Pyro Documentation

Uber AI Labs

Jul 18, 2018

Pyro Core:

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
```

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

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
- **infer** (*dict*) – Optional dictionary of inference parameters specified in kwargs. See inference documentation for details.

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

module(*name*, *nn_module*, *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
- **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

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

class 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_prob` computations. Defaults to `torch.Tensor.is_cuda`.

Returns A reusable iterator yielding a sequence of integers.

Examples:

```
>>> for i in irange('data', 100, subsample_size=10):
...     if z[i]:  # Prevents vectorization.
...         obs = sample('obs_{}'.format(i), dist.Normal(loc, scale),␣
↪obs=data[i])
```

See SVI Part II for an extended discussion.

class iarange(*name*, *size=None*, *subsample_size=None*, *subsample=None*, *dim=None*, *use_cuda=None*)

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)*.
- **dim** (*int*) – An optional dimension to use for this independence index. If specified, `dim` should be negative, i.e. should index from the right. If not specified, `dim` is set to the rightmost dim that is left of all enclosing `iarange` contexts.
- **use_cuda** (*bool*) – Optional bool specifying whether to use cuda tensors for *subsample* and *log_prob*. Defaults to *torch.Tensor.is_cuda*.

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

Examples:

```
>>> # This version simply declares independence:
>>> with iarange('data'):
...     obs = sample('obs', dist.Normal(loc, scale), obs=data)
```

```
>>> # This version subsamples data in vectorized way:
>>> with iarange('data', 100, subsample_size=10) as ind:
...     obs = sample('obs', dist.Normal(loc, scale), obs=data[ind])
```

```
>>> # This wraps a user-defined subsampling method for use in pyro:
>>> ind = torch.randint(0, 100, (10,)).long() # custom subsample
>>> with iarange('data', 100, subsample=ind):
...     obs = sample('obs', dist.Normal(loc, scale), obs=data[ind])
```

```
>>> # This reuses two different independence contexts.
>>> x_axis = iarange('outer', 320, dim=-1)
>>> y_axis = iarange('inner', 200, dim=-2)
>>> with x_axis:
...     x_noise = sample("x_noise", dist.Normal(loc, scale).expand_
↪by([320]))
>>> with y_axis:
...     y_noise = sample("y_noise", dist.Normal(loc, scale).expand_
↪by([200, 1]))
```

(continues on next page)

(continued from previous page)

```
>>> with x_axis, y_axis:
...     xy_noise = sample("xy_noise", dist.Normal(loc, scale).expand_
↪by([200, 320]))
```

See SVI Part II for an extended discussion.

get_param_store()
Returns the ParamStore

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

validation_enabled(**args*, ***kwds*)
Context manager that is useful when temporarily enabling/disabling validation checks.

Parameters **is_validate** (*bool*) – (optional; defaults to True) temporary validation check override.

enable_validation(*is_validate=True*)
Enable or disable validation checks in Pyro. Validation checks provide useful warnings and errors, e.g. NaN checks, validating distribution arguments and support values, etc. which is useful for debugging. Since some of these checks may be expensive, we recommend turning this off for mature models.

Parameters **is_validate** (*bool*) – (optional; defaults to True) whether to enable validation checks.

compile(*fn=None*, ***jit_options*)
Drop-in replacement for `torch.jit.compile()` that works with Pyro functions that call *pyro.param()*.

The actual compilation artifact is stored in the `compiled` attribute of the output. Call diagnostic methods on this attribute.

Example:

```
def model(x):
    scale = pyro.param("scale", torch.tensor(0.5), constraint=constraints.
↪positive)
    return pyro.sample("y", dist.Normal(x, scale))

@pyro.ops.jit.compile(nderivs=1)
def model_log_prob_fn(x, y):
    cond_model = pyro.condition(model, data={"y": y})
    tr = pyro.poutine.trace(cond_model).get_trace(x)
    return tr.log_prob_sum()
```

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*, *num_samples=10*, *num_steps=0*, ***kwargs*)

Bases: *pyro.infer.abstract_infer.TracePosterior*

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** (pyro.infer.elbo.ELBO) – an instance of a subclass of *ELBO*. Pyro provides three built-in losses: *Trace_ELBO*, *TraceGraph_ELBO*, and *TraceEnum_ELBO*. See the *ELBO* docs to learn how to implement a custom loss.
- **num_samples** – the number of samples for Monte Carlo posterior approximation
- **num_steps** – the number of optimization steps to take in `run()`

A unified interface for stochastic variational inference in Pyro. The most commonly used loss is `loss=Trace_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.

run(**args*, ***kwargs*)

Calls *self._traces* to populate execution traces from a stochastic Pyro model.

Parameters

- **args** – optional args taken by *self._traces*.
- **kwargs** – optional keywords args taken by *self._traces*.

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*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: object

ELBO is the top-level interface for stochastic variational inference via optimization of the evidence lower bound.

Most users will not interact with this base class *ELBO* directly; instead they will create instances of derived classes: *Trace_ELBO*, *TraceGraph_ELBO*, or *TraceEnum_ELBO*.

Parameters

- **num_particles** – The number of particles/samples used to form the ELBO (gradient) estimators.
- **max_iarange_nesting** (*int*) – Optional bound on max number of nested *pyro.iarange()* contexts. This is only required to enumerate over sample sites in parallel, e.g. if a site sets `infer={"enumerate": "parallel"}`.
- **vectorize_particles** (*bool*) – Whether to vectorize the ELBO computation over *num_particles*. Defaults to False. This requires static structure in model and guide. In addition, this wraps the model and guide inside a *broadcast* poutine for automatic broadcasting of sample site batch shapes, and requires specifying a finite value for *max_iarange_nesting*.
- **strict_enumeration_warning** (*bool*) – Whether to warn about possible misuse of enumeration, i.e. that *pyro.infer.traceenum_elbo.TraceEnum_ELBO* is used iff there are enumerated sample sites.

References

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

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

class Trace_ELBO(*num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.elbo.ELBO*

A trace implementation of ELBO-based SVI. The estimator is constructed along the lines of references [1] and [2]. There are no restrictions on the dependency structure of the model or the guide. The gradient estimator includes partial Rao-Blackwellization for reducing the variance of the estimator when non-reparameterizable random variables are present. The Rao-Blackwellization is partial in that it only uses conditional independence information that is marked by *iarange* contexts. For more fine-grained Rao-Blackwellization, see *TraceGraph_ELBO*.

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*)

Returns returns an estimate of the ELBO

Return type float

Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

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

Returns returns an estimate of the ELBO

Return type float

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.

class JitTrace_ELBO(*num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.trace_elbo.Trace_ELBO*

Like *Trace_ELBO* but uses *pyro.ops.jit.compile()* to compile *loss_and_grads()*.

This works only for a limited set of models:

- Models must have static structure.
- Models must not depend on any global data (except the param store).
- All model inputs that are tensors must be passed in via `*args`.
- All model inputs that are *not* tensors must be passed in via `*kwargs`, and these will be fixed to their values on the first call to `jit_loss_and_grads()`.

Warning: Experimental. Interface subject to change.

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

Returns returns an estimate of the ELBO

Return type float

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.

class TraceGraph_ELBO(*num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.elbo.ELBO*

A TraceGraph implementation of ELBO-based SVI. The gradient estimator is constructed along the lines of reference [1] specialized to the case of the ELBO. It supports arbitrary dependency structure for the model and

guide as well as baselines for non-reparameterizable random variables. Where possible, conditional dependency information as recorded in the `Trace` is used to reduce the variance of the gradient estimator. In particular three kinds of conditional dependency information are used to reduce variance: - the sequential order of samples (z is sampled after y => y does not depend on z) - *iarange* generators - *irange* generators

References

[1] ***Gradient Estimation Using Stochastic Computation Graphs***, John Schulman, Nicolas Heess, Theophane Weber, Pieter Abbeel

[2] ***Neural Variational Inference and Learning in Belief Networks*** Andriy Mnih, Karol Gregor

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

Returns returns an estimate of the ELBO

Return type float

Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

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

Returns returns an estimate of the ELBO

Return type float

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. If baselines are present, a baseline loss is also constructed and differentiated.

class JitTraceGraph_ELBO(*num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.tracegraph_elbo.TraceGraph_ELBO*

Like *TraceGraph_ELBO* but uses `torch.jit.compile()` to compile *loss_and_grads()*.

This works only for a limited set of models:

- Models must have static structure.
- Models must not depend on any global data (except the param store).
- All model inputs that are tensors must be passed in via `*args`.
- All model inputs that are *not* tensors must be passed in via `*kwargs`, and these will be fixed to their values on the first call to *loss_and_grads()*.

Warning: Experimental. Interface subject to change.

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

Returns returns an estimate of the ELBO

Return type float

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. If baselines are present, a baseline loss is also constructed and differentiated.

class TraceEnum_ELBO(*num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.elbo.ELBO*

A trace implementation of ELBO-based SVI that supports enumeration over discrete sample sites.

To enumerate over a sample site, the `guide`'s sample site must specify either `infer={'enumerate': 'sequential'}` or `infer={'enumerate': 'parallel'}`. To configure all sites at once, use `config_enumerate`().

This assumes restricted dependency structure on the model and guide: variables outside of an *iarange* can never depend on variables inside that *iarange*.

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

Returns an estimate of the ELBO

Return type float

Estimates the ELBO using `num_particles` many samples (particles).

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

Returns a differentiable estimate of the ELBO

Return type torch.Tensor

Raises **ValueError** – if the ELBO is not differentiable (e.g. is identically zero)

Estimates a differentiable ELBO using `num_particles` many samples (particles). The result should be infinitely differentiable (as long as underlying derivatives have been implemented).

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

Returns an estimate of the ELBO

Return type float

Estimates the ELBO using `num_particles` many samples (particles). Performs backward on the ELBO of each particle.

class JitTraceEnum_ELBO(*num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.traceenum_elbo.TraceEnum_ELBO*

Like *TraceEnum_ELBO* but uses *pyro.ops.jit.compile()* to compile *loss_and_grads()*.

This works only for a limited set of models:

- Models must have static structure.
- Models must not depend on any global data (except the param store).
- All model inputs that are tensors must be passed in via `*args`.
- All model inputs that are *not* tensors must be passed in via `*kwargs`, and these will be fixed to their values on the first call to `jit_loss_and_grads`().

Warning: Experimental. Interface subject to change.

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

Returns an estimate of the ELBO

Return type float

Estimates the ELBO using `num_particles` many samples (particles). Performs backward on the ELBO of each particle.

class RenyiELBO(*alpha=0*, *num_particles=1*, *max_iarange_nesting=inf*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.elbo.ELBO*

An implementation of Renyi's α-divergence variational inference follows reference [1].

To have a lower bound, we require $\alpha \geq 0$. However, according to reference [1], depending on the dataset, $\alpha < 0$ might give better results. In the special case $\alpha = 0$, we have important weighted lower bound derived in reference [2].

Note: Setting $\alpha < 1$ gives a better bound than the usual ELBO. For $\alpha = 1$, it is better to use *Trace_ELBO* class because it helps reduce variances of gradient estimations.

Warning: Mini-batch training is not supported yet.

Parameters

- **alpha** (*float*) – The order of α-divergence. Here $\alpha \neq 1$. Default is 0.
- **num_particles** – The number of particles/samples used to form the ELBO (gradient) estimators. Default is 2.
- **max_iarange_nesting** (*int*) – Bound on max number of nested *pyro.iarange()* contexts. Default is 2.
- **strict_enumeration_warning** (*bool*) – Whether to warn about possible misuse of enumeration, i.e. that *TraceEnum_ELBO* is used iff there are enumerated sample sites.

References:

[1] ***Renyi Divergence Variational Inference***, Yingzhen Li, Richard E. Turner

[2] ***Importance Weighted Autoencoders***, Yuri Burda, Roger Grosse, Ruslan Salakhutdinov

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

Returns returns an estimate of the ELBO

Return type float

Evaluates the ELBO with an estimator that uses num_particles many samples/particles.

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

Returns returns an estimate of the ELBO

Return type float

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.

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 Inference Utilities

class EmpiricalMarginal(*trace_posterior*, *sites=None*, *validate_args=None*)
Bases: pyro.distributions.empirical.Empirical

Marginal distribution, that wraps over a TracePosterior object to provide a a marginal over one or more latent sites or the return values of the TracePosterior's model. If multiple sites are specified, they must have the same tensor shape.

Parameters

- **trace_posterior** (TracePosterior) – a TracePosterior instance representing a Monte Carlo posterior.
- **sites** (*list*) – optional list of sites for which we need to generate the marginal distribution. Note that for multiple sites, the shape for the site values must match (needed by the underlying Empirical class).

class TracePosterior
Bases: object

Abstract TracePosterior object from which posterior inference algorithms inherit. When run, collects a bag of execution traces from the approximate posterior. This is designed to be used by other utility classes like *EmpiricalMarginal*, that need access to the collected execution traces.

run(**args*, ***kwargs*)
Calls *self._traces* to populate execution traces from a stochastic Pyro model.

Parameters

- **args** – optional args taken by *self._traces*.
- **kwargs** – optional keywords args taken by *self._traces*.

class TracePredictive(*model*, *posterior*, *num_samples*)
Bases: *pyro.infer.abstract_infer.TracePosterior*

Generates and holds traces from the posterior predictive distribution, given model execution traces from the approximate posterior. This is achieved by constraining latent sites to randomly sampled parameter values from the model execution traces and running the model forward to generate traces with new response ("_RETURN") sites.

Parameters

- **model** – arbitrary Python callable containing Pyro primitives.
- **posterior** (TracePosterior) – trace posterior instance holding samples from the model's approximate posterior.
- **num_samples** (*int*) – number of samples to generate.

4.5 MCMC

4.5.1 MCMC

class MCMC(*kernel*, *num_samples*, *warmup_steps=0*)

Bases: *pyro.infer.abstract_infer.TracePosterior*

Wrapper class for Markov Chain Monte Carlo algorithms. Specific MCMC algorithms are TraceKernel instances and need to be supplied as a `kernel` argument to the constructor.

Parameters

- **kernel** – An instance of the `TraceKernel` class, which when given an execution trace returns another sample trace from the target (posterior) distribution.
- **num_samples** (*int*) – The number of samples that need to be generated, excluding the samples discarded during the warmup phase.
- **warmup_steps** (*int*) – Number of warmup iterations. The samples generated during the warmup phase are discarded.

4.5.2 HMC

class HMC(*model*, *step_size=None*, *trajectory_length=None*, *num_steps=None*, *adapt_step_size=False*, *transforms=None*)

Bases: `pyro.infer.mcmc.trace_kernel.TraceKernel`

Simple Hamiltonian Monte Carlo kernel, where `step_size` and `num_steps` need to be explicitly specified by the user.

References

[1] *MCMC Using Hamiltonian Dynamics*, Radford M. Neal

Parameters

- **model** – Python callable containing Pyro primitives.
- **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.
- **trajectory_length** (*float*) – Length of a MCMC trajectory. If not specified, it will be set to `step_size x num_steps`. In case `num_steps` is not specified, it will be set to 2π.
- **num_steps** (*int*) – The number of discrete steps over which to simulate Hamiltonian dynamics. The state at the end of the trajectory is returned as the proposal. This value is always equal to `int(trajectory_length / step_size)`.
- **adapt_step_size** (*bool*) – A flag to decide if we want to adapt step_size during warm-up phase using Dual Averaging scheme.
- **transforms** (*dict*) – Optional dictionary that specifies a transform for a sample site with constrained support to unconstrained space. The transform should be invertible, and implement *log_abs_det_jacobian*. If not specified and the model has sites with constrained support, automatic transformations will be applied, as specified in `torch.distributions.constraint_registry`.

Example:

```
>>> true_coefs = torch.tensor([1., 2., 3.])
>>> data = torch.randn(2000, 3)
>>> dim = 3
>>> labels = dist.Bernoulli(logits=(true_coefs * data).sum(-1)).sample()
>>>
>>> def model(data):
...     coefs_mean = torch.zeros(dim)
...     coefs = pyro.sample('beta', dist.Normal(coefs_mean, torch.ones(3)))
...     y = pyro.sample('y', dist.Bernoulli(logits=(coefs * data).sum(-1)),␣
↪obs=labels)
...     return y
>>>
>>> hmc_kernel = HMC(model, step_size=0.0855, num_steps=4)
>>> mcmc_run = MCMC(hmc_kernel, num_samples=500, warmup_steps=100).run(data)
>>> posterior = EmpiricalMarginal(mcmc_run, 'beta')
>>> posterior.mean
tensor([ 0.9819,  1.9258,  2.9737])
```

cleanup()
Optional method to clean up any residual state on termination.

diagnostics()
Relevant diagnostics (optional) to be printed at regular intervals of the MCMC run. Returns *None* by default.

Returns String containing the diagnostic summary. e.g. acceptance rate

Return type string

end_warmup()
Optional method to tell kernel that warm-up phase has been finished.

initial_trace()
Returns an initial trace from the prior to initiate the MCMC run.

Returns Trace instance.

sample(*trace*)
Samples a trace from the approximate posterior distribution, when given an existing trace.

Parameters

- **trace** – Current execution trace.
- **time_step** (*int*) – Current time step.

Returns New trace sampled from the approximate posterior distribution.

setup(**args*, ***kwargs*)
Optional method to set up any state required at the start of the simulation run.

Parameters

- ***args** – Algorithm specific positional arguments.
- ****kwargs** – Algorithm specific keyword arguments.

4.5.3 NUTS

class NUTS(*model*, *step_size=None*, *adapt_step_size=False*, *transforms=None*)
Bases: `pyro.infer.mcmc.hmc.HMC`

No-U-Turn Sampler kernel, which provides an efficient and convenient way to run Hamiltonian Monte Carlo. The number of steps taken by the integrator is dynamically adjusted on each call to `sample` to ensure an optimal length for the Hamiltonian trajectory [1]. As such, the samples generated will typically have lower autocorrelation than those generated by the *HMC* kernel. Optionally, the NUTS kernel also provides the ability to adapt step size during the warmup phase.

Refer to the baseball example to see how to do Bayesian inference in Pyro using NUTS.

References

[1] *The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo*, Matthew D. Hoffman, and Andrew Gelman. [2] *A Conceptual Introduction to Hamiltonian Monte Carlo*, Michael Betancourt [3] *Slice Sampling*, Radford M. Neal

Parameters

- **model** – Python callable containing Pyro primitives.
- **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.
- **transforms** (*dict*) – Optional dictionary that specifies a transform for a sample site with constrained support to unconstrained space. The transform should be invertible, and implement *log_abs_det_jacobian*. If not specified and the model has sites with constrained support, automatic transformations will be applied, as specified in `torch.distributions.constraint_registry`.

Example:

```
>>> true_coefs = torch.tensor([1., 2., 3.])
>>> data = torch.randn(2000, 3)
>>> dim = 3
>>> labels = dist.Bernoulli(logits=(true_coefs * data).sum(-1)).sample()
>>>
>>> def model(data):
...     coefs_mean = torch.zeros(dim)
...     coefs = pyro.sample('beta', dist.Normal(coefs_mean, torch.ones(3)))
...     y = pyro.sample('y', dist.Bernoulli(logits=(coefs * data).sum(-1)),␣
↪obs=labels)
...     return y
>>>
>>> nuts_kernel = NUTS(model, adapt_step_size=True)
>>> mcmc_run = MCMC(nuts_kernel, num_samples=500, warmup_steps=300).run(data)
>>> posterior = EmpiricalMarginal(mcmc_run, 'beta')
>>> posterior.mean
tensor([ 0.9221,  1.9464,  2.9228])
```

sample(*trace*)

Samples a trace from the approximate posterior distribution, when given an existing trace.

Parameters

- **trace** – Current execution trace.
- **time_step** (*int*) – Current time step.

Returns New trace sampled from the approximate posterior distribution.

CHAPTER 5

Distributions

5.1 PyTorch Distributions

Most distributions in Pyro are thin wrappers around PyTorch distributions. For details on the PyTorch distribution interface, see `torch.distributions.distribution.Distribution`. For differences between the Pyro and PyTorch interfaces, see *TorchDistributionMixin*.

5.1.1 Bernoulli

class Bernoulli(*probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.bernoulli.Bernoulli` with *TorchDistributionMixin*.

5.1.2 Beta

class Beta(*concentration1*, *concentration0*, *validate_args=None*)
Wraps `torch.distributions.beta.Beta` with *TorchDistributionMixin*.

5.1.3 Categorical

class Categorical(*probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.categorical.Categorical` with *TorchDistributionMixin*.

5.1.4 Cauchy

class Cauchy(*loc*, *scale*, *validate_args=None*)
Wraps `torch.distributions.cauchy.Cauchy` with *TorchDistributionMixin*.

5.1.5 Chi2

class Chi2(*df*, *validate_args=None*)
Wraps `torch.distributions.chi2.Chi2` with *TorchDistributionMixin*.

5.1.6 Dirichlet

class Dirichlet(*concentration*, *validate_args=None*)
Wraps `torch.distributions.dirichlet.Dirichlet` with *TorchDistributionMixin*.

5.1.7 Exponential

class Exponential(*rate*, *validate_args=None*)
Wraps `torch.distributions.exponential.Exponential` with *TorchDistributionMixin*.

5.1.8 ExponentialFamily

class ExponentialFamily(*batch_shape=torch.Size([])*, *event_shape=torch.Size([])*, *validate_args=None*)
Wraps `torch.distributions.exp_family.ExponentialFamily` with *TorchDistributionMixin*.

5.1.9 FisherSnedecor

class FisherSnedecor(*df1*, *df2*, *validate_args=None*)
Wraps `torch.distributions.fishersnedecor.FisherSnedecor` with *TorchDistributionMixin*.

5.1.10 Gamma

class Gamma(*concentration*, *rate*, *validate_args=None*)
Wraps `torch.distributions.gamma.Gamma` with *TorchDistributionMixin*.

5.1.11 Geometric

class Geometric(*probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.geometric.Geometric` with *TorchDistributionMixin*.

5.1.12 Gumbel

class Gumbel(*loc*, *scale*, *validate_args=None*)
Wraps `torch.distributions.gumbel.Gumbel` with *TorchDistributionMixin*.

5.1.13 Independent

class Independent(*base_distribution*, *reinterpreted_batch_ndims*, *validate_args=None*)
Wraps `torch.distributions.independent.Independent` with *TorchDistributionMixin*.

5.1.14 Laplace

class Laplace(*loc*, *scale*, *validate_args=None*)
Wraps `torch.distributions.laplace.Laplace` with *TorchDistributionMixin*.

5.1.15 LogNormal

class LogNormal(*loc*, *scale*, *validate_args=None*)
Wraps `torch.distributions.log_normal.LogNormal` with *TorchDistributionMixin*.

5.1.16 LogisticNormal

class LogisticNormal(*loc*, *scale*, *validate_args=None*)
Wraps `torch.distributions.logistic_normal.LogisticNormal` with *TorchDistributionMixin*.

5.1.17 Multinomial

class Multinomial(*total_count=1*, *probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.multinomial.Multinomial` with *TorchDistributionMixin*.

5.1.18 MultivariateNormal

class MultivariateNormal(*loc*, *covariance_matrix=None*, *precision_matrix=None*, *scale_tril=None*, *validate_args=None*)
Wraps `torch.distributions.multivariate_normal.MultivariateNormal` with *TorchDistributionMixin*.

5.1.19 Normal

class Normal(*loc*, *scale*, *validate_args=None*)
Wraps `torch.distributions.normal.Normal` with *TorchDistributionMixin*.

5.1.20 OneHotCategorical

class OneHotCategorical(*probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.one_hot_categorical.OneHotCategorical` with *TorchDistributionMixin*.

5.1.21 Pareto

class Pareto(*scale*, *alpha*, *validate_args=None*)
Wraps `torch.distributions.pareto.Pareto` with *TorchDistributionMixin*.

5.1.22 Poisson

class Poisson(*rate*, *validate_args=None*)
Wraps `torch.distributions.poisson.Poisson` with *TorchDistributionMixin*.

5.1.23 RelaxedBernoulli

class RelaxedBernoulli(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.relaxed_bernoulli.RelaxedBernoulli` with *TorchDistributionMixin*.

5.1.24 RelaxedOneHotCategorical

class RelaxedOneHotCategorical(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)
Wraps `torch.distributions.relaxed_categorical.RelaxedOneHotCategorical` with *TorchDistributionMixin*.

5.1.25 StudentT

class StudentT(*df*, *loc=0.0*, *scale=1.0*, *validate_args=None*)
Wraps `torch.distributions.studentT.StudentT` with *TorchDistributionMixin*.

5.1.26 TransformedDistribution

class TransformedDistribution(*base_distribution*, *transforms*, *validate_args=None*)
Wraps `torch.distributions.transformed_distribution.TransformedDistribution` with *TorchDistributionMixin*.

5.1.27 Uniform

class Uniform(*low*, *high*, *validate_args=None*)
Wraps `torch.distributions.uniform.Uniform` with *TorchDistributionMixin*.

5.2 Pyro Distributions

5.2.1 Abstract Distribution

class Distribution
Bases: `object`

Base class for parameterized probability distributions.

Distributions in Pyro are stochastic function objects with *sample()* and *log_prob()* methods. Distribution are stochastic functions with fixed parameters:

```
d = dist.Bernoulli(param)
x = d()                                 # Draws a random sample.
p = d.log_prob(x)                       # Evaluates log probability of x.
```

Implementing New Distributions:

Derived classes must implement the methods: *sample()*, *log_prob()*.

Examples:

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

__call__(**args*, ***kwargs*)
Samples a random value (just an alias for `.sample(*args, **kwargs)`).

For tensor distributions, the returned tensor should have the same `.shape` as the parameters.

Returns A random value.

Return type torch.Tensor

enumerate_support()
Returns a representation of the parametrized distribution's support, along the first dimension. This is implemented only by discrete distributions.

Note that this returns support values of all the batched RVs in lock-step, rather than the full cartesian product.

Returns An iterator over the distribution's discrete support.

Return type iterator

has_enumerate_support = False

has_rsample = False

log_prob(*x*, **args*, ***kwargs*)
Evaluates log probability densities for each of a batch of samples.

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

Returns log probability densities as a one-dimensional `Tensor` with same batch size as value and params. The shape of the result should be `self.batch_size`.

Return type torch.Tensor

sample(**args*, ***kwargs*)
Samples a random value.

For tensor distributions, the returned tensor should have the same `.shape` as the parameters, unless otherwise noted.

Parameters **sample_shape** (*torch.Size*) – the size of the iid batch to be drawn from the distribution.

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

Return type torch.Tensor

score_parts(*x*, **args*, ***kwargs*)
Computes ingredients for stochastic gradient estimators of ELBO.

The default implementation is correct both for non-reparameterized and for fully reparameterized distributions. Partially reparameterized distributions should override this method to compute correct *.score_function* and *.entropy_term* parts.

Parameters **x** (*torch.Tensor*) – A single value or batch of values.

Returns A *ScoreParts* object containing parts of the ELBO estimator.

Return type ScoreParts

5.2.2 TorchDistributionMixin

class TorchDistributionMixin

Bases: pyro.distributions.distribution.Distribution

Mixin to provide Pyro compatibility for PyTorch distributions.

You should instead use *TorchDistribution* for new distribution classes.

This is mainly useful for wrapping existing PyTorch distributions for use in Pyro. Derived classes must first inherit from torch.distributions.distribution.Distribution and then inherit from *TorchDistributionMixin*.

__call__(*sample_shape=torch.Size([])*)

Samples a random value.

This is reparameterized whenever possible, calling rsample() for reparameterized distributions and sample() for non-reparameterized distributions.

Parameters **sample_shape** (*torch.Size*) – the size of the iid batch to be drawn from the distribution.

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

Return type torch.Tensor

event_dim

Returns Number of dimensions of individual events.

Return type int

shape(*sample_shape=torch.Size([])*)

The tensor shape of samples from this distribution.

Samples are of shape:

```
d.shape(sample_shape) == sample_shape + d.batch_shape + d.event_shape
```

Parameters **sample_shape** (*torch.Size*) – the size of the iid batch to be drawn from the distribution.

Returns Tensor shape of samples.

Return type torch.Size

expand(*batch_shape*)

Expands a distribution to a desired batch_shape.

Note that this is more general than *expand_by()* because d.expand_by(sample_shape) can be reduced to d.expand(sample_shape + d.batch_shape).

Parameters **batch_shape** (*torch.Size*) – The target batch_shape. This must compatible with self.batch_shape similar to the requirements of torch.Tensor.expand(): the target batch_shape must be at least as long as self.batch_shape, and for each non-singleton dim of self.batch_shape, batch_shape must either agree or be set to -1.

Returns An expanded version of this distribution.

Return type ReshapedDistribution

expand_by(*sample_shape*)
Expands a distribution by adding sample_shape to the left side of its batch_shape.

To expand internal dims of self.batch_shape from 1 to something larger, use *expand()* instead.

Parameters **sample_shape** (*torch.Size*) – The size of the iid batch to be drawn from the distribution.

Returns An expanded version of this distribution.

Return type ReshapedDistribution

reshape(*sample_shape=None*, *extra_event_dims=None*)

independent(*reinterpreted_batch_ndims=None*)
Reinterprets the n rightmost dimensions of this distributions batch_shape as event dims, adding them to the left side of event_shape.

Example:

```
>>> [d1.batch_shape, d1.event_shape]
[torch.Size([2, 3]), torch.Size([4, 5])]
>>> d2 = d1.independent(1)
>>> [d2.batch_shape, d2.event_shape]
[torch.Size([2]), torch.Size([3, 4, 5])]
>>> d3 = d1.independent(2)
>>> [d3.batch_shape, d3.event_shape]
[torch.Size([]), torch.Size([2, 3, 4, 5])]
```

Parameters **reinterpreted_batch_ndims** (*int*) – The number of batch dimensions to reinterpret as event dimensions.

Returns A reshaped version of this distribution.

Return type ReshapedDistribution

mask(*mask*)
Masks a distribution by a zero-one tensor that is broadcastable to the distributions batch_shape.

Parameters **mask** (*torch.Tensor*) – A zero-one valued float tensor.

Returns A masked copy of this distribution.

Return type MaskedDistribution

5.2.3 TorchDistribution

class TorchDistribution(*batch_shape=torch.Size([])*, *event_shape=torch.Size([])*, *validate_args=None*)
Bases: torch.distributions.distribution.Distribution, *pyro.distributions.torch_distribution.TorchDistributionMixin*

Base class for PyTorch-compatible distributions with Pyro support.

This should be the base class for almost all new Pyro distributions.

Note: Parameters and data should be of type Tensor and all methods return type Tensor unless otherwise noted.

Tensor Shapes:

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

```
x = d.sample(sample_shape)
assert x.shape == d.shape(sample_shape)
```

Pyro follows the same distribution shape semantics as PyTorch. It distinguishes between three different roles for tensor shapes of samples:

- *sample shape* corresponds to the shape of the iid samples drawn from the distribution. This is taken as an argument by the distribution's *sample* method.
- *batch shape* corresponds to non-identical (independent) parameterizations of the distribution, inferred from the distribution's parameter shapes. This is fixed for a distribution instance.
- *event shape* corresponds to the event dimensions of the distribution, which is fixed for a distribution class. These are collapsed when we try to score a sample from the distribution via *d.log_prob(x)*.

These shapes are related by the equation:

```
assert d.shape(sample_shape) == sample_shape + d.batch_shape + d.event_shape
```

Distributions provide a vectorized :meth‘~torch.distributions.distribution.Distribution.log_prob‘ method that evaluates the log probability density of each event in a batch independently, returning a tensor of shape `sample_shape + d.batch_shape`:

```
x = d.sample(sample_shape)
assert x.shape == d.shape(sample_shape)
log_p = d.log_prob(x)
assert log_p.shape == sample_shape + d.batch_shape
```

Implementing New Distributions:

Derived classes must implement the methods `sample()` (or `rsample()` if `.has_rsample == True`) and `log_prob()`, and must implement the properties `batch_shape`, and `event_shape`. Discrete classes may also implement the `enumerate_support()` method to improve gradient estimates and set `.has_enumerate_support = True`.

5.2.4 AVFMultivariateNormal

class AVFMultivariateNormal(*loc*, *scale_tril*, *control_var*)

Bases: `pyro.distributions.torch.MultivariateNormal`

Multivariate normal (Gaussian) distribution with transport equation inspired control variates (adaptive velocity fields).

A distribution over vectors in which all the elements have a joint Gaussian density.

Parameters

- **loc** (*torch.Tensor*) – D-dimensional mean vector.
- **scale_tril** (*torch.Tensor*) – Cholesky of Covariance matrix; D x D matrix.
- **control_var** (*torch.Tensor*) – 2 x L x D tensor that parameterizes the control variate; L is an arbitrary positive integer. This parameter needs to be learned (i.e. adapted) to achieve lower variance gradients. In a typical use case this parameter will be adapted concurrently with the *loc* and *scale_tril* that define the distribution.

Example usage:

```
control_var = torch.tensor(0.1 * torch.ones(2, 1, D), requires_grad=True)
opt_cv = torch.optim.Adam([control_var], lr=0.1, betas=(0.5, 0.999))

for _ in range(1000):
    d = AVFMultivariateNormal(loc, scale_tril, control_var)
    z = d.rsample()
    cost = torch.pow(z, 2.0).sum()
    cost.backward()
    opt_cv.step()
    opt_cv.zero_grad()
```

arg_constraints = {'control_var': <torch.distributions.constraints._Real object at 0x

rsample(*sample_shape=torch.Size([])*)

Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

5.2.5 Binomial

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

Bases: `torch.distributions.distribution.Distribution`, *pyro.distributions.torch_distribution.TorchDistributionMixin*

Creates a Binomial distribution parameterized by *total_count* and either *probs* or *logits* (but not both). *total_count* must be broadcastable with *probs*/*logits*.

This is adapted from `torch.distributions.binomial.Binomial`, with the important difference that *total_count* is not limited to being a single *int*, but can be a *torch.Tensor*.

Example:

```
>>> m = Binomial(100, torch.Tensor([0 , .2, .8, 1]))
>>> m.sample()
 0
 22
 71
 100
[torch.FloatTensor of size 4]]

>>> m = Binomial(torch.Tensor([[5.], [10.]]), torch.Tensor([0.5, 0.8]))
>>> m.sample()
 4  5
 7  6
[torch.FloatTensor of size (2,2)]
```

Parameters

- **total_count** (*(Tensor)*) – number of Bernoulli trials
- **probs** (*(Tensor)*) – Event probabilities
- **logits** (*(Tensor)*) – Event log-odds

arg_constraints = {'probs': <torch.distributions.constraints._Interval object at 0x7f

enumerate_support()

Returns tensor containing all values supported by a discrete distribution. The result will enumerate

over dimension 0, so the shape of the result will be *(cardinality,) + batch_shape + event_shape* (where *event_shape = ()* for univariate distributions).

Note that this enumerates over all batched tensors in lock-step *[[0, 0], [1, 1], ...]*. To iterate over the full Cartesian product use *itertools.product(m.enumerate_support())*.

Returns: Tensor iterating over dimension 0.

expand(*batch_shape*)
Expands a distribution to a desired batch_shape.

Note that this is more general than `expand_by()` because `d.expand_by(sample_shape)` can be reduced to `d.expand(sample_shape + d.batch_shape)`.

Parameters **batch_shape** (*torch.Size*) – The target batch_shape. This must compatible with `self.batch_shape` similar to the requirements of `torch.Tensor.expand()`: the target batch_shape must be at least as long as `self.batch_shape`, and for each non-singleton dim of `self.batch_shape`, batch_shape must either agree or be set to `-1`.

Returns An expanded version of this distribution.

Return type ReshapedDistribution

has_enumerate_support = True

log_prob(*value*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped sample or sample_shape shaped batch of samples if the distribution parameters are batched.

support

variance

5.2.6 Delta

class Delta(*v*, *log_density=0.0*, *event_dim=0*, *validate_args=None*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

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.Tensor*) – The single support element.
- **log_density** (*torch.Tensor*) – An optional density for this Delta. This is useful to keep the class of *Delta* distributions closed under differentiable transformation.

- **event_dim** (*int*) – Optional event dimension, defaults to zero.

arg_constraints = {'log_density': <torch.distributions.constraints._Real object at 0x

expand(*batch_shape*)
Expands a distribution to a desired batch_shape.

Note that this is more general than expand_by() because d.expand_by(sample_shape) can be reduced to d.expand(sample_shape + d.batch_shape).

Parameters **batch_shape** (*torch.Size*) – The target batch_shape. This must compatible with self.batch_shape similar to the requirements of torch.Tensor.expand(): the target batch_shape must be at least as long as self.batch_shape, and for each non-singleton dim of self.batch_shape, batch_shape must either agree or be set to -1.

Returns An expanded version of this distribution.

Return type ReshapedDistribution

has_rsample = True

log_prob(*x*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

mean

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

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

variance

5.2.7 EmpiricalDistribution

class Empirical(*validate_args=None*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Empirical distribution associated with the sampled data.

add(*value*, *weight=None*, *log_weight=None*)
Adds a new data point to the sample. The values in successive calls to add must have the same tensor shape and size. Optionally, an importance weight can be specified via log_weight or weight (default value of *1* is used if not specified).

Parameters

- **value** (*torch.Tensor*) – tensor to add to the sample.
- **weight** (*torch.Tensor*) – log weight (optional) corresponding to the sample.
- **log_weight** (*torch.Tensor*) – weight (optional) corresponding to the sample.

arg_constraints = {}

enumerate_support()
See pyro.distributions.torch_distribution.TorchDistribution.enumerate_support()

event_shape
See pyro.distributions.torch_distribution.TorchDistribution.event_shape()

get_samples_and_weights()

has_enumerate_support = True

log_prob(*value*)
Returns the log of the probability mass function evaluated at value. Note that this currently only supports scoring values with empty sample_shape, i.e. an arbitrary batched sample is not allowed.

Parameters **value** (*torch.Tensor*) – scalar or tensor value to be scored.

mean
See pyro.distributions.torch_distribution.TorchDistribution.mean()

sample(*sample_shape=torch.Size([])*)
See pyro.distributions.torch_distribution.TorchDistribution.sample()

sample_size
Number of samples that constitute the empirical distribution.

Return int number of samples collected.

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

variance
See pyro.distributions.torch_distribution.TorchDistribution.variance()

5.2.8 HalfCauchy

class HalfCauchy(*loc=0*, *scale=1*)
Bases: pyro.distributions.torch.TransformedDistribution

Half-Cauchy distribution.

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

Parameters

- **loc** (*torch.Tensor*) – lower bound of the distribution.
- **scale** (*torch.Tensor*) – half width at half maximum.

arg_constraints = {'loc': <torch.distributions.constraints._Real object at 0x7f7207bc

entropy()
Returns entropy of distribution, batched over batch_shape.

Returns: Tensor of shape batch_shape.

expand(*batch_shape*)
Expands a distribution to a desired batch_shape.

Note that this is more general than expand_by() because d.expand_by(sample_shape) can be reduced to d.expand(sample_shape + d.batch_shape).

Parameters **batch_shape** (*torch.Size*) – The target batch_shape. This must compatible with self.batch_shape similar to the requirements of torch.Tensor.expand(): the target batch_shape must be at least as long as self.batch_shape, and for each non-singleton dim of self.batch_shape, batch_shape must either agree or be set to -1.

Returns An expanded version of this distribution.

Return type `ReshapedDistribution`

loc

log_prob(*value*)
Scores the sample by inverting the transform(s) and computing the score using the score of the base distribution and the log abs det jacobian.

scale

support

5.2.9 LowRankMultivariateNormal

class LowRankMultivariateNormal(*loc*, *W_term*, *D_term*, *trace_term=None*)
Bases: `pyro.distributions.torch_distribution.TorchDistribution`

Low Rank Multivariate Normal distribution.

Implements fast computation for log probability of Multivariate Normal distribution when the covariance matrix has the form:

