
NumPyro Documentation

Uber AI Labs

Oct 19, 2020

1	Pyro Primitives	1
1.1	param	1
1.2	sample	1
1.3	plate	2
1.4	plate_stack	2
1.5	subsample	3
1.6	deterministic	3
1.7	factor	4
1.8	module	4
1.9	flax_module	4
1.10	haiku_module	4
1.11	random_flax_module	5
1.12	random_haiku_module	6
1.13	scan	7
2	Effect Handlers	11
2.1	block	12
2.2	collapse	13
2.3	condition	13
2.4	do	14
2.5	lift	14
2.6	mask	15
2.7	reparam	15
2.8	replay	16
2.9	scale	16
2.10	scope	16
2.11	seed	17
2.12	substitute	18
2.13	trace	18
3	Base Distribution	21
3.1	Distribution	21
3.2	ExpandedDistribution	24
3.3	ImproperUniform	24
3.4	Independent	26
3.5	MaskedDistribution	27
3.6	TransformedDistribution	28

3.7	Unit	29
4	Continuous Distributions	31
4.1	Beta	31
4.2	Cauchy	32
4.3	Chi2	32
4.4	Dirichlet	32
4.5	Exponential	33
4.6	Gamma	33
4.7	Gumbel	34
4.8	GaussianRandomWalk	34
4.9	HalfCauchy	35
4.10	HalfNormal	36
4.11	InverseGamma	36
4.12	Laplace	37
4.13	LKJ	37
4.14	LKJCholesky	38
4.15	LogNormal	39
4.16	Logistic	39
4.17	MultivariateNormal	40
4.18	LowRankMultivariateNormal	40
4.19	Normal	41
4.20	Pareto	41
4.21	StudentT	42
4.22	TruncatedCauchy	42
4.23	TruncatedNormal	43
4.24	TruncatedPolyaGamma	43
4.25	Uniform	44
5	Discrete Distributions	45
5.1	Bernoulli	45
5.2	BernoulliLogits	45
5.3	BernoulliProbs	46
5.4	BetaBinomial	46
5.5	Binomial	47
5.6	BinomialLogits	47
5.7	BinomialProbs	48
5.8	Categorical	49
5.9	CategoricalLogits	49
5.10	CategoricalProbs	49
5.11	Delta	50
5.12	DirichletMultinomial	51
5.13	GammaPoisson	51
5.14	Geometric	52
5.15	GeometricLogits	52
5.16	GeometricProbs	53
5.17	Multinomial	53
5.18	MultinomialLogits	53
5.19	MultinomialProbs	54
5.20	OrderedLogistic	54
5.21	Poisson	55
5.22	PRNGIdentity	55
5.23	ZeroInflatedPoisson	56

6	Directional Distributions	57
6.1	VonMises	57
7	TensorFlow Distributions	59
7.1	BijectorConstraint	59
7.2	BijectorTransform	59
7.3	TFPDistributionMixin	59
7.4	Autoregressive	59
7.5	BatchReshape	60
7.6	Bates	60
7.7	Bernoulli	60
7.8	Beta	60
7.9	BetaBinomial	60
7.10	Binomial	60
7.11	Blockwise	61
7.12	Categorical	61
7.13	Cauchy	61
7.14	Chi	61
7.15	Chi2	61
7.16	CholeskyLKJ	61
7.17	ContinuousBernoulli	61
7.18	Deterministic	62
7.19	Dirichlet	62
7.20	DirichletMultinomial	62
7.21	DoublesidedMaxwell	62
7.22	Empirical	62
7.23	ExpGamma	62
7.24	ExpInverseGamma	63
7.25	ExpRelaxedOneHotCategorical	63
7.26	Exponential	63
7.27	FiniteDiscrete	63
7.28	Gamma	63
7.29	GammaGamma	63
7.30	GaussianProcess	64
7.31	GaussianProcessRegressionModel	64
7.32	GeneralizedNormal	64
7.33	GeneralizedPareto	64
7.34	Geometric	64
7.35	Gumbel	64
7.36	HalfCauchy	65
7.37	HalfNormal	65
7.38	HalfStudentT	65
7.39	HiddenMarkovModel	65
7.40	Horseshoe	65
7.41	Independent	65
7.42	InverseGamma	66
7.43	InverseGaussian	66
7.44	JohnsonSU	66
7.45	JointDistribution	66
7.46	JointDistributionCoroutine	66
7.47	JointDistributionCoroutineAutoBatched	66
7.48	JointDistributionNamed	67
7.49	JointDistributionNamedAutoBatched	67
7.50	JointDistributionSequential	67

7.51	JointDistributionSequentialAutoBatched	67
7.52	Kumaraswamy	67
7.53	LKJ	67
7.54	Laplace	68
7.55	LinearGaussianStateSpaceModel	68
7.56	LogLogistic	68
7.57	LogNormal	68
7.58	Logistic	68
7.59	LogitNormal	68
7.60	MixtureSameFamily	68
7.61	Moyal	69
7.62	Multinomial	69
7.63	MultivariateNormalDiag	69
7.64	MultivariateNormalDiagPlusLowRank	69
7.65	MultivariateNormalFullCovariance	69
7.66	MultivariateNormalLinearOperator	69
7.67	MultivariateNormalTriL	70
7.68	MultivariateStudentTLinearOperator	70
7.69	NegativeBinomial	70
7.70	Normal	70
7.71	OneHotCategorical	70
7.72	OrderedLogistic	70
7.73	PERT	71
7.74	Pareto	71
7.75	PlackettLuce	71
7.76	Poisson	71
7.77	PoissonLogNormalQuadratureCompound	71
7.78	PowerSpherical	71
7.79	ProbitBernoulli	72
7.80	QuantizedDistribution	72
7.81	RelaxedBernoulli	72
7.82	RelaxedOneHotCategorical	72
7.83	Sample	72
7.84	SinhArcsinh	72
7.85	SphericalUniform	72
7.86	StudentT	73
7.87	StudentTProcess	73
7.88	TransformedDistribution	73
7.89	Triangular	73
7.90	TruncatedCauchy	73
7.91	TruncatedNormal	73
7.92	Uniform	74
7.93	VariationalGaussianProcess	74
7.94	VectorDeterministic	74
7.95	VectorExponentialDiag	74
7.96	VonMises	74
7.97	VonMisesFisher	74
7.98	Weibull	75
7.99	WishartLinearOperator	75
7.100	WishartTriL	75
8	Constraints	77
8.1	Constraint	77
8.2	boolean	77

8.3	corr_cholesky	77
8.4	corr_matrix	77
8.5	dependent	78
8.6	greater_than	78
8.7	integer_interval	78
8.8	integer_greater_than	78
8.9	interval	78
8.10	less_than	78
8.11	lower_cholesky	79
8.12	multinomial	79
8.13	nonnegative_integer	79
8.14	ordered_vector	79
8.15	positive	79
8.16	positive_definite	79
8.17	positive_integer	79
8.18	real	79
8.19	real_vector	79
8.20	simplex	80
8.21	unit_interval	80
9	Transforms	81
9.1	biject_to	81
9.2	Transform	81
9.3	AbsTransform	81
9.4	AffineTransform	82
9.5	ComposeTransform	82
9.6	CorrCholeskyTransform	82
9.7	ExpTransform	83
9.8	IdentityTransform	83
9.9	InvCholeskyTransform	84
9.10	LowerCholeskyAffine	84
9.11	LowerCholeskyTransform	84
9.12	OrderedTransform	85
9.13	PermuteTransform	85
9.14	PowerTransform	85
9.15	SigmoidTransform	85
9.16	StickBreakingTransform	86
10	Flows	87
10.1	InverseAutoregressiveTransform	87
10.2	BlockNeuralAutoregressiveTransform	88
11	Markov Chain Monte Carlo (MCMC)	89
11.1	MCMC Kernels	91
11.2	TensorFlow Kernels	100
11.3	MCMC Utilities	102
12	Stochastic Variational Inference (SVI)	105
12.1	ELBO	107
12.2	RenyiELBO	107
12.3	Trace_ELBO	108
13	Automatic Guide Generation	109
13.1	AutoContinuous	109
13.2	AutoBNAFNormal	110

13.3	AutoDiagonalNormal	111
13.4	AutoMultivariateNormal	112
13.5	AutoIAFNormal	113
13.6	AutoLaplaceApproximation	114
13.7	AutoLowRankMultivariateNormal	115
13.8	AutoNormal	116
14	Reparameterizers	119
14.1	Loc-Scale Decentering	119
14.2	Neural Transport	120
14.3	Transformed Distributions	121
15	Funsor-based NumPyro	123
15.1	Effect handlers	123
15.2	Inference Utilities	125
16	Optimizers	127
16.1	Adam	127
16.2	Adagrad	128
16.3	ClippedAdam	128
16.4	Minimize	129
16.5	Momentum	130
16.6	RMSProp	131
16.7	RMSPropMomentum	132
16.8	SGD	133
16.9	SM3	133
17	Diagnostics	135
17.1	Autocorrelation	135
17.2	Autocovariance	135
17.3	Effective Sample Size	136
17.4	Gelman Rubin	136
17.5	Split Gelman Rubin	136
17.6	HPDI	136
17.7	Summary	137
18	Runtime Utilities	139
18.1	enable_validation	139
18.2	validation_enabled	139
18.3	enable_x64	139
18.4	set_platform	140
18.5	set_host_device_count	140
19	Inference Utilities	141
19.1	Predictive	141
19.2	log_density	142
19.3	transform_fn	142
19.4	constrain_fn	142
19.5	potential_energy	143
19.6	log_likelihood	143
19.7	find_valid_initial_params	144
19.8	Initialization Strategies	144
19.9	Tensor Indexing	145
20	Indices and tables	147

Python Module Index	149
Index	151

1.1 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*) – initial value specified by the user. 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.

1.2 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*.

1.3 plate

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

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

Note: This can be used to subsample minibatches of data:

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

Parameters

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

1.4 plate_stack

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

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

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

Parameters

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

1.5 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*

1.6 deterministic

deterministic (*name*, *value*)

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

Parameters

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

1.7 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.

1.8 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.

1.9 flax_module

flax_module (*name*, *nn_module*, *, *input_shape=None*)

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.

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

1.10 haiku_module

haiku_module (*name*, *nn_module*, *, *input_shape=None*)

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.

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

1.11 random_flax_module

random_flax_module (*name*, *nn_module*, *prior*, *, *input_shape=None*)

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.

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
```

(continues on next page)

(continued from previous page)

```

>>>
>>> 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())
>>>
>>> batch_size = 256
>>> n_iterations = 3000
>>> svi_state = svi.init(random.PRNGKey(0), x_train, y_train, batch_size=batch_
↳ size)
>>> update_fn = jit(svi.update, static_argnums=(3,))
>>> for i in tqdm.trange(n_iterations):
...     svi_state, loss = update_fn(svi_state, x_train, y_train, batch_size)
>>>
>>> params = svi.get_params(svi_state)
>>> 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 loss < 3000
>>> assert np.sqrt(np.mean(np.square(y_test - y_pred))) < 1

```

1.12 random_haiku_module

random_haiku_module (*name, nn_module, prior, *, input_shape=None*)

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

1.13 scan

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

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

Currently, only the equivalence to `markov(history_size=1)` is supported. A *trace* of *scan* with discrete latent variables will contain the following sites:

- **init sites:** those sites belong to the first trace of *f*. Each of them will have name prefixed with `_init/`.
- **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

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”.

CHAPTER 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))
```

(continues on next page)

(continued from previous page)

```

>>> 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
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

```

2.1 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** – Python callable with NumPyro primitives.
- **hide_fn** – function which when given a dictionary containing site-level metadata returns whether it should be blocked.

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.))

```

(continues on next page)

(continued from previous page)

```

...     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)
```

2.2 collapse

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

Bases: `numpyro.handlers.trace`

EXPERIMENTAL Collapses all sites in the context by lazily sampling and attempting to use conjugacy relations. If no conjugacy is known this will fail. Code using the results of sample sites must be written to accept Funsors rather than Tensors. This requires `functor` to be installed.

