NumPyro Documentation

Release 0.0

Uber AI Labs

Sep 08, 2019

Inference:

CHAPTER 1

Markov Chain Monte Carlo (MCMC)

1.1 Hamiltonian Monte Carlo

class MCMC(*sampler*, *num_warmup*, *num_samples*, *num_chains=1*, *constrain_fn=None*, *chain_method='parallel'*, *progress_bar=True*)

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.

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.
- **constrain_fn** – Callable that converts a collection of unconstrained sample values returned from the sampler to constrained values that lie within the support of the sample sites.
- **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`.

run(*rng*, **args*, *collect_fields=('z',)*, *collect_warmup=False*, *init_params=None*, ***kwargs*)
Run the MCMC samplers and collect samples.

Parameters

- **rng** (*random.PRNGKey*) – Random number generator key to be used for the sampling.
- **args** – Arguments to be provided to the `numpyro.mcmc.MCMCKernel.init()` method. These are typically the arguments needed by the *model*.
- **collect_fields** (*tuple or list*) – Fields from *numpyro.mcmc.HMCState* to collect during the MCMC run. By default, only the latent sample sites *z* is collected.
- **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.mcmc.MCMCKernel.init()` method. These are typically the keyword arguments needed by the *model*.

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*. If multiple fields are collected via the *collect_fields* arg to *run()*, then a tuple with the same data type is returned, one for each of the fields. The data type for a particular field 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).

print_summary()

class HMC(*model=None*, *potential_fn=None*, *kinetic_fn=None*, *step_size=1.0*, *adapt_step_size=True*, *adapt_mass_matrix=True*, *dense_mass=False*, *target_accept_prob=0.8*, *trajectory_length=6.283185307179586*)
Bases: `numpyro.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_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. Default value is 2π.

init (*rng*, *num_warmup*, *init_params=None*, *model_args=()*, *model_kwargs={}*)

sample (*state*)

Run HMC from the given *HMCState* and return the resulting *HMCState*.

Parameters **state** (*HMCState*) – Represents the current state.

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=6.283185307179586*, *max_tree_depth=10*)

Bases: *numpyro.mcmc.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. 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.

hmc (*potential_fn*, *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.
- **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.mcmc.MCMC* API instead.

Example

```
>>> true_coefs = np.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):
...     coefs_mean = np.zeros(dim)
...     coefs = numpyro.sample('beta', dist.Normal(coefs_mean, np.ones(3)))
...     intercept = numpyro.sample('intercept', dist.Normal(0., 10.))
...     return numpyro.sample('y', dist.Bernoulli(logits=(coefs * data +␣
↪intercept).sum(-1)), obs=labels)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> init_params, potential_fn, constrain_fn = initialize_model(random.PRNGKey(0),
...                                                            model, data,␣
↪labels)
>>> init_kernel, sample_kernel = hmc(potential_fn, algo='NUTS')
>>> hmc_state = init_kernel(init_params,
...                         trajectory_length=10,
...                         num_warmup=300)
>>> samples = fori_collect(0, 500, sample_kernel, hmc_state,
...                        transform=lambda state: constrain_fn(state.z))
>>> print(np.mean(samples['beta'], axis=0))
[0.9153987 2.0754058 2.9621222]
```

init_kernel (*init_params, num_warmup, step_size=1.0, adapt_step_size=True, adapt_mass_matrix=True, dense_mass=False, target_accept_prob=0.8, trajectory_length=6.283185307179586, max_tree_depth=10, run_warmup=True, progbar=True, rng=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.
- **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.
- **run_warmup** (*bool*) – Flag to decide whether warmup is run. If `True`, *init_kernel* returns an initial *HMCState* that can be used to generate samples using MCMC. Else, returns the arguments and callable that does the initial adaptation.
- **progbar** (*bool*) – Whether to enable progress bar updates. Defaults to `True`.
- **rng** (*jax.random.PRNGKey*) – random key to be used as the source of randomness.

sample_kernel (*hmc_state*)

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).

Returns new proposed *HMCState* from simulating Hamiltonian dynamics given existing state.

HMCState = <class 'numpyro.mcmc.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`.
- **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 `AdaptState` 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** - random number generator seed used for the iteration.