```
covariance_matrix = W @ W.T + D.
```

Here D is a diagonal vector and `W` is a matrix of size `N x M`. The computation will be beneficial when `M << N`.

Parameters

- **loc** (*torch.Tensor*) – Mean. Must be a 1D or 2D tensor with the last dimension of size N.
- **W_term** (*torch.Tensor*) – W term of covariance matrix. Must be in 2 dimensional of size N x M.
- **D_term** (*torch.Tensor*) – D term of covariance matrix. Must be in 1 dimensional of size N.
- **trace_term** (*float*) – A optional term to be added into Mahalabonis term according to p(y) = N(y|loc, cov).exp(-1/2 * trace_term).

arg_constraints = {'covariance_matrix_D_term': <torch.distributions.constraints._Grea

has_rsample = True

log_prob(*value*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

mean

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

scale_tril

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

variance

5.2.10 MixtureOfDiagNormalsSharedCovariance

class MixtureOfDiagNormalsSharedCovariance(*locs*, *scale*, *logits*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Mixture of Normal distributions with diagonal covariance matrices.

That is, this distribution is a mixture with K components, where each component distribution is a D-dimensional Normal distribution with a D-dimensional mean parameter loc and a D-dimensional diagonal covariance matrix specified by a scale parameter *scale*. The K different component means are gathered into the parameter *locs* and the scale parameter is shared between all K components. The mixture weights are controlled by a K-dimensional vector of softmax logits, *logits*. This distribution implements pathwise derivatives for samples from the distribution.

See reference [1] for details on the implementations of the pathwise derivative. Please consider citing this reference if you use the pathwise derivative in your research.

[1] Pathwise Derivatives for Multivariate Distributions, Martin Jankowiak & Theofanis Karaletsos. arXiv:1806.01856

Parameters

- **locs** (*torch.Tensor*) – K x D mean matrix
- **scale** (*torch.Tensor*) – shared D-dimensional scale vector
- **logits** (*torch.Tensor*) – K-dimensional vector of softmax logits

arg_constraints = {'locs': <torch.distributions.constraints._Real object at 0x7f7207b

has_rsample = True

log_prob(*value*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

5.2.11 OMTMultivariateNormal

class OMTMultivariateNormal(*loc*, *scale_tril*)
Bases: pyro.distributions.torch.MultivariateNormal

Multivariate normal (Gaussian) distribution with OMT gradients w.r.t. both parameters. Note the gradient computation w.r.t. the Cholesky factor has cost O(D^3), although the resulting gradient variance is generally expected to be lower.

A distribution over vectors in which all the elements have a joint Gaussian density.

Parameters

- **loc** (*torch.Tensor*) – Mean.
- **scale_tril** (*torch.Tensor*) – Cholesky of Covariance matrix.

arg_constraints = {'loc': <torch.distributions.constraints._Real object at 0x7f7207bc

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

5.2.12 RelaxedBernoulliStraightThrough

class RelaxedBernoulliStraightThrough(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: pyro.distributions.torch.RelaxedBernoulli

An implementation of RelaxedBernoulli with a straight-through gradient estimator.

This distribution has the following properties:

- The samples returned by the *rsample()* method are discrete/quantized.
- The *log_prob()* method returns the log probability of the relaxed/unquantized sample using the GumbelSoftmax distribution.
- In the backward pass the gradient of the sample with respect to the parameters of the distribution uses the relaxed/unquantized sample.

References:

[1] The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables, Chris J. Maddison, Andriy Mnih, Yee Whye Teh

[2] Categorical Reparameterization with Gumbel-Softmax, Eric Jang, Shixiang Gu, Ben Poole

log_prob(*value*)
See pyro.distributions.torch.RelaxedBernoulli.log_prob()

rsample(*sample_shape=torch.Size([])*)
See pyro.distributions.torch.RelaxedBernoulli.rsample()

5.2.13 RelaxedOneHotCategoricalStraightThrough

class RelaxedOneHotCategoricalStraightThrough(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: pyro.distributions.torch.RelaxedOneHotCategorical

An implementation of RelaxedOneHotCategorical with a straight-through gradient estimator.

This distribution has the following properties:

- The samples returned by the *rsample()* method are discrete/quantized.
- The *log_prob()* method returns the log probability of the relaxed/unquantized sample using the GumbelSoftmax distribution.
- In the backward pass the gradient of the sample with respect to the parameters of the distribution uses the relaxed/unquantized sample.

References:

[1] The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables, Chris J. Maddison, Andriy Mnih, Yee Whye Teh

[2] Categorical Reparameterization with Gumbel-Softmax, Eric Jang, Shixiang Gu, Ben Poole

log_prob(*value*)
See pyro.distributions.torch.RelaxedOneHotCategorical.log_prob()

rsample(*sample_shape=torch.Size([])*)
See pyro.distributions.torch.RelaxedOneHotCategorical.rsample()

5.2.14 Rejector

class Rejector(*propose*, *log_prob_accept*, *log_scale*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Rejection sampled distribution given an acceptance rate function.

Parameters

- **propose** (Distribution) – A proposal distribution that samples batched proposals via `propose()`. *rsample()* supports a `sample_shape` arg only if `propose()` supports a `sample_shape` arg.
- **log_prob_accept** (*callable*) – A callable that inputs a batch of proposals and returns a batch of log acceptance probabilities.
- **log_scale** – Total log probability of acceptance.

has_rsample = True

log_prob(*x*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

score_parts(*x*)
Computes ingredients for stochastic gradient estimators of ELBO.

The default implementation is correct both for non-reparameterized and for fully reparameterized distributions. Partially reparameterized distributions should override this method to compute correct *.score_function* and *.entropy_term* parts.

Parameters **x** (*torch.Tensor*) – A single value or batch of values.

Returns A *ScoreParts* object containing parts of the ELBO estimator.

Return type ScoreParts

5.2.15 VonMises

class VonMises(*loc*, *concentration*, *validate_args=None*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

A circular von Mises distribution.

This implementation uses polar coordinates. The `loc` and `value` args can be any real number (to facilitate unconstrained optimization), but are interpreted as angles modulo 2 pi.

See *VonMises3D* for a 3D cartesian coordinate cousin of this distribution.

Currently only *log_prob()* is implemented.

Parameters

- **loc** (*torch.Tensor*) – an angle in radians.
- **concentration** (*torch.Tensor*) – concentration parameter

arg_constraints = {'concentration': <torch.distributions.constraints._GreaterThan obj

expand(*batch_shape*)
Expands a distribution to a desired batch_shape.

Note that this is more general than expand_by() because d.expand_by(sample_shape) can be reduced to d.expand(sample_shape + d.batch_shape).

Parameters **batch_shape** (*torch.Size*) – The target batch_shape. This must compatible with self.batch_shape similar to the requirements of torch.Tensor.expand(): the target batch_shape must be at least as long as self.batch_shape, and for each non-singleton dim of self.batch_shape, batch_shape must either agree or be set to −1.

Returns An expanded version of this distribution.

Return type ReshapedDistribution

log_prob(*value*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

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

5.2.16 VonMises3D

class VonMises3D(*concentration*, *validate_args=None*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Spherical von Mises distribution.

This implementation combines the direction parameter and concentration parameter into a single combined parameter that contains both direction and magnitude. The value arg is represented in cartesian coordinates: it must be a normalized 3-vector that lies on the 2-sphere.

See *VonMises* for a 2D polar coordinate cousin of this distribution.

Currently only *log_prob()* is implemented.

Parameters **concentration** (*torch.Tensor*) – A combined location-and-concentration vector. The direction of this vector is the location, and its magnitude is the concentration.

arg_constraints = {'concentration': <torch.distributions.constraints._Real object at

expand(*batch_shape*)
Expands a distribution to a desired batch_shape.

Note that this is more general than expand_by() because d.expand_by(sample_shape) can be reduced to d.expand(sample_shape + d.batch_shape).

Parameters **batch_shape** (*torch.Size*) – The target batch_shape. This must compatible with self.batch_shape similar to the requirements of torch.Tensor.expand(): the target batch_shape must be at least as long as self.batch_shape, and for each non-singleton dim of self.batch_shape, batch_shape must either agree or be set to −1.

Returns An expanded version of this distribution.

Return type ReshapedDistribution

log_prob(*value*)
Returns the log of the probability density/mass function evaluated at *value*.

Args: value (Tensor):

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

5.3 Transformed Distributions

5.3.1 InverseAutoRegressiveFlow

class InverseAutoregressiveFlow(*input_dim*, *hidden_dim*, *sigmoid_bias=2.0*, *permutation=None*)

Bases: torch.distributions.transforms.Transform

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

Example usage:

```
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> iaf = InverseAutoregressiveFlow(10, 40)
>>> iaf_module = pyro.module("my_iaf", iaf.module)
>>> iaf_dist = dist.TransformedDistribution(base_dist, [iaf])
>>> iaf_dist.sample()
    tensor([-0.4071, -0.5030,  0.7924, -0.2366, -0.2387, -0.1417,  0.0868,
            0.1389, -0.4629,  0.0986])
```

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

arn

Return type pyro.nn.AutoRegressiveNN

Return the AutoRegressiveNN associated with the InverseAutoregressiveFlow

codomain = <torch.distributions.constraints._Real object>

log_abs_det_jacobian(*x*, *y*)
Calculates the elementwise determinant of the log jacobian

CHAPTER 6

Parameters

Parameters in Pyro are basically thin wrappers around PyTorch Tensors 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 saved and loaded from disk using *save* and *load*.

clear()

Clear the ParamStore

get_all_param_names()
Get all parameter names in the ParamStore

get_param(*name*, *init_tensor=None*, *constraint=<torch.distributions.constraints._Real object>*)
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.Tensor*) – initial tensor

Returns parameter

Return type torch.Tensor

get_state()
Get the ParamStore state.

load(*filename*)
Loads parameters from disk

Note: If using *pyro.module()* on parameters loaded from disk, be sure to set the `update_module_params` flag:

```
pyro.get_param_store().load('saved_params.save')
pyro.module('module', nn, update_module_params=True)
```

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

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.Tensor*) – 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

module_from_param_with_module_name(*param_name*)

param_with_module_name(*pyro_name*, *param_name*)

user_param_name(*param_name*)

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.Tensor*) – 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 Pyro Optimizers

class PyroOptim(*optim_constructor*, *optim_args*)

Bases: object

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

__call__(*params*, **args*, ***kwargs*)

Parameters **params** (*an iterable of strings*) – a list of parameters

Do an optimization step for each param in params. If a given param has never been seen before, initialize an optimizer for it.

get_state()

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

set_state(*state_dict*)

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

save(*filename*)

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

Save optimizer state to disk

load(*filename*)

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

Load optimizer state from disk

AdagradRMSProp(*optim_args*)
A wrapper for an optimizer that is a mash-up of Adagrad and RMSprop.

ClippedAdam(*optim_args*)
A wrapper for a modification of the Adam optimization algorithm that supports gradient clipping.

class PyroLRScheduler(*scheduler_constructor*, *optim_args*)
Bases: *pyro.optim.optim.PyroOptim*

A wrapper for torch.optim.lr_scheduler objects that adjust learning rates for dynamically generated parameters.

Parameters

- **optim_constructor** – a torch.optim.lr_scheduler
- **optim_args** – a dictionary of learning arguments for the optimizer or a callable that returns such dictionaries. must contain the key 'optimizer' with pytorch optimizer value

Example:

```
optimizer = torch.optim.SGD
pyro_scheduler = pyro.optim.ExponentialLR({'optimizer': optimizer, 'optim_args': {
↪'lr': 0.01}, 'gamma': 0.1})
```

set_epoch(*epoch*)

8.2 PyTorch Optimizers

LBFGS(*optim_args*)
Wraps torch.optim.LBFGS with *PyroOptim*.

Adamax(*optim_args*)
Wraps torch.optim.Adamax with *PyroOptim*.

Adagrad(*optim_args*)
Wraps torch.optim.Adagrad with *PyroOptim*.

SGD(*optim_args*)
Wraps torch.optim.SGD with *PyroOptim*.

Adam(*optim_args*)
Wraps torch.optim.Adam with *PyroOptim*.

Rprop(*optim_args*)
Wraps torch.optim.Rprop with *PyroOptim*.

ASGD(*optim_args*)
Wraps torch.optim.ASGD with *PyroOptim*.

RMSprop(*optim_args*)
Wraps torch.optim.RMSprop with *PyroOptim*.

SparseAdam(*optim_args*)
Wraps torch.optim.SparseAdam with *PyroOptim*.

Adadelta(*optim_args*)
Wraps `torch.optim.Adadelta` with *PyroOptim*.

MultiStepLR(*optim_args*)
Wraps `torch.optim.MultiStepLR` with *PyroLRScheduler*.

ReduceLROnPlateau(*optim_args*)
Wraps `torch.optim.ReduceLROnPlateau` with *PyroLRScheduler*.

StepLR(*optim_args*)
Wraps `torch.optim.StepLR` with *PyroLRScheduler*.

CosineAnnealingLR(*optim_args*)
Wraps `torch.optim.CosineAnnealingLR` with *PyroLRScheduler*.

LambdaLR(*optim_args*)
Wraps `torch.optim.LambdaLR` with *PyroLRScheduler*.

ExponentialLR(*optim_args*)
Wraps `torch.optim.ExponentialLR` with *PyroLRScheduler*.

8.3 Higher-Order Optimizers

class MultiOptimizer
Bases: `object`

Base class of optimizers that make use of higher-order derivatives.

Higher-order optimizers generally use `torch.autograd.grad()` rather than `torch.Tensor.backward()`, and therefore require a different interface from usual Pyro and PyTorch optimizers. In this interface, the *step()* method inputs a `loss` tensor to be differentiated, and backpropagation is triggered one or more times inside the optimizer.

Derived classes must implement *step()* to compute derivatives and update parameters in-place.

Example:

```
tr = poutine.trace(model).get_trace(*args, **kwargs)
loss = -tr.log_prob_sum()
params = {name: site['value'].unconstrained()
          for name, site in tr.nodes.items()
          if site['type'] == 'param'}
optim.step(loss, params)
```

step(*loss*, *params*)
Performs an in-place optimization step on parameters given a differentiable `loss` tensor.

Note that this detaches the updated tensors.

Parameters

- **loss** (*torch.Tensor*) – A differentiable tensor to be minimized. Some optimizers require this to be differentiable multiple times.
- **params** (*dict*) – A dictionary mapping param name to unconstrained value as stored in the param store.

get_step(*loss*, *params*)
Computes an optimization step of parameters given a differentiable `loss` tensor, returning the updated values.

Note that this preserves derivatives on the updated tensors.

Parameters

- **loss** (*torch.Tensor*) – A differentiable tensor to be minimized. Some optimizers require this to be differentiable multiple times.
- **params** (*dict*) – A dictionary mapping param name to unconstrained value as stored in the param store.

Returns A dictionary mapping param name to updated unconstrained value.

Return type dict

class PyroMultiOptimizer(*optim*)

Bases: *pyro.optim.multi.MultiOptimizer*

Facade to wrap *PyroOptim* objects in a *MultiOptimizer* interface.

step(*loss*, *params*)

Performs an in-place optimization step on parameters given a differentiable `loss` tensor.

Note that this detaches the updated tensors.

Parameters

- **loss** (*torch.Tensor*) – A differentiable tensor to be minimized. Some optimizers require this to be differentiable multiple times.
- **params** (*dict*) – A dictionary mapping param name to unconstrained value as stored in the param store.

class TorchMultiOptimizer(*optim_constructor*, *optim_args*)

Bases: *pyro.optim.multi.PyroMultiOptimizer*

Facade to wrap `Optimizer` objects in a *MultiOptimizer* interface.

class MixedMultiOptimizer(*parts*)

Bases: *pyro.optim.multi.MultiOptimizer*

Container class to combine different *MultiOptimizer* instances for different parameters.

Parameters **parts** (*list*) – A list of `(names, optim)` pairs, where each `names` is a list of parameter names, and each `optim` is a *MultiOptimizer* object to be used for the named parameters. Together the `names` should partition up all desired parameters to optimize.

step(*loss*, *params*)

Performs an in-place optimization step on parameters given a differentiable `loss` tensor.

Note that this detaches the updated tensors.

Parameters

- **loss** (*torch.Tensor*) – A differentiable tensor to be minimized. Some optimizers require this to be differentiable multiple times.
- **params** (*dict*) – A dictionary mapping param name to unconstrained value as stored in the param store.

get_step(*loss*, *params*)

Computes an optimization step of parameters given a differentiable `loss` tensor, returning the updated values.

Note that this preserves derivatives on the updated tensors.

Parameters

- **loss** (*torch.Tensor*) – A differentiable tensor to be minimized. Some optimizers require this to be differentiable multiple times.
- **params** (*dict*) – A dictionary mapping param name to unconstrained value as stored in the param store.

Returns A dictionary mapping param name to updated unconstrained value.

Return type dict

class Newton(*trust_radii={}*)

Bases: *pyro.optim.multi.MultiOptimizer*

Implementation of *MultiOptimizer* that performs a Newton update on batched low-dimensional variables, optionally regularizing via a per-parameter `trust_radius`. See *newton_step()* for details.

Parameters **trust_radii** (*dict*) – a dict mapping parameter name to radius of trust region. Missing names will use unregularized Newton update, equivalent to infinite trust radius.

get_step(*loss*, *params*)

Computes an optimization step of parameters given a differentiable `loss` tensor, returning the updated values.

Note that this preserves derivatives on the updated tensors.

Parameters

- **loss** (*torch.Tensor*) – A differentiable tensor to be minimized. Some optimizers require this to be differentiable multiple times.
- **params** (*dict*) – A dictionary mapping param name to unconstrained value as stored in the param store.

Returns A dictionary mapping param name to updated unconstrained value.

Return type dict

CHAPTER 9

Poutine (Effect handlers)

Beneath the built-in inference algorithms, Pyro has a library of composable effect handlers for creating new inference algorithms and working with probabilistic programs. Pyro's inference algorithms are all built by applying these handlers to stochastic functions.

9.1 Handlers

Poutine is a library of composable effect handlers for recording and modifying the behavior of Pyro programs. These lower-level ingredients simplify the implementation of new inference algorithms and behavior.

Handlers can be used as higher-order functions, decorators, or context managers to modify the behavior of functions or blocks of code:

For example, consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

We can mark sample sites as observed using `condition`, which returns a callable with the same input and output signatures as `model`:

```
>>> conditioned_model = poutine.condition(model, data={"z": 1.0})
```

We can also use handlers as decorators:

```
>>> @pyro.condition(data={"z": 1.0})
... def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

Or as context managers:

```
>>> with pyro.condition(data={"z": 1.0}):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(0., s))
...     y = z ** 2
```

Handlers compose freely:

```
>>> conditioned_model = poutine.condition(model, data={"z": 1.0})
>>> traced_model = poutine.trace(conditioned_model)
```

Many inference algorithms or algorithmic components can be implemented in just a few lines of code:

```
guide_tr = poutine.trace(guide).get_trace(...)
model_tr = poutine.trace(poutine.replay(conditioned_model, trace=tr)).get_trace(...)
monte_carlo_elbo = model_tr.log_prob_sum() - guide_tr.log_prob_sum()
```

block (*fn=None*, *hide_fn=None*, *expose_fn=None*, *hide=None*, *expose=None*, *hide_types=None*, *expose_types=None*)

This handler selectively hides Pyro primitive sites from the outside world. Default behavior: block everything.

A site is hidden if at least one of the following holds:

0. `hide_fn(msg) is True` or `(not expose_fn(msg)) is True`
1. `msg["name"] in hide`
2. `msg["type"] in hide_types`
3. `msg["name"] not in expose and msg["type"] not in expose_types`
4. `hide`, `hide_types`, and `expose_types` are all `None`

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

```
>>> def fn():
...     a = pyro.sample("a", dist.Normal(0., 1.))
...     return pyro.sample("b", dist.Normal(a, 1.))
>>> fn_inner = trace(fn)
>>> fn_outer = trace(block(fn_inner, 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
```

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **hide** – list of site names to hide
- **expose** – list of site names to be exposed while all others hidden
- **hide_types** – list of site types to be hidden
- **expose_types** – list of site types to be exposed while all others hidden

Param hide_fn: function that takes a site and returns True to hide the site or False/None to expose it. If specified, all other parameters are ignored. Only specify one of hide_fn or expose_fn, not both.

