Numpyro Documentation

Release 0.0

Uber AI Labs

Jun 03, 2019

Inference:

CHAPTER 1

Markov Chain Monte Carlo (MCMC)

1.1 Hamiltonian Monte Carlo

mcmc(*num_warmup*, *num_samples*, *init_params*, *sampler='hmc'*, *constrain_fn=None*, *print_summary=True*, ***sampler_kwargs*)

Convenience wrapper for MCMC samplers – runs warmup, prints diagnostic summary and returns a collections of samples from the posterior.

Parameters

- **num_warmup** – Number of warmup steps.
- **num_samples** – Number of samples to generate from the Markov chain.
- **init_params** – Initial parameters to begin sampling. The type can must be consistent with the input type to *potential_fn*.
- **sampler** – currently, only *hmc* is implemented (default).
- **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.
- **print_summary** – Whether to print diagnostics summary for each sample site. Default is `True`.
- ****sampler_kwargs** – Sampler specific keyword arguments.
  - *HMC*: Refer to *hmc()* and *init_kernel()* for accepted arguments. Note that all arguments must be provided as keywords.

Returns collection of samples from the posterior.

```
>>> 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))
```

(continues on next page)

(continued from previous page)

```
>>>
>>> def model(data, labels):
...     coefs_mean = np.zeros(dim)
...     coefs = sample('beta', dist.Normal(coefs_mean, np.ones(3)))
...     intercept = sample('intercept', dist.Normal(0., 10.))
...     return sample('y', dist.Bernoulli(logits=(coefs * data + intercept).sum(-
↪1)), obs=labels)
>>>
>>> init_params, potential_fn, constrain_fn = initialize_model(random.PRNGKey(0),␣
↪model,
...                                                          data, labels)
>>> num_warmup, num_samples = 1000, 1000
>>> samples = mcmc(num_warmup, num_samples, init_params,
...                potential_fn=potential_fn,
...                constrain_fn=constrain_fn)
warmup: 100%|| 1000/1000 [00:09<00:00, 109.40it/s, 1 steps of size 5.83e-01. acc.␣
↪prob=0.79]
sample: 100%|| 1000/1000 [00:00<00:00, 1252.39it/s, 1 steps of size 5.83e-01. acc.
↪ prob=0.85]


                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
```

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.

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 = sample('beta', dist.Normal(coefs_mean, np.ones(3)))
...     intercept = sample('intercept', dist.Normal(0., 10.))
...     return sample('y', dist.Bernoulli(logits=(coefs * data + intercept).sum(-
↪1)), obs=labels)
>>>
>>> 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(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_steps** (*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`.
- **heuristic_step_size** (*bool*) – If `True`, a coarse grained adjustment of step size is done at the beginning of each adaptation window to achieve *target_acceptance_prob*.
- **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).
- **step_size** - Step size to be used by the integrator in the next iteration. This is adapted during warmup.
- **inverse_mass_matrix** - The inverse mass matrix to be be used for the next iteration. This is adapted during warmup.
- **rng** - random number generator seed used for the iteration.

1.2 MCMC Utilities

initialize_model(*rng*, *model*, **model_args*, *init_strategy='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.
- **model** – Python callable containing Pyro primitives.
- ***model_args** – args provided to the model.

- **init_strategy** (*str*) – initialization strategy - *uniform* initializes the unconstrained parameters by drawing from a *Uniform(-2, 2)* distribution (as used by Stan), whereas *prior* initializes the parameters by sampling from the prior for each of the sample sites.
- ****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(*n*, *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, in some cases, *progbar=False* can be faster, when collecting a lot of samples. Refer to example usage in *hmc()*.

Parameters

- **n** (*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
- **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.

summary(*samples*, *prob=0.89*)

Prints a summary table displaying diagnostics of `samples` from the posterior. The diagnostics displayed are mean, standard deviation, the 89% Credibility Interval, *effective_sample_size()* *split_gelman_rubin()*.

Parameters

- **samples** – a collection of input samples.
- **prob** (*float*) – the probability mass of samples within the HPDI interval.

CHAPTER 2

Stochastic Variational Inference (SVI)

svi (*model*, *guide*, *loss*, *optim_init*, *optim_update*, *get_params*, ***kwargs*)
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_init** – initialization function returned by a JAX optimizer. see: `jax.experimental.optimizers`.
- **optim_update** – update function for the optimizer
- **get_params** – function to get current parameters values given the optimizer state.
- ****kwargs** – static arguments for the model / guide, i.e. arguments that remain constant during fitting.

Returns tuple of *(init_fn, update_fn, evaluate)*.

init_fn (*rng*, *model_args=()*, *guide_args=()*, *params=None*)

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).
- **params** (*dict*) – initial parameter values to condition on. This can be useful forx

Returns initial optimizer state.

update_fn (*i*, *opt_state*, *rng*, *model_args=()*, *guide_args=()*)
Take a single step of SVI (possibly on a batch / minibatch of data), using the optimizer.

Parameters

- **i** (*int*) – represents the i'th iteration over the epoch, passed as an argument to the optimizer's update function.
- **opt_state** – current optimizer state.
- **rng** (*jax.random.PRNGKey*) – random number generator seed.
- **model_args** (*tuple*) – dynamic arguments to the model.
- **guide_args** (*tuple*) – dynamic arguments to the guide.

Returns tuple of *(loss_val, opt_state, rng)*.

evaluate(*opt_state*, *rng*, *model_args=()*, *guide_args=()*)

Take a single step of SVI (possibly on a batch / minibatch of data).

Parameters

- **opt_state** – current optimizer state.
- **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 evaluate ELBo loss given the current parameter values (held within *opt_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 variablbes 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.
- **size** – the sample shape for the distribution.

Returns a *numpy.ndarray* of shape *sample_shape* + *batch_shape* + *event_shape*

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 a *numpy.ndarray* with shape *value.shape[:-self.event_shape]*

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

is_reparametrized

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

log_prob(*value*)
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>

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._Real 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.TransformedDistribution*

reparametrized_params = ['scale']

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

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.TransformedDistribution*

reparametrized_params = ['scale']

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

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 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()*

4.11 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()*

4.12 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()*

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

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

4.13 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

4.14 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()*

4.15 TruncatedCauchy

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

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

reparametrized_params = ['low', '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()*

support

4.16 TruncatedNormal

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

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

reparametrized_params = ['low', '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()*

support

4.17 Uniform

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

arg_constraints = {'high': <numpyro.distributions.constraints._Dependent object>, 'lo

reparametrized_params = ['low', 'high']

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

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 Multinomial

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

5.11 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

5.12 MultinomialProbs

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

Bases: *numpyro.distributions.distribution.Distribution*

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

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

5.13 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()*

CHAPTER 6

Pyro Primitives

param(*name*, *init_value*)

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.

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

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

Returns sample from the stochastic *fn*.

CHAPTER 7

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 = sample('coefs', dist.Normal(np.zeros(D), np.ones(D)))
...     intercept = sample('intercept', dist.Normal(0., 10.))
...     logits = np.sum(coefs * data + intercept, axis=-1)
...     return 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))

>>> init_params, potential_fn, constrain_fn = initialize_model(random.PRNGKey(2),␣
↪logistic_regression, data, labels)
>>> num_warmup, num_samples = 1000, 1000
>>> samples = mcmc(num_warmup, num_samples, init_params,
...                potential_fn=potential_fn,
...                constrain_fn=constrain_fn)
warmup: 100%|| 1000/1000 [00:09<00:00, 109.40it/s, 1 steps of size 5.83e-01. acc.␣
↪prob=0.79]
```

(continues on next page)

(continued from previous page)

```
sample: 100%|| 1000/1000 [00:00<00:00, 1252.39it/s, 1 steps of size 5.83e-01. acc.␣
↪prob=0.85]


                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 = substitute(seed(model, rng), params)
...     model_trace = 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
```

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 = sample('a', dist.Normal(0., 1.))
...     return 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*)

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():
...     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*)

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*)

class substitute(*fn=None*, *param_map=None*)

Bases: numpyro.handlers.Messenger

Given a callable *fn* and a dict *param_map* keyed by site names, 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.

Parameters

- **fn** – Python callable with NumPyro primitives.
- **param_map** (*dict*) – dictionary of *numpy.ndarray* values keyed by site names.

Example:

```
>>> def model():
...     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*)

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():
...     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 8

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

CHAPTER 9

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

CHAPTER 10

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

CHAPTER 11

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 `input.shape[0] >= 2` and `input.shape[1] >= 2`.

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

Returns R-hat of `x`.

Return type numpy.ndarray

CHAPTER 12

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 `input.shape[1] >= 4`.

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

Returns split R-hat of `x`.

Return type numpy.ndarray

CHAPTER 13

HPDI

hpdi (*x*, *prob=0.89*, *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 `input` at `(1 - probs) / 2` and `(1 + probs) / 2`.

Return type numpy.ndarray

CHAPTER 14

Summary

summary(*samples*, *prob=0.89*)

Prints a summary table displaying diagnostics of `samples` from the posterior. The diagnostics displayed are mean, standard deviation, the 89% Credibility Interval, *effective_sample_size()* *split_gelman_rubin()*.

Parameters

- **samples** – a collection of input samples.
- **prob** (*float*) – the probability mass of samples within the HPDI interval.

CHAPTER 15

Indices and tables

- genindex
- search

Python Module Index

n

Index

F

G

H

I

L

M

N

P

R

S

T

U

V