Pyro Documentation

Release 0.1.0

Uber AI Labs

Feb 25, 2018

Contents:

CHAPTER 1

Installation

1.1 Install from Source

Pyro supports Python 2.7.* and Python 3. To setup, install PyTorch then run:

```
pip install pyro-ppl
```

or install from source:

```
git clone https://github.com/uber/pyro.git
cd pyro
python setup.py install
```

Warning: Some bleeding-edge features of Pyro (e.g. *enum_discrete*) require a very recent version of PyTorch. We recommend installing PyTorch from source using the *master* branch. See PyTorch install instructions for details.

CHAPTER 2

Getting Started

- Install Pyro.
- Learn the basic concepts of Pyro: models and inference.
- Dive in to other tutorials and examples.

CHAPTER 3

Primitives

clear_param_store()
Clears the ParamStore. This is especially useful if you're working in a REPL.

get_param_store()
Returns the ParamStore

iarange(**args*, ***kwds*)
Context manager for conditionally independent ranges of variables.

`iarange` is similar to `torch.arange` in that it yields an array of indices by which other tensors can be indexed. `iarange` differs from `torch.arange` in that it also informs inference algorithms that the variables being indexed are conditionally independent. To do this, `iarange` is a provided as context manager rather than a function, and users must guarantee that all computation within an `iarange` context is conditionally independent:

```
with iarange("name", size) as ind:
    # ...do conditionally independent stuff with ind...
```

Additionally, `iarange` can take advantage of the conditional independence assumptions by subsampling the indices and informing inference algorithms to scale various computed values. This is typically used to subsample minibatches of data:

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

By default `subsample_size=False` and this simply yields a `torch.arange(0, size)`. If `0 < subsample_size <= size` this yields a single random batch of indices of size `subsample_size` and scales all log likelihood terms by `size/batch_size`, within this context.

Warning: This is only correct if all computation is conditionally independent within the context.

Parameters

- **name** (*str*) – A unique name to help inference algorithms match `iarange` sites between models and guides.
- **size** (*int*) – Optional size of the collection being subsampled (like *stop* in builtin *range*).
- **subsample_size** (*int*) – Size of minibatches used in subsampling. Defaults to *size*.
- **subsample** (Anything supporting *len()*.) – Optional custom subsample for user-defined subsampling schemes. If specified, then *subsample_size* will be set to *len(subsample)*.
- **use_cuda** (*bool*) – Optional bool specifying whether to use cuda tensors for *subsample* and *log_pdf*. Defaults to *torch.Tensor.is_cuda*.

Returns A context manager yielding a single 1-dimensional *torch.Tensor* of indices.

Examples:

```
# This version simply declares independence:
>>> with iarange('data'):
        observe('obs', normal, data, mu, sigma)

# This version subsamples data in vectorized way:
>>> with iarange('data', 100, subsample_size=10) as ind:
        observe('obs', normal, data.index_select(0, ind), mu, sigma)

# This wraps a user-defined subsampling method for use in pyro:
>>> ind = my_custom_subsample
>>> with iarange('data', 100, subsample=ind):
        observe('obs', normal, data.index_select(0, ind), mu, sigma)
```

See SVI Part II for an extended discussion.

irange(*name*, *size*, *subsample_size=None*, *subsample=None*, *use_cuda=None*)

Non-vectorized version of `iarange`. See `iarange` for details.

Parameters

- **name** (*str*) – A name that will be used for this site in a Trace.
- **size** (*int*) – The size of the collection being subsampled (like `stop` in builtin `range`).
- **subsample_size** (*int*) – Size of minibatches used in subsampling. Defaults to `size`.
- **subsample** (Anything supporting `len()`.) – Optional custom subsample for user-defined subsampling schemes. If specified, then `subsample_size` will be set to `len(subsample)`.
- **use_cuda** (*bool*) – Optional bool specifying whether to use cuda tensors for internal `log_pdf` computations. Defaults to `torch.Tensor.is_cuda`.

Returns A generator yielding a sequence of integers.

Examples:

```
>>> for i in irange('data', 100, subsample_size=10):
        if z[i]:  # Prevents vectorization.
            observe('obs_{}'.format(i), normal, data[i], mu, sigma)
```

See SVI Part II for an extended discussion.

module(*name*, *nn_module*, *tags='default'*, *update_module_params=False*)

Takes a torch.nn.Module and registers its parameters with the ParamStore. In conjunction with the ParamStore save() and load() functionality, this allows the user to save and load modules.

Parameters

- **name** (*str*) – name of module
- **nn_module** (*torch.nn.Module*) – the module to be registered with Pyro
- **tags** (*string or iterable of strings*) – optional; tags to associate with any parameters inside the module
- **update_module_params** – determines whether Parameters in the PyTorch module get overridden with the values found in the ParamStore (if any). Defaults to *False*

Returns torch.nn.Module

observe(*name*, *fn*, *obs*, **args*, ***kwargs*)

Alias of *pyro.sample(name, fn, *args, obs=obs, **kwargs)*.

Parameters

- **name** – name of observation
- **fn** – distribution class or function
- **obs** – observed datum

Returns sample

param(*name*, **args*, ***kwargs*)

Saves the variable as a parameter in the param store. To interact with the param store or write to disk, see Parameters.

Parameters **name** – name of parameter

Returns parameter

random_module(*name*, *nn_module*, *prior*, **args*, ***kwargs*)

Places a prior over the parameters of the module *nn_module*. Returns a distribution (callable) over *nn.Module's, which upon calling returns a sampled 'nn.Module*.

See the Bayesian Regression tutorial for an example.

Parameters

- **name** (*str*) – name of pyro module
- **nn_module** (*torch.nn.Module*) – the module to be registered with pyro
- **prior** – pyro distribution, stochastic function, or python dict with parameter names as keys and respective distributions/stochastic functions as values.