Param expose_fn: function that takes a site and returns True to expose the site or False/None to hide it. If specified, all other parameters are ignored. Only specify one of hide_fn or expose_fn, not both.

Returns stochastic function decorated with a *BlockMessenger*

broadcast (*fn=None*)

Automatically broadcasts the batch shape of the stochastic function at a sample site when inside a single or nested iarange context. The existing *batch_shape* must be broadcastable with the size of the *iarange* contexts installed in the *cond_indep_stack*.

Notice how *model_automatic_broadcast* below automates expanding of distribution batch shapes. This makes it easy to modularize a Pyro model as the sub-components are agnostic of the wrapping *iarange* contexts.

```
>>> def model_broadcast_by_hand():
...     with pyro.iarange("batch", 100, dim=-2):
...         with pyro.iarange("components", 3, dim=-1):
...             sample = pyro.sample("sample", dist.Bernoulli(torch.ones(3) * 0.5)
...                                                 .expand_by(100))
...             assert sample.shape == torch.Size((100, 3))
...     return sample
```

```
>>> @poutine.broadcast
... def model_automatic_broadcast():
...     with pyro.iarange("batch", 100, dim=-2):
...         with pyro.iarange("components", 3, dim=-1):
...             sample = pyro.sample("sample", dist.Bernoulli(torch.tensor(0.5)))
...             assert sample.shape == torch.Size((100, 3))
...     return sample
```

condition (*fn=None*, *data=None*)

Given a stochastic function with some sample statements and a dictionary of observations at names, change the sample statements at those names into observes with those values.

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

To observe a value for site *z*, we can write

```
>>> conditioned_model = condition(model, data={"z": torch.tensor(1.)})
```

This is equivalent to adding *obs=value* as a keyword argument to *pyro.sample("z", ...)* in *model*.

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **data** – a dict or a *Trace*

Returns stochastic function decorated with a *ConditionMessenger*

do (*fn=None*, *data=None*)
Given a stochastic function with some sample statements and a dictionary of values at names, set the return values of those sites equal to the values and hide them from the rest of the stack as if they were hard-coded to those values by using `block`.

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

To intervene with a value for site *z*, we can write

```
>>> intervened_model = do(model, data={"z": torch.tensor(1.)})
```

This is equivalent to replacing *z = pyro.sample("z", ...)* with *z = value*.

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **data** – a `dict` or a *Trace*

Returns stochastic function decorated with a *BlockMessenger* and *pyro.poutine.condition_messenger.ConditionMessenger*

enum (*fn=None*, *first_available_dim=None*)
Enumerates in parallel over discrete sample sites marked `infer={"enumerate": "parallel"}`.

Parameters **first_available_dim** (*int*) – The first tensor dimension (counting from the right) that is available for parallel enumeration. This dimension and all dimensions left may be used internally by Pyro.

escape (*fn=None*, *escape_fn=None*)
Given a callable that contains Pyro primitive calls, evaluate escape_fn on each site, and if the result is True, raise a *NonlocalExit* exception that stops execution and returns the offending site.

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **escape_fn** – function that takes a partial trace and a site, and returns a boolean value to decide whether to exit at that site

Returns stochastic function decorated with *EscapeMessenger*

indep (*fn=None*, *name=None*, *size=None*, *dim=None*)

Note: Low-level; use *iarange* instead.

This messenger 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 *TraceMessenger*.

infer_config (*fn=None*, *config_fn=None*)
Given a callable that contains Pyro primitive calls and a callable taking a trace site and returning a dictionary, updates the value of the infer kwarg at a sample site to config_fn(site).

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **config_fn** – a callable taking a site and returning an infer dict

Returns stochastic function decorated with `InferConfigMessenger`

lift (*fn=None*, *prior=None*)

Given a stochastic function with param calls and a prior distribution, create a stochastic function where all param calls are replaced by sampling from prior. Prior should be a callable or a dict of names to callables.

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
>>> lifted_model = lift(model, prior={"s": dist.Exponential(0.3)})
```

`lift` makes `param` statements behave like `sample` statements using the distributions in `prior`. In this example, site *s* will now behave as if it was replaced with `s = pyro.sample("s", dist.Exponential(0.3))`:

```
>>> tr = trace(lifted_model).get_trace(0.0)
>>> tr.nodes["s"]["type"] == "sample"
True
>>> tr2 = trace(lifted_model).get_trace(0.0)
>>> bool((tr2.nodes["s"]["value"] == tr.nodes["s"]["value"]).all())
False
```

Parameters

- **fn** – function whose parameters will be lifted to random values
- **prior** – prior function in the form of a Distribution or a dict of stochastic fns

Returns `fn` decorated with a *`LiftMessenger`*

replay (*fn=None*, *trace=None*, *params=None*)

Given a callable that contains Pyro primitive calls, return a callable that runs the original, reusing the values at sites in trace at those sites in the new trace

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

`replay` makes `sample` statements behave as if they had sampled the values at the corresponding sites in the trace:

```
>>> old_trace = trace(model).get_trace(1.0)
>>> replayed_model = replay(model, trace=old_trace)
>>> bool(replayed_model(0.0) == old_trace.nodes["_RETURN"]["value"])
True
```

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)

- **trace** – a *Trace* data structure to replay against
- **params** – dict of names of param sites and constrained values in fn to replay against

Returns a stochastic function decorated with a *ReplayMessenger*

queue (*fn=None*, *queue=None*, *max_tries=None*, *extend_fn=None*, *escape_fn=None*, *num_samples=None*)
Used in sequential enumeration over discrete variables.

Given a stochastic function and a queue, return a return value from a complete trace in the queue.

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **queue** – a queue data structure like multiprocessing.Queue to hold partial traces
- **max_tries** – maximum number of attempts to compute a single complete trace
- **extend_fn** – function (possibly stochastic) that takes a partial trace and a site, and returns a list of extended traces
- **escape_fn** – function (possibly stochastic) that takes a partial trace and a site, and returns a boolean value to decide whether to exit
- **num_samples** – optional number of extended traces for extend_fn to return

Returns stochastic function decorated with poutine logic

scale (*fn=None*, *scale=None*)
Given a stochastic function with some sample statements and a positive scale factor, scale the score of all sample and observe sites in the function.

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s), obs=1.0)
...     return z ** 2
```

`scale` multiplicatively scales the log-probabilities of sample sites:

```
>>> scaled_model = scale(model, scale=0.5)
>>> scaled_tr = trace(scaled_model).get_trace(0.0)
>>> unscaled_tr = trace(model).get_trace(0.0)
>>> bool((scaled_tr.log_prob_sum() == 0.5 * unscaled_tr.log_prob_sum()).all())
True
```

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **scale** – a positive scaling factor

Returns stochastic function decorated with a *ScaleMessenger*

trace (*fn=None*, *graph_type=None*, *param_only=None*, *strict_names=None*)
Return a handler that records the inputs and outputs of primitive calls and their dependencies.

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

We can record its execution using `trace` and use the resulting data structure to compute the log-joint probability of all of the sample sites in the execution or extract all parameters.

```
>>> trace = trace(model).get_trace(0.0)
>>> logp = trace.log_prob_sum()
>>> params = [trace.nodes[name]["value"].unconstrained() for name in trace.param_
↪nodes]
```

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **graph_type** – string that specifies the kind of graph to construct
- **param_only** – if true, only records params and not samples

Returns stochastic function decorated with a *TraceMessenger*

9.2 Trace

class Trace(**args*, ***kwargs*)

Bases: `object`

Execution trace data structure built on top of `networkx.DiGraph`.

An execution trace of a Pyro program is a record of every call to `pyro.sample()` and `pyro.param()` in a single execution of that program. Traces are directed graphs whose nodes represent primitive calls or input/output, and whose edges represent conditional dependence relationships between those primitive calls. They are created and populated by `poutine.trace`.

Each node (or site) in a trace contains the name, input and output value of the site, as well as additional metadata added by inference algorithms or user annotation. In the case of `pyro.sample`, the trace also includes the stochastic function at the site, and any observed data added by users.

Consider the following Pyro program:

```
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
...     z = pyro.sample("z", dist.Normal(x, s))
...     return z ** 2
```

We can record its execution using `pyro.poutine.trace` and use the resulting data structure to compute the log-joint probability of all of the sample sites in the execution or extract all parameters.

```
>>> trace = pyro.poutine.trace(model).get_trace(0.0)
>>> logp = trace.log_prob_sum()
>>> params = [trace.nodes[name]["value"].unconstrained() for name in trace.param_
↪nodes]
```

We can also inspect or manipulate individual nodes in the trace. `trace.nodes` contains a `collections.OrderedDict` of site names and metadata corresponding to `x`, `s`, `z`, and the return value:

```
>>> list(name for name in trace.nodes.keys())
["_INPUT", "s", "z", "_RETURN"]
```

As in networkx.DiGraph, values of trace.nodes are dictionaries of node metadata:

```
>>> trace.nodes["z"]
{'type': 'sample', 'name': 'z', 'is_observed': False,
 'fn': Normal(), 'value': tensor(0.6480), 'args': (), 'kwargs': {},
 'infer': {}, 'scale': 1.0, 'cond_indep_stack': (),
 'done': True, 'stop': False, 'continuation': None}
```

'infer' is a dictionary of user- or algorithm-specified metadata. 'args' and 'kwargs' are the arguments passed via pyro.sample to fn.__call__ or fn.log_prob. 'scale' is used to scale the log-probability of the site when computing the log-joint. 'cond_indep_stack' contains data structures corresponding to pyro.iarange contexts appearing in the execution. 'done', 'stop', and 'continuation' are only used by Pyro's internals.

add_edge
Identical to networkx.DiGraph.add_edge()

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 networkx.DiGraph.add_node() but raises an error when attempting to add a duplicate node instead of silently overwriting.

compute_log_prob(*site_filter=<function <lambda>>*)
Compute the site-wise log probabilities of the trace. Each log_prob has shape equal to the corresponding batch_shape. Each log_prob_sum is a scalar. Both computations are memoized.

compute_score_parts()
Compute the batched local score parts at each site of the trace. Each log_prob has shape equal to the corresponding batch_shape. Each log_prob_sum is a scalar. All computations are memoized.

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

edges
Identical to networkx.DiGraph.edges

graph
Identical to networkx.DiGraph.graph

in_degree
Identical to networkx.DiGraph.in_degree()

is_directed
Identical to networkx.DiGraph.is_directed

iter_stochastic_nodes()

Returns an iterator over stochastic nodes in the trace.

log_prob_sum(*site_filter=<function <lambda>>*)
Compute the site-wise log probabilities of the trace. Each log_prob has shape equal to the corresponding batch_shape. Each log_prob_sum is a scalar. The computation of log_prob_sum is memoized.

Returns total log probability.

Return type torch.Tensor

nodes
Identical to networkx.DiGraph.nodes

nonreparam_stochastic_nodes

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

observation_nodes

Returns a list of names of observe sites

param_nodes

Returns a list of names of param sites

remove_node
Identical to networkx.DiGraph.remove_node()

reparameterized_nodes

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

stochastic_nodes

Returns a list of names of sample sites

successors
Identical to networkx.DiGraph.successors()

9.3 Messengers

Messenger objects contain the implementations of the effects exposed by handlers. Advanced users may modify the implementations of messengers behind existing handlers or write new messengers that implement new effects and compose correctly with the rest of the library.

9.3.1 Messenger

class Messenger
Bases: object

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

This is the base Messenger 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 `Messenger()(fn)`.

Class of transformers for messages passed during inference. Most inference operations are implemented in subclasses of this.

9.3.2 BlockMessenger

class BlockMessenger(*hide_fn=None*, *expose_fn=None*, *hide_all=True*, *expose_all=False*, *hide=None*, *expose=None*, *hide_types=None*, *expose_types=None*)
Bases: *pyro.poutine.messenger.Messenger*

This Messenger selectively hides Pyro primitive sites from the outside world. Default behavior: block everything. BlockMessenger has a flexible interface that allows users to specify in several different ways which sites should be hidden or exposed.

A site is hidden if at least one of the following holds:

0. `hide_fn(msg) is True` or `(not expose_fn(msg)) is True`
1. `msg["name"] in hide`
2. `msg["type"] in hide_types`
3. `msg["name"] not in expose and msg["type"] not in expose_types`
4. `hide`, `hide_types`, and `expose_types` are all `None`

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

```
>>> def fn():
...     a = pyro.sample("a", dist.Normal(0., 1.))
...     return pyro.sample("b", dist.Normal(a, 1.))
```

```
>>> fn_inner = TraceMessenger()(fn)
>>> fn_outer = TraceMessenger()(BlockMessenger(hide=["a"])(TraceMessenger()(fn)))
>>> 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
```

See the constructor for details.

Param hide_fn: function that takes a site and returns True to hide the site or False/None to expose it. If specified, all other parameters are ignored. Only specify one of hide_fn or expose_fn, not both.

Param expose_fn: function that takes a site and returns True to expose the site or False/None to hide it. If specified, all other parameters are ignored. Only specify one of hide_fn or expose_fn, not both.

Parameters

- **hide_all** (*bool*) – hide all sites
- **expose_all** (*bool*) – expose all sites normally
- **hide** (*list*) – list of site names to hide, rest will be exposed normally
- **expose** (*list*) – list of site names to expose, rest will be hidden
- **hide_types** (*list*) – list of site types to hide, rest will be exposed normally
- **expose_types** (*list*) – list of site types to expose normally, rest will be hidden

9.3.3 BroadcastMessenger

class BroadcastMessenger
Bases: *pyro.poutine.messenger.Messenger*

BroadcastMessenger automatically broadcasts the batch shape of the stochastic function at a sample site when inside a single or nested iarange context. The existing *batch_shape* must be broadcastable with the size of the *iarange* contexts installed in the *cond_indep_stack*.

9.3.4 ConditionMessenger

class ConditionMessenger(*data*)
Bases: *pyro.poutine.messenger.Messenger*

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

9.3.5 EscapeMessenger

class EscapeMessenger(*escape_fn*)
Bases: *pyro.poutine.messenger.Messenger*

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

9.3.6 IndepMessenger

class CondIndepStackFrame
Bases: *pyro.poutine.indep_messenger.CondIndepStackFrame*

vectorized

class IndepMessenger(*name*, *size*, *dim=None*)
Bases: *pyro.poutine.messenger.Messenger*

This messenger 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 `TraceMessenger`.

next_context()
Increments the counter.

9.3.7 LiftMessenger

class LiftMessenger(*prior*)
Bases: *pyro.poutine.messenger.Messenger*

Messenger 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.

9.3.8 ReplayMessenger

class ReplayMessenger(*trace=None*, *params=None*)
Bases: *pyro.poutine.messenger.Messenger*

Messenger for replaying from an existing execution trace.

9.3.9 ScaleMessenger

class ScaleMessenger(*scale*)
Bases: *pyro.poutine.messenger.Messenger*

This messenger rescales the log probability score.

This is typically used for data subsampling or for stratified sampling of data (e.g. in fraud detection where negatives vastly outnumber positives).

Parameters **scale** (*float or torch.Tensor*) – a positive scaling factor

9.3.10 TraceMessenger

class TraceHandler(*msngr*, *fn*)
Bases: object

Execution trace poutine.

A TraceHandler 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.

trace

class TraceMessenger(*graph_type=None*, *param_only=None*, *strict_names=None*)
Bases: *pyro.poutine.messenger.Messenger*

Execution trace messenger.

A TraceMessenger 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()

Returns data structure

Return type *pyro.poutine.Trace*

Helper method for a very common use case. Returns a shallow copy of `self.trace`.

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

9.4 Runtime

exception NonlocalExit(*site*, **args*, ***kwargs*)
Bases: exceptions.Exception

Exception for exiting nonlocally from poutine execution.

Used by poutine.EscapeMessenger to return site information.

reset_stack()
Reset the state of the frames remaining in the stack. Necessary for multiple re-executions in poutine.queue.

am_i_wrapped()
Checks whether the current computation is wrapped in a poutine. :returns: bool

apply_stack(*initial_msg*)
Execute the effect stack at a single site according to the following scheme:

1. For each Messenger in the stack from bottom to top, execute Messenger._process_message with the message; if the message field "stop" is True, stop; otherwise, continue
2. Apply default behavior (default_process_message) to finish remaining site execution
3. For each Messenger in the stack from top to bottom, execute _postprocess_message to update the message and internal messenger state with the site results
4. If the message field "continuation" is not None, call it with the message

Parameters **initial_msg** (*dict*) – the starting version of the trace site

Returns None

default_process_message(*msg*)
Default method for processing messages in inference. :param msg: a message to be processed :returns: None

validate_message(*msg*)
Asserts that the message has a valid format. :returns: None

9.5 Utilities

all_escape(*trace*, *msg*)

Parameters

- **trace** – a partial trace
- **msg** – the message at a Pyro primitive site

Returns boolean decision value

Utility function that checks if a site is not already in a trace.

Used by EscapeMessenger to decide whether to do a nonlocal exit at a site. Subroutine for approximately integrating out variables for variance reduction.

discrete_escape(*trace*, *msg*)

Parameters

- **trace** – a partial trace
- **msg** – the message at a Pyro primitive site

Returns boolean decision value

Utility function that checks if a sample site is discrete and not already in a trace.

Used by EscapeMessenger to decide whether to do a nonlocal exit at a site. Subroutine for integrating out discrete variables for variance reduction.

enable_validation(*is_validate*)

enum_extend(*trace*, *msg*, *num_samples=None*)

Parameters

- **trace** – a partial trace
- **msg** – the message at a Pyro primitive site
- **num_samples** – maximum number of extended traces to return.

Returns a list of traces, copies of input trace with one extra site

Utility function to copy and extend a trace with sites based on the input site whose values are enumerated from the support of the input site's distribution.

Used for exact inference and integrating out discrete variables.

is_validation_enabled()

mc_extend(*trace*, *msg*, *num_samples=None*)

Parameters

- **trace** – a partial trace
- **msg** – the message at a Pyro primitive site
- **num_samples** – maximum number of extended traces to return.

Returns a list of traces, copies of input trace with one extra site

Utility function to copy and extend a trace with sites based on the input site whose values are sampled from the input site's function.

Used for Monte Carlo marginalization of individual sample sites.

prune_subsample_sites(*trace*)
Copies and removes all subsample sites from a trace.

site_is_subsample(*site*)
Determines whether a trace site originated from a subsample statement inside an *iarange*.

CHAPTER 10

Miscellaneous Ops

The `pyro.ops` module implements high-level utilities that are mostly independent of the rest of Pyro.

class DualAveraging(*prox_center=0*, *t0=10*, *kappa=0.75*, *gamma=0.05*)

Bases: `object`

Dual Averaging is a scheme to solve convex optimization problems. It belongs to a class of subgradient methods which uses subgradients to update parameters (in primal space) of a model. Under some conditions, the averages of generated parameters during the scheme are guaranteed to converge to an optimal value. However, a counter-intuitive aspect of traditional subgradient methods is "new subgradients enter the model with decreasing weights" (see [1]). Dual Averaging scheme solves that phenomenon by updating parameters using weights equally for subgradients (which lie in a dual space), hence we have the name "dual averaging".

This class implements a dual averaging scheme which is adapted for Markov chain Monte Carlo (MCMC) algorithms. To be more precise, we will replace subgradients by some statistics calculated during an MCMC trajectory. In addition, introducing some free parameters such as `t0` and `kappa` is helpful and still guarantees the convergence of the scheme.

