Pyro Documentation

Uber AI Labs

Oct 15, 2020
## Pyro Core

### 1 Getting Started

### 2 Primitives

### 3 Inference
- 3.1 SVI
- 3.2 ELBO
- 3.3 Importance
- 3.4 Reweighted Wake-Sleep
- 3.5 Sequential Monte Carlo
- 3.6 Stein Methods
- 3.7 Likelihood free methods
- 3.8 Discrete Inference
- 3.9 Inference Utilities
- 3.10 MCMC
- 3.11 Automatic Guide Generation
- 3.12 Reparameterizers

### 4 Distributions
- 4.1 PyTorch Distributions
- 4.2 Pyro Distributions
- 4.3 Transforms
- 4.4 TransformModules
- 4.5 Transform Factories
- 4.6 Constraints

### 5 Parameters
- 5.1 ParamStore

### 6 Neural Networks
- 6.1 Pyro Modules
- 6.2 AutoRegressiveNN
- 6.3 DenseNN
- 6.4 ConditionalAutoRegressiveNN
- 6.5 ConditionalDenseNN

### 7 Optimization
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Pyro Optimizers</td>
<td>147</td>
</tr>
<tr>
<td>7.2</td>
<td>PyTorch Optimizers</td>
<td>150</td>
</tr>
<tr>
<td>7.3</td>
<td>Higher-Order Optimizers</td>
<td>151</td>
</tr>
<tr>
<td>8</td>
<td>Poutine (Effect handlers)</td>
<td>153</td>
</tr>
<tr>
<td>8.1</td>
<td>Handlers</td>
<td>153</td>
</tr>
<tr>
<td>8.2</td>
<td>Trace</td>
<td>161</td>
</tr>
<tr>
<td>8.3</td>
<td>Runtime</td>
<td>163</td>
</tr>
<tr>
<td>8.4</td>
<td>Utilities</td>
<td>164</td>
</tr>
<tr>
<td>8.5</td>
<td>Messengers</td>
<td>165</td>
</tr>
<tr>
<td>9</td>
<td>Miscellaneous Ops</td>
<td>177</td>
</tr>
<tr>
<td>9.1</td>
<td>Utilities for HMC</td>
<td>177</td>
</tr>
<tr>
<td>9.2</td>
<td>Newton Optimizers</td>
<td>179</td>
</tr>
<tr>
<td>9.3</td>
<td>Special Functions</td>
<td>181</td>
</tr>
<tr>
<td>9.4</td>
<td>Tensor Utilities</td>
<td>182</td>
</tr>
<tr>
<td>9.5</td>
<td>Tensor Indexing</td>
<td>184</td>
</tr>
<tr>
<td>9.6</td>
<td>Tensor Contraction</td>
<td>186</td>
</tr>
<tr>
<td>9.7</td>
<td>Gaussian Contraction</td>
<td>188</td>
</tr>
<tr>
<td>9.8</td>
<td>Statistical Utilities</td>
<td>191</td>
</tr>
<tr>
<td>9.9</td>
<td>State Space Model and GP Utilities</td>
<td>194</td>
</tr>
<tr>
<td>10</td>
<td>Automatic Name Generation</td>
<td>195</td>
</tr>
<tr>
<td>10.1</td>
<td>Named Data Structures</td>
<td>197</td>
</tr>
<tr>
<td>10.2</td>
<td>Scoping</td>
<td>199</td>
</tr>
<tr>
<td>11</td>
<td>Bayesian Neural Networks</td>
<td>203</td>
</tr>
<tr>
<td>11.1</td>
<td>HiddenLayer</td>
<td>203</td>
</tr>
<tr>
<td>12</td>
<td>Causal Effect VAE</td>
<td>205</td>
</tr>
<tr>
<td>12.1</td>
<td>CEVAE Class</td>
<td>205</td>
</tr>
<tr>
<td>12.2</td>
<td>CEVAE Components</td>
<td>207</td>
</tr>
<tr>
<td>12.3</td>
<td>Utilities</td>
<td>208</td>
</tr>
<tr>
<td>13</td>
<td>Easy Custom Guides</td>
<td>211</td>
</tr>
<tr>
<td>13.1</td>
<td>EasyGuide</td>
<td>211</td>
</tr>
<tr>
<td>13.2</td>
<td>easy_guide</td>
<td>212</td>
</tr>
<tr>
<td>13.3</td>
<td>Group</td>
<td>212</td>
</tr>
<tr>
<td>14</td>
<td>Epidemiology</td>
<td>215</td>
</tr>
<tr>
<td>14.1</td>
<td>Base Compartmental Model</td>
<td>215</td>
</tr>
<tr>
<td>14.2</td>
<td>Example Models</td>
<td>220</td>
</tr>
<tr>
<td>14.3</td>
<td>Distributions</td>
<td>227</td>
</tr>
<tr>
<td>15</td>
<td>Pyro Examples</td>
<td>231</td>
</tr>
<tr>
<td>15.1</td>
<td>Datasets</td>
<td>231</td>
</tr>
<tr>
<td>15.2</td>
<td>Utilities</td>
<td>232</td>
</tr>
<tr>
<td>16</td>
<td>Forecasting</td>
<td>233</td>
</tr>
<tr>
<td>16.1</td>
<td>Forecaster Interface</td>
<td>233</td>
</tr>
<tr>
<td>16.2</td>
<td>Evaluation</td>
<td>237</td>
</tr>
<tr>
<td>17</td>
<td>Funsor-based Pyro</td>
<td>239</td>
</tr>
<tr>
<td>17.1</td>
<td>Effect handlers</td>
<td>239</td>
</tr>
</tbody>
</table>
18 Gaussian Processes
  18.1 Models ................................................................. 244
  18.2 Kernels ..................................................................... 255
  18.3 Likelihoods ............................................................... 261
  18.4 Parameterized .......................................................... 264
  18.5 Util .......................................................................... 265

19 Minipyro ..................................................................... 267
  19.1 Mini Pyro ................................................................. 267

20 Optimal Experiment Design
  20.1 Expected Information Gain ........................................... 270
  20.2 Generalised Linear Mixed Models ............................... 276

21 Random Variables .......................................................... 279
  21.1 Random Variable ...................................................... 279

22 Time Series .................................................................... 281
  22.1 Abstract Models .......................................................... 281
  22.2 Gaussian Processes .................................................... 282
  22.3 Linear Gaussian State Space Models .......................... 285

23 Tracking ....................................................................... 287
  23.1 Data Association ........................................................ 287
  23.2 Distributions ............................................................. 290
  23.3 Dynamic Models ....................................................... 290
  23.4 Extended Kalman Filter .............................................. 294
  23.5 Hashing ................................................................. 296
  23.6 Measurements ........................................................ 298

24 Indices and tables ........................................................... 301

Python Module Index .......................................................... 303

Index ................................................................................. 305
CHAPTER 1

Getting Started

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

get_param_store()
Returns the ParamStore

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

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 (str) – name of parameter
• init_tensor (torch.Tensor or callable) – initial tensor or lazy callable that returns a tensor. For large tensors, it may be cheaper to write e.g. lambda: torch.randn(100000), which will only be evaluated on the initial statement.
• constraint (torch.distributions.constraints.Constraint) – torch constraint, defaults to constraints.real.
• event_dim (int) – (optional) number of rightmost dimensions unrelated to batching. Dimension to the left of this will be considered batch dimensions; if the param statement is inside a subsampled plate, then corresponding batch dimensions of the parameter will be correspondingly subsampled. If unspecified, all dimensions will be considered event dims and no subsampling will be performed.

Returns parameter

Return type torch.Tensor

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

**factor**(*name, log_factor*)

Factor statement to add arbitrary log probability factor to a probabilistic model.

**Parameters**

- **name** (*str*) – Name of the trivial sample
- **log_factor** (*torch.Tensor*) – A possibly batched log probability factor.

**deterministic**(*name, value, event_dim=None*)

EXPERIMENTAL Deterministic statement to add a Delta site with name *name* and value *value* to the trace. This is useful when we want to record values which are completely determined by their parents. For example:

```python
x = sample("x", dist.Normal(0, 1))
x2 = deterministic("x2", x ** 2)
```

**Note:** The site does not affect the model density. This currently converts to a `sample()` statement, but may change in the future.

**Parameters**

- **name** (*str*) – Name of the site.
- **value** (*torch.Tensor*) – Value of the site.
- **event_dim** (*int*) – Optional event dimension, defaults to `value.ndim`.

**subsample**(*data, event_dim*)

EXPERIMENTAL Subsampling statement to subsample data based on enclosing plates.

This is typically called on arguments to `model()` when subsampling is performed automatically by plates by passing either the subsample or subsample_size kwarg. For example the following are equivalent:

```python
# Version 1. using indexing
def model(data):
    with pyro.plate("data", len(data), subsample_size=10, dim=-data.dim()) as ind:
        data = data[ind]
    # ...

# Version 2. using pyro.subsample()
def model(data):
    with pyro.plate("data", len(data), subsample_size=10, dim=-data.dim()):
        data = pyro.subsample(data, event_dim=0)
    # ...
```

**Parameters**

- **data** (*Tensor*) – A tensor of batched data.
**event_dim** (*int*) – The event dimension of the data tensor. Dimensions to the left are considered batch dimensions.

**Returns** A subsampled version of data

**Return type** Tensor

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

**Bases:** pyro.poutine.plate_messenger.PlateMessenger

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:

```python
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:

```python
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 reusable context manager yielding a single 1-dimensional `torch.Tensor` of indices.

Examples:

```python
>>> # 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_{i}'.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.

**class iarange** (*args, **kwargs*)
Bases: `pyro.primitives.plate`

**class irange** (*args, **kwargs*)
Bases: `pyro.poutine.subsample_messenger.SubsampleMessenger`

**plate_stack** (prefix, sizes, rightmost_dim=-1)
Create a contiguous stack of `plate`s with dimensions:

```
rightmost_dim - len(sizes), ..., rightmost_dim
```

**Parameters**

• **prefix** (*str*) – Name prefix for plates.

• **sizes** (*iterable*) – An iterable of plate sizes.
rightmost_dim (int) – The rightmost dim, counting from the right.

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)

Warning: The random_module primitive is deprecated, and will be removed in a future release. Use PyroModule instead to create Bayesian modules from torch.nn.Module instances. See the Bayesian Regression tutorial for an example.

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

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

barrier (data)
EXPERIMENTAL Ensures all values in data are ground, rather than lazy funsor values. This is useful in combination with pyro.poutine.collapse().

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.

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.

trace (fn=None, ignore_warnings=False, jit_options=None)
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:

```python
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()
```

Parameters

- **fn (callable)** – The function to be traced.
- **ignore_warnings (bool)** – Whether to ignore jit warnings.
- **jit_options (dict)** – Optional dict of options to pass to `torch.jit.trace()`, e.g. {"optimize": False}. 

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.

## 3.1 SVI

```python
class SVI(model, guide, optim, loss, loss_and_grads=None, num_samples=0, 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` (`PyroOptim`) – a wrapper 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` – (DEPRECATED) the number of samples for Monte Carlo posterior approximation
- `num_steps` – (DEPRECATED) 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.

```python
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)

Warning: This method is deprecated, and will be removed in a future release. For inference, use step() directly, and for predictions, use the Predictive class.

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

### 3.2 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, jit options=None, retain_graph=None, tail adaptive beta=-1.0)

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 this is True, all torch.jit.TracerWarning will be ignored. Defaults to False.

- **jit options** (bool) – Optional dict of options to pass to torch.jit.trace(), e.g. {"check_trace": True}.

- **retain graph** (bool) – Whether to retain autograd graph during an SVI step. Defaults to None (False).
Pyro Documentation

- **tail_adaptive_beta** (*float*) – Exponent beta with \(-1.0 \leq \beta < 0.0\) for use with `TraceTailAdaptive_ELBO`.

References


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, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0*)

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


def 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.

def 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

def 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, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0*)

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\( (\text{model}, \text{guide}, *, \text{args}, **\text{kwargs}) \)

differentiable_loss\( (\text{model}, \text{guide}, *, \text{args}, **\text{kwargs}) \)

loss_and_grads\( (\text{model}, \text{guide}, *, \text{args}, **\text{kwargs}) \)

class TraceGraph_ELBO\( (\text{num\_particles}=1, \ \text{max\_plate\_nesting}=\text{inf}, \ \text{max\_iarange\_nesting}=\text{None}, \ \text{vectorize\_particles}=\text{False}, \ \text{strict\_enumeration\_warning}=\text{True}, \ \text{ignore\_jit\_warnings}=\text{False}, \ \text{jit\_options}=\text{None}, \ \text{retain\_graph}=\text{None}, \ \text{tail\_adaptive\_beta}=-1.0) \)

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 two kinds of conditional dependency information are used to reduce variance:

- the sequential order of samples (\( z \) is sampled after \( y \) \( \Rightarrow \) \( y \) does not depend on \( z \))
- plate generators

References


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

loss\( (\text{model}, \text{guide}, *, \text{args}, **\text{kwargs}) \)

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\( (\text{model}, \text{guide}, *, \text{args}, **\text{kwargs}) \)

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\( (\text{num\_particles}=1, \ \text{max\_plate\_nesting}=\text{inf}, \ \text{max\_iarange\_nesting}=\text{None}, \ \text{vectorize\_particles}=\text{False}, \ \text{strict\_enumeration\_warning}=\text{True}, \ \text{ignore\_jit\_warnings}=\text{False}, \ \text{jit\_options}=\text{None}, \ \text{retain\_graph}=\text{None}, \ \text{tail\_adaptive\_beta}=-1.0) \)

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\( (\text{model}, \text{guide}, *, \text{args}, **\text{kwargs}) \)
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, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0)

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 in the guide.

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 a 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, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0)

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, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0)

**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, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0)

**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.

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

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

class **TraceTailAdaptive_ELBO**

```python
class TraceTailAdaptive_ELBO(num_particles=1, max_plate_nesting=inf, max iarange nesting=None, vectorize_particles=False, strict enumeration warning=True, ignore jit warnings=False, jit options=None, retain_graph=None, tail adaptive beta=-1.0)
```

Bases: `pyro.infer.trace_elbo.Trace_ELBO`

Interface for Stochastic Variational Inference with an adaptive f-divergence as described in ref. [1]. Users should specify `num_particles > 1` and `vectorize_particles==True`. The argument `tail_adaptive_beta` can be specified to modify how the adaptive f-divergence is constructed. See reference for details.

Note that this interface does not support computing the varational objective itself; rather it only supports computing gradients of the variational objective. Consequently, one might want to use another SVI interface (e.g. RenyiELBO) in order to monitor convergence.

Note that this interface only supports models in which all the latent variables are fully reparameterized. It also does not support data subsampling.


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

It is not necessary to estimate the tail-adaptive f-divergence itself in order to compute the corresponding gradients. Consequently the loss method is left unimplemented.

class **RenyiELBO**

```python
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.

**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:

3.2. ELBO
[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.

**class TraceTMC_ELBO** *(num_particles=1, max_plate_nesting=inf, max_iarange_nesting=None, vectorize_particles=False, strict Enumeration warning=True, ignore_jit_warnings=False, jit_options=None, retain_graph=None, tail_adaptive_beta=-1.0) *

Bases: pyro.infer.elbo.ELBO

A trace-based implementation of Tensor Monte Carlo [1] by way of Tensor Variable Elimination [2] that supports: - local parallel sampling over any sample site in the model or guide - exhaustive enumeration over any sample site in the model or guide

To take multiple samples, mark the site with infer={'enumerate': 'parallel', 'num_samples': N}. To configure all sites in a model or guide at once, use config_enumerate() . To enumerate or sample a sample site in the model, mark the site 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.

References


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

Returns a differentiable estimate of the marginal log-likelihood

Return type torch.Tensor

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

Computes a differentiable TMC estimate using num_particles many samples (particles). The result should be infinitely differentiable (as long as underlying derivatives have been implemented).

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

**loss_and_grads** *(model, guide, *args, **kwargs) *
3.3 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.

psis_diagnostic(model, guide, *args, **kwargs)
    Computes the Pareto tail index k for a model/guide pair using the technique described in [1], which builds on previous work in [2]. If $0 < k < 0.5$ the guide is a good approximation to the model posterior, in the sense described in [1]. If $0.5 \leq k \leq 0.7$, the guide provides a suboptimal approximation to the posterior, but may still be useful in practice. If $k > 0.7$ the guide program provides a poor approximation to the full posterior, and caution should be used when using the guide. Note, however, that a guide may be a poor fit to the full posterior while still yielding reasonable model predictions. If $k < 0.0$ the importance weights corresponding to the model and guide appear to be bounded from above; this would be a bizarre outcome for a guide trained via ELBO maximization. Please see [1] for a more complete discussion of how the tail index $k$ should be interpreted.

Please be advised that a large number of samples may be required for an accurate estimate of $k$.

Note that we assume that the model and guide are both vectorized and have static structure. As is canonical in Pyro, the args and kwargs are passed to the model and guide.

References
[1] ‘Yes, but Did It Work?: Evaluating Variational Inference.’ Yuling Yao, Aki Vehtari, Daniel Simpson, Andrew Gelman

Parameters
    • model (callable) – the model program.
    • guide (callable) – the guide program.
    • num_particles (int) – the total number of times we run the model and guide in order to compute the diagnostic. defaults to 1000.
    • max_simultaneous_particles – the maximum number of simultaneous samples drawn from the model and guide. defaults to num_particles. num_particles must be divisible by max_simultaneous_particles. compute the diagnostic. defaults to 1000.
    • max_plate_nesting (int) – optional bound on max number of nested pyro.plate() contexts in the model/guide. defaults to 7.

Returns float the PSIS diagnostic $k$

vectorized_importance_weights(model, guide, *args, **kwargs)
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 1)
- **max_plate_nesting** (int) – Bound on max number of nested `pyro.plate()` contexts.
- **normalized** (bool) – set to True to return self-normalized importance weights

Returns returns a `(num_samples,)`-shaped tensor of importance weights and the model and guide traces that produced them

Vectorized computation of importance weights for models with static structure:

```python
log_weights, model_trace, guide_trace = \
    vectorized_importance_weights(model, guide, *args,
        num_samples=1000,
        max_plate_nesting=4,
        normalized=False)
```

3.4 Reweighted Wake-Sleep

class ReweightedWakeSleep(num_particles=2, insomnia=1.0, model_has_params=True, num_sleep_particles=None, vectorize_particles=True, max_plate_nesting=inf, strict EnumerationWarning=True)

Bases: `pyro.infer.elbo.ELBO`

An implementation of Reweighted Wake Sleep following reference [1].

**Note:** Sampling and log_prob evaluation asymptotic complexity:

1) **Using wake-theta and/or wake-phi** $O(num\_particles)$ samples from guide, $O(num\_particles)$ log_prob evaluations of model and guide

2) **Using sleep-phi** $O(num\_sleep\_particles)$ samples from model, $O(num\_sleep\_particles)$ log_prob evaluations of guide

if 1) and 2) are combined, $O(num\_particles)$ samples from the guide, $O(num\_sleep\_particles)$ from the model, $O(num\_particles + num\_sleep\_particles)$ log_prob evaluations of the guide, and $O(num\_particles)$ evaluations of the model

**Note:** This is particularly useful for models with stochastic branching, as described in [2].

**Note:** This returns _two_ losses, one each for (a) the model parameters (`theta`), computed using the `iwae` objective, and (b) the guide parameters (`phi`), computed using (a combination of) the `csis` objective and a self-normalized importance-sampled version of the `csis` objective.
Note: In order to enable computing the sleep-phi terms, the guide program must have its observations explicitly passed in through the keyworded argument `observations`. Where the value of the observations is unknown during definition, such as for amortized variational inference, it may be given a default argument as `observations=None`, and the correct value supplied during learning through `svi.step(observations=...)`.

**Warning:** Mini-batch training is not supported yet.

**Parameters**

- `num_particles (int)`: The number of particles/samples used to form the objective (gradient) estimator. Default is 2.
- `insomnia`: The scaling between the wake-phi and sleep-phi terms. Default is 1.0 [wake-phi]
- `model_has_params (bool)`: Indicate if model has learnable params. Useful in avoiding extra computation when running in pure sleep mode [csis]. Default is True.
- `num_sleep_particles (int)`: The number of particles used to form the sleep-phi estimator. Matches `num_particles` by default.
- `vectorize_particles (bool)`: Whether the traces should be vectorised across `num_particles`. Default is True.
- `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:**


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

Returns returns model loss and guide loss

Return type float, float

Computes the re-weighted wake-sleep estimators for the model (wake-theta) and the guide (insomnia * wake-phi + (1 - insomnia) * sleep-phi).

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

Returns returns model loss and guide loss

Return type float

Computes the RWS estimators for the model (wake-theta) and the guide (wake-phi). Performs backward as appropriate on both, using num_particle many samples/particles.
3.5 Sequential Monte Carlo

exception SMCFailed
    Bases: ValueError

Exception raised when SMCFilter fails to find any hypothesis with nonzero probability.

class SMCFilter (model, guide, num_particles, max_plate_nesting, *, ess_threshold=0.5)
    Bases: object

SMCFilter is the top-level interface for filtering via sequential monte carlo.

The model and guide should be objects with two methods: .init(state, ...) and .step(state, ...), intended to be called first with init(), then with step() repeatedly. These two methods should have the same signature as SMCFilter's init() and step() of this class, but with an extra first argument state that should be used to store all tensors that depend on sampled variables. The state will be a dict-like object, SMCState, with arbitrary keys and torch.Tensor values. Models can read and write state but guides can only read from it.