Returns a callable which returns a sampled module

sample(*name*, *fn*, **args*, ***kwargs*)

Calls the stochastic function *fn* with additional side-effects depending on *name* and the enclosing context (e.g. an inference algorithm). See Intro I and Intro II for a discussion.

Parameters

- **name** – name of sample
- **fn** – distribution class or function
- **obs** – observed datum (optional; should only be used in context of inference) optionally specified in kwargs
- **baseline** (*dict*) – Optional dictionary of baseline parameters specified in kwargs. See inference documentation for details.

Returns sample

CHAPTER 4

Inference

In the context of probabilistic modeling, learning is usually called inference. In the particular case of Bayesian inference, this often involves computing (approximate) posterior distributions. In the case of parameterized models, this usually involves some sort of optimization. Pyro supports multiple inference algorithms, with support for stochastic variational inference (SVI) being the most extensive. Look here for more inference algorithms in future versions of Pyro.

See Intro II for a discussion of inference in Pyro.

4.1 SVI

class SVI(*model*, *guide*, *optim*, *loss*, *loss_and_grads=None*, **args*, ***kwargs*)
Bases: object

Parameters

- **model** – the model (callable containing Pyro primitives)
- **guide** – the guide (callable containing Pyro primitives)
- **optim** (*pyro.optim.PyroOptim*) – a wrapper a for a PyTorch optimizer
- **loss** – this is either a string that specifies the loss function to be used (currently the only supported built-in loss is 'ELBO') or a user-provided loss function; in the case this is a built-in loss *loss_and_grads* will be filled in accordingly
- **loss_and_grads** – if specified, this user-provided callable computes gradients for use in *step()* and marks which parameters in the param store are to be optimized

A unified interface for stochastic variational inference in Pyro. Most users will interact with *SVI* with the argument *loss="ELBO"*. See the tutorial SVI Part I for a discussion.

evaluate_loss(**args*, ***kwargs*)

Returns estimate of the loss

Return type float

Evaluate the loss function. Any args or kwargs are passed to the model and guide.

step (**args*, ***kwargs*)

Returns estimate of the loss

Return type float

Take a gradient step on the loss function (and any auxiliary loss functions generated under the hood by *loss_and_grads*). Any args or kwargs are passed to the model and guide

4.2 ELBO

class ELBO (*num_particles=1*, *trace_graph=False*, *enum_discrete=False*)

Bases: object

Parameters

- **num_particles** – the number of particles (samples) used to form the ELBO estimator.
- **trace_graph** – boolean. whether to keep track of dependency information when running the model and guide. this information can be used to form a gradient estimator with lower variance in the case that some of the random variables are non-reparameterized. note: for a model with many random variables, keeping track of the dependency information can be expensive. see the tutorial SVI Part III for a discussion.
- **enum_discrete** (*bool*) – whether to sum over discrete latent variables, rather than sample them.

ELBO is the top-level interface for stochastic variational inference via optimization of the evidence lower bound. Most users will not interact with *ELBO* directly; instead they will interact with *SVI*. *ELBO* dispatches to *Trace_ELBO* and *TraceGraph_ELBO*, where the internal implementations live.

Warning: *enum_discrete* is a bleeding edge feature. see SS-VAE for a discussion.

References

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

[2] *Black Box Variational Inference*, Rajesh Ranganath, Sean Gerrish, David M. Blei

loss (*model*, *guide*, **args*, ***kwargs*)

Evaluates the ELBO with an estimator that uses *num_particles* many samples/particles, where *num_particles* is specified in the constructor.

Returns returns an estimate of the ELBO

Return type float

loss_and_grads (*model*, *guide*, **args*, ***kwargs*)

Computes the ELBO as well as the surrogate ELBO that is used to form the gradient estimator. Performs backward on the latter. Num_particle many samples are used to form the estimators, where *num_particles* is specified in the constructor.

Returns returns an estimate of the ELBO

Return type float

4.3 Importance

class Importance(*model*, *guide=None*, *num_samples=None*)
Bases: pyro.infer.abstract_infer.TracePosterior

Parameters

- **model** – probabilistic model defined as a function
- **guide** – guide used for sampling defined as a function
- **num_samples** – number of samples to draw from the guide (default 10)

This method performs posterior inference by importance sampling using the guide as the proposal distribution. If no guide is provided, it defaults to proposing from the model's prior.

4.4 Search

class Search(*model*, *max_tries=1000000.0*)
Bases: pyro.infer.abstract_infer.TracePosterior

Trace and Poutine-based implementation of systematic search.

Parameters

- **model** (*callable*) – Probabilistic model defined as a function.
- **max_tries** (*int*) – The maximum number of times to try completing a trace from the queue.

CHAPTER 5

Distributions

5.1 Primitive Distributions

class Distribution(*reparameterized=None*)

Bases: object

Base class for parameterized probability distributions.

Distributions in Pyro are stochastic function objects with `.sample()` and `.log_pdf()` methods. Pyro provides two versions of each stochastic function:

(i) lowercase versions that take parameters:

```
x = dist.bernoulli(param)              # Returns a sample of size size(param).
p = dist.bernoulli.log_pdf(x, param)   # Evaluates log probability of x.
```

and *(ii)* UpperCase distribution classes that can construct stochastic functions with fixed parameters:

```
d = dist.Bernoulli(param)
x = d()                                # Samples a sample of size size(param).
p = d.log_pdf(x)                       # Evaluates log probability of x.
```

Under the hood the lowercase versions are aliases for the UpperCase versions.

Note: Parameters and data should be of type *torch.autograd.Variable* and all methods return type *torch.autograd.Variable* unless otherwise noted.

