*AffineTransform method*), 79  
[inverse\\_shape\(\)](#) (*ComposeTransform method*), 79  
[inverse\\_shape\(\)](#) (*CorrCholeskyTransform method*), 80  
[inverse\\_shape\(\)](#) (*LowerCholeskyAffine method*), 81  
[inverse\\_shape\(\)](#) (*LowerCholeskyTransform method*), 81  
[inverse\\_shape\(\)](#) (*PowerTransform method*), 82  
[inverse\\_shape\(\)](#) (*StickBreakingTransform method*), 82  
[inverse\\_shape\(\)](#) (*Transform method*), 78  
[InverseAutoregressiveTransform](#) (class in [numpyro.distributions.flows](#)), 83  
[InverseGamma](#) (class in [numpyro.contrib.tfp.distributions](#)), 67  
[InverseGamma](#) (class in [numpyro.distributions.continuous](#)), 42  
[InverseGaussian](#) (class in [numpyro.contrib.tfp.distributions](#)), 67  
[is\\_discrete](#) (*BernoulliLogits attribute*), 50  
[is\\_discrete](#) (*BernoulliProbs attribute*), 50  
[is\\_discrete](#) (*BetaBinomial attribute*), 51  
[is\\_discrete](#) (*BinomialLogits attribute*), 52  
[is\\_discrete](#) (*BinomialProbs attribute*), 52  
[is\\_discrete](#) (*CategoricalLogits attribute*), 53  
[is\\_discrete](#) (*CategoricalProbs attribute*), 54  
[is\\_discrete](#) (*Delta attribute*), 36  
[is\\_discrete](#) (*DirichletMultinomial attribute*), 55  
[is\\_discrete](#) (*Distribution attribute*), 28  
[is\\_discrete](#) (*ExpandedDistribution attribute*), 30  
[is\\_discrete](#) (*GammaPoisson attribute*), 55  
[is\\_discrete](#) (*GeometricLogits attribute*), 56  
[is\\_discrete](#) (*GeometricProbs attribute*), 56  
[is\\_discrete](#) (*Independent attribute*), 33  
[is\\_discrete](#) (*MaskedDistribution attribute*), 34  
[is\\_discrete](#) (*MultinomialLogits attribute*), 57  
[is\\_discrete](#) (*MultinomialProbs attribute*), 58  
[is\\_discrete](#) (*Poisson attribute*), 59  
[is\\_discrete](#) (*PRNGIdentity attribute*), 59  
[is\\_discrete](#) (*ZeroInflatedPoisson attribute*), 60

## J

[JohnsonSU](#) (class in [numpyro.contrib.tfp.distributions](#)), 67

[JointDistribution](#) (class in [numpyro.contrib.tfp.distributions](#)), 67  
[JointDistributionCoroutine](#) (class in [numpyro.contrib.tfp.distributions](#)), 67  
[JointDistributionCoroutineAutoBatched](#) (class in [numpyro.contrib.tfp.distributions](#)), 67  
[JointDistributionNamed](#) (class in [numpyro.contrib.tfp.distributions](#)), 68  
[JointDistributionNamedAutoBatched](#) (class in [numpyro.contrib.tfp.distributions](#)), 68  
[JointDistributionSequential](#) (class in [numpyro.contrib.tfp.distributions](#)), 68  
[JointDistributionSequentialAutoBatched](#) (class in [numpyro.contrib.tfp.distributions](#)), 68

## K

[Kumaraswamy](#) (class in [numpyro.contrib.tfp.distributions](#)), 68

## L

[Laplace](#) (class in [numpyro.contrib.tfp.distributions](#)), 68  
[Laplace](#) (class in [numpyro.distributions.continuous](#)), 42  
[last\\_state](#) (*MCMC attribute*), 85  
[less\\_than\(\)](#) (in module [numpyro.distributions.constraints](#)), 77  
[lift](#) (class in [numpyro.handlers](#)), 22  
[LinearGaussianStateSpaceModel](#) (class in [numpyro.contrib.tfp.distributions](#)), 69  
[LKJ](#) (class in [numpyro.contrib.tfp.distributions](#)), 68  
[LKJ](#) (class in [numpyro.distributions.continuous](#)), 43  
[LKJCholesky](#) (class in [numpyro.distributions.continuous](#)), 43  
[LocScaleReparam](#) (class in [numpyro.infer.reparam](#)), 117  
[log\\_abs\\_det\\_jacobian\(\)](#) (*AffineTransform method*), 79  
[log\\_abs\\_det\\_jacobian\(\)](#) (*BlockNeuralAutoregressiveTransform method*), 83  
[log\\_abs\\_det\\_jacobian\(\)](#) (*ComposeTransform method*), 79  
[log\\_abs\\_det\\_jacobian\(\)](#) (*CorrCholeskyTransform method*), 80  
[log\\_abs\\_det\\_jacobian\(\)](#) (*ExpTransform method*), 80  
[log\\_abs\\_det\\_jacobian\(\)](#) (*IdentityTransform method*), 80  
[log\\_abs\\_det\\_jacobian\(\)](#) (*InvCholeskyTransform method*), 80  
[log\\_abs\\_det\\_jacobian\(\)](#) (*InverseAutoregressiveTransform method*), 83  
[log\\_abs\\_det\\_jacobian\(\)](#) (*LowerCholeskyAffine method*), 81