Inference complexity is \(O(len(state) \times num\_time\_steps)\), so to avoid quadratic complexity in Markov models, ensure that state has fixed size.

Parameters

- **model**(object) – probabilistic model with init and step methods
- **guide**(object) – guide used for sampling, with init and step methods
- **num_particles**(int) – The number of particles used to form the distribution.
- **max_plate_nesting**(int) – Bound on max number of nested pyro.plate() contexts.
- **ess_threshold**(float) – Effective sample size threshold for deciding when to importance resample: resampling occurs when \(ess < ess\_threshold \times num\_particles\).

get_empirical()

Returns a marginal distribution over all state tensors.

Return type a dictionary with keys which are latent variables and values which are Empirical objects.

init(*args, **kwargs)

Perform any initialization for sequential importance resampling. Any args or kwargs are passed to the model and guide

step(*args, **kwargs)

Take a filtering step using sequential importance resampling updating the particle weights and values while resampling if desired. Any args or kwargs are passed to the model and guide

class SMCState(num_particles)
    Bases: dict

Dictionary-like object to hold a vectorized collection of tensors to represent all state during inference with SMCFilter. During inference, the SMCFilter resample these tensors.

Keys may have arbitrary hashable type. Values must be torch.Tensors.

Parameters num_particles(int) –
3.6 Stein Methods

class IMQSteinKernel (alpha=0.5, beta=-0.5, bandwidth_factor=None)
Bases: pyro.infer.svgd.SteinKernel

An IMQ (inverse multi-quadratic) kernel for use in the SVGD inference algorithm [1]. The bandwidth of the kernel is chosen from the particles using a simple heuristic as in reference [2]. The kernel takes the form

$$K(x, y) = (\alpha + ||x - y||^2 / h)^\beta$$

where $\alpha$ and $\beta$ are user-specified parameters and $h$ is the bandwidth.

Parameters

- **alpha** (float) – Kernel hyperparameter, defaults to 0.5.
- **beta** (float) – Kernel hyperparameter, defaults to -0.5.
- **bandwidth_factor** (float) – Optional factor by which to scale the bandwidth, defaults to 1.0.

Variables **bandwidth_factor** (float) – Property that controls the factor by which to scale the bandwidth at each iteration.

References


bandwidth_factor

log_kernel_and_grad (particles)
See pyro.infer.svgd.SteinKernel.log_kernel_and_grad()

class RBFSteinKernel (bandwidth_factor=None)
Bases: pyro.infer.svgd.SteinKernel

A RBF kernel for use in the SVGD inference algorithm. The bandwidth of the kernel is chosen from the particles using a simple heuristic as in reference [1].

Parameters **bandwidth_factor** (float) – Optional factor by which to scale the bandwidth, defaults to 1.0.

Variables **bandwidth_factor** (float) – Property that controls the factor by which to scale the bandwidth at each iteration.

References


bandwidth_factor

log_kernel_and_grad (particles)
See pyro.infer.svgd.SteinKernel.log_kernel_and_grad()

class SVGD (model, kernel, optim, num_particles, max_plate_nesting, mode='univariate')
Bases: object

A basic implementation of Stein Variational Gradient Descent as described in reference [1].

Parameters
• **model** – The model (callable containing Pyro primitives). Model must be fully vectorized and may only contain continuous latent variables.

• **kernel** – a SVGD compatible kernel like `RBFSteinKernel`.

• **optim** (`pyro.optim.PyroOptim`) – A wrapper for a PyTorch optimizer.

• **num_particles** (`int`) – The number of particles used in SVGD.

• **max_plate_nesting** (`int`) – The max number of nested `pyro.plate()` contexts in the model.

• **mode** (`str`) – Whether to use a Kernelized Stein Discrepancy that makes use of multivariate test functions (as in [1]) or univariate test functions (as in [2]). Defaults to `univariate`.

Example usage:

```python
from pyro.infer import SVGD, RBFSteinKernel
from pyro.optim import Adam

kernel = RBFSteinKernel()
adam = Adam({"lr": 0.1})
svgd = SVGD(model, kernel, adam, num_particles=50, max_plate_nesting=0)

for step in range(500):
    svgd.step(model_arg1, model_arg2)

final_particles = svgd.get_named_particles()
```

References


`get_named_particles()` – Create a dictionary mapping name to vectorized value, of the form `{name: tensor}`. The leading dimension of each tensor corresponds to particles, i.e. this creates a struct of arrays.

`step(*args, **kwargs)` – Computes the SVGD gradient, passing args and kwargs to the model, and takes a gradient step.

    **Return dict** A dictionary of the form `{name: float}`, where each float is a mean squared gradient. This can be used to monitor the convergence of SVGD.

### class SteinKernel

**Bases:** `object`

Abstract class for kernels used in the SVGD inference algorithm.

`log_kernel_and_grad(particles)` – Compute the component kernels and their gradients.

**Parameters** `particles` – a tensor with shape (N, D)

**Returns** A pair `(log_kernel, kernel_grad)` where `log_kernel` is a (N, N, D)-shaped tensor equal to the logarithm of the kernel and `kernel_grad` is a (N, N, D)-shaped tensor where the entry `(n, m, d)` represents the derivative of `log_kernel` w.r.t. `x_{m,d}`, where `x_{m,d}` is the d'th dimension of particle m.

`vectorize(fn, num_particles, max_plate_nesting)`
3.7 Likelihood free methods

```python
class EnergyDistance(beta=1.0, prior_scale=0.0, num_particles=2, max_plate_nesting=inf):
    Bases: object
    Posterior predictive energy distance \([1,2]\) with optional Bayesian regularization by the prior.

Let \(p(x,z) = p(z) p(x|z)\) be the model, \(q(z|x)\) be the guide. Then given data \(x\) and drawing an iid pair of samples \((Z, X)\) and \((Z', X')\) (where \(Z\) is latent and \(X\) is the posterior predictive),

\[
Z \sim q(z|x); \quad X \sim p(x|Z)
\]
\[
Z' \sim q(z|x); \quad X' \sim p(x|Z')
\]

\[
\text{loss} = \mathbb{E}_X \|X - x\|^\beta - \frac{1}{2} \mathbb{E}_{X,X'} \|X - X'\|^\beta - \lambda \mathbb{E}_Z \log p(Z)
\]

This is a likelihood-free inference algorithm, and can be used for likelihoods without tractable density functions. The \(\beta\) energy distance is a robust loss functions, and is well defined for any distribution with finite fractional moment \(\mathbb{E}[\|X\|^\beta]\).

This requires static model structure, a fully reparametrized guide, and reparametrized likelihood distributions in the model. Model latent distributions may be non-reparametrized.

References


Parameters

- **beta** (float) – Exponent \(\beta\) from \([1,2]\). The loss function is strictly proper for distributions with finite \(\beta\)-absolute moment \(\mathbb{E}[^\beta X]\). Thus for heavy tailed distributions \(\beta\) should be small, e.g. for Cauchy distributions, \(\beta < 1\) is strictly proper. Defaults to 1. Must be in the open interval (0,2).

- **prior_scale** (float) – Nonnegative scale for prior regularization. Model parameters are trained only if this is positive. If zero (default), then model log densities will not be computed (guide log densities are never computed).

- **num_particles** (int) – The number of particles/samples used to form the gradient estimators. Must be at least 2.

- **max_plate_nesting** (int) – Optional bound on max number of nested pyro.plate() contexts. If omitted, this will guess a valid value by running the (model,guide) pair once.

___call___(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**(*args, **kwargs)

Not implemented. Added for compatibility with unit tests only.
```

3.8 Discrete Inference

```python
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:

```python
@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).

```python
class TraceEnumSample_ELBO(num_particles=1, max_plate_nesting=inf, max_iarange_nesting=None, vectorize_particles=False, strict Enumeration warning=True, ignore jit warnings=False, jit options=None, retain graph=None, tail adaptive beta=-1.0)
```

This extends TraceEnum_ELBO to make it cheaper to sample from discrete latent states during SVI.

The following are equivalent but the first is cheaper, sharing work between the computations of `loss` and `z`:

```python
# Version 1.
elbo = TraceEnumSample_ELBO(max_plate_nesting=1)
loss = elbo.loss(*args, **kwargs)
z = elbo.sample_saved()

# Version 2.
elbo = TraceEnum_ELBO(max_plate_nesting=1)
loss = elbo.loss(*args, **kwargs)
guide_trace = poutine.trace(guide).get_trace(*args, **kwargs)
z = infer_discrete(poutine.replay(model, guide_trace),
    first_available_dim=-2)(*args, **kwargs)
```

`sample_saved()`

Generate latent samples while reusing work from SVI.step().
3.9 Inference Utilities

**class Predictive**

```python
class Predictive(model, posterior_samples=None, guide=None, num_samples=None, return_sites=(), parallel=False)
```

Bases: torch.nn.modules.module.Module

EXPERIMENTAL class used to construct predictive distribution. The predictive distribution is obtained by running the model conditioned on latent samples from posterior_samples. If a guide is provided, then posterior samples from all the latent sites are also returned.

**Warning:** The interface for the Predictive class is experimental, and might change in the future.

**Parameters**

- **model** – Python callable containing Pyro primitives.
- **posterior_samples** *(dict)* – dictionary of samples from the posterior.
- **guide** *(callable)* – optional guide to get posterior samples of sites not present in posterior_samples.
- **num_samples** *(int)* – number of samples to draw from the predictive distribution. This argument has no effect if posterior_samples is non-empty, in which case, the leading dimension size of samples in posterior_samples is used.
- **return_sites** *(list, tuple, or set)* – sites to return; by default only sample sites not present in posterior_samples are returned.
- **parallel** *(bool)* – predict in parallel by wrapping the existing model in an outermost plate messenger. Note that this requires that the model has all batch dims correctly annotated via plate. Default is False.

**call** *(args, **kwargs)*

Method that calls forward() and returns parameter values of the guide as a tuple instead of a dict, which is a requirement for JIT tracing. Unlike forward(), this method can be traced by torch.jit.trace_module().

**Warning:** This method may be removed once PyTorch JIT tracer starts accepting dict as valid return types. See issue.

**forward** *(args, **kwargs)*

Returns dict of samples from the predictive distribution. By default, only sample sites not contained in posterior_samples are returned. This can be modified by changing the return_sites keyword argument of this Predictive instance.

**Parameters**

- **args** – model arguments.
- **kwargs** – model keyword arguments.

**get_samples** *(args, **kwargs)*

**get_vectorized_trace** *(args, **kwargs)*

Returns a single vectorized trace from the predictive distribution. Note that this requires that the model has all batch dims correctly annotated via plate.
Parameters

- **args** – model arguments.
- **kwargs** – model keyword arguments.

### class EmpiricalMarginal

```
(TracePosterior, 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**

A dictionary of sites' names and their corresponding EmpiricalMarginal distribution.

**Type** OrderedDict

**support** *(flatten=False)*

Gets support of this marginal distribution.

Parameters **flatten** *(bool)* – A flag to decide if we want to flatten batch_shape when the marginal distribution is collected from the posterior with num_chains > 1. Defaults to False.

Returns a dict with keys are sites' names and values are sites' supports.

**Return type** OrderedDict

### 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:


**Parameters**

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

**Returns**

A dictionary containing values of WAIC and its effective number of parameters.

**Return type**

`OrderedDict`

**marginal** *(sites=None)*

Generates the marginal distribution of this posterior.

**Parameters**

- `sites` *(list)* – optional list of sites for which we need to generate the marginal distribution.

**Returns**

A `Marginals` class instance.

**Return type**

`Marginals`

**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, keep_sites=None)*

Bases: `pyro.infer.abstract_infer.TracePosterior`

**Warning:** This class is deprecated and will be removed in a future release. Use the `Predictive` class instead.

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` : trace posterior instance holding samples from the model’s approximate posterior.
- `num_samples` : number of samples to generate.
- `keep_sites` : The sites which should be sampled from posterior distribution (default: all)

**marginal** *(sites=None)*

Gets marginal distribution for this predictive posterior distribution.
3.10 MCMC

3.10.1 MCMC

class MCMC(kernel, num_samples, warmup_steps=None, initial_params=None, num_chains=1, hook_fn=None, mp_context=None, disable_progbar=False, disable_validation=True, transforms=None)

Bases: object

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 the same as 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 _UnarySampler or _MultiSampler.

- **initial_params** (`dict`) – dict containing initial tensors in unconstrained space to initiate the markov chain. The leading dimension’s size must match that of num_chains. If not specified, parameter values will be sampled from the prior.

- **hook_fn** – Python callable that takes in (kernel, samples, stage, i) as arguments. stage is either sample or warmup and i refers to the i’th sample for the given stage. This can be used to implement additional logging, or more generally, run arbitrary code per generated sample.

- **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.

- **disable_validation** (`bool`) – Disables distribution validation check. Defaults to True, disabling validation, since divergent transitions will lead to exceptions. Switch to False to enable validation, or to None to preserve existing global values.

- **transforms** (`dict`) – dictionary that specifies a transform for a sample site with constrained support to unconstrained space.

**diagnostics()**

Gets some diagnostics statistics such as effective sample size, split Gelman-Rubin, or divergent transitions from the sampler.
**get_samples** *(num_samples=None, group_by_chain=False)*

Get samples from the MCMC run, potentially resampling with replacement.

**Parameters**

- **num_samples** *(int)* – Number of samples to return. If *None*, all the samples from an MCMC chain are returned in their original ordering.

- **group_by_chain** *(bool)* – Whether to preserve the chain dimension. If True, all samples will have num_chains as the size of their leading dimension.

**Returns** dictionary of samples keyed by site name.

**run**

Run MCMC to generate samples and populate `self._samples`.

Example usage:

```python
def model(data):
    ...

nuts_kernel = NUTS(model)
mcmc = MCMC(nuts_kernel, num_samples=500)
mcmc.run(data)
samples = mcmc.get_samples()
```

**Parameters**

- **args** – optional arguments taken by `MCMCKernel.setup`.

- **kwargs** – optional keywords arguments taken by `MCMCKernel.setup`.

**summary** *(prob=0.9)*

Prints a summary table displaying diagnostics of samples obtained from posterior. The diagnostics displayed are mean, standard deviation, median, the 90% Credibility Interval, `effective_sample_size()`, `split_gelman_rubin()`.

**Parameters** **prob** *(float)* – the probability mass of samples within the credibility interval.

### 3.10.2 MCMCKernel

class MCMCKernel
    Bases: object

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

    **diagnostics**()
    
    Returns a dict of useful diagnostics after finishing sampling process.

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

    **initial_params**
    
    Returns a dict of initial params (by default, from the prior) to initiate the MCMC run.

    **Returns** dict of parameter values keyed by their name.

    **logging**()
    
    Relevant logging information to be printed at regular intervals of the MCMC run. Returns *None* by default.
Returns  String containing the diagnostic summary. e.g. acceptance rate
Return type  string

**sample** *(params)*
Samples parameters from the posterior distribution, when given existing parameters.

Parameters

- **params** *(dict)* – Current parameter values.
- **time_step** *(int)* – Current time step.

Returns  New parameters from the posterior distribution.

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

Parameters

- **warmup_steps** *(int)* – Number of warmup iterations.
- ***args** – Algorithm specific positional arguments.
- ****kwargs** – Algorithm specific keyword arguments.

### 3.10.3 HMC

class HMC(model=None, potential_fn=None, 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, jit_options=None, ignore_jit_warnings=False, target_accept_prob=0.8, init_strategy=<function init_to_uniform>)

Bases: `pyro.infer.mcmc.mcmc_kernel.MCMCKernel`

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

References


Parameters

- **model** – Python callable containing Pyro primitives.
- **potential_fn** – Python callable calculating potential energy with input is a dict of real support parameters.
- **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 $2\pi$.
- **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.poutine.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.

• **jit_options**(dict) – A dictionary contains optional arguments for `torch.jit.trace()` function.

• **ignore_jit_warnings**(bool) – Flag to ignore warnings from the JIT tracer when `jit_compile=True`. Default is False.

• **target_accept_prob**(float) – Increasing this value will lead to a smaller step size, hence the sampling will be slower and more robust. Default to 0.8.

• **init_strategy**(callable) – A per-site initialization function. See Initialization section for available functions.

**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:

```python
>>> 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
```

```python
>>> hmc_kernel = HMC(model, step_size=0.0855, num_steps=4)
>>> mcmc = MCMC(hmc_kernel, num_samples=500, warmup_steps=100)
>>> mcmc.run(data)
>>> mcmc.get_samples()['beta'].mean(0)  # doctest: +SKIP
tensor([ 0.9819, 1.9258, 2.9737])
```

cleanup()
clear_cache()
diagnostics()
initial_params
inverse_mass_matrix
logging()
mass_matrix_adapter
num_steps
sample(params)
setup(warmup_steps, *args, **kwargs)
step_size

3.10.4 NUTS

class NUTS(model=None, potential_fn=None, 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, jit_options=None,
ignore_jit_warnings=False, target_accept_prob=0.8, max_tree_depth=10,
init_strategy=<function init_to_uniform>)

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


[2] A Conceptual Introduction to Hamiltonian Monte Carlo, Michael Betancourt

[3] Slice Sampling, Radford M. Neal

Parameters

- **model** – Python callable containing Pyro primitives.
- **potential_fn** – Python callable calculating potential energy with input is a dict of real support parameters.
- **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.

• **jit_options** (*dict*) – A dictionary contains optional arguments for `torch.jit.trace()` function.

• **ignore_jit_warnings** (*bool*) – Flag to ignore warnings from the JIT tracer when `jit_compile=True`. Default is False.

• **target_accept_prob** (*float*) – Target acceptance probability of step size adaptation scheme. Increasing this value will lead to a smaller step size, so the sampling will be slower but more robust. Default to 0.8.

• **max_tree_depth** (*int*) – Max depth of the binary tree created during the doubling scheme of NUTS sampler. Default to 10.

• **init_strategy** (*callable*) – A per-site initialization function. See *Initialization* section for available functions.

Example:

```python
>>> 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 = MCMC(nuts_kernel, num_samples=500, warmup_steps=300)
>>> mcmc.run(data)

>>> mcmc.get_samples()['beta'].mean(0)  # doctest: +SKIP
tensor([ 0.9221,  1.9464,  2.9228])
```

**3.10.5 BlockMassMatrix**

