Pyro Documentation

Uber AI Labs

Sep 17, 2019

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 plate(*name*, *size=None*, *subsample_size=None*, *subsample=None*, *dim=None*, *use_cuda=None*, *device=None*)

Construct for conditionally independent sequences of variables.

`plate` can be used either sequentially as a generator or in parallel as a context manager (formerly `irange` and `iarange`, respectively).

Sequential *plate* is similar to `range()` in that it generates a sequence of values.

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

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

Additionally, *plate* 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 plate("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 *plate* 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 `plate` contexts.
- **use_cuda** (*bool*) – DEPRECATED, use the *device* arg instead. Optional bool specifying whether to use cuda tensors for *subsample* and *log_prob*. Defaults to `torch.Tensor.is_cuda`.
- **device** (*str*) – Optional keyword specifying which device to place the results of *subsample* and *log_prob* on. By default, results are placed on the same device as the default tensor.

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

Examples:

```
>>> # This version declares sequential independence and subsamples data:
>>> for i in plate('data', 100, subsample_size=10):
...     if z[i]:  # Control flow in this example prevents vectorization.
...         obs = sample('obs_{}'.format(i), dist.Normal(loc, scale),␣
↪obs=data[i])
```

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

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

```
>>> # This reuses two different independence contexts.
>>> x_axis = plate('outer', 320, dim=-1)
>>> y_axis = plate('inner', 200, dim=-2)
>>> with x_axis:
...     x_noise = sample("x_noise", dist.Normal(loc, scale))
...     assert x_noise.shape == (320,)
>>> with y_axis:
...     y_noise = sample("y_noise", dist.Normal(loc, scale))
...     assert y_noise.shape == (200, 1)
>>> with x_axis, y_axis:
...     xy_noise = sample("xy_noise", dist.Normal(loc, scale))
...     assert xy_noise.shape == (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(*is_validate=True*)
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.

trace(*fn=None*, *ignore_warnings=False*)

Lazy replacement for `torch.jit.trace()` 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.trace
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*)

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_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

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_plate_nesting** (*int*) – Optional bound on max number of nested *pyro.plate()* contexts. This is only required when enumerating over sample sites in parallel, e.g. if a site sets `infer={"enumerate": "parallel"}`. If omitted, ELBO may guess a valid value by running the (model,guide) pair once, however this guess may be incorrect if model or guide structure is dynamic.
- **vectorize_particles** (*bool*) – Whether to vectorize the ELBO computation over *num_particles*. Defaults to False. This requires static structure in model and guide.
- **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.
- **ignore_jit_warnings** (*bool*) – Flag to ignore warnings from the JIT tracer, when . All `torch.jit.TracerWarning` will be ignored.
- **retain_graph** (*bool*) – Whether to retain autograd graph during an SVI step. Defaults to None (False).

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_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

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 *plate* 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.

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

Computes the surrogate loss that can be differentiated with autograd to produce gradient estimates for the model and guide parameters

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_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

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 compilation will be triggered once per unique `**kwargs`.

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

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

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

class TraceGraph_ELBO(*num_particles=1*, *max_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

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) - *plate* 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_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

Bases: *pyro.infer.tracegraph_elbo.TraceGraph_ELBO*

Like *TraceGraph_ELBO* but uses `torch.jit.trace()` 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 compilation will be triggered once per unique `**kwargs`.

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

class BackwardSampleMessenger(*enum_trace*, *guide_trace*)

Bases: *pyro.poutine.messenger.Messenger*

Implements forward filtering / backward sampling for sampling from the joint posterior distribution

class TraceEnum_ELBO(*num_particles=1*, *max_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

Bases: *pyro.infer.elbo.ELBO*

A trace implementation of ELBO-based SVI that supports - exhaustive enumeration over discrete sample sites, and - local parallel sampling over any sample site.

To enumerate over a sample site in the `guide`, mark the site with either `infer={'enumerate': 'sequential'}` or `infer={'enumerate': 'parallel'}`. To configure all guide sites at once, use *config_enumerate()*. To enumerate over a sample site in the `model`, mark the site `infer={'enumerate': 'parallel'}` and ensure the site does not appear in the `guide`.

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

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.

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

Computes marginal distributions at each model-enumerated sample site.

Returns a dict mapping site name to marginal `Distribution` object

Return type OrderedDict

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

Sample from the joint posterior distribution of all model-enumerated sites given all observations

class JitTraceEnum_ELBO(*num_particles=1*, *max_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

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 compilation will be triggered once per unique `**kwargs`.

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

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

class TraceMeanField_ELBO(*num_particles=1*, *max_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

Bases: *pyro.infer.trace_elbo.Trace_ELBO*

A trace implementation of ELBO-based SVI. This is currently the only ELBO estimator in Pyro that uses analytic KL divergences when those are available.

In contrast to, e.g., *TraceGraph_ELBO* and `Trace_ELBO` this estimator places restrictions on the dependency structure of the model and guide. In particular it assumes that the guide has a mean-field structure, i.e. that it factorizes across the different latent variables present in the guide. It also assumes that all of the latent variables in the guide are reparameterized. This latter condition is satisfied for, e.g., the Normal distribution but is not satisfied for, e.g., the Categorical distribution.

Warning: This estimator may give incorrect results if the mean-field condition is not satisfied.

Note for advanced users:

The mean field condition is a sufficient but not necessary condition for this estimator to be correct. The precise condition is that for every latent variable z in the guide, its parents in the model must not include any latent variables that are descendants of z in the guide. Here 'parents in the model' and 'descendants in the guide' is with respect to the corresponding (statistical) dependency structure. For example, this condition is always satisfied if the model and guide have identical dependency structures.

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.

class JitTraceMeanField_ELBO(*num_particles=1*, *max_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*, *ignore_jit_warnings=False*, *retain_graph=None*)

Bases: *pyro.infer.trace_mean_field_elbo.TraceMeanField_ELBO*

Like *TraceMeanField_ELBO* but uses *pyro.ops.jit.trace()* 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 compilation will be triggered once per unique `**kwargs`.

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

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

class RenyiELBO(*alpha=0*, *num_particles=2*, *max_plate_nesting=inf*, *max_iarange_nesting=None*, *vectorize_particles=False*, *strict_enumeration_warning=True*)

Bases: *pyro.infer.elbo.ELBO*

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

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

Note: Setting $\alpha < 1$ gives a better bound than the usual ELBO. 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 objective (gradient) estimator. Default is 2.
- **max_plate_nesting** (*int*) – Bound on max number of nested *pyro.plate()* contexts. Default is infinity.
- **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.

logsumexp(*input*, *dim*, *keepdim=False*, *out=None*)

Returns the log of summed exponentials of each row of the `input` tensor in the given dimension `dim`. The computation is numerically stabilized.

For summation index j given by *dim* and other indices i, the result is

$$\text{logsumexp}(x)_i = \log \sum_j \exp(x_{ij})$$

If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed (see `torch.squeeze()`), resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).

Args: input (Tensor): the input tensor dim (int or tuple of ints): the dimension or dimensions to reduce keepdim (bool): whether the output tensor has `dim` retained or not out (Tensor, optional): the output tensor

Example::

```
>>> a = torch.randn(3, 3)
>>> torch.logsumexp(a, 1)
tensor([ 0.8442,  1.4322,  0.8711])
```

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.

get_ESS()

Compute (Importance Sampling) Effective Sample Size (ESS).

get_log_normalizer()

Estimator of the normalizing constant of the target distribution. (mean of the unnormalized weights)

get_normalized_weights(*log_scale=False*)

Compute the normalized importance weights.

logsumexp(*input*, *dim*, *keepdim=False*, *out=None*)

Returns the log of summed exponentials of each row of the `input` tensor in the given dimension `dim`. The computation is numerically stabilized.

For summation index j given by *dim* and other indices i, the result is

$$\text{logsumexp}(x)_i = \log \sum_j \exp(x_{ij})$$

If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed (see `torch.squeeze()`), resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).

Args: input (Tensor): the input tensor dim (int or tuple of ints): the dimension or dimensions to reduce keepdim (bool): whether the output tensor has `dim` retained or not out (Tensor, optional): the output tensor

Example::

```
>>> a = torch.randn(3, 3)
>>> torch.logsumexp(a, 1)
tensor([ 0.8442,  1.4322,  0.8711])
```

4.4 Discrete Inference

infer_discrete(*fn=None*, *first_available_dim=None*, *temperature=1*)

A poutine that samples discrete sites marked with `site["infer"]["enumerate"] = "parallel"`

from the posterior, conditioned on observations.

Example:

```
@infer_discrete(first_available_dim=-1, temperature=0)
@config_enumerate
def viterbi_decoder(data, hidden_dim=10):
    transition = 0.3 / hidden_dim + 0.7 * torch.eye(hidden_dim)
    means = torch.arange(float(hidden_dim))
    states = [0]
    for t in pyro.markov(range(len(data))):
        states.append(pyro.sample("states_{}".format(t),
                                  dist.Categorical(transition[states[-1]])))
        pyro.sample("obs_{}".format(t),
                    dist.Normal(means[states[-1]], 1.),
                    obs=data[t])
    return states  # returns maximum likelihood states
```

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **first_available_dim** (*int*) – The first tensor dimension (counting from the right) that is available for parallel enumeration. This dimension and all dimensions left may be used internally by Pyro. This should be a negative integer.
- **temperature** (*int*) – Either 1 (sample via forward-filter backward-sample) or 0 (optimize via Viterbi-like MAP inference). Defaults to 1 (sample).

4.5 Inference Utilities

class EmpiricalMarginal(*trace_posterior*, *sites=None*, *validate_args=None*)

Bases: pyro.distributions.empirical.Empirical

Marginal distribution over a single site (or multiple, provided they have the same shape) from the TracePosterior's model.

..note:: If multiple sites are specified, they must have the same tensor shape. Samples from each site will be stacked and stored within a single tensor. See *Empirical*. To hold the marginal distribution of sites having different shapes, use *Marginals* instead.

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.

class Marginals(*trace_posterior*, *sites=None*, *validate_args=None*)

Bases: object

Holds the marginal distribution over one or more sites from the TracePosterior's model. This is a convenience container class, which can be extended by TracePosterior subclasses. e.g. for implementing diagnostics.

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.

empirical

support (*flatten=False*)

class TracePosterior (*num_chains=1*)
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.

information_criterion (*pointwise=False*)
Computes information criterion of the model. Currently, returns only "Widely Applicable/Watanabe-Akaike Information Criterion" (WAIC) and the corresponding effective number of parameters.

Reference:

[1] *Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC*, Aki Vehtari, Andrew Gelman, and Jonah Gabry

Parameters **pointwise** (*bool*) – a flag to decide if we want to get a vectorized WAIC or not. When `pointwise=False`, returns the sum.

Returns OrderedDict a dictionary containing values of WAIC and its effective number of parameters.

marginal (*sites=None*)

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.

marginal (*sites=None*)

4.6 MCMC

4.6.1 MCMC

class MCMC(*kernel*, *num_samples*, *warmup_steps=0*, *num_chains=1*, *mp_context=None*, *disable_progbar=False*)

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.

Note: The case of *num_chains > 1* uses python multiprocessing to run parallel chains in multiple processes. This goes with the usual caveats around multiprocessing in python, e.g. the model used to initialize the `kernel` must be serializable via *pickle*, and the performance / constraints will be platform dependent (e.g. only the "spawn" context is available in Windows). This has also not been extensively tested on the Windows platform.

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. If not provided, default is half of *num_samples*.
- **num_chains** (*int*) – Number of MCMC chains to run in parallel. Depending on whether *num_chains* is 1 or more than 1, this class internally dispatches to either *_SingleSampler* or *_ParallelSampler*.
- **mp_context** (*str*) – Multiprocessing context to use when *num_chains > 1*. Only applicable for Python 3.5 and above. Use *mp_context="spawn"* for CUDA.
- **disable_progbar** (*bool*) – Disable progress bar and diagnostics update.

marginal(*sites=None*)

4.6.2 HMC

class HMC(*model*, *step_size=1*, *trajectory_length=None*, *num_steps=None*, *adapt_step_size=True*, *adapt_mass_matrix=True*, *full_mass=False*, *transforms=None*, *max_plate_nesting=None*, *jit_compile=False*, *ignore_jit_warnings=False*)

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.
- **adapt_mass_matrix** (*bool*) – A flag to decide if we want to adapt mass matrix during warm-up phase using Welford scheme.
- **full_mass** (*bool*) – A flag to decide if mass matrix is dense or diagonal.
- **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`.
- **max_plate_nesting** (*int*) – Optional bound on max number of nested *pyro.plate()* contexts. This is required if model contains discrete sample sites that can be enumerated over in parallel.
- **jit_compile** (*bool*) – Optional parameter denoting whether to use the PyTorch JIT to trace the log density computation, and use this optimized executable trace in the integrator.
- **ignore_jit_warnings** (*bool*) – Flag to ignore warnings from the JIT tracer when `jit_compile=True`. Default is False.

Note: Internally, the mass matrix will be ordered according to the order of the names of latent variables, not the order of their appearance in the model.

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 = mcmc_run.marginal('beta').empirical['beta']
>>> posterior.mean  # doctest: +SKIP
tensor([ 0.9819,  1.9258,  2.9737])
```

cleanup()

diagnostics()

initial_trace
Find a valid trace to initiate the MCMC sampler. This is also used as a prototype trace to inter-convert between Pyro's trace object and dict object used by the integrator.

inverse_mass_matrix

num_steps

sample(*trace*)

setup(*warmup_steps*, **args*, ***kwargs*)

step_size

4.6.3 NUTS

class NUTS(*model*, *step_size=1*, *adapt_step_size=True*, *adapt_mass_matrix=True*, *full_mass=False*, *use_multinomial_sampling=True*, *transforms=None*, *max_plate_nesting=None*, *jit_compile=False*, *ignore_jit_warnings=False*)
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.
- **adapt_mass_matrix** (*bool*) – A flag to decide if we want to adapt mass matrix during warm-up phase using Welford scheme.
- **full_mass** (*bool*) – A flag to decide if mass matrix is dense or diagonal.
- **use_multinomial_sampling** (*bool*) – A flag to decide if we want to sample candidates along its trajectory using "multinomial sampling" or using "slice sampling". Slice sampling is used in the original NUTS paper [1], while multinomial sampling is suggested in [2]. By default, this flag is set to True. If it is set to *False*, NUTS uses slice sampling.
- **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`.

- **max_plate_nesting** (*int*) – Optional bound on max number of nested *pyro.plate()* contexts. This is required if model contains discrete sample sites that can be enumerated over in parallel.
- **jit_compile** (*bool*) – Optional parameter denoting whether to use the PyTorch JIT to trace the log density computation, and use this optimized executable trace in the integrator.

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 = mcmc_run.marginal('beta').empirical['beta']
>>> posterior.mean  # doctest: +SKIP
tensor([ 0.9221,  1.9464,  2.9228])
```

sample(*trace*)

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 Binomial

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

5.1.4 Categorical

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

5.1.5 Cauchy

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

5.1.6 Chi2

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

5.1.7 Dirichlet

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

5.1.8 Exponential

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

5.1.9 ExponentialFamily

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

5.1.10 FisherSnedecor

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

5.1.11 Gamma

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

5.1.12 Geometric

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

5.1.13 Gumbel

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

5.1.14 HalfCauchy

class HalfCauchy(*scale*, *validate_args=None*)
Wraps `torch.distributions.half_cauchy.HalfCauchy` with *TorchDistributionMixin*.

5.1.15 HalfNormal

class HalfNormal(*scale*, *validate_args=None*)
Wraps `torch.distributions.half_normal.HalfNormal` with *TorchDistributionMixin*.

5.1.16 Independent

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

5.1.17 Laplace

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

5.1.18 LogNormal

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

5.1.19 LogisticNormal

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

5.1.20 LowRankMultivariateNormal

class LowRankMultivariateNormal(*loc*, *cov_factor*, *cov_diag*, *validate_args=None*)
Wraps `torch.distributions.lowrank_multivariate_normal.LowRankMultivariateNormal` with *TorchDistributionMixin*.

5.1.21 Multinomial

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

5.1.22 MultivariateNormal

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

5.1.23 NegativeBinomial

class NegativeBinomial(*total_count*, *probs=None*, *logits=None*, *validate_args=None*)
Wraps torch.distributions.negative_binomial.NegativeBinomial with *TorchDistributionMixin*.

5.1.24 Normal

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

5.1.25 OneHotCategorical

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

5.1.26 Pareto

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

5.1.27 Poisson

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

5.1.28 RelaxedBernoulli

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

5.1.29 RelaxedOneHotCategorical

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

5.1.30 StudentT

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

5.1.31 TransformedDistribution

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

5.1.32 Uniform

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

5.1.33 Weibull

class Weibull(*scale*, *concentration*, *validate_args=None*)
Wraps `torch.distributions.weibull.Weibull` 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(*expand=True*)
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.

Parameters **expand** (*bool*) – whether to expand the result to a tensor of shape `(n,) + batch_shape + event_shape`. If false, the return value has unexpanded shape `(n,) + (1,)*len(batch_shape) + event_shape` which can be broadcasted to the full shape.

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_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*)

to_event(*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.to_event(1)
>>> [d2.batch_shape, d2.event_shape]
[torch.Size([2]), torch.Size([3, 4, 5])]
>>> d3 = d1.to_event(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 pyro.distributions.torch.Independent

independent(*reinterpreted_batch_ndims=None*)

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 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': Real(), 'loc': Real(), 'scale_tril': LowerTriangu

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

5.2.5 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': Real(), 'v': Real()}

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*x*)

mean

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

support = Real()

variance

5.2.6 EmpiricalDistribution

class Empirical(*samples*, *log_weights*, *validate_args=None*)

Bases: pyro.distributions.torch_distribution.TorchDistribution

Empirical distribution associated with the sampled data.

Parameters

- **samples** (*torch.Tensor*) – samples from the empirical distribution.
- **log_weights** (*torch.Tensor*) – log weights (optional) corresponding to the samples. The leftmost shape of `log_weights` must match that of samples

arg_constraints = {}

enumerate_support(*expand=True*)

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

event_shape

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

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`.

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

log_weights

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 = Real()

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

5.2.7 GaussianScaleMixture

class GaussianScaleMixture(*coord_scale*, *component_logits*, *component_scale*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Mixture of Normal distributions with zero mean and diagonal covariance matrices.

That is, this distribution is a mixture with K components, where each component distribution is a D-dimensional Normal distribution with zero mean and a D-dimensional diagonal covariance matrix. The K different covariance matrices are controlled by the parameters *coord_scale* and *component_scale*. That is, the covariance matrix of the k'th component is given by

Sigma_ii = (component_scale_k * coord_scale_i) ** 2 (i = 1, . . . , D)

where *component_scale'_k is a positive scale factor and 'coord_scale'_i are positive scale parameters shared between all K components. The mixture weights are controlled by a K-dimensional vector of softmax logits, 'component_logits*. This distribution implements pathwise derivatives for samples from the distribution. This distribution does not currently support batched parameters.

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

Note that this distribution supports both even and odd dimensions, but the former should be more a bit higher precision, since it doesn't use any erfs in the backward call. Also note that this distribution does not support D = 1.

Parameters

- **coord_scale** (*torch.tensor*) – D-dimensional vector of scales
- **component_logits** (*torch.tensor*) – K-dimensional vector of logits
- **component_scale** (*torch.tensor*) – K-dimensional vector of scale multipliers

arg_constraints = {'component_logits': Real(), 'component_scale': GreaterThan(lower_

has_rsample = True

log_prob(*value*)

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

5.2.8 MaskedMixture

class MaskedMixture(*mask*, *component0*, *component1*, *validate_args=None*)
Bases: pyro.distributions.torch_distribution.TorchDistribution

A masked deterministic mixture of two distributions.

This is useful when the mask is sampled from another distribution, possibly correlated across the batch. Often the mask can be marginalized out via enumeration.

Example:

```
change_point = pyro.sample("change_point",
                           dist.Categorical(torch.ones(len(data) + 1)),
                           infer={'enumerate': 'parallel'})
mask = torch.arange(len(data), dtype=torch.long) >= changepoint
with pyro.plate("data", len(data)):
    pyro.sample("obs", MaskedMixture(mask, dist1, dist2), obs=data)
```

Parameters

- **mask** (*torch.Tensor*) – A byte tensor toggling between `component0` and `component1`.
- **component0** (`pyro.distributions.TorchDistribution`) – a distribution for batch elements `mask == 0`.
- **component1** (`pyro.distributions.TorchDistribution`) – a distribution for batch elements `mask == 1`.

arg_constraints = {}

expand(*batch_shape*)

has_rsample

log_prob(*value*)

mean

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

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

support

variance

5.2.9 MixtureOfDiagNormalsSharedCovariance

class MixtureOfDiagNormalsSharedCovariance(*locs*, *coord_scale*, *component_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 *coord_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, *component_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. Note that this distribution does not support dimension D = 1.

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

Parameters

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

arg_constraints = {'component_logits': Real(), 'coord_scale': GreaterThan(lower_boun

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

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

5.2.10 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': Real(), 'scale_tril': LowerTriangular()}

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

5.2.11 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()

5.2.12 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()

5.2.13 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*)

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

score_parts(*x*)

5.2.14 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': GreaterThan(lower_bound=0.0), 'loc': Real()}

expand(*batch_shape*)

log_prob(*value*)

support = Real()

5.2.15 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': Real()}

expand(*batch_shape*)

log_prob(*value*)

support = Real()

5.3 Transformed Distributions

5.3.1 InverseAutoRegressiveFlow

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

Bases: pyro.distributions.torch_transform.TransformModule

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

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

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

Together with *TransformedDistribution* this provides a way to create richer variational approximations.

Example usage:

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

The inverse of the Bijector is required when, e.g., scoring the log density of a sample with *TransformedDistribution*. This implementation caches the inverse of the Bijector when its forward operation is called, e.g., when sampling from *TransformedDistribution*. However, if the cached value isn't available, either because it was already popped from the cache, or an arbitary value is being scored, it will calculate it manually. Note that this is an operation that scales as O(D) where D is the input dimension, and so should be avoided for large dimensional uses. So in general, it is cheap to sample from IAF and score a value that was sampled by IAF, but expensive to score an arbitrary value.

Parameters

- **autoregressive_nn** (*nn.Module*) – an autoregressive neural network whose forward call returns a real-valued mean and logit-scale as a tuple
- **log_scale_min_clip** (*float*) – The minimum value for clipping the log(scale) from the autoregressive NN
- **log_scale_max_clip** (*float*) – The maximum value for clipping the log(scale) from the autoregressive NN

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

codomain = Real()

log_abs_det_jacobian(*x*, *y*)

Calculates the elementwise determinant of the log jacobian

5.3.2 InverseAutoRegressiveFlowStable

class InverseAutoregressiveFlowStable(*autoregressive_nn*, *sigmoid_bias=2.0*)
Bases: pyro.distributions.torch_transform.TransformModule

An implementation of an Inverse Autoregressive Flow, using Eqs (13)/(14) from Kingma Et Al., 2016,

$$\mathbf{y} = \sigma_t \odot \mathbf{x} + (1 - \sigma_t) \odot \mu_t$$

where $\mathbf{x}$ are the inputs, $\mathbf{y}$ are the outputs, μ_t, σ_t are calculated from an autoregressive network on $\mathbf{x}$, and σ_t is restricted to $[0, 1]$.

This variant of IAF is claimed by the authors to be more numerically stable than one using Eq (10), although in practice it leads to a restriction on the distributions that can be represented, presumably since the input is restricted to rescaling by a number on $[0, 1]$.

Example usage:

```
>>> from pyro.nn import AutoRegressiveNN
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> iaf = InverseAutoregressiveFlowStable(AutoRegressiveNN(10, [40]))
>>> iaf_module = pyro.module("my_iaf", iaf)
>>> iaf_dist = dist.TransformedDistribution(base_dist, [iaf])
>>> iaf_dist.sample()  # doctest: +SKIP
    tensor([-0.4071, -0.5030,  0.7924, -0.2366, -0.2387, -0.1417,  0.0868,
            0.1389, -0.4629,  0.0986])
```

See *InverseAutoregressiveFlow* docs for a discussion of the running cost.

Parameters

- **autoregressive_nn** (*nn.Module*) – an autoregressive neural network whose forward call returns a real-valued mean and logit-scale as a tuple
- **sigmoid_bias** (*float*) – bias on the hidden units fed into the sigmoid; default=‘2.0‘

References:

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

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

codomain = Real()

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

5.3.3 PermuteTransform

class PermuteTransform(*permutation*)
Bases: torch.distributions.transforms.Transform

A bijection that reorders the input dimensions, that is, multiplies the input by a permutation matrix. This is useful in between *InverseAutoregressiveFlow* transforms to increase the flexibility of the resulting distribution and stabilize learning. Whilst not being an autoregressive transform, the log absolute determinate of the Jacobian is easily calculable as 0. Note that reordering the input dimension between two layers of

InverseAutoregressiveFlow is not equivalent to reordering the dimension inside the MADE networks that those IAFs use; using a PermuteTransform results in a distribution with more flexibility.

Example usage:

```
>>> from pyro.nn import AutoRegressiveNN
>>> from pyro.distributions import InverseAutoregressiveFlow, PermuteTransform
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> iaf1 = InverseAutoregressiveFlow(AutoRegressiveNN(10, [40]))
>>> ff = PermuteTransform(torch.randperm(10, dtype=torch.long))
>>> iaf2 = InverseAutoregressiveFlow(AutoRegressiveNN(10, [40]))
>>> iaf_dist = dist.TransformedDistribution(base_dist, [iaf1, ff, iaf2])
>>> iaf_dist.sample()  # doctest: +SKIP
    tensor([-0.4071, -0.5030,  0.7924, -0.2366, -0.2387, -0.1417,  0.0868,
            0.1389, -0.4629,  0.0986])
```

Parameters **permutation** (*torch.LongTensor*) – a permutation ordering that is applied to the inputs.

bijective = True

codomain = Real()

inv_permutation

log_abs_det_jacobian(*x*, *y*)

Calculates the elementwise determinant of the log Jacobian, i.e. log(abs([dy_0/dx_0, ..., dy_{N-1}/dx_{N-1}])). Note that this type of transform is not autoregressive, so the log Jacobian is not the sum of the previous expression. However, it turns out it's always 0 (since the determinant is -1 or +1), and so returning a vector of zeros works.

5.3.4 PlanarFlow

class PlanarFlow(*input_dim*)

Bases: pyro.distributions.torch_transform.TransformModule

A 'planar' normalizing flow that uses the transformation

$$\mathbf{y} = \mathbf{x} + \mathbf{u}\tanh(\mathbf{w}^T\mathbf{z} + b)$$

where $\mathbf{x}$ are the inputs, $\mathbf{y}$ are the outputs, and the learnable parameters are $b \in \mathbb{R}$, $\mathbf{u} \in \mathbb{R}^D$, $\mathbf{w} \in \mathbb{R}^D$ for input dimension D. For this to be an invertible transformation, the condition $\mathbf{w}^T\mathbf{u} > -1$ is enforced.

Together with *TransformedDistribution* this provides a way to create richer variational approximations.

Example usage:

```
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> plf = PlanarFlow(10)
>>> plf_module = pyro.module("my_plf", plf)
>>> plf_dist = dist.TransformedDistribution(base_dist, [plf])
>>> plf_dist.sample()  # doctest: +SKIP
    tensor([-0.4071, -0.5030,  0.7924, -0.2366, -0.2387, -0.1417,  0.0868,
            0.1389, -0.4629,  0.0986])
```

The inverse of this transform does not possess an analytical solution and is left unimplemented. However, the inverse is cached when the forward operation is called during sampling, and so samples drawn using planar flow can be scored.

Parameters **input_dim** – the dimension of the input (and output) variable.

References:

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

codomain = Real()

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

reset_parameters()

u_hat()

5.3.5 TransformModule

class TransformModule
Bases: torch.distributions.transforms.Transform, torch.nn.modules.module.Module

Transforms with learnable parameters such as normalizing flows should inherit from this class rather than *Transform* so they are also a subclass of *nn.Module* and inherit all the useful methods of that class.

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

items()
Iterate over `(name, constrained_param)` pairs.

keys()
Iterate over param names.

values()
Iterate over constrained parameter values.

setdefault(*name*, *init_constrained_value*, *constraint=Real()*)
Retrieve a constrained parameter value from the if it exists, otherwise set the initial value. Note that this is a little fancier than `dict.setdefault()`.

If the parameter already exists, `init_constrained_tensor` will be ignored. To avoid expensive creation of `init_constrained_tensor` you can wrap it in a `lambda` that will only be evaluated if the parameter does not already exist:

```
param_store.get("foo", lambda: (0.001 * torch.randn(1000, 1000)).exp(),
                constraint=constraints.positive)
```

Parameters

- **name** (*str*) – parameter name
- **init_constrained_value** (*torch.Tensor or callable returning a torch.Tensor*) – initial constrained value
- **constraint** (*torch.distributions.constraints.Constraint*) – torch constraint object

Returns constrained parameter value

Return type torch.Tensor

named_parameters()
Returns an iterator over `(name, unconstrained_value)` tuples for each parameter in the ParamStore.

get_all_param_names()

replace_param(*param_name*, *new_param*, *old_param*)

get_param(*name*, *init_tensor=None*, *constraint=Real()*)
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
- **constraint** (*torch.distributions.constraints.Constraint*) – torch constraint

Returns parameter

Return type torch.Tensor

match(*name*)
Get all parameters that match regex. The parameter must exist.

Parameters **name** (*str*) – regular expression

Returns dict with key param name and value torch Tensor

param_name(*p*)
Get parameter name from parameter

Parameters **p** – parameter

Returns parameter name

get_state()
Get the ParamStore state.

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

save(*filename*)
Save parameters to disk

Parameters **filename** (*str*) – file name to save to

load(*filename*, *map_location=None*)
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** (*str*) – file name to load from
- **map_location** (*function, torch.device, string or a dict*) – specifies how to remap storage locations

param_with_module_name(*pyro_name*, *param_name*)

module_from_param_with_module_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_dims, param_dims=[1, 1], permutation=None, skip_connections=False, nonlinearity=ReLU()*)

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

An implementation of a MADE-like auto-regressive neural network.

Example usage: >>> x = torch.randn(100, 10) >>> arn = AutoRegressiveNN(10, [50], param_dims=[1]) >>> p = arn(x) # 1 parameters of size (100, 10) >>> arn = AutoRegressiveNN(10, [50], param_dims=[1, 1]) >>> m, s = arn(x) # 2 parameters of size (100, 10) >>> arn = AutoRegressiveNN(10, [50], param_dims=[1, 5, 3]) >>> a, b, c = arn(x) # 3 parameters of sizes, (100, 1, 10), (100, 5, 10), (100, 3, 10)

Parameters

- **input_dim** (*int*) – the dimensionality of the input
- **hidden_dims** (*list[int]*) – the dimensionality of the hidden units per layer
- **param_dims** (*list[int]*) – shape the output into parameters of dimension (p_n, input_dim) for p_n in param_dims when p_n > 1 and dimension (input_dim) when p_n == 1. The default is [1, 1], i.e. output two parameters of dimension (input_dim), which is useful for inverse autoregressive flow.
- **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.
- **skip_connections** (*bool*) – Whether to add skip connections from the input to the output.

- **nonlinearity** (*torch.nn.module*) – The nonlinearity to use in the feedforward network such as torch.nn.ReLU(). Note that no nonlinearity is applied to the final network output, so the output is an unbounded real number.

Reference:

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

forward(*x*)
The forward method

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

create_mask(*input_dim*, *observed_dim*, *hidden_dims*, *permutation*, *output_dim_multiplier*)
Creates MADE masks for a conditional distribution

Parameters

- **input_dim** (*int*) – the dimensionality of the input variable
- **observed_dim** (*int*) – the dimensionality of the variable that is conditioned on (for conditional densities)
- **hidden_dims** (*list[int]*) – the dimensionality of the hidden layers(s)
- **permutation** (*torch.LongTensor*) – the order of the input variables
- **output_dim_multiplier** (*int*) – tiles the output (e.g. for when a separate mean and scale parameter are desired)

sample_mask_indices(*input_dim*, *hidden_dim*, *simple=True*)
Samples the indices assigned to hidden units during the construction of MADE masks

Parameters

- **input_dim** (*int*) – the dimensionality of the input variable
- **hidden_dim** (*int*) – the dimensionality of the hidden layer
- **simple** (*bool*) – True to space fractional indices by rounding to nearest int, false round randomly

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})
svi = SVI(model, guide, pyro_scheduler, loss=TraceGraph_ELBO())
svi.step()
```

set_epoch(*epoch*)

8.2 PyTorch Optimizers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CosineAnnealingWarmRestarts (*optim_args*)
Wraps `torch.optim.CosineAnnealingWarmRestarts` 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*)

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* or *PyroOptim* object to be used for the named parameters. Together the `names` should partition up all desired parameters to optimize.

Raises **ValueError** – if any name is optimized by multiple optimizers.

step(*loss*, *params*)

get_step(*loss*, *params*)

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.

The result of *get_step()* will be differentiable, however the updated values from `step()` will be detached.

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*)

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=guide_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 plate context. The existing *batch_shape* must be broadcastable with the size of the *plate* 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 *plate* contexts.

```
>>> def model_broadcast_by_hand():
...     with IndepMessenger("batch", 100, dim=-2):
...         with IndepMessenger("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 IndepMessenger("batch", 100, dim=-2):
...         with IndepMessenger("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. This should be a negative integer.

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*

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`*

markov (*fn=None*, *history=1*, *keep=False*)

Markov dependency declaration.

This can be used in a variety of ways: - as a context manager - as a decorator for recursive functions - as an iterator for markov chains

Parameters

- **history** (*int*) – The number of previous contexts visible from the current context. Defaults to 1. If zero, this is similar to *pyro.plate*.
- **keep** (*bool*) – If true, frames are replayable. This is important when branching: if `keep=True`, neighboring branches at the same level can depend on each other; if `keep=False`, neighboring branches are independent (conditioned on their share"

mask (*fn=None*, *mask=None*)

Given a stochastic function with some batched sample statements and masking tensor, mask out some of the sample statements elementwise.

Parameters

- **fn** – a stochastic function (callable containing Pyro primitive calls)
- **mask** (*torch.ByteTensor*) – a `{0,1}`-valued masking tensor (1 includes a site, 0 excludes a site)

Returns stochastic function decorated with a `MaskMessenger`

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

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*

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*)

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*

config_enumerate(*guide=None*, *default='parallel'*, *expand=False*, *num_samples=None*)

Configures enumeration for all relevant sites in a guide. This is mainly used in conjunction with *TraceEnum_ELBO*.

When configuring for exhaustive enumeration of discrete variables, this configures all sample sites whose distribution satisfies `.has_enumerate_support == True`. When configuring for local parallel Monte Carlo sampling via `default="parallel", num_samples=n`, this configures all sample sites. This does not overwrite existing annotations `infer={"enumerate": ...}`.

This can be used as either a function:

```
guide = config_enumerate(guide)
```

or as a decorator:

```
@config_enumerate
def guide1(*args, **kwargs):
    ...

@config_enumerate(default="sequential", expand=True)
def guide2(*args, **kwargs):
    ...
```

Parameters

- **guide** (*callable*) – a pyro model that will be used as a guide in *SVI*.
- **default** (*str*) – Which enumerate strategy to use, one of "sequential", "parallel", or None. Defaults to "parallel".
- **expand** (*bool*) – Whether to expand enumerated sample values. See *enumerate_support()* for details. This only applies to exhaustive enumeration, where num_samples=None. If num_samples is not None, then this samples will always be expanded.
- **num_samples** (*int or None*) – if not None, use local Monte Carlo sampling rather than exhaustive enumeration. This makes sense for both continuous and discrete distributions.

Returns an annotated guide

Return type callable

9.2 Trace

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

Bases: networkx.classes.digraph.DiGraph

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())  # doctest: +SKIP
["_INPUT", "s", "z", "_RETURN"]
```

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

```
>>> trace.nodes["z"]  # doctest: +SKIP
{'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.plate` contexts appearing in the execution. `'done'`, `'stop'`, and `'continuation'` are only used by Pyro's internals.

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 Trace.<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

format_shapes(*title='Trace Shapes:'*, *last_site=None*)
> Returns a string showing a table of the shapes of all sites in the trace.

iter_stochastic_nodes()

> **Returns** an iterator over stochastic nodes in the trace.

log_prob_sum(*site_filter=<function Trace.<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

node_dict_factory
> alias of `collections.OrderedDict`

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

pack_tensors(*plate_to_symbol=None*)
> Computes packed representations of tensors in the trace. This should be called after *compute_log_prob()* or *compute_score_parts()*.

param_nodes

Returns a list of names of param sites

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

symbolize_dims (*plate_to_symbol=None*)
Assign unique symbols to all tensor dimensions.

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.

classmethod register (*fn=None*, *type=None*, *post=None*)

Parameters

- **fn** – function implementing operation
- **type** (*str*) – name of the operation (also passed to :func:~‘pyro.poutine.runtime.effectful‘)
- **post** (*bool*) – if *True*, use this operation as postprocess

Dynamically add operations to an effect. Useful for generating wrappers for libraries.

Example:

```
@SomeMessengerClass.register
def some_function(msg)
    ...do_something...
    return msg
```

classmethod unregister (*fn=None*, *type=None*)

Parameters

- **fn** – function implementing operation
- **type** (*str*) – name of the operation (also passed to :func:~‘pyro.poutine.runtime.effectful‘)

Dynamically remove operations from an effect. Useful for removing wrappers from libraries.

Example:

```
SomeMessengerClass.unregister(some_function, "name")
```

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 plate context. The existing *batch_shape* must be broadcastable with the size of the *plate* 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=None*, *size=None*, *dim=None*, *device=None*)

Bases: *pyro.poutine.messenger.Messenger*

This messenger keeps track of stack of independence information declared by nested `plate` contexts. This information is stored in a `cond_indep_stack` at each sample/observe site for consumption by `TraceMessenger`.

Example:

```
x_axis = IndepMessenger('outer', 320, dim=-1)
y_axis = IndepMessenger('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]))
with x_axis, y_axis:
    xy_noise = sample("xy_noise", dist.Normal(loc, scale).expand_by([200, 320]))
```

indices

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*)
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: `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

effectful(*fn=None*, *type=None*)

Parameters

- **fn** – function or callable that performs an effectful computation
- **type** (*str*) – the type label of the operation, e.g. *"sample"*

Wrapper for calling :func:~'pyro.poutine.runtime.apply_stack' to apply any active effects.

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 *plate*.

CHAPTER 10

Miscellaneous Ops

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

10.1 Utilities for HMC

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.

reset()

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*, *inverse_mass_matrix*, *step_size*, *num_steps=1*, *z_grads=None*)

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`.
- **inverse_mass_matrix** (*torch.Tensor*) – a tensor M^{-1} which is used to calculate kinetic energy: $E_{kinetic} = \frac{1}{2}z^T M^{-1} z$. Here M can be a 1D tensor (diagonal matrix) or a 2D tensor (dense matrix).
- **step_size** (*float*) – step size for each time step iteration.
- **num_steps** (*int*) – number of discrete time steps over which to integrate.
- **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`.

10.2 Newton Optimizers

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

10.3 Tensor Contraction

contract_expression(*equation*, **shapes*, ***kwargs*)

Wrapper around `opt_einsum.contract_expression()` that optionally uses Pyro's cheap optimizer and optionally caches contraction paths.

Parameters **cache_path** (*bool*) – whether to cache the contraction path. Defaults to True.

contract(*equation*, **operands*, ***kwargs*)

Wrapper around `opt_einsum.contract()` that optionally uses Pyro's cheap optimizer and optionally caches contraction paths.

Parameters **cache_path** (*bool*) – whether to cache the contraction path. Defaults to True.

ubersum(*equation*, **operands*, ***kwargs*)

Generalized batched sum-product algorithm via tensor message passing.

This generalizes *contract()* in two ways:

1. Multiple outputs are allowed, and intermediate results can be shared.
2. Inputs and outputs can be batched along symbols given in `batch_dims`; reductions along `batch_dims` are product reductions.

The best way to understand this function is to try the examples below, which show how *ubersum()* calls can be implemented as multiple calls to *contract()* (which is generally more expensive).

To illustrate multiple outputs, note that the following are equivalent:

```
z1, z2, z3 = ubersum('ab,bc->a,b,c', x, y)  # multiple outputs

backend = 'pyro.ops.einsum.torch_log'
z1 = contract('ab,bc->a', x, y, backend=backend)
z2 = contract('ab,bc->b', x, y, backend=backend)
z3 = contract('ab,bc->c', x, y, backend=backend)
```

To illustrate batched inputs, note that the following are equivalent:

```
assert len(x) == 3 and len(y) == 3
z = ubersum('ab,ai,bi->b', w, x, y, batch_dims='i')

z = contract('ab,a,a,a,b,b,b->b', w, *x, *y, backend=backend)
```

When a sum dimension *a* always appears with a batch dimension *i*, then *a* corresponds to a distinct symbol for each slice of *a*. Thus the following are equivalent:

```
assert len(x) == 3 and len(y) == 3
z = ubersum('ai,ai->', x, y, batch_dims='i')

z = contract('a,b,c,a,b,c->', *x, *y, backend=backend)
```

When such a sum dimension appears in the output, it must be accompanied by all of its batch dimensions, e.g. the following are equivalent:

```
assert len(x) == 3 and len(y) == 3
z = ubersum('abi,abi->bi', x, y, batch_dims='i')

z0 = contract('ab,ac,ad,ab,ac,ad->b', *x, *y, backend=backend)
z1 = contract('ab,ac,ad,ab,ac,ad->c', *x, *y, backend=backend)
z2 = contract('ab,ac,ad,ab,ac,ad->d', *x, *y, backend=backend)
z = torch.stack([z0, z1, z2])
```

Note that each batch slice through the output is multilinear in all batch slices through all inptus, thus e.g. batch matrix multiply would be implemented *without* `batch_dims`, so the following are all equivalent:

```
xy = ubersum('abc,acd->abd', x, y, batch_dims='')
xy = torch.stack([xa.mm(ya) for xa, ya in zip(x, y)])
xy = torch.bmm(x, y)
```

Among all valid equations, some computations are polynomial in the sizes of the input tensors and other computations are exponential in the sizes of the input tensors. This function raises `NotImplementedError` whenever the computation is exponential.

Parameters

- **equation** (*str*) – An einsum equation, optionally with multiple outputs.
- **operands** (*torch.Tensor*) – A collection of tensors.
- **batch_dims** (*str*) – An optional string of batch dims.
- **cache** (*dict*) – An optional `shared_intermediates()` cache.

- **modulo_total** (*bool*) – Optionally allow ubersum to arbitrarily scale each result batch, which can significantly reduce computation. This is safe to set whenever each result batch denotes a nonnormalized probability distribution whose total is not of interest.

Returns a tuple of tensors of requested shape, one entry per output.

Return type tuple

Raises

- **ValueError** – if tensor sizes mismatch or an output requests a batched dim without that dim's batch dims.
- **NotImplementedError** – if contraction would have cost exponential in the size of any input tensor.

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 AutoCallable.<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 plates, 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*)

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`.

..note:: This class does MAP inference in constrained space.

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 *get_posterior()* 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

get_posterior(**args*, ***kwargs*)

Returns the posterior distribution.

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)
```

get_posterior(**args*, ***kwargs*)
Returns a MultivariateNormal posterior distribution.

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)
```

get_posterior(**args*, ***kwargs*)
Returns a diagonal Normal posterior distribution.

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 `cov_diag` is initialized to 1/2 and the `cov_factor` is intialized randomly such that `cov_factor.matmul(cov_factor.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_cov_factor", torch.randn(latent_dim, rank)))
pyro.param("auto_cov_diag", 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

`get_posterior`(**args*, ***kwargs*)
Returns a LowRankMultivariateNormal posterior distribution.

11.9 AutoIAFNormal

`class AutoIAFNormal`(*model*, *hidden_dim=None*, *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
- **`prefix`** (*`str`*) – a prefix that will be prefixed to all param internal sites

`get_posterior`(**args*, ***kwargs*)
Returns a diagonal Normal posterior distribution transformed by `InverseAutoregressiveFlow`.

11.10 AutoLaplaceApproximation

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

Laplace approximation (quadratic approximation) approximates the posterior math:*log p(z | x)* by a multivariate normal distribution in the unconstrained space. Under the hood, it uses Delta distributions to construct a MAP guide over the entire (unconstrained) latent space. Its covariance is given by the inverse of the hessian of $-\log p(x, z)$ at the MAP point of z.

Usage:

```
delta_guide = AutoLaplaceApproximation(model)
svi = SVI(model, delta_guide, ...)
# ...then train the delta_guide...
guide = delta_guide.laplace_approximation()
```

By default the mean vector is initialized to zero. 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))
```

get_posterior(**args*, ***kwargs*)
Returns a Delta posterior distribution for MAP inference.

laplace_approximation(**args*, ***kwargs*)
Returns a *AutoMultivariateNormal* instance whose posterior's *loc* and *scale_tril* are given by Laplace approximation.

11.11 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()
```

name_count (*fn=None*)

`name_count` is a very simple autonaming scheme that simply appends a suffix "__" plus a counter to any name that appears multiple tims in an execution. Only duplicate instances of a name get a suffix; the first instance is not modified.

Example:

```
>>> @name_count
... def model():
...     for i in range(3):
...         pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "x" in poutine.trace(model).get_trace()
>>> assert "x__1" in poutine.trace(model).get_trace()
>>> assert "x__2" in poutine.trace(model).get_trace()
```

`name_count` also composes with *`scope()`* by adding a suffix to duplicate scope entrances:

Example:

```
>>> @name_count
... def model():
...     for i in range(3):
...         with pyro.contrib.autoname.scope(prefix="a"):
...             pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
>>> assert "a__1/x" in poutine.trace(model).get_trace()
>>> assert "a__2/x" in poutine.trace(model).get_trace()
```

Example:

```
>>> @name_count
... def model():
...     with pyro.contrib.autoname.scope(prefix="a"):
...         for i in range(3):
...             pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
```

(continues on next page)

(continued from previous page)

```
>>> assert "a/x__1" in poutine.trace(model).get_trace()
>>> assert "a/x__2" 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 NameCountMessenger

Bases: *pyro.poutine.messenger.Messenger*

NameCountMessenger is the implementation of *pyro.contrib.autoname.name_count()*

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()
```

name_count (*fn=None*)

`name_count` is a very simple autonaming scheme that simply appends a suffix *"__"* plus a counter to any name that appears multiple tims in an execution. Only duplicate instances of a name get a suffix; the first instance is not modified.

Example:

```
>>> @name_count
... def model():
...     for i in range(3):
...         pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "x" in poutine.trace(model).get_trace()
>>> assert "x__1" in poutine.trace(model).get_trace()
>>> assert "x__2" in poutine.trace(model).get_trace()
```

`name_count` also composes with *`scope()`* by adding a suffix to duplicate scope entrances:

Example:

```
>>> @name_count
... def model():
...     for i in range(3):
...         with pyro.contrib.autoname.scope(prefix="a"):
...             pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
>>> assert "a__1/x" in poutine.trace(model).get_trace()
>>> assert "a__2/x" in poutine.trace(model).get_trace()
```

Example:

```
>>> @name_count
... def model():
...     with pyro.contrib.autoname.scope(prefix="a"):
...         for i in range(3):
...             pyro.sample("x", dist.Bernoulli(0.5))
...
>>> assert "a/x" in poutine.trace(model).get_trace()
>>> assert "a/x__1" in poutine.trace(model).get_trace()
>>> assert "a/x__2" in poutine.trace(model).get_trace()
```

CHAPTER 13

Bayesian Neural Networks

13.1 HiddenLayer

class HiddenLayer(*X=None*, *A_mean=None*, *A_scale=None*, *non_linearity=<function relu>*, *KL_factor=1.0*, *A_prior_scale=1.0*, *include_hidden_bias=True*, *weight_space_sampling=False*)

This distribution is a basic building block in a Bayesian neural network. It represents a single hidden layer, i.e. an affine transformation applied to a set of inputs *X* followed by a non-linearity. The uncertainty in the weights is encoded in a Normal variational distribution specified by the parameters *A_scale* and *A_mean*. The so-called 'local reparameterization trick' is used to reduce variance (see reference below). In effect, this means the weights are never sampled directly; instead one samples in pre-activation space (i.e. before the non-linearity is applied). Since the weights are never directly sampled, when this distribution is used within the context of variational inference, care must be taken to correctly scale the KL divergence term that corresponds to the weight matrix. This term is folded into the *log_prob* method of this distributions.

In effect, this distribution encodes the following generative process:

A ~ Normal(A_mean, A_scale) output ~ non_linearity(AX)

Parameters

- **X** (*torch.Tensor*) – B x D dimensional mini-batch of inputs
- **A_mean** (*torch.Tensor*) – D x H dimensional specifiying weight mean
- **A_scale** (*torch.Tensor*) – D x H dimensional (diagonal covariance matrix) specifying weight uncertainty
- **non_linearity** (*callable*) – a callable that specifies the non-linearity used. defaults to ReLU.
- **KL_factor** (*float*) – scaling factor for the KL divergence. prototypically this is equal to the size of the mini-batch divided by the size of the whole dataset. defaults to *1.0*.
- **A_prior** (*float or torch.Tensor*) – the prior over the weights is assumed to be normal with mean zero and scale factor *A_prior*. default value is 1.0.

- **include_hidden_bias** (*bool*) – controls whether the activations should be augmented with a 1, which can be used to incorporate bias terms. defaults to *True*.
- **weight_space_sampling** (*bool*) – controls whether the local reparameterization trick is used. this is only intended to be used for internal testing. defaults to *False*.

Reference:

Kingma, Diederik P., Tim Salimans, and Max Welling. "Variational dropout and the local reparameterization trick." Advances in Neural Information Processing Systems. 2015.

CHAPTER 14

Generalised Linear Mixed Models

The *pyro.contrib.glmm* module provides models and guides for generalised linear mixed models (GLMM). It also includes the Normal-inverse-gamma family.

To create a classical Bayesian linear model, use:

```
from pyro.contrib.glmm import known_covariance_linear_model

# Note: coef is a p-vector, observation_sd is a scalar
# Here, p=1 (one feature)
model = known_covariance_linear_model(coef_mean=torch.tensor([0.]),
                                      coef_sd=torch.tensor([10.]),
                                      observation_sd=torch.tensor(2.))

# An n x p design tensor
# Here, n=2 (two observations)
design = torch.tensor(torch.tensor([[1.], [-1.]]))

model(design)
```

A non-linear link function may be introduced, for instance:

```
from pyro.contrib.glmm import logistic_regression_model

# No observation_sd is needed for logistic models
model = logistic_regression_model(coef_mean=torch.tensor([0.]),
                                  coef_sd=torch.tensor([10.]))
```

Random effects may be incorporated as regular Bayesian regression coefficients. For random effects with a shared covariance matrix, see `pyro.contrib.glmm.lmer_model()`.

CHAPTER 15

Gaussian Processes

See the Gaussian Processes tutorial for an introduction.

15.1 Models

15.1.1 GPModel

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

Bases: *pyro.contrib.gp.parameterized.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 = "GPR/RBF/lengthscale"
>>> for trace, _ in mcmc_run._traces():
...     posterior_ls_trace.append(trace.nodes[ls_name]["value"])
```

- Using a variational inference on the pair *model()*, *guide()*:

```
>>> optimizer = torch.optim.Adam(gpr.parameters(), lr=0.01)
>>> loss_fn = pyro.infer.TraceMeanField_ELBO().differentiable_loss
>>>
>>> for i in range(1000):
...     svi.step()  # doctest: +SKIP
...     optimizer.zero_grad()
...     loss = loss_fn(gpr.model, gpr.guide)  # doctest: +SKIP
...     loss.backward()  # doctest: +SKIP
...     optimizer.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.

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)
>>> optimizer = torch.optim.Adam(vsgp.parameters(), lr=0.01)
>>> loss_fn = pyro.infer.TraceMeanField_ELBO().differentiable_loss
>>> batched_X, batched_y = X.split(split_size=10), y.split(split_size=10)
>>> for Xi, yi in zip(batched_X, batched_y):
...     optimizer.zero_grad()
...     vsgp.set_data(Xi, yi)
...     svi.step()  # doctest: +SKIP
...     loss = loss_fn(vsgp.model, vsgp.guide)  # doctest: +SKIP
...     loss.backward()  # doctest: +SKIP
...     optimizer.step()
```

- Making a two-layer Gaussian Process stochastic function:

```
>>> gpr1 = gp.models.GPRegression(X, None, kernel)
>>> Z, _ = gpr1.model()
>>> gpr2 = gp.models.GPRegression(Z, y, kernel)
>>> 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.

15.1.2 GPRegression

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

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.

model()

guide()

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)

iter_sample(*noiseless=True*)

Iteratively constructs a sample from the Gaussian Process posterior.

Recall that at test input points X_{new}, the posterior is multivariate Gaussian distributed with mean and covariance matrix given by *forward()*.

This method samples lazily from this multivariate Gaussian. The advantage of this approach is that later query points can depend upon earlier ones. Particularly useful when the querying is to be done by an optimisation routine.

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

Parameters **noiseless** (*bool*) – A flag to decide if we want to add sampling noise to the samples beyond the noise inherent in the GP posterior.

Returns sampler

Return type function

15.1.3 SparseGPRegression

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

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 Nyström 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()

guide()

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)

15.1.4 VariationalGP

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

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.

model()

guide()

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)

15.1.5 VariationalSparseGP

class VariationalSparseGP(*X*, *y*, *kernel*, *Xu*, *likelihood*, *mean_function=None*, *latent_shape=None*, *num_data=None*, *whiten=False*, *jitter=1e-06*)

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.

model()

guide()

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)

15.1.6 GPLVM

class GPLVM(*base_model*)

Bases: *pyro.contrib.gp.parameterized.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 parameter:
>>> X_init = torch.zeros(150, 2)
>>> # Then, define a Gaussian Process model with input X_init 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
>>> gpmodule = gp.models.SparseGPRegression(X_init, y, kernel, Xu)
>>> # Finally, wrap gpmodule by GPLVM, optimize, and get the "learned"␣
↪mean of X:
>>> gplvm = gp.models.GPLVM(gpmodule)
>>> gp.util.train(gplvm)  # doctest: +SKIP
>>> X = gplvm.X
```

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`.

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.

15.2 Kernels

15.2.1 Kernel

class Kernel(*input_dim*, *active_dims=None*)

Bases: *pyro.contrib.gp.parameterized.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.

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

15.2.2 Brownian

class Brownian(*input_dim*, *variance=None*, *active_dims=None*)

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*)

15.2.3 Combination

class Combination(*kern0*, *kern1*)

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.

15.2.4 Constant

class Constant(*input_dim*, *variance=None*, *active_dims=None*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Implementation of Constant kernel:

$$k(x, z) = \sigma^2.$$

forward(*X*, *Z=None*, *diag=False*)

15.2.5 Coregionalize

class Coregionalize(*input_dim*, *rank=None*, *components=None*, *diagonal=None*, *active_dims=None*)
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*)

15.2.6 Cosine

class Cosine(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*)
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*)

15.2.7 DotProduct

class DotProduct(*input_dim*, *variance=None*, *active_dims=None*)
Bases: pyro.contrib.gp.kernels.kernel.Kernel

Base class for kernels which are functions of $x \cdot z$.

15.2.8 Exponent

class Exponent(*kern*)
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*)

15.2.9 Exponential

class Exponential(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*)
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*)

15.2.10 Isotropy

class Isotropy(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=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.

15.2.11 Linear

class Linear(*input_dim*, *variance=None*, *active_dims=None*)
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*)

15.2.12 Matern32

class Matern32(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*)

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*)

15.2.13 Matern52

class Matern52(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*)

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*)

15.2.14 Periodic

class Periodic(*input_dim*, *variance=None*, *lengthscale=None*, *period=None*, *active_dims=None*)

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*)

15.2.15 Polynomial

class Polynomial(*input_dim*, *variance=None*, *bias=None*, *degree=1*, *active_dims=None*)
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*)

15.2.16 Product

class Product(*kern0*, *kern1*)
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*)

15.2.17 RBF

class RBF(*input_dim*, *variance=None*, *lengthscale=None*, *active_dims=None*)
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*)

15.2.18 RationalQuadratic

class RationalQuadratic(*input_dim*, *variance=None*, *lengthscale=None*, *scale_mixture=None*, *active_dims=None*)
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*)

15.2.19 Sum

class Sum(*kern0*, *kern1*)
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*)

15.2.20 Transforming

class Transforming(*kern*)
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.

15.2.21 VerticalScaling

class VerticalScaling(*kern*, *vscaling_fn*)
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*)

15.2.22 Warping

class Warping(*kern*, *iwarping_fn=None*, *owarping_coef=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*)

15.2.23 WhiteNoise

class WhiteNoise(*input_dim*, *variance=None*, *active_dims=None*)
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*)

15.3 Likelihoods

15.3.1 Likelihood

class Likelihood
Bases: *pyro.contrib.gp.parameterized.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

15.3.2 Binary

class Binary(*response_function=None*)
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

$$f \sim \mathbb{N}\bowtie\backslash\gg\supset\ll(f_{loc}, f_{var}),$$
$$y \sim \mathbb{B}\backslash\ltimes\bowtie\cong\ll\ll\sqsupset(f).$$

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

15.3.3 Gaussian

class Gaussian(*variance=None*)
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

$$y \sim \mathbb{N}\bowtie\backslash\gg\supset\ll(f_{loc}, f_{var} + \epsilon),$$

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

15.3.4 MultiClass

class MultiClass(*num_classes*, *response_function=None*)
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

$$f \sim \mathbb{N}(f_{loc}, f_{var}),$$
$$y \sim \mathbb{C}(\text{softmax}(f)).$$

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

15.3.5 Poisson

class Poisson(*response_function=None*)
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

15.4 Parameterized

class Parameterized

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

A wrapper of `torch.nn.Module` whose parameters can be set constraints, set priors.

Under the hood, we move parameters to a buffer store and create "root" parameters which are used to generate that parameter's value. For example, if we set a contraint to a parameter, an "unconstrained" parameter will be created, and the constrained value will be transformed from that "unconstrained" parameter.

By default, when we set a prior to a parameter, an auto Delta guide will be created. We can use the method *autoguide()* to setup other auto guides. To fix a parameter to a specific value, it is enough to turn off its "root" parameters' `requires_grad` flags.

Example:

```
>>> class Linear(Parameterized):
...     def __init__(self, a, b):
...         super(Linear, self).__init__()
...         self.a = Parameter(a)
...         self.b = Parameter(b)
...
...     def forward(self, x):
...         return self.a * x + self.b
...
>>> linear = Linear(torch.tensor(1.), torch.tensor(0.))
>>> linear.set_constraint("a", constraints.positive)
>>> linear.set_prior("b", dist.Normal(0, 1))
>>> linear.autoguide("b", dist.Normal)
>>> assert "a_unconstrained" in dict(linear.named_parameters())
>>> assert "b_loc" in dict(linear.named_parameters())
>>> assert "b_scale_unconstrained" in dict(linear.named_parameters())
>>> assert "a" in dict(linear.named_buffers())
>>> assert "b" in dict(linear.named_buffers())
>>> assert "b_scale" in dict(linear.named_buffers())
```

Note that 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.

set_constraint (*name*, *constraint*)
Sets the constraint of an existing parameter.

Parameters

- **name** (*str*) – Name of the parameter.
- **constraint** (*Constraint*) – A PyTorch constraint. See `torch.distributions.constraints` for a list of constraints.

set_prior (*name*, *prior*)
Sets the constraint of an existing parameter.

Parameters

- **name** (*str*) – Name of the parameter.
- **prior** (*Distribution*) – A Pyro prior distribution.

autoguide (*name*, *dist_constructor*)
Sets an autoguide for an existing parameter with name `name` (mimic the behavior of module *pyro.contrib.autoguide*).

..note:: *dist_constructor* should be one of *Delta*, *Normal*, and *MultivariateNormal*. More distribution constructor will be supported in the future if needed.

Parameters

- **name** (*str*) – Name of the parameter.
- **dist_constructor** – A `Distribution` constructor.

set_mode (*mode*)
Sets `mode` of this object to be able to use its parameters in stochastic functions. If `mode="model"`, a parameter will get its value from its prior. If `mode="guide"`, the value will be drawn from its guide.

..note:: This method automatically sets `mode` for submodules which belong to *Parameterized* class.

Parameters **mode** (*str*) – Either "model" or "guide".

mode

15.5 Util

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)

train(*gpmodule*, *optimizer=None*, *loss_fn=None*, *retain_graph=None*, *num_steps=1000*)

A helper to optimize parameters for a GP module.