References

[1] *Primal-dual subgradient methods for convex problems*, Yurii Nesterov

[2] *The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo*, Matthew D. Hoffman, Andrew Gelman

Parameters

- **prox_center** (*float*) – A "prox-center" parameter introduced in [1] which pulls the primal sequence towards it.
- **t0** (*float*) – A free parameter introduced in [2] that stabilizes the initial steps of the scheme.
- **kappa** (*float*) – A free parameter introduced in [2] that controls the weights of steps of the scheme. For a small `kappa`, the scheme will quickly forget states from early steps. This should be a number in $(0.5, 1]$.
- **gamma** (*float*) – A free parameter which controls the speed of the convergence of the scheme.

step(*g*)
Updates states of the scheme given a new statistic/subgradient `g`.

Parameters **g** (*float*) – A statistic calculated during an MCMC trajectory or subgradient.

get_state()
Returns the latest x_t and average of $\{x_i\}_{i=1}^{t}$ in primal space.

velocity_verlet(*z*, *r*, *potential_fn*, *step_size*, *num_steps=1*)
Second order symplectic integrator that uses the velocity verlet algorithm.

Parameters

- **z** (*dict*) – dictionary of sample site names and their current values (type `Tensor`).
- **r** (*dict*) – dictionary of sample site names and corresponding momenta (type `Tensor`).
- **potential_fn** (*callable*) – function that returns potential energy given `z` for each sample site. The negative gradient of the function with respect to `z` determines the rate of change of the corresponding sites' momenta `r`.
- **step_size** (*float*) – step size for each time step iteration.
- **num_steps** (*int*) – number of discrete time steps over which to integrate.

Return tuple (z_next, r_next) final position and momenta, having same types as (z, r).

single_step_velocity_verlet(*z*, *r*, *potential_fn*, *step_size*, *z_grads=None*)
A special case of `velocity_verlet` integrator where `num_steps=1`. It is particular helpful for NUTS kernel.

Parameters **z_grads** (*torch.Tensor*) – optional gradients of potential energy at current `z`.

Return tuple (z_next, r_next, z_grads, potential_energy) next position and momenta, together with the potential energy and its gradient w.r.t. `z_next`.

newton_step(*loss*, *x*, *trust_radius=None*)
Performs a Newton update step to minimize loss on a batch of variables, optionally constraining to a trust region [1].

This is especially usful because the final solution of newton iteration is differentiable wrt the inputs, even when all but the final `x` is detached, due to this method's quadratic convergence [2]. `loss` must be twice-differentiable as a function of `x`. If `loss` is `2+d`-times differentiable, then the return value of this function is `d`-times differentiable.

When `loss` is interpreted as a negative log probability density, then the return values `mode,cov` of this function can be used to construct a Laplace approximation `MultivariateNormal(mode,cov)`.

Warning: Take care to detach the result of this function when used in an optimization loop. If you forget to detach the result of this function during optimization, then backprop will propagate through the entire iteration process, and worse will compute two extra derivatives for each step.

Example use inside a loop:

```
x = torch.zeros(1000, 2)  # arbitrary initial value
for step in range(100):
    x = x.detach()          # block gradients through previous steps
    x.requires_grad = True  # ensure loss is differentiable wrt x
    loss = my_loss_function(x)
    x = newton_step(loss, x, trust_radius=1.0)
# the final x is still differentiable
```

[1] Yuan, Ya-xiang. Iciam. Vol. 99. 2000. "A review of trust region algorithms for optimization." ftp://ftp.cc.ac.cn/pub/yyx/papers/p995.pdf

[2] Christianson, Bruce. Optimization Methods and Software 3.4 (1994) "Reverse accumulation and attractive fixed points." http://uhra.herts.ac.uk/bitstream/handle/2299/4338/903839.pdf

Parameters

- **loss** (*torch.Tensor*) – A scalar function of `x` to be minimized.
- **x** (*torch.Tensor*) – A dependent variable of shape `(N, D)` where `N` is the batch size and `D` is a small number.
- **trust_radius** (*float*) – An optional trust region trust_radius. The updated value `mode` of this function will be within `trust_radius` of the input `x`.

Returns A pair `(mode, cov)` where `mode` is an updated tensor of the same shape as the original value `x`, and `cov` is an esitmate of the covariance DxD matrix with `cov.shape == x.shape[:-1] + (D,D)`.

Return type tuple

newton_step_1d(*loss*, *x*, *trust_radius=None*)

Performs a Newton update step to minimize loss on a batch of 1-dimensional variables, optionally regularizing to constrain to a trust region.

See *newton_step()* for details.

Parameters

- **loss** (*torch.Tensor*) – A scalar function of `x` to be minimized.
- **x** (*torch.Tensor*) – A dependent variable with rightmost size of 1.
- **trust_radius** (*float*) – An optional trust region trust_radius. The updated value `mode` of this function will be within `trust_radius` of the input `x`.

Returns A pair `(mode, cov)` where `mode` is an updated tensor of the same shape as the original value `x`, and `cov` is an esitmate of the covariance 1x1 matrix with `cov.shape == x.shape[:-1] + (1,1)`.

Return type tuple

newton_step_2d(*loss*, *x*, *trust_radius=None*)

Performs a Newton update step to minimize loss on a batch of 2-dimensional variables, optionally regularizing to constrain to a trust region.

See *newton_step()* for details.

Parameters

- **loss** (*torch.Tensor*) – A scalar function of `x` to be minimized.
- **x** (*torch.Tensor*) – A dependent variable with rightmost size of 2.
- **trust_radius** (*float*) – An optional trust region trust_radius. The updated value `mode` of this function will be within `trust_radius` of the input `x`.

Returns A pair `(mode, cov)` where `mode` is an updated tensor of the same shape as the original value `x`, and `cov` is an esitmate of the covariance 2x2 matrix with `cov.shape == x.shape[:-1] + (2,2)`.

Return type tuple

newton_step_3d(*loss*, *x*, *trust_radius=None*)

Performs a Newton update step to minimize loss on a batch of 3-dimensional variables, optionally regularizing to constrain to a trust region.

See *newton_step()* for details.

Parameters

- **loss** (*torch.Tensor*) – A scalar function of `x` to be minimized.
- **x** (*torch.Tensor*) – A dependent variable with rightmost size of 2.
- **trust_radius** (*float*) – An optional trust region trust_radius. The updated value `mode` of this function will be within `trust_radius` of the input `x`.

Returns A pair `(mode, cov)` where `mode` is an updated tensor of the same shape as the original value `x`, and `cov` is an esitmate of the covariance 3x3 matrix with `cov.shape == x.shape[:-1] + (3,3)`.

Return type tuple

CHAPTER 11

Automatic Guide Generation

The *pyro.contrib.autoguide* module provides algorithms to automatically generate guides from simple models, for use in *SVI*. For example to generate a mean field Gaussian guide:

```
def model():
    ...

guide = AutoDiagonalNormal(model)  # a mean field guide
svi = SVI(model, guide, Adam({'lr': 1e-3}), Trace_ELBO())
```

Automatic guides can also be combined using *pyro.poutine.block()* and *AutoGuideList*.

11.1 AutoGuide

class AutoGuide(*model*, *prefix='auto'*)

Bases: object

Base class for automatic guides.

Derived classes must implement the *__call__()* method.

Auto guides can be used individually or combined in an *AutoGuideList* object.

Parameters

- **model** (*callable*) – a pyro model
- **prefix** (*str*) – a prefix that will be prefixed to all param internal sites

__call__(**args*, ***kwargs*)

A guide with the same `*args, **kwargs` as the base `model`.

Returns A dict mapping sample site name to sampled value.

Return type dict

median(**args*, ***kwargs*)
Returns the posterior median value of each latent variable.

Returns A dict mapping sample site name to median tensor.

Return type dict

sample_latent(***kwargs*)
Samples an encoded latent given the same `*args, **kwargs` as the base `model`.

11.2 AutoGuideList

class AutoGuideList(*model*, *prefix='auto'*)
Bases: *pyro.contrib.autoguide.AutoGuide*

Container class to combine multiple automatic guides.

Example usage:

```
guide = AutoGuideList(my_model)
guide.add(AutoDiagonalNormal(poutine.block(model, hide=["assignment"])))
guide.add(AutoDiscreteParallel(poutine.block(model, expose=["assignment"])))
svi = SVI(model, guide, optim, Trace_ELBO())
```

Parameters

- **model** (*callable*) – a Pyro model
- **prefix** (*str*) – a prefix that will be prefixed to all param internal sites

__call__(**args*, ***kwargs*)
A composite guide with the same `*args, **kwargs` as the base `model`.

Returns A dict mapping sample site name to sampled value.

Return type dict

add(*part*)
Add an automatic guide for part of the model. The guide should have been created by blocking the model to restrict to a subset of sample sites. No two parts should operate on any one sample site.

Parameters **part** (*AutoGuide or callable*) – a partial guide to add

median(**args*, ***kwargs*)
Returns the posterior median value of each latent variable.

Returns A dict mapping sample site name to median tensor.

Return type dict

11.3 AutoCallable

class AutoCallable(*model*, *guide*, *median=<function <lambda>>*)
Bases: *pyro.contrib.autoguide.AutoGuide*

AutoGuide wrapper for simple callable guides.

This is used internally for composing autoguides with custom user-defined guides that are simple callables, e.g.:

```
def my_local_guide(*args, **kwargs):
    ...

guide = AutoGuideList(model)
guide.add(AutoDelta(poutine.block(model, expose=['my_global_param'])))
guide.add(my_local_guide)  # automatically wrapped in an AutoCallable
```

To specify a median callable, you can instead:

```
def my_local_median(*args, **kwargs)
    ...

guide.add(AutoCallable(model, my_local_guide, my_local_median))
```

For more complex guides that need e.g. access to iaranges, users should instead subclass `AutoGuide`.

Parameters

- **model** (*callable*) – a Pyro model
- **guide** (*callable*) – a Pyro guide (typically over only part of the model)
- **median** (*callable*) – an optional callable returning a dict mapping sample site name to computed median tensor.

__call__(**args*, ***kwargs*)

A guide with the same `*args, **kwargs` as the base `model`.

Returns A dict mapping sample site name to sampled value.

Return type dict

11.4 AutoDelta

class AutoDelta(*model*, *prefix='auto'*)

Bases: *pyro.contrib.autoguide.AutoGuide*

This implementation of *AutoGuide* uses Delta distributions to construct a MAP guide over the entire latent space. The guide does not depend on the model's `*args, **kwargs`.

Usage:

```
guide = AutoDelta(model)
svi = SVI(model, guide, ...)
```

By default latent variables are randomly initialized by the model. To change this default behavior the user should call *pyro.param()* before beginning inference, with `"auto_"` prefixed to the targetd sample site names e.g. for sample sites named "level" and "concentration", initialize via:

```
pyro.param("auto_level", torch.tensor([-1., 0., 1.]))
pyro.param("auto_concentration", torch.ones(k),
           constraint=constraints.positive)
```

__call__(**args*, ***kwargs*)

An automatic guide with the same `*args, **kwargs` as the base `model`.

Returns A dict mapping sample site name to sampled value.

Return type dict

median (**args*, ***kwargs*)
Returns the posterior median value of each latent variable.

Returns A dict mapping sample site name to median tensor.

Return type dict

11.5 AutoContinuous

class AutoContinuous (*model*, *prefix='auto'*)
Bases: *pyro.contrib.autoguide.AutoGuide*

Base class for implementations of continuous-valued Automatic Differentiation Variational Inference [1].

Each derived class implements its own *sample_latent()* method.

Assumes model structure and latent dimension are fixed, and all latent variables are continuous.

Parameters **model** (*callable*) – a Pyro model

Reference:

[1] 'Automatic Differentiation Variational Inference', Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, David M. Blei

__call__ (**args*, ***kwargs*)
An automatic guide with the same `*args, **kwargs` as the base `model`.

Returns A dict mapping sample site name to sampled value.

Return type dict

median (**args*, ***kwargs*)
Returns the posterior median value of each latent variable.

Returns A dict mapping sample site name to median tensor.

Return type dict

quantiles (*quantiles*, **args*, ***kwargs*)
Returns posterior quantiles each latent variable. Example:

```
print(guide.quantiles([0.05, 0.5, 0.95]))
```

Parameters **quantiles** (*torch.Tensor or list*) – A list of requested quantiles between 0 and 1.

Returns A dict mapping sample site name to a list of quantile values.

Return type dict

sample_latent (**args*, ***kwargs*)
Samples an encoded latent given the same `*args, **kwargs` as the base `model`.

11.6 AutoMultivariateNormal

class AutoMultivariateNormal (*model*, *prefix='auto'*)
Bases: *pyro.contrib.autoguide.AutoContinuous*

This implementation of *AutoContinuous* uses a Cholesky factorization of a Multivariate Normal distribution to construct a guide over the entire latent space. The guide does not depend on the model's `*args, **kwargs`.

Usage:

```
guide = AutoMultivariateNormal(model)
svi = SVI(model, guide, ...)
```

By default the mean vector is initialized to zero and the Cholesky factor is initialized to the identity. To change this default behavior the user should call *pyro.param()* before beginning inference, e.g.:

```
latent_dim = 10
pyro.param("auto_loc", torch.randn(latent_dim))
pyro.param("auto_scale_tril", torch.tril(torch.rand(latent_dim)),
           constraint=constraints.lower_cholesky)
```

sample_latent(**args*, ***kwargs*)
Samples the (single) multivariate normal latent used in the auto guide.

11.7 AutoDiagonalNormal

class AutoDiagonalNormal(*model*, *prefix='auto'*)
Bases: *pyro.contrib.autoguide.AutoContinuous*

This implementation of *AutoContinuous* uses a Normal distribution with a diagonal covariance matrix to construct a guide over the entire latent space. The guide does not depend on the model's `*args, **kwargs`.

Usage:

```
guide = AutoDiagonalNormal(model)
svi = SVI(model, guide, ...)
```

By default the mean vector is initialized to zero and the scale is initialized to the identity. To change this default behavior the user should call *pyro.param()* before beginning inference, e.g.:

```
latent_dim = 10
pyro.param("auto_loc", torch.randn(latent_dim))
pyro.param("auto_scale", torch.ones(latent_dim),
           constraint=constraints.positive)
```

sample_latent(**args*, ***kwargs*)
Samples the (single) diagnoal normal latent used in the auto guide.

11.8 AutoLowRankMultivariateNormal

class AutoLowRankMultivariateNormal(*model*, *prefix='auto'*, *rank=1*)
Bases: *pyro.contrib.autoguide.AutoContinuous*

This implementation of *AutoContinuous* uses a low rank plus diagonal Multivariate Normal distribution to construct a guide over the entire latent space. The guide does not depend on the model's `*args, **kwargs`.

Usage:

```
guide = AutoLowRankMultivariateNormal(model, rank=10)
svi = SVI(model, guide, ...)
```

By default the `D_term` is initialized to 1/2 and the `W_term` is intialized randomly such that `W_term.matmul(W_term.t())` is half the identity matrix. To change this default behavior the user should call *pyro.param()* before beginning inference, e.g.:

```
latent_dim = 10
pyro.param("auto_loc", torch.randn(latent_dim))
pyro.param("auto_W_term", torch.randn(latent_dim, rank)))
pyro.param("auto_D_term", torch.randn(latent_dim).exp()),
           constraint=constraints.positive)
```

Parameters

- **model** (*callable*) – a generative model
- **rank** (*int*) – the rank of the low-rank part of the covariance matrix
- **prefix** (*str*) – a prefix that will be prefixed to all param internal sites

sample_latent (**args*, ***kwargs*)
Samples the (single) multivariate normal latent used in the auto guide.

11.9 AutoIAFNormal

class AutoIAFNormal (*model*, *hidden_dim=None*, *sigmoid_bias=2.0*, *prefix='auto'*)
Bases: *pyro.contrib.autoguide.AutoContinuous*

This implementation of *AutoContinuous* uses a Diagonal Normal distribution transformed via a `InverseAutoregressiveFlow` to construct a guide over the entire latent space. The guide does not depend on the model's `*args, **kwargs`.

Usage:

```
guide = AutoIAFNormal(model, hidden_dim=latent_dim)
svi = SVI(model, guide, ...)
```

Parameters

- **model** (*callable*) – a generative model
- **hidden_dim** (*int*) – number of hidden dimensions in the IAF
- **sigmoid_bias** (*float*) – sigmoid bias in the IAF. Defaults to `2.0`
- **prefix** (*str*) – a prefix that will be prefixed to all param internal sites

sample_latent (**args*, ***kwargs*)
Samples an encoded latent given the same `*args, **kwargs` as the base `model`.

11.10 AutoDiscreteParallel

class AutoDiscreteParallel (*model*, *prefix='auto'*)
Bases: *pyro.contrib.autoguide.AutoGuide*

A discrete mean-field guide that learns a latent discrete distribution for each discrete site in the model.

__call__(**args*, ***kwargs*)

An automatic guide with the same `*args, **kwargs` as the base `model`.

Returns A dict mapping sample site name to sampled value.

Return type dict

CHAPTER 12

Automatic Name Generation

The *pyro.contrib.autoname* module provides tools for automatically generating unique, semantically meaningful names for sample sites.

scope (*fn=None*, *prefix=None*, *inner=None*)

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **prefix** – a string to prepend to sample names (optional if `fn` is provided)
- **inner** – switch to determine where duplicate name counters appear

Returns `fn` decorated with a *ScopeMessenger*

`scope` prepends a prefix followed by a `/` to the name at a Pyro sample site. It works much like TensorFlow's `name_scope` and `variable_scope`, and can be used as a context manager, a decorator, or a higher-order function.

`scope` is very useful for aligning compositional models with guides or data.

Example:

```
>>> @scope(prefix="a")
... def model():
...     return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
```

Example:

```
>>> def model():
...     with scope(prefix="a"):
...         return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
```

Scopes compose as expected, with outer scopes appearing before inner scopes in names:

```
>>> @scope(prefix="b")
... def model():
...     with scope(prefix="a"):
...         return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "b/a/x" in poutine.trace(model).get_trace()
```

When used as a decorator or higher-order function, `scope` will use the name of the input function as the prefix if no user-specified prefix is provided.

Example:

```
>>> @scope
... def model():
...     return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "model/x" in poutine.trace(model).get_trace()
```

12.1 Named Data Structures

The `pyro.contrib.named` module is a thin syntactic layer on top of Pyro. It allows Pyro models to be written to look like programs with operating on Python data structures like `latent.x.sample_(...)`, rather than programs with string-labeled statements like `x = pyro.sample("x", ...)`.

This module provides three container data structures `named.Object`, `named.List`, and `named.Dict`. These data structures are intended to be nested in each other. Together they track the address of each piece of data in each data structure, so that this address can be used as a Pyro site. For example:

```
>>> state = named.Object("state")
>>> print(str(state))
state

>>> z = state.x.y.z  # z is just a placeholder.
>>> print(str(z))
state.x.y.z

>>> state.xs = named.List()  # Create a contained list.
>>> x0 = state.xs.add()
>>> print(str(x0))
state.xs[0]

>>> state.ys = named.Dict()
>>> foo = state.ys['foo']
>>> print(str(foo))
state.ys['foo']
```

These addresses can now be used inside `sample`, `observe` and `param` statements. These named data structures even provide in-place methods that alias Pyro statements. For example:

```
>>> state = named.Object("state")
>>> loc = state.loc.param_(torch.zeros(1, requires_grad=True))
>>> scale = state.scale.param_(torch.ones(1, requires_grad=True))
>>> z = state.z.sample_(dist.Normal(loc, scale))
>>> obs = state.x.sample_(dist.Normal(loc, scale), obs=z)
```

For deeper examples of how these can be used in model code, see the Tree Data and Mixture examples.

Authors: Fritz Obermeyer, Alexander Rush

class Object(*name*)

Bases: object

Object to hold immutable latent state.

This object can serve either as a container for nested latent state or as a placeholder to be replaced by a tensor via a named.sample, named.observe, or named.param statement. When used as a placeholder, Object objects take the place of strings in normal pyro.sample statements.

Parameters **name** (*str*) – The name of the object.