Tensor Shapes:

Distributions provide a method `.shape()` for the tensor shape of samples:

```
x = d.sample(*args, **kwargs)
assert x.shape == d.shape(*args, **kwargs)
```

Pyro distinguishes two different roles for tensor shapes of samples:

- The leftmost dimension corresponds to iid *batching*, which can be treated specially during inference via the `.batch_log_pdf()` method.
- The rightmost dimensions correspond to *event shape*.

These shapes are related by the equation:

```
assert d.shape(*args, **kwargs) == (d.batch_shape(*args, **kwargs) +
                                    d.event_shape(*args, **kwargs))
```

There are exceptions, for instance, in the case of the Categorical distribution, without one hot encoding.

Distributions provide a vectorized `.batch_log_pdf()` method that evaluates the log probability density of each event in a batch independently, returning a tensor of shape `d.batch_shape(x) + (1,)`:

```
x = d.sample(*args, **kwargs)
assert x.shape == d.shape(*args, **kwargs)
log_p = d.batch_log_pdf(x, *args, **kwargs)
assert log_p.shape == d.batch_shape(*args, **kwargs) + (1,)
```

Distributions may also support broadcasting of the `.log_pdf()` and `.batch_log_pdf()` methods, which may each be evaluated with a sample tensor *x* that is larger than (but broadcastable from) the parameters. In this case, `d.batch_shape(x)` will return the shape of the broadcasted batch shape using the data tensor *x*:

```
x = d.sample()
xx = torch.stack([x, x])
d.batch_log_pdf(xx).size() == d.batch_shape(xx) + (1,))  # returns True
```

Implementing New Distributions:

Derived classes must implement the following methods: `.sample()`, `.batch_log_pdf()`, `.batch_shape()`, and `.event_shape()`. Discrete classes may also implement the `.enumerate_support()` method to improve gradient estimates and set `.enumerable = True`.

Examples:

Take a look at the examples to see how they interact with inference algorithms.

analytic_mean(**args*, ***kwargs*)

Analytic mean of the distribution, to be implemented by derived classes.

Note that this is optional, and currently only used for testing distributions.

Returns Analytic mean.

Return type torch.autograd.Variable.

Raises NotImplementedError if mean cannot be analytically computed.

analytic_var(**args*, ***kwargs*)

Analytic variance of the distribution, to be implemented by derived classes.

Note that this is optional, and currently only used for testing distributions.

Returns Analytic variance.

Return type torch.autograd.Variable.

Raises NotImplementedError if variance cannot be analytically computed.

batch_log_pdf(*x*, **args*, ***kwargs*)

Evaluates log probability densities for each of a batch of samples.

Parameters **x** (*torch.autograd.Variable*) – A single value or a batch of values batched along axis 0.

Returns log probability densities as a one-dimensional *torch.autograd.Variable* with same batch size as value and params. The shape of the result should be *self.batch_size()*.

Return type torch.autograd.Variable

batch_shape(*x=None*, **args*, ***kwargs*)

The left-hand tensor shape of samples, used for batching.

Samples are of shape `d.shape(x) == d.batch_shape(x) + d.event_shape()`.

Parameters **x** – Data that is used to determine the batch shape. This is optional. If not specified, the distribution parameters are used to determine the shape of the batch that is returned from `sample()`.

Returns Tensor shape used for batching.

Return type torch.Size

Raises ValueError if the parameters are not broadcastable to the data shape

enumerable = False

enumerate_support(**args*, ***kwargs*)

Returns a representation of the parametrized distribution's support.

This is implemented only by discrete distributions.

Returns An iterator over the distribution's discrete support.

Return type iterator

event_dim(**args*, ***kwargs*)

Returns Number of dimensions of individual events.

Return type int

event_shape(*x=None*, **args*, ***kwargs*)

The right-hand tensor shape of samples, used for individual events. The event dimension(/s) is used to designate random variables that could potentially depend on each other, for instance in the case of Dirichlet or the categorical distribution, but could also simply be used for logical grouping, for example in the case of a normal distribution with a diagonal covariance matrix.

Samples are of shape *d.shape(x) == d.batch_shape(x) + d.event_shape()*.

Returns Tensor shape used for individual events.

Return type torch.Size

log_pdf(*x*, **args*, ***kwargs*)

Evaluates total log probability density of a batch of samples.

Parameters **x** (*torch.autograd.Variable*) – A value.

Returns total log probability density as a one-dimensional torch.autograd.Variable of size 1.

Return type torch.autograd.Variable

reparameterized = False

sample(**args*, ***kwargs*)

Samples a random value.

For tensor distributions, the returned Variable should have the same *.size()* as the parameters, unless otherwise noted.

Returns A random value or batch of random values (if parameters are batched). The shape of the result should be *self.size()*.

Return type torch.autograd.Variable

shape (*x=None*, **args*, ***kwargs*)
The tensor shape of samples from this distribution.

Samples are of shape *d.shape(x) == d.batch_shape(x) + d.event_shape()*.

Returns Tensor shape of samples.

Return type torch.Size

5.1.1 Bernoulli

class Bernoulli (*ps=None*, *logits=None*, *batch_size=None*, *log_pdf_mask=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Bernoulli distribution.

Distribution over a vector of independent Bernoulli variables. Each element of the vector takes on a value in *{0, 1}*.

This is often used in conjunction with *torch.nn.Sigmoid* to ensure the *ps* parameters are in the interval *[0, 1]*.

Parameters

- **ps** (*torch.autograd.Variable*) – Probabilities. Should lie in the interval *[0,1]*.
- **logits** – Log odds, i.e. $\log(\frac{p}{1-p})$. Either *ps* or *logits* should be specified, but not both.
- **batch_size** – The number of elements in the batch used to generate a sample. The batch dimension will be the leftmost dimension in the generated sample.
- **log_pdf_mask** – Tensor that is applied to the batch log pdf values as a multiplier. The most common use case is supplying a boolean tensor mask to mask out certain batch sites in the log pdf computation.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*.