- `log_abs_det_jacobian()` (*LowerCholeskyTransform method*), 81  
`log_abs_det_jacobian()` (*OrderedTransform method*), 81  
`log_abs_det_jacobian()` (*PermuteTransform method*), 82  
`log_abs_det_jacobian()` (*PowerTransform method*), 82  
`log_abs_det_jacobian()` (*SigmoidTransform method*), 82  
`log_abs_det_jacobian()` (*StickBreakingTransform method*), 82  
`log_abs_det_jacobian()` (*Transform method*), 78  
`log_density()` (in module *numpyro.contrib.functor.infer\_util*), 121  
`log_density()` (in module *numpyro.infer.util*), 133  
`log_likelihood()` (in module *numpyro.infer.util*), 134  
`log_prob()` (*BernoulliLogits method*), 50  
`log_prob()` (*BernoulliProbs method*), 51  
`log_prob()` (*Beta method*), 37  
`log_prob()` (*BetaBinomial method*), 51  
`log_prob()` (*BinomialLogits method*), 52  
`log_prob()` (*BinomialProbs method*), 53  
`log_prob()` (*CategoricalLogits method*), 53  
`log_prob()` (*CategoricalProbs method*), 54  
`log_prob()` (*Cauchy method*), 38  
`log_prob()` (*Delta method*), 36  
`log_prob()` (*Dirichlet method*), 38  
`log_prob()` (*DirichletMultinomial method*), 55  
`log_prob()` (*Distribution method*), 29  
`log_prob()` (*ExpandedDistribution method*), 30  
`log_prob()` (*Exponential method*), 39  
`log_prob()` (*Gamma method*), 39  
`log_prob()` (*GammaPoisson method*), 56  
`log_prob()` (*GaussianRandomWalk method*), 41  
`log_prob()` (*GeometricLogits method*), 56  
`log_prob()` (*GeometricProbs method*), 57  
`log_prob()` (*Gumbel method*), 40  
`log_prob()` (*HalfCauchy method*), 41  
`log_prob()` (*HalfNormal method*), 42  
`log_prob()` (*ImproperUniform method*), 32  
`log_prob()` (*Independent method*), 33  
`log_prob()` (*Laplace method*), 43  
`log_prob()` (*LKJCholesky method*), 44  
`log_prob()` (*Logistic method*), 45  
`log_prob()` (*LowRankMultivariateNormal method*), 46  
`log_prob()` (*MaskedDistribution method*), 34  
`log_prob()` (*MultinomialLogits method*), 57  
`log_prob()` (*MultinomialProbs method*), 58  
`log_prob()` (*MultivariateNormal method*), 46  
`log_prob()` (*Normal method*), 47  
`log_prob()` (*Poisson method*), 59  
`log_prob()` (*StudentT method*), 48  
`log_prob()` (*TransformedDistribution method*), 35  
`log_prob()` (*TruncatedPolyaGamma method*), 49  
`log_prob()` (*Unit method*), 37  
`log_prob()` (*VonMises method*), 60  
`log_prob()` (*ZeroInflatedPoisson method*), 60  
*Logistic* (class in *numpyro.contrib.tfp.distributions*), 69  
*Logistic* (class in *numpyro.distributions.continuous*), 45  
*LogitNormal* (class in *numpyro.contrib.tfp.distributions*), 69  
*logits* (*BernoulliProbs* attribute), 51  
*logits* (*BinomialProbs* attribute), 53  
*logits* (*CategoricalProbs* attribute), 54  
*logits* (*GeometricProbs* attribute), 57  
*logits* (*MultinomialProbs* attribute), 58  
*LogLogistic* (class in *numpyro.contrib.tfp.distributions*), 69  
*LogNormal* (class in *numpyro.contrib.tfp.distributions*), 69  
*LogNormal* (class in *numpyro.distributions.continuous*), 44  
`loss()` (*RenyiELBO method*), 108  
`loss()` (*Trace\_ELBO method*), 107  
`loss()` (*TraceMeanField\_ELBO method*), 107  
*lower\_cholesky* (in module *numpyro.distributions.constraints*), 77  
*LowerCholeskyAffine* (class in *numpyro.distributions.transforms*), 81  
*LowerCholeskyTransform* (class in *numpyro.distributions.transforms*), 81  
*LowRankMultivariateNormal* (class in *numpyro.distributions.continuous*), 46
- ## M
- `markov()` (in module *numpyro.contrib.functor.enum\_messenger*), 120  
*mask* (class in *numpyro.handlers*), 23  
`mask()` (*Distribution method*), 30  
*MaskedDistribution* (class in *numpyro.distributions.distribution*), 33  
*MCMC* (class in *numpyro.infer.mcmc*), 84  
*MCMCKernel* (class in *numpyro.infer.mcmc*), 86  
*mean* (*BernoulliLogits* attribute), 50  
*mean* (*BernoulliProbs* attribute), 51  
*mean* (*Beta* attribute), 37  
*mean* (*BetaBinomial* attribute), 51  
*mean* (*BinomialLogits* attribute), 52  
*mean* (*BinomialProbs* attribute), 53  
*mean* (*CategoricalLogits* attribute), 53  
*mean* (*CategoricalProbs* attribute), 54  
*mean* (*Cauchy* attribute), 38