Example:

```
state = named.Object("state")
state.x = 0
state.ys = named.List()
state.zs = named.Dict()
state.a.b.c.d.e.f.g = 0  # Creates a chain of named.Objects.
```

Warning: This data structure is write-once: data may be added but may not be mutated or removed. Trying to mutate this data structure may result in silent errors.

sample_(*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
- **infer** (*dict*) – Optional dictionary of inference parameters specified in kwargs. See inference documentation for details.

Returns sample

param_(**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

class List(*name=None*)

Bases: list

List-like object to hold immutable latent state.

This must either be given a name when constructed:

```
latent = named.List("root")
```

or must be immediately stored in a `named.Object`:

```
latent = named.Object("root")
latent.xs = named.List()  # Must be bound to a Object before use.
```

Warning: This data structure is write-once: data may be added but may not be mutated or removed. Trying to mutate this data structure may result in silent errors.

add()
Append one new named.Object.

Returns a new latent object at the end

Return type *named.Object*

class Dict(*name=None*)
Bases: dict

Dict-like object to hold immutable latent state.

This must either be given a name when constructed:

```
latent = named.Dict("root")
```

or must be immediately stored in a named.Object:

```
latent = named.Object("root")
latent.xs = named.Dict()  # Must be bound to a Object before use.
```

Warning: This data structure is write-once: data may be added but may not be mutated or removed. Trying to mutate this data structure may result in silent errors.

12.2 Scoping

pyro.contrib.autoname.scoping contains the implementation of *pyro.contrib.autoname.scope()*, a tool for automatically appending a semantically meaningful prefix to names of sample sites.

class ScopeMessenger(*prefix=None*, *inner=None*)
Bases: *pyro.poutine.messenger.Messenger*

ScopeMessenger is the implementation of *pyro.contrib.autoname.scope()*

scope(*fn=None*, *prefix=None*, *inner=None*)

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **prefix** – a string to prepend to sample names (optional if fn is provided)
- **inner** – switch to determine where duplicate name counters appear

Returns fn decorated with a *ScopeMessenger*

scope prepends a prefix followed by a / to the name at a Pyro sample site. It works much like TensorFlow's name_scope and variable_scope, and can be used as a context manager, a decorator, or a higher-order function.

`scope` is very useful for aligning compositional models with guides or data.

Example:

```
>>> @scope(prefix="a")
... def model():
...     return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
```

Example:

```
>>> def model():
...     with scope(prefix="a"):
...         return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
```

Scopes compose as expected, with outer scopes appearing before inner scopes in names:

```
>>> @scope(prefix="b")
... def model():
...     with scope(prefix="a"):
...         return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "b/a/x" in poutine.trace(model).get_trace()
```

When used as a decorator or higher-order function, `scope` will use the name of the input function as the prefix if no user-specified prefix is provided.

Example:

```
>>> @scope
... def model():
...     return pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "model/x" in poutine.trace(model).get_trace()
```

CHAPTER 13

Gaussian Processes

See the Gaussian Processes tutorial for an introduction.

13.1 Models

13.1.1 GPModel

class GPModel(*X*, *y*, *kernel*, *mean_function=None*, *jitter=1e-06*, *name=None*)

Bases: *pyro.contrib.gp.util.Parameterized*

Base class for Gaussian Process models.

The core of a Gaussian Process is a covariance function k which governs the similarity between input points. Given k, we can establish a distribution over functions f by a multivarite normal distribution

$$p(f(X)) = \mathcal{N}(0, k(X, X)),$$

where X is any set of input points and $k(X, X)$ is a covariance matrix whose entries are outputs $k(x, z)$ of k over input pairs (x, z). This distribution is usually denoted by

$$f \sim \mathcal{GP}(0, k).$$

Note: Generally, beside a covariance matrix k, a Gaussian Process can also be specified by a mean function m (which is a zero-value function by default). In that case, its distribution will be

$$p(f(X)) = \mathcal{N}(m(X), k(X, X)).$$

Gaussian Process models are *Parameterized* subclasses. So its parameters can be learned, set priors, or fixed by using corresponding methods from *Parameterized*. A typical way to define a Gaussian Process model is

```
>>> X = torch.tensor([[1., 5, 3], [4, 3, 7]])
>>> y = torch.tensor([2., 1])
>>> kernel = gp.kernels.RBF(input_dim=3)
>>> kernel.set_prior("variance", dist.Uniform(torch.tensor(0.5), torch.tensor(1.
↪5)))
>>> kernel.set_prior("lengthscale", dist.Uniform(torch.tensor(1.0), torch.
↪tensor(3.0)))
>>> gpr = gp.models.GPRegression(X, y, kernel)
```

There are two ways to train a Gaussian Process model:

- Using an MCMC algorithm (in module `pyro.infer.mcmc`) on *`model()`* to get posterior samples for the Gaussian Process's parameters. For example:

```
>>> hmc_kernel = HMC(gpr.model)
>>> mcmc_run = MCMC(hmc_kernel, num_samples=10)
>>> posterior_ls_trace = []  # store lengthscale trace
>>> ls_name = param_with_module_name(gpr.kernel.name, "lengthscale")
>>> for trace, _ in mcmc_run._traces():
...     posterior_ls_trace.append(trace.nodes[ls_name]["value"])
```

- Using a variational inference (e.g. *`SVI`*) on the pair *`model()`*, *`guide()`* as in SVI tutorial:

```
>>> optimizer = pyro.optim.Adam({"lr": 0.01})
>>> svi = SVI(gpr.model, gpr.guide, optimizer, loss=Trace_ELBO())
>>> for i in range(1000):
...     svi.step()
```

To give a prediction on new dataset, simply use *`forward()`* like any PyTorch `torch.nn.Module`:

```
>>> Xnew = torch.tensor([[2., 3, 1]])
>>> f_loc, f_cov = gpr(Xnew, full_cov=True)
```

Reference:

[1] *Gaussian Processes for Machine Learning*, Carl E. Rasmussen, Christopher K. I. Williams

Parameters

- **`X`** (*`torch.Tensor`*) – A input data for training. Its first dimension is the number of data points.
- **`y`** (*`torch.Tensor`*) – An output data for training. Its last dimension is the number of data points.
- **`kernel`** (*`Kernel`*) – A Pyro kernel object, which is the covariance function k.
- **`mean_function`** (*`callable`*) – An optional mean function m of this Gaussian process. By default, we use zero mean.
- **`jitter`** (*`float`*) – A small positive term which is added into the diagonal part of a covariance matrix to help stablize its Cholesky decomposition.
- **`name`** (*`str`*) – Name of this model.

`model``()`

A "model" stochastic function. If `self.y` is `None`, this method returns mean and variance of the Gaussian Process prior.

guide()
A "guide" stochastic function to be used in variational inference methods. It also gives posterior information to the method *forward()* for prediction.

forward(*Xnew*, *full_cov=False*)
Computes the mean and covariance matrix (or variance) of Gaussian Process posterior on a test input data X_{new}:

$$p(f^* \mid X_{new}, X, y, k, \theta),$$

where θ are parameters of this model.

Note: Model's parameters θ together with kernel's parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- **Xnew** (*torch.Tensor*) – A input data for testing. Note that `Xnew.shape[1:]` must be the same as `X.shape[1:]`.
- **full_cov** (*bool*) – A flag to decide if we want to predict full covariance matrix or just variance.

Returns loc and covariance matrix (or variance) of $p(f^*(X_{new}))$

Return type tuple(torch.Tensor, torch.Tensor)

set_data(*X*, *y=None*)
Sets data for Gaussian Process models.

Some examples to utilize this method are:

- Batch training on a sparse variational model:

```
>>> Xu = torch.tensor([[1., 0, 2]])  # inducing input
>>> likelihood = gp.likelihoods.Gaussian()
>>> vsgp = gp.models.VariationalSparseGP(X, y, kernel, Xu, likelihood)
>>> svi = SVI(vsgp.model, vsgp.guide, optimizer, Trace_ELBO())
>>> batched_X, batched_y = X.split(split_size=10), y.split(split_size=10)
>>> for Xi, yi in zip(batched_X, batched_y):
...     vsgp.set_data(Xi, yi)
...     svi.step()
```

- Making a two-layer Gaussian Process stochastic function:

```
>>> gpr1 = gp.models.GPRegression(X, None, kernel, name="GPR1")
>>> Z, _ = gpr1.model()
>>> gpr2 = gp.models.GPRegression(Z, y, kernel, name="GPR2")
>>> def two_layer_model():
...     Z, _ = gpr1.model()
...     gpr2.set_data(Z, y)
...     return gpr2.model()
```

References:

[1] *Scalable Variational Gaussian Process Classification*, James Hensman, Alexander G. de G. Matthews, Zoubin Ghahramani

[2] *Deep Gaussian Processes*, Andreas C. Damianou, Neil D. Lawrence

Parameters

- **X** (*torch.Tensor*) – A input data for training. Its first dimension is the number of data points.
- **y** (*torch.Tensor*) – An output data for training. Its last dimension is the number of data points.

optimize(*optimizer=None*, *loss=None*, *num_steps=1000*)

A convenient method to optimize parameters for the Gaussian Process model using *SVI*.

Parameters

- **optimizer** (PyroOptim) – A Pyro optimizer.
- **loss** (ELBO) – A Pyro loss instance.
- **num_steps** (*int*) – Number of steps to run SVI.

Returns a list of losses during the training procedure

Return type list

13.1.2 GPRegression

class GPRegression(*X*, *y*, *kernel*, *noise=None*, *mean_function=None*, *jitter=1e-06*, *name='GPR'*)

Bases: *pyro.contrib.gp.models.model.GPModel*

Gaussian Process Regression model.

The core of a Gaussian Process is a covariance function k which governs the similarity between input points. Given k, we can establish a distribution over functions f by a multivarite normal distribution

$$p(f(X)) = \mathcal{N}(0, k(X, X)),$$

where X is any set of input points and $k(X, X)$ is a covariance matrix whose entries are outputs $k(x, z)$ of k over input pairs (x, z). This distribution is usually denoted by

$$f \sim \mathcal{GP}(0, k).$$

Note: Generally, beside a covariance matrix k, a Gaussian Process can also be specified by a mean function m (which is a zero-value function by default). In that case, its distribution will be

$$p(f(X)) = \mathcal{N}(m(X), k(X, X)).$$

Given inputs X and their noisy observations y, the Gaussian Process Regression model takes the form

$$f \sim \mathcal{GP}(0, k(X, X)),$$
$$y \sim f + \epsilon,$$

where ϵ is Gaussian noise.

Note: This model has $\mathcal{O}(N^3)$ complexity for training, $\mathcal{O}(N^3)$ complexity for testing. Here, N is the number of train inputs.

Reference:

[1] *Gaussian Processes for Machine Learning*, Carl E. Rasmussen, Christopher K. I. Williams

Parameters

- **X** (*torch.Tensor*) – A input data for training. Its first dimension is the number of data points.
- **y** (*torch.Tensor*) – An output data for training. Its last dimension is the number of data points.
- **kernel** (*Kernel*) – A Pyro kernel object, which is the covariance function k.
- **noise** (*torch.Tensor*) – Variance of Gaussian noise of this model.
- **mean_function** (*callable*) – An optional mean function m of this Gaussian process. By default, we use zero mean.
- **jitter** (*float*) – A small positive term which is added into the diagonal part of a covariance matrix to help stablize its Cholesky decomposition.
- **name** (*str*) – Name of this model.

model()

A "model" stochastic function. If `self.y` is `None`, this method returns mean and variance of the Gaussian Process prior.

guide()

A "guide" stochastic function to be used in variational inference methods. It also gives posterior information to the method *forward()* for prediction.

forward(*Xnew*, *full_cov=False*, *noiseless=True*)

Computes the mean and covariance matrix (or variance) of Gaussian Process posterior on a test input data X_{new}:

$$p(f^* \mid X_{new}, X, y, k, \epsilon) = \mathcal{N}(loc, cov).$$

Note: The noise parameter `noise` (ϵ) together with kernel's parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- **Xnew** (*torch.Tensor*) – A input data for testing. Note that `Xnew.shape[1:]` must be the same as `self.X.shape[1:]`.
- **full_cov** (*bool*) – A flag to decide if we want to predict full covariance matrix or just variance.
- **noiseless** (*bool*) – A flag to decide if we want to include noise in the prediction output or not.

Returns loc and covariance matrix (or variance) of $p(f^*(X_{new}))$

Return type tuple(torch.Tensor, torch.Tensor)

13.1.3 SparseGPRegression

class SparseGPRegression(*X*, *y*, *kernel*, *Xu*, *noise=None*, *mean_function=None*, *approx=None*, *jitter=1e-06*, *name='SGPR'*)

Bases: *pyro.contrib.gp.models.model.GPModel*

Sparse Gaussian Process Regression model.

In `GPRegression` model, when the number of input data X is large, the covariance matrix $k(X, X)$ will require a lot of computational steps to compute its inverse (for log likelihood and for prediction). By introducing an additional inducing-input parameter X_u, we can reduce computational cost by approximate $k(X, X)$ by a low-rank Nymström approximation Q (see reference [1]), where

$$Q = k(X, X_u)k(X, X)^{-1}k(X_u, X).$$

Given inputs X, their noisy observations y, and the inducing-input parameters X_u, the model takes the form:

$$\begin{aligned}
&u \sim \mathcal{GP}(0, k(X_u, X_u)),\\
&f \sim q(f \mid X, X_u) = \mathbb{E}_{p(u)} q(f \mid X, X_u, u),\\
&y \sim f + \epsilon,
\end{aligned}$$

where ϵ is Gaussian noise and the conditional distribution $q(f \mid X, X_u, u)$ is an approximation of

$$p(f \mid X, X_u, u) = \mathcal{N}(m, k(X, X) - Q),$$

whose terms m and $k(X, X) - Q$ is derived from the joint multivariate normal distribution:

$$[f, u] \sim \mathcal{GP}(0, k([X, X_u], [X, X_u])).$$

This class implements three approximation methods:

- Deterministic Training Conditional (DTC):

  $$q(f \mid X, X_u, u) = \mathcal{N}(m, 0),$$

  which in turns will imply

  $$f \sim \mathcal{N}(0, Q).$$

- Fully Independent Training Conditional (FITC):

  $$q(f \mid X, X_u, u) = \mathcal{N}(m, diag(k(X, X) - Q)),$$

  which in turns will correct the diagonal part of the approximation in DTC:

  $$f \sim \mathcal{N}(0, Q + diag(k(X, X) - Q)).$$

- Variational Free Energy (VFE), which is similar to DTC but has an additional *trace_term* in the model's log likelihood. This additional term makes "VFE" equivalent to the variational approach in `SparseVariationalGP` (see reference [2]).

Note: This model has $\mathcal{O}(NM^2)$ complexity for training, $\mathcal{O}(NM^2)$ complexity for testing. Here, N is the number of train inputs, M is the number of inducing inputs.

References:

[1] *A Unifying View of Sparse Approximate Gaussian Process Regression*, Joaquin Quiñonero-Candela, Carl E. Rasmussen

[2] *Variational learning of inducing variables in sparse Gaussian processes*, Michalis Titsias

Parameters

- **X** (*torch.Tensor*) – A input data for training. Its first dimension is the number of data points.
- **y** (*torch.Tensor*) – An output data for training. Its last dimension is the number of data points.
- **kernel** (*Kernel*) – A Pyro kernel object, which is the covariance function k.
- **Xu** (*torch.Tensor*) – Initial values for inducing points, which are parameters of our model.
- **noise** (*torch.Tensor*) – Variance of Gaussian noise of this model.
- **mean_function** (*callable*) – An optional mean function m of this Gaussian process. By default, we use zero mean.
- **approx** (*str*) – One of approximation methods: "DTC", "FITC", and "VFE" (default).
- **jitter** (*float*) – A small positive term which is added into the diagonal part of a covariance matrix to help stablize its Cholesky decomposition.
- **name** (*str*) – Name of this model.

model()

A "model" stochastic function. If `self.y` is `None`, this method returns mean and variance of the Gaussian Process prior.

guide()

A "guide" stochastic function to be used in variational inference methods. It also gives posterior information to the method *forward()* for prediction.

forward(*Xnew*, *full_cov=False*, *noiseless=True*)

Computes the mean and covariance matrix (or variance) of Gaussian Process posterior on a test input data X_{new}:

$$p(f^* \mid X_{new}, X, y, k, X_u, \epsilon) = \mathcal{N}(loc, cov).$$

Note: The noise parameter `noise` (ϵ), the inducing-point parameter `Xu`, together with kernel's parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- **Xnew** (*torch.Tensor*) – A input data for testing. Note that `Xnew.shape[1:]` must be the same as `self.X.shape[1:]`.
- **full_cov** (*bool*) – A flag to decide if we want to predict full covariance matrix or just variance.
- **noiseless** (*bool*) – A flag to decide if we want to include noise in the prediction output or not.

Returns loc and covariance matrix (or variance) of $p(f^*(X_{new}))$

Return type tuple(torch.Tensor, torch.Tensor)

13.1.4 VariationalGP

class VariationalGP(*X*, *y*, *kernel*, *likelihood*, *mean_function=None*, *latent_shape=None*, *whiten=False*, *jitter=1e-06*, *name='VGP'*)

Bases: *pyro.contrib.gp.models.model.GPModel*

Variational Gaussian Process model.

This model deals with both Gaussian and non-Gaussian likelihoods. Given inputs X and their noisy observations y, the model takes the form

$$f \sim \mathcal{GP}(0, k(X, X)),$$
$$y \sim p(y) = p(y \mid f)p(f),$$

where $p(y \mid f)$ is the likelihood.

We will use a variational approach in this model by approximating $q(f)$ to the posterior $p(f \mid y)$. Precisely, $q(f)$ will be a multivariate normal distribution with two parameters `f_loc` and `f_scale_tril`, which will be learned during a variational inference process.

Note: This model can be seen as a special version of `SparseVariationalGP` model with $X_u = X$.

Note: This model has $\mathcal{O}(N^3)$ complexity for training, $\mathcal{O}(N^3)$ complexity for testing. Here, N is the number of train inputs. Size of variational parameters is $\mathcal{O}(N^2)$.

Parameters

- **X** (*torch.Tensor*) – A input data for training. Its first dimension is the number of data points.
- **y** (*torch.Tensor*) – An output data for training. Its last dimension is the number of data points.
- **kernel** (*Kernel*) – A Pyro kernel object, which is the covariance function k.
- **Likelihood likelihood** (*likelihood*) – A likelihood object.
- **mean_function** (*callable*) – An optional mean function m of this Gaussian process. By default, we use zero mean.
- **latent_shape** (*torch.Size*) – Shape for latent processes (*batch_shape* of $q(f)$). By default, it equals to output batch shape `y.shape[:-1]`. For the multi-class classification problems, `latent_shape[-1]` should corresponse to the number of classes.
- **whiten** (*bool*) – A flag to tell if variational parameters `f_loc` and `f_scale_tril` are transformed by the inverse of `Lff`, where `Lff` is the lower triangular decomposition of $kernel(X, X)$. Enable this flag will help optimization.
- **jitter** (*float*) – A small positive term which is added into the diagonal part of a covariance matrix to help stablize its Cholesky decomposition.
- **name** (*str*) – Name of this model.

model()

A "model" stochastic function. If `self.y` is `None`, this method returns mean and variance of the Gaussian Process prior.

guide()
A "guide" stochastic function to be used in variational inference methods. It also gives posterior information to the method `forward()` for prediction.

forward(*Xnew*, *full_cov=False*)
Computes the mean and covariance matrix (or variance) of Gaussian Process posterior on a test input data X_{new}:

$$p(f^* \mid X_{new}, X, y, k, f_{loc}, f_{scale\_tril}) = \mathcal{N}(loc, cov).$$

Note: Variational parameters `f_loc`, `f_scale_tril`, together with kernel's parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- **Xnew** (*torch.Tensor*) – A input data for testing. Note that `Xnew.shape[1:]` must be the same as `self.X.shape[1:]`.
- **full_cov** (*bool*) – A flag to decide if we want to predict full covariance matrix or just variance.