```
process_message(msg)
```

2.3 condition

```
class condition(fn=None, data=None, condition_fn=None)
```

Bases: `numpyro.primitives.Messenger`

Conditions unobserved sample sites to values from `data` or `condition_fn`. Similar to `substitute` except that it only affects `sample` sites and changes the `is_observed` property to `True`.

Parameters

- **fn** – Python callable with NumPyro primitives.
- **data** (*dict*) – dictionary of `numpy.ndarray` values keyed by site names.
- **condition_fn** – callable that takes in a site dict and returns a numpy array or `None` (in which case the handler has no side effect).

Example:

```

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

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

>>> 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)
```

2.4 do

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

Bases: `numpyro.primitives.Messenger`

Given a stochastic function with some sample statements and a dictionary of values at names, set the return values of those sites equal to the values as if they were hard-coded to those values and introduce fresh sample sites with the same names whose values do not propagate. Composes freely with `condition()` to represent counterfactual distributions over potential outcomes. See Single World Intervention Graphs [1] for additional details and theory.

This is equivalent to replacing `z = numpyro.sample("z", ...)` with `z = 1.` and introducing a fresh sample site `numpyro.sample("z", ...)` whose value is not used elsewhere.

References

[1] *Single World Intervention Graphs: A Primer*, Thomas Richardson, James Robins

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **data** – a dict mapping sample site names to interventions

Example:

```
>>> import jax.numpy as jnp
>>> import numpyro
>>> from numpyro.handlers import do, trace, seed
>>> import numpyro.distributions as dist
>>> def model(x):
...     s = numpyro.sample("s", dist.LogNormal())
...     z = numpyro.sample("z", dist.Normal(x, s))
...     return z ** 2
>>> intervened_model = handlers.do(model, data={"z": 1.})
>>> with trace() as exec_trace:
...     z_square = seed(intervened_model, 0)(1)
>>> assert exec_trace['z']['value'] != 1.
>>> assert not exec_trace['z']['is_observed']
>>> assert not exec_trace['z'].get('stop', None)
>>> assert z_square == 1
```

```
process_message(msg)
```

2.5 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*)

2.6 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*)

2.7 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*)

2.8 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*)

2.9 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*) – a positive scaling factor

process_message (*msg*)

2.10 scope

class `scope` (*fn=None, prefix=""*)

Bases: `numpyro.primitives.Messenger`

This handler prepend a prefix followed by a / 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"):
...             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

process_message (*msg*)

2.11 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():
```

(continues on next page)

(continued from previous page)

```

...     return numpyro.sample('y', dist.Normal(0., 1.))

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

```

```
process_message(msg)
```

2.12 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)
```

2.13 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

>>> 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 0x7f9e689bleb8>,
               '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.

3.1 Distribution

class Distribution (*batch_shape=()*, *event_shape=()*, *validate_args=None*)

Bases: `object`

Base class for probability distributions in NumPyro. The design largely follows from `torch.distributions`.

Parameters

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

As an example:

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

```
arg_constraints = {}
```

```
support = None
```

```
has_enumerate_support = False
```

```
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`

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`

3.2 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.

    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)
```

3.3 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()`

3.4 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.

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

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

Parameters

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

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

Return type `numpy.ndarray`

log_prob (*value*)

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

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

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

Return type `numpy.ndarray`

expand (*batch_shape*)

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

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

Returns an instance of `ExpandedDistribution`.

Return type `ExpandedDistribution`

tree_flatten ()

classmethod tree_unflatten (*aux_data*, *params*)

3.5 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.

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 $len(support) \times batch_shape$ containing all values in the support.

mean

Mean of the distribution.

variance

Variance of the distribution.

tree_flatten ()

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

3.6 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 = {}

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 ()

3.7 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`

4.1 Beta

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

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'concentration0': *<numpyro.distributions.constraints._GreaterThan object>*

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.

4.2 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.
```

4.3 Chi2

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

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

4.4 Dirichlet

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

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

4.5 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.

4.6 Gamma

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

Bases: `numpyro.distributions.distribution.Distribution`

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

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

reparametrized_params = ['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.

4.7 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.

4.8 GaussianRandomWalk

class GaussianRandomWalk (*scale*=1.0, *num_steps*=1, *validate_args*=None)

Bases: `numpyro.distributions.distribution.Distribution`

```

arg_constraints = {'num_steps': <numpyro.distributions.constraints._IntegerGreaterThan object>
support = <numpyro.distributions.constraints._RealVector 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)

```

4.9 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.

4.10 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.

4.11 InverseGamma

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

Bases: `numpyro.distributions.distribution.TransformedDistribution`

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

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

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

reparametrized_params = ['rate']

mean

Mean of the distribution.

variance

Variance of the distribution.

tree_flatten ()

4.12 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.

4.13 LKJ

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

Bases: `numpyro.distributions.distribution.TransformedDistribution`

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

When *concentration* > 1, the distribution favors samples with large large 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 η)
- **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 object>
support = <numpyro.distributions.constraints._CorrMatrix object>
mean
    Mean of the distribution.
tree_flatten()
classmethod tree_unflatten(aux_data, params)
```

4.14 LKJCholesky

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

Bases: `numpyro.distributions.distribution.Distribution`

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

When `concentration > 1`, the distribution favors samples with large diagonal entries (hence large 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 η)
- **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 object>
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)
```

4.15 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()
```

4.16 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', 'real']
```

```
sample(key, sample_shape=())
```

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

Parameters

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

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

Return type `numpy.ndarray`

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

mean

Mean of the distribution.

variance

Variance of the distribution.

4.17 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._PositiveD
    support = <numpyro.distributions.constraints._RealVector 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)
```

4.18 LowRankMultivariateNormal

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

    arg_constraints = {'cov_diag':  <numpyro.distributions.constraints._GreaterThan object>
    support = <numpyro.distributions.constraints._RealVector object>

    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 ()

4.19 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._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.

4.20 Pareto

class Pareto (*scale*, *alpha*, *validate_args=None*)

Bases: `numpyro.distributions.distribution.TransformedDistribution`

arg_constraints = {'alpha': `<numpyro.distributions.constraints._GreaterThan object>`,

mean
Mean of the distribution.

variance
Variance of the distribution.

support

tree_flatten()

4.21 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 = ['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.
```

4.22 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': <numpyro.distributions.constraints._Real object>, 'scale': <numpyro.distributions.constraints._Real object>}
    reparametrized_params = ['low', 'loc', 'scale']
    support

    mean
        Mean of the distribution.

    variance
        Variance of the distribution.
```

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

4.23 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': <
    reparametrized_params = ['low', 'loc', 'scale']
    support
    mean
        Mean of the distribution.
    variance
        Variance of the distribution.
    tree_flatten()
    classmethod tree_unflatten(aux_data, params)
```

4.24 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)
```

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

5.1 Bernoulli

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

5.2 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 $\text{len}(\text{support}) \times \text{batch_shape}$ containing all values in the support.

5.3 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*)

mean

Mean of the distribution.

variance

Variance of the distribution.

enumerate_support (*expand=True*)

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

5.4 BetaBinomial

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

Bases: `numpyro.distributions.distribution.Distribution`

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

Parameters

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

arg_constraints = {'concentration0': <numpyro.distributions.constraints._GreaterThan

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*)

mean

Mean of the distribution.

variance

Variance of the distribution.

support

5.5 Binomial

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

5.6 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

5.7 BinomialProbs

class BinomialProbs (*probs*, *total_count*=1, *validate_args*=None)

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'probs': <numpyro.distributions.constraints._Interval object>, 'to

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*)

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.

5.8 Categorical

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

5.9 CategoricalLogits

class CategoricalLogits (*logits, validate_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

arg_constraints = {'logits': `<numpyro.distributions.constraints._RealVector 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.

support

enumerate_support (*expand=True*)

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

5.10 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 *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

enumerate_support (*expand=True*)

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

5.11 Delta

class Delta (*v=0.0*, *log_density=0.0*, *event_dim=0*, *validate_args=None*, *value=None*)

Bases: `numpyro.distributions.distribution.Distribution`

arg_constraints = {'log_density': <numpyro.distributions.constraints._Real object>, 'value': <numpyro.distributions.constraints._Real object>}

support = <numpyro.distributions.constraints._Real 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.

tree_flatten ()

`classmethod tree_unflatten (aux_data, params)`

5.12 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 of`

`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

5.13 GammaPoisson

class `GammaPoisson (concentration, rate=1.0, validate_args=None)`

Bases: `numpyro.distributions.distribution.Distribution`

Compound distribution comprising of a gamma-poisson pair, also referred to as a gamma-poisson mixture. The `rate` parameter for the Poisson distribution is unknown and randomly drawn from a Gamma distribution.

Parameters

- **concentration** (`numpy.ndarray`) – shape parameter (alpha) of the Gamma distribution.
- **rate** (`numpy.ndarray`) – rate parameter (beta) for the Gamma distribution.

```
arg_constraints = {'concentration': <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.

5.14 Geometric

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

5.15 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.

5.16 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)

    mean
        Mean of the distribution.

    variance
        Variance of the distribution.
```

5.17 Multinomial

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

5.18 MultinomialLogits

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

    arg_constraints = {'logits': <numpyro.distributions.constraints._RealVector 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*)

probs

mean

Mean of the distribution.

variance

Variance of the distribution.

support

5.19 MultinomialProbs

class MultinomialProbs (*probs, total_count=1, validate_args=None*)

Bases: `numpyro.distributions.distribution.Distribution`

arg_constraints = {'probs': <numpyro.distributions.constraints._Simplex object>, 'tot

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

5.20 OrderedLogistic

class OrderedLogistic (*predictor, cutpoints, validate_args=None*)

Bases: `numpyro.distributions.discrete.CategoricalProbs`

A categorical distribution with ordered outcomes.

References:

1. *Stan Functions Reference*, v2.20 section 12.6, Stan Development Team

Parameters

- **predictor** (*numpy.ndarray*) – prediction in real domain; typically this is output of a linear model.
- **cutpoints** (*numpy.ndarray*) – positions in real domain to separate categories.

```
arg_constraints = {'cutpoints': <numpyro.distributions.constraints._OrderedVector object>
```

5.21 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.

5.22 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`

5.23 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': `<numpyro.distributions.constraints._IntegerGreaterThanOrEqualTo object>`}

support = `<numpyro.distributions.constraints._IntegerGreaterThanOrEqualTo 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

6.1 VonMises

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

Bases: *numpyro.distributions.distribution.Distribution*

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

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

sample (*key, sample_shape=()*)

Generate sample from von Mises distribution

Parameters

- **sample_shape** – shape of samples
- **key** – random number generator key

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

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

7.1 BijectorConstraint

class `BijectorConstraint` (*bijector*)

A constraint which is codomain of a TensorFlow bijector.

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

7.2 BijectorTransform

class `BijectorTransform` (*bijector*)

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

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

7.3 TFPDistributionMixin

class `TFPDistributionMixin` (*batch_shape=()*, *event_shape=()*, *validate_args=None*)

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

7.4 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.

7.5 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.
```

7.6 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.
```

7.7 Bernoulli

```
class Bernoulli (logits=None, probs=None, dtype=<class 'jax.numpy.lax_numpy.int32'>, vali-
                 date_args=False, allow_nan_stats=True, name='Bernoulli')
    Wraps      tensorflow_probability.substrates.jax.distributions.bernoulli.Bernoulli    with
    TFPDistributionMixin.
```

7.8 Beta

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

7.9 BetaBinomial

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

7.10 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.
```

7.11 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.
```

7.12 Categorical

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

7.13 Cauchy

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

7.14 Chi

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

7.15 Chi2

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

7.16 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.
```

7.17 ContinuousBernoulli

```
class ContinuousBernoulli (logits=None, probs=None, lims=(0.499, 0.501), dtype=<class
                           'jax.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.
```

7.18 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.
```

7.19 Dirichlet

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

7.20 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.
```

7.21 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.
```

7.22 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.
```

7.23 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.
```

7.24 ExpInverseGamma

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

7.25 ExpRelaxedOneHotCategorical

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

7.26 Exponential

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

7.27 FiniteDiscrete

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

7.28 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.
```

7.29 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.
```

7.30 GaussianProcess

```
class GaussianProcess(kernel,          index_points=None,          mean_fn=None,          observa-
                        tion_noise_variance=0.0,          jitter=1e-06,          validate_args=False,          al-
                        low_nan_stats=False, name='GaussianProcess')
Wraps      tensorflow_probability.substrates.jax.distributions.gaussian_process.GaussianProcess      with
TFPDistributionMixin.
```

7.31 GaussianProcessRegressionModel

```
class GaussianProcessRegressionModel(kernel,          index_points=None,          observa-
                                      tion_index_points=None,          observations=None,
                                      observation_noise_variance=0.0,          predic-
                                      tive_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.
```

7.32 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.
```

7.33 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.
```

7.34 Geometric

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

7.35 Gumbel

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

7.36 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.
```

7.37 HalfNormal

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

7.38 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.
```

7.39 HiddenMarkovModel

```
class HiddenMarkovModel (initial_distribution,      transition_distribution,      observa-
                        tion_distribution,      num_steps,      validate_args=False,      al-
                        low_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.
```

7.40 Horseshoe

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

7.41 Independent

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

7.42 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.
```

7.43 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.
```

7.44 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.
```

7.45 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.
```

7.46 JointDistributionCoroutine

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

7.47 JointDistributionCoroutineAutoBatched

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

7.48 JointDistributionNamed

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

7.49 JointDistributionNamedAutoBatched

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

7.50 JointDistributionSequential

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

7.51 JointDistributionSequentialAutoBatched

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

7.52 Kumaraswamy

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

7.53 LKJ

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

7.54 Laplace

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

7.55 LinearGaussianStateSpaceModel

class LinearGaussianStateSpaceModel (*num_timesteps, transition_matrix, transition_noise, observation_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*.

7.56 LogLogistic

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

7.57 LogNormal

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

7.58 Logistic

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

7.59 LogitNormal

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

7.60 MixtureSameFamily

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

7.61 Moyal

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

7.62 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*.

7.63 MultivariateNormalDiag

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

7.64 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*.

7.65 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*.

7.66 MultivariateNormalLinearOperator

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

7.67 MultivariateNormalTriL

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

7.68 MultivariateStudentTLinearOperator

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

7.69 NegativeBinomial

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

7.70 Normal

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

7.71 OneHotCategorical

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

7.72 OrderedLogistic

```
class OrderedLogistic (cutpoints,  loc,  dtype=<class 'jax.numpy.lax_numpy.int32'>,  vali-
                      date_args=False, allow_nan_stats=True, name='OrderedLogistic')
    Wraps      tensorflow_probability.substrates.jax.distributions.ordered_logistic.OrderedLogistic      with
    TFPDistributionMixin.
```

7.73 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.
```

7.74 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.
```

7.75 PlackettLuce

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

7.76 Poisson

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

7.77 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.
```

7.78 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.
```

7.79 ProbitBernoulli

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

7.80 QuantizedDistribution

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

7.81 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.
```

7.82 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.
```

7.83 Sample

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

7.84 SinhArcsinh

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

7.85 SphericalUniform

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

TFPDistributionMixin.

7.86 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.
```

7.87 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.
```

7.88 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.
```

7.89 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.
```

7.90 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.
```

7.91 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.
```

7.92 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.
```

7.93 VariationalGaussianProcess

```
class VariationalGaussianProcess(kernel,          index_points,          inducing_index_points,
                                variational_inducing_observations_loc,          varia-
                                tional_inducing_observations_scale,          mean_fn=None,
                                observation_noise_variance=None,          predic-
                                tive_noise_variance=None,          jitter=1e-06,          val-
                                idate_args=False,          allow_nan_stats=False,
                                name='VariationalGaussianProcess')
    Wraps tensorflow_probability.substrates.jax.distributions.variational_gaussian_process.VariationalGaussianProcess
    with TFPDistributionMixin.
```

7.94 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.
```

7.95 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.
```

7.96 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.
```

7.97 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.
```

7.98 Weibull

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

7.99 WishartLinearOperator

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

7.100 WishartTriL

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

8.1 Constraint

class `Constraint`

Bases: `object`

Abstract base class for constraints.