[mean \(Delta attribute\)](#), 36  
[mean \(Dirichlet attribute\)](#), 38  
[mean \(DirichletMultinomial attribute\)](#), 55  
[mean \(Distribution attribute\)](#), 29  
[mean \(ExpandedDistribution attribute\)](#), 31  
[mean \(Exponential attribute\)](#), 39  
[mean \(Gamma attribute\)](#), 40  
[mean \(GammaPoisson attribute\)](#), 56  
[mean \(GaussianRandomWalk attribute\)](#), 41  
[mean \(GeometricLogits attribute\)](#), 56  
[mean \(GeometricProbs attribute\)](#), 57  
[mean \(Gumbel attribute\)](#), 40  
[mean \(HalfCauchy attribute\)](#), 41  
[mean \(HalfNormal attribute\)](#), 42  
[mean \(Independent attribute\)](#), 33  
[mean \(InverseGamma attribute\)](#), 42  
[mean \(Laplace attribute\)](#), 43  
[mean \(LKJ attribute\)](#), 43  
[mean \(Logistic attribute\)](#), 45  
[mean \(LogNormal attribute\)](#), 44  
[mean \(LowRankMultivariateNormal attribute\)](#), 46  
[mean \(MaskedDistribution attribute\)](#), 34  
[mean \(MultinomialLogits attribute\)](#), 57  
[mean \(MultinomialProbs attribute\)](#), 58  
[mean \(MultivariateNormal attribute\)](#), 46  
[mean \(Normal attribute\)](#), 47  
[mean \(Pareto attribute\)](#), 47  
[mean \(Poisson attribute\)](#), 59  
[mean \(StudentT attribute\)](#), 48  
[mean \(TransformedDistribution attribute\)](#), 35  
[mean \(TruncatedCauchy attribute\)](#), 48  
[mean \(TruncatedNormal attribute\)](#), 48  
[mean \(Uniform attribute\)](#), 49  
[mean \(VonMises attribute\)](#), 60  
[mean \(ZeroInflatedPoisson attribute\)](#), 60  
[median\(\) \(AutoContinuous method\)](#), 110  
[median\(\) \(AutoDelta method\)](#), 117  
[median\(\) \(AutoDiagonalNormal method\)](#), 111  
[median\(\) \(AutoLaplaceApproximation method\)](#), 114  
[median\(\) \(AutoLowRankMultivariateNormal method\)](#), 115  
[median\(\) \(AutoMultivariateNormal method\)](#), 112  
[median\(\) \(AutoNormal method\)](#), 116  
[MetropolisAdjustedLangevinAlgorithm \(class in \[numpyro.contrib.tfp.mcmc\]\(#\)\)](#), 101  
[Minimize \(class in \[numpyro.optim\]\(#\)\)](#), 124  
[MixtureSameFamily \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 69  
[model \(HMC attribute\)](#), 89  
[model \(HMCGibbs attribute\)](#), 91  
[module\(\) \(in module \[numpyro.primitives\]\(#\)\)](#), 14  
[Momentum \(class in \[numpyro.optim\]\(#\)\)](#), 126  
[Moyal \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 69

[Multinomial \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 70  
[multinomial\(\) \(in module \[numpyro.distributions.constraints\]\(#\)\)](#), 77  
[Multinomial\(\) \(in module \[numpyro.distributions.discrete\]\(#\)\)](#), 57  
[MultinomialLogits \(class in \[numpyro.distributions.discrete\]\(#\)\)](#), 57  
[MultinomialProbs \(class in \[numpyro.distributions.discrete\]\(#\)\)](#), 58  
[MultivariateNormal \(class in \[numpyro.distributions.continuous\]\(#\)\)](#), 45  
[MultivariateNormalDiag \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 70  
[MultivariateNormalDiagPlusLowRank \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 70  
[MultivariateNormalFullCovariance \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 70  
[MultivariateNormalLinearOperator \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 70  
[MultivariateNormalTriL \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 70  
[MultivariateStudentTLinearOperator \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 71