1.2 MCMC Utilities

initialize_model(*rng*, *model*, **model_args*, *init_strategy=<function init_to_uniform>*, ***model_kwargs*)

Given a model with Pyro primitives, returns a function which, given unconstrained parameters, evaluates the potential energy (negative joint density). In addition, this also returns initial parameters sampled from the prior to initiate MCMC sampling and functions to transform unconstrained values at sample sites to constrained values within their respective support.

Parameters

- **rng** (*jax.random.PRNGKey*) – random number generator seed to sample from the prior. The returned *init_params* will have the batch shape `rng.shape[:-1]`.
- **model** – Python callable containing Pyro primitives.
- ***model_args** – args provided to the model.
- **init_strategy** (*callable*) – a per-site initialization function.
- ****model_kwargs** – kwargs provided to the model.

Returns tuple of (*init_params*, *potential_fn*, *constrain_fn*), *init_params* are values from the prior used to initiate MCMC, *constrain_fn* is a callable that uses inverse transforms to convert unconstrained HMC samples to constrained values that lie within the site's support.

fori_collect(*lower*, *upper*, *body_fun*, *init_val*, *transform=<function identity>*, *progbar=True*, ***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.
- ****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=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** (*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=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** (*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.

CHAPTER 2

Stochastic Variational Inference (SVI)

class SVI (*model*, *guide*, *loss*, *optim*, ***kwargs*)

Bases: `object`

Stochastic Variational Inference given an ELBo loss objective.

Parameters

- **model** – Python callable with Pyro primitives for the model.
- **guide** – Python callable with Pyro primitives for the guide (recognition network).
- **loss** – ELBo loss, i.e. negative Evidence Lower Bound, to minimize.
- **optim** – an instance of `_NumpyroOptim`.
- ****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*, *model_args=()*, *guide_args=()*)

Parameters

- **rng** (*jax.random.PRNGKey*) – random number generator seed.
- **model_args** (*tuple*) – arguments to the model (these can possibly vary during the course of fitting).
- **guide_args** (*tuple*) – arguments to the 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*, *model_args=()*, *guide_args=()*)
Take a single step of SVI (possibly on a batch / minibatch of data), using the optimizer.

Parameters

- **svi_state** – current state of SVI.
- **model_args** (*tuple*) – dynamic arguments to the model.
- **guide_args** (*tuple*) – dynamic arguments to the guide.

Returns tuple of *(svi_state, loss)*.

evaluate (*svi_state*, *model_args=()*, *guide_args=()*)
Take a single step of SVI (possibly on a batch / minibatch of data).

Parameters

- **svi_state** – current state of SVI.
- **model_args** (*tuple*) – arguments to the model (these can possibly vary during the course of fitting).
- **guide_args** (*tuple*) – arguments to the guide (these can possibly vary during the course of fitting).

Returns evaluate ELBo loss given the current parameter values (held within *svi_state.optim_state*).

2.1 ELBo

elbo (*param_map*, *model*, *guide*, *model_args*, *guide_args*, *kwargs*)
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.

Parameters

- **param_map** (*dict*) – dictionary of current parameter values keyed by site name.
- **model** – Python callable with Pyro primitives for the model.
- **guide** – Python callable with Pyro primitives for the guide (recognition network).
- **model_args** (*tuple*) – arguments to the model (these can possibly vary during the course of fitting).
- **guide_args** (*tuple*) – arguments to the guide (these can possibly vary during the course of fitting).
- **kwargs** (*dict*) – static keyword arguments to the model / guide.

Returns negative of the Evidence Lower Bound (ELBo) to be minimized.

CHAPTER 3

Base Distribution

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:

```
>>> d = dist.Dirichlet(np.ones((2, 3, 4)))
>>> d.batch_shape
(2, 3)
>>> d.event_shape
(4,)
```

arg_constraints = {}

support = None

reparametrized_params = []

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

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 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 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

transform_with_intermediates(*base_value*)

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.

3.2 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=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

sample_with_intermediates(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample_with_intermediates()*

transform_with_intermediates(*base_value*)

log_prob(*value*, *intermediates=None*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean

variance

CHAPTER 4

Continuous Distributions

4.1 Beta

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

Bases: *numpyro.distributions.distribution.Distribution*

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

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

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

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

mean

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

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=()*)

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

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 ob

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

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

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=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.6 Gamma

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

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

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

reparametrized_params = ['rate']

sample (*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob (*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.7 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

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

reparametrized_params = ['scale']

sample (*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob (*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.8 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=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.9 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=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.10 InverseGamma

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

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

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

reparametrized_params = ['rate']

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.11 LKJCholesky

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

Bases: *numpyro.distributions.distribution.Distribution*

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

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

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

Parameters

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

References

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

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

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

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

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

4.12 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

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

4.13 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=()*)

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

covariance_matrix

precision_matrix

mean

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

4.14 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':

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

reparametrized_params = ['loc', 'scale']

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

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

icdf(*q*)

mean

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

4.15 Pareto

class Pareto(*alpha*, *scale=1.0*, *validate_args=None*)

Bases: *numpyro.distributions.distribution.TransformedDistribution*

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

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

support

4.16 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>, 'lo

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

reparametrized_params = ['loc', 'scale']

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.17 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': <

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

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.18 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']

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

4.19 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>, 'lo

reparametrized_params = ['low', 'high']

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

CHAPTER 5

Discrete Distributions

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>

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

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

probs

mean

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

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>

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

5.4 Binomial

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

5.5 BinomialLogits

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

arg_constraints = {'logits': <numpyro.distributions.constraints._Real object>, 'total

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

probs

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

support

5.6 BinomialProbs

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

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

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

support

5.7 Categorical

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

5.8 CategoricalLogits

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

arg_constraints = {'logits': <numpyro.distributions.constraints._Real object>}

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

probs

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

support

5.9 CategoricalProbs

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

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

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

support

5.10 Delta

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

Bases: *numpyro.distributions.distribution.Distribution*

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

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

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

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*x*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

mean

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

5.11 Multinomial

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

5.12 MultinomialLogits

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

Bases: *numpyro.distributions.distribution.Distribution*

arg_constraints = {'logits': <numpyro.distributions.constraints._Real object>, 'total

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

See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)

See *numpyro.distributions.distribution.Distribution.log_prob()*

probs

mean

See *numpyro.distributions.distribution.Distribution.mean()*

variance

See *numpyro.distributions.distribution.Distribution.variance()*

support

5.13 MultinomialProbs

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

Bases: *numpyro.distributions.distribution.Distribution*

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

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

support

5.14 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>

sample(*key*, *sample_shape=()*)
See *numpyro.distributions.distribution.Distribution.sample()*

log_prob(*value*)
See *numpyro.distributions.distribution.Distribution.log_prob()*

mean
See *numpyro.distributions.distribution.Distribution.mean()*

variance
See *numpyro.distributions.distribution.Distribution.variance()*

5.15 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.

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

CHAPTER 6

Constraints

6.1 biject_to

biject_to(*constraint*)

6.2 boolean

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

6.3 corr_cholesky

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

6.4 dependent

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

6.5 greater_than

greater_than(*lower_bound*)

6.6 integer_interval

integer_interval(*lower_bound*, *upper_bound*)

6.7 integer_greater_than

integer_greater_than(*lower_bound*)

6.8 interval

interval(*lower_bound*, *upper_bound*)

6.9 lower_cholesky

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

6.10 multinomial

multinomial(*upper_bound*)

6.11 nonnegative_integer

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

6.12 positive

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

6.13 positive_definite

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

6.14 positive_integer

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

6.15 real

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

6.16 real_vector

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

6.17 simplex

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

6.18 unit_interval

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

CHAPTER 7

Transforms

7.1 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*)

7.2 AbsTransform

class AbsTransform

Bases: *numpyro.distributions.constraints.Transform*

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

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

inv(*y*)

7.3 AffineTransform

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

Bases: *numpyro.distributions.constraints.Transform*

codomain

event_dim

inv(*y*)

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

7.4 ComposeTransform

class ComposeTransform(*parts*)

Bases: *numpyro.distributions.constraints.Transform*

domain

codomain

event_dim

inv(*y*)

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

call_with_intermediates(*x*)

7.5 CorrCholeskyTransform

class CorrCholeskyTransform

Bases: *numpyro.distributions.constraints.Transform*

Transforms a uncontrained real vector x with length $D * (D-1)/2$ into the Cholesky factor of a D-dimension correlation matrix. This Cholesky factor is a lower triangular matrix with positive diagonals and unit Euclidean norm for each row. The transform is processed as follows:

1. First we convert x into a lower triangular matrix with the following order:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ x_0 & 1 & 0 & 0 \\ x_1 & x_2 & 1 & 0 \\ x_3 & x_4 & x_5 & 1 \end{bmatrix}$$

2. For each row X_i of the lower triangular part, we apply a *signed* version of class *StickBreakingTransform* to transform X_i into a unit Euclidean length vector using the following steps:

   a. Scales into the interval $(-1, 1)$ domain: $r_i = \tanh(X_i)$.

   b. Transforms into an unsigned domain: $z_i = r_i^2$.

   c. Applies $s_i = StickBreakingTransform(z_i)$.

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

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

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

event_dim = 1

inv(*y*)

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

7.6 ExpTransform

class ExpTransform(*domain=<numpyro.distributions.constraints._Real object>*)

Bases: *numpyro.distributions.constraints.Transform*

codomain

inv(*y*)

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

7.7 IdentityTransform

class IdentityTransform(*event_dim=0*)

Bases: *numpyro.distributions.constraints.Transform*

inv(*y*)

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

7.8 LowerCholeskyTransform

class LowerCholeskyTransform

Bases: *numpyro.distributions.constraints.Transform*

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

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

event_dim = 1

inv(*y*)

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

7.9 PermuteTransform

class PermuteTransform(*permutation*)

Bases: *numpyro.distributions.constraints.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*)

7.10 PowerTransform

class PowerTransform(*exponent*)

Bases: *numpyro.distributions.constraints.Transform*

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

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

inv(*y*)

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

7.11 SigmoidTransform

class SigmoidTransform

Bases: *numpyro.distributions.constraints.Transform*

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

inv(*y*)

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

7.12 StickBreakingTransform

class StickBreakingTransform

Bases: *numpyro.distributions.constraints.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*)

CHAPTER 8

Flows

8.1 InverseAutoregressiveTransform

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

Bases: *numpyro.distributions.constraints.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

CHAPTER 9

Pyro Primitives

9.1 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.

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

9.2 sample

sample(*name*, *fn*, *obs=None*, *sample_shape=()*)

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

Parameters

- **name** (*str*) – name of the sample site
- **fn** – Python callable
- **obs** (*numpy.ndarray*) – observed value
- **sample_shape** – Shape of samples to be drawn.

Returns sample from the stochastic *fn*.

9.3 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.

CHAPTER 10

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 *expected_log_likelihood* function computes the log likelihood for each draw from the joint posterior and aggregates the results, 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.

```
>>> N, D = 3000, 3
>>> def logistic_regression(data, labels):
...     coefs = numpyro.sample('coefs', dist.Normal(np.zeros(D), np.ones(D)))
...     intercept = numpyro.sample('intercept', dist.Normal(0., 10.))
...     logits = np.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 = np.arange(1., D + 1.)
>>> logits = np.sum(true_coefs * data, axis=-1)
>>> labels = dist.Bernoulli(logits=logits).sample(random.PRNGKey(1))

>>> num_warmup, num_samples = 1000, 1000
>>> mcmc = MCMC(NUTS(model=logistic_regression), num_warmup, num_samples)
>>> mcmc.run(random.PRNGKey(2), data, labels)
sample: 100%|| 1000/1000 [00:00<00:00, 1252.39it/s, 1 steps of size 5.83e-01. acc.␣
↪prob=0.85]
>>> mcmc.print_summary()
```

(continues on next page)

(continued from previous page)

```
                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, params, model, *args, **kwargs):
...     model = handlers.substitute(handlers.seed(model, rng), params)
...     model_trace = handlers.trace(model).get_trace(*args, **kwargs)
...     obs_node = model_trace['obs']
...     return np.sum(obs_node['fn'].log_prob(obs_node['value']))

>>> def expected_log_likelihood(rng, params, model, *args, **kwargs):
...     n = list(params.values())[0].shape[0]
...     log_lk_fn = vmap(lambda rng, params: log_likelihood(rng, params, model, *args,
↪ **kwargs))
...     log_lk_vals = log_lk_fn(random.split(rng, n), params)
...     return logsumexp(log_lk_vals) - np.log(n)

>>> print(expected_log_likelihood(random.PRNGKey(2), samples, logistic_regression,␣
↪data, labels))
-876.172
```

10.1 block

class block(*fn=None*, *hide_fn=<function block.<lambda>>*)

Bases: numpyro.handlers.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:

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

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

process_message(*msg*)

10.2 condition

class condition(*fn=None*, *param_map=None*, *substitute_fn=None*)

Bases: numpyro.handlers.Messenger

Conditions unobserved sample sites to values from *param_map* 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.
- **param_map** (*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:

```
>>> 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*)

10.3 replay

class replay(*fn*, *guide_trace*)

Bases: numpyro.handlers.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

```
>>> 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*)

10.4 scale

class scale(*fn=None*, *scale_factor=1.0*)
Bases: numpyro.handlers.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_factor** (*float*) – a positive scaling factor

process_message(*msg*)

10.5 seed

class seed(*fn*, *rng*)
Bases: numpyro.handlers.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.

process_message(*msg*)

10.6 substitute

class substitute(*fn=None*, *param_map=None*, *base_param_map=None*, *substitute_fn=None*)
Bases: numpyro.handlers.Messenger

Given a callable *fn* and a dict *param_map* keyed by site names (alternatively, a callable *substitute_fn*), return a callable which substitutes all primitive calls in *fn* with values from *param_map* whose key matches the site name. If the site name is not present in *param_map*, 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.
- **param_map** (*dict*) – dictionary of *numpy.ndarray* values keyed by site names.
- **base_param_map** (*dict*) – similar to *param_map* but only holds samples from base distributions.
- **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:

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

(continues on next page)

(continued from previous page)

```
>>> model = seed(model, random.PRNGKey(0))
>>> exec_trace = trace(substitute(model, {'a': -1})).get_trace()
>>> assert exec_trace['a']['value'] == -1
```

process_message(*msg*)

10.7 trace

class trace(*fn=None*)

Bases: numpyro.handlers.Messenger

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

Example

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

>>> exec_trace = trace(seed(model, random.PRNGKey(0))).get_trace()
>>> pp.pprint(exec_trace)
OrderedDict([('a',
              {'args': (),
               'fn': <numpyro.distributions.continuous.Normal object at␣
↪0x7f9e689b1eb8>,
               'is_observed': False,
               'kwargs': {'random_state': DeviceArray([0, 0], dtype=uint32)},
               'name': 'a',
               'type': 'sample',
               'value': DeviceArray(-0.20584235, dtype=float32)})])
```

postprocess_message(*msg*)

get_trace(**args*, ***kwargs*)

Run the wrapped callable and return the recorded trace.

Parameters

- ***args** – arguments to the callable.
- ****kwargs** – keyword arguments to the callable.

Returns *OrderedDict* containing the execution trace.

CHAPTER 11

NumPyro Optimizers

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

11.1 Adam

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

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

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.

11.2 Adagrad

class Adagrad(**args*, ***kwargs*)
Wrapper class for the JAX optimizer: `adagrad()`

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.

11.3 Momentum

class Momentum(**args*, ***kwargs*)
Wrapper class for the JAX optimizer: `momentum()`

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.

11.4 RMSProp

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

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.

11.5 RMSPropMomentum

class RMSPropMomentum(**args*, ***kwargs*)
Wrapper class for the JAX optimizer: rmsprop_momentum()

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.

11.6 SGD

class SGD(**args*, ***kwargs*)
Wrapper class for the JAX optimizer: `sgd()`

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.

11.7 SM3

class SM3(**args*, ***kwargs*)
Wrapper class for the JAX optimizer: `sm3()`

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.

CHAPTER 12

Diagnostics

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

12.1 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

12.2 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

12.3 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

12.4 Gelman Rubin

gelman_rubin(*x*)

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

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

Returns R-hat of `x`.

Return type numpy.ndarray

12.5 Split Gelman Rubin

split_gelman_rubin(*x*)

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

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

Returns split R-hat of `x`.

Return type numpy.ndarray

12.6 HPDI

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

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

Parameters

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

Returns quantiles of `x` at `(1 - prob) / 2` and `(1 + prob) / 2`.

Return type numpy.ndarray

12.7 Summary

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

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.

CHAPTER 13

Inference Utilities

13.1 predictive

predictive(*rng*, *model*, *posterior_samples*, *return_sites=None*, **args*, ***kwargs*)

Run model by sampling latent parameters from *posterior_samples*, and return values at sample sites from the forward run. By default, only sites not contained in *posterior_samples* are returned. This can be modified by changing the *return_sites* keyword argument.

> **Warning:** The interface for the *predictive* function is experimental, and might change in the future.

Parameters

- **rng** (*jax.random.PRNGKey*) – seed to draw samples
- **model** – Python callable containing Pyro primitives.
- **posterior_samples** (*dict*) – dictionary of samples from the posterior.
- **return_sites** (*list*) – sites to return; by default only sample sites not present in *posterior_samples* are returned.
- **args** – model arguments.
- **kwargs** – model kwargs.

Returns dict of samples from the predictive distribution.

13.2 log_density

log_density(*model*, *model_args*, *model_kwargs*, *params*, *skip_dist_transforms=False*)

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.
- **skip_dist_transforms** (*bool*) – whether to compute log probability of a site (if its prior is a transformed distribution) in its base distribution domain.

Returns log of joint density and a corresponding model trace

13.3 transform_fn

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

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.

13.4 constrain_fn

constrain_fn(*model*, *model_args*, *model_kwargs*, *transforms*, *params*)

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.
- **transforms** (*dict*) – dictionary of transforms keyed by names. Names in *transforms* and *params* should align.
- **params** (*dict*) – dictionary of unconstrained values keyed by site names.

Returns *dict* of transformed params.

13.5 potential_energy

potential_energy(*model*, *model_args*, *model_kwargs*, *inv_transforms*, *params*)

Makes a callable which 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.
- **inv_transforms** (*dict*) – dictionary of transforms keyed by names.

Returns a callable that computes potential energy given unconstrained parameters.

13.6 init_to_median

init_to_median(*site*, *num_samples=15*, *skip_param=False*)
Initialize to the prior median.

13.7 init_to_prior

init_to_prior(*site*, *skip_param=False*)
Initialize to a prior sample.

13.8 init_to_uniform

init_to_uniform(*site*, *radius=2*, *skip_param=False*)
Initialize to an arbitrary feasible point, ignoring distribution parameters.

13.9 init_to_feasible

init_to_feasible(*site*, *skip_param=False*)
Initialize to an arbitrary feasible point, ignoring distribution parameters.

13.10 find_valid_initial_params

find_valid_initial_params(*rng*, *model*, **model_args*, *init_strategy=<function init_to_uniform>*, *param_as_improper=False*, *prototype_params=None*, ***model_kwargs*)
Given a model with Pyro primitives, returns an initial valid unconstrained parameters. This function also returns an *is_valid* flag to say whether the initial parameters are valid.

Parameters

- **rng** (*jax.random.PRNGKey*) – random number generator seed to sample from the prior. The returned *init_params* will have the batch shape `rng.shape[:-1]`.
- **model** – Python callable containing Pyro primitives.

- ***model_args** – args provided to the model.
- **init_strategy** (*callable*) – a per-site initialization function.
- **param_as_improper** (*bool*) – a flag to decide whether to consider sites with *param* statement as sites with improper priors.
- ****model_kwargs** – kwargs provided to the model.

Returns tuple of (*init_params*, *is_valid*).

CHAPTER 14

Automatic Guide Generation

14.1 AutoDiagonalNormal

class AutoDiagonalNormal(*model*, *prefix='auto'*, *init_strategy=<function init_to_median>*)

Bases: `numpyro.contrib.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(rng, model, ...)
svi = SVI(model, guide, ...)
```

median(*params*)

Returns the posterior median value of each latent variable.

Parameters **params** (*dict*) – A dict containing parameter values.

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

- **opt_state** – Current state of the optimizer.
- **quantiles** (*torch.Tensor or 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

14.2 AutoIAFNormal

class AutoIAFNormal(*model*, *prefix='auto'*, *init_strategy=<function init_to_median>*, *num_flows=3*, ***arn_kwargs*)

Bases: `numpyro.contrib.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(rng, model, get_params, hidden_dims=[20], skip_
↪connections=True, ...)
svi_init, svi_update, _ = svi(model, guide, ...)
```

Parameters

- **rng** (*jax.random.PRNGKey*) – random key to be used as the source of randomness to initialize the guide.
- **model** (*callable*) – a generative model.
- **prefix** (*str*) – a prefix that will be prefixed to all param internal sites.
- **init_strategy** (*callable*) – A per-site initialization function.
- **num_flows** (*int*) – the number of flows to be used, defaults to 3.
- ****arn_kwargs** – keywords for constructing autoregressive neural networks, which includes:
  - **hidden_dims** (`list[int]`) - the dimensionality of the hidden units per layer. Defaults to `[latent_size, latent_size]`.
  - **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.Relu()`.

CHAPTER 15

Indices and tables

- genindex
- search

Python Module Index

n

Index

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V