A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

check (*value*)

Returns a byte tensor of *sample_shape* + *batch_shape* indicating whether each event in value satisfies this constraint.

8.2 boolean

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

8.3 corr_cholesky

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

8.4 corr_matrix

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

8.5 dependent

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

8.6 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.

8.7 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.

8.8 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.

8.9 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.

8.10 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.

8.11 lower_cholesky

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

8.12 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.

8.13 nonnegative_integer

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

8.14 ordered_vector

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

8.15 positive

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

8.16 positive_definite

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

8.17 positive_integer

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

8.18 real

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

8.19 real_vector

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

8.20 simplex

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

8.21 unit_interval

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

9.1 `biject_to`

`biject_to`(*constraint*)

9.2 Transform

```
class Transform
    Bases: object
    domain = <numpyro.distributions.constraints._Real object>
    codomain = <numpyro.distributions.constraints._Real object>
    event_dim = 0
    inv(y)
    log_abs_det_jacobian(x, y, intermediates=None)
    call_with_intermediates(x)
```

9.3 AbsTransform

```
class AbsTransform
    Bases: numpyro.distributions.transforms.Transform
    domain = <numpyro.distributions.constraints._Real object>
    codomain = <numpyro.distributions.constraints._GreaterThan object>
    inv(y)
```

9.4 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

event_dim

`int([x]) -> integer` `int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If *x* is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

inv(*y*)

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

9.5 ComposeTransform

class `ComposeTransform`(*parts*)

Bases: `numpyro.distributions.transforms.Transform`

domain

codomain

event_dim

`int([x]) -> integer` `int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If *x* is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

inv(*y*)

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

call_with_intermediates(*x*)

9.6 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._RealVector object>
codomain = <numpyro.distributions.constraints._CorrCholesky object>
event_dim = 2
inv(y)
log_abs_det_jacobian(x, y, intermediates=None)
```

9.7 ExpTransform

```
class ExpTransform(domain=<numpyro.distributions.constraints._Real object>)
    Bases: numpyro.distributions.transforms.Transform
    codomain
    inv(y)
    log_abs_det_jacobian(x, y, intermediates=None)
```

9.8 IdentityTransform

```
class IdentityTransform(event_dim=0)
    Bases: numpyro.distributions.transforms.Transform
    inv(y)
    log_abs_det_jacobian(x, y, intermediates=None)
```

9.9 InvCholeskyTransform

```
class InvCholeskyTransform(domain=<numpyro.distributions.constraints._LowerCholesky    ob-
                           ject>)
    Bases: numpyro.distributions.transforms.Transform
    Transform via the mapping  $y = x @ x.T$ , where  $x$  is a lower triangular matrix with positive diagonal.
    event_dim = 2
    codomain
    inv(y)
    log_abs_det_jacobian(x, y, intermediates=None)
```

9.10 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._RealVector object>
    codomain = <numpyro.distributions.constraints._RealVector object>
    event_dim = 1
    inv(y)
    log_abs_det_jacobian(x, y, intermediates=None)
```

9.11 LowerCholeskyTransform

```
class LowerCholeskyTransform
    Bases: numpyro.distributions.transforms.Transform
    domain = <numpyro.distributions.constraints._RealVector object>
    codomain = <numpyro.distributions.constraints._LowerCholesky object>
    event_dim = 2
    inv(y)
    log_abs_det_jacobian(x, y, intermediates=None)
```

9.12 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._RealVector object>`

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

`event_dim = 1`

`inv(y)`

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

9.13 PermuteTransform

class PermuteTransform(*permutation*)

Bases: `numpyro.distributions.transforms.Transform`

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

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

`event_dim = 1`

`inv(y)`

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

9.14 PowerTransform

class PowerTransform(*exponent*)

Bases: `numpyro.distributions.transforms.Transform`

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

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

`inv(y)`

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

9.15 SigmoidTransform

class SigmoidTransform

Bases: `numpyro.distributions.transforms.Transform`

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

`inv(y)`

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

9.16 StickBreakingTransform

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

    domain = <numpyro.distributions.constraints._RealVector object>
    codomain = <numpyro.distributions.constraints._Simplex object>
    event_dim = 1

    inv(y)

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

10.1 InverseAutoregressiveTransform

class InverseAutoregressiveTransform (*autoregressive_nn*, *log_scale_min_clip=-5.0*,
log_scale_max_clip=3.0)
 Bases: *numpyro.distributions.transforms.Transform*

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

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

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

References

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

domain = `<numpyro.distributions.constraints._RealVector object>`
codomain = `<numpyro.distributions.constraints._RealVector object>`
event_dim = 1
call_with_intermediates (*x*)
inv (*y*)

Parameters *y* (*numpy.ndarray*) – the output of the transform to be inverted

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

10.2 BlockNeuralAutoregressiveTransform

class BlockNeuralAutoregressiveTransform(*bn_arn*)

Bases: *numpyro.distributions.transforms.Transform*

An implementation of Block Neural Autoregressive flow.

References

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

event_dim = 1

call_with_intermediates (*x*)

inv (*y*)

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

Markov Chain Monte Carlo (MCMC)

```
class MCMC (sampler, num_warmup, num_samples, num_chains=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.
- **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.

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

Run the MCMC warmup adaptation phase. After this call, 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.mcmc.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 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`)

11.1 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", ["z", "rng_key"])

>>> class MetropolisHastings(numpyro.infer.mcmc.MCMCKernel):
...     sample_field = "z"
...
...     def __init__(self, potential_fn, step_size=0.1):
...         self.potential_fn = potential_fn
...         self.step_size = step_size
...
...     def init(self, rng_key, num_warmup, init_params, model_args, model_
↳kwargs):
...         return MHState(init_params, rng_key)
...
...     def sample(self, state, model_args, model_kwargs):
...         z, rng_key = state
...         rng_key, key_proposal, key_accept = random.split(rng_key, 3)
...         z_proposal = dist.Normal(z, self.step_size).sample(key_proposal)
...         accept_prob = jnp.exp(self.potential_fn(z) - self.potential_fn(z_
↳proposal))
...         z_new = jnp.where(dist.Uniform().sample(key_accept) < accept_prob, z_
↳proposal, z)
...         return MHState(z_new, rng_key)

>>> def f(x):
```

(continues on next page)

(continued from previous page)

```

...     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()

```

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, tra-  
          jectory_length=6.283185307179586, init_strategy=<function init_to_uniform>,  
          find_heuristic_step_size=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 2π .
- **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.

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*)

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.

```
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))
>>> dim = 3
>>> labels = dist.Bernoulli(logits=(true_coefs * data).sum(-1)).sample(random.
↳ PRNGKey(3))
>>>
>>> def model(data, labels):
```

(continues on next page)

(continued from previous page)

```

...     coefs_mean = jnp.zeros(dim)
...     coefs = numpyro.sample('beta', dist.Normal(coefs_mean, jnp.ones(3)))
...     intercept = numpyro.sample('intercept', dist.Normal(0., 10.))
...     return numpyro.sample('y', dist.Bernoulli(logits=(coefs * data +
↪ intercept).sum(-1)), obs=labels)
>>>
>>> model_info = initialize_model(random.PRNGKey(0), model, model_args=(data,
↪ labels,))
>>> init_kernel, sample_kernel = hmc(model_info.potential_fn, algo='NUTS')
>>> hmc_state = init_kernel(model_info.param_info,
...                         trajectory_length=10,
...                         num_warmup=300)
>>> samples = fori_collect(0, 500, sample_kernel, hmc_state,
...                        transform=lambda state: model_info.postprocess_
↪ fn(state.z))
>>> print(jnp.mean(samples['beta'], 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*,
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 2π .
- **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).
- **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.

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.

11.2 TensorFlow Kernels

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

11.2.1 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.

11.2.2 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`.

11.2.3 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`.

11.2.4 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`.

11.2.5 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`.

11.2.6 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`.

11.2.7 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`.

11.2.8 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`.

11.2.9 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`.

11.2.10 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`.

11.3 MCMC Utilities

```
initialize_model (rng_key, model, init_strategy=<function init_to_uniform>, dynamic_args=False,
                  model_args=(), model_kwargs=None)
(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.

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, ***progbar_opts*)

This looping construct works like `fori_loop()` but with the additional effect of collecting values from the loop body. In addition, this allows for post-processing of these samples via *transform*, and progress bar updates. Note that, *progbar=False* will be faster, especially when collecting a lot of samples. Refer to example usage in `hmc()`.

Parameters

- **lower** (*int*) – the index to start the collective work. In other words, we will skip collecting the first *lower* values.
- **upper** (*int*) – number of times to run the loop body.
- **body_fun** – a callable that takes a collection of *np.ndarray* and returns a collection with the same shape and *dtype*.
- **init_val** – initial value to pass as argument to *body_fun*. Can be any Python collection type containing *np.ndarray* objects.
- **transform** – a callable to post-process the values returned by *body_fn*.
- **progbar** – whether to post progress bar updates.
- **return_last_val** (*bool*) – If *True*, the last value is also returned. This has the same type as *init_val*.
- **collection_size** (*int*) – Size of the returned collection. If not specified, the size will be `upper - lower`. If the size is larger than `upper - lower`, only the top `upper - lower` 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.

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)

Example:

```
>>> from jax import lax, 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", 15., 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())
>>> init_state = svi.init(random.PRNGKey(0), data)
>>> state = lax.fori_loop(0, 2000, lambda i, state: svi.update(state, data)[0],
↳ init_state)
>>> # or to collect losses during the loop
```

(continues on next page)

(continued from previous page)

```
>>> # state, losses = lax.scan(lambda state, i: svi.update(state, data), init_
↳ state, jnp.arange(2000))
>>> params = svi.get_params(state)
>>> 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*)

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 tuple containing initial `SVIState`, and *get_params*, a callable that transforms unconstrained parameter values from the optimizer to the specified constrained domain

get_params (*svi_state*)

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

Parameters **svi_state** – current state of the optimizer.

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)*.

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`).

12.1 ELBO

class `ELBO` (`num_particles=1`)

Bases: `numpyro.infer.elbo.Trace_ELBO`

12.2 RenyiELBO

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

Bases: `numpyro.infer.elbo.Trace_ELBO`

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

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

Parameters

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

References:

1. *Renyi Divergence Variational Inference*, Yingzhen Li, Richard E. Turner
2. *Importance Weighted Autoencoders*, Yuri Burda, Roger Grosse, Ruslan Salakhutdinov

loss (`rng_key`, `param_map`, `model`, `guide`, `*args`, `**kwargs`)

Evaluates the 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.

12.3 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.

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.

13.1 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*

13.2 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*.

13.3 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*

13.4 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`

13.5 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*.

13.6 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*

13.7 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*

13.8 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*)

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 ReparamBases: `abc.ABC`

Base class for reparameterizers.

14.1 Loc-Scale Decentering

class LocScaleReparam (*centered=None, shape_params=()*)Bases: `numpyro.infer.reparam.Reparam`

Generic decentering reparameterizer [1] for latent variables parameterized by `loc` and `scale` (and possibly additional `shape_params`).

This reparameterization works only for latent variables, not likelihoods.

References:

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

Parameters

- **centered** (*float*) – optional centered parameter. If `None` (default) learn a per-site per-element centering parameter in `[0, 1]`. If 0, fully decenter the distribution; if 1, preserve the centered distribution unchanged.
- **shape_params** (*tuple or list*) – list of additional parameter names to copy unchanged from the centered to decentered distribution.

`__call__` (*name, fn, obs*)

Parameters

- **name** (*str*) – A sample site name.
- **fn** (*Distribution*) – A distribution.
- **obs** (*numpy.ndarray*) – Observed value or None.

Returns A pair (*new_fn*, *value*).

14.2 Neural Transport

class `NeuTraReparam` (*guide*, *params*)

Bases: `numpyro.infer.reparam.Reparam`

Neural Transport reparameterizer [1] of multiple latent variables.

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

```
# Step 1. Train a guide
guide = AutoIAFNormal(model)
svi = SVI(model, guide, ...)
# ...train the guide...