## N

[NegativeBinomial \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 71  
[NeuTraReparam \(class in \[numpyro.infer.reparam\]\(#\)\)](#), 118  
[nonnegative\\_integer \(in module \[numpyro.distributions.constraints\]\(#\)\)](#), 77  
[Normal \(class in \[numpyro.contrib.tfp.distributions\]\(#\)\)](#), 71  
[Normal \(class in \[numpyro.distributions.continuous\]\(#\)\)](#), 46  
[NoUTurnSampler \(class in \[numpyro.contrib.tfp.mcmc\]\(#\)\)](#), 101  
[num\\_gamma\\_variates \(TruncatedPolyaGamma attribute\)](#), 49  
[num\\_log\\_prob\\_terms \(TruncatedPolyaGamma attribute\)](#), 49  
[numpyro.contrib.functor \(module\)](#), 119  
[numpyro.contrib.indexing \(module\)](#), 136  
[numpyro.contrib.tfp.distributions \(module\)](#), 60  
[numpyro.contrib.tfp.mcmc \(module\)](#), 100  
[numpyro.diagnostics \(module\)](#), 129  
[numpyro.handlers \(module\)](#), 19  
[numpyro.infer.autoguide \(module\)](#), 109  
[numpyro.infer.reparam \(module\)](#), 117  
[numpyro.infer.util \(module\)](#), 132  
[numpyro.optim \(module\)](#), 122  
[numpyro.primitives \(module\)](#), 11  
[numpyro.util \(module\)](#), 131  
[NUTS \(class in \[numpyro.infer.hmc\]\(#\)\)](#), 90



## O

OneHotCategorical (class in *numpyro.contrib.tfp.distributions*), 71  
 ordered\_vector (in module *numpyro.distributions.constraints*), 77  
 OrderedLogistic (class in *numpyro.contrib.tfp.distributions*), 71  
 OrderedLogistic (class in *numpyro.distributions.discrete*), 58  
 OrderedTransform (class in *numpyro.distributions.transforms*), 81

## P

param() (in module *numpyro.primitives*), 11  
 parametric() (in module *numpyro.infer.hmc\_util*), 104  
 parametric\_draws() (in module *numpyro.infer.hmc\_util*), 104  
 Pareto (class in *numpyro.contrib.tfp.distributions*), 71  
 Pareto (class in *numpyro.distributions.continuous*), 47  
 PermuteTransform (class in *numpyro.distributions.transforms*), 82  
 PERT (class in *numpyro.contrib.tfp.distributions*), 71  
 PlackettLuce (class in *numpyro.contrib.tfp.distributions*), 72  
 plate (class in *numpyro.contrib.functor.enum\_messenger*), 120  
 plate (class in *numpyro.primitives*), 12  
 plate\_stack() (in module *numpyro.primitives*), 13  
 plate\_to\_enum\_plate() (in module *numpyro.contrib.functor.infer\_util*), 122  
 Poisson (class in *numpyro.contrib.tfp.distributions*), 72  
 Poisson (class in *numpyro.distributions.discrete*), 58  
 PoissonLogNormalQuadratureCompound (class in *numpyro.contrib.tfp.distributions*), 72  
 positive (in module *numpyro.distributions.constraints*), 77  
 positive\_definite (in module *numpyro.distributions.constraints*), 78  
 positive\_integer (in module *numpyro.distributions.constraints*), 78  
 post\_warmup\_state (MCMC attribute), 84  
 postprocess\_fn() (HMC method), 89  
 postprocess\_fn() (HMC Gibbs method), 91  
 postprocess\_fn() (MCMC Kernel method), 87  
 postprocess\_fn() (SA method), 96  
 postprocess\_message() (plate method), 120  
 postprocess\_message() (trace method), 27, 121  
 potential\_energy() (in module *numpyro.infer.util*), 134  
 PowerSpherical (class in *numpyro.contrib.tfp.distributions*), 72  
 PowerTransform (class in *numpyro.distributions.transforms*), 82

precision\_matrix (*LowRankMultivariateNormal* attribute), 46  
 precision\_matrix (*MultivariateNormal* attribute), 46  
 Predictive (class in *numpyro.infer.util*), 132  
 print\_summary() (in module *numpyro.diagnostics*), 131  
 print\_summary() (MCMC method), 86  
 prng\_key() (in module *numpyro.primitives*), 14  
 PRNGIdentity (class in *numpyro.distributions.discrete*), 59  
 ProbitBernoulli (class in *numpyro.contrib.tfp.distributions*), 72  
 probs (*BernoulliLogits* attribute), 50  
 probs (*BinomialLogits* attribute), 52  
 probs (*CategoricalLogits* attribute), 53  
 probs (*GeometricLogits* attribute), 56  
 probs (*MultinomialLogits* attribute), 57  
 process\_message() (block method), 20  
 process\_message() (collapse method), 21  
 process\_message() (condition method), 21  
 process\_message() (do method), 22  
 process\_message() (enum method), 119  
 process\_message() (infer\_config method), 22, 120  
 process\_message() (lift method), 23  
 process\_message() (mask method), 23  
 process\_message() (plate method), 120  
 process\_message() (reparam method), 23  
 process\_message() (replay method), 24  
 process\_message() (scale method), 24  
 process\_message() (scope method), 25  
 process\_message() (seed method), 26  
 process\_message() (substitute method), 26