```python
class BlockMassMatrix(init_scale=1.0)
  Bases: object

  EXPERIMENTAL This class is used to adapt (inverse) mass matrix and provide useful methods to calculate algebraic terms which involves the mass matrix.

```
The mass matrix will have block structure, which can be specified by using the method `configure()` with the corresponding structured `mass_matrix_shape` arg.

**Parameters**

- **init_scale** (*float*) – initial scale to construct the initial mass matrix.

**configure** *(mass_matrix_shape, adapt_mass_matrix=True, options=*)

Sets up an initial mass matrix.

**Parameters**

- **mass_matrix_shape** (*dict*) – a dict that maps tuples of site names to the shape of the corresponding mass matrix. Each tuple of site names corresponds to a block.
- **adapt_mass_matrix** (*bool*) – a flag to decide whether an adaptation scheme will be used.
- **options** (*dict*) – tensor options to construct the initial mass matrix.

**end_adaptation**()

Updates the current mass matrix using the adaptation scheme.

**inverse_mass_matrix**

**kinetic_grad** *(r)*

Computes the gradient of kinetic energy w.r.t. the momentum `r`. It is equivalent to compute velocity given the momentum `r`.

**Parameters**

- **r** (*dict*) – a dictionary maps site names to a tensor momentum.

**Returns**

a dictionary maps site names to the corresponding gradient

**mass_matrix_size**

A dict that maps site names to the size of the corresponding mass matrix.

**scale** *(r_unscaled, r_prototype)*

Computes $M^{1/2} @ r_{unscaled}$. Note that `r` is generated from a gaussian with scale `mass_matrix_sqrt`. This method will scale it.

**Parameters**

- **r_unscaled** (*dict*) – a dictionary maps site names to a tensor momentum.
- **r_prototype** (*dict*) – a dictionary mapes site names to prototype momentum. Those prototype values are used to get shapes of the scaled version.

**Returns**

a dictionary maps site names to the corresponding tensor

**unscale** *(r)*

Computes $\text{inv}(M^{1/2}) @ r$. Note that `r` is generated from a gaussian with scale `mass_matrix_sqrt`. This method will unscale it.

**Parameters**

- **r** (*dict*) – a dictionary maps site names to a tensor momentum.

**Returns**

a dictionary maps site names to the corresponding tensor

**update** *(z, z_grad)*

Updates the adaptation scheme using the new sample `z` or its grad `z_grad`.

**Parameters**

- **z** (*dict*) – the current value.
- **z_grad** (*dict*) – grad of the current value.
### 3.10.6 Utilities

**initialize_model**(model, model_args=(), model_kwargs={}, transforms=None, max_plate_nesting=None, jit_compile=False, jit_options=None, skip_jit_warnings=False, num_chains=1, init_strategy=<function init_to_uniform>, initial_params=None)

Given a Python callable with Pyro primitives, generates the following model-specific properties needed for inference using HMC/NUTS kernels:

- initial parameters to be sampled using a HMC kernel,
- a potential function whose input is a dict of parameters in unconstrained space,
- transforms to transform latent sites of model to unconstrained space,
- a prototype trace to be used in MCMC to consume traces from sampled parameters.

**Parameters**

- **model** – a Pyro model which contains Pyro primitives.
- **model_args**(tuple) – optional args taken by model.
- **model_kwargs**(dict) – optional kwargs taken by model.
- **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.
- **jit_options**(dict) – A dictionary contains optional arguments for torch.jit.trace() function.
- **ignore_jit_warnings**(bool) – Flag to ignore warnings from the JIT tracer when jit_compile=True. Default is False.
- **num_chains**(int) – Number of parallel chains. If num_chains > 1, the returned initial_params will be a list with num_chains elements.
- **init_strategy**(callable) – A per-site initialization function. See Initialization section for available functions.
- **initial_params**(dict) – dict containing initial tensors in unconstrained space to initiate the markov chain.

**Returns** a tuple of (initial_params, potential_fn, transforms, prototype_trace)

**diagnostics**(samples, group_by_chain=True)

Gets diagnostics statistics such as effective sample size and split Gelman-Rubin using the samples drawn from the posterior distribution.

**Parameters**

- **samples**(dict) – dictionary of samples keyed by site name.
• **group_by_chain**(bool) – If True, each variable in `samples` will be treated as having shape `num_chains x num_samples x sample_shape`. Otherwise, the corresponding shape will be `num_samples x sample_shape` (i.e. without chain dimension).

**Returns** dictionary of diagnostic stats for each sample site.

### 3.11 Automatic Guide Generation

#### 3.11.1 AutoGuide

class **AutoGuide**(model, *, create_plates=None)

**Bases:** pyro.nn.module.PyroModule

Base class for automatic guides.

Derived classes must implement the `forward()` method, with the same *args, **kwargs as the base model.

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

**Parameters**

- **model**(callable) – A pyro model.
- **create_plates**(callable) – An optional function inputing the same *args, **kwargs as `model()` and returning a pyro.plate or iterable of plates. Plates not returned will be created automatically as usual. This is useful for data subsampling.

**call**(args, **kwargs)

Method that calls `forward()` and returns parameter values of the guide as a *tuple* instead of a dict, which is a requirement for JIT tracing. Unlike `forward()`, this method can be traced by `torch.jit.trace_module()`.

**Warning:** This method may be removed once PyTorch JIT tracer starts accepting dict as valid return types. See issue <https://github.com/pytorch/pytorch/issues/27743>_.

**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

**model**

**sample_latent**(**kwargs)

Samples an encoded latent given the same *args, **kwargs as the base model.

#### 3.11.2 AutoGuideList

class **AutoGuideList**(model, *, create_plates=None)

**Bases:** pyro.infer.autoguide.guides.AutoGuide, torch.nn.modules.container.ModuleList

Container class to combine multiple automatic guides.

Example usage:
guide = AutoGuideList(my_model)
guide.append(AutoDiagonalNormal(poutine.block(model, hide="assignment")))
guide.append(AutoDiscreteParallel(poutine.block(model, expose="assignment")))
svi = SVI(model, guide, optim, Trace_ELBO())

Parameters

model (callable) – a Pyro model

add (part)
Depreciated alias for append().

append (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

forward (*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

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

3.11.3 AutoCallable

class AutoCallable (model, guide, median=<function AutoCallable.<lambda>>)
Bases: pyro.infer.autoguide.guides.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.

**forward** (*args, **kwargs*)

### 3.11.4 AutoNormal

**class** `AutoNormal(model, *init_loc_fn=<function init_to_feasible>, init_scale=0.1, create_plates=None)`

Bases: `pyro.infer.autoguide.guides.AutoGuide`

This implementation of `AutoGuide` uses Normal(0, 1) distributions to construct a guide over the entire latent space. The guide does not depend on the model’s *args, **kwargs.*

It should be equivalent to :class: `AutoDiagonalNormal`, but with more convenient site names and with better support for `TraceMeanField_ELBO`.

In `AutoDiagonalNormal`, if your model has N named parameters with dimensions k_i and sum k_i = D, you get a single vector of length D for your mean, and a single vector of length D for sigmas. This guide gives you N distinct normals that you can call by name.

**Usage:**

```python
guide = AutoNormal(model)
svi = SVI(model, guide, ...)
```

**Parameters**

- **model** *(callable)* – A Pyro model.
- **init_loc_fn** *(callable)* – A per-site initialization function. See *Initialization* section for available functions.
- **init_scale** *(float)* – Initial scale for the standard deviation of each (unconstrained transformed) latent variable.
- **create_plates** *(callable)* – An optional function inputing the same *args, **kwargs as model() and returning a `pyro.plate` or iterable of plates. Plates not returned will be created automatically as usual. This is useful for data subsampling.

**forward** (*args, **kwargs*)

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

**Returns** A dict mapping sample site name to sampled value.

**Return type** `dict`

**median** (*args, **kwargs*)

Returns the posterior median value of each latent variable.

**Returns** A dict mapping sample site name to median tensor.

**Return type** `dict`

**quantiles** *(quantiles, *args, **kwargs)*

Returns posterior quantiles each latent variable. Example:

```python
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

3.11.5 AutoDelta

class AutoDelta *(model, init_loc_fn=<function init_to_median>, *, create_plates=None)*

Bases: pyro.infer.autoguide.guides.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:

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

Latent variables are initialized using `init_loc_fn()`. To change the default behavior, create a custom `init_loc_fn()` as described in *Initialization*, for example:

```python
def my_init_fn(site):
    if site['name'] == "level":
        return torch.tensor([-1., 0., 1.])
    if site['name'] == "concentration":
        return torch.ones(k)
    return init_to_sample(site)
```

Parameters

- `model` *(callable)* – A Pyro model.
- `init_loc_fn` *(callable)* – A per-site initialization function. See *Initialization* section for available functions.
- `create_plates` *(callable)* – An optional function inputing the same `*args, **kwargs` as `model()` and returning a `pyro.plate` or iterable of plates. Plates not returned will be created automatically as usual. This is useful for data subsampling.

`forward(*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
3.11.6 AutoContinuous

class AutoContinuous(model, init_loc_fn=<function init_to_median>)
    Bases: pyro.infer.autoguide.guides.AutoGuide

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

This uses torch.distributions.transforms to transform each constrained latent variable to an unconstrained space, then concatenate all variables into a single unconstrained latent variable. Each derived class implements a get_posterior() method returning a distribution over this single unconstrained latent variable.

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

Parameters
• model(callable) – A Pyro model.
• init_loc_fn(callable) – A per-site initialization function. See Initialization section for available functions.

forward(*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_base_dist()
    Returns the base distribution of the posterior when reparameterized as a TransformedDistribution. This should not depend on the model’s *args, **kwargs.

    posterior = TransformedDistribution(self.get_base_dist(), self.get_
→transform(*args, **kwargs))

    Returns TorchDistribution instance representing the base distribution.

get_posterior(*args, **kwargs)
    Returns the posterior distribution.

get_transform(*args, **kwargs)
    Returns the transform applied to the base distribution when the posterior is reparameterized as a TransformedDistribution. This may depend on the model’s *args, **kwargs.

    posterior = TransformedDistribution(self.get_base_dist(), self.get_
→transform(*args, **kwargs))

    Returns a Transform instance.

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:
    ```python
    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.

3.11.7 AutoMultivariateNormal
class AutoMultivariateNormal (model, init_loc_fn=<function init_to_median>, init_scale=0.1)
    Bases: pyro.infer.autoguide.guides.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:
    ```python
guide = AutoMultivariateNormal(model)
svi = SVI(model, guide, ...)
    ```
    By default the mean vector is initialized by init_loc_fn() and the Cholesky factor is initialized to the identity times a small factor.

Parameters

- model (callable) – A generative model.
- init_loc_fn (callable) – A per-site initialization function. See Initialization section for available functions.
- init_scale (float) – Initial scale for the standard deviation of each (unconstrained transformed) latent variable.

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

3.11.8 AutoDiagonalNormal
class AutoDiagonalNormal (model, init_loc_fn=<function init_to_median>, init_scale=0.1)
    Bases: pyro.infer.autoguide.guides.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:

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

By default the mean vector is initialized to zero and the scale is initialized to the identity times a small factor.

Parameters

- `model (callable)` – A generative model.
- `init_loc_fn (callable)` – A per-site initialization function. See Initialization section for available functions.
- `init_scale (float)` – Initial scale for the standard deviation of each (unconstrained transformed) latent variable.

get_base_dist()

get_posterior(*args, **kwargs)

Returns a diagonal Normal posterior distribution.

3.11.9 AutoLowRankMultivariateNormal

class AutoLowRankMultivariateNormal(model, init_loc_fn=<function init_to_median>, init_scale=0.1, rank=None)

Bases: pyro.infer.autoguide.guides.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:

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

By default the cov_diag is initialized to a small constant and the cov_factor is initialized randomly such that on average cov_factor.matmul(cov_factor.t()) has the same scale as cov_diag.

Parameters

- `model (callable)` – A generative model.
- `rank (int or None)` – The rank of the low-rank part of the covariance matrix. Defaults to approximately \( \sqrt{\text{latent dim}} \).
- `init_loc_fn (callable)` – A per-site initialization function. See Initialization section for available functions.
- `init_scale (float)` – Approximate initial scale for the standard deviation of each (unconstrained transformed) latent variable.

get_posterior(*args, **kwargs)

Returns a LowRankMultivariateNormal posterior distribution.

3.11.10 AutoNormalizingFlow

class AutoNormalizingFlow(model, init_transform_fn)

Bases: pyro.infer.autoguide.guides.AutoContinuous
This implementation of *AutoContinuous* uses a Diagonal Normal distribution transformed via a sequence of bijective transforms (e.g. various *TransformModule* subclasses) to construct a guide over the entire latent space. The guide does not depend on the model’s `*args, **kwargs`.

Usage:

```python
transform_init = partial(iterated, block_autoregressive, repeats=2)
guide = AutoNormalizingFlow(model, transform_init)
svi = SVI(model, guide, ...)
```

**Parameters**

- `model (callable)` – a generative model
- `init_transform_fn` – a callable which when provided with the latent dimension returns an instance of *Transform*, or *TransformModule* if the transform has trainable params.

- `get_base_dist ()`
- `get_posterior (*args, **kwargs)`
- `get_transform (*args, **kwargs)`

### 3.11.11 AutoIAFNormal

class AutoIAFNormal(model, hidden_dim=None, init_loc_fn=None, num_transforms=1, **init_transform_kwargs)

Bases: pyro.infer.autoguide.guides.AutoNormalizingFlow

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

Usage:

```python
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
- `init_loc_fn (callable)` – A per-site initialization function. See *Initialization* section for available functions.

**Warning:** This argument is only to preserve backwards compatibility and has no effect in practice.

- `num_transforms (int)` – number of *AffineAutoregressive* transforms to use in sequence.
- `init_transform_kwargs` – other keyword arguments taken by *affine_autoregressive()*.
3.11.12 AutoLaplaceApproximation

class AutoLaplaceApproximation (model, init_loc_fn=<function init_to_median>)
    Bases: pyro.infer.autoguide.guides.AutoContinuous

Laplace approximation (quadratic approximation) approximates the posterior \( \log p(z|x) \) 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:

```python
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 an empirical prior median.

Parameters

- **model** (callable) – a generative model
- **init_loc_fn** (callable) – A per-site initialization function. See Initialization section for available functions.

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.

3.11.13 AutoDiscreteParallel

class AutoDiscreteParallel (model, *, create_plates=None)
    Bases: pyro.infer.autoguide.guides.AutoGuide

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

forward (*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

3.11.14 Initialization

The pyro.infer.autoguide.initialization module contains initialization functions for automatic guides.

The standard interface for initialization is a function that inputs a Pyro trace site dict and returns an appropriately sized value to serve as an initial constrained value for a guide estimate.

**init_to_feasible** (site=None)
    Initialize to an arbitrary feasible point, ignoring distribution parameters.

**init_to_sample** (site=None)
    Initialize to a random sample from the prior.
**init_to_median** (*site=None, num_samples=15*)

Initialize to the prior median; fallback to a feasible point if median is undefined.

**init_to_mean** (*site=None*)

Initialize to the prior mean; fallback to median if mean is undefined.

**init_to_uniform** (*site=None, radius=2*)

Initialize to a random point in the area \((-\text{radius}, \text{radius})\) of unconstrained domain.

**Parameters**

- **radius** (*float*) – specifies the range to draw an initial point in the unconstrained domain.

**init_to_value** (*site=None, values={})

Initialize to the value specified in `values`. We defer to `init_to_uniform()` strategy for sites which do not appear in `values`.

**Parameters**

- **values** (*dict*) – dictionary of initial values keyed by site name.

**init_to_generated** (*site=None, generate=<function <lambda>>*)

Initialize to another initialization strategy returned by the callback `generate` which is called once per model execution.

This is like `init_to_value()` but can produce different (e.g. random) values once per model execution. For example to generate values and return `init_to_value` you could define:

```python
def generate():
    values = {"x": torch.randn(100), "y": torch.rand(5)}
    return init_to_value(values=values)

my_init_fn = init_to_generated(generate=generate)
```

**Parameters**

- **generate** (*callable*) – A callable returning another initialization function, e.g. returning an `init_to_value(values={...})` populated with a dictionary of random samples.

**class InitMessenger** (*init_fn*)

**Bases:** `pyro.poutine.messenger.Messenger`

Initializes a site by replacing `.sample()` calls with values drawn from an initialization strategy. This is mainly for internal use by autoguide classes.

**Parameters**

- **init_fn** (*callable*) – An initialization function.

## 3.12 Reparameterizers

The `pyro.infer.reparam` module contains reparameterization strategies for the `pyro.poutine.handlers.reparam()` effect. These are useful for altering geometry of a poorly-conditioned parameter space to make the posterior better shaped. These can be used with a variety of inference algorithms, e.g. Auto*Normal guides and MCMC.

**class Reparam**

**Base class for reparameterizers.**

**__call__** (*name, fn, obs*)

**Parameters**

- **name** (*str*) – A sample site name.
• `fn(TorchDistribution)` – A distribution.
• `obs(Tensor)` – Observed value or None.

Returns A pair (new_fn, value).

### 3.12.1 Conjugate Updating

class ConjugateReparam(guide)

**Bases:** `pyro.infer.reparam.reparam.Reparam`

EXPERIMENTAL Reparameterize to a conjugate updated distribution.

This updates a prior distribution `fn` using the `conjugate_update()` method. The guide may be either a distribution object or a callable inputting model `*args, **kwargs` and returning a distribution object. The guide may be approximate or learned.

For example consider the model and naive variational guide:

```python
total = torch.tensor(10.)
count = torch.tensor(2.)

def model():
    prob = pyro.sample("prob", dist.Beta(0.5, 1.5))
    pyro.sample("count", dist.Binomial(total, prob), obs=count)
guide = AutoDiagonalNormal(model)  # learns the posterior over prob
```

Instead of using this learned guide, we can hand-compute the conjugate posterior distribution over “prob”, and then use a simpler guide during inference, in this case an empty guide:

```python
reparam_model = poutine.reparam(model, {
    "prob": ConjugateReparam(dist.Beta(1 + count, 1 + total - count))
})

def reparam_guide():
    pass  # nothing remains to be modeled!
```

Parameters

- **guide** (*Distribution or callable*) – A likelihood distribution or a callable returning a guide distribution. Only a few distributions are supported, depending on the prior distribution’s `conjugate_update()` implementation.

__call__(name, fn, obs)

### 3.12.2 Loc-Scale Decentering

class LocScaleReparam(centered=None, shape_params=())

**Bases:** `pyro.infer.reparam.reparam.Reparam`

Generic decentering reparameterizer [1] for latent variables parameterized by `loc` and `scale` (and possibly additional `shape_params`).

This reparameterization works only for latent variables, not likelihoods.

Parameters

- **centered** *(float)* – optional centered parameter. If None (default) learn a per-site per-element centering parameter in \([0, 1]\). If 0, fully decenter the distribution; if 1, preserve the centered distribution unchanged.

- **shape_params** *(tuple or list)* – list of additional parameter names to copy unchanged from the centered to decentered distribution.

__call__(*name, fn, obs*)

### 3.12.3 Transformed Distributions

class **TransformReparam**

   Bases: *pyro.infer.reparam.reparam.Reparam*

Reparameterizer for *pyro.distributions.torch.TransformedDistribution* latent variables.

This is useful for transformed distributions with complex, geometry-changing transforms, where the posterior has simple shape in the space of *base_dist*.

This reparameterization works only for latent variables, not likelihoods.

__call__(*name, fn, obs*)

### 3.12.4 Discrete Cosine Transform

class **DiscreteCosineReparam** *(dim=-1, smooth=0.0, *, experimental_allow_batch=False)*

   Bases: *pyro.infer.reparam.unit_jacobian.UnitJacobianReparam*

Discrete Cosine reparameterizer, using a *DiscreteCosineTransform*.

This is useful for sequential models where coupling along a time-like axis (e.g. a banded precision matrix) introduces long-range correlation. This reparameterizes to a frequency-domain representation where posterior covariance should be closer to diagonal, thereby improving the accuracy of diagonal guides in SVI and improving the effectiveness of a diagonal mass matrix in HMC.

When reparameterizing variables that are approximately continuous along the time dimension, set smooth=1. For variables that are approximately continuously differentiable along the time axis, set smooth=2.

This reparameterization works only for latent variables, not likelihoods.

Parameters

- **dim** *(int)* – Dimension along which to transform. Must be negative. This is an absolute dim counting from the right.

- **smooth** *(float)* – Smoothing parameter. When 0, this transforms white noise to white noise; when 1 this transforms Brownian noise to to white noise; when -1 this transforms violet noise to white noise; etc. Any real number is allowed. [https://en.wikipedia.org/wiki/Colors_of_noise](https://en.wikipedia.org/wiki/Colors_of_noise).

### 3.12.5 Haar Transform

class **HaarReparam** *(dim=-1, flip=False, *, experimental_allow_batch=False)*

   Bases: *pyro.infer.reparam.unit_jacobian.UnitJacobianReparam*

Haar wavelet reparameterizer, using a *HaarTransform*.

3.12. Reparameterizers 47
This is useful for sequential models where coupling along a time-like axis (e.g. a banded precision matrix) introduces long-range correlation. This reparameterizes to a frequency-domain representation where posterior covariance should be closer to diagonal, thereby improving the accuracy of diagonal guides in SVI and improving the effectiveness of a diagonal mass matrix in HMC.

This reparameterization works only for latent variables, not likelihoods.

**Parameters**

- **dim** (int) – Dimension along which to transform. Must be negative. This is an absolute dim counting from the right.
- **flip** (bool) – Whether to flip the time axis before applying the Haar transform. Defaults to false.

### 3.12.6 Unit Jacobian Transforms

**class UnitJacobianReparam**(transform, suffix='transformed', *, experimental_allow_batch=False)

Bases: pyro.infer.reparam.reparam.Reparam

Reparameterizer for Transform objects whose Jacobian determinant is one.

**Parameters**

- **transform** (Transform) – A transform whose Jacobian has determinant 1.
- **suffix** (str) – A suffix to append to the transformed site.

**__call__**(name, fn, obs)

### 3.12.7 StudentT Distributions

**class StudentTReparam**

Bases: pyro.infer.reparam.reparam.Reparam

Auxiliary variable reparameterizer for StudentT random variables.

This is useful in combination with LinearHMMReparam because it allows StudentT processes to be treated as conditionally Gaussian processes, permitting cheap inference via GaussianHMM.

This reparameterizes a StudentT by introducing an auxiliary Gamma variable conditioned on which the result is Normal.

**__call__**(name, fn, obs)

### 3.12.8 Stable Distributions

**class LatentStableReparam**

Bases: pyro.infer.reparam.reparam.Reparam

Auxiliary variable reparameterizer for Stable latent variables.

This is useful in inference of latent Stable variables because the log_prob() is not implemented.

This uses the Chambers-Mallows-Stuck method [1], creating a pair of parameter-free auxiliary distributions (Uniform(-pi/2, pi/2) and Exponential(1)) with well-defined .log_prob() methods, thereby permitting use of reparameterized stable distributions in likelihood-based inference algorithms like SVI and MCMC.
This reparameterization works only for latent variables, not likelihoods. For likelihood-compatible reparameterization see `SymmetricStableReparam` or `StableReparam`.


```python
__call__(name, fn, obs)
```

```py
class SymmetricStableReparam
Bases: pyro.infer.reparam.reparam.Reparam

Auxiliary variable reparameterizer for symmetric `Stable` random variables (i.e. those for which `skew=0`).

This is useful in inference of symmetric `Stable` variables because the `log_prob()` is not implemented.

This reparameterizes a symmetric `Stable` random variable as a totally-skewed (`skew=1`) `Stable` scale mixture of `Normal` random variables. See Proposition 3. of [1] (but note we differ since `Stable` uses Nolan’s continuous S0 parameterization).


```python
__call__(name, fn, obs)
```

```py
class StableReparam
Bases: pyro.infer.reparam.reparam.Reparam

Auxiliary variable reparameterizer for arbitrary `Stable` random variables.

This is useful in inference of non-symmetric `Stable` variables because the `log_prob()` is not implemented.