# Step 2. Use trained guide in NeuTra MCMC
neutra = NeuTraReparam(guide)
model = 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`

14.3 Transformed Distributions

class `TransformReparam`

Bases: `numpyro.infer.reparam.Reparam`

Reparameterizer for `TransformedDistribution` latent variables.

This is useful for transformed distributions with complex, geometry-changing transforms, where the posterior has simple shape in the space of `base_dist`.

This reparameterization works only for latent variables, not likelihoods.

__call__ (*name, fn, obs*)

Parameters

- **name** (*str*) – A sample site name.
- **fn** (*Distribution*) – A distribution.
- **obs** (*numpy.ndarray*) – Observed value or None.

Returns A pair (*new_fn, value*).

15.1 Effect handlers

class `enum` (*fn=None, first_available_dim=None*)

Bases: `numpyro.contrib.funsor.enum_messenger.BaseEnumMessenger`

Enumerates in parallel over discrete sample sites marked `infer={"enumerate": "parallel"}`.

Parameters

- **fn** (*callable*) – Python callable with NumPyro primitives.
- **first_available_dim** (*int*) – The first tensor dimension (counting from the right) that is available for parallel enumeration. This dimension and all dimensions left may be used internally by Pyro. This should be a negative integer or None.

process_message (*msg*)

class `infer_config` (*fn, config_fn*)

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 Pyro primitive calls)
- **config_fn** – a callable taking a site and returning an `infer` dict

process_message (*msg*)

class `markov`

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*)

class to_data

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*.

class to_functor

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

Parameters

- **x** – An object.
- **output** (*functor.domains.Domain*) – An optional output hint to uniquely convert a data to a `Functor` (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*)

15.2 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)
```

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.

16.1 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.

16.2 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.

16.3 ClippedAdam

class ClippedAdam (**args, clip_norm=10.0, **kwargs*)

Adam optimizer with gradient clipping.

Parameters `clip_norm` (*float*) – All gradient values will be clipped between $[-clip_norm, clip_norm]$.

Reference:

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

eval_and_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.

16.4 Minimize

class Minimize (*method='BFGS', **kwargs*)

Wrapper class for the JAX minimizer: `minimize()`.

Example:

```
>>> from numpy.testing import assert_allclose
>>> from jax import random
>>> import jax.numpy as jnp
>>> import numpyro
>>> import numpyro.distributions as dist
>>> from numpyro.infer import SVI, Trace_ELBO
>>> from numpyro.infer.autoguide import AutoLaplaceApproximation

>>> def model(x, y):
```

(continues on next page)

(continued from previous page)

```

...     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.

16.5 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.

16.6 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.

16.7 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.

16.8 SGD

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

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

eval_and_update (*fn: Callable, state: Tuple[int, _OptState]*) $\rightarrow$ `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]*) $\rightarrow$ `_Params`

Get current parameter values.

Parameters **state** – current optimizer state.

Returns collection with current value for parameters.

init (*params: _Params*) $\rightarrow$ `Tuple[int, _OptState]`

Initialize the optimizer with parameters designated to be optimized.

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

Returns initial optimizer state.

update (*g: _Params, state: Tuple[int, _OptState]*) $\rightarrow$ `Tuple[int, _OptState]`

Gradient update for the optimizer.

Parameters

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

Returns new optimizer state after the update.

16.9 SM3

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

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

eval_and_update (*fn: Callable, state: Tuple[int, _OptState]*) $\rightarrow$ `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.

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

17.1 Autocorrelation

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

Computes the autocorrelation of samples at dimension *axis*.

Parameters

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

Returns autocorrelation of *x*.

Return type *numpy.ndarray*

17.2 Autocovariance

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

Computes the autocovariance of samples at dimension *axis*.

Parameters

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

Returns autocovariance of *x*.

Return type *numpy.ndarray*

17.3 Effective Sample Size

`effective_sample_size(x)`

Computes effective sample size of input x , where the first dimension of x is chain dimension and the second dimension of x is draw dimension.

References:

1. *Introduction to Markov Chain Monte Carlo*, Charles J. Geyer
2. *Stan Reference Manual version 2.18*, Stan Development Team

Parameters x (`numpy.ndarray`) – the input array.

Returns effective sample size of x .

Return type `numpy.ndarray`

17.4 Gelman Rubin

`gelman_rubin(x)`

Computes R-hat over chains of samples x , where the first dimension of x is chain dimension and the second dimension of x is draw dimension. It is required that $x.shape[0] \geq 2$ and $x.shape[1] \geq 2$.

Parameters x (`numpy.ndarray`) – the input array.

Returns R-hat of x .

Return type `numpy.ndarray`

17.5 Split Gelman Rubin

`split_gelman_rubin(x)`

Computes split R-hat over chains of samples x , where the first dimension of x is chain dimension and the second dimension of x is draw dimension. It is required that $x.shape[1] \geq 4$.

Parameters x (`numpy.ndarray`) – the input array.

Returns split R-hat of x .

Return type `numpy.ndarray`

17.6 HPDI

`hpdi(x, prob=0.9, axis=0)`

Computes “highest posterior density interval” (HPDI) which is the narrowest interval with probability mass `prob`.

Parameters

- x (`numpy.ndarray`) – the input array.
- `prob` (`float`) – the probability mass of samples within the interval.
- `axis` (`int`) – the dimension to calculate hpdi.

Returns quantiles of x at $(1 - \text{prob}) / 2$ and $(1 + \text{prob}) / 2$.

Return type `numpy.ndarray`

17.7 Summary

summary (*samples*, *prob*=0.9, *group_by_chain*=True)

Returns a summary table displaying diagnostics of *samples* from the posterior. The diagnostics displayed are mean, standard deviation, median, the 90% Credibility Interval `hpdi()`, `effective_sample_size()`, and `split_gelman_rubin()`.

Parameters

- **samples** (*dict* or *numpy.ndarray*) – a collection of input samples with left most dimension is chain dimension and second to left most dimension is draw dimension.
- **prob** (*float*) – the probability mass of samples within the HPDI interval.
- **group_by_chain** (*bool*) – If True, each variable in *samples* will be treated as having shape *num_chains* $\times$ *num_samples* $\times$ *sample_shape*. Otherwise, the corresponding shape will be *num_samples* $\times$ *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* $\times$ *num_samples* $\times$ *sample_shape*. Otherwise, the corresponding shape will be *num_samples* $\times$ *sample_shape* (i.e. without chain dimension).

18.1 `enable_validation`

`enable_validation` (*is_validate=True*)

Enable or disable validation checks in NumPyro. Validation checks provide useful warnings and errors, e.g. NaN checks, validating distribution arguments and support values, etc. which is useful for debugging.

Note: This utility does not take effect under JAX's JIT compilation or vectorized transformation `jax.vmap()`.

Parameters `is_validate` (*bool*) – whether to enable validation checks.

18.2 `validation_enabled`

`validation_enabled` (*is_validate=True*)

Context manager that is useful when temporarily enabling/disabling validation checks.

Parameters `is_validate` (*bool*) – whether to enable validation checks.

18.3 `enable_x64`

`enable_x64` (*use_x64=True*)

Changes the default array type to use 64 bit precision as in NumPy.

Parameters `use_x64` (*bool*) – when *True*, JAX arrays will use 64 bits by default; else 32 bits.

18.4 set_platform

set_platform (*platform=None*)

Changes platform to CPU, GPU, or TPU. This utility only takes effect at the beginning of your program.

Parameters **platform** (*str*) – either ‘cpu’, ‘gpu’, or ‘tpu’.

18.5 set_host_device_count

set_host_device_count (*n*)

By default, XLA considers all CPU cores as one device. This utility tells XLA that there are *n* host (CPU) devices available to use. As a consequence, this allows parallel mapping in JAX `jax.pmap()` to work in CPU platform.

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

Warning: Our understanding of the side effects of using the `xla_force_host_platform_device_count` flag in XLA is incomplete. If you observe some strange phenomenon when using this utility, please let us know through our [issue](#) or forum page. More information is available in this [JAX issue](#).

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

19.1 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* $\times$...)
- set `batch_ndims=2` to get prediction for *posterior_samples* with shapes (*num_chains* $\times$ *N* $\times$...). Note that if *num_samples* argument is not None, its value should be equal to *num_chains* $\times$ *N*.

Returns dict of samples from the predictive distribution.

19.2 log_density

log_density (*model*, *model_args*, *model_kwargs*, *params*)

(EXPERIMENTAL INTERFACE) Computes log of joint density for the model given latent values *params*.

Parameters

- **model** – Python callable containing NumPyro primitives.
- **model_args** (*tuple*) – args provided to the model.
- **model_kwargs** (*dict*) – kwargs provided to the model.
- **params** (*dict*) – dictionary of current parameter values keyed by site name.

Returns log of joint density and a corresponding model trace

19.3 transform_fn

transform_fn (*transforms*, *params*, *invert=False*)

(EXPERIMENTAL INTERFACE) Callable that applies a transformation from the *transforms* dict to values in the *params* dict and returns the transformed values keyed on the same names.

Parameters

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

Returns *dict* of transformed params.

19.4 constrain_fn

constrain_fn (*model*, *model_args*, *model_kwargs*, *params*, *return_deterministic=False*)

(EXPERIMENTAL INTERFACE) Gets value at each latent site in *model* given unconstrained parameters *params*. The *transforms* is used to transform these unconstrained parameters to base values of the corresponding priors in *model*. If a prior is a transformed distribution, the corresponding base value lies in the support of base distribution. Otherwise, the base value lies in the support of the distribution.

Parameters

- **model** – a callable containing NumPyro primitives.
- **model_args** (*tuple*) – args provided to the model.

- **model_kwargs** (*dict*) – kwargs provided to the model.
- **params** (*dict*) – dictionary of unconstrained values keyed by site names.
- **return_deterministic** (*bool*) – whether to return the value of *deterministic* sites from the model. Defaults to *False*.

Returns *dict* of transformed params.

19.5 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.

19.6 log_likelihood

log_likelihood (*model*, *posterior_samples*, **args*, *parallel=False*, *batch_ndims=1*, ***kwargs*)

(EXPERIMENTAL INTERFACE) Returns log likelihood at observation nodes of model, given samples of all latent variables.

Parameters

- **model** – Python callable containing Pyro primitives.
- **posterior_samples** (*dict*) – dictionary of samples from the posterior.
- **args** – model arguments.
- **batch_ndims** – the number of batch dimensions in posterior samples. Some usages:
 - set *batch_ndims=0* to get 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* ...)
- **kwargs** – model kwargs.

Returns dict of log likelihoods at observation sites.

19.7 find_valid_initial_params

find_valid_initial_params (*rng_key*, *model*, *init_strategy*=<function *init_to_uniform*>, *enum*=False, *model_args*=(), *model_kwargs*=None, *prototype_params*=None)

(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.

19.8 Initialization Strategies

19.8.1 init_to_feasible

init_to_feasible (*site*=None)

Initialize to an arbitrary feasible point, ignoring distribution parameters.

19.8.2 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.

19.8.3 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.

19.8.4 `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.

19.8.5 `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.

19.9 Tensor Indexing

`vindex` (*tensor, args*)

Vectorized advanced indexing with broadcasting semantics.

See also the convenience wrapper `Vindex`.

This is useful for writing indexing code that is compatible with batching and enumeration, especially for selecting mixture components with discrete random variables.