## Q

quantiles() (*AutoContinuous* method), 110  
 quantiles() (*AutoDiagonalNormal* method), 111  
 quantiles() (*AutoLaplaceApproximation* method), 114  
 quantiles() (*AutoLowRankMultivariateNormal* method), 115  
 quantiles() (*AutoMultivariateNormal* method), 112  
 quantiles() (*AutoNormal* method), 116  
 QuantizedDistribution (class in *numpyro.contrib.tfp.distributions*), 72

## R

random\_flax\_module() (in module *numpyro.contrib.module*), 15  
 random\_haiku\_module() (in module *numpyro.contrib.module*), 17  
 RandomWalkMetropolis (class in *numpyro.contrib.tfp.mcmc*), 101  
 real (in module *numpyro.distributions.constraints*), 78

`real_vector` (in `numpyro.distributions.constraints`), 78  
`RelaxedBernoulli` (class in `numpyro.contrib.tfp.distributions`), 72  
`RelaxedOneHotCategorical` (class in `numpyro.contrib.tfp.distributions`), 73  
`RenyiELBO` (class in `numpyro.infer.elbo`), 108  
`reparam` (class in `numpyro.handlers`), 23  
`Reparam` (class in `numpyro.infer.reparam`), 117  
`reparam()` (*NeuTraReparam* method), 118  
`reparameterized_params` (*Delta* attribute), 36  
`reparameterized_params` (*Independent* attribute), 33  
`reparametrized_params` (*Beta* attribute), 37  
`reparametrized_params` (*Cauchy* attribute), 37  
`reparametrized_params` (*Chi2* attribute), 38  
`reparametrized_params` (*Dirichlet* attribute), 38  
`reparametrized_params` (*Distribution* attribute), 28  
`reparametrized_params` (*Exponential* attribute), 39  
`reparametrized_params` (*Gamma* attribute), 39  
`reparametrized_params` (*GaussianRandomWalk* attribute), 40  
`reparametrized_params` (*Gumbel* attribute), 40  
`reparametrized_params` (*HalfCauchy* attribute), 41  
`reparametrized_params` (*HalfNormal* attribute), 41  
`reparametrized_params` (*InverseGamma* attribute), 42  
`reparametrized_params` (*Laplace* attribute), 42  
`reparametrized_params` (*LKJ* attribute), 43  
`reparametrized_params` (*LKJCholesky* attribute), 44  
`reparametrized_params` (*Logistic* attribute), 45  
`reparametrized_params` (*LogNormal* attribute), 44  
`reparametrized_params` (*LowRankMultivariateNormal* attribute), 46  
`reparametrized_params` (*MultivariateNormal* attribute), 45  
`reparametrized_params` (*Normal* attribute), 47  
`reparametrized_params` (*Pareto* attribute), 47  
`reparametrized_params` (*StudentT* attribute), 47  
`reparametrized_params` (*TruncatedCauchy* attribute), 48  
`reparametrized_params` (*TruncatedNormal* attribute), 48  
`reparametrized_params` (*Uniform* attribute), 49  
`reparametrized_params` (*VonMises* attribute), 60  
`replay` (class in `numpyro.handlers`), 24  
`ReplicaExchangeMC` (class in `numpyro.contrib.tfp.mcmc`), 101  
`RMSProp` (class in `numpyro.optim`), 126  
`RMSPropMomentum` (class in `numpyro.optim`), 127  
`rsample()` (*Distribution* method), 28  
`rsample()` (*ExpandedDistribution* method), 30  
`rsample()` (*Independent* method), 33  
`rsample()` (*MaskedDistribution* method), 34  
`rsample()` (*TransformedDistribution* method), 35  
`run()` (*MCMC* method), 85  
`run()` (*SVI* method), 106

## S

`SA` (class in `numpyro.infer.sa`), 95  
`Sample` (class in `numpyro.contrib.tfp.distributions`), 73  
`sample()` (*BernoulliLogits* method), 50  
`sample()` (*BernoulliProbs* method), 50  
`sample()` (*Beta* method), 37  
`sample()` (*BetaBinomial* method), 51  
`sample()` (*BinomialLogits* method), 52  
`sample()` (*BinomialProbs* method), 52  
`sample()` (*CategoricalLogits* method), 53  
`sample()` (*CategoricalProbs* method), 54  
`sample()` (*Cauchy* method), 37  
`sample()` (*Delta* method), 36  
`sample()` (*Dirichlet* method), 38  
`sample()` (*DirichletMultinomial* method), 55  
`sample()` (*DiscreteHMC* Gibbs method), 93  
`sample()` (*Distribution* method), 28  
`sample()` (*ExpandedDistribution* method), 30  
`sample()` (*Exponential* method), 39  
`sample()` (*Gamma* method), 39  
`sample()` (*GammaPoisson* method), 55  
`sample()` (*GaussianRandomWalk* method), 40  
`sample()` (*GeometricLogits* method), 56  
`sample()` (*GeometricProbs* method), 56  
`sample()` (*Gumbel* method), 40  
`sample()` (*HalfCauchy* method), 41  
`sample()` (*HalfNormal* method), 41  
`sample()` (*HMC* method), 89  
`sample()` (*HMCECS* method), 95  
`sample()` (*HMC* Gibbs method), 92  
`sample()` (in module `numpyro.primitives`), 12  
`sample()` (*Independent* method), 33  
`sample()` (*Laplace* method), 42  
`sample()` (*LKJCholesky* method), 44  
`sample()` (*Logistic* method), 45  
`sample()` (*LowRankMultivariateNormal* method), 46  
`sample()` (*MaskedDistribution* method), 34  
`sample()` (*MCMCKernel* method), 87  
`sample()` (*MultinomialLogits* method), 57  
`sample()` (*MultinomialProbs* method), 58  
`sample()` (*MultivariateNormal* method), 45  
`sample()` (*Normal* method), 47  
`sample()` (*Poisson* method), 59  
`sample()` (*PRNGIdentity* method), 59