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*.

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*.

enumerable = True

enumerate_support()
Returns the Bernoulli distribution's support, as a tensor along the first dimension.

Note that this returns support values of all the batched RVs in lock-step, rather than the full cartesian product. To iterate over the cartesian product, you must construct univariate Bernoullis and use itertools.product() over all univariate variables (may be expensive).

Returns torch variable enumerating the support of the Bernoulli distribution. Each item in the return value, when enumerated along the first dimensions, yields a value from the distribution's support which has the same dimension as would be returned by sample.

Return type torch.autograd.Variable.

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*.

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*.

5.1.2 Beta

class Beta(*alpha*, *beta*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Univariate beta distribution parameterized by *alpha* and *beta*.

This is often used in conjunction with *torch.nn.Softplus* to ensure *alpha* and *beta* parameters are positive.

Parameters

- **alpha** (*torch.autograd.Variable*) – Lower shape parameter. Should be positive.
- **beta** (*torch.autograd.Variable*) – Upper shape parameter. Should be positive.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*.

sample()
Ref: pyro.distributions.distribution.Distribution.sample.()

5.1.3 Categorical

class Categorical(*ps=None*, *vs=None*, *logits=None*, *one_hot=True*, *batch_size=None*, *log_pdf_mask=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Categorical (discrete) distribution.

Discrete distribution over elements of *vs* with $P(vs[i]) \propto ps[i]$. If `one_hot=True`, `.sample()` returns a one-hot vector; otherwise `.sample()` returns the category index.

Parameters

- **ps** (*torch.autograd.Variable*) – Probabilities. These should be non-negative and normalized along the rightmost axis.

- **logits** (*torch.autograd.Variable*) – Log probability values. When exponentiated, these should sum to 1 along the last axis. Either *ps* or *logits* should be specified but not both.
- **vs** (*list or numpy.ndarray or torch.autograd.Variable*) – Optional list of values in the support.
- **one_hot** – Whether `sample()` returns a *one_hot* sample. Defaults to *False* if *vs* is specified, or *True* if *vs* is not specified.
- **batch_size** (*int*) – Optional number of elements in the batch used to generate a sample. The batch dimension will be the leftmost dimension in the generated sample.

batch_log_pdf(*x*)

Evaluates log probability densities for one or a batch of samples and parameters. The last dimension for *ps* encodes the event probabilities, and the remaining dimensions are considered batch dimensions.

ps and *vs* are first broadcasted to the size of the data *x*. The data tensor is used to to create a mask over *vs* where a 1 in the mask indicates that the corresponding value in *vs* was selected. Since, *ps* and *vs* have the same size, this mask when applied over *ps* gives the probabilities of the selected events. The method returns the logarithm of these probabilities.

Returns tensor with log probabilities for each of the batches.

Return type torch.autograd.Variable

batch_shape(*x=None*)

Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

enumerable = True

enumerate_support()

Returns the categorical distribution's support, as a tensor along the first dimension.

Note that this returns support values of all the batched RVs in lock-step, rather than the full cartesian product. To iterate over the cartesian product, you must construct univariate Categoricals and use itertools.product() over all univariate variables (but this is very expensive).

Parameters

- **ps** (*torch.autograd.Variable*) – Tensor where the last dimension denotes the event probabilities, *p_k*, which must sum to 1. The remaining dimensions are considered batch dimensions.
- **vs** (*list or numpy.ndarray or torch.autograd.Variable*) – Optional parameter, enumerating the items in the support. This could either have a numeric or string type. This should have the same dimension as `ps`.
- **one_hot** (*boolean*) – Denotes whether one hot encoding is enabled. This is True by default. When set to false, and no explicit *vs* is provided, the last dimension gives the one-hot encoded value from the support.

Returns Torch variable or numpy array enumerating the support of the categorical distribution. Each item in the return value, when enumerated along the first dimensions, yields a value from the distribution's support which has the same dimension as would be returned by sample. If `one_hot=True`, the last dimension is used for the one-hot encoding.

Return type torch.autograd.Variable or numpy.ndarray.

event_shape()

Ref: *pyro.distributions.distribution.Distribution.event_shape()*

sample()
Returns a sample which has the same shape as *ps* (or *vs*), except that if `one_hot=True` (and no *vs* is specified), the last dimension will have the same size as the number of events. The type of the sample is *numpy.ndarray* if *vs* is a list or a numpy array, else a tensor is returned.

Returns sample from the Categorical distribution

Return type numpy.ndarray or torch.LongTensor

shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.shape()*

5.1.4 Cauchy

class Cauchy(*mu*, *gamma*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Cauchy (a.k.a. Lorentz) distribution.

This is a continuous distribution which is roughly the ratio of two Gaussians if the second Gaussian is zero mean. The distribution is over tensors that have the same shape as the parameters *mu'and 'gamma*, which in turn must have the same shape as each other.

This is often used in conjunction with *torch.nn.Softplus* to ensure the *gamma* parameter is positive.

Parameters

- **mu** (*torch.autograd.Variable*) – Location parameter.
- **gamma** (*torch.autograd.Variable*) – Scale parameter. Should be positive.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.5 Delta

class Delta(*v*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Degenerate discrete distribution (a single point).

Discrete distribution that assigns probability one to the single element in its support. Delta distribution parameterized by a random choice should not be used with MCMC based inference, as doing so produces incorrect results.

Parameters **v** (*torch.autograd.Variable*) – The single support element.

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

enumerable = True

enumerate_support(*v=None*)
Returns the delta distribution's support, as a tensor along the first dimension.

Parameters **v** – torch variable where each element of the tensor represents the point at which the delta distribution is concentrated.

Returns torch variable enumerating the support of the delta distribution.