This reparameterizes a `Stable` random variable as sum of two other stable random variables, one symmetric and the other totally skewed (applying Property 2.3.a of [1]). The totally skewed variable is sampled as in `LatentStableReparam`, and the symmetric variable is decomposed as in `SymmetricStableReparam`.


```python
__call__(name, fn, obs)
```

3.12.9 Hidden Markov Models

```py
class LinearHMMReparam(init=None, trans=None, obs=None)
Bases: pyro.infer.reparam.reparam.Reparam

Auxiliary variable reparameterizer for `LinearHMM` random variables.

This defers to component reparameterizers to create auxiliary random variables conditioned on which the process becomes a `GaussianHMM`. If the observation_dist is a `TransformedDistribution` this reorders those transforms so that the result is a `TransformedDistribution` of `GaussianHMM`.

This is useful for training the parameters of a `LinearHMM` distribution, whose `log_prob()` method is undefined. To perform inference in the presence of non-Gaussian factors such as `Stable()`, `StudentT()` or `LogNormal()`, configure with `StudentTReparam`, `StableReparam`, `SymmetricStableReparam`, etc. component reparameterizers for init, trans, and scale. For example:

```python
hmm = LinearHMM(
    init_dist=Stable(1,0,1,0).expand([2]).to_event(1),
    trans_matrix=torch.eye(2),
    trans_dist=MultivariateNormal(torch.zeros(2), torch.eye(2)),
)
```

(continues on next page)
obs_matrix=torch.eye(2),
obs_dist=TransformedDistribution(
    Stable(1.5,-0.5,1.0).expand([2]).to_event(1),
    ExpTransform())
rep = LinearHMMReparam(init=SymmetricStableReparam(),
    obs=StableReparam())

with poutine.reparam(config={"hmm": rep}):
    pyro.sample("hmm", hmm, obs=data)

Parameters

- **init** (Reparam) – Optional reparameterizer for the initial distribution.
- **trans** (Reparam) – Optional reparameterizer for the transition distribution.
- **obs** (Reparam) – Optional reparameterizer for the observation distribution.

__call__ (name, fn, obs)

3.12.10 Site Splitting

class SplitReparam (sections, dim)
Bases: pyro.infer.reparam.reparam.Reparam.Reparam

Reparameterizer to split a random variable along a dimension, similar to torch.split().
This is useful for treating different parts of a tensor with different reparameterizers or inference methods. For example when performing HMC inference on a time series, you can first apply DiscreteCosineReparam or HaarReparam, then apply SplitReparam to split into low-frequency and high-frequency components, and finally add the low-frequency components to the full_mass matrix together with globals.

Parameters

- **sections** – Size of a single chunk or list of sizes for each chunk.
- **dim** (int) – Dimension along which to split. Defaults to -1.

Type list(int)
__call__ (name, fn, obs)

3.12.11 Neural Transport

class NeuTraReparam (guide)
Bases: pyro.infer.reparam.reparam.Reparam.Reparam

This uses a trained AutoContinuous guide to alter the geometry of a model, typically for use e.g. in MCMC.
Example usage:

```python
# Step 1. Train a guide
guide = AutoIAFNormal(model)
svi = SVI(model, guide, ...)
# ...train the guide...
```

50 Chapter 3. Inference
# Step 2. Use trained guide in NeuTra MCMC

neutra = NeuTraReparam(guide)
model = poutine.reparam(model, config=lambda _: neutra)
nuts = NUTS(model)
# ...now use the model in HMC or NUTS...

This reparameterization works only for latent variables, not likelihoods. Note that all sites must share a single common NeuTraReparam instance, and that the model must have static structure.


**Parameters**

- **guide** *(AutoContinuous)* – A trained guide.

- **reparam** *(fn=None)*

- **__call__** *(name, fn, obs)*

- **transform_sample** *(latent)*

  Given latent samples from the warped posterior (with a possible batch dimension), return a dict of samples from the latent sites in the model.

  **Parameters**

  - **latent** – sample from the warped posterior (possibly batched). Note that the batch dimension must not collide with plate dimensions in the model, i.e. any batch dims \( d < - \text{max_plate_nesting} \).

  **Returns**

  - a dict of samples keyed by latent sites in the model.

  **Return type**

  - dict
4.1 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`.

4.1.1 Bernoulli

```python
class Bernoulli (probs=None, logits=None, validate_args=None)
```

4.1.2 Beta

```python
class Beta (concentration1, concentration0, validate_args=None)
```

4.1.3 Binomial

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

4.1.4 Categorical

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

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

4.1.6 Chi2

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

4.1.7 ContinuousBernoulli

class ContinuousBernoulli (probs=None, logits=None, lims=(0.499, 0.501), validate_args=None)

4.1.8 Dirichlet

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

4.1.9 Exponential

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

4.1.10 ExponentialFamily

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

4.1.11 FisherSnedecor

class FisherSnedecor (df1, df2, validate_args=None)

4.1.12 Gamma

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

class Geometric(probs=None, logits=None, validate_args=None)

4.1.14 Gumbel

class Gumbel(loc, scale, validate_args=None)

4.1.15 HalfCauchy

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

4.1.16 HalfNormal

class HalfNormal(scale, validate_args=None)

4.1.17 Independent

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

4.1.18 Laplace

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

4.1.19 LogNormal

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

4.1.20 LogisticNormal

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

4.1.21 LowRankMultivariateNormal

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

class MixtureSameFamily (mixture_distribution, component_distribution, validate_args=None)
    Wraps torch.distributions.mixture_same_family.MixtureSameFamily with TorchDistributionMixin.

4.1.23 Multinomial

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

4.1.24 MultivariateNormal

class MultivariateNormal (loc, covariance_matrix=None, precision_matrix=None, scale_tril=None, validate_args=None)

4.1.25 NegativeBinomial

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

4.1.26 Normal

class Normal (loc, scale, validate_args=None)

4.1.27 OneHotCategorical

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

4.1.28 Pareto

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

4.1.29 Poisson

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

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

4.1.31 RelaxedOneHotCategorical

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

4.1.32 StudentT

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

4.1.33 TransformedDistribution

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

4.1.34 Uniform

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

4.1.35 VonMises

class VonMises(loc, concentration, validate_args=None)

4.1.36 Weibull

class Weibull(scale, concentration, validate_args=None)
    Wraps torch.distributions.weibull.Weibull with TorchDistributionMixin.

4.2 Pyro Distributions

4.2.1 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:

```python
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.

```python
has_rsample = False
has_enumerate_support = False

__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

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

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

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.

    Setting .has_rsample on a distribution instance will determine whether inference engines like SVI use reparameterized samplers or the score function estimator.

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

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

**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

**conjugate_update** *(other)*

EXPERIMENTAL Creates an updated distribution fusing information from another compatible distribution. This is supported by only a few conjugate distributions.

This should satisfy the equation:

```python
fg, log_normalizer = f.conjugate_update(g)
assert f.log_prob(x) + g.log_prob(x) == fg.log_prob(x) + log_normalizer
```

Note this is equivalent to funsor.ops.add on Funsor distributions, but we return a lazy sum \((updated,\ log\_normalizer)\) because PyTorch distributions must be normalized. Thus `conjugate_update()` should commute with `dist_to_funsor()` and `tensor_to_funsor()`

```python
dist_to_funsor(f) + dist_to_funsor(g)  
== dist_to_funsor(fg) + tensor_to_funsor(log_normalizer)
```

**Parameters**  
- **other** – A distribution representing \(p(data|latent)\) but normalized over latent rather than data. Here latent is a candidate sample from self and data is a ground observation of unrelated type.

**Returns**  
a pair \((updated,\ log\_normalizer)\) where updated is an updated distribution of type type(self), and log_normalizer is a Tensor representing the normalization factor.

**has_rsample** *(value)*

Force reparameterized or detached sampling on a single distribution instance. This sets the .has_rsample attribute in-place.

This is useful to instruct inference algorithms to avoid reparameterized gradients for variables that discontinuously determine downstream control flow.

**Parameters**  
- **value** *(bool)* – Whether samples will be pathwise differentiable.

**Returns**  
self

**Return type**  
Distribution

**rv**

EXPERIMENTAL Switch to the Random Variable DSL for applying transformations to random variables. Supports either chaining operations or arithmetic operator overloading.

Example usage:
# This should be equivalent to an Exponential distribution.
Uniform(0, 1).rv.log().neg().dist

# These two distributions Y1, Y2 should be the same
X = Uniform(0, 1).rv
Y1 = X.mul(4).pow(0.5).sub(1).abs().neg().dist
Y2 = (-abs(((4*X)**(0.5) - 1))).dist

Returns A :class:`pyro.contrib.randomvariable.random_variable.RandomVariable` object wrapping this distribution.

Return type RandomVariable

## 4.2.2 TorchDistributionMixin
class TorchDistributionMixin
Bases: pyro.distributions.distribution.Distribution

Mixin to provide Pyro compatibility for PyTorch distributions.

You should instead use TorchDistribution for new distribution classes.

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

__call__ (sample_shape=torch.Size([]))
Samples a random value.

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

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

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

Return type torch.Tensor

event_dim

Returns Number of dimensions of individual events.

Return type int

shape (sample_shape=torch.Size([]))
The tensor shape of samples from this distribution.

Samples are of shape:

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

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

Returns Tensor shape of samples.

Return type torch.Size
**expand**\(^{(batch\_shape,\_instance=\text{None})}\)
Returns a new ExpandedDistribution instance with batch dimensions expanded to \(batch\_shape\).

**Parameters**
- \(\text{batch\_shape} \text{ (tuple)}\) – batch shape to expand to.
- \(\_\text{instance} \text{ – unused argument for compatibility with torch.distributions. Distribution.expand()}

**Returns** an instance of ExpandedDistribution.
**Return type** ExpandedDistribution

**expand**\(_\text{by} \text{ (sample\_shape)}\)
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 \(\text{expand()}\) instead.

**Parameters** \(\text{sample\_shape} \text{ (torch.Size)}\) – The size of the iid batch to be drawn from the distribution.

**Returns** An expanded version of this distribution.
**Return type** ExpandedDistribution

**reshape**\((sample\_shape=\text{None}, \text{extra\_event\_dims=\text{None}})\)

**to**\(_**\text{event} \text{ (reinterpreted\_batch\_ndims=\text{None})}\)
Reinterprets the \(n\) rightmost dimensions of this distributions \(batch\_shape\) as event dims, adding them to the left side of \(event\_shape\).

**Example:**

```plaintext
>>> [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** \(\text{reinterpreted\_batch\_ndims} \text{ (int)}\) – The number of batch dimensions to reinterpret as event dimensions. May be negative to remove dimensions from an \text{pyro.distributions.torch.Independent}. If None, convert all dimensions to event dimensions.

**Returns** A reshaped version of this distribution.
**Return type** pyro.distributions.torch.Independent

**independent**\((\text{reinterpreted\_batch\_ndims=\text{None}})\)

**mask**\((\text{mask})\)
Masks a distribution by a boolean or boolean-valued tensor that is broadcastable to the distributions \(batch\_shape\).

**Parameters** \(\text{mask} \text{ (bool or torch.Tensor)}\) – A boolean or boolean valued tensor.

**Returns** A masked copy of this distribution.
**Return type** MaskedDistribution

---

4.2. Pyro Distributions 61
4.2.3 TorchDistribution

class TorchDistribution(batch_shape=torch.Size([]), event_shape=torch.Size([]), validate_args=None)


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:

```python
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:

```python
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:

```python
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.

```python
expand(batch_shape, _instance=None)
```
Returns a new ExpandedDistribution instance with batch dimensions expanded to batch_shape.

Parameters
- **batch_shape** (tuple) – batch shape to expand to.
• `_instance` – unused argument for compatibility with `torch.distributions.Distribution.expand()`

Returns an instance of `ExpandedDistribution`.

Return type `ExpandedDistribution`

### 4.2.4 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:

```python
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’: LowerTriangular()}

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

### 4.2.5 BetaBinomial

class `BetaBinomial`(`concentration1`, `concentration0`, `total_count`=1, `validate_args`=None)

Bases: `pyro.distributions.torch_distribution.TorchDistribution`

Compound distribution comprising of a beta-binomial pair. The probability of success (`probs` for the `Binomial` distribution) is unknown and randomly drawn from a `Beta` distribution prior to a certain number of Bernoulli trials given by `total_count`.

Parameters

- `concentration1` ([`float` or `torch.Tensor`]) – 1st concentration parameter (alpha) for the Beta distribution.
• **concentration0** (*float or torch.Tensor*) – 2nd concentration parameter (beta) for the Beta distribution.

• **total_count** (*float or torch.Tensor*) – Number of Bernoulli trials.

approx_log_prob_tol = 0.0

arg_constraints = {'concentration0': GreaterThan(lower_bound=0.0), 'concentration1': GreaterThan(lower_bound=0.0), 'total_count': IntegerGreaterThan(lower_bound=0)}

concentration0
concentration1
enumerate_support (expand=True)
expand (batch_shape, _instance=None)
has enumerate_support = True
log_prob (value)
mean
sample (sample_shape=())
support
variance

### 4.2.6 CoalescentTimes

class CoalescentTimes (*leaf_times*, *, validate_args=None)

Bases: pyro.distributions.torch_distribution.TorchDistribution

Distribution over coalescent times given irregular sampled *leaf_times*.

Sample values will be sorted sets of binary coalescent times. Each sample *value* will have cardinality *value.size(-1) = leaf_times.size(-1) - 1*, so that phylogenies are complete binary trees. This distribution can thus be batched over multiple samples of phylogenies given fixed (number of) leaf times, e.g. over phylogeny samples from BEAST or MrBayes.

References


Parameters **leaf_times** (*torch.Tensor*) – Vector of times of sampling events, i.e. leaf nodes in the phylogeny. These can be arbitrary real numbers with arbitrary order and duplicates.

arg_constraints = {'leaf_times': Real()}

log_prob (value)

sample (sample_shape=torch.Size(()))
support
4.2.7 CoalescentTimesWithRate

class CoalescentTimesWithRate(leaf_times, rate_grid, *, validate_args=None)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Distribution over coalescent times given irregular sampled leaf_times and piecewise constant coalescent rates defined on a regular time grid.

This assumes a piecewise constant base coalescent rate specified on time intervals (-inf,1], [1,2], ..., [T-1,inf), where T = rate_grid.size(-1). Leaves may be sampled at arbitrary real times, but are commonly sampled in the interval [0, T].

Sample values will be sorted sets of binary coalescent times. Each sample value will have cardinality value.size(-1) = leaf_times.size(-1) - 1, so that phylogenies are complete binary trees. This distribution can thus be batched over multiple samples of phylogenies given fixed (number of) leaf times, e.g. over phylogeny samples from BEAST or MrBayes.

This distribution implements log_prob() but not .sample().

See also CoalescentRateLikelihood.

References


Parameters

- leaf_times (torch.Tensor) – Tensor of times of sampling events, i.e. leaf nodes in the phylogeny. These can be arbitrary real numbers with arbitrary order and duplicates.

- rate_grid (torch.Tensor) – Tensor of base coalescent rates (pairwise rate of coalescence). For example in a simple SIR model this might be beta S / I. The rightmost dimension is time, and this tensor represents a (batch of) rates that are piecwise constant in time.

arg_constraints = {'leaf_times': Real(), 'rate_grid': GreaterThan(lower_bound=0.0)}
duration
expand(batch_shape, _instance=None)
log_prob(value)

Computes likelihood as in equations 7-8 of [3].

This has time complexity \(O(T + S N \log(N))\) where \(T\) is the number of time steps, \(N\) is the number of leaves, and \(S = \text{sample_shape}.\text{numel()}\) is the number of samples of value.

This is differentiable wrt rate_grid but neither leaf_times nor value = coal_times.

Parameters value (torch.Tensor) – A tensor of coalescent times. These denote sets of size leaf_times.size(-1) - 1 along the trailing dimension and should be sorted along that dimension.

Returns Likelihood p(coal_times | leaf_times, rate_grid)

Return type torch.Tensor

support
4.2.8 ConditionalDistribution

class ConditionalDistribution
    Bases: abc.ABC

    condition(context)

    Return type torch.distributions.Distribution

4.2.9 ConditionalTransformedDistribution

class ConditionalTransformedDistribution(base_dist, transforms)
    Bases: pyro.distributions.conditional.ConditionalDistribution

    clear_cache()

    condition(context)

4.2.10 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

4.2.11 DirichletMultinomial

class DirichletMultinomial(concentration, total_count=1, is_sparse=False, validate_args=None)
    Bases: pyro.distributions.torch_distribution.TorchDistribution
Compound distribution comprising of a dirichlet-multinomial pair. The probability of classes (probs for the Multinomial distribution) is unknown and randomly drawn from a Dirichlet distribution prior to a certain number of Categorical trials given by total_count.

Parameters

- or torch.Tensor concentration (float) – concentration parameter (alpha) for the Dirichlet distribution.
- or torch.Tensor total_count (int) – number of Categorical trials.
- is_sparse (bool) – Whether to assume value is mostly zero when computing log_prob(), which can speed up computation when data is sparse.

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'total_count': IntegerGreaterThan(lower_bound=0)}

concentration
expand(batch_shape, _instance=None)
log_prob(value)
mean
sample(sample_shape=())
support
variance

### 4.2.12 DiscreteHMM

class DiscreteHMM(initial_logits, transition_logits, observation_dist, validate_args=None, duration=None)
Bases: pyro.distributions.hmm.HiddenMarkovModel

Hidden Markov Model with discrete latent state and arbitrary observation distribution. This uses [1] to parallelize over time, achieving O(log(time)) parallel complexity.

The event_shape of this distribution includes time on the left:

event_shape = (num_steps,) + observation_dist.event_shape

This distribution supports any combination of homogeneous/heterogeneous time dependency of transition_logits and observation_dist. However, because time is included in this distribution’s event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with num_steps = 1, allowing log_prob() to work with arbitrary length data:

# homogeneous + homogeneous case:
event_shape = (1,) + observation_dist.event_shape

References:


Parameters

- initial_logits (Tensor) – A logits tensor for an initial categorical distribution over latent states. Should have rightmost size state_dim and be broadcastable to batch_shape + (state_dim,).
• **transition_logits** (*Tensor*) – A logits tensor for transition conditional distributions between latent states. Should have rightmost shape `(state_dim, state_dim)` (old, new), and be broadcastable to `batch_shape + (num_steps, state_dim, state_dim)`.

• **observation_dist** (*Distribution*) – A conditional distribution of observed data conditioned on latent state. The `.batch_shape` should have rightmost size `state_dim` and be broadcastable to `batch_shape + (num_steps, state_dim)`. The `.event_shape` may be arbitrary.

• **duration** (*int*) – Optional size of the time axis `event_shape[0]`. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

```python
arg_constraints = {'initial_logits': Real(), 'transition_logits': Real()}
```

```python
expand(batch_shape, _instance=None)
```

```python
filter(value)
```

Parameters `value` (*Tensor*) – A sequence of observations.

Returns A posterior distribution over latent states at the final time step. `result.logits` can then be used as `initial_logits` in a sequential Pyro model for prediction.

Return type **Categorical**

```python
log_prob(value)
```

```python
support
```

### 4.2.13 EmpiricalDistribution

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

Bases: `pyro.distributions.torch_distribution.TorchDistribution`

Empirical distribution associated with the sampled data. Note that the shape requirement for `log_weights` is that its shape must match the leftmost shape of `samples`. Samples are aggregated along the `aggregation_dim`, which is the rightmost dim of `log_weights`.

Example:

```python
>>> emp_dist = Empirical(torch.randn(2, 3, 10), torch.ones(2, 3))
>>> emp_dist.batch_shape
torch.Size([2])
>>> emp_dist.event_shape
torch.Size([10])
```

```python
>>> single_sample = emp_dist.sample()
>>> single_sample.shape
torch.Size([2, 10])
>>> batch_sample = emp_dist.sample((100,))
>>> batch_sample.shape
torch.Size([100, 2, 10])
```

```python
>>> emp_dist.log_prob(single_sample).shape
torch.Size([2])
```

(continues on next page)
>>> # Vectorized samples cannot be scored by log_prob.
>>> with pyro.validation_enabled():
...   emp_dist.log_prob(batch_sample).shape
Traceback (most recent call last):
...  
ValueError: `value.shape` must be torch.Size([2, 10])

Parameters

- **samples** (*torch.Tensor*) – samples from the empirical distribution.
- **log_weights** (*torch.Tensor*) – log weights (optional) corresponding to the 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()

hasEnumerateSupport = 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

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

sample_size  
Number of samples that constitute the empirical distribution.

Return int number of samples collected.

support = Real()

variance

4.2.14 ExtendedBetaBinomial

class ExtendedBetaBinomial (**concentration0, concentration0, total_count=1, validate_args=None**)  
Bases: pyro.distributions.conjugate.BetaBinomial

EXPERIMENTAL BetaBinomial distribution extended to have logical support the entire integers and to allow arbitrary integer total_count. Numerical support is still the integer interval [0, total_count].

arg_constraints = {'concentration0': GreaterThan(lower_bound=0.0), 'concentration1': GreaterThan(lower_bound=0.0), 'total_count': Integer}

log_prob (**value**)
4.2.15 ExtendedBinomial

class ExtendedBinomial(total_count=1, probs=None, logits=None, validate_args=None)
    Bases: pyro.distributions.torch.Binomial

    EXPERIMENTAL Binomial distribution extended to have logical support the entire integers and to allow
    arbitrary integer total_count. Numerical support is still the integer interval [0, total_count].

    arg_constraints = {'logits': Real(), 'probs': Interval(lower_bound=0.0, upper_bound=1.0), 'total_count': Integer}

    log_prob(value)
    support = Integer

4.2.16 FoldedDistribution

class FoldedDistribution(base_dist, validate_args=None)
    Bases: pyro.distributions.torch.TransformedDistribution

    Equivalent to TransformedDistribution(base_dist, AbsTransform()), but additionally sup-
    ports log_prob().