For example suppose `x` is a parameter with `len(x.shape) == 3` and we wish to generalize the expression `x[i, :, j]` from integer `i, j` to tensors `i, j` with batch dims and enum dims (but no event dims). Then we can write the generalize version using `Vindex`

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

batch_shape = broadcast_shape(i.shape, j.shape)
event_shape = (x.size(1),)
assert xij.shape == batch_shape + event_shape
```

To handle the case when `x` may also contain batch dimensions (e.g. if `x` was sampled in a plated context as when using vectorized particles), `vindex()` uses the special convention that `Ellipsis` denotes batch dimensions (hence `...` can appear only on the left, never in the middle or in the right). Suppose `x` has event dim 3. Then we can write:

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

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

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

CHAPTER 20

Indices and tables

- `genindex`
- `search`

n

- `numpyro.contrib.functor`, 123
- `numpyro.contrib.indexing`, 145
- `numpyro.contrib.tfp.distributions`, 59
- `numpyro.contrib.tfp.mcmc`, 100
- `numpyro.diagnostics`, 135
- `numpyro.handlers`, 11
- `numpyro.infer.autoguide`, 109
- `numpyro.infer.reparam`, 119
- `numpyro.infer.util`, 141
- `numpyro.optim`, 127
- `numpyro.primitives`, 1
- `numpyro.util`, 139

Symbols

`__call__()` (*LocScaleReparam* method), 119
`__call__()` (*NeuTraReparam* method), 120
`__call__()` (*TransformReparam* method), 121

A

`AbsTransform` (class in `numpyro.distributions.transforms`), 81
`Adagrad` (class in `numpyro.optim`), 128
`Adam` (class in `numpyro.optim`), 127
`AffineTransform` (class in `numpyro.distributions.transforms`), 82
`arg_constraints` (*BernoulliLogits* attribute), 45
`arg_constraints` (*BernoulliProbs* attribute), 46
`arg_constraints` (*Beta* attribute), 31
`arg_constraints` (*BetaBinomial* attribute), 47
`arg_constraints` (*BinomialLogits* attribute), 47
`arg_constraints` (*BinomialProbs* attribute), 48
`arg_constraints` (*CategoricalLogits* attribute), 49
`arg_constraints` (*CategoricalProbs* attribute), 49
`arg_constraints` (*Cauchy* attribute), 32
`arg_constraints` (*Chi2* attribute), 32
`arg_constraints` (*Delta* attribute), 50
`arg_constraints` (*Dirichlet* attribute), 32
`arg_constraints` (*DirichletMultinomial* attribute), 51
`arg_constraints` (*Distribution* attribute), 21
`arg_constraints` (*ExpandedDistribution* attribute), 24
`arg_constraints` (*Exponential* attribute), 33
`arg_constraints` (*Gamma* attribute), 33
`arg_constraints` (*GammaPoisson* attribute), 52
`arg_constraints` (*GaussianRandomWalk* attribute), 34
`arg_constraints` (*GeometricLogits* attribute), 52
`arg_constraints` (*GeometricProbs* attribute), 53
`arg_constraints` (*Gumbel* attribute), 34
`arg_constraints` (*HalfCauchy* attribute), 35
`arg_constraints` (*HalfNormal* attribute), 36

`arg_constraints` (*ImproperUniform* attribute), 25
`arg_constraints` (*Independent* attribute), 26
`arg_constraints` (*InverseGamma* attribute), 36
`arg_constraints` (*Laplace* attribute), 37
`arg_constraints` (*LKJ* attribute), 38
`arg_constraints` (*LKJCholesky* attribute), 38
`arg_constraints` (*Logistic* attribute), 39
`arg_constraints` (*LogNormal* attribute), 39
`arg_constraints` (*LowRankMultivariateNormal* attribute), 40
`arg_constraints` (*MaskedDistribution* attribute), 27
`arg_constraints` (*MultinomialLogits* attribute), 53
`arg_constraints` (*MultinomialProbs* attribute), 54
`arg_constraints` (*MultivariateNormal* attribute), 40
`arg_constraints` (*Normal* attribute), 41
`arg_constraints` (*OrderedLogistic* attribute), 55
`arg_constraints` (*Pareto* attribute), 41
`arg_constraints` (*Poisson* attribute), 55
`arg_constraints` (*StudentT* attribute), 42
`arg_constraints` (*TransformedDistribution* attribute), 28
`arg_constraints` (*TruncatedCauchy* attribute), 42
`arg_constraints` (*TruncatedNormal* attribute), 43
`arg_constraints` (*TruncatedPolyaGamma* attribute), 43
`arg_constraints` (*Uniform* attribute), 44
`arg_constraints` (*Unit* attribute), 29
`arg_constraints` (*VonMises* attribute), 57
`arg_constraints` (*ZeroInflatedPoisson* attribute), 56
`AutoBNFNormal` (class in `numpyro.infer.autoguide`), 110
`AutoContinuous` (class in `numpyro.infer.autoguide`), 109
`autocorrelation()` (in module `numpyro.diagnostics`), 135
`autocovariance()` (in module `numpyro.diagnostics`), 135
`AutoDiagonalNormal` (class in `numpyro.infer.autoguide`), 111

AutoIAFNormal (class in `numpyro.infer.autoguide`), 113
 AutoLaplaceApproximation (class in `numpyro.infer.autoguide`), 114
 AutoLowRankMultivariateNormal (class in `numpyro.infer.autoguide`), 115
 AutoMultivariateNormal (class in `numpyro.infer.autoguide`), 112
 AutoNormal (class in `numpyro.infer.autoguide`), 116
 Autoregressive (class in `numpyro.contrib.tfp.distributions`), 59

B

batch_shape (Distribution attribute), 22
 BatchReshape (class in `numpyro.contrib.tfp.distributions`), 60
 Bates (class in `numpyro.contrib.tfp.distributions`), 60
 Bernoulli (class in `numpyro.contrib.tfp.distributions`), 60
 Bernoulli() (in module `numpyro.distributions.discrete`), 45
 BernoulliLogits (class in `numpyro.distributions.discrete`), 45
 BernoulliProbs (class in `numpyro.distributions.discrete`), 46
 Beta (class in `numpyro.contrib.tfp.distributions`), 60
 Beta (class in `numpyro.distributions.continuous`), 31
 BetaBinomial (class in `numpyro.contrib.tfp.distributions`), 60
 BetaBinomial (class in `numpyro.distributions.conjugate`), 46
 biject_to() (in module `numpyro.distributions.transforms`), 81
 BijectorConstraint (class in `numpyro.contrib.tfp.distributions`), 59
 BijectorTransform (class in `numpyro.contrib.tfp.distributions`), 59
 Binomial (class in `numpyro.contrib.tfp.distributions`), 60
 Binomial() (in module `numpyro.distributions.discrete`), 47
 BinomialLogits (class in `numpyro.distributions.discrete`), 47
 BinomialProbs (class in `numpyro.distributions.discrete`), 48
 block (class in `numpyro.handlers`), 12
 BlockNeuralAutoregressiveTransform (class in `numpyro.distributions.flows`), 88
 Blockwise (class in `numpyro.contrib.tfp.distributions`), 61
 boolean (in module `numpyro.distributions.constraints`), 77

C

call_with_intermediates() (BlockNeuralAutoregressiveTransform method), 88
 call_with_intermediates() (ComposeTransform method), 82
 call_with_intermediates() (InverseAutoregressiveTransform method), 87
 call_with_intermediates() (Transform method), 81
 Categorical (class in `numpyro.contrib.tfp.distributions`), 61
 Categorical() (in module `numpyro.distributions.discrete`), 49
 CategoricalLogits (class in `numpyro.distributions.discrete`), 49
 CategoricalProbs (class in `numpyro.distributions.discrete`), 49
 Cauchy (class in `numpyro.contrib.tfp.distributions`), 61
 Cauchy (class in `numpyro.distributions.continuous`), 32
 check() (Constraint method), 77
 Chi (class in `numpyro.contrib.tfp.distributions`), 61
 Chi2 (class in `numpyro.contrib.tfp.distributions`), 61
 Chi2 (class in `numpyro.distributions.continuous`), 32
 CholeskyLKJ (class in `numpyro.contrib.tfp.distributions`), 61
 ClippedAdam (class in `numpyro.optim`), 128
 codomain (AbsTransform attribute), 81
 codomain (AffineTransform attribute), 82
 codomain (ComposeTransform attribute), 82
 codomain (CorrCholeskyTransform attribute), 83
 codomain (ExpTransform attribute), 83
 codomain (InvCholeskyTransform attribute), 84
 codomain (InverseAutoregressiveTransform attribute), 87
 codomain (LowerCholeskyAffine attribute), 84
 codomain (LowerCholeskyTransform attribute), 84
 codomain (OrderedTransform attribute), 85
 codomain (PermuteTransform attribute), 85
 codomain (PowerTransform attribute), 85
 codomain (SigmoidTransform attribute), 85
 codomain (StickBreakingTransform attribute), 86
 codomain (Transform attribute), 81
 collapse (class in `numpyro.handlers`), 13
 ComposeTransform (class in `numpyro.distributions.transforms`), 82
 condition (class in `numpyro.handlers`), 13
 config_enumerate() (in module `numpyro.contrib.functor.infer_util`), 125
 consensus() (in module `numpyro.infer.hmc_util`), 103
 constrain_fn() (in module `numpyro.infer.util`), 142
 Constraint (class in `numpyro.distributions.constraints`), 77
 ContinuousBernoulli (class in `numpyro.contrib.tfp.distributions`), 61

- corr_cholesky (in module *numpyro.distributions.constraints*), 77
- corr_matrix (in module *numpyro.distributions.constraints*), 77
- CorrCholeskyTransform (class in *numpyro.distributions.transforms*), 82
- covariance_matrix (*LowRankMultivariateNormal* attribute), 40
- covariance_matrix (*MultivariateNormal* attribute), 40
- ## D
- default_fields (*HMC* attribute), 93
- default_fields (*MCMCKernel* attribute), 92
- default_fields (*SA* attribute), 96
- Delta (class in *numpyro.distributions.discrete*), 50
- dependent (in module *numpyro.distributions.constraints*), 78
- Deterministic (class in *numpyro.contrib.tfp.distributions*), 62
- deterministic() (in module *numpyro.primitives*), 3
- Dirichlet (class in *numpyro.contrib.tfp.distributions*), 62
- Dirichlet (class in *numpyro.distributions.continuous*), 32
- DirichletMultinomial (class in *numpyro.contrib.tfp.distributions*), 62
- DirichletMultinomial (class in *numpyro.distributions.conjugate*), 51
- Distribution (class in *numpyro.distributions.distribution*), 21
- do (class in *numpyro.handlers*), 14
- domain (*AbsTransform* attribute), 81
- domain (*ComposeTransform* attribute), 82
- domain (*CorrCholeskyTransform* attribute), 83
- domain (*InverseAutoregressiveTransform* attribute), 87
- domain (*LowerCholeskyAffine* attribute), 84
- domain (*LowerCholeskyTransform* attribute), 84
- domain (*OrderedTransform* attribute), 85
- domain (*PermuteTransform* attribute), 85
- domain (*PowerTransform* attribute), 85
- domain (*StickBreakingTransform* attribute), 86
- domain (*Transform* attribute), 81
- DoublesidedMaxwell (class in *numpyro.contrib.tfp.distributions*), 62
- ## E
- effective_sample_size() (in module *numpyro.diagnostics*), 136
- ELBO (class in *numpyro.infer.elbo*), 107
- Empirical (class in *numpyro.contrib.tfp.distributions*), 62
- enable_validation() (in module *numpyro.distributions.distribution*), 139
- enable_x64() (in module *numpyro.util*), 139
- entropy() (*LowRankMultivariateNormal* method), 41
- enum (class in *numpyro.contrib.functor.enum_messenger*), 123
- enumerate_support() (*BernoulliLogits* method), 46
- enumerate_support() (*BernoulliProbs* method), 46
- enumerate_support() (*BetaBinomial* method), 47
- enumerate_support() (*BinomialLogits* method), 47
- enumerate_support() (*BinomialProbs* method), 48
- enumerate_support() (*CategoricalLogits* method), 49
- enumerate_support() (*CategoricalProbs* method), 50
- enumerate_support() (*Distribution* method), 23
- enumerate_support() (*ExpandedDistribution* method), 24
- enumerate_support() (*MaskedDistribution* method), 28
- eval_and_update() (*Adagrad* method), 128
- eval_and_update() (*Adam* method), 127
- eval_and_update() (*ClippedAdam* method), 129
- eval_and_update() (*Minimize* method), 130
- eval_and_update() (*Momentum* method), 130
- eval_and_update() (*RMSProp* method), 131
- eval_and_update() (*RMSPropMomentum* method), 132
- eval_and_update() (*SGD* method), 133
- eval_and_update() (*SM3* method), 133
- evaluate() (*SVI* method), 106
- event_dim (*AffineTransform* attribute), 82
- event_dim (*BlockNeuralAutoregressiveTransform* attribute), 88
- event_dim (*ComposeTransform* attribute), 82
- event_dim (*CorrCholeskyTransform* attribute), 83
- event_dim (*Distribution* attribute), 22
- event_dim (*InvCholeskyTransform* attribute), 84
- event_dim (*InverseAutoregressiveTransform* attribute), 87
- event_dim (*LowerCholeskyAffine* attribute), 84
- event_dim (*LowerCholeskyTransform* attribute), 84
- event_dim (*OrderedTransform* attribute), 85
- event_dim (*PermuteTransform* attribute), 85
- event_dim (*StickBreakingTransform* attribute), 86
- event_dim (*Transform* attribute), 81
- event_shape (*Distribution* attribute), 22
- expand() (*Distribution* method), 23
- expand() (*Independent* method), 27
- expand_by() (*Distribution* method), 23
- ExpandedDistribution (class in *numpyro.distributions.distribution*), 24

- ExpGamma (class in `numpyro.contrib.tfp.distributions`), 62
- ExpInverseGamma (class in `numpyro.contrib.tfp.distributions`), 63
- Exponential (class in `numpyro.contrib.tfp.distributions`), 63
- Exponential (class in `numpyro.distributions.continuous`), 33
- ExpRelaxedOneHotCategorical (class in `numpyro.contrib.tfp.distributions`), 63
- ExpTransform (class in `numpyro.distributions.transforms`), 83
- ## F
- `factor()` (in module `numpyro.primitives`), 4
- `find_valid_initial_params()` (in module `numpyro.infer.util`), 144
- FiniteDiscrete (class in `numpyro.contrib.tfp.distributions`), 63
- `flax_module()` (in module `numpyro.contrib.module`), 4
- `fori_collect()` (in module `numpyro.util`), 103
- ## G
- Gamma (class in `numpyro.contrib.tfp.distributions`), 63
- Gamma (class in `numpyro.distributions.continuous`), 33
- GammaGamma (class in `numpyro.contrib.tfp.distributions`), 63
- GammaPoisson (class in `numpyro.distributions.conjugate`), 51
- GaussianProcess (class in `numpyro.contrib.tfp.distributions`), 64
- GaussianProcessRegressionModel (class in `numpyro.contrib.tfp.distributions`), 64
- GaussianRandomWalk (class in `numpyro.distributions.continuous`), 34
- `gelman_rubin()` (in module `numpyro.diagnostics`), 136
- GeneralizedNormal (class in `numpyro.contrib.tfp.distributions`), 64
- GeneralizedPareto (class in `numpyro.contrib.tfp.distributions`), 64
- Geometric (class in `numpyro.contrib.tfp.distributions`), 64
- Geometric() (in module `numpyro.distributions.discrete`), 52
- GeometricLogits (class in `numpyro.distributions.discrete`), 52
- GeometricProbs (class in `numpyro.distributions.discrete`), 53
- `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*), 114
- `get_base_dist()` (*AutoLowRankMultivariateNormal method*), 115
- `get_base_dist()` (*AutoMultivariateNormal method*), 112
- `get_diagnostics_str()` (*HMC method*), 93
- `get_diagnostics_str()` (*MCMCKernel method*), 92
- `get_diagnostics_str()` (*SA method*), 96
- `get_extra_fields()` (*MCMC method*), 91
- `get_params()` (*Adagrad method*), 128
- `get_params()` (*Adam method*), 127
- `get_params()` (*ClippedAdam method*), 129
- `get_params()` (*Minimize method*), 130
- `get_params()` (*Momentum method*), 131
- `get_params()` (*RMSProp method*), 131
- `get_params()` (*RMSPropMomentum method*), 132
- `get_params()` (*SGD method*), 133
- `get_params()` (*SM3 method*), 133
- `get_params()` (*SVI method*), 106
- `get_posterior()` (*AutoContinuous method*), 110
- `get_posterior()` (*AutoDiagonalNormal method*), 111
- `get_posterior()` (*AutoLaplaceApproximation method*), 114
- `get_posterior()` (*AutoLowRankMultivariateNormal method*), 115
- `get_posterior()` (*AutoMultivariateNormal method*), 112
- `get_samples()` (*MCMC method*), 90
- `get_trace()` (*trace method*), 19
- `get_transform()` (*AutoContinuous method*), 109
- `get_transform()` (*AutoDiagonalNormal method*), 111
- `get_transform()` (*AutoLaplaceApproximation method*), 114
- `get_transform()` (*AutoLowRankMultivariateNormal method*), 115
- `get_transform()` (*AutoMultivariateNormal method*), 112
- `greater_than()` (in module `numpyro.distributions.constraints`), 78
- Gumbel (class in `numpyro.contrib.tfp.distributions`), 64
- Gumbel (class in `numpyro.distributions.continuous`), 34
- ## H
- `haiku_module()` (in module `numpyro.contrib.module`), 4
- HalfCauchy (class in `numpyro.contrib.tfp.distributions`), 65
- HalfCauchy (class in `numpyro.distributions.continuous`), 35

HalfNormal (class *numpyro.contrib.tfp.distributions*), 65
 HalfNormal (class *numpyro.distributions.continuous*), 36
 HalfStudentT (class *numpyro.contrib.tfp.distributions*), 65
 HamiltonianMonteCarlo (class *numpyro.contrib.tfp.mcmc*), 101
 has_enumerate_support (BernoulliLogits attribute), 45
 has_enumerate_support (BernoulliProbs attribute), 46
 has_enumerate_support (BetaBinomial attribute), 47
 has_enumerate_support (BinomialLogits attribute), 47
 has_enumerate_support (BinomialProbs attribute), 48
 has_enumerate_support (CategoricalLogits attribute), 49
 has_enumerate_support (CategoricalProbs attribute), 49
 has_enumerate_support (Distribution attribute), 21
 has_enumerate_support (ExpandedDistribution attribute), 24
 has_enumerate_support (Independent attribute), 26
 has_enumerate_support (MaskedDistribution attribute), 27
 HiddenMarkovModel (class *numpyro.contrib.tfp.distributions*), 65
 HMC (class in *numpyro.infer.hmc*), 92
 hmc() (in module *numpyro.infer.hmc*), 97
 HMCState (in module *numpyro.infer.hmc*), 99
 Horseshoe (class in *numpyro.contrib.tfp.distributions*), 65
 hpdi() (in module *numpyro.diagnostics*), 136
 |
 icdf() (Normal method), 41
 IdentityTransform (class *numpyro.distributions.transforms*), 83
 ImproperUniform (class *numpyro.distributions.distribution*), 24
 Independent (class *numpyro.contrib.tfp.distributions*), 65
 Independent (class *numpyro.distributions.distribution*), 26
 infer_config (class *numpyro.contrib.functor.enum_messenger*), 123
 init() (Adagrad method), 128
 init() (Adam method), 127
 in init() (ClippedAdam method), 129
 init() (HMC method), 94
 in init() (MCMCKernel method), 92
 init() (Minimize method), 130
 in init() (Momentum method), 131
 init() (RMSPProp method), 131
 in init() (RMSPPropMomentum method), 132
 init() (SA method), 96
 init() (SGD method), 133
 init() (SM3 method), 134
 init() (SVI method), 106
 init_kernel() (in module *numpyro.infer.hmc.hmc*), 98
 init_to_feasible() (in module *numpyro.infer.initialization*), 144
 init_to_median() (in module *numpyro.infer.initialization*), 144
 init_to_sample() (in module *numpyro.infer.initialization*), 144
 init_to_uniform() (in module *numpyro.infer.initialization*), 145
 init_to_value() (in module *numpyro.infer.initialization*), 145
 initialize_model() (in module *numpyro.infer.util*), 102
 integer_greater_than() (in module *numpyro.distributions.constraints*), 78
 integer_interval() (in module *numpyro.distributions.constraints*), 78
 interval() (in module *numpyro.distributions.constraints*), 78
 inv() (AbsTransform method), 81
 inv() (AffineTransform method), 82
 inv() (BlockNeuralAutoregressiveTransform method), 88
 inv() (ComposeTransform method), 82
 inv() (CorrCholeskyTransform method), 83
 inv() (ExpTransform method), 83
 inv() (IdentityTransform method), 83
 inv() (InvCholeskyTransform method), 84
 inv() (InverseAutoregressiveTransform method), 87
 in inv() (LowerCholeskyAffine method), 84
 inv() (LowerCholeskyTransform method), 84
 in inv() (OrderedTransform method), 85
 inv() (PermuteTransform method), 85
 in inv() (PowerTransform method), 85
 inv() (SigmoidTransform method), 85
 in inv() (StickBreakingTransform method), 86
 inv() (Transform method), 81
 in InvCholeskyTransform (class *numpyro.distributions.transforms*), 84
 InverseAutoregressiveTransform (class in *numpyro.distributions.flows*), 87

- InverseGamma (class *numpyro.contrib.tfp.distributions*), 66
- InverseGamma (class *numpyro.distributions.continuous*), 36
- InverseGaussian (class *numpyro.contrib.tfp.distributions*), 66
- is_discrete (*BernoulliLogits* attribute), 45
- is_discrete (*BernoulliProbs* attribute), 46
- is_discrete (*BetaBinomial* attribute), 47
- is_discrete (*BinomialLogits* attribute), 47
- is_discrete (*BinomialProbs* attribute), 48
- is_discrete (*CategoricalLogits* attribute), 49
- is_discrete (*CategoricalProbs* attribute), 49
- is_discrete (*Delta* attribute), 50
- is_discrete (*DirichletMultinomial* attribute), 51
- is_discrete (*Distribution* attribute), 21
- is_discrete (*ExpandedDistribution* attribute), 24
- is_discrete (*GammaPoisson* attribute), 52
- is_discrete (*GeometricLogits* attribute), 52
- is_discrete (*GeometricProbs* attribute), 53
- is_discrete (*Independent* attribute), 26
- is_discrete (*MaskedDistribution* attribute), 27
- is_discrete (*MultinomialLogits* attribute), 53
- is_discrete (*MultinomialProbs* attribute), 54
- is_discrete (*Poisson* attribute), 55
- is_discrete (*PRNGIdentity* attribute), 55
- is_discrete (*ZeroInflatedPoisson* attribute), 56
- ## J
- JohnsonSU (class in *numpyro.contrib.tfp.distributions*), 66
- JointDistribution (class *numpyro.contrib.tfp.distributions*), 66
- JointDistributionCoroutine (class in *numpyro.contrib.tfp.distributions*), 66
- JointDistributionCoroutineAutoBatched (class in *numpyro.contrib.tfp.distributions*), 66
- JointDistributionNamed (class in *numpyro.contrib.tfp.distributions*), 67
- JointDistributionNamedAutoBatched (class in *numpyro.contrib.tfp.distributions*), 67
- JointDistributionSequential (class in *numpyro.contrib.tfp.distributions*), 67
- JointDistributionSequentialAutoBatched (class in *numpyro.contrib.tfp.distributions*), 67
- ## K
- Kumaraswamy (class *numpyro.contrib.tfp.distributions*), 67
- ## L
- Laplace (class in *numpyro.contrib.tfp.distributions*), 68
- Laplace (class in *numpyro.distributions.continuous*), 37
- in less_than() (in module *numpyro.distributions.constraints*), 78
- in lift (class in *numpyro.handlers*), 14
- LinearGaussianStateSpaceModel (class in *numpyro.contrib.tfp.distributions*), 68
- LKJ (class in *numpyro.contrib.tfp.distributions*), 67
- LKJ (class in *numpyro.distributions.continuous*), 37
- LKJCholesky (class in *numpyro.distributions.continuous*), 38
- LocScaleReparam (class in *numpyro.infer.reparam*), 119
- log_abs_det_jacobian() (*AffineTransform* method), 82
- log_abs_det_jacobian() (*BlockNeuralAutoregressiveTransform* method), 88
- log_abs_det_jacobian() (*ComposeTransform* method), 82
- log_abs_det_jacobian() (*CorrCholeskyTransform* method), 83
- log_abs_det_jacobian() (*ExpTransform* method), 83
- log_abs_det_jacobian() (*IdentityTransform* method), 83
- log_abs_det_jacobian() (*InvCholeskyTransform* method), 84
- log_abs_det_jacobian() (*InverseAutoregressiveTransform* method), 87
- log_abs_det_jacobian() (*LowerCholeskyAffine* method), 84
- log_abs_det_jacobian() (*LowerCholeskyTransform* method), 84
- log_abs_det_jacobian() (*OrderedTransform* method), 85
- log_abs_det_jacobian() (*PermuteTransform* method), 85
- log_abs_det_jacobian() (*PowerTransform* method), 85
- log_abs_det_jacobian() (*SigmoidTransform* method), 85
- log_abs_det_jacobian() (*StickBreakingTransform* method), 86
- log_abs_det_jacobian() (*Transform* method), 81
- log_density() (in module *numpyro.contrib.functor.infer_util*), 125
- log_density() (in module *numpyro.infer.util*), 142
- log_likelihood() (in module *numpyro.infer.util*), 143
- log_prob() (*BernoulliLogits* method), 45
- log_prob() (*BernoulliProbs* method), 46
- log_prob() (*Beta* method), 31
- log_prob() (*BetaBinomial* method), 47
- log_prob() (*BinomialLogits* method), 48
- log_prob() (*BinomialProbs* method), 48
- log_prob() (*CategoricalLogits* method), 49

- `log_prob()` (*CategoricalProbs method*), 50
 - `log_prob()` (*Cauchy method*), 32
 - `log_prob()` (*Delta method*), 50
 - `log_prob()` (*Dirichlet method*), 33
 - `log_prob()` (*DirichletMultinomial method*), 51
 - `log_prob()` (*Distribution method*), 23
 - `log_prob()` (*ExpandedDistribution method*), 24
 - `log_prob()` (*Exponential method*), 33
 - `log_prob()` (*Gamma method*), 34
 - `log_prob()` (*GammaPoisson method*), 52
 - `log_prob()` (*GaussianRandomWalk method*), 35
 - `log_prob()` (*GeometricLogits method*), 52
 - `log_prob()` (*GeometricProbs method*), 53
 - `log_prob()` (*Gumbel method*), 34
 - `log_prob()` (*HalfCauchy method*), 35
 - `log_prob()` (*HalfNormal method*), 36
 - `log_prob()` (*ImproperUniform method*), 25
 - `log_prob()` (*Independent method*), 27
 - `log_prob()` (*Laplace method*), 37
 - `log_prob()` (*LKJCholesky method*), 39
 - `log_prob()` (*Logistic method*), 39
 - `log_prob()` (*LowRankMultivariateNormal method*), 41
 - `log_prob()` (*MaskedDistribution method*), 28
 - `log_prob()` (*MultinomialLogits method*), 54
 - `log_prob()` (*MultinomialProbs method*), 54
 - `log_prob()` (*MultivariateNormal method*), 40
 - `log_prob()` (*Normal method*), 41
 - `log_prob()` (*Poisson method*), 55
 - `log_prob()` (*StudentT method*), 42
 - `log_prob()` (*TransformedDistribution method*), 29
 - `log_prob()` (*TruncatedPolyaGamma method*), 43
 - `log_prob()` (*Unit method*), 29
 - `log_prob()` (*VonMises method*), 57
 - `log_prob()` (*ZeroInflatedPoisson method*), 56
 - `Logistic` (class in `numpyro.contrib.tfp.distributions`), 68
 - `Logistic` (class in `numpyro.distributions.continuous`), 39
 - `LogitNormal` (class in `numpyro.contrib.tfp.distributions`), 68
 - `LogLogistic` (class in `numpyro.contrib.tfp.distributions`), 68
 - `LogNormal` (class in `numpyro.contrib.tfp.distributions`), 68
 - `LogNormal` (class in `numpyro.distributions.continuous`), 39
 - `loss()` (*RenyiELBO method*), 107
 - `loss()` (*Trace_ELBO method*), 108
 - `lower_cholesky` (in `numpyro.distributions.constraints`), 79
 - `LowerCholeskyAffine` (class in `numpyro.distributions.transforms`), 84
 - `LowerCholeskyTransform` (class in `numpyro.distributions.transforms`), 84
 - `LowRankMultivariateNormal` (class in `numpyro.distributions.continuous`), 40
- ## M
- `markov` (class in `numpyro.contrib.functor.enum_messenger`), 123
 - `mask` (class in `numpyro.handlers`), 15
 - `mask()` (*Distribution method*), 23
 - `MaskedDistribution` (class in `numpyro.distributions.distribution`), 27
 - `MCMC` (class in `numpyro.infer.mcmc`), 89
 - `MCMCKernel` (class in `numpyro.infer.mcmc`), 91
 - `mean` (*BernoulliLogits attribute*), 45
 - `mean` (*BernoulliProbs attribute*), 46
 - `mean` (*Beta attribute*), 31
 - `mean` (*BetaBinomial attribute*), 47
 - `mean` (*BinomialLogits attribute*), 48
 - `mean` (*BinomialProbs attribute*), 48
 - `mean` (*CategoricalLogits attribute*), 49
 - `mean` (*CategoricalProbs attribute*), 50
 - `mean` (*Cauchy attribute*), 32
 - `mean` (*Delta attribute*), 50
 - `mean` (*Dirichlet attribute*), 33
 - `mean` (*DirichletMultinomial attribute*), 51
 - `mean` (*Distribution attribute*), 23
 - `mean` (*ExpandedDistribution attribute*), 24
 - `mean` (*Exponential attribute*), 33
 - `mean` (*Gamma attribute*), 34
 - `mean` (*GammaPoisson attribute*), 52
 - `mean` (*GaussianRandomWalk attribute*), 35
 - `mean` (*GeometricLogits attribute*), 53
 - `mean` (*GeometricProbs attribute*), 53
 - `mean` (*Gumbel attribute*), 34
 - `mean` (*HalfCauchy attribute*), 35
 - `mean` (*HalfNormal attribute*), 36
 - `mean` (*Independent attribute*), 26
 - `mean` (*InverseGamma attribute*), 36
 - `mean` (*Laplace attribute*), 37
 - `mean` (*LKJ attribute*), 38
 - `mean` (*Logistic attribute*), 39
 - `mean` (*LogNormal attribute*), 39
 - `mean` (*LowRankMultivariateNormal attribute*), 40
 - `mean` (*MaskedDistribution attribute*), 28
 - `mean` (*MultinomialLogits attribute*), 54
 - `mean` (*MultinomialProbs attribute*), 54
 - `mean` (*MultivariateNormal attribute*), 40
 - `mean` (*Normal attribute*), 41
 - `mean` (*Pareto attribute*), 41
 - `mean` (*Poisson attribute*), 55
 - `mean` (*StudentT attribute*), 42
 - `mean` (*TransformedDistribution attribute*), 29
 - `mean` (*TruncatedCauchy attribute*), 42

- mean (*TruncatedNormal* attribute), 43
mean (*Uniform* attribute), 44
mean (*VonMises* attribute), 57
mean (*ZeroInflatedPoisson* attribute), 56
median() (*AutoContinuous* method), 110
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*), 129
MixtureSameFamily (class in *numpyro.contrib.tfp.distributions*), 68
model (*HMC* attribute), 93
module() (in module *numpyro.primitives*), 4
Momentum (class in *numpyro.optim*), 130
Moyal (class in *numpyro.contrib.tfp.distributions*), 69
Multinomial (class in *numpyro.contrib.tfp.distributions*), 69
multinomial() (in module *numpyro.distributions.constraints*), 79
Multinomial() (in module *numpyro.distributions.discrete*), 53
MultinomialLogits (class in *numpyro.distributions.discrete*), 53
MultinomialProbs (class in *numpyro.distributions.discrete*), 54
MultivariateNormal (class in *numpyro.distributions.continuous*), 40
MultivariateNormalDiag (class in *numpyro.contrib.tfp.distributions*), 69
MultivariateNormalDiagPlusLowRank (class in *numpyro.contrib.tfp.distributions*), 69
MultivariateNormalFullCovariance (class in *numpyro.contrib.tfp.distributions*), 69
MultivariateNormalLinearOperator (class in *numpyro.contrib.tfp.distributions*), 69
MultivariateNormalTriL (class in *numpyro.contrib.tfp.distributions*), 70
MultivariateStudentTLinearOperator (class in *numpyro.contrib.tfp.distributions*), 70
- ## N
- NegativeBinomial (class in *numpyro.contrib.tfp.distributions*), 70
NeuTraReparam (class in *numpyro.infer.reparam*), 120
nonnegative_integer (in module *numpyro.distributions.constraints*), 79
Normal (class in *numpyro.contrib.tfp.distributions*), 70
Normal (class in *numpyro.distributions.continuous*), 41
NoUTurnSampler (class in *numpyro.contrib.tfp.mcmc*), 101
num_gamma_variates (*TruncatedPolyaGamma* attribute), 43
num_log_prob_terms (*TruncatedPolyaGamma* attribute), 43
numpyro.contrib.functor (module), 123
numpyro.contrib.indexing (module), 145
numpyro.contrib.tfp.distributions (module), 59
numpyro.contrib.tfp.mcmc (module), 100
numpyro.diagnostics (module), 135
numpyro.handlers (module), 11
numpyro.infer.autoguide (module), 109
numpyro.infer.reparam (module), 119
numpyro.infer.util (module), 141
numpyro.optim (module), 127
numpyro.primitives (module), 1
numpyro.util (module), 139
NUTS (class in *numpyro.infer.hmc*), 94
- ## O
- OneHotCategorical (class in *numpyro.contrib.tfp.distributions*), 70
ordered_vector (in module *numpyro.distributions.constraints*), 79
OrderedLogistic (class in *numpyro.contrib.tfp.distributions*), 70
OrderedLogistic (class in *numpyro.distributions.discrete*), 54
OrderedTransform (class in *numpyro.distributions.transforms*), 85
- ## P
- param() (in module *numpyro.primitives*), 1
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*), 41
PermuteTransform (class in *numpyro.distributions.transforms*), 85
PERT (class in *numpyro.contrib.tfp.distributions*), 71
PlackettLuce (class in *numpyro.contrib.tfp.distributions*), 71
plate (class in *numpyro.contrib.functor.enum_messenger*), 124
plate (class in *numpyro.primitives*), 2
plate_stack() (in module *numpyro.primitives*), 2
plate_to_enum_plate() (in module *numpyro.contrib.functor.infer_util*), 126
Poisson (class in *numpyro.contrib.tfp.distributions*), 71
Poisson (class in *numpyro.distributions.discrete*), 55

PoissonLogNormalQuadratureCompound (class in `numpyro.contrib.tfp.distributions`), 71

positive (in module `numpyro.distributions.constraints`), 79

positive_definite (in module `numpyro.distributions.constraints`), 79

positive_integer (in module `numpyro.distributions.constraints`), 79

postprocess_fn() (HMC method), 94

postprocess_fn() (MCMCKernel method), 92

postprocess_fn() (SA method), 96

postprocess_message() (plate method), 124

postprocess_message() (trace method), 19, 125

potential_energy() (in module `numpyro.infer.util`), 143

PowerSpherical (class in `numpyro.contrib.tfp.distributions`), 71

PowerTransform (class in `numpyro.distributions.transforms`), 85

precision_matrix (LowRankMultivariateNormal attribute), 40

precision_matrix (MultivariateNormal attribute), 40

Predictive (class in `numpyro.infer.util`), 141

print_summary() (in module `numpyro.diagnostics`), 137

print_summary() (MCMC method), 91

PRNGIdentity (class in `numpyro.distributions.discrete`), 55

ProbitBernoulli (class in `numpyro.contrib.tfp.distributions`), 72

probs (BernoulliLogits attribute), 45

probs (BinomialLogits attribute), 48

probs (CategoricalLogits attribute), 49

probs (GeometricLogits attribute), 52

probs (MultinomialLogits attribute), 54

process_message() (block method), 13

process_message() (collapse method), 13

process_message() (condition method), 13

process_message() (do method), 14

process_message() (enum method), 123

process_message() (infer_config method), 123

process_message() (lift method), 15

process_message() (mask method), 15

process_message() (plate method), 124

process_message() (reparam method), 15

process_message() (replay method), 16

process_message() (scale method), 16

process_message() (scope method), 17

process_message() (seed method), 18

process_message() (substitute method), 18

Q

quantiles() (AutoContinuous method), 110

quantiles() (AutoDiagonalNormal method), 112

quantiles() (AutoLaplaceApproximation method), 114

quantiles() (AutoLowRankMultivariateNormal method), 115

quantiles() (AutoMultivariateNormal method), 113

quantiles() (AutoNormal method), 116

QuantizedDistribution (class in `numpyro.contrib.tfp.distributions`), 72

R

random_flax_module() (in module `numpyro.contrib.module`), 5

random_haiku_module() (in module `numpyro.contrib.module`), 6

RandomWalkMetropolis (class in `numpyro.contrib.tfp.mcmc`), 101

real (in module `numpyro.distributions.constraints`), 79

real_vector (in module `numpyro.distributions.constraints`), 79

RelaxedBernoulli (class in `numpyro.contrib.tfp.distributions`), 72

RelaxedOneHotCategorical (class in `numpyro.contrib.tfp.distributions`), 72

RenyiELBO (class in `numpyro.infer.elbo`), 107

reparam (class in `numpyro.handlers`), 15

Reparam (class in `numpyro.infer.reparam`), 119

reparam() (NeuTraReparam method), 120

reparameterized_params (Independent attribute), 26

reparametrized_params (Cauchy attribute), 32

reparametrized_params (Distribution attribute), 22

reparametrized_params (Exponential attribute), 33

reparametrized_params (Gamma attribute), 33

reparametrized_params (GaussianRandomWalk attribute), 35

reparametrized_params (Gumbel attribute), 34

reparametrized_params (HalfCauchy attribute), 35

reparametrized_params (HalfNormal attribute), 36

reparametrized_params (InverseGamma attribute), 36

reparametrized_params (Laplace attribute), 37

reparametrized_params (Logistic attribute), 39

reparametrized_params (LogNormal attribute), 39

reparametrized_params (MultivariateNormal attribute), 40

reparametrized_params (Normal attribute), 41

reparametrized_params (StudentT attribute), 42

- reparametrized_params (*TruncatedCauchy attribute*), 42
 reparametrized_params (*TruncatedNormal attribute*), 43
 reparametrized_params (*Uniform attribute*), 44
 replay (*class in numpyro.handlers*), 16
 ReplicaExchangeMC (*class in numpyro.contrib.tfp.mcmc*), 101
 RMSProp (*class in numpyro.optim*), 131
 RMSPropMomentum (*class in numpyro.optim*), 132
 run () (*MCMC method*), 90
- ## S
- SA (*class in numpyro.infer.sa*), 95
 Sample (*class in numpyro.contrib.tfp.distributions*), 72
 sample () (*BernoulliLogits method*), 45
 sample () (*BernoulliProbs method*), 46
 sample () (*Beta method*), 31
 sample () (*BetaBinomial method*), 47
 sample () (*BinomialLogits method*), 47
 sample () (*BinomialProbs method*), 48
 sample () (*CategoricalLogits method*), 49
 sample () (*CategoricalProbs method*), 49
 sample () (*Cauchy method*), 32
 sample () (*Delta method*), 50
 sample () (*Dirichlet method*), 32
 sample () (*DirichletMultinomial method*), 51
 sample () (*Distribution method*), 22