- `sample()` (*SA method*), 96
- `sample()` (*StudentT method*), 47
- `sample()` (*TransformedDistribution method*), 35
- `sample()` (*TruncatedPolyaGamma method*), 49
- `sample()` (*Unit method*), 36
- `sample()` (*VonMises method*), 60
- `sample()` (*ZeroInflatedPoisson method*), 60
- `sample_field` (*HMC attribute*), 89
- `sample_field` (*HMCGibbs attribute*), 91
- `sample_field` (*MCMCKernel attribute*), 88
- `sample_field` (*SA attribute*), 96
- `sample_kernel()` (in module `numpyro.infer.hmc.hmc`), 98
- `sample_posterior()` (*AutoContinuous method*), 109
- `sample_posterior()` (*AutoDelta method*), 117
- `sample_posterior()` (*AutoLaplaceApproximation method*), 114
- `sample_posterior()` (*AutoNormal method*), 116
- `sample_with_intermediates()` (*Distribution method*), 29
- `sample_with_intermediates()` (*Transformed-Distribution method*), 35
- `SASState` (in module `numpyro.infer.sa`), 99
- `scale` (class in `numpyro.handlers`), 24
- `scale_tril` (*LowRankMultivariateNormal attribute*), 46
- `scan()` (in module `numpyro.contrib.control_flow`), 17
- `scope` (class in `numpyro.handlers`), 24
- `seed` (class in `numpyro.handlers`), 25
- `set_default_validate_args()` (*Distribution static method*), 28
- `set_host_device_count()` (in module `numpyro.util`), 132
- `set_platform()` (in module `numpyro.util`), 132
- `SGD` (class in `numpyro.optim`), 128
- `shape()` (*Distribution method*), 28
- `SigmoidTransform` (class in `numpyro.distributions.transforms`), 82
- `simplex` (in module `numpyro.distributions.constraints`), 78
- `SinhArcsinh` (class in `numpyro.contrib.tfp.distributions`), 73
- `Skellam` (class in `numpyro.contrib.tfp.distributions`), 73
- `SliceSampler` (class in `numpyro.contrib.tfp.mcmc`), 101
- `SM3` (class in `numpyro.optim`), 128
- `SphericalUniform` (class in `numpyro.contrib.tfp.distributions`), 73
- `split_gelman_rubin()` (in module `numpyro.diagnostics`), 130
- `StickBreakingTransform` (class in `numpyro.distributions.transforms`), 82
- `StoppingRatioLogistic` (class in `numpyro.contrib.tfp.distributions`), 73
- `StudentT` (class in `numpyro.contrib.tfp.distributions`), 73
- `StudentT` (class in `numpyro.distributions.continuous`), 47
- `StudentTProcess` (class in `numpyro.contrib.tfp.distributions`), 74
- `subsample()` (in module `numpyro.primitives`), 13
- `substitute` (class in `numpyro.handlers`), 26
- `summary()` (in module `numpyro.diagnostics`), 131
- `support` (*BernoulliLogits attribute*), 50
- `support` (*BernoulliProbs attribute*), 50
- `support` (*Beta attribute*), 37
- `support` (*BetaBinomial attribute*), 52
- `support` (*BinomialLogits attribute*), 52
- `support` (*BinomialProbs attribute*), 53
- `support` (*CategoricalLogits attribute*), 54
- `support` (*CategoricalProbs attribute*), 54
- `support` (*Cauchy attribute*), 37
- `support` (*Delta attribute*), 36
- `support` (*Dirichlet attribute*), 38
- `support` (*DirichletMultinomial attribute*), 55
- `support` (*Distribution attribute*), 28
- `support` (*ExpandedDistribution attribute*), 30
- `support` (*Exponential attribute*), 39
- `support` (*Gamma attribute*), 39
- `support` (*GammaPoisson attribute*), 55
- `support` (*GaussianRandomWalk attribute*), 40
- `support` (*GeometricLogits attribute*), 56
- `support` (*GeometricProbs attribute*), 56
- `support` (*Gumbel attribute*), 40
- `support` (*HalfCauchy attribute*), 41
- `support` (*HalfNormal attribute*), 41
- `support` (*Independent attribute*), 32
- `support` (*InverseGamma attribute*), 42
- `support` (*Laplace attribute*), 42
- `support` (*LKJ attribute*), 43
- `support` (*LKJCholesky attribute*), 44
- `support` (*Logistic attribute*), 45
- `support` (*LowRankMultivariateNormal attribute*), 46
- `support` (*MaskedDistribution attribute*), 34
- `support` (*MultinomialLogits attribute*), 57
- `support` (*MultinomialProbs attribute*), 58
- `support` (*MultivariateNormal attribute*), 45
- `support` (*Normal attribute*), 47
- `support` (*Pareto attribute*), 47
- `support` (*Poisson attribute*), 58
- `support` (*StudentT attribute*), 47
- `support` (*TransformedDistribution attribute*), 35
- `support` (*TruncatedCauchy attribute*), 48
- `support` (*TruncatedNormal attribute*), 48
- `support` (*TruncatedPolyaGamma attribute*), 49
- `support` (*Uniform attribute*), 49
- `support` (*Unit attribute*), 36

support (*VonMises* attribute), 60  
 support (*ZeroInflatedPoisson* attribute), 60  
 SVI (class in `numpyro.infer.svi`), 104

## T

`TFPDistributionMixin` (class in `numpyro.contrib.tfp.distributions`), 61  
`TFPKernel` (class in `numpyro.contrib.tfp.mcmc`), 100  
`to_data()` (in module `numpyro.contrib.functor.enum_messenger`), 120  
`to_event()` (*Distribution* method), 29  
`to_functor()` (in module `numpyro.contrib.functor.enum_messenger`), 120  
`trace` (class in `numpyro.contrib.functor.enum_messenger`), 121  
`trace` (class in `numpyro.handlers`), 26  
`Trace_ELBO` (class in `numpyro.infer.elbo`), 107  
`TraceMeanField_ELBO` (class in `numpyro.infer.elbo`), 107  
`Transform` (class in `numpyro.distributions.transforms`), 78  
`transform_fn()` (in module `numpyro.infer.util`), 133  
`transform_sample()` (*NeuTraReparam* method), 118  
`TransformedDistribution` (class in `numpyro.contrib.tfp.distributions`), 74  
`TransformedDistribution` (class in `numpyro.distributions.distribution`), 35  
`TransformReparam` (class in `numpyro.infer.reparam`), 119  
`tree_flatten()` (*Delta* method), 36  
`tree_flatten()` (*Distribution* method), 28  
`tree_flatten()` (*ExpandedDistribution* method), 31  
`tree_flatten()` (*GaussianRandomWalk* method), 41  
`tree_flatten()` (*ImproperUniform* method), 32  
`tree_flatten()` (*Independent* method), 33  
`tree_flatten()` (*InverseGamma* method), 42  
`tree_flatten()` (*LKJ* method), 43  
`tree_flatten()` (*LKJCholesky* method), 44  
`tree_flatten()` (*LogNormal* method), 45  
`tree_flatten()` (*MaskedDistribution* method), 34  
`tree_flatten()` (*MultivariateNormal* method), 46  
`tree_flatten()` (*Pareto* method), 47  
`tree_flatten()` (*TransformedDistribution* method), 35  
`tree_flatten()` (*TruncatedCauchy* method), 48  
`tree_flatten()` (*TruncatedNormal* method), 48  
`tree_flatten()` (*TruncatedPolyaGamma* method), 49  
`tree_flatten()` (*Uniform* method), 49  
`tree_unflatten()` (`numpyro.distributions.continuous.GaussianRandomWalk` class method), 41

`tree_unflatten()` (`numpyro.distributions.continuous.LKJ` class method), 43  
`tree_unflatten()` (`numpyro.distributions.continuous.LKJCholesky` class method), 44  
`tree_unflatten()` (`numpyro.distributions.continuous.MultivariateNormal` class method), 46  
`tree_unflatten()` (`numpyro.distributions.continuous.TruncatedCauchy` class method), 48  
`tree_unflatten()` (`numpyro.distributions.continuous.TruncatedNormal` class method), 48  
`tree_unflatten()` (`numpyro.distributions.continuous.TruncatedPolyaGamma` class method), 49  
`tree_unflatten()` (`numpyro.distributions.continuous.Uniform` class method), 49  
`tree_unflatten()` (`numpyro.distributions.distribution.Delta` class method), 36  
`tree_unflatten()` (`numpyro.distributions.distribution.Distribution` class method), 28  
`tree_unflatten()` (`numpyro.distributions.distribution.ExpandedDistribution` class method), 31  
`tree_unflatten()` (`numpyro.distributions.distribution.Independent` class method), 33  
`tree_unflatten()` (`numpyro.distributions.distribution.MaskedDistribution` class method), 34  
`Triangular` (class in `numpyro.contrib.tfp.distributions`), 74  
`TruncatedCauchy` (class in `numpyro.contrib.tfp.distributions`), 74  
`TruncatedCauchy` (class in `numpyro.distributions.continuous`), 48  
`TruncatedNormal` (class in `numpyro.contrib.tfp.distributions`), 74  
`TruncatedNormal` (class in `numpyro.distributions.continuous`), 48  
`TruncatedPolyaGamma` (class in `numpyro.distributions.continuous`), 49  
`truncation_point` (*TruncatedPolyaGamma* attribute), 49

## U

`UncalibratedHamiltonianMonteCarlo` (class in `numpyro.contrib.tfp.mcmc`), 102  
`UncalibratedLangevin` (class in `numpyro.contrib.tfp.mcmc`), 102  
`UncalibratedRandomWalk` (class in `numpyro.contrib.tfp.mcmc`), 102  
`Uniform` (class in `numpyro.contrib.tfp.distributions`), 74  
`Uniform` (class in `numpyro.distributions.continuous`), 49  
`Unit` (class in `numpyro.distributions.distribution`), 36  
`unit_interval` (in module `numpyro.distributions.constraints`), 78  
`update()` (*Adam* method), 123  
`update()` (*AdamW* method), 123



[update\(\)](#) (*ClippedAdam method*), 124  
[update\(\)](#) (*Minimize method*), 125  
[update\(\)](#) (*Momentum method*), 126  
[update\(\)](#) (*RMSProp method*), 127  
[update\(\)](#) (*RMSPropMomentum method*), 127  
[update\(\)](#) (*SGD method*), 128  
[update\(\)](#) (*SM3 method*), 129  
[update\(\)](#) (*SVI method*), 105

## V

[validation\\_enabled\(\)](#) (in module *numpyro.distributions.distribution*), 131  
[variance](#) (*BernoulliLogits attribute*), 50  
[variance](#) (*BernoulliProbs attribute*), 51  
[variance](#) (*Beta attribute*), 37  
[variance](#) (*BetaBinomial attribute*), 51  
[variance](#) (*BinomialLogits attribute*), 52  
[variance](#) (*BinomialProbs attribute*), 53  
[variance](#) (*CategoricalLogits attribute*), 54  
[variance](#) (*CategoricalProbs attribute*), 54  
[variance](#) (*Cauchy attribute*), 38  
[variance](#) (*Delta attribute*), 36  
[variance](#) (*Dirichlet attribute*), 38  
[variance](#) (*DirichletMultinomial attribute*), 55  
[variance](#) (*Distribution attribute*), 29  
[variance](#) (*ExpandedDistribution attribute*), 31  
[variance](#) (*Exponential attribute*), 39  
[variance](#) (*Gamma attribute*), 40  
[variance](#) (*GammaPoisson attribute*), 56  
[variance](#) (*GaussianRandomWalk attribute*), 41  
[variance](#) (*GeometricLogits attribute*), 56  
[variance](#) (*GeometricProbs attribute*), 57  
[variance](#) (*Gumbel attribute*), 40  
[variance](#) (*HalfCauchy attribute*), 41  
[variance](#) (*HalfNormal attribute*), 42  
[variance](#) (*Independent attribute*), 33  
[variance](#) (*InverseGamma attribute*), 42  
[variance](#) (*Laplace attribute*), 43  
[variance](#) (*Logistic attribute*), 45  
[variance](#) (*LogNormal attribute*), 44  
[variance](#) (*LowRankMultivariateNormal attribute*), 46  
[variance](#) (*MaskedDistribution attribute*), 34  
[variance](#) (*MultinomialLogits attribute*), 57  
[variance](#) (*MultinomialProbs attribute*), 58  
[variance](#) (*MultivariateNormal attribute*), 46  
[variance](#) (*Normal attribute*), 47  
[variance](#) (*Pareto attribute*), 47  
[variance](#) (*Poisson attribute*), 59  
[variance](#) (*StudentT attribute*), 48  
[variance](#) (*TransformedDistribution attribute*), 35  
[variance](#) (*TruncatedCauchy attribute*), 48  
[variance](#) (*TruncatedNormal attribute*), 48  
[variance](#) (*Uniform attribute*), 49  
[variance](#) (*VonMises attribute*), 60

[variance](#) (*ZeroInflatedPoisson attribute*), 60  
[VariationalGaussianProcess](#) (class in *numpyro.contrib.tfp.distributions*), 74  
[VectorDeterministic](#) (class in *numpyro.contrib.tfp.distributions*), 75  
[VectorExponentialDiag](#) (class in *numpyro.contrib.tfp.distributions*), 75  
[Vindex](#) (class in *numpyro.contrib.indexing*), 137  
[vindex\(\)](#) (in module *numpyro.contrib.indexing*), 136  
[VonMises](#) (class in *numpyro.contrib.tfp.distributions*), 75  
[VonMises](#) (class in *numpyro.distributions.directional*), 60  
[VonMisesFisher](#) (class in *numpyro.contrib.tfp.distributions*), 75

## W

[warmup\(\)](#) (*MCMC method*), 85  
[Weibull](#) (class in *numpyro.contrib.tfp.distributions*), 75  
[WishartLinearOperator](#) (class in *numpyro.contrib.tfp.distributions*), 75  
[WishartTriL](#) (class in *numpyro.contrib.tfp.distributions*), 75

## Z

[ZeroInflatedPoisson](#) (class in *numpyro.distributions.discrete*), 59