    Parameters
    base_dist (Distribution) – The distribution to reflect.

    expand(batch_shape, _instance=None)
    log_prob(value)
    support = GreaterThan(lower_bound=0.0)

4.2.17 GammaGaussianHMM

class GammaGaussianHMM(scale_dist, initial_dist, transition_matrix, transition_dist, observation_matrix, observation_dist, validate_args=None, duration=None)
    Bases: pyro.distributions.hmm.HiddenMarkovModel

    Hidden Markov Model with the joint distribution of initial state, hidden state, and observed state is a
    MultivariateStudentT distribution along the line of references [2] and [3]. This adapts [1] to paral-
    lelize over time to achieve O(log(time)) parallel complexity.

    This GammaGaussianHMM class corresponds to the generative model:

    ```python
    s = Gamma(df/2, df/2).sample()
    z = scale(initial_dist, s).sample()
    x = []
    for t in range(num_events):
        z = z @ transition_matrix + scale(transition_dist, s).sample()
        x.append(z @ observation_matrix + scale(observation_dist, s).sample())
    ```

    where scale(mvn(loc, precision), s) := mvn(loc, s * precision).

    The event_shape of this distribution includes time on the left:

    ```python
event_shape = (num_steps,) + observation_dist.event_shape
```

70 Chapter 4. Distributions
This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_dist` and `observation_dist`. However, because time is included in this distribution’s event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with `num_steps = 1`, allowing `log_prob()` to work with arbitrary length data:

```
event_shape = (1, obs_dim) # homogeneous + homogeneous case
```

References:


Variables

- `hidden_dim (int)` – The dimension of the hidden state.
- `obs_dim (int)` – The dimension of the observed state.

Parameters

- `scale_dist (Gamma)` – Prior of the mixing distribution.
- `initial_dist (MultivariateNormal)` – A distribution with unit scale mixing over initial states. This should have batch_shape broadcastable to `self.batch_shape`. This should have event_shape `(hidden_dim,)`.
- `transition_matrix (Tensor)` – A linear transformation of hidden state. This should have shape broadcastable to `self.batch_shape + (num_steps, hidden_dim, hidden_dim)` where the rightmost dims are ordered `(old, new)`.
- `transition_dist (MultivariateNormal)` – A process noise distribution with unit scale mixing. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(hidden_dim,)`.
- `observation_matrix (Tensor)` – A linear transformation from hidden to observed state. This should have shape broadcastable to `self.batch_shape + (num_steps, hidden_dim, obs_dim)`.
- `observation_dist (MultivariateNormal)` – An observation noise distribution with unit scale mixing. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(obs_dim,)`.
- `duration (int)` – Optional size of the time axis event_shape[0]. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

```
arg_constraints = {}
```

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

```
filter(value)
```

Compute posteriors over the multiplier and the final state given a sequence of observations. The posterior is a pair of Gamma and MultivariateNormal distributions (i.e. a GammaGaussian instance).

Parameters

- `value (Tensor)` – A sequence of observations.

Returns A pair of posterior distributions over the mixing and the latent state at the final time step.
4.2.18 GammaPoisson

class GammaPoisson(concentration, rate, validate_args=None)
    Bases: pyro.distributions.torch_distribution.TorchDistribution

Compound distribution comprising of a gamma-poisson pair, also referred to as a gamma-poisson mixture. The rate parameter for the Poisson distribution is unknown and randomly drawn from a Gamma distribution.

Note: This can be treated as an alternate parametrization of the NegativeBinomial (total_count, probs) distribution, with concentration = total_count and rate = (1 - probs) / probs.

Parameters

- or torch.Tensor concentration (float) – shape parameter (alpha) of the Gamma distribution.
- or torch.Tensor rate (float) – rate parameter (beta) for the Gamma distribution.

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'rate': GreaterThan(lower_bound=0.0)}

expand(batch_shape, _instance=None)

log_prob(value)

mean

rate

sample(sample_shape=())

support = IntegerGreaterThan(lower_bound=0)

variance

4.2.19 GaussianHMM

class GaussianHMM(initial_dist, transition_matrix, transition_dist, observation_matrix, observation_dist, validate_args=None, duration=None)
    Bases: pyro.distributions.hmm.HiddenMarkovModel

Hidden Markov Model with Gaussians for initial, transition, and observation distributions. This adapts [1] to parallelize over time to achieve O(log(time)) parallel complexity, however it differs in that it tracks the log normalizer to ensure log_prob() is differentiable.

This corresponds to the generative model:

```
z = initial_distribution.sample()
x = []
for t in range(num_events):
    z = z @ transition_matrix + transition_dist.sample()
    x.append(z @ observation_matrix + observation_dist.sample())
```
The event_shape of this distribution includes time on the left:

```python
event_shape = (num_steps,) + observation_dist.event_shape
```

This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_dist` and `observation_dist`. However, because time is included in this distribution’s event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with `num_steps = 1`, allowing `log_prob()` to work with arbitrary length data:

```python
event_shape = (1, obs_dim)  # homogeneous + homogeneous case
```

References:


Variables

- `hidden_dim (int)`: The dimension of the hidden state.
- `obs_dim (int)`: The dimension of the observed state.

Parameters

- `initial_dist (MultivariateNormal)`: A distribution over initial states. This should have `batch_shape` broadcastable to `self.batch_shape`. This should have `event_shape (hidden_dim,)`.
- `transition_matrix (Tensor)`: A linear transformation of hidden state. This should have `shape` broadcastable to `self.batch_shape + (num_steps, hidden_dim, hidden_dim)` where the rightmost dims are ordered (old, new).
- `transition_dist (MultivariateNormal)`: A process noise distribution. This should have `batch_shape` broadcastable to `self.batch_shape + (num_steps,)`. This should have `event_shape (hidden_dim,)`.
- `observation_matrix (Tensor)`: A linear transformation from hidden to observed state. This should have `shape` broadcastable to `self.batch_shape + (num_steps, hidden_dim, obs_dim)`.
- `observation_dist (MultivariateNormal or Normal)`: An observation noise distribution. This should have `batch_shape` broadcastable to `self.batch_shape + (num_steps,)`. This should have `event_shape (obs_dim,)`.
- `duration (int)`: Optional size of the time axis `event_shape[0]`. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

```python
arg_constraints = {}

conjugate_update (other)
EXPERIMENTAL Creates an updated GaussianHMM fusing information from another compatible distribution.

This should satisfy:

```python
fg, log_normalizer = f.conjugate_update(g)
assert f.log_prob(x) + g.log_prob(x) == fg.log_prob(x) + log_normalizer
```
Parameters `other` *(MultivariateNormal or Normal)* – A distribution representing \( p(\text{data}|\text{self.probs}) \) but normalized over `self.probs` rather than `data`.

Returns a pair `(updated, log_normalizer)` where `updated` is an updated `GaussianHMM`, and `log_normalizer` is a `Tensor` representing the normalization factor.

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

`filter(value)`  
Compute posterior over final state given a sequence of observations.

Parameters `value` *(Tensor)* – A sequence of observations.

Returns A posterior distribution over latent states at the final time step. `result` can then be used as `initial_dist` in a sequential Pyro model for prediction.

Return type `MultivariateNormal`

```
has_rsample = True

log_prob(value)
```

```
prefix_condition(data)
```

EXPERIMENTAL Given self has event_shape == (t+f, d) and data of shape batch_shape + (t, d), compute a conditional distribution of event_shape (f, d). Typically t is the number of training time steps, f is the number of forecast time steps, and d is the data dimension.

Parameters `data` *(Tensor)* – data of dimension at least 2.

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

```
rsample_posterior(value, sample_shape=torch.Size([]))
```

EXPERIMENTAL Sample from the latent state conditioned on observation.

```
support = Real()
```

### 4.2.20 GaussianMRF

**class GaussianMRF** *(initial_dist, transition_dist, observation_dist, validate_args=None)*  
**Bases:** pyro.distributions.torch_distribution.TorchDistribution

Temporal Markov Random Field with Gaussian factors for initial, transition, and observation distributions. This adapts [1] to parallelize over time to achieve O(log(time)) parallel complexity, however it differs in that it tracks the log normalizer to ensure `log_prob()` is differentiable.

The event_shape of this distribution includes time on the left:

```
event_shape = (num_steps,) + observation_dist.event_shape
```

This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_dist` and `observation_dist`. However, because time is included in this distribution’s event_shape, the homogeneous+homogeneous case will have a broadcastable event_shape with num_steps = 1, allowing `log_prob()` to work with arbitrary length data:

```
event_shape = (1, obs_dim)  # homogeneous + homogeneous case
```

References:

Variables

- `hidden_dim (int)` – The dimension of the hidden state.
- `obs_dim (int)` – The dimension of the observed state.

Parameters

- `initial_dist (MultivariateNormal)` – A distribution over initial states. This should have `batch_shape` broadcastable to `self.batch_shape`. This should have `event_shape` `(hidden_dim,)`.
- `transition_dist (MultivariateNormal)` – A joint distribution factor over a pair of successive time steps. This should have `batch_shape` broadcastable to `self.batch_shape + (num_steps,)`. This should have `event_shape` `(hidden_dim + hidden_dim,) (old+new)`.
- `observation_dist (MultivariateNormal)` – A joint distribution factor over a hidden and an observed state. This should have `batch_shape` broadcastable to `self.batch_shape + (num_steps,)`. This should have `event_shape` `(hidden_dim + obs_dim,)`.

```
arg_constraints = {}
expand(batch_shape, _instance=None)
log_prob(value)
```

### 4.2.21 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

\[ \Sigma_{ii} = (\text{component_scale}_k \times \text{coord_scale}_i)^2 \quad (i = 1, \ldots, D) \]

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_bound=0.0)}

    has_rsample = True

    log_prob(value)

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

### 4.2.22 ImproperUniform

**class ImproperUniform**(support, batch_shape, event_shape)

Bases: *pyro.distributions.torch_distribution.TorchDistribution*

Improper distribution with zero log_prob() and undefined sample().

This is useful for transforming a model from generative dag form to factor graph form for use in HMC. For example the following are equal in distribution:

```python
# Version 1. a generative dag
x = pyro.sample("x", Normal(0, 1))
y = pyro.sample("y", Normal(x, 1))
z = pyro.sample("z", Normal(y, 1))

# Version 2. a factor graph
xyz = pyro.sample("xyz", ImproperUniform(constraints.real, (), (3,)))
x, y, z = xyz.unbind(-1)
pyro.sample("x", Normal(0, 1), obs=x)
pyro.sample("y", Normal(x, 1), obs=y)
pyro.sample("z", Normal(y, 1), obs=z)
```

Note this distribution errors when `sample()` is called. To create a similar distribution that instead samples from a specified distribution consider using `.mask(False)` as in:

```python
xyz = dist.Normal(0, 1).expand([3]).to_event(1).mask(False)
```

**Parameters**

- **support** (*Constraint*) – The support of the distribution.

  - **batch_shape** (*torch.Size*) – The batch shape.

  - **event_shape** (*torch.Size*) – The event shape.

**arg_constraints** = {}  

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

**log_prob**(value)

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

**support**

### 4.2.23 IndependentHMM

**class IndependentHMM**(base_dist)

Bases: *pyro.distributions.torch_distribution.TorchDistribution*
Wrapper class to treat a batch of independent univariate HMMs as a single multivariate distribution. This converts distribution shapes as follows:

<table>
<thead>
<tr>
<th></th>
<th>.batch_shape</th>
<th>.event_shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_dist</td>
<td>shape + (obs_dim,)</td>
<td>(duration, 1)</td>
</tr>
<tr>
<td>result</td>
<td>shape</td>
<td>(duration, obs_dim)</td>
</tr>
</tbody>
</table>

Parameters  

**base_dist** (*HiddenMarkovModel*) – A base hidden Markov model instance.

arg_constraints = {}  
duration  
expand(*batch_shape, _instance=None*)  
has_rsample  
log_prob(*value*)  
rsample(*sample_shape=torch.Size([])*)  
support

### 4.2.24 InverseGamma

class **InverseGamma** (*concentration, rate, validate_args=None*)  
**Bases:** pyro.distributions.torch.TransformedDistribution

Creates an inverse-gamma distribution parameterized by *concentration* and *rate*.

\[ X \sim \text{Gamma}(\text{concentration}, \text{rate}) \quad Y = \frac{1}{X} \sim \text{InverseGamma}(\text{concentration}, \text{rate}) \]

Parameters

- **concentration** (*torch.Tensor*) – the concentration parameter (i.e. alpha).
- **rate** (*torch.Tensor*) – the rate parameter (i.e. beta).

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'rate': GreaterThan(lower_bound=0.0)}  
expand(*batch_shape, _instance=None*)  
has_rsample = True  
rate  
support = GreaterThan(lower_bound=0.0)

### 4.2.25 LinearHMM

class **LinearHMM** (*initial_dist, transition_matrix, transition_dist, observation_matrix, observation_dist, validate_args=None, duration=None*)  
**Bases:** pyro.distributions.hmm.HiddenMarkovModel

Hidden Markov Model with linear dynamics and observations and arbitrary noise for initial, transition, and observation distributions. Each of those distributions can be e.g. **MultivariateNormal** or **Independent** of **Normal**, **StudentT**, or **Stable**. Additionally the observation distribution may be constrained, e.g. **LogNormal**
This corresponds to the generative model:

```python
z = initial_distribution.sample()
x = []
for t in range(num_events):
    z = z @ transition_matrix + transition_dist.sample()
y = z @ observation_matrix + obs_base_dist.sample()
x.append(obs_transform(y))
```

where `observation_dist` is split into `obs_base_dist` and an optional `obs_transform` (defaulting to the identity).

This implements a reparameterized `rsample()` method but does not implement a `log_prob()` method. Derived classes may implement `log_prob()`.

Inference without `log_prob()` can be performed using either reparameterization with `LinearHMMReparam` or likelihood-free algorithms such as `EnergyDistance`. Note that while stable processes generally require a common shared stability parameter \( \alpha \), this distribution and the above inference algorithms allow heterogeneous stability parameters.

The event_shape of this distribution includes time on the left:

```
event_shape = (num_steps,) + observation_dist.event_shape
```

This distribution supports any combination of homogeneous/heterogeneous time dependency of `transition_dist` and `observation_dist`. However at least one of the distributions or matrices must be expanded to contain the time dimension.

**Variables**

- `hidden_dim (int)` – The dimension of the hidden state.
- `obs_dim (int)` – The dimension of the observed state.

**Parameters**

- `initial_dist` – A distribution over initial states. This should have batch_shape broadcastable to `self.batch_shape`. This should have event_shape `(hidden_dim,)`.
- `transition_matrix (Tensor)` – A linear transformation of hidden state. This should have shape broadcastable to `self.batch_shape + (num_steps, hidden_dim, hidden_dim)` where the rightmost dims are ordered `(old, new)`.
- `transition_dist` – A distribution over process noise. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(hidden_dim,)`.
- `observation_matrix (Tensor)` – A linear transformation from hidden to observed state. This should have shape broadcastable to `self.batch_shape + (num_steps, hidden_dim, obs_dim)`.
- `observation_dist` – A observation noise distribution. This should have batch_shape broadcastable to `self.batch_shape + (num_steps,)`. This should have event_shape `(obs_dim,)`.
- `duration (int)` – Optional size of the time axis event_shape[0]. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

```
arg_constraints = {} 
expand(batch_shape, _instance=None)
```
has_rsample = True
log_prob(value)
rsample(sample_shape=torch.Size([]))
support

### 4.2.26 LKJCorrCholesky

class LKJCorrCholesky(d, eta, validate_args=None)

Generates cholesky factors of correlation matrices using an LKJ prior.

The expected use is to combine it with a vector of variances and pass it to the scale_tril parameter of a multi-variate distribution such as MultivariateNormal.

E.g., if theta is a (positive) vector of covariances with the same dimensionality as this distribution, and Omega is sampled from this distribution, scale_tril=torch.mm(torch.diag(sqrt(theta)), Omega)

Note that the event_shape of this distribution is [d, d]

Note: When using this distribution with HMC/NUTS, it is important to use a step_size such as 1e-4. If not, you are likely to experience LAPACK errors regarding positive-definiteness.

For example usage, refer to pyro/examples/lkj.py.

Parameters

- d (int) – Dimensionality of the matrix
- eta (torch.Tensor) – A single positive number parameterizing the distribution.

arg_constraints = {'eta': GreaterThan(lower_bound=0.0)}
expand(batch_shape, _instance=None)
has_rsample = False
lkj_constant(eta, K)
log_prob(x)
sample(sample_shape=torch.Size([]))
support = CorrCholesky()

### 4.2.27 MaskedDistribution

class MaskedDistribution(base_dist, mask)

Bases: pyro.distributions.torch_distribution.TorchDistribution

Masks a distribution by a boolean tensor that is broadcastable to the distribution’s batch_shape.

In the special case mask is False, computation of log_prob(), score_parts(), and kl_divergence() is skipped, and constant zero values are returned instead.

Parameters mask (torch.Tensor or bool) – A boolean or boolean-valued tensor.

arg_constraints = {}
conjugate_update\(\text{(other)}\)
  EXPERIMENTAL.
enumerate_support\(\text{(expand=True)}\)
expand\(\text{(batch\_shape, \_instance=None)}\)
has_enumerate_support
has_rsample
log_prob\(\text{(value)}\)
mean
rsample\(\text{(sample\_shape=torch.Size([]))}\)
sample\(\text{(sample\_shape=torch.Size([]))}\)
score_parts\(\text{(value)}\)
support
variance

### 4.2.28 MaskedMixture

class MaskedMixture\(\text{(mask, component0, component1, validate\_args=None)}\)  
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:

```python
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** \(\text{(torch.Tensor)}\) – A boolean tensor toggling between component0 and component1.
- **component0** \(\text{(pyro.distributions.TorchDistribution)}\) – a distribution for batch elements mask == False.
- **component1** \(\text{(pyro.distributions.TorchDistribution)}\) – a distribution for batch elements mask == True.

arg_constraints = {}
expand\(\text{(batch\_shape)}\)
has_rsample
log_prob\(\text{(value)}\)
mean
rsample(sample_shape=torch.Size([]))
sample(sample_shape=torch.Size([]))
support

4.2.29 MixtureOfDiagNormals

class MixtureOfDiagNormals(locs, coord_scale, component_logits)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Mixture of Normal distributions with arbitrary means and arbitrary 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 and a D-dimensional diagonal covariance matrix. The K different component means are gathered into the K x D dimensional parameter locs and the K different scale parameters are gathered into the K x D dimensional parameter coord_scale. 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.


Parameters

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

arg_constraints = {'component_logits': Real(), 'coord_scale': GreaterThan(lower_bound=0.0), 'locs': Real()}

expand(batch_shape, _instance=None)

has_rsample = True

log_prob(value)

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

4.2.30 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 $D = 1$.


**Parameters**

- $\text{locs}$ (*torch.Tensor*) – $K \times D$ mean matrix
- $\text{coord_scale}$ (*torch.Tensor*) – shared $D$-dimensional scale vector
- $\text{component_logits}$ (*torch.Tensor*) – $K$-dimensional vector of softmax logits

$\text{arg_constraints} = \{ \text{'component_logits': Real()}, \text{'coord_scale': GreaterThan(lower_bound=0.0)} \}$

**expand** ($\text{batch_shape}$, _instance=None)

**has_rsample** = True

**log_prob** ($\text{value}$)

**rsample** ($\text{sample_shape}=$torch.Size([]))

### 4.2.31 MultivariateStudentT

**class MultivariateStudentT** ($\text{df}$, $\text{loc}$, $\text{scale_tril}$, _validate_args=None)

**Bases:** pyro.distributions.torch_distribution.TorchDistribution

Creates a multivariate Student’s t-distribution parameterized by degree of freedom $\text{df}$, mean $\text{loc}$ and scale $\text{scale_tril}$.

**Parameters**

- $\text{df}$ (*Tensor*) – degrees of freedom
- $\text{loc}$ (*Tensor*) – mean of the distribution
- $\text{scale_tril}$ (*Tensor*) – scale of the distribution, which is a lower triangular matrix with positive diagonal entries

$\text{arg_constraints} = \{ \text{'df': GreaterThan(lower_bound=0.0)}, \text{'loc': RealVector()}, \text{'scale_tril': LowerCholesky()} \}$

**covariance_matrix**

**expand** ($\text{batch_shape}$, _instance=None)

**has_rsample** = True

**log_prob** ($\text{value}$)

**mean**

**precision_matrix**

**rsample** ($\text{sample_shape}=$torch.Size([]))

**scale_tril**

**support** = RealVector()

**variance**
4.2.32 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([]))

4.2.33 OrderedLogistic

class OrderedLogistic(predictor, cutpoints, validate_args=None)
Bases: pyro.distributions.torch.Categorical

Alternative parametrization of the distribution over a categorical variable.

Instead of the typical parametrization of a categorical variable in terms of the probability mass of the individual categories $p$, this provides an alternative that is useful in specifying ordered categorical models. This accepts a vector of cutpoints which are an ordered vector of real numbers denoting baseline cumulative log-odds of the individual categories, and a model vector predictor which modifies the baselines for each sample individually.

These cumulative log-odds are then transformed into a discrete cumulative probability distribution, that is finally differenced to return the probability mass matrix $p$ that specifies the categorical distribution.

Parameters

- `predictor`(Tensor) – A tensor of predictor variables of arbitrary shape. The output shape of non-batched samples from this distribution will be the same shape as predictor.
- `cutpoints`(Tensor) – A tensor of cutpoints that are used to determine the cumulative probability of each entry in predictor belonging to a given category. The first cutpoints.ndim-1 dimensions must be broadcastable to predictor, and the -1 dimension is monotonically increasing.

arg_constraints = {'cutpoints': OrderedVector(), 'predictor': Real()}

expand(batch_shape, _instance=None)

4.2.34 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:


[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()`

4.2.35 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:


[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()`

4.2.36 Rejector

class Rejector(propose, log_prob_accept, log_scale, *, batch_shape=None, event_shape=None)
    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.