Returns loc and covariance matrix (or variance) of $p(f^*(X_{new}))$

Return type tuple(torch.Tensor, torch.Tensor)

13.1.5 VariationalSparseGP

class VariationalSparseGP(*X*, *y*, *kernel*, *Xu*, *likelihood*, *mean_function=None*, *latent_shape=None*, *num_data=None*, *whiten=False*, *jitter=1e-06*, *name='SVGP'*)
Bases: *pyro.contrib.gp.models.model.GPModel*

Variational Sparse Gaussian Process model.

In *VariationalGP* model, when the number of input data X is large, the covariance matrix $k(X, X)$ will require a lot of computational steps to compute its inverse (for log likelihood and for prediction). This model introduces an additional inducing-input parameter X_u to solve that problem. Given inputs X, their noisy observations y, and the inducing-input parameters X_u, the model takes the form:

$$[f, u] \sim \mathcal{GP}(0, k([X, X_u], [X, X_u])),$$
$$y \sim p(y) = p(y \mid f)p(f),$$

where $p(y \mid f)$ is the likelihood.

We will use a variational approach in this model by approximating $q(f, u)$ to the posterior $p(f, u \mid y)$. Precisely, $q(f) = p(f \mid u)q(u)$, where $q(u)$ is a multivariate normal distribution with two parameters `u_loc` and `u_scale_tril`, which will be learned during a variational inference process.

Note: This model can be learned using MCMC method as in reference [2]. See also *GPModel*.

Note: This model has $\mathcal{O}(NM^2)$ complexity for training, $\mathcal{O}(M^3)$ complexity for testing. Here, N is the number of train inputs, M is the number of inducing inputs. Size of variational parameters is $\mathcal{O}(M^2)$.

References:

[1] *Scalable variational Gaussian process classification*, James Hensman, Alexander G. de G. Matthews, Zoubin Ghahramani

[2] *MCMC for Variationally Sparse Gaussian Processes*, James Hensman, Alexander G. de G. Matthews, Maurizio Filippone, Zoubin Ghahramani

Parameters

- **X** (*torch.Tensor*) – A input data for training. Its first dimension is the number of data points.
- **y** (*torch.Tensor*) – An output data for training. Its last dimension is the number of data points.
- **kernel** (*Kernel*) – A Pyro kernel object, which is the covariance function k.
- **Xu** (*torch.Tensor*) – Initial values for inducing points, which are parameters of our model.
- **Likelihood likelihood** (*likelihood*) – A likelihood object.
- **mean_function** (*callable*) – An optional mean function m of this Gaussian process. By default, we use zero mean.
- **latent_shape** (*torch.Size*) – Shape for latent processes (*batch_shape* of $q(u)$). By default, it equals to output batch shape `y.shape[:-1]`. For the multi-class classification problems, `latent_shape[-1]` should corresponse to the number of classes.
- **num_data** (*int*) – The size of full training dataset. It is useful for training this model with mini-batch.
- **whiten** (*bool*) – A flag to tell if variational parameters `u_loc` and `u_scale_tril` are transformed by the inverse of `Luu`, where `Luu` is the lower triangular decomposition of $kernel(X_u, X_u)$. Enable this flag will help optimization.
- **jitter** (*float*) – A small positive term which is added into the diagonal part of a covariance matrix to help stablize its Cholesky decomposition.
- **name** (*str*) – Name of this model.

model()

A "model" stochastic function. If `self.y` is `None`, this method returns mean and variance of the Gaussian Process prior.

guide()

A "guide" stochastic function to be used in variational inference methods. It also gives posterior information to the method *forward()* for prediction.

forward(*Xnew*, *full_cov=False*)

Computes the mean and covariance matrix (or variance) of Gaussian Process posterior on a test input data X_{new}:

$$p(f^* \mid X_{new}, X, y, k, X_u, u_{loc}, u_{scale\_tril}) = \mathcal{N}(loc, cov).$$

Note: Variational parameters `u_loc`, `u_scale_tril`, the inducing-point parameter `Xu`, together with kernel's parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- **Xnew** (*torch.Tensor*) – A input data for testing. Note that `Xnew.shape[1:]` must be the same as `self.X.shape[1:]`.
- **full_cov** (*bool*) – A flag to decide if we want to predict full covariance matrix or just variance.

Returns loc and covariance matrix (or variance) of $p(f^*(X_{new}))$

Return type tuple(torch.Tensor, torch.Tensor)

class GPLVM(*base_model*, *name='GPLVM'*)

Bases: *pyro.contrib.gp.util.Parameterized*

Gaussian Process Latent Variable Model (GPLVM) model.

GPLVM is a Gaussian Process model with its train input data is a latent variable. This model is useful for dimensional reduction of high dimensional data. Assume the mapping from low dimensional latent variable to is a Gaussian Process instance. Then the high dimensional data will play the role of train output `y` and our target is to learn latent inputs which best explain `y`. For the purpose of dimensional reduction, latent inputs should have lower dimensions than `y`.

We follows reference [1] to put a unit Gaussian prior to the input and approximate its posterior by a multivariate normal distribution with two variational parameters: `X_loc` and `X_scale_tril`.

For example, we can do dimensional reduction on Iris dataset as follows:

```
>>> # With y as the 2D Iris data of shape 150x4 and we want to reduce its␣
↪dimension
>>> # to a tensor X of shape 150x2, we will use GPLVM.
```

```
>>> # First, define the initial values for X_loc parameter:
>>> X_loc = torch.zeros(150, 2)
>>> # Then, define a Gaussian Process model with input X_loc and output y:
>>> kernel = gp.kernels.RBF(input_dim=2, lengthscale=torch.ones(2))
>>> Xu = torch.zeros(20, 2)  # initial inducing inputs of sparse model
>>> gpmodel = gp.models.SparseGPRegression(X_loc, y, kernel, Xu)
>>> # Finally, wrap gpmodel by GPLVM, optimize, and get the "learned"␣
↪mean of X:
>>> gplvm = gp.models.GPLVM(gpmodel)
>>> gplvm.optimize()
>>> X = gplvm.get_param("X_loc")
```

Reference:

[1] Bayesian Gaussian Process Latent Variable Model Michalis K. Titsias, Neil D. Lawrence

Parameters

- **base_model** (*GPModel*) – A Pyro Gaussian Process model object. Note that `base_model.X` will be the initial value for the variational parameter `X_loc`.
- **name** (*str*) – Name of this model.

model()

guide()

forward(***kwargs*)

Forward method has the same signal as its `base_model`. Note that the train input data of `base_model` is sampled from GPLVM.

optimize(*optimizer=<pyro.optim.optim.PyroOptim object>*, *num_steps=1000*)

A convenient method to optimize parameters for GPLVM model using *SVI*.

Parameters

- **optimizer** (PyroOptim) – A Pyro optimizer.
- **num_steps** (*int*) – Number of steps to run SVI.

Returns a list of losses during the training procedure

Return type list

13.2 Kernels

13.2.1 Brownian

class Brownian(*input_dim*, *variance=None*, *active_dims=None*, *name='Brownian'*)

Bases: pyro.contrib.gp.kernels.kernel.Kernel

This kernel correponds to a two-sided Brownion motion (Wiener process):

$$k(x,z) = \begin{cases} \sigma^2 \min(|x|,|z|), & \text{if } x \cdot z \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

Note that the input dimension of this kernel must be 1.

Reference:

[1] *Theory and Statistical Applications of Stochastic Processes*, Yuliya Mishura, Georgiy Shevchenko

forward(*X*, *Z=None*, *diag=False*)

Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.2 Combination

class Combination(*kern0*, *kern1*, *name=None*)

Bases: pyro.contrib.gp.kernels.kernel.Kernel

Base class for kernels derived from a combination of kernels.

Parameters

- **kern0** (Kernel) – First kernel to combine.
- **kern1** (*Kernel or numbers.Number*) – Second kernel to combine.

13.2.3 Constant

class Constant(*input_dim*, *variance=None*, *active_dims=None*, *name='Constant'*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Implementation of Constant kernel:

$$k(x, z) = \sigma^2.$$

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.4 Coregionalize

class Coregionalize(*input_dim*, *rank=None*, *components=None*, *diagonal=None*, *active_dims=None*, *name='coregionalize'*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

A kernel for the linear model of coregionalization $k(x, z) = x^T(WW^T + D)z$ where W is an `input_dim`-by-`rank` matrix and typically `rank < input_dim`, and `D` is a diagonal matrix.

This generalizes the `Linear` kernel to multiple features with a low-rank-plus-diagonal weight matrix. The typical use case is for modeling correlations among outputs of a multi-output GP, where outputs are coded as distinct data points with one-hot coded features denoting which output each datapoint represents.

If only `rank` is specified, the kernel `(W W^T + D)` will be randomly initialized to a matrix with expected value the identity matrix.

References:

[1] Mauricio A. Alvarez, Lorenzo Rosasco, Neil D. Lawrence (2012) Kernels for Vector-Valued Functions: a Review

Parameters

- **input_dim** (*int*) – Number of feature dimensions of inputs.
- **rank** (*int*) – Optional rank. This is only used if `components` is unspecified. If neigher `rank` nor `components` is specified, then `rank` defaults to `input_dim`.
- **components** (*torch.Tensor*) – An optional `(input_dim, rank)` shaped matrix that maps features to `rank`-many components. If unspecified, this will be randomly initialized.
- **diagonal** (*torch.Tensor*) – An optional vector of length `input_dim`. If unspecified, this will be set to constant `0.5`.
- **active_dims** (*list*) – List of feature dimensions of the input which the kernel acts on.
- **name** (*str*) – Name of the kernel.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.5 Cosine

class Cosine(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*, *name='Cosine'*)
Bases: pyro.contrib.gp.kernels.isotropic.Isotropy

Implementation of Cosine kernel:

$$k(x,z) = \sigma^2 \cos\left(\frac{|x-z|}{l}\right).$$

Parameters **lengthscale** (*torch.Tensor*) – Length-scale parameter of this kernel.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.6 DotProduct

class DotProduct(*input_dim*, *variance=None*, *active_dims=None*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Base class for kernels which are functions of $x \cdot z$.

13.2.7 Exponent

class Exponent(*kern*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Transforming

Creates a new kernel according to

$$k_{new}(x,z) = \exp(k(x,z)).$$

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.8 Exponential

class Exponential(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*, *name='Exponential'*)
Bases: pyro.contrib.gp.kernels.isotropic.Isotropy

Implementation of Exponential kernel:

$$k(x, z) = \sigma^2 \exp\left(-\frac{|x-z|}{l}\right).$$

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.9 Isotropy

class Isotropy(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Base class for a family of isotropic covariance kernels which are functions of the distance $|x - z|/l$, where l is the length-scale parameter.

By default, the parameter `lengthscale` has size 1. To use the isotropic version (different lengthscale for each dimension), make sure that `lengthscale` has size equal to `input_dim`.

Parameters **lengthscale** (*torch.Tensor*) – Length-scale parameter of this kernel.

13.2.10 Kernel

class Kernel(*input_dim*, *active_dims=None*, *name=None*)
Bases: *pyro.contrib.gp.util.Parameterized*

Base class for kernels used in this Gaussian Process module.

Every inherited class should implement a *forward()* pass which takes inputs X, Z and returns their covariance matrix.

To construct a new kernel from the old ones, we can use methods *add()*, *mul()*, *exp()*, *warp()*, *vertical_scale()*.

References:

[1] *Gaussian Processes for Machine Learning*, Carl E. Rasmussen, Christopher K. I. Williams

Parameters

- **input_dim** (*int*) – Number of feature dimensions of inputs.
- **variance** (*torch.Tensor*) – Variance parameter of this kernel.
- **active_dims** (*list*) – List of feature dimensions of the input which the kernel acts on.
- **name** (*str*) – Name of the kernel.

forward(*X*, *Z=None*, *diag=False*)

Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

add(*other*, *name=None*)

Creates a new kernel from a sum/direct sum of this kernel object and `other`.

Parameters

- **other** (*Kernel or numbers.Number*) – A kernel to be combined with this kernel object.
- **name** (*str*) – An optional name for the derived kernel.

Returns a Sum kernel

Return type *Sum*

mul(*other*, *name=None*)

Creates a new kernel from a product/tensor product of this kernel object and `other`.

Parameters

- **other** (*Kernel or numbers.Number*) – A kernel to be combined with this kernel object.
- **name** (*str*) – An optional name for the derived kernel.

Returns a Product kernel

Return type *Product*

exp(*name=None*)

Creates a new kernel according to

$$k_{new}(x, z) = \exp(k(x, z)).$$

Parameters **name** (*str*) – An optional name for the derived kernel.

Returns an Exponent kernel

Return type *Exponent*

vertical_scale (*vscaling_fn*, *name=None*)

Creates a new kernel according to

$$k_{new}(x, z) = f(x)k(x, z)f(z),$$

where f is a function.

Parameters

- **vscaling_fn** (*callable*) – A vertical scaling function f.
- **name** (*str*) – An optional name for the derived kernel.

Returns a vertical scaled kernel

Return type *VerticalScaling*

warp (*iwarping_fn=None*, *owarping_coef=None*, *name=None*)

Creates a new kernel according to

$$k_{new}(x, z) = q(k(f(x), f(z))),$$

where f is an function and q is a polynomial with non-negative coefficients `owarping_coef`.

Parameters

- **iwarping_fn** (*callable*) – An input warping function f.
- **owarping_coef** (*list*) – A list of coefficients of the output warping polynomial. These coefficients must be non-negative.
- **name** (*str*) – An optional name for the derived kernel.

Returns a warped kernel

Return type *Warping*

get_subkernel (*name*)

Returns the subkernel corresponding to `name`.

Parameters **name** (*str*) – Name of the subkernel.

Returns A subkernel.

Return type *Kernel*

13.2.11 Linear

class Linear (*input_dim*, *variance=None*, *active_dims=None*, *name='Linear'*)

Bases: `pyro.contrib.gp.kernels.dot_product.DotProduct`

Implementation of Linear kernel:

$$k(x, z) = \sigma^2 x \cdot z.$$

Doing Gaussian Process regression with linear kernel is equivalent to doing a linear regression.

Note: Here we implement the homogeneous version. To use the inhomogeneous version, consider using `Polynomial` kernel with `degree=1` or making a `Sum` with a `Bias` kernel.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.12 Matern32

class Matern32(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*, *name='Matern32'*)
Bases: `pyro.contrib.gp.kernels.isotropic.Isotropy`

Implementation of Matern32 kernel:

$$k(x,z) = \sigma^2 \left(1 + \sqrt{3} \times \frac{|x-z|}{l}\right) \exp\left(-\sqrt{3} \times \frac{|x-z|}{l}\right).$$

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.13 Matern52

class Matern52(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*, *name='Matern52'*)
Bases: `pyro.contrib.gp.kernels.isotropic.Isotropy`

Implementation of Matern52 kernel:

$$k(x,z) = \sigma^2 \left(1 + \sqrt{5} \times \frac{|x-z|}{l} + \frac{5}{3} \times \frac{|x-z|^2}{l^2}\right) \exp\left(-\sqrt{5} \times \frac{|x-z|}{l}\right).$$

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.

- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.14 Periodic

class Periodic(*input_dim*, *variance=None*, *lengthscale=None*, *period=None*, *active_dims=None*, *name='Periodic'*)

Bases: pyro.contrib.gp.kernels.kernel.Kernel

Implementation of Periodic kernel:

$$k(x,z) = \sigma^2 \exp\left(-2 \times \frac{\sin^2(\pi(x-z)/p)}{l^2}\right),$$

where p is the period parameter.

References:

[1] *Introduction to Gaussian processes*, David J.C. MacKay

Parameters

- **lengthscale** (*torch.Tensor*) – Length scale parameter of this kernel.
- **period** (*torch.Tensor*) – Period parameter of this kernel.

forward(*X*, *Z=None*, *diag=False*)

Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.15 Polynomial

class Polynomial(*input_dim*, *variance=None*, *bias=None*, *degree=1*, *active_dims=None*, *name='Polynomial'*)

Bases: pyro.contrib.gp.kernels.dot_product.DotProduct

Implementation of Polynomial kernel:

$$k(x,z) = \sigma^2(\text{bias} + x \cdot z)^d.$$

Parameters

- **bias** (*torch.Tensor*) – Bias parameter of this kernel. Should be positive.
- **degree** (*int*) – Degree d of the polynomial.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.16 Product

class Product(*kern0*, *kern1*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Combination

Returns a new kernel which acts like a product/tensor product of two kernels. The second kernel can be a constant.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.17 RBF

class RBF(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*, *name='RBF'*)
Bases: pyro.contrib.gp.kernels.isotropic.Isotropy

Implementation of Radial Basis Function kernel:

$$k(x, z) = \sigma^2 \exp\left(-0.5 \times \frac{|x-z|^2}{l^2}\right).$$

Note: This kernel also has name *Squared Exponential* in literature.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.

- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.18 RationalQuadratic

class RationalQuadratic(*input_dim*, *variance=None*, *lengthscale=None*, *scale_mixture=None*, *active_dims=None*, *name='RationalQuadratic'*)

Bases: pyro.contrib.gp.kernels.isotropic.Isotropy

Implementation of RationalQuadratic kernel:

$$k(x,z) = \sigma^2 \left(1 + 0.5 \times \frac{|x-z|^2}{\alpha l^2}\right)^{-\alpha}.$$

Parameters **scale_mixture** (*torch.Tensor*) – Scale mixture (α) parameter of this kernel. Should have size 1.

forward(*X*, *Z=None*, *diag=False*)

Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.19 Sum

class Sum(*kern0*, *kern1*, *name=None*)

Bases: pyro.contrib.gp.kernels.kernel.Combination

Returns a new kernel which acts like a sum/direct sum of two kernels. The second kernel can be a constant.

forward(*X*, *Z=None*, *diag=False*)

Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.20 Transforming

class Transforming(*kern*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Base class for kernels derived from a kernel by some transforms such as warping, exponent, vertical scaling.

Parameters **kern** (Kernel) – The original kernel.

13.2.21 VerticalScaling

class VerticalScaling(*kern*, *vscaling_fn*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Transforming

Creates a new kernel according to

$$k_{new}(x, z) = f(x)k(x, z)f(z),$$

where f is a function.

Parameters **vscaling_fn** (*callable*) – A vertical scaling function f.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.22 Warping

class Warping(*kern*, *iwarping_fn=None*, *owarping_coef=None*, *name=None*)
Bases: pyro.contrib.gp.kernels.kernel.Transforming

Creates a new kernel according to

$$k_{new}(x, z) = q(k(f(x), f(z))),$$

where f is an function and q is a polynomial with non-negative coefficients owarping_coef.

We can take advantage of f to combine a Gaussian Process kernel with a deep learning architecture. For example:

```
>>> linear = torch.nn.Linear(10, 3)
>>> # register its parameters to Pyro's ParamStore and wrap it by lambda
>>> # to call the primitive pyro.module each time we use the linear function
>>> pyro_linear_fn = lambda x: pyro.module("linear", linear)(x)
>>> kernel = gp.kernels.Matern52(input_dim=3, lengthscale=torch.ones(3))
>>> warped_kernel = gp.kernels.Warping(kernel, pyro_linear_fn)
```

Reference:

[1] *Deep Kernel Learning*, Andrew G. Wilson, Zhiting Hu, Ruslan Salakhutdinov, Eric P. Xing

Parameters

- **iwarping_fn** (*callable*) – An input warping function f.
- **owarping_coef** (*list*) – A list of coefficients of the output warping polynomial. These coefficients must be non-negative.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.2.23 WhiteNoise

class WhiteNoise(*input_dim*, *variance=None*, *active_dims=None*, *name='WhiteNoise'*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Implementation of WhiteNoise kernel:

$$k(x, z) = \sigma^2 \delta(x, z),$$

where δ is a Dirac delta function.

forward(*X*, *Z=None*, *diag=False*)
Calculates covariance matrix of inputs on active dimensionals.

Parameters

- **X** (*torch.Tensor*) – A 2D tensor with shape $N \times input\_dim$.
- **Z** (*torch.Tensor*) – An (optional) 2D tensor with shape $M \times input\_dim$.
- **diag** (*bool*) – A flag to decide if we want to return full covariance matrix or just its diagonal part.

Returns covariance matrix of X and Z with shape $N \times M$

Return type torch.Tensor

13.3 Likelihoods

13.3.1 Binary

class Binary(*response_function=None*, *name='Binary'*)
Bases: pyro.contrib.gp.likelihoods.likelihood.Likelihood

Implementation of Binary likelihood, which is used for binary classification problems.

Binary likelihood uses *Bernoulli* distribution, so the output of response_function should be in range $(0, 1)$. By default, we use *sigmoid* function.

Parameters **response_function** (*callable*) – A mapping to correct domain for Binary likelihood.

forward (*f_loc*, *f_var*, *y=None*)
Samples y given f_{loc}, f_{var} according to

$$f \sim \mathbb{N} \bowtie \backslash \gg \partial \ll (f_{loc}, f_{var}),$$
$$y \sim \mathbb{B} \backslash \ltimes \bowtie \cong \ll \ll \sqsupset (f).$$

Note: The log likelihood is estimated using Monte Carlo with 1 sample of f.

Parameters

- **f_loc** (*torch.Tensor*) – Mean of latent function output.
- **f_var** (*torch.Tensor*) – Variance of latent function output.
- **y** (*torch.Tensor*) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

13.3.2 Gaussian

class Gaussian (*variance=None*, *name='Gaussian'*)
Bases: pyro.contrib.gp.likelihoods.likelihood.Likelihood

Implementation of Gaussian likelihood, which is used for regression problems.

Gaussian likelihood uses *Normal* distribution.

Parameters **variance** (*torch.Tensor*) – A variance parameter, which plays the role of `noise` in regression problems.

forward (*f_loc*, *f_var*, *y=None*)
Samples y given f_{loc}, f_{var} according to

$$y \sim \mathbb{N} \bowtie \backslash \gg \partial \ll (f_{loc}, f_{var} + \epsilon),$$

where ϵ is the `variance` parameter of this likelihood.

Parameters

- **f_loc** (*torch.Tensor*) – Mean of latent function output.
- **f_var** (*torch.Tensor*) – Variance of latent function output.
- **y** (*torch.Tensor*) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

13.3.3 Likelihood

class Likelihood(*name=None*)
Bases: *pyro.contrib.gp.util.Parameterized*

Base class for likelihoods used in Gaussian Process.

Every inherited class should implement a forward pass which takes an input f and returns a sample y.

forward(*f_loc*, *f_var*, *y=None*)
Samples y given f_{loc}, f_{var}.

Parameters

- **f_loc** (*torch.Tensor*) – Mean of latent function output.
- **f_var** (*torch.Tensor*) – Variance of latent function output.
- **y** (*torch.Tensor*) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

13.3.4 MultiClass

class MultiClass(*num_classes*, *response_function=None*, *name='MultiClass'*)
Bases: pyro.contrib.gp.likelihoods.likelihood.Likelihood

Implementation of MultiClass likelihood, which is used for multi-class classification problems.

MultiClass likelihood uses *Categorical* distribution, so `response_function` should normalize its input's rightmost axis. By default, we use *softmax* function.

Parameters

- **num_classes** (*int*) – Number of classes for prediction.
- **response_function** (*callable*) – A mapping to correct domain for MultiClass likelihood.

forward(*f_loc*, *f_var*, *y=None*)
Samples y given f_{loc}, f_{var} according to

$$f \sim \mathbb{N}\bowtie\backslash\gg\supset\prec(f_{loc}, f_{var}),$$
$$y \sim \mathbb{C}\supset\approx\eth\bowtie\backslash\sqsupset\supset\prec(f).$$

Note: The log likelihood is estimated using Monte Carlo with 1 sample of f.

Parameters

- **f_loc** (*torch.Tensor*) – Mean of latent function output.
- **f_var** (*torch.Tensor*) – Variance of latent function output.
- **y** (*torch.Tensor*) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

13.3.5 Poisson

class Poisson(*response_function=None*, *name='Poisson'*)
Bases: pyro.contrib.gp.likelihoods.likelihood.Likelihood

Implementation of Poisson likelihood, which is used for count data.

Poisson likelihood uses the *Poisson* distribution, so the output of response_function should be positive. By default, we use torch.exp() as response function, corresponding to a log-Gaussian Cox process.

Parameters **response_function** (*callable*) – A mapping to positive real numbers.

forward(*f_loc*, *f_var*, *y=None*)
Samples y given f_{loc}, f_{var} according to

$$f \sim \mathbb{N}(f_{loc}, f_{var}),$$
$$y \sim \mathbb{P}(\exp(f)).$$

Note: The log likelihood is estimated using Monte Carlo with 1 sample of f.

Parameters

- **f_loc** (*torch.Tensor*) – Mean of latent function output.
- **f_var** (*torch.Tensor*) – Variance of latent function output.
- **y** (*torch.Tensor*) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

13.4 Util

class Parameterized(*name=None*)
Bases: torch.nn.modules.module.Module

Base class for other modules in Gaussin Process module.

Parameters of this object can be set priors, set constraints, or fixed to a specific value.

By default, data of a parameter is a float torch.Tensor (unless we use torch.set_default_tensor_type() to change default tensor type). To cast these parameters to a correct data type or GPU device, we can call methods such as double() or cuda(). See torch.nn.Module for more information.

Parameters **name** (*str*) – Name of this object.

set_prior(*param*, *prior*)
Sets a prior to a parameter.

Parameters

- **param** (*str*) – Name of the parameter.
- **prior** (*Distribution*) – A Pyro prior distribution.

set_constraint(*param*, *constraint*)
Sets a constraint to a parameter.

Parameters

- **param** (*str*) – Name of the parameter.
- **constraint** (*Constraint*) – A PyTorch constraint. See `torch.distributions.constraints` for a list of constraints.

fix_param(*param*, *value=None*)
Fixes a parameter to a specic value. If `value=None`, fixes the parameter to the default value.

Parameters

- **param** (*str*) – Name of the parameter.
- **value** (*torch.Tensor*) – Fixed value.

set_mode(*mode*, *recursive=True*)
Sets `mode` of this object to be able to use its parameters in stochastic functions. If `mode="model"`, a parameter with prior will get its value from the primitive *pyro.sample()*. If `mode="guide"` or there is no prior on a parameter, *pyro.param()* will be called.

This method automatically sets `mode` for submodules which belong to *Parameterized* class unless `recursive=False`.

Parameters

- **mode** (*str*) – Either "model" or "guide".
- **recursive** (*bool*) – A flag to tell if we want to set mode for all submodules.

get_param(*param*)
Gets the current value of a parameter. The correct behavior will depend on `mode` of this object (see *set_mode()* method).

Parameters **param** (*str*) – Name of the parameter.

conditional(*Xnew*, *X*, *kernel*, *f_loc*, *f_scale_tril=None*, *Lff=None*, *full_cov=False*, *whiten=False*, *jitter=1e-06*)
Given X_{new}, predicts loc and covariance matrix of the conditional multivariate normal distribution

$$p(f^*(X_{new}) \mid X, k, f_{loc}, f_{scale\_tril}).$$

Here `f_loc` and `f_scale_tril` are variation parameters of the variational distribution

$$q(f \mid f_{loc}, f_{scale\_tril}) \sim p(f|X, y),$$

where f is the function value of the Gaussian Process given input X

$$p(f(X)) \sim \mathcal{N}(0, k(X, X))$$

and y is computed from f by some likelihood function $p(y|f)$.

In case `f_scale_tril=None`, we consider $f = f_{loc}$ and computes

$$p(f^*(X_{new}) \mid X, k, f).$$

In case `f_scale_tril` is not `None`, we follow the derivation from reference [1]. For the case `f_scale_tril=None`, we follow the popular reference [2].

References:

[1] Sparse GPs: approximate the posterior, not the model

[2] *Gaussian Processes for Machine Learning*, Carl E. Rasmussen, Christopher K. I. Williams

Parameters

- **Xnew** (*torch.Tensor*) – A new input data.
- **X** (*torch.Tensor*) – An input data to be conditioned on.
- **kernel** (*Kernel*) – A Pyro kernel object.
- **f_loc** (*torch.Tensor*) – Mean of $q(f)$. In case `f_scale_tril=None`, $f_{loc} = f$.
- **f_scale_tril** (*torch.Tensor*) – Lower triangular decomposition of covariance matrix of $q(f)$'s .
- **Lff** (*torch.Tensor*) – Lower triangular decomposition of $kernel(X, X)$ (optional).
- **full_cov** (*bool*) – A flag to decide if we want to return full covariance matrix or just variance.
- **whiten** (*bool*) – A flag to tell if `f_loc` and `f_scale_tril` are already transformed by the inverse of `Lff`.
- **jitter** (*float*) – A small positive term which is added into the diagonal part of a covariance matrix to help stablize its Cholesky decomposition.

Returns loc and covariance matrix (or variance) of $p(f^*(X_{new}))$

Return type tuple(torch.Tensor, torch.Tensor)

CHAPTER 14

Tracking

14.1 Data Association

class MarginalAssignment(*exists_logits*, *assign_logits*, *bp_iters=None*)

Computes marginal data associations between objects and detections.

This assumes that each detection corresponds to zero or one object, and each object corresponds to zero or more detections. Specifically this does not assume detections have been partitioned into frames of mutual exclusion as is common in 2-D assignment problems.

Parameters

- **exists_logits** (*torch.Tensor*) – a tensor of shape `[num_objects]` representing per-object factors for existence of each potential object.
- **assign_logits** (*torch.Tensor*) – a tensor of shape `[num_detections, num_objects]` representing per-edge factors of assignment probability, where each edge denotes that a given detection associates with a single object.
- **bp_iters** (*int*) – optional number of belief propagation iterations. If unspecified or `None` an expensive exact algorithm will be used.

Variables

- **num_detections** (*int*) – the number of detections
- **num_objects** (*int*) – the number of (potentially existing) objects
- **exists_dist** (`pyro.distributions.Bernoulli`) – a mean field posterior distribution over object existence.
- **assign_dist** (`pyro.distributions.Categorical`) – a mean field posterior distribution over the object (or None) to which each detection associates. This has `.event_shape == (num_objects + 1,)` where the final element denotes spurious detection, and `.batch_shape == (num_frames, num_detections)`.

class MarginalAssignmentSparse(*num_objects*, *num_detections*, *edges*, *exists_logits*, *assign_logits*, *bp_iters*)

A cheap sparse version of *MarginalAssignment*.

Parameters

- **num_detections** (*int*) – the number of detections
- **num_objects** (*int*) – the number of (potentially existing) objects
- **edges** (*torch.LongTensor*) – a `[2, num_edges]`-shaped tensor of (detection, object) index pairs specifying feasible associations.
- **exists_logits** (*torch.Tensor*) – a tensor of shape `[num_objects]` representing per-object factors for existence of each potential object.
- **assign_logits** (*torch.Tensor*) – a tensor of shape `[num_edges]` representing per-edge factors of assignment probability, where each edge denotes that a given detection associates with a single object.
- **bp_iters** (*int*) – optional number of belief propagation iterations. If unspecified or `None` an expensive exact algorithm will be used.

Variables

- **num_detections** (*int*) – the number of detections
- **num_objects** (*int*) – the number of (potentially existing) objects
- **exists_dist** (`pyro.distributions.Bernoulli`) – a mean field posterior distribution over object existence.
- **assign_dist** (`pyro.distributions.Categorical`) – a mean field posterior distribution over the object (or None) to which each detection associates. This has `.event_shape == (num_objects + 1,)` where the final element denotes spurious detection, and `.batch_shape == (num_frames, num_detections)`.

class MarginalAssignmentPersistent(*exists_logits*, *assign_logits*, *bp_iters=None*, *bp_momentum=0.5*)

This computes marginal distributions of a multi-frame multi-object data association problem with an unknown number of persistent objects.

The inputs are factors in a factor graph (existence probabilites for each potential object and assignment probabilities for each object-detection pair), and the outputs are marginal distributions of posterior existence probability of each potential object and posterior assignment probabilites of each object-detection pair.

This assumes a shared (maximum) number of detections per frame; to handle variable number of detections, simply set corresponding elements of `assign_logits` to `-float('inf')`.

Parameters

- **exists_logits** (*torch.Tensor*) – a tensor of shape `[num_objects]` representing per-object factors for existence of each potential object.
- **assign_logits** (*torch.Tensor*) – a tensor of shape `[num_frames, num_detections, num_objects]` representing per-edge factors of assignment probability, where each edge denotes that at a given time frame a given detection associates with a single object.
- **bp_iters** (*int*) – optional number of belief propagation iterations. If unspecified or `None` an expensive exact algorithm will be used.
- **bp_momentum** (*float*) – optional momentum to use for belief propagation. Should be in the interval `[0,1)`.

Variables

- **num_frames** (*int*) – the number of time frames
- **num_detections** (*int*) – the (maximum) number of detections per frame
- **num_objects** (*int*) – the number of (potentially existing) objects
- **exists_dist** (pyro.distributions.Bernoulli) – a mean field posterior distribution over object existence.
- **assign_dist** (pyro.distributions.Categorical) – a mean field posterior distribution over the object (or None) to which each detection associates. This has `.event_shape == (num_objects + 1,)` where the final element denotes spurious detection, and `.batch_shape == (num_frames, num_detections)`.

compute_marginals(*exists_logits*, *assign_logits*)

This implements exact inference of pairwise marginals via enumeration. This is very expensive and is only useful for testing.

See *MarginalAssignment* for args and problem description.

compute_marginals_bp(*exists_logits*, *assign_logits*, *bp_iters*)

This implements approximate inference of pairwise marginals via loopy belief propagation, adapting the approach of [1].

See *MarginalAssignment* for args and problem description.

[1] Jason L. Williams, Roslyn A. Lau (2014) Approximate evaluation of marginal association probabilities with belief propagation https://arxiv.org/abs/1209.6299

compute_marginals_sparse_bp(*num_objects*, *num_detections*, *edges*, *exists_logits*, *assign_logits*, *bp_iters*)

This implements approximate inference of pairwise marginals via loopy belief propagation, adapting the approach of [1].

See *MarginalAssignmentSparse* for args and problem description.

[1] Jason L. Williams, Roslyn A. Lau (2014) Approximate evaluation of marginal association probabilities with belief propagation https://arxiv.org/abs/1209.6299

compute_marginals_persistent(*exists_logits*, *assign_logits*)

This implements exact inference of pairwise marginals via enumeration. This is very expensive and is only useful for testing.

See *MarginalAssignmentPersistent* for args and problem description.

compute_marginals_persistent_bp(*exists_logits*, *assign_logits*, *bp_iters*, *bp_momentum=0.5*)

This implements approximate inference of pairwise marginals via loopy belief propagation, adapting the approach of [1], [2].

See *MarginalAssignmentPersistent* for args and problem description.

[1] Jason L. Williams, Roslyn A. Lau (2014) Approximate evaluation of marginal association probabilities with belief propagation https://arxiv.org/abs/1209.6299

[2] Ryan Turner, Steven Bottone, Bhargav Avasarala (2014) A Complete Variational Tracker https://papers.nips.cc/paper/5572-a-complete-variational-tracker.pdf

14.2 Hashing

class LSH(*radius*)

Implements locality-sensitive hashing for low-dimensional euclidean space.

Allows to efficiently find neighbours of a point. Provides 2 guarantees:

- Difference between coordinates of points not returned by *nearby()* and input point is larger than `radius`.
- Difference between coordinates of points returned by *nearby()* and input point is smaller than 2 `radius`.

Example:

```
>>> radius = 1
>>> lsh = LSH(radius)
>>> a = torch.tensor([-0.51, -0.51]) # hash(a)=(-1,-1)
>>> b = torch.tensor([-0.49, -0.49]) # hash(a)=(0,0)
>>> c = torch.tensor([1.0, 1.0]) # hash(b)=(1,1)
>>> lsh.add('a', a)
>>> lsh.add('b', b)
>>> lsh.add('c', c)
>>> lsh.nearby('a') # even though c is within 2radius of a
set(['b'])
>>> lsh.nearby('b')
set(['a', 'c'])
>>> lsh.remove('b')
>>> lsh.nearby('a')
set([])
```

Parameters **radius** (*float*) – Scaling parameter used in hash function. Determines the size of the neighbourhood.

add(*key*, *point*)

Adds (`key`, `point`) pair to the hash.

Parameters

- **key** – Key used identify `point`.
- **point** (*torch.Tensor*) – data, should be detached and on cpu.

remove(*key*)

Removes `key` and corresponding point from the hash.

Raises `KeyError` if key is not in hash.

Parameters **key** – key used to identify point.

nearby(*key*)

Returns a set of keys which are neighbours of the point identified by `key`.

Two points are nearby if difference of each element of their hashes is smaller than 2. In euclidean space, this corresponds to all points $\mathbf{p}$ where $|\mathbf{p}_k - (\mathbf{p_{key}})_k| < r$, and some points (all points not guaranteed) where $|\mathbf{p}_k - (\mathbf{p_{key}})_k| < 2r$.

Parameters **key** – key used to identify input point.

Returns a set of keys identifying neighbours of the input point.

Return type set

class ApproxSet(*radius*)
Queries low-dimensional euclidean space for approximate occupancy.

Parameters **radius** (*float*) – scaling parameter used in hash function. Determines the size of the bin. See *LSH* for details.

try_add(*point*)
Attempts to add `point` to set. Only adds there are no points in the `point`'s bin.

Parameters **point** (*torch.Tensor*) – Point to be queried, should be detached and on cpu.

Returns `True` if point is successfully added, `False` if there is already a point in `point`'s bin.

Return type bool

merge_points(*points*, *radius*)
Greedily merge points that are closer than given radius.

This uses *LSH* to achieve complexity that is linear in the number of merged clusters and quadratic in the size of the largest merged cluster.

Parameters

- **points** (*torch.Tensor*) – A tensor of shape `(K,D)` where `K` is the number of points and `D` is the number of dimensions.
- **radius** (*float*) – The minimum distance nearer than which points will be merged.

Returns A tuple `(merged_points, groups)` where `merged_points` is a tensor of shape `(J,D)` where `J <= K`, and `groups` is a list of tuples of indices mapping merged points to original points. Note that `len(groups) == J` and `sum(len(group) for group in groups) == K`.

Return type tuple

CHAPTER 15

Optimal Experiment Design

The *pyro.contrib.oed* module provides tools to create optimal experiment designs for pyro models. In particular, it provides estimators for the average posterior entropy (APE) criterion.

To estimate the APE for a particular design, use:

```
def model(design):
    ...

eig = vi_ape(model, design, ...)
```

APE can then be minimised using existing optimisers in `pyro.optim`.

15.1 Expected Information Gain

vi_ape (*model*, *design*, *observation_labels*, *vi_parameters*, *is_parameters*)

Estimates the average posterior entropy (APE) loss function using variational inference (VI).

The APE loss function estimated by this method is defined as

$$APE(d) = E_{Y\sim p(y|\theta,d)}[H(p(\theta|Y,d))]$$

where $H[p(x)]$ is the differential entropy. The APE is related to expected information gain (EIG) by the equation

$$EIG(d) = H[p(\theta)] - APE(d)$$

in particular, minimising the APE is equivalent to maximising EIG.

Parameters

- **model** (*function*) – A pyro model accepting *design* as only argument.
- **design** (*torch.Tensor*) – Tensor representation of design
- **observation_labels** (*list*) – A subset of the sample sites present in *model*. These sites are regarded as future observations and other sites are regarded as latent variables over which a posterior is to be inferred.

- **vi_parameters** (*dict*) – Variational inference parameters which should include: *optim*: an instance of `pyro.Optim`, *guide*: a guide function compatible with *model*, *num_steps*: the number of VI steps to make, and *loss*: the loss function to use for VI
- **is_parameters** (*dict*) – Importance sampling parameters for the marginal distribution of Y. May include *num_samples*: the number of samples to draw from the marginal.

Returns Loss function estimate

Return type *torch.Tensor*

CHAPTER 16

Indices and tables

- genindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W