Parameters

- **gpmodule** (*GPModel*) – A GP module.
- **optimizer** (*Optimizer*) – A PyTorch optimizer instance. By default, we use Adam with `lr=0.01`.
- **loss_fn** (*callable*) – A loss function which takes inputs are `gpmodule.model`, `gpmodule.guide`, and returns ELBO loss. By default, `loss_fn=TraceMeanField_ELBO().differentiable_loss`.
- **retain_graph** (*bool*) – An optional flag of `torch.autograd.backward`.
- **num_steps** (*int*) – Number of steps to run SVI.

Returns a list of losses during the training procedure

Return type list

CHAPTER 16

Mini Pyro

This file contains a minimal implementation of the Pyro Probabilistic Programming Language. The API (method signatures, etc.) match that of the full implementation as closely as possible. This file is independent of the rest of Pyro, with the exception of the `pyro.distributions` module.

An accompanying example that makes use of this implementation can be found at examples/minipyro.py.

class Adam(*optim_args*)
Bases: `object`

__call__(*params*)

class Messenger(*fn=None*)
Bases: `object`

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

postprocess_message(*msg*)

process_message(*msg*)

class PlateMessenger(*fn*, *size*, *dim*)
Bases: *pyro.contrib.minipyro.Messenger*

process_message(*msg*)

class SVI(*model*, *guide*, *optim*, *loss*)
Bases: `object`

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

apply_stack(*msg*)

class block(*fn=None*, *hide_fn=<function block.<lambda>>*)
Bases: *pyro.contrib.minipyro.Messenger*

process_message(*msg*)

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

get_param_store()

param(*name*, *init_value=None*)

plate(*name*, *size*, *dim*)

class replay(*fn*, *guide_trace*)
Bases: *pyro.contrib.minipyro.Messenger*

process_message(*msg*)

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

class trace(*fn=None*)
Bases: *pyro.contrib.minipyro.Messenger*

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

postprocess_message(*msg*)

CHAPTER 17

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`.

17.1 Expected Information Gain

vi_ape (*model*, *design*, *observation_labels*, *target_labels*, *vi_parameters*, *is_parameters*, *y_dist=None*)

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.

- **target_labels** (*list*) – A subset of the sample sites over which the posterior entropy is to be measured.
- **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.
- **y_dist** (`pyro.distributions.Distribution`) – (optional) the distribution assumed for the response variable Y

Returns Loss function estimate

Return type *torch.Tensor*

naive_rainforth_eig(*model*, *design*, *observation_labels*, *target_labels=None*, *N=100*, *M=10*, *M_prime=None*)

Naive Rainforth (i.e. Nested Monte Carlo) estimate of the expected information gain (EIG). The estimate is

$$\frac{1}{N}\sum_{n=1}^{N}\log p(y_n|\theta_n,d) - \log\left(\frac{1}{M}\sum_{m=1}^{M}p(y_n|\theta_m,d)\right)$$

Monte Carlo estimation is attempted for the $\log p(y|\theta,d)$ term if the parameter *M_prime* is passed. Otherwise, it is assumed that that $\log p(y|\theta,d)$ can safely be read from the model itself.

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.
- **target_labels** (*list*) – A subset of the sample sites over which the posterior entropy is to be measured.
- **N** (*int*) – Number of outer expectation samples.
- **M** (*int*) – Number of inner expectation samples for *p(y|d)*.
- **M_prime** (*int*) – Number of samples for *p(y | theta, d)* if required.

Returns EIG estimate

Return type *torch.Tensor*

donsker_varadhan_eig(*model*, *design*, *observation_labels*, *target_labels*, *num_samples*, *num_steps*, *T*, *optim*, *return_history=False*, *final_design=None*, *final_num_samples=None*)

Donsker-Varadhan estimate of the expected information gain (EIG).

The Donsker-Varadhan representation of EIG is

$$\sup_{T} E_{p(y,\theta|d)}[T(y,\theta)] - \log E_{p(y|d)p(\theta)}[\exp(T(\bar{y},\bar{\theta}))]$$

where T is any (measurable) function.

This methods optimises the loss function over a pre-specified class of functions *T*.

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.
- **target_labels** (*list*) – A subset of the sample sites over which the posterior entropy is to be measured.
- **num_samples** (*int*) – Number of samples per iteration.
- **num_steps** (*int*) – Number of optimisation steps.
- **or torch.nn.Module T** (*function*) – optimisable function T for use in the Donsker-Varadhan loss function.
- **optim** (*pyro.optim.Optim*) – Optimiser to use.
- **return_history** (*bool*) – If *True*, also returns a tensor giving the loss function at each step of the optimisation.
- **final_design** (*torch.Tensor*) – The final design tensor to evaluate at. If *None*, uses *design*.
- **final_num_samples** (*int*) – The number of samples to use at the final evaluation, If *None, uses 'num_samples*.

Returns EIG estimate, optionally includes full optimisatio history

Return type *torch.Tensor* or *tuple*

barber_agakov_ape(*model*, *design*, *observation_labels*, *target_labels*, *num_samples*, *num_steps*, *guide*, *optim*, *return_history=False*, *final_design=None*, *final_num_samples=None*)

Barber-Agakov estimate of average posterior entropy (APE).

The Barber-Agakov representation of APE is

$sup_q E_{p(y,\theta|d)}[\log q(\theta|y,d)]$

where q is any distribution on θ.

This method optimises the loss over a given guide family *guide* representing q.

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.
- **target_labels** (*list*) – A subset of the sample sites over which the posterior entropy is to be measured.
- **num_samples** (*int*) – Number of samples per iteration.
- **num_steps** (*int*) – Number of optimisation steps.
- **guide** (*function*) – guide family for use in the (implicit) posterior estimation. The parameters of *guide* are optimised to maximise the Barber-Agakov objective.
- **optim** (*pyro.optim.Optim*) – Optimiser to use.
- **return_history** (*bool*) – If *True*, also returns a tensor giving the loss function at each step of the optimisation.

- **final_design** (*torch.Tensor*) – The final design tensor to evaluate at. If *None*, uses *design*.
- **final_num_samples** (*int*) – The number of samples to use at the final evaluation, If *None, uses 'num_samples*.

Returns EIG estimate, optionally includes full optimisatio history

Return type *torch.Tensor* or *tuple*

class EwmaLog(*alpha*)

Logarithm function with exponentially weighted moving average for gradients.

For input *inputs* this function return `inputs.log()`. However, it computes the gradient as

$$\frac{\sum_{t=0}^{T-1} \alpha^t}{\sum_{t=0}^{T-1} \alpha^t x_{T-t}}$$

where x_t are historical input values passed to this function, x_T being the most recently seen value.

This gradient may help with numerical stability when the sequence of inputs to the function form a convergent sequence.

CHAPTER 18

Tracking

18.1 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

18.2 Distributions

class EKFDistribution(*x0*, *P0*, *dynamic_model*, *measurement_cov*, *time_steps=1*, *dt=1.0*, *validate_args=None*)

Distribution over EKF states. See *EKFState*. Currently only supports *log_prob*.

Parameters

- **x0** (*torch.Tensor*) – PV tensor (mean)
- **P0** (*torch.Tensor*) – covariance
- **dynamic_model** – *DynamicModel* object
- **measurement_cov** (*torch.Tensor*) – measurement covariance
- **time_steps** (*int*) – number time step
- **dt** (*torch.Tensor*) – time step

filter_states(*value*)

Returns the ekf states given measurements

Parameters **value** (*torch.Tensor*) – measurement means of shape *(time_steps, event_shape)*

log_prob(*value*)

Returns the joint log probability of the innovations of a tensor of measurements

Parameters **value** (*torch.Tensor*) – measurement means of shape *(time_steps, event_shape)*

18.3 Dynamic Models

class DynamicModel(*dimension*, *dimension_pv*, *num_process_noise_parameters=None*)

Dynamic model interface.

Parameters

- **dimension** – native state dimension.
- **dimension_pv** – PV state dimension.
- **num_process_noise_parameters** – process noise parameter space dimension. This for UKF applications. Can be left as `None` for EKF and most other filters.

dimension

Native state dimension access.

dimension_pv

PV state dimension access.

num_process_noise_parameters

Process noise parameters space dimension access.

forward(*x*, *dt*, *do_normalization=True*)

Integrate native state `x` over time interval `dt`.

Parameters

- **x** – current native state. If the DynamicModel is non-differentiable, be sure to handle the case of `x` being augmented with process noise parameters.

- **dt** – time interval to integrate over.
- **do_normalization** – whether to perform normalization on output, e.g., mod'ing angles into an interval.

Returns Native state x integrated dt into the future.

geodesic_difference(*x1*, *x0*)

Compute and return the geodesic difference between 2 native states. This is a generalization of the Euclidean operation `x1 - x0`.

Parameters

- **x1** – native state.
- **x0** – native state.

Returns Geodesic difference between native states `x1` and `x2`.

mean2pv(*x*)

Compute and return PV state from native state. Useful for combining state estimates of different types in IMM (Interacting Multiple Model) filtering.

Parameters **x** – native state estimate mean.

Returns PV state estimate mean.

cov2pv(*P*)

Compute and return PV covariance from native covariance. Useful for combining state estimates of different types in IMM (Interacting Multiple Model) filtering.

Parameters **P** – native state estimate covariance.

Returns PV state estimate covariance.

process_noise_cov(*dt=0.0*)

Compute and return process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q). For a DifferentiableDynamicModel, this is the covariance of the native state `x` resulting from stochastic integration (for use with EKF). Otherwise, it is the covariance directly of the process noise parameters (for use with UKF).

process_noise_dist(*dt=0.0*)

Return a distribution object of state displacement from the process noise distribution over a time interval.

Parameters **dt** – time interval that process noise accumulates over.

Returns `MultivariateNormal`.

class DifferentiableDynamicModel(*dimension*, *dimension_pv*, *num_process_noise_parameters=None*)

DynamicModel for which state transition Jacobians can be efficiently calculated, usu. analytically or by automatic differentiation.

jacobian(*dt*)

Compute and return native state transition Jacobian (F) over time interval `dt`.

Parameters **dt** – time interval to integrate over.

Returns Read-only Jacobian (F) of integration map (f).

class Ncp(*dimension*, *sv2*)

NCP (Nearly-Constant Position) dynamic model. May be subclassed, e.g., with CWNV (Continuous White Noise Velocity) or DWNV (Discrete White Noise Velocity).

Parameters

- **dimension** – native state dimension.
- **sv2** – variance of velocity. Usually chosen so that the standard deviation is roughly half of the max velocity one would ever expect to observe.

forward(*x*, *dt*, *do_normalization=True*)
Integrate native state x over time interval dt.

Parameters

- **x** – current native state. If the DynamicModel is non-differentiable, be sure to handle the case of x being augmented with process noise parameters.
- **dt** – time interval to integrate over. do_normalization: whether to perform normalization on output, e.g., mod'ing angles into an interval. Has no effect for this subclass.

Returns Native state x integrated dt into the future.

mean2pv(*x*)
Compute and return PV state from native state. Useful for combining state estimates of different types in IMM (Interacting Multiple Model) filtering.

Parameters **x** – native state estimate mean.

Returns PV state estimate mean.

cov2pv(*P*)
Compute and return PV covariance from native covariance. Useful for combining state estimates of different types in IMM (Interacting Multiple Model) filtering.

Parameters **P** – native state estimate covariance.

Returns PV state estimate covariance.

jacobian(*dt*)
Compute and return cached native state transition Jacobian (F) over time interval dt.

Parameters **dt** – time interval to integrate over.

Returns Read-only Jacobian (F) of integration map (f).

process_noise_cov(*dt=0.0*)
Compute and return cached process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q) of the native state x resulting from stochastic integration (for use with EKF).

class Ncv(*dimension*, *sa2*)
NCV (Nearly-Constant Velocity) dynamic model. May be subclassed, e.g., with CWNA (Continuous White Noise Acceleration) or DWNA (Discrete White Noise Acceleration).

Parameters

- **dimension** – native state dimension.
- **sa2** – variance of acceleration. Usually chosen so that the standard deviation is roughly half of the max acceleration one would ever expect to observe.

forward(*x*, *dt*, *do_normalization=True*)
Integrate native state x over time interval dt.

Parameters

- **x** – current native state. If the DynamicModel is non-differentiable, be sure to handle the case of x being augmented with process noise parameters.
- **dt** – time interval to integrate over.
- **do_normalization** – whether to perform normalization on output, e.g., mod'ing angles into an interval. Has no effect for this subclass.

Returns Native state x integrated dt into the future.

mean2pv(*x*)
Compute and return PV state from native state. Useful for combining state estimates of different types in IMM (Interacting Multiple Model) filtering.

Parameters **x** – native state estimate mean.

Returns PV state estimate mean.

cov2pv(*P*)
Compute and return PV covariance from native covariance. Useful for combining state estimates of different types in IMM (Interacting Multiple Model) filtering.

Parameters **P** – native state estimate covariance.

Returns PV state estimate covariance.

jacobian(*dt*)
Compute and return cached native state transition Jacobian (F) over time interval dt.

Parameters **dt** – time interval to integrate over.

Returns Read-only Jacobian (F) of integration map (f).

process_noise_cov(*dt=0.0*)
Compute and return cached process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q) of the native state x resulting from stochastic integration (for use with EKF).

class NcpContinuous(*dimension*, *sv2*)
NCP (Nearly-Constant Position) dynamic model with CWNV (Continuous White Noise Velocity).

References: "Estimation with Applications to Tracking and Navigation" by Y. Bar- Shalom et al, 2001, p.269.

Parameters

- **dimension** – native state dimension.
- **sv2** – variance of velocity. Usually chosen so that the standard deviation is roughly half of the max velocity one would ever expect to observe.

process_noise_cov(*dt=0.0*)
Compute and return cached process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q) of the native state x resulting from stochastic integration (for use with EKF).

class NcvContinuous(*dimension*, *sa2*)
NCV (Nearly-Constant Velocity) dynamic model with CWNA (Continuous White Noise Acceleration).

References: "Estimation with Applications to Tracking and Navigation" by Y. Bar- Shalom et al, 2001, p.269.

Parameters

- **dimension** – native state dimension.
- **sa2** – variance of acceleration. Usually chosen so that the standard deviation is roughly half of the max acceleration one would ever expect to observe.

process_noise_cov(*dt=0.0*)
Compute and return cached process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q) of the native state x resulting from stochastic integration (for use with EKF).

class NcpDiscrete(*dimension*, *sv2*)
NCP (Nearly-Constant Position) dynamic model with DWNV (Discrete White Noise Velocity).

Parameters

- **dimension** – native state dimension.
- **sv2** – variance of velocity. Usually chosen so that the standard deviation is roughly half of the max velocity one would ever expect to observe.

References: "Estimation with Applications to Tracking and Navigation" by Y. Bar- Shalom et al, 2001, p.273.

process_noise_cov(*dt=0.0*)
Compute and return cached process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q) of the native state *x* resulting from stochastic integration (for use with EKF).

class NcvDiscrete(*dimension*, *sa2*)
NCV (Nearly-Constant Velocity) dynamic model with DWNA (Discrete White Noise Acceleration).

Parameters

- **dimension** – native state dimension.
- **sa2** – variance of acceleration. Usually chosen so that the standard deviation is roughly half of the max acceleration one would ever expect to observe.

References: "Estimation with Applications to Tracking and Navigation" by Y. Bar- Shalom et al, 2001, p.273.

process_noise_cov(*dt=0.0*)
Compute and return cached process noise covariance (Q).

Parameters **dt** – time interval to integrate over.

Returns Read-only covariance (Q) of the native state *x* resulting from stochastic integration (for use with EKF). (Note that this Q, modulo numerical error, has rank *dimension/2*. So, it is only positive semi-definite.)

18.4 Extended Kalman Filter

class EKFState(*dynamic_model*, *mean*, *cov*, *time=None*, *frame_num=None*)
State-Centric EKF (Extended Kalman Filter) for use with either an NCP (Nearly-Constant Position) or NCV

(Nearly-Constant Velocity) target dynamic model. Stores a target dynamic model, state estimate, and state time. Incoming `Measurement` provide sensor information for updates.

Warning: For efficiency, the dynamic model is only shallow-copied. Make a deep copy outside as necessary to protect against unexpected changes.

Parameters

- **dynamic_model** – target dynamic model.
- **mean** – mean of target state estimate.
- **cov** – covariance of target state estimate.
- **time** – time of state estimate.

dynamic_model
Dynamic model access.

dimension
Native state dimension access.

mean
Native state estimate mean access.

cov
Native state estimate covariance access.

dimension_pv
PV state dimension access.

mean_pv
Compute and return cached PV state estimate mean.

cov_pv
Compute and return cached PV state estimate covariance.

time
Continuous State time access.

frame_num
Discrete State time access.

predict(*dt=None*, *destination_time=None*, *destination_frame_num=None*)
Use dynamic model to predict (aka propagate aka integrate) state estimate in-place.

Parameters

- **dt** – time to integrate over. The state time will be automatically incremented this amount unless you provide `destination_time`. Using `destination_time` may be preferable for prevention of roundoff error accumulation.
- **destination_time** – optional value to set continuous state time to after integration. If this is not provided, then *destination_frame_num* must be.
- **destination_frame_num** – optional value to set discrete state time to after integration. If this is not provided, then *destination_frame_num* must be.

innovation(*measurement*)
Compute and return the innovation that a measurement would induce if it were used for an update, but

don't actually perform the update. Assumes state and measurement are time-aligned. Useful for computing Chi^2 stats and likelihoods.

Parameters **measurement** – measurement

Returns Innovation mean and covariance of hypothetical update.

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

log_likelihood_of_update(*measurement*)
Compute and return the likelihood of a potential update, but don't actually perform the update. Assumes state and measurement are time- aligned. Useful for gating and calculating costs in assignment problems for data association.

Param measurement.

Returns Likelihood of hypothetical update.

update(*measurement*)
Use measurement to update state estimate in-place and return innovation. The innovation is useful, e.g., for evaluating filter consistency or updating model likelihoods when the EKFState is part of an IMMFState.

Param measurement.

Returns EKF State, Innovation mean and covariance.

18.5 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)
>>> # even though c is within 2radius of a
>>> lsh.nearby('a') # doctest: +SKIP
{'b'}
>>> lsh.nearby('b') # doctest: +SKIP
{'a', 'c'}
>>> lsh.remove('b')
>>> lsh.nearby('a') # doctest: +SKIP
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

18.6 Measurements

class Measurement(*mean*, *cov*, *time=None*, *frame_num=None*)
Gaussian measurement interface.

Parameters

- **mean** – mean of measurement distribution.
- **cov** – covariance of measurement distribution.
- **time** – continuous time of measurement. If this is not provided, *frame_num* must be.
- **frame_num** – discrete time of measurement. If this is not provided, *time* must be.

dimension
Measurement space dimension access.

mean
Measurement mean (`z` in most Kalman Filtering literature).

cov
Noise covariance (`R` in most Kalman Filtering literature).

time
Continuous time of measurement.

frame_num
Discrete time of measurement.

geodesic_difference(*z1*, *z0*)
Compute and return the geodesic difference between 2 measurements. This is a generalization of the Euclidean operation `z1 - z0`.

Parameters

- **z1** – measurement.
- **z0** – measurement.

Returns Geodesic difference between `z1` and `z2`.

class DifferentiableMeasurement(*mean*, *cov*, *time=None*, *frame_num=None*)
Interface for Gaussian measurement for which Jacobians can be efficiently calculated, usu. analytically or by automatic differentiation.

jacobian(*x=None*)
Compute and return Jacobian (H) of measurement map (h) at target PV state `x` .

Parameters **x** – PV state. Use default argument `None` when the Jacobian is not state-dependent.

Returns Read-only Jacobian (H) of measurement map (h).

class PositionMeasurement(*mean*, *cov*, *time=None*, *frame_num=None*)
Full-rank Gaussian position measurement in Euclidean space.

Parameters

- **mean** – mean of measurement distribution.
- **cov** – covariance of measurement distribution.
- **time** – time of measurement.

jacobian(*x=None*)
Compute and return Jacobian (H) of measurement map (h) at target PV state `x` .

Parameters **x** – PV state. The default argument `None` may be used in this subclass since the Jacobian is not state-dependent.

Returns Read-only Jacobian (H) of measurement map (h).

CHAPTER 19

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