```python
class SpanningTree(edge_logits, sampler_options=None, validate_args=None)
Bases: pyro.distributions.torch_distribution.TorchDistribution

Distribution over spanning trees on a fixed number $V$ of vertices.
A tree is represented as torch.LongTensor edges of shape $(V-1, 2)$ satisfying the following properties:
1. The edges constitute a tree, i.e. are connected and cycle free.
2. Each edge $(v_1, v_2) = \text{edges}[e]$ is sorted, i.e. $v_1 < v_2$.
3. The entire tensor is sorted in colexicographic order.

Use `validate_edges()` to verify edges are correctly formed.

The `edge_logits` tensor has one entry for each of the $V*(V-1)//2$ edges in the complete graph on $V$ vertices, where edges are each sorted and the edge order is colexicographic:

\[
(0,1), (0,2), (1,2), (0,3), (1,3), (2,3), (0,4), (1,4), (2,4), \ldots
\]

This ordering corresponds to the size-independent pairing function:

\[
k = v_1 + v_2 \cdot (v_2 - 1) // 2
\]

where $k$ is the rank of the edge $(v_1, v_2)$ in the complete graph. To convert a matrix of edge logits to the linear representation used here:

```assert` my_matrix.shape == (V, V)
i, j = make_complete_graph(V)
edge_logits = my_matrix[i, j]
```

**Parameters**

- **edge_logits** *(torch.Tensor)* – A tensor of length $V*(V-1)//2$ containing logits (aka negative energies) of all edges in the complete graph on $V$ vertices. See above comment for edge ordering.

- **sampler_options** *(dict)* – An optional dict of sampler options including: mcmc_steps defaulting to a single MCMC step (which is pretty good); initial_edges defaulting to a cheap approximate sample; backend one of “python” or “cpp”, defaulting to “python”.

4.2.37 SpanningTree
arg_constraints = {'edge_logits': Real()}

enumerate_support (expand=True)
    This is implemented for trees with up to 6 vertices (and 5 edges).

has.enumerate_support = True

log_partition_function

log_prob (edges)

sample (sample_shape=torch.Size([]))
    This sampler is implemented using MCMC run for a small number of steps after being initialized by a cheap approximate sampler. This sampler is approximate and cubic time. This is faster than the classic Aldous-Broder sampler [1,2], especially for graphs with large mixing time. Recent research [3,4] proposes samplers that run in sub-matrix-multiply time but are more complex to implement.

References


support = IntegerGreaterThan(lower_bound=0)

validate_edges (edges)
    Validates a batch of edges tensors, as returned by sample() or enumerate_support() or as input to log_prob().

    Parameters edges (torch.LongTensor) – A batch of edges.

    Raises ValueError

    Returns None

4.2.38 Stable

class Stable (stability, skew, scale=1.0, loc=0.0, coords='S0', validate_args=None)
    Bases: pyro.distributions.torch_distribution.TorchDistribution

    Levy α-stable distribution. See [1] for a review.

    This uses Nolan’s parametrization [2] of the loc parameter, which is required for continuity and differentiability. This corresponds to the notation \( S_\alpha^0(\beta, \sigma, \mu_0) \) of [1], where \( \alpha = \text{stability} \), \( \beta = \text{skew} \), \( \sigma = \text{scale} \), and \( \mu_0 = \text{loc} \). To instead use the S parameterization as in scipy, pass coords="S", but BEWARE this is discontinuous at stability=1 and has poor geometry for inference.

    This implements a reparametrized sampler rsample(), but does not implement log_prob(). Inference can be performed using either likelihood-free algorithms such as EnergyDistance, or reparameterization via the reparam() handler with one of the reparameterizers LatentStableReparam, SymmetricStableReparam, or StableReparam e.g.:

    ```python
    with poutine.reparam(config=dict(x=StableReparam())):
        pyro.sample("x", Stable(stability, skew, scale, loc))
    ```


Parameters

- **stability** (Tensor) – Levy stability parameter \( \alpha \in (0, 2] \).
- **skew** (Tensor) – Skewness \( \beta \in [-1, 1] \).
- **scale** (Tensor) – Scale \( \sigma > 0 \). Defaults to 1.
- **loc** (Tensor) – Location \( \mu_0 \) when using Nolan’s S0 parametrization [2], or \( \mu \) when using the S parameterization. Defaults to 0.
- **coords** (str) – Either “S0” (default) to use Nolan’s continuous S0 parametrization, or “S” to use the discontinuous parameterization.

arg_constraints = {'loc': Real(), 'scale': GreaterThan(lower_bound=0.0), 'skew': Interval(lower_bound=-1, upper_bound=1), 'stability': Interval(lower_bound=0, upper_bound=2)}

expand (batch_shape, _instance=None)

has_rsample = True

log_prob (value)

mean

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

support = Real()

variance

4.2.39 TruncatedPolyaGamma

class TruncatedPolyaGamma (prototype, validate_args=None)
Bases: pyro.distributions.torch_distribution.TorchDistribution

This is a PolyaGamma(1, 0) distribution truncated to have finite support in the interval (0, 2.5). See [1] for details. As a consequence of the truncation the log_prob method is only accurate to about six decimal places. In addition the provided sampler is a rough approximation that is only meant to be used in contexts where sample accuracy is not important (e.g. in initialization). Broadly, this implementation is only intended for usage in cases where good approximations of the log_prob are sufficient, as is the case e.g. in HMC.

Parameters prototype (tensor) – A prototype tensor of arbitrary shape used to determine the dtype and device returned by sample and log_prob.

References


arg_constraints = {}

expand (batch_shape, _instance=None)
has_rsample = False

log_prob(value)

num_gamma_variates = 8
num_log_prob_terms = 7
sample(sample_shape=())
support = Interval(lower_bound=0.0, upper_bound=2.5)
truncation_point = 2.5

4.2.40 Unit

class Unit(log_factor, validate_args=None):
    Bases: pyro.distributions.torch_distribution.TorchDistribution
    Trivial nonnormalized distribution representing the unit type.
    The unit type has a single value with no data, i.e. value.numel() == 0.
    This is used for pyro.factor() statements.
    arg_constraints = {'log_factor': Real()}

    expand(batch_shape, _instance=None)
    log_prob(value)
    sample(sample_shape=torch.Size(()))
    support = Real()

4.2.41 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 vec-
    tor. 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()
4.2.42 ZeroInflatedDistribution

```python
class ZeroInflatedDistribution(base_dist, *, gate=None, gate_logits=None, validate_args=None):
    Bases: pyro.distributions.torch_distribution.TorchDistribution

Generic Zero Inflated distribution.
This can be used directly or can be used as a base class as e.g. for ZeroInflatedPoisson and
ZeroInflatedNegativeBinomial.

Parameters

- **base_dist** (TorchDistribution) – the base distribution.
- **gate** (torch.Tensor) – probability of extra zeros given via a Bernoulli distribution.
- **gate_logits** (torch.Tensor) – logits of extra zeros given via a Bernoulli distribution.

arg_constraints = {'gate': Interval(lower_bound=0.0, upper_bound=1.0), 'gate_logits': Real()}

expand(batch_shape, _instance=None)

gate
gate_logits
log_prob(value)
mean
sample(sample_shape=torch.Size([]))
support
variance
```

4.2.43 ZeroInflatedNegativeBinomial

```python
class ZeroInflatedNegativeBinomial(total_count, *, probs=None, logits=None, gate=None, gate_logits=None, validate_args=None):
    Bases: pyro.distributions.zero_inflated.ZeroInflatedDistribution

A Zero Inflated Negative Binomial distribution.

Parameters

- **total_count** (float or torch.Tensor) – non-negative number of negative Bernoulli trials.
- **probs** (torch.Tensor) – Event probabilities of success in the half open interval [0, 1).
- **gate** (torch.Tensor) – probability of extra zeros.
- **gate_logits** (torch.Tensor) – logits of extra zeros.

arg_constraints = {'gate': Interval(lower_bound=0.0, upper_bound=1.0), 'gate_logits': Real(), 'logits': Real(), 'probs': HalfOpenInterval(lower_bound=0.0, upper_bound=1.0), 'total_count': GreaterThanEq(lower_bound=0)}

logits
probs
support = IntegerGreaterThan(lower_bound=0)
```
4.2.44 ZeroInflatedPoisson

class ZeroInflatedPoisson(rate, *, gate=\text{None}, gate_logits=\text{None}, validate_args=\text{None})

Bases: pyro.distributions.zero_inflated.ZeroInflatedDistribution

A Zero Inflated Poisson distribution.

Parameters

- \text{rate} (\text{torch.Tensor}) – rate of poisson distribution.
- \text{gate} (\text{torch.Tensor}) – probability of extra zeros.
- \text{gate_logits} (\text{torch.Tensor}) – logits of extra zeros.

arg_constraints = \{ 'gate': \text{Interval(lower_bound=0.0, upper_bound=1.0)}, 'gate_logits': \text{Real()}, 'rate': \text{GreaterThan(lower_bound=0.0)} \}

rate

support = \text{IntegerGreaterThan(lower_bound=0)}

4.3 Transforms

4.3.1 ConditionalTransform

class ConditionalTransform

Bases: abc.ABC

condition(context)

Return type \text{torch.distributions.Transform}

4.3.2 CorrLCholeskyTransform

class CorrLCholeskyTransform(cache_size=0)

Bases: torch.distributions.transforms.Transform

Transforms a vector into the cholesky factor of a correlation matrix.

The input should have shape \( [\text{batch_shape}] + [d \times (d-1)/2] \). The output will have shape \( [\text{batch_shape}] + [d, d] \).

References:


bijective = True

codomain = CorrCholesky()

domain = RealVector()

event_dim = 1

log_abs_det_jacobian(x, y)

sign = 1
4.3.3 ELUTransform

class ELUTransform(cache_size=0)
    Bases: torch.distributions.transforms.Transform
    Bijective transform via the mapping \( y = \text{ELU}(x) \).
    bijective = True
    codomain = GreaterThan(lower_bound=0.0)
    domain = Real()
    log_abs_det_jacobian(x, y)
    sign = 1

4.3.4 HaarTransform

class HaarTransform(dim=-1, flip=False, cache_size=0)
    Bases: torch.distributions.transforms.Transform
    Discrete Haar transform.
    This uses \( \text{haar_transform()} \) and \( \text{inverse_haar_transform()} \) to compute (orthonormal) Haar and inverse Haar transforms. The jacobian is 1. For sequences with length \( T \) not a power of two, this implementation is equivalent to a block-structured Haar transform in which block sizes decrease by factors of one half from left to right.

    Parameters
        • \text{dim} (int) – Dimension along which to transform. Must be negative. This is an absolute dim counting from the right.
        • \text{flip} (bool) – Whether to flip the time axis before applying the Haar transform. Defaults to false.
    bijective = True
    codomain = RealVector()
    domain = RealVector()
    log_abs_det_jacobian(x, y)
    with_cache(cache_size=1)

4.3.5 LeakyReLUTransform

class LeakyReLUTransform(cache_size=0)
    Bases: torch.distributions.transforms.Transform
    Bijective transform via the mapping \( y = \text{LeakyReLU}(x) \).
    bijective = True
    codomain = GreaterThan(lower_bound=0.0)
    domain = Real()
    log_abs_det_jacobian(x, y)
    sign = 1
### 4.3.6 LowerCholeskyAffine

```python
class LowerCholeskyAffine(loc, scale_tril, cache_size=0):
    Bases: torch.distributions.transforms.Transform

    A bijection of the form,
    \[ y = Lx + r \]
    where \( L \) is a lower triangular matrix and \( r \) is a vector.

    Parameters
    - `loc` (torch.tensor) – the fixed D-dimensional vector to shift the input by.
    - `scale_tril` (torch.tensor) – the D x D lower triangular matrix used in the transformation.

    bijective = True
    codomain = RealVector()
    event_dim = 1
    log_abs_det_jacobian(x, y)
    volume_preserving = False
    with_cache(cache_size=1)
```

### 4.3.7 OrderedTransform

```python
class OrderedTransform(cache_size=0):
    Bases: torch.distributions.transforms.Transform

    Transforms a real vector into an ordered vector.
    Specifically, enforces monotonically increasing order on the last dimension of a given tensor via the transformation
    \[ y_0 = x_0, y_i = \sum_{1 \leq j \leq i} \exp(x_j) \]

    bijective = True
    codomain = OrderedVector()
    domain = RealVector()
    event_dim = 1
    log_abs_det_jacobian(x, y)
    sign = 1
```

### 4.3.8 Permute

```python
class Permute(permutation, *, dim=-1, cache_size=1):
    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 AffineAutoregressive 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
**AffineAutoregressive** is not equivalent to reordering the dimension inside the MADE networks that those IAFs use; using a **Permute** transform results in a distribution with more flexibility.

Example usage:

```python
>>> from pyro.nn import AutoRegressiveNN
>>> from pyro.distributions.transforms import AffineAutoregressive, Permute

>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))

>>> iaf1 = AffineAutoregressive(AutoRegressiveNN(10, [40]))

>>> ff = Permute(torch.randperm(10, dtype=torch.long))

>>> iaf2 = AffineAutoregressive(AutoRegressiveNN(10, [40]))

>>> flow_dist = dist.TransformedDistribution(base_dist, [iaf1, ff, iaf2])

>>> flow_dist.sample()  # doctest: +SKIP
```

Parameters

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

- **dim** *(int)* – the tensor dimension to permute. This value must be negative and defines the event dim as abs(dim).

**bijective** = True

**codomain** = RealVector()

**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.

**volume_preserving** = True

**with_cache** *(cache_size=1)*

### 4.3.9 DiscreteCosineTransform

**class DiscreteCosineTransform** *(dim=-1, smooth=0.0, cache_size=0)*

Bases: torch.distributions.transforms.Transform

Discrete Cosine Transform of type-II.

This uses **dct()** and **idct()** to compute orthonormal DCT and inverse DCT transforms. The jacobian is 1.

**Parameters**

- **dim** *(int)* – Dimension along which to transform. Must be negative. This is an absolute dim counting from the right.

- **smooth** *(float)* – Smoothing parameter. When 0, this transforms white noise to white noise; when 1 this transforms Brownian noise to to white noise; when -1 this transforms violet noise to white noise; etc. Any real number is allowed. https://en.wikipedia.org/wiki/Colors_of_noise.

**bijective** = True

**codomain** = RealVector()
domain = RealVector()
log_abs_det_jacobian(x, y)
with_cache(cache_size=1)

4.4 TransformModules

4.4.1 AffineAutoregressive
class AffineAutoregressive(autoregressive_nn, log_scale_min_clip=-5.0, log_scale_max_clip=3.0, sigmoid_bias=2.0, stable=False)
Bases: pyro.distributions.torch_transform.TransformModule

An implementation of the bijective transform of Inverse Autoregressive Flow (IAF), using by default Eq (10) from Kingma et al., 2016,
y = \mu_t + \sigma_t \odot x

where x are the inputs, y are the outputs, \mu_t, \sigma_t are calculated from an autoregressive network on x, and \sigma_t > 0.
If the stable keyword argument is set to True then the transformation used is,
y = \sigma_t \odot x + (1 - \sigma_t) \odot \mu_t

where \sigma_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).
Together with TransformedDistribution this provides a way to create richer variational approximations.
Example usage:

```python
>>> from pyro.nn import AutoRegressiveNN
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> transform = AffineAutoregressive(AutoRegressiveNN(10, [40]))
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```

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 overwritten during sampling a new value or an arbitrary 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** (callable) – 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
sigmoid_bias (float) – A term to add the logit of the input when using the stable transform.

stable (bool) – When true, uses the alternative “stable” version of the transform (see above).

References:


autoregressive = True
bijective = True
codomain = RealVector()
domain = RealVector()
event_dim = 1

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

sign = 1

4.4.2 AffineCoupling

class AffineCoupling (split_dim, hypernet, *, dim=1, log_scale_min_clip=-5.0, log_scale_max_clip=3.0)
Bases: pyro.distributions.torch_transform.TransformModule

An implementation of the affine coupling layer of RealNVP (Dinh et al., 2017) that uses the bijective transform,

\[ y_{1:d} = x_{1:d}, \quad y_{(d+1):D} = \mu + \sigma \odot x_{(d+1):D} \]

where \( x \) are the inputs, \( y \) are the outputs, e.g. \( x_{1:d} \) represents the first \( d \) elements of the inputs, and \( \mu, \sigma \) are shift and translation parameters calculated as the output of a function inputting only \( x_{1:d} \).

That is, the first \( d \) components remain unchanged, and the subsequent \( D - d \) are shifted and translated by a function of the previous components.

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

Example usage:

```python
>>> from pyro.nn import DenseNN
>>> input_dim = 10
>>> split_dim = 6
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [input_dim-split_dim, input_dim-split_dim]
>>> hypernet = DenseNN(split_dim, [10*input_dim], param_dims)
>>> transform = AffineCoupling(split_dim, hypernet)
>>> pyro.module("my_transform", transform) # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample() # doctest: +SKIP
```
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 overwritten during sampling a new value or an arbitrary value is being scored, it will calculate it manually.

This is an operation that scales as $O(1)$, i.e. constant in the input dimension. So in general, it is cheap to sample and score (an arbitrary value) from `AffineCoupling`.

**Parameters**

- `split_dim` (*int*) – Zero-indexed dimension $d$ upon which to perform input/output split for transformation.
- `hypernet` (*callable*) – A neural network whose forward call returns a real-valued mean and logit-scale as a tuple. The input should have final dimension `split_dim` and the output final dimension `input_dim-split_dim` for each member of the tuple.
- `dim` (*int*) – The tensor dimension on which to split. This value must be negative and defines the event dim as $-abs(dim)$.
- `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:**


```python
bijective = True
codomain = RealVector()
domain = RealVector()

log_abs_det_jacobian(x, y)
```

Calculates the elementwise determinant of the log jacobian.

### 4.4.3 BatchNorm

```python
class BatchNorm(input_dim, momentum=0.1, epsilon=1e-05)
Bases: pyro.distributions.torch_transform.TransformModule
```

A type of batch normalization that can be used to stabilize training in normalizing flows. The inverse operation is defined as

$$x = (y - \hat{\mu}) \otimes \sqrt{\hat{\sigma}^2} \otimes \gamma + \beta$$

that is, the standard batch norm equation, where $x$ is the input, $y$ is the output, $\gamma, \beta$ are learnable parameters, and $\hat{\mu}/\hat{\sigma}^2$ are smoothed running averages of the sample mean and variance, respectively. The constraint $\gamma > 0$ is enforced to ease calculation of the log-det-Jacobian term.

This is an element-wise transform, and when applied to a vector, learns two parameters $(\gamma, \beta)$ for each dimension of the input.

When the module is set to training mode, the moving averages of the sample mean and variance are updated every time the inverse operator is called, e.g., when a normalizing flow scores a minibatch with the `log_prob` method.
Also, when the module is set to training mode, the sample mean and variance on the current minibatch are used in place of the smoothed averages, i.e., for the inverse operator. For this reason it is not the case that \( x = g(g^{-1}(x)) \) during training, i.e., that the inverse operation is the inverse of the forward one.

Example usage:

```python
>>> from pyro.nn import AutoRegressiveNN
>>> from pyro.distributions.transforms import AffineAutoregressive

>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))

>>> iafs = [AffineAutoregressive(AutoRegressiveNN(10, [40])) for _ in range(2)]

>>> bn = BatchNorm(10)

>>> flow_dist = dist.TransformedDistribution(base_dist, [iafs[0], bn, iafs[1]])

>>> flow_dist.sample()  # doctest: +SKIP
```

Parameters

- **input_dim** (*int*) – the dimension of the input
- **momentum** (*float*) – momentum parameter for updating moving averages
- **epsilon** (*float*) – small number to add to variances to ensure numerical stability

References:


**bijective** = True

**codomain** = Real()

**constrained_gamma**

**domain** = Real()

**event_dim** = 0

**log_abs_det_jacobian**(x, y)

Calculates the elementwise determinant of the log Jacobian, \(dx/dy\)

### 4.4.4 BlockAutoregressive

#### class BlockAutoregressive

```python
class BlockAutoregressive(input_dim, hidden_factors=[8, 8], activation='tanh', residual=None):
    Bases: pyro.distributions.torch_transform.TransformModule
```

An implementation of Block Neural Autoregressive Flow (block-NAF) (De Cao et al., 2019) bijective transform. Block-NAF uses a similar transformation to deep dense NAF, building the autoregressive NN into the structure of the transform, in a sense.

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

Example usage:
base_dist = dist.Normal(torch.zeros(10), torch.ones(10))

naf = BlockAutoregressive(input_dim=10)

pyro.module("my_naf", naf)  # doctest: +SKIP

naf_dist = dist.TransformedDistribution(base_dist, [naf])

naf_dist.sample()  # doctest: +SKIP

The inverse operation is not implemented. This would require numerical inversion, e.g., using a root finding method - a possibility for a future implementation.

Parameters

- **input_dim** (*int*) – The dimensionality of the input and output variables.
- **hidden_factors** (*list*) – Hidden layer \(i\) has \(\text{hidden_factors}[i]\) hidden units per input dimension. This corresponds to both \(a\) and \(b\) in De Cao et al. (2019). The elements of \(\text{hidden_factors}\) must be integers.
- **activation** (*string*) – Activation function to use. One of ‘ELU’, ‘LeakyReLU’, ‘sigmoid’, or ‘tanh’.
- **residual** (*string*) – Type of residual connections to use. Choices are “None”, “normal” for \(y + f(y)\), and “gated” for \(\alpha y + (1 - \alpha y)\) for learnable parameter \(\alpha\).