 sample () (*ExpandedDistribution method*), 24
 sample () (*Exponential method*), 33
 sample () (*Gamma method*), 33
 sample () (*GammaPoisson method*), 52
 sample () (*GaussianRandomWalk method*), 35
 sample () (*GeometricLogits method*), 52
 sample () (*GeometricProbs method*), 53
 sample () (*Gumbel method*), 34
 sample () (*HalfCauchy method*), 35
 sample () (*HalfNormal method*), 36
 sample () (*HMC method*), 94
 sample () (*in module numpyro.primitives*), 1
 sample () (*Independent method*), 26
 sample () (*Laplace method*), 37
 sample () (*LKJCholesky method*), 38
 sample () (*Logistic method*), 39
 sample () (*LowRankMultivariateNormal method*), 40
 sample () (*MaskedDistribution method*), 27
 sample () (*MCMCKernel method*), 92
 sample () (*MultinomialLogits method*), 53
 sample () (*MultinomialProbs method*), 54
 sample () (*MultivariateNormal method*), 40
 sample () (*Normal method*), 41
 sample () (*Poisson method*), 55
 sample () (*PRNGIdentity method*), 55
 sample () (*SA method*), 96
 sample () (*StudentT method*), 42
 sample () (*TransformedDistribution method*), 28
 sample () (*TruncatedPolyaGamma method*), 43
 sample () (*Unit method*), 29
 sample () (*VonMises method*), 57
 sample () (*ZeroInflatedPoisson method*), 56
 sample_field (*HMC attribute*), 93
 sample_field (*MCMCKernel attribute*), 92
 sample_field (*SA attribute*), 96
 sample_kernel () (*in module numpyro.infer.hmc.hmc*), 99
 sample_posterior () (*AutoContinuous method*), 110
 sample_posterior () (*AutoLaplaceApproximation method*), 114
 sample_posterior () (*AutoNormal method*), 116
 sample_with_intermediates () (*Distribution method*), 22
 sample_with_intermediates () (*Transformed-Distribution method*), 28
 SASTate (*in module numpyro.infer.sa*), 99
 scale (*class in numpyro.handlers*), 16
 scale_tril (*LowRankMultivariateNormal attribute*), 40
 scan () (*in module numpyro.contrib.control_flow*), 7
 scope (*class in numpyro.handlers*), 16
 seed (*class in numpyro.handlers*), 17
 set_default_validate_args () (*Distribution static method*), 22
 set_host_device_count () (*in module numpyro.util*), 140
 set_platform () (*in module numpyro.util*), 140
 SGD (*class in numpyro.optim*), 133
 shape () (*Distribution method*), 22
 SigmoidTransform (*class in numpyro.distributions.transforms*), 85
 simplex (*in module numpyro.distributions.constraints*), 80
 SinhArcsinh (*class in numpyro.contrib.tfp.distributions*), 72
 SliceSampler (*class in numpyro.contrib.tfp.mcmc*), 101
 SM3 (*class in numpyro.optim*), 133
 SphericalUniform (*class in numpyro.contrib.tfp.distributions*), 72
 split_gelman_rubin () (*in module numpyro.diagnostics*), 136
 StickBreakingTransform (*class in numpyro.distributions.transforms*), 86
 StudentT (*class in numpyro.contrib.tfp.distributions*), 73
 StudentT (*class in numpyro.distributions.continuous*), 42
 StudentTProcess (*class in*

[numpyro.contrib.tfp.distributions](#)), 73
[subsample\(\)](#) (in module [numpyro.primitives](#)), 3
[substitute](#) (class in [numpyro.handlers](#)), 18
[summary\(\)](#) (in module [numpyro.diagnostics](#)), 137
[support](#) ([BernoulliLogits](#) attribute), 45
[support](#) ([BernoulliProbs](#) attribute), 46
[support](#) ([Beta](#) attribute), 31
[support](#) ([BetaBinomial](#) attribute), 47
[support](#) ([BinomialLogits](#) attribute), 48
[support](#) ([BinomialProbs](#) attribute), 48
[support](#) ([CategoricalLogits](#) attribute), 49
[support](#) ([CategoricalProbs](#) attribute), 50
[support](#) ([Cauchy](#) attribute), 32
[support](#) ([Delta](#) attribute), 50
[support](#) ([Dirichlet](#) attribute), 32
[support](#) ([DirichletMultinomial](#) attribute), 51
[support](#) ([Distribution](#) attribute), 21
[support](#) ([ExpandedDistribution](#) attribute), 24
[support](#) ([Exponential](#) attribute), 33
[support](#) ([Gamma](#) attribute), 33
[support](#) ([GammaPoisson](#) attribute), 52
[support](#) ([GaussianRandomWalk](#) attribute), 35
[support](#) ([GeometricLogits](#) attribute), 52
[support](#) ([GeometricProbs](#) attribute), 53
[support](#) ([Gumbel](#) attribute), 34
[support](#) ([HalfCauchy](#) attribute), 35
[support](#) ([HalfNormal](#) attribute), 36
[support](#) ([Independent](#) attribute), 26
[support](#) ([InverseGamma](#) attribute), 36
[support](#) ([Laplace](#) attribute), 37
[support](#) ([LKJ](#) attribute), 38
[support](#) ([LKJCholesky](#) attribute), 38
[support](#) ([Logistic](#) attribute), 39
[support](#) ([LowRankMultivariateNormal](#) attribute), 40
[support](#) ([MaskedDistribution](#) attribute), 27
[support](#) ([MultinomialLogits](#) attribute), 54
[support](#) ([MultinomialProbs](#) attribute), 54
[support](#) ([MultivariateNormal](#) attribute), 40
[support](#) ([Normal](#) attribute), 41
[support](#) ([Pareto](#) attribute), 42
[support](#) ([Poisson](#) attribute), 55
[support](#) ([StudentT](#) attribute), 42
[support](#) ([TransformedDistribution](#) attribute), 28
[support](#) ([TruncatedCauchy](#) attribute), 42
[support](#) ([TruncatedNormal](#) attribute), 43
[support](#) ([TruncatedPolyaGamma](#) attribute), 43
[support](#) ([Uniform](#) attribute), 44
[support](#) ([Unit](#) attribute), 29
[support](#) ([VonMises](#) attribute), 57
[support](#) ([ZeroInflatedPoisson](#) attribute), 56
[SVI](#) (class in [numpyro.infer.svi](#)), 105

T

TFPDistributionMixin (class

[numpyro.contrib.tfp.distributions](#)), 59
[TFPKernel](#) (class in [numpyro.contrib.tfp.mcmc](#)), 100
[to_data](#) (class in [numpyro.contrib.functor.enum_messenger](#)), 124
[to_event\(\)](#) (*Distribution method*), 23
[to_functor](#) (class in [numpyro.contrib.functor.enum_messenger](#)), 124
[trace](#) (class in [numpyro.contrib.functor.enum_messenger](#)), 125
[trace](#) (class in [numpyro.handlers](#)), 18
[Trace_ELBO](#) (class in [numpyro.infer.elbo](#)), 108
[Transform](#) (class in [numpyro.distributions.transforms](#)), 81
[transform_fn\(\)](#) (in module [numpyro.infer.util](#)), 142
[transform_sample\(\)](#) (*NeuTraReparam method*), 120
[TransformedDistribution](#) (class in [numpyro.contrib.tfp.distributions](#)), 73
[TransformedDistribution](#) (class in [numpyro.distributions.distribution](#)), 28
[TransformReparam](#) (class in [numpyro.infer.reparam](#)), 121
[tree_flatten\(\)](#) (*Delta method*), 50
[tree_flatten\(\)](#) (*Distribution method*), 22
[tree_flatten\(\)](#) (*ExpandedDistribution method*), 24
[tree_flatten\(\)](#) (*GaussianRandomWalk method*), 35
[tree_flatten\(\)](#) (*ImproperUniform method*), 25
[tree_flatten\(\)](#) (*Independent method*), 27
[tree_flatten\(\)](#) (*InverseGamma method*), 36
[tree_flatten\(\)](#) (*LKJ method*), 38
[tree_flatten\(\)](#) (*LKJCholesky method*), 39
[tree_flatten\(\)](#) (*LogNormal method*), 39
[tree_flatten\(\)](#) (*MaskedDistribution method*), 28
[tree_flatten\(\)](#) (*MultivariateNormal method*), 40
[tree_flatten\(\)](#) (*Pareto method*), 42
[tree_flatten\(\)](#) (*TransformedDistribution method*), 29
[tree_flatten\(\)](#) (*TruncatedCauchy method*), 42
[tree_flatten\(\)](#) (*TruncatedNormal method*), 43
[tree_flatten\(\)](#) (*TruncatedPolyaGamma method*), 43
[tree_flatten\(\)](#) (*Uniform method*), 44
[tree_unflatten\(\)](#) ([numpyro.distributions.continuous.GaussianRandomWalk](#) class method), 35
[tree_unflatten\(\)](#) ([numpyro.distributions.continuous.LKJ](#) class method), 38
[tree_unflatten\(\)](#) ([numpyro.distributions.continuous.LKJCholesky](#) class method), 39
[tree_unflatten\(\)](#) ([numpyro.distributions.continuous.MultivariateNormal](#) class method), 40
[tree_unflatten\(\)](#) ([numpyro.distributions.continuous.TruncatedCauchy](#) class method), 43
[tree_unflatten\(\)](#) ([numpyro.distributions.continuous.TruncatedNormal](#) class method), 43

- class method), 43
 tree_unflatten() (numpyro.distributions.continuous.TruncatedPolyaGamma
 class method), 43
 tree_unflatten() (numpyro.distributions.continuous.Uniform
 class method), 44
 tree_unflatten() (numpyro.distributions.discrete.Delta
 class method), 50
 tree_unflatten() (numpyro.distributions.distribution.Distribution
 class method), 22
 tree_unflatten() (numpyro.distributions.distribution.ExpandedDistribution
 class method), 24
 tree_unflatten() (numpyro.distributions.distribution.Independent
 class method), 27
 tree_unflatten() (numpyro.distributions.distribution.MaskedDistribution
 class method), 28
 Triangular (class in numpyro.contrib.tfp.distributions), 73
 TruncatedCauchy (class in numpyro.contrib.tfp.distributions), 73
 TruncatedCauchy (class in numpyro.distributions.continuous), 42
 TruncatedNormal (class in numpyro.contrib.tfp.distributions), 73
 TruncatedNormal (class in numpyro.distributions.continuous), 43
 TruncatedPolyaGamma (class in numpyro.distributions.continuous), 43
 truncation_point (TruncatedPolyaGamma attribute), 43
- ## 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), 44
 Unit (class in numpyro.distributions.distribution), 29
 unit_interval (in module numpyro.distributions.constraints), 80
 update() (Adagrad method), 128
 update() (Adam method), 127
 update() (ClippedAdam method), 129
 update() (Minimize method), 130
 update() (Momentum method), 131
 update() (RMSPProp method), 132
 update() (RMSPPropMomentum method), 132
 update() (SGD method), 133
 update() (SM3 method), 134
 update() (SVI method), 106
- ## V
- validation_enabled() (in module numpyro.distributions.distribution), 139
 variance (BernoulliLogits attribute), 46
 variance (BernoulliProbs attribute), 46
 variance (Beta attribute), 31
 variance (BetaBinomial attribute), 47
 variance (BinomialLogits attribute), 48
 variance (BinomialProbs attribute), 48
 variance (CategoricalLogits attribute), 49
 variance (CategoricalProbs attribute), 50
 variance (Cauchy attribute), 32
 variance (Delta attribute), 50
 variance (Dirichlet attribute), 33
 variance (DirichletMultinomial attribute), 51
 variance (Distribution attribute), 23
 variance (ExpandedDistribution attribute), 24
 variance (Exponential attribute), 33
 variance (Gamma attribute), 34
 variance (GammaPoisson attribute), 52
 variance (GaussianRandomWalk attribute), 35
 variance (GeometricLogits attribute), 53
 variance (GeometricProbs attribute), 53
 variance (Gumbel attribute), 34
 variance (HalfCauchy attribute), 35
 variance (HalfNormal attribute), 36
 variance (Independent attribute), 26
 variance (InverseGamma attribute), 36
 variance (Laplace attribute), 37
 variance (Logistic attribute), 39
 variance (LogNormal attribute), 39
 variance (LowRankMultivariateNormal attribute), 40
 variance (MaskedDistribution attribute), 28
 variance (MultinomialLogits attribute), 54
 variance (MultinomialProbs attribute), 54
 variance (MultivariateNormal attribute), 40
 variance (Normal attribute), 41
 variance (Pareto attribute), 42
 variance (Poisson attribute), 55
 variance (StudentT attribute), 42
 variance (TransformedDistribution attribute), 29
 variance (TruncatedCauchy attribute), 42
 variance (TruncatedNormal attribute), 43
 variance (Uniform attribute), 44
 variance (VonMises attribute), 57
 variance (ZeroInflatedPoisson attribute), 56
 VariationalGaussianProcess (class in numpyro.contrib.tfp.distributions), 74
 VectorDeterministic (class in numpyro.contrib.tfp.distributions), 74
 VectorExponentialDiag (class in numpyro.contrib.tfp.distributions), 74
 Vindex (class in numpyro.contrib.indexing), 146
 vindex() (in module numpyro.contrib.indexing), 145

VonMises (*class in `numpyro.contrib.tfp.distributions`*),
[74](#)

VonMises (*class in `numpyro.distributions.directional`*),
[57](#)

VonMisesFisher (*class in `numpyro.contrib.tfp.distributions`*), [74](#)

W

warmup() (*MCMC method*), [90](#)

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`*), [56](#)