Return type torch.autograd.Variable.

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.6 Normal

class Normal(*mu*, *sigma*, *batch_size=None*, *log_pdf_mask=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Univariate normal (Gaussian) distribution.

A distribution over tensors in which each element is independent and Gaussian distributed, with its own mean and standard deviation. The distribution is over tensors that have the same shape as the parameters *mu* and *sigma*, which in turn must have the same shape as each other.

This is often used in conjunction with *torch.nn.Softplus* to ensure the *sigma* parameters are positive.

Parameters

- **mu** (*torch.autograd.Variable*) – Means.
- **sigma** (*torch.autograd.Variable*) – Standard deviations. Should be positive and the same shape as *mu*.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Diagonal Normal log-likelihood

Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

reparameterized = True

sample()
Reparameterized Normal sampler.

Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.7 Exponential

class Exponential(*lam*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Exponential parameterized by scale *lambda*.

This is often used in conjunction with *torch.nn.Softplus* to ensure the *lam* parameter is positive.

Parameters **lam** (*torch.autograd.Variable*) – Scale parameter (a.k.a. *lambda*). Should be positive.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

reparameterized = True

sample()
Reparameterized sampler.

Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.8 Gamma

class Gamma(*alpha*, *beta*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Gamma distribution parameterized by *alpha* and *beta*.

This is often used in conjunction with *torch.nn.Softplus* to ensure *alpha* and *beta* parameters are positive.

Parameters

- **alpha** (*torch.autograd.Variable*) – Shape parameter. Should be positive.
- **beta** (*torch.autograd.Variable*) – Shape parameter. Should be positive. Shouldb be the same shape as *alpha*.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.9 HalfCauchy

class HalfCauchy(*mu*, *gamma*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Half-Cauchy distribution.

This is a continuous distribution with lower-bounded domain ($x > mu$). See also the *Cauchy* distribution.

This is often used in conjunction with *torch.nn.Softplus* to ensure the *gamma* parameter is positive.

Parameters

- **mu** – mean *(tensor)*
- **gamma** – scale *(tensor (0, Infinity))*

analytic_mean()

analytic_var()

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.10 LogNormal

class LogNormal(*mu*, *sigma*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Log-normal distribution.

A distribution over probability vectors obtained by exp-transforming a random variable drawn from `Normal({mu: mu, sigma: sigma})`.

This is often used in conjunction with *torch.nn.Softplus* to ensure the *sigma* parameters are positive.

Parameters

- **mu** (*torch.autograd.Variable*) – log mean parameter.
- **sigma** (*torch.autograd.Variable*) – log standard deviations. Should be positive.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

reparameterized = True

sample()
Reparameterized log-normal sampler. Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.11 Multinomial

class Multinomial(*ps*, *n*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Multinomial distribution.

Distribution over counts for *n* independent *Categorical(ps)* trials.

This is often used in conjunction with *torch.nn.Softmax* to ensure probabilites *ps* are normalized.

Parameters

- **ps** (*torch.autograd.Variable*) – Probabilities (real). Should be positive and should normalized over the rightmost axis.
- **n** (*int*) – Number of trials. Should be positive.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

expanded_sample()

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.12 Poisson

class Poisson(*lam*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Poisson distribution over integers parameterized by scale *lambda*.

This is often used in conjunction with *torch.nn.Softplus* to ensure the *lam* parameter is positive.

Parameters **lam** (*torch.autograd.Variable*) – Mean parameter (a.k.a. *lambda*). Should be positive.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

5.1.13 Uniform

class Uniform(*a*, *b*, *batch_size=None*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Uniform distribution over the continuous interval *[a, b]*.

Parameters

- **a** (*torch.autograd.Variable*) – lower bound (real).
- **b** (*torch.autograd.Variable*) – upper bound (real). Should be greater than *a*.

analytic_mean()
Ref: *pyro.distributions.distribution.Distribution.analytic_mean()*

analytic_var()
Ref: *pyro.distributions.distribution.Distribution.analytic_var()*

batch_log_pdf(*x*)
Ref: *pyro.distributions.distribution.Distribution.batch_log_pdf()*

batch_shape(*x=None*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape()
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

reparameterized = False

sample()
Ref: *pyro.distributions.distribution.Distribution.sample()*

shape (*x=None*)
Ref: *pyro.distributions.distribution.Distribution.shape()*

5.2 Transformed Distribution

5.2.1 TransformedDistribution

class TransformedDistribution (*base_distribution*, *bijectors*, **args*, ***kwargs*)
Bases: *pyro.distributions.distribution.Distribution*

Transforms the base distribution by applying a sequence of *Bijector's to it. This results in a scorable distribution (i.e. it has a 'log_pdf()* method).

Parameters

- **base_distribution** (*pyro.distribution.Distribution*) – a (continuous) base distribution; samples from this distribution are passed through the sequence of *Bijector's to yield a sample from the 'TransformedDistribution*
- **bijectors** – either a single Bijector or a sequence of Bijectors wrapped in a nn.ModuleList

Returns the transformed distribution

batch_log_pdf (*y*, **args*, ***kwargs*)

batch_shape (*x=None*, **args*, ***kwargs*)
Ref: *pyro.distributions.distribution.Distribution.batch_shape()*

event_shape (**args*, ***kwargs*)
Ref: *pyro.distributions.distribution.Distribution.event_shape()*

log_pdf (*y*, **args*, ***kwargs*)

Parameters **y** (*torch.autograd.Variable*) – a value sampled from the transformed distribution

Returns the score (the log pdf) of y

Return type torch.autograd.Variable

Scores the sample by inverting the bijector(s) and computing the score using the score of the base distribution and the log det jacobian

sample (**args*, ***kwargs*)

Returns a sample y

Return type torch.autograd.Variable

Sample from base distribution and pass through bijector(s)

5.2.2 Bijector

class Bijector (**args*, ***kwargs*)
Bases: torch.nn.modules.module.Module

Abstract class *Bijector*. *Bijector* are bijective transformations with computable log det jacobians. They are meant for use in *TransformedDistribution*.

inverse(**args*, ***kwargs*)
Virtual inverse method

Inverts the bijection y => x.

log_det_jacobian(**args*, ***kwargs*)
Virtual logdet jacobian method.

Computes the log det jacobian |*dy/dx*|

5.2.3 InverseAutoRegressiveFlow

class InverseAutoregressiveFlow(*input_dim*, *hidden_dim*, *sigmoid_bias=2.0*, *permutation=None*)
Bases: *pyro.distributions.transformed_distribution.Bijector*

An implementation of an Inverse Autoregressive Flow. Together with the *TransformedDistribution* this provides a way to create richer variational approximations.

Example usage:

```
>>> base_dist = Normal(...)
>>> iaf = InverseAutoregressiveFlow(...)
>>> pyro.module("my_iaf", iaf)
>>> iaf_dist = TransformedDistribution(base_dist, iaf)
```

Note that this implementation is only meant to be used in settings where the inverse of the Bijector is never explicitly computed (rather the result is cached from the forward call). In the context of variational inference, this means that the InverseAutoregressiveFlow should only be used in the guide, i.e. in the variational distribution. In other contexts the inverse could in principle be computed but this would be a (potentially) costly computation that scales with the dimension of the input (and in any case support for this is not included in this implementation).

Parameters

- **input_dim** (*int*) – dimension of input
- **hidden_dim** (*int*) – hidden dimension (number of hidden units)
- **sigmoid_bias** (*float*) – bias on the hidden units fed into the sigmoid; default=‘2.0‘
- **permutation** (*bool*) – whether the order of the inputs should be permuted (by default the conditional dependence structure of the autoregression follows the sequential order)

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

2. Variational Inference with Normalizing Flows [arXiv:1505.05770] Danilo Jimenez Rezende, Shakir Mohamed

3. MADE: Masked Autoencoder for Distribution Estimation [arXiv:1502.03509] Mathieu Germain, Karol Gregor, Iain Murray, Hugo Larochelle

get_arn()

Return type pyro.nn.AutoRegressiveNN

Return the AutoRegressiveNN associated with the InverseAutoregressiveFlow

inverse(*y*, **args*, ***kwargs*)

Parameters **y** (*torch.autograd.Variable*) – the output of the bijection

Inverts y => x. As noted above, this implementation is incapable of inverting arbitrary values *y*; rather it assumes *y* is the result of a previously computed application of the bijector to some *x* (which was cached on the forward call)

log_det_jacobian(*y*, **args*, ***kwargs*)
Calculates the determinant of the log jacobian

CHAPTER 6

Parameters

Parameters in Pyro are basically thin wrappers around PyTorch Variables that carry unique names. As such Parameters are the primary stateful objects in Pyro. Users typically interact with parameters via the Pyro primitive *pyro.param*. Parameters play a central role in stochastic variational inference, where they are used to represent point estimates for the parameters in parameterized families of models and guides.

6.1 ParamStore

class ParamStoreDict

Bases: object

Global store for parameters in Pyro. This is basically a key-value store. The typical user interacts with the ParamStore primarily through the primitive *pyro.param*.

See Intro Part II for further discussion and SVI Part I for some examples.

Some things to bear in mind when using parameters in Pyro:

- parameters must be assigned unique names
- the *init_tensor* argument to *pyro.param* is only used the first time that a given (named) parameter is registered with Pyro.
- for this reason, a user may need to use the *clear()* method if working in a REPL in order to get the desired behavior. this method can also be invoked with *pyro.clear_param_store()*.
- the internal name of a parameter within a PyTorch *nn.Module* that has been registered with Pyro is prepended with the Pyro name of the module. so nothing prevents the user from having two different modules each of which contains a parameter named *weight*. by contrast, a user can only have one top-level parameter named *weight* (outside of any module).
- parameters can be 'tagged' with (string) tags. by default each parameter is tagged with the 'default' tag. this mechanism allows the user to group parameters together and e.g. customize learning rates for different tags. for an example where this is useful see the tutorial SVI Part III.
- parameters can be saved and loaded from disk using *save* and *load*.

clear()
Clear the ParamStore

delete_tag(*tag*)
Removes the tag; any parameters with that tag are unaffected but are no longer associated with that tag.

Parameters **tag** (*str*) – tag to remove

get_active_params(*tags=None*)

Parameters **tag** – optional argument specifying that only active params carrying a particular tag or any of several tags should be returned

Returns all active params in the ParamStore, possibly filtered to a particular tag or tags

Return type set

get_all_param_names()
Get all parameter names in the ParamStore

get_param(*name*, *init_tensor=None*, *tags='default'*)
Get parameter from its name. If it does not yet exist in the ParamStore, it will be created and stored. The Pyro primitive *pyro.param* dispatches to this method.

Parameters

- **name** (*str*) – parameter name
- **init_tensor** (*torch.autograd.Variable*) – initial tensor
- **tags** (*a string or iterable of strings*) – the tag(s) to assign to the parameter

Returns parameter

Return type torch.autograd.Variable

get_param_tags(*param_name*)
Return the tags associated with the parameter

Parameters **param_name** (*str*) – a (single) parameter name

Return type set

get_state()
Get the ParamStore state.

load(*filename*)
Loads parameters from disk

Parameters **filename** – file name to load from

mark_params_active(*params*)

Parameters **params** – iterable of params the user wishes to mark as active in the ParamStore. this information is used to determine which parameters are being optimized, e.g. in the context of pyro.infer.SVI

mark_params_inactive(*params*)

Parameters **params** – iterable of params the user wishes to mark as inactive in the ParamStore. this information is used to determine which parameters are being optimized, e.g. in the context of pyro.infer.SVI

named_parameters()
Returns an iterator over tuples of the form (name, parameter) for each parameter in the ParamStore

param_name(*p*)
Get parameter name from parameter

Parameters **p** – parameter

Returns parameter name

replace_param(*param_name*, *new_param*, *old_param*)
Replace the param param_name with current value old_param with the new value new_param

Parameters

- **param_name** (*str*) – parameter name
- **new_param** (*torch.autograd.Variable*) – the paramater to be put into the ParamStore
- **old_param** – the paramater to be removed from the ParamStore

save(*filename*)
Save parameters to disk

Parameters **filename** – file name to save to

set_state(*state*)
Set the ParamStore state using state from a previous get_state() call

tag_params(*param_names*, *tags*)
Tags the parameter(s) specified by param_names with the tag(s) specified by tags.

Parameters

- **param_name** – either a single parameter name or an iterable of parameter names
- **tags** – either a single string or an iterable of strings

untag_params(*param_names*, *tags*)
Disassociates the parameter(s) specified by param_names with the tag(s) specified by tags.

Parameters

- **param_name** – either a single parameter name or an iterable of parameter names
- **tags** – either a single string or an iterable of strings

CHAPTER 7

Neural Network

The module *pyro.nn* provides implementations of neural network modules that are useful in the context of deep probabilistic programming. None of these modules is really part of the core language.

7.1 AutoRegressiveNN

class AutoRegressiveNN(*input_dim*, *hidden_dim*, *output_dim_multiplier=1*, *mask_encoding=None*, *permutation=None*)

Bases: `torch.nn.modules.module.Module`

A simple implementation of a MADE-like auto-regressive neural network.

Reference: MADE: Masked Autoencoder for Distribution Estimation [arXiv:1502.03509] Mathieu Germain, Karol Gregor, Iain Murray, Hugo Larochelle

Parameters

- **input_dim** (*int*) – the dimensionality of the input
- **hidden_dim** (*int*) – the dimensionality of the hidden units
- **output_dim_multiplier** (*int*) – the dimensionality of the output is given by input_dim x output_dim_multiplier. specifically the shape of the output for a single vector input is [output_dim_multiplier, input_dim]. for any i, j in range(0, output_dim_multiplier) the subset of outputs [i, :] has identical autoregressive structure to [j, :]. defaults to *1*
- **mask_encoding** (*torch.LongTensor*) – a torch Tensor that controls the autoregressive structure (see reference). by default this is chosen at random.
- **permutation** (*torch.LongTensor*) – an optional permutation that is applied to the inputs and controls the order of the autoregressive factorization. in particular for the identity permutation the autoregressive structure is such that the Jacobian is upper triangular. by default this is chosen at random.

forward(*z*)

the forward method

get_mask_encoding()
Get the mask encoding associated with the neural network: basically the quantity m(k) in the MADE paper.

get_permutation()
Get the permutation applied to the inputs (by default this is chosen at random)

class MaskedLinear(*in_features*, *out_features*, *mask*, *bias=True*)
Bases: torch.nn.modules.linear.Linear

A linear mapping with a given mask on the weights (arbitrary bias)

Parameters

- **in_features** (*int*) – the number of input features
- **out_features** (*int*) – the number of output features
- **mask** (*torch.autograd.Variable*) – the mask to apply to the in_features x out_features weight matrix
- **bias** (*bool*) – whether or not *MaskedLinear* should include a bias term. defaults to *True*

forward(*_input*)
the forward method that does the masked linear computation and returns the result

CHAPTER 8

Optimization

The module *pyro.optim* provides support for optimization in Pyro. In particular it provides *PyroOptim*, which is used to wrap PyTorch optimizers and manage optimizers for dynamically generated parameters (see the tutorial SVI Part I for a discussion). Any custom optimization algorithms are also to be found here.

8.1 PyroOptim

class PyroOptim(*optim_constructor*, *optim_args*)
Bases: object

A wrapper for torch.optim.Optimizer objects that helps managing with dynamically generated parameters

Parameters

- **optim_constructor** – a torch.optim.Optimizer
- **optim_args** – a dictionary of learning arguments for the optimizer or a callable that returns such dictionaries

get_state()
Get state associated with all the optimizers in the form of a dictionary with key-value pairs (parameter name, optim state dicts)

load(*filename*)

Parameters **filename** – file name to load from

Load optimizer state from disk

save(*filename*)

Parameters **filename** – file name to save to

Save optimizer state to disk

set_state(*state_dict*)
Set the state associated with all the optimizers using the state obtained from a previous call to get_state()

8.2 ClippedAdam

class ClippedAdam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *clip_norm=10.0*, *lrd=1.0*)

Bases: `torch.optim.optimizer.Optimizer`

Parameters

- **params** – iterable of parameters to optimize or dicts defining parameter groups
- **lr** – learning rate (default: 1e-3)
- **betas** (*Tuple*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** – weight decay (L2 penalty) (default: 0)
- **clip_norm** – magnitude of norm to which gradients are clipped (default: 10.0)
- **lrd** – rate at which learning rate decays (default: 1.0)

Small modification to the Adam algorithm implemented in torch.optim.Adam to include gradient clipping and learning rate decay.

Reference

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

step(*closure=None*)

:param closure:: An optional closure that reevaluates the model and returns the loss.

Performs a single optimization step.

CHAPTER 9

Advanced Features

Beneath the built-in inference algorithms, Pyro has a library of flexible primitives for creating new inference algorithms and working with probabilistic programs. The core abstraction of composable inference in Pyro is the *poutine* (short for Pyro Coroutine). Pyro's inference algorithms are all built by applying *poutine* s to stochastic functions.

9.1 Poutines (Pyro Coroutines)

9.1.1 Poutine

class Poutine(*fn*)

Bases: object

Context manager class that modifies behavior and adds side effects to stochastic functions i.e. callables containing pyro primitive statements.

See the Poutine execution model writeup in the documentation for a description of the entire Poutine system.

This is the base Poutine class. It implements the default behavior for all pyro primitives, so that the joint distribution induced by a stochastic function fn is identical to the joint distribution induced by Poutine(fn).

9.1.2 BlockPoutine

class BlockPoutine(*fn*, *hide_all=True*, *expose_all=False*, *hide=None*, *expose=None*, *hide_types=None*, *expose_types=None*)

Bases: pyro.poutine.poutine.Poutine

This Poutine selectively hides pyro primitive sites from the outside world.

For example, suppose the stochastic function fn has two sample sites "a" and "b". Then any poutine outside of BlockPoutine(fn, hide=["a"]) will not be applied to site "a" and will only see site "b":

```
>>> fn_inner = TracePoutine(fn)
>>> fn_outer = TracePoutine(BlockPoutine(TracePoutine(fn), hide=["a"]))
>>> trace_inner = fn_inner.get_trace()
>>> trace_outer  = fn_outer.get_trace()
>>> "a" in trace_inner
True
>>> "a" in trace_outer
False
>>> "b" in trace_inner
True
>>> "b" in trace_outer
True
```

BlockPoutine has a flexible interface that allows users to specify in several different ways which sites should be hidden or exposed. See the constructor for details.

9.1.3 ConditionPoutine

class ConditionPoutine(*fn*, *data*)
Bases: *pyro.poutine.poutine.Poutine*

Adds values at observe sites to condition on data and override sampling

9.1.4 EscapePoutine

class EscapePoutine(*fn*, *escape_fn*)
Bases: *pyro.poutine.poutine.Poutine*

Poutine that does a nonlocal exit by raising a util.NonlocalExit exception

9.1.5 IndepPoutine

class CondIndepStackFrame(*name*, *counter*, *vectorized*)
Bases: tuple

counter
Alias for field number 1

name
Alias for field number 0

vectorized
Alias for field number 2

class IndepPoutine(*fn*, *name*, *vectorized*)
Bases: *pyro.poutine.poutine.Poutine*

This poutine keeps track of stack of independence information declared by nested `irange` and `iarange` contexts. This information is stored in a `cond_indep_stack` at each sample/observe site for consumption by `TracePoutine`.

9.1.6 LiftPoutine

class LiftPoutine(*fn*, *prior*)
Bases: *pyro.poutine.poutine.Poutine*

Poutine which "lifts" parameters to random samples. Given a stochastic function with param calls and a prior, creates a stochastic function where all param calls are replaced by sampling from prior.

Prior should be a callable or a dict of names to callables.

9.1.7 ReplayPoutine

class ReplayPoutine(*fn*, *guide_trace*, *sites=None*)
Bases: *pyro.poutine.poutine.Poutine*

Poutine for replaying from an existing execution trace.

9.1.8 ScalePoutine

class ScalePoutine(*fn*, *scale*)
Bases: *pyro.poutine.poutine.Poutine*

This poutine 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

- **fn** (*callable or None*) – an optional function to be scaled
- **scale** (*float or torch.autograd.Variable*) – a positive scaling factor

9.1.9 Trace

class Trace(**args*, ***kwargs*)
Bases: networkx.classes.digraph.DiGraph

Execution trace data structure

add_node(*site_name*, **args*, ***kwargs*)

Parameters **site_name** (*string*) – the name of the site to be added

Adds a site to the trace.

Identical to super(Trace, self).add_node, but raises an error when attempting to add a duplicate node instead of silently overwriting.

batch_log_pdf(*site_filter=<function <lambda>>*)
Compute the batched local and overall log-probabilities of the trace.

The local computation is memoized, and also stores the local *.log_pdf()*.

compute_batch_log_pdf(*site_filter=<function <lambda>>*)
Compute the batched local log-probabilities at each site of the trace.

The local computation is memoized, and also stores the local *.log_pdf()*.

copy()
Makes a shallow copy of self with nodes and edges preserved. Identical to super(Trace, self).copy(), but preserves the type and the self.graph_type attribute

log_pdf (*site_filter=<function <lambda>>*)
Compute the local and overall log-probabilities of the trace.

The local computation is memoized.

Returns total log probability.

Return type torch.autograd.Variable

node_dict_factory
alias of `OrderedDict`

nonreparam_stochastic_nodes
Gets a list of names of sample sites whose stochastic functions are not reparameterizable primitive distributions

observation_nodes
Gets a list of names of observe sites

reparameterized_nodes
Gets a list of names of sample sites whose stochastic functions are reparameterizable primitive distributions

stochastic_nodes
Gets a list of names of sample sites

9.1.10 TracePoutine

class TracePoutine (*fn*, *graph_type=None*)
Bases: *pyro.poutine.poutine.Poutine*

Execution trace poutine.

A TracePoutine records the input and output to every pyro primitive and stores them as a site in a Trace(). This should, in theory, be sufficient information for every inference algorithm (along with the implicit computational graph in the Variables?)

We can also use this for visualization.

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

Returns data structure

Return type pyro.poutine.Trace

Helper method for a very common use case. Calls this poutine and returns its trace instead of the function's return value.

get_vectorized_map_data_info (*trace*)
This determines whether the vectorized map_datas are rao-blackwellizable by *TraceGraph_ELBO*. This also gathers information to be consumed by downstream by *TraceGraph_ELBO*.

identify_dense_edges (*trace*)
Modifies a trace in-place by adding all edges based on the *cond_indep_stack* information stored at each site.

CHAPTER 10

Indices and tables

- genindex
- search

Python Module Index

p

Index

R

S

T

U

V