References:


autoregressive = True
bijective = True
codomain = RealVector()
domain = RealVector()
event_dim = 1
log_abs_det_jacobian(x, y)

Calculates the elementwise determinant of the log jacobian

4.4.5 ConditionalAffineAutoregressive

class ConditionalAffineAutoregressive (autoregressive_nn, **kwargs)

Bases: pyro.distributions.conditional.ConditionalTransformModule

An implementation of the bijective transform of Inverse Autoregressive Flow (IAF) that conditions on an additional context variable and uses, by default, Eq (10) from Kingma Et Al., 2016,

\[ y = \mu_t + \sigma_t \odot x \]

where \(x\) are the inputs, \(y\) are the outputs, \(\mu_t, \sigma_t\) are calculated from an autoregressive network on \(x\) and context \(z \in \mathbb{R}^M\), and \(\sigma_t > 0\).

If the stable keyword argument is set to True then the transformation used is,

\[ y = \sigma_t \odot x + (1 - \sigma_t) \odot \mu_t \]

where \(\sigma_t\) 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 \((0, 1)\).

Together with ConditionalTransformedDistribution this provides a way to create richer variational approximations.
Example usage:

```python
>>> from pyro.nn import ConditionalAutoRegressiveNN
>>> input_dim = 10
>>> context_dim = 4
>>> batch_size = 3
>>> hidden_dims = [10*input_dim, 10*input_dim]
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> hypernet = ConditionalAutoRegressiveNN(input_dim, context_dim, hidden_dims)
>>> transform = ConditionalAffineAutoregressive(hypernet)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist,...
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```

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 overwritten during sampling a new value or an arbitrary 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.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
- `sigmoid_bias(float)` – A term to add the logit of the input when using the stable tranform.
- `stable(bool)` – When true, uses the alternative “stable” version of the transform (see above).

**References:**


```python
bijective = True
codomain = RealVector()

condition(context)
    Conditions on a context variable, returning a non-conditional transform of of type AffineAutoregressive.

domain = RealVector()
event_dim = 1
```

4.4. TransformModules
4.4.6 ConditionalAffineCoupling

class ConditionalAffineCoupling(split_dim, hypernet, **kwargs)

Bases: pyro.distributions.conditional.ConditionalTransformModule

An implementation of the affine coupling layer of RealNVP (Dinh et al., 2017) that conditions on an additional context variable and uses the bijective transform,

$$y_{1:d} = x_{1:d} y_{(d+1):D} = \mu + \sigma \odot x_{(d+1):D}$$

where \(x\) are the inputs, \(y\) are the outputs, e.g. \(x_{1:d}\) represents the first \(d\) elements of the inputs, and \(\mu, \sigma\) are shift and translation parameters calculated as the output of a function input \(x_{1:d}\) and a context variable \(z \in \mathbb{R}^M\).

That is, the first \(d\) components remain unchanged, and the subsequent \(D - d\) are shifted and translated by a function of the previous components.

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

Example usage:

```python
>>> from pyro.nn import ConditionalDenseNN
>>> input_dim = 10
>>> split_dim = 6
>>> context_dim = 4
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [input_dim-split_dim, input_dim-split_dim]
>>> hypernet = ConditionalDenseNN(split_dim, context_dim, [10*input_dim],
... param_dims)
>>> transform = ConditionalAffineCoupling(split_dim, hypernet)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist,
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```

The inverse of the Bijector is required when, e.g., scoring the log density of a sample with ConditionalTransformedDistribution. This implementation caches the inverse of the Bijector when its forward operation is called, e.g., when sampling from ConditionalTransformedDistribution. However, if the cached value isn’t available, either because it was overwritten during sampling a new value or an arbitrary value is being scored, it will calculate it manually.

This is an operation that scales as \(O(1)\), i.e. constant in the input dimension. So in general, it is cheap to sample and score (an arbitrary value) from ConditionalAffineCoupling.

Parameters

- **split_dim** (int) – Zero-indexed dimension \(d\) upon which to perform input/output split for transformation.

- **hypernet** (callable) – A neural network whose forward call returns a real-valued mean and logit-scale as a tuple. The input should have final dimension split_dim and the output final dimension input_dim-split_dim for each member of the tuple. The network also inputs a context variable as a keyword argument in order to condition the output upon it.

- **log_scale_min_clip** (float) – The minimum value for clipping the log(scale) from the NN.
• log_scale_max_clip (float) – The maximum value for clipping the log(scale) from
the NN

References:

bijective = True

codomain = RealVector()

condition (context)

domain = RealVector()
event_dim = 1

4.4.7 ConditionalGeneralizedChannelPermute

class ConditionalGeneralizedChannelPermute (nn, channels=3, permutation=None)
Bases: pyro.distributions.conditional.ConditionalTransformModule

A bijection that generalizes a permutation on the channels of a batch of 2D image in \([\ldots, C, H, W]\) format
conditioning on an additional context variable. Specifically this transform performs the operation,

\[ y = \text{torch.nn.functional.conv2d}(x, W) \]

where \(x\) are the inputs, \(y\) are the outputs, and \(W \sim C \times C \times 1 \times 1\) is the filter matrix for a 1x1 convolution
with \(C\) input and output channels.

Ignoring the final two dimensions, \(W\) is restricted to be the matrix product,

\[ W = PLU \]

where \(P \sim C \times C\) is a permutation matrix on the channel dimensions, and \(LU \sim C \times C\) is an invertible product
of a lower triangular and an upper triangular matrix that is the output of an NN with input \(z \in \mathbb{R}^M\) representing
the context variable to condition on.

The input \(x\) and output \(y\) both have shape \(\ldots, C, H, W\), where \(C\) is the number of channels set at initialization.

This operation was introduced in [1] for Glow normalizing flow, and is also known as 1x1 invertible convolution.
It appears in other notable work such as [2,3], and corresponds to the class tfp.bijectors.MatvecLU of TensorFlow
Probability.

Example usage:

```python
>>> from pyro.nn.dense_nn import DenseNN
>>> context_dim = 5
>>> batch_size = 3
>>> channels = 3
>>> base_dist = dist.Normal(torch.zeros(channels, 32, 32),
... torch.ones(channels, 32, 32))
>>> hidden_dims = [context_dim*10, context_dim*10]
>>> nn = DenseNN(context_dim, hidden_dims, param_dims=[channels*channels])
>>> transform = ConditionalGeneralizedChannelPermute(nn, channels=channels)
>>> z = torch.randn(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist,
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```
Parameters

- **nn** – a function inputting the context variable and outputting real-valued parameters of dimension \( C^2 \).
- **channels (int)** – Number of channel dimensions in the input.


```python
bijective = True
codomain = Real()
condition (context)
    condition()

domain = Real()
event_dim = 3
```

### 4.4.8 ConditionalHouseholder

```python
class ConditionalHouseholder (input_dim, nn, count_transforms=1)
Bases: pyro.distributions.conditional.ConditionalTransformModule

Represents multiple applications of the Householder bijective transformation conditioning on an additional context. A single Householder transformation takes the form,

\[ y = (I - 2 * uu^T/||u||^2)x \]

where \( x \) are the inputs with dimension \( D \), \( y \) are the outputs, and \( u \in \mathbb{R}^D \) is the output of a function, e.g. a NN, with input \( z \in \mathbb{R}^M \) representing the context variable to condition on.

The transformation represents the reflection of \( x \) through the plane passing through the origin with normal \( u \).

\( D \) applications of this transformation are able to transform standard i.i.d. standard Gaussian noise into a Gaussian variable with an arbitrary covariance matrix. With \( K < D \) transformations, one is able to approximate a full-rank Gaussian distribution using a linear transformation of rank \( K \).

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

Example usage:

```python
>>> from pyro.nn.dense_nn import DenseNN
>>> input_dim = 10
>>> context_dim = 5
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [input_dim]
>>> hypernet = DenseNN(context_dim, [50, 50], param_dims)
>>> transform = ConditionalHouseholder(input_dim, hypernet)
>>> z = torch.rand(batch_size, context_dim)
```

(continues on next page)
```python
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist,
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size])) # doctest: +SKIP
```

Parameters

- **input_dim** *(int)* – the dimension of the input (and output) variable.
- **nn** *(callable)* – a function inputting the context variable and outputting a triplet of real-valued parameters of dimensions \((1, D, D)\).
- **count_transforms** *(int)* – number of applications of Householder transformation to apply.

References:


**bijective** = True

codomain = RealVector()

```python
def condition(context):
    See pyro.distributions.conditional.ConditionalTransformModule.condition()
    domain = RealVector()
event_dim = 1
```

## 4.4.9 ConditionalMatrixExponential

class ConditionalMatrixExponential(input_dim, nn, iterations=8, normalization='none', bound=None)

Bases: pyro.distributions.conditional.ConditionalTransformModule

A dense matrix exponential bijective transform (Hoogeboom et al., 2020) that conditions on an additional context variable with equation,

\[
y = \exp(M)x
\]

where \(x\) are the inputs, \(y\) are the outputs, \(\exp(\cdot)\) represents the matrix exponential, and \(M \in \mathbb{R}^{D \times D}\) is the output of a neural network conditioning on a context variable \(z\) for input dimension \(D\). In general, \(M\) is not required to be invertible.

Due to the favourable mathematical properties of the matrix exponential, the transform has an exact inverse and a log-determinate-Jacobian that scales in time-complexity as \(O(D)\). Both the forward and reverse operations are approximated with a truncated power series. For numerical stability, the norm of \(M\) can be restricted with the `normalization` keyword argument.

Example usage:

```python
>>> from pyro.nn.dense_nn import DenseNN
>>> input_dim = 10
>>> context_dim = 5
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [input_dim,input_dim]
```

(continues on next page)
>>> hypernet = DenseNN(context_dim, [50, 50], param_dims)
>>> transform = ConditionalMatrixExponential(input_dim, hypernet)
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist, ...
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP

Parameters

- **input_dim** (*int*) – the dimension of the input (and output) variable.
- **iterations** (*int*) – the number of terms to use in the truncated power series that approximates matrix exponentiation.
- **normalization** (*string*) – One of ['none', 'weight', 'spectral'] normalization that selects what type of normalization to apply to the weight matrix. weight corresponds to weight normalization (Salimans and Kingma, 2016) and spectral to spectral normalization (Miyato et al, 2018).
- **bound** (*float*) – a bound on either the weight or spectral norm, when either of those two types of regularization are chosen by the normalization argument. A lower value for this results in fewer required terms of the truncated power series to closely approximate the exact value of the matrix exponential.

References:


**bijective** = True
codomain = RealVector()

condition (**context**)
  See pyro.distributions.conditional.ConditionalTransformModule.condition()
domain = RealVector()
event_dim = 1

### 4.4.10 ConditionalNeuralAutoregressive

class **ConditionalNeuralAutoregressive** (*autoregressive_nn*, **kwargs)
Bases: pyro.distributions.conditional.ConditionalTransformModule

An implementation of the deep Neural Autoregressive Flow (NAF) bijective transform of the “IAF flavour” conditioning on an additiona context variable that can be used for sampling and scoring samples drawn from it (but not arbitrary ones).

Example usage:
```python
>>> from pyro.nn import ConditionalAutoRegressiveNN

>>> input_dim = 10
>>> context_dim = 5
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> arn = ConditionalAutoRegressiveNN(input_dim, context_dim, [40], ... param_dims=[16]*3)
>>> transform = ConditionalNeuralAutoregressive(arn, hidden_units=16)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP

>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist, ... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```

The inverse operation is not implemented. This would require numerical inversion, e.g., using a root finding method - a possibility for a future implementation.

**Parameters**

- `autoregressive_nn(nn.Module)` – an autoregressive neural network whose forward call returns a tuple of three real-valued tensors, whose last dimension is the input dimension, and whose penultimate dimension is equal to hidden_units.

- `hidden_units(int)` – the number of hidden units to use in the NAF transformation (see Eq (8) in reference)

- `activation(string)` – Activation function to use. One of ‘ELU’, ‘LeakyReLU’, ‘sigmoid’, or ‘tanh’.

Reference:


**bijective** = True

**codomain** = RealVector()

**condition(context)**

Conditions on a context variable, returning a non-conditional transform of of type `NeuralAutoregressive`.

**domain** = RealVector()

**event_dim** = 1

### 4.4.11 ConditionalPlanar

class ConditionalPlanar(nn)

Bases: pyro.distributions.conditional.ConditionalTransformModule

A conditional ‘planar’ bijective transform using the equation,

\[ y = x + u \tanh(w^T z + b) \]

where x are the inputs with dimension \( D \), y are the outputs, and the pseudo-parameters \( b \in \mathbb{R} \), \( u \in \mathbb{R}^D \), and \( w \in \mathbb{R}^D \) are the output of a function, e.g. a NN, with input \( z \in \mathbb{R}^M \) representing the context variable to condition on. For this to be an invertible transformation, the condition \( w^T u > -1 \) is enforced.

Together with `ConditionalTransformedDistribution` this provides a way to create richer variational approximations.
Example usage:

```python
>>> from pyro.nn.dense_nn import DenseNN
>>> input_dim = 10
>>> context_dim = 5
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [1, input_dim, input_dim]
>>> hypernet = DenseNN(context_dim, [50, 50], param_dims)
>>> transform = ConditionalPlanar(hypernet)
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist, ... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```

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 the planar transform can be scored.

Parameters:

- **nn** *(callable)* – a function inputting the context variable and outputting a triplet of real-valued parameters of dimensions \((1, D, D)\).

References:


**bijective** = True

**codomain** = RealVector()

**condition**(context)

See pyro.distributions.conditional.ConditionalTransformModule.condition()

**domain** = RealVector()

**event_dim** = 1

### 4.4.12 ConditionalRadial

class **ConditionalRadial**(nn)

Bases: pyro.distributions.conditional.ConditionalTransformModule

A conditional ‘radial’ bijective transform context using the equation,

\[
y = x + \beta h(\alpha, r)(x - x_0)
\]

where \(x\) are the inputs, \(y\) are the outputs, and \(\alpha \in \mathbb{R}^+, \beta \in \mathbb{R}, \text{ and } x_0 \in \mathbb{R}^D\), are the output of a function, e.g. a NN, with input \(z \in \mathbb{R}^M\) representing the context variable to condition on. The input dimension is \(D\), \(r = ||x - x_0||_2\), and \(h(\alpha, r) = 1/(\alpha + r)\). For this to be an invertible transformation, the condition \(\beta > -\alpha\) is enforced.

Example usage:

```python
>>> from pyro.nn.dense_nn import DenseNN
>>> input_dim = 10
>>> context_dim = 5
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [input_dim, 1, 1]
>>> hypernet = DenseNN(context_dim, [50, 50], param_dims)
```
>>> transform = ConditionalRadial(hypernet)
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist,
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size])) # doctest: +SKIP

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 the radial
transform can be scored.

**Parameters**

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

References:

    [arXiv:1505.05770]

| bijective = True |
| codomain = RealVector() |
| condition(\(context\)) |
| domain = RealVector() |
| event_dim = 1 |

**4.4.13 ConditionalSpline**

class **ConditionalSpline** *(nn, input_dim, count_bins, bound=3.0, order='linear')*

**Bases:** pyro.distributions.conditional.ConditionalTransformModule

An implementation of the element-wise rational spline bijections of linear and quadratic order (Durkan et al.,
2019; Dolatabadi et al., 2020) conditioning on an additional context variable.

Rational splines are functions that are comprised of segments that are the ratio of two polynomials. For instance,
for the \(d\)-th dimension and the \(k\)-th segment on the spline, the function will take the form,

\[ y_d = \frac{\alpha(k)(x_d)}{\beta(k)(x_d)}, \]

where \(\alpha(k)\) and \(\beta(k)\) are two polynomials of order \(d\) whose parameters are the output of a function, e.g. a NN,
with input \(z\) in \(mathbb{R}^M\) representing the context variable to condition on.. For \(d = 1\), we say that the spline is linear, and
for \(d = 2\), quadratic. The spline is constructed on the specified bounding box, \([-K, K] \times [-K, K]\), with the
identity function used elsewhere.

Rational splines offer an excellent combination of functional flexibility whilst maintaining a numerically stable
inverse that is of the same computational and space complexities as the forward operation. This element-wise
transform permits the accurate representation of complex univariate distributions.

Example usage:

```python
>>> from pyro.nn.dense_nn import DenseNN
>>> input_dim = 10
>>> context_dim = 5
```
```python
>>> batch_size = 3
>>> count_bins = 8
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [input_dim * count_bins, input_dim * count_bins,
... input_dim * (count_bins - 1), input_dim * count_bins]
>>> hypernet = DenseNN(context_dim, [50, 50], param_dims)
>>> transform = ConditionalSpline(hypernet, input_dim, count_bins)
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist,
... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```

Parameters

- **input_dim** *(int)* – Dimension of the input vector. This is required so we know how many parameters to store.
- **count_bins** *(int)* – The number of segments comprising the spline.
- **bound** *(float)* – The quantity $K$ determining the bounding box, $[-K, K] \times [-K, K]$, of the spline.
- **order** *(string)* – One of [‘linear’, ‘quadratic’] specifying the order of the spline.

References:


**bijective** = True

**codomain** = Real()

**condition** *(context)*

See pyro.distributions.conditional.ConditionalTransformModule.condition()

domain = Real()

event_dim = 0

4.4.14 ConditionalSplineAutoregressive

class ConditionalSplineAutoregressive(*(input_dim, autoregressive_nn, **kwargs))
Bases: pyro.distributions.conditional.ConditionalTransformModule

An implementation of the autoregressive layer with rational spline bijections of linear and quadratic order (Durkan et al., 2019; Dolatabadi et al., 2020) that conditions on an additional context variable. Rational splines are functions that are comprised of segments that are the ratio of two polynomials (see Spline).

The autoregressive layer uses the transformation,

$$y_d = g_{\theta_d}(x_d) \quad d = 1, 2, \ldots, D$$

where $x = (x_1, x_2, \ldots, x_D)$ are the inputs, $y = (y_1, y_2, \ldots, y_D)$ are the outputs, $g_{\theta_d}$ is an elementwise rational monotonic spline with parameters $\theta_d$, and $\theta = (\theta_1, \theta_2, \ldots, \theta_D)$ is the output of a conditional autoregressive NN inputting $x$ and conditioning on the context variable $z$. 

108 Chapter 4. Distributions
Example usage:

```python
>>> from pyro.nn import ConditionalAutoRegressiveNN
>>> input_dim = 10
>>> count_bins = 8
>>> context_dim = 5
>>> batch_size = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> hidden_dims = [input_dim * 10, input_dim * 10]
>>> param_dims = [count_bins, count_bins, count_bins - 1, count_bins]
>>> hypernet = ConditionalAutoRegressiveNN(input_dim, context_dim, hidden_dims, ... param_dims=param_dims)
>>> transform = ConditionalSplineAutoregressive(input_dim, hypernet, ... count_bins=count_bins)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> z = torch.rand(batch_size, context_dim)
>>> flow_dist = dist.ConditionalTransformedDistribution(base_dist, ... [transform]).condition(z)
>>> flow_dist.sample(sample_shape=torch.Size([batch_size]))  # doctest: +SKIP
```

Parameters

- `input_dim (int)` – Dimension of the input vector. Despite operating element-wise, this is required so we know how many parameters to store.
- `autoregressive_nn (callable)` – an autoregressive neural network whose forward call returns tuple of the spline parameters
- `count_bins (int)` – The number of segments comprising the spline.
- `bound (float)` – The quantity \( K \) determining the bounding box, \([-K, K] \times [-K, K]\), of the spline.
- `order (string)` – One of ['linear', 'quadratic'] specifying the order of the spline.

References:


`bijective = True`
`codomain = RealVector()`

```
condition (context)
Conditions on a context variable, returning a non-conditional transform of of type
SplineAutoregressive.
```

domain = RealVector()
```
event_dim = 1
```

### 4.4.15 ConditionalTransformModule

class ConditionalTransformModule (*args, **kwargs)
        module.Module
Conditional transforms with learnable parameters such as normalizing flows should inherit from this class rather than `ConditionalTransform` so they are also a subclass of `Module` and inherit all the useful methods of that class.

### 4.4.16 GeneralizedChannelPermute

```python
class GeneralizedChannelPermute(channels=3, permutation=None):
    Bases: pyro.distributions.transforms.generalized_channel_permute.ConditionedGeneralizedChannelPermute,
           pyro.distributions.torch_transform.TransformModule
    A bijection that generalizes a permutation on the channels of a batch of 2D image in \([\ldots, C, H, W]\) format. Specifically this transform performs the operation,
    \[ y = \text{torch.nn.functional.conv2d}(x, W) \]
    where \(x\) are the inputs, \(y\) are the outputs, and \(W \sim C \times C \times 1 \times 1\) is the filter matrix for a 1x1 convolution with \(C\) input and output channels.
    
    Ignoring the final two dimensions, \(W\) is restricted to be the matrix product,
    \[ W = PLU \]
    where \(P \sim C \times C\) is a permutation matrix on the channel dimensions, \(L \sim C \times C\) is a lower triangular matrix with ones on the diagonal, and \(U \sim C \times C\) is an upper triangular matrix. \(W\) is initialized to a random orthogonal matrix. Then, \(P\) is fixed and the learnable parameters set to \(L, U\).
    
    The input \(x\) and output \(y\) both have shape \([\ldots, C, H, W]\), where \(C\) is the number of channels set at initialization.
    
    This operation was introduced in [1] for Glow normalizing flow, and is also known as 1x1 invertible convolution. It appears in other notable work such as [2,3], and corresponds to the class `tfp.bijectors.MatvecLU` of TensorFlow Probability.
    
    Example usage:

```IPython
>>> channels = 3
>>> base_dist = dist.Normal(torch.zeros(channels, 32, 32), ...
... torch.ones(channels, 32, 32))
>>> inv_conv = GeneralizedChannelPermute(channels=channels)
>>> flow_dist = dist.TransformedDistribution(base_dist, [inv_conv])
>>> flow_dist.sample()  # doctest: +SKIP
```

**Parameters**

- **channels** (`int`) – Number of channel dimensions in the input.


4.4.17 Householder

class Householder (input_dim, count_transforms=1)


Represents multiple applications of the Householder bijective transformation. A single Householder transformation takes the form,
\[ y = (I - 2 \frac{uu^T}{||u||^2})x \]

where \( x \) are the inputs, \( y \) are the outputs, and the learnable parameters are \( u \in \mathbb{R}^D \) for input dimension \( D \).

The transformation represents the reflection of \( x \) through the plane passing through the origin with normal \( u \).

\( D \) applications of this transformation are able to transform standard i.i.d. standard Gaussian noise into a Gaussian variable with an arbitrary covariance matrix. With \( K < D \) transformations, one is able to approximate a full-rank Gaussian distribution using a linear transformation of rank \( K \).

Together with \texttt{TransformedDistribution} this provides a way to create richer variational approximations.

Example usage:

```python
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> transform = Householder(10, count_transforms=5)
>>> pyro.module("my_transform", p) # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample() # doctest: +SKIP
```

Parameters

- \texttt{input_dim (int)} – the dimension of the input (and output) variable.
- \texttt{count_transforms (int)} – number of applications of Householder transformation to apply.

References:


\texttt{bijective} = True
codomain = RealVector()
domain = RealVector()
event_dim = 1
reset_parameters()
volume_preserving = True

4.4.18 MatrixExponential

class MatrixExponential (input_dim, iterations=8, normalization='none', bound=None)


A dense matrix exponential bijective transform (Hoogeboom et al., 2020) with equation,
where $x$ are the inputs, $y$ are the outputs, $\exp(\cdot)$ represents the matrix exponential, and the learnable parameters are $M \in \mathbb{R}^D \times \mathbb{R}^D$ for input dimension $D$. In general, $M$ is not required to be invertible.

Due to the favourable mathematical properties of the matrix exponential, the transform has an exact inverse and a log-determinate-Jacobian that scales in time-complexity as $O(D)$. Both the forward and reverse operations are approximated with a truncated power series. For numerical stability, the norm of $M$ can be restricted with the `normalization` keyword argument.

Example usage:

```python
generals = base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
generals = transform = MatrixExponential(10)  # doctest: +SKIP
generals = flow_dist = dist.TransformedDistribution(base_dist, [transform])
generals = flow_dist.sample()  # doctest: +SKIP
```

Parameters

- **input_dim** *(int)* – the dimension of the input (and output) variable.
- **iterations** *(int)* – the number of terms to use in the truncated power series that approximates matrix exponentiation.
- **normalization** *(string)* – One of ['none', 'weight', 'spectral'] normalization that selects what type of normalization to apply to the weight matrix. weight corresponds to weight normalization (Salimans and Kingma, 2016) and spectral to spectral normalization (Miyato et al., 2018).
- **bound** *(float)* – a bound on either the weight or spectral norm, when either of those two types of regularization are chosen by the `normalization` argument. A lower value for this results in fewer required terms of the truncated power series to closely approximate the exact value of the matrix exponential.

References:


```
bivariate = True
codomain = Real()
domain = Real()
event_dim = 1
reset_parameters()
```

4.4.19 NeuralAutoregressive

class NeuralAutoregressive *(autoregressive_nn, hidden_units=16, activation='sigmoid')*

Bases: pyro.distributions.torch_transform.TransformModule
An implementation of the deep Neural Autoregressive Flow (NAF) bijective transform of the “IAF flavour” that can be used for sampling and scoring samples drawn from it (but not arbitrary ones).

Example usage:

```python
>>> from pyro.nn import AutoRegressiveNN
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> arn = AutoRegressiveNN(10, [40], param_dims=[16]*3)
>>> transform = NeuralAutoregressive(arn, hidden_units=16)
>>> pyro.module("my_transform", transform) # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample() # doctest: +SKIP
```

The inverse operation is not implemented. This would require numerical inversion, e.g., using a root finding method - a possibility for a future implementation.

Parameters

- `autoregressive_nn (nn.Module)` – an autoregressive neural network whose forward call returns a tuple of three real-valued tensors, whose last dimension is the input dimension, and whose penultimate dimension is equal to hidden_units.

- `hidden_units (int)` – the number of hidden units to use in the NAF transformation (see Eq (8) in reference)

- `activation (string)` – Activation function to use. One of ‘ELU’, ‘LeakyReLU’, ‘sigmoid’, or ‘tanh’.

Reference:


\[ autoregressive = True \]
\[ bijective = True \]
\[ codomain = RealVector() \]
\[ domain = RealVector() \]
\[ eps = 1e-08 \]
\[ event_dim = 1 \]
\[ log_abs_det_jacobian(x, y) \]

Calculates the elementwise determinant of the log Jacobian

4.4.20 Planar

class Planar(input_dim)

A ‘planar’ bijective transform with equation,

\[ y = x + u \tanh(w^T z + b) \]

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

Together with `TransformedDistribution` this provides a way to create richer variational approximations.
Example usage:

```python
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> transform = Planar(10)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```

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 the planar transform can be scored.

**Parameters**

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

**References**:


- **bijective** = True
- **codomain** = RealVector()
- **domain** = RealVector()
- **event_dim** = 1
- **reset_parameters**()

### 4.4.21 Polynomial

**class** **Polynomial** *(autoregressive_nn, input_dim, count_degree, count_sum)*

**Bases:** pyro.distributions.torch_transform.TransformModule

An autoregressive bijective transform as described in Jaini et al. (2019) applying following equation element-wise,

\[
y_n = c_n + \int_0^x \sum_{r=0}^K \left( \sum_{r=0}^R a_{r,k}^n u^r \right) du
\]

where \( x_n \) is the \( n \) is the \( n \), \( \left\{ a_{r,k}^n \in \mathbb{R} \right\} \) are learnable parameters that are the output of an autoregressive NN inputting \( x_{<n} = x_1, x_2, ..., x_{n-1} \).

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

Example usage:

```python
>>> from pyro.nn import AutoRegressiveNN
>>> input_dim = 10
>>> count_degree = 4
>>> count_sum = 3
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [(count_degree + 1)*count_sum]
>>> arn = AutoRegressiveNN(input_dim, [input_dim*10], param_dims)
>>> transform = Polynomial(arn, input_dim=input_dim, count_degree=count_degree, ...
... count_sum=count_sum)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```
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 a polynomial transform can be scored.

Parameters

- **autoregressive_nn** *(nn.Module)* – an autoregressive neural network whose forward call returns a tensor of real-valued numbers of size (batch_size, (count_degree+1)*count_sum, input_dim)
- **count_degree** *(int)* – The degree of the polynomial to use for each element-wise transformation.
- **count_sum** *(int)* – The number of polynomials to sum in each element-wise transformation.

References:


```python
autoregressive = True
bijective = True
codomain = RealVector()
domain = RealVector()
event_dim = 1
log_abs_det_jacobian(x, y)
    Calculates the elementwise determinant of the log Jacobian
reset_parameters()
```

4.4.22 Radial

class **Radial** *(input_dim)*

```

A ‘radial’ bijective transform using the equation,

\[
y = x + \beta h(\alpha, r)(x - x_0)
\]

where \(x\) are the inputs, \(y\) are the outputs, and the learnable parameters are \(\alpha \in \mathbb{R}^+, \beta \in \mathbb{R}, x_0 \in \mathbb{R}^D\), for input dimension \(D, r = \|x - x_0\|_2, h(\alpha, r) = 1/(\alpha + r)\). For this to be an invertible transformation, the condition \(\beta > -\alpha\) is enforced.

Example usage:

```python
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> transform = Radial(10)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```

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 the radial transform can be scored.

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


```python
bijection = True
codomain = RealVector()
domain = RealVector()
event_dim = 1
reset_parameters()
```

### 4.4.23 Spline

```python
class Spline(input_dim=8, count_bins=8, bound=3.0, order='linear')
```


An implementation of the element-wise rational spline bijections of linear and quadratic order (Durkan et al., 2019; Dolatabadi et al., 2020). Rational splines are functions that are comprised of segments that are the ratio of two polynomials. For instance, for the \(d\)-th dimension and the \(k\)-th segment on the spline, the function will take the form,

\[
y_d = \frac{\alpha(k)(x_d)}{\beta(k)(x_d)},
\]

where \(\alpha(k)\) and \(\beta(k)\) are two polynomials of order \(d\). For \(d = 1\), we say that the spline is linear, and for \(d = 2\), quadratic. The spline is constructed on the specified bounding box, \([-K, K] \times [-K, K]\), with the identity function used elsewhere.

Rational splines offer an excellent combination of functional flexibility whilst maintaining a numerically stable inverse that is of the same computational and space complexities as the forward operation. This element-wise transform permits the accurate representation of complex univariate distributions.

Example usage:

```python
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> transform = Spline(10, count_bins=4, bound=3.)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```

**Parameters**

- **`input_dim (int)`** – Dimension of the input vector. This is required so we know how many parameters to store.
- **`count_bins (int)`** – The number of segments comprising the spline.
- **`bound (float)`** – The quantity \(K\) determining the bounding box, \([-K, K] \times [-K, K]\), of the spline.
- **`order (string)`** – One of [‘linear’, ‘quadratic’] specifying the order of the spline.

References:


```python
bijective = True
codomain = Real()
domain = Real()
event_dim = 0
```

### 4.4.24 SplineAutoregressive

```python
class SplineAutoregressive(input_dim, autoregressive_nn, count_bins=8, bound=3.0, order='linear')
```

Bases: `pyro.distributions.torch_transform.TransformModule`

An implementation of the autoregressive layer with rational spline bijections of linear and quadratic order (Durkan et al., 2019; Dolatabadi et al., 2020). Rational splines are functions that are comprised of segments that are the ratio of two polynomials (see `Spline`).

The autoregressive layer uses the transformation,

\[ y_d = g_{\theta_d}(x_d) \quad d = 1, 2, \ldots, D \]

where \( x = (x_1, x_2, \ldots, x_D) \) are the inputs, \( y = (y_1, y_2, \ldots, y_D) \) are the outputs, \( g_{\theta_d} \) is an elementwise rational monotonic spline with parameters \( \theta_d \), and \( \theta = (\theta_1, \theta_2, \ldots, \theta_D) \) is the output of an autoregressive NN inputting \( x \).

Example usage:

```python
>>> from pyro.nn import AutoRegressiveNN
>>> input_dim = 10
>>> count_bins = 8
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> hidden_dims = [input_dim * 10, input_dim * 10]
>>> param_dims = [count_bins, count_bins, count_bins - 1, count_bins]
>>> hypernet = AutoRegressiveNN(input_dim, hidden_dims, param_dims=param_dims)
>>> transform = SplineAutoregressive(input_dim, hypernet, count_bins=count_bins)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```

**Parameters**

- **`input_dim (int)`** – Dimension of the input vector. Despite operating element-wise, this is required so we know how many parameters to store.

- **`autoregressive_nn (callable)`** – an autoregressive neural network whose forward call returns tuple of the spline parameters

- **`count_bins (int)`** – The number of segments comprising the spline.

- **`bound (float)`** – The quantity \( K \) determining the bounding box, \([-K, K] \times [-K, K]\), of the spline.

- **`order (string)`** – One of [‘linear’, ‘quadratic’] specifying the order of the spline.

**References:**

autoregressive = True
bijective = True
codomain = RealVector()
domain = RealVector()
event_dim = 1

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

4.4.25 SplineCoupling
class SplineCoupling(input_dim, split_dim, hypernet, count_bins=8, bound=3.0, order='linear', identity=False)
Bases: pyro.distributions.torch_transform.TransformModule

An implementation of the coupling layer with rational spline bijections of linear and quadratic order (Durkan et al., 2019; Dolatabadi et al., 2020). Rational splines are functions that are comprised of segments that are the ratio of two polynomials (see Spline).

The spline coupling layer uses the transformation,

\[ y_{1:d} = g_\theta(x_{1:d}) y_{(d+1):D} = h_\phi(x_{(d+1):D}; x_{1:d}) \]

where \( x \) are the inputs, \( y \) are the outputs, e.g. \( x_{1:d} \) represents the first \( d \) elements of the inputs, \( g_\theta \) is either the identity function or an elementwise rational monotonic spline with parameters \( \theta \), and \( h_\phi \) is a conditional elementwise spline spline, conditioning on the first \( d \) elements.

Example usage:

```python
>>> from pyro.nn import DenseNN
>>> input_dim = 10
>>> split_dim = 6
>>> count_bins = 8
>>> base_dist = dist.Normal(torch.zeros(input_dim), torch.ones(input_dim))
>>> param_dims = [(input_dim - split_dim) * count_bins,
...                ... (input_dim - split_dim) * count_bins,
...                ... (input_dim - split_dim) * (count_bins - 1),
...                ... (input_dim - split_dim) * count_bins]
>>> hypernet = DenseNN(split_dim, [10*input_dim], param_dims)
>>> transform = SplineCoupling(input_dim, split_dim, hypernet)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_dist.sample()  # doctest: +SKIP
```

Parameters

- **input_dim (int)** – Dimension of the input vector. Despite operating element-wise, this is required so we know how many parameters to store.
- **split_dim** – Zero-indexed dimension \( d \) upon which to perform input/ output split for transformation.
- **hypernet (callable)** – a neural network whose forward call returns a tuple of spline parameters (see ConditionalSpline).
- `count_bins (int)` – The number of segments comprising the spline.
- `bound (float)` – The quantity $K$ determining the bounding box, $[-K, K] \times [-K, K]$, of the spline.
- `order (string)` – One of ['linear', 'quadratic'] specifying the order of the spline.

References:

```python
bijective = True
codomain = RealVector()
domain = RealVector()
event_dim = 1
log_abs_det_jacobian(x, y)
```
Calculates the elementwise determinant of the log jacobian

### 4.4.26 Sylvester

**class Sylvester**(input_dim, count_transforms=1)

**Bases:** pyro.distributions.transforms.householder.Householder

An implementation of the Sylvester bijective transform of the Householder variety (Van den Berg Et Al., 2018),

$$y = x + QR \tanh(SQ^T x + b)$$

where $x$ are the inputs, $y$ are the outputs, $R, S \sim D \times D$ are upper triangular matrices for input dimension $D$, $Q \sim D \times D$ is an orthogonal matrix, and $b \sim D$ is learnable bias term.

The Sylvester transform is a generalization of `Planar`. In the Householder type of the Sylvester transform, the orthogonality of $Q$ is enforced by representing it as the product of Householder transformations.

Together with `TransformedDistribution` it provides a way to create richer variational approximations.

**Example usage:**

```python
>>> base_dist = dist.Normal(torch.zeros(10), torch.ones(10))
>>> transform = Sylvester(10, count_transforms=4)
>>> pyro.module("my_transform", transform)  # doctest: +SKIP
>>> flow_dist = dist.TransformedDistribution(base_dist, [transform])
>>> flow_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 the Sylvester transform can be scored.

References:

$Q(x)$
R()
S()
bijective = True
codomain = RealVector()
domain = RealVector()
dtanh_dx(x)
event_dim = 1
log_abs_det_jacobian(x, y)
  Calculates the elementwise determinant of the log Jacobian
reset_parameters2()

4.4.27 TransformModule

class TransformModule(*args, **kwargs)

  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.

4.4.28 ComposeTransformModule

class ComposeTransformModule(parts)

  This allows us to use a list of TransformModule in the same way as ComposeTransform. This is needed so that transform parameters are automatically registered by Pyro’s param store when used in PyroModule instances.

4.5 Transform Factories

Each Transform and TransformModule includes a corresponding helper function in lower case that inputs, at minimum, the input dimensions of the transform, and possibly additional arguments to customize the transform in an intuitive way. The purpose of these helper functions is to hide from the user whether or not the transform requires the construction of a hypernet, and if so, the input and output dimensions of that hypernet.

4.5.1 iterated

iterated(repeats, base_fn, *args, **kwargs)
  Helper function to compose a sequence of bijective transforms with potentially learnable parameters using ComposeTransformModule.

  Parameters
  • repeats – number of repeated transforms.
  • base_fn – function to construct the bijective transform.
• **args** – arguments taken by `base_fn`.
• **kwargs** – keyword arguments taken by `base_fn`.

Returns instance of `TransformModule`.

### 4.5.2 affine_autoregressive

`affine_autoregressive`(*input_dim*, *hidden_dims=None*, **kwargs)

A helper function to create an `AffineAutoregressive` object that takes care of constructing an autoregressive network with the correct input/output dimensions.

**Parameters**

- **input_dim** *(int)* – Dimension of input variable
- **hidden_dims** *(list[int]*) – The desired hidden dimensions of the autoregressive network. Defaults to using \([3*\text{input\_dim} + 1]\)
- **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
- **sigmoid_bias** *(float)* – A term to add the logit of the input when using the stable transform.
- **stable** *(bool)* – When true, uses the alternative “stable” version of the transform (see above).

### 4.5.3 affine_coupling

`affine_coupling`(*input_dim*, *hidden_dims=None*, *split_dim=None*, *dim=-1*, **kwargs)

A helper function to create an `AffineCoupling` object that takes care of constructing a dense network with the correct input/output dimensions.

**Parameters**

- **input_dim** *(int)* – Dimension(s) of input variable to permute. Note that when `dim < -1` this must be a tuple corresponding to the event shape.
- **hidden_dims** *(list[int]*) – The desired hidden dimensions of the dense network. Defaults to using \([10*\text{input\_dim}]\)
- **split_dim** *(int)* – The dimension to split the input on for the coupling transform. Defaults to using `input\_dim // 2`
- **dim** *(int)* – the tensor dimension on which to split. This value must be negative and defines the event dim as `abs(dim)`.
- **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
4.5.4 `batchnorm`

`batchnorm(input_dim, **kwargs)`

A helper function to create a `BatchNorm` object for consistency with other helpers.

**Parameters**

- `input_dim (int)` – Dimension of input variable
- `momentum (float)` – momentum parameter for updating moving averages
- `epsilon (float)` – small number to add to variances to ensure numerical stability

4.5.5 `block_autoregressive`

`block_autoregressive(input_dim, **kwargs)`

A helper function to create a `BlockAutoregressive` object for consistency with other helpers.

**Parameters**

- `input_dim (int)` – Dimension of input variable
- `hidden_factors (list)` – Hidden layer i has hidden_factors[i] hidden units per input dimension. This corresponds to both a and b in De Cao et al. (2019). The elements of hidden_factors must be integers.
- `activation (string)` – Activation function to use. One of ‘ELU’, ‘LeakyReLU’, ‘sigmoid’, or ‘tanh’.
- `residual (string)` – Type of residual connections to use. Choices are “None”, “normal” for $y + f(y)$, and “gated” for $\alpha y + (1 - \alpha) y$ for learnable parameter $\alpha$.

4.5.6 `conditional_affine_autoregressive`

`conditional_affine_autoregressive(input_dim, context_dim, hidden_dims=None, **kwargs)`

A helper function to create a `ConditionalAffineAutoregressive` object that takes care of constructing a dense network with the correct input/output dimensions.

**Parameters**

- `input_dim (int)` – Dimension of input variable
- `context_dim (int)` – Dimension of context variable
- `hidden_dims (list[int])` – The desired hidden dimensions of the dense network. Defaults to using [10*input_dim]
- `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
- `sigmoid_bias (float)` – A term to add the logit of the input when using the stable transform.
- `stable (bool)` – When true, uses the alternative “stable” version of the transform (see above).
4.5.7 conditional_affine_coupling

conditional_affine_coupling(input_dim, context_dim, hidden_dims=None, split_dim=None, dim=-1, **kwargs)

A helper function to create a `ConditionalAffineCoupling` object that takes care of constructing a dense network with the correct input/output dimensions.

Parameters

- input_dim (int) – Dimension of input variable
- context_dim (int) – Dimension of context variable
- hidden_dims (list[int]) – The desired hidden dimensions of the dense network. Defaults to using [10*input_dim]
- split_dim (int) – The dimension to split the input on for the coupling transform. Defaults to using input_dim // 2
- dim (int) – The tensor dimension on which to split. This value must be negative and defines the event dim as abs(dim).
- 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

4.5.8 conditional_generalized_channel_permute

conditional_generalized_channel_permute(context_dim, channels=3, hidden_dims=None)

A helper function to create a `ConditionalGeneralizedChannelPermute` object for consistency with other helpers.

Parameters

- channels (int) – Number of channel dimensions in the input.

4.5.9 conditional_householder

conditional_householder(input_dim, context_dim, hidden_dims=None, count_transforms=1)

A helper function to create a `ConditionalHouseholder` object that takes care of constructing a dense network with the correct input/output dimensions.

Parameters

- input_dim (int) – Dimension of input variable
- context_dim (int) – Dimension of context variable
- hidden_dims (list[int]) – The desired hidden dimensions of the dense network. Defaults to using [input_dim * 10, input_dim * 10]

4.5.10 conditional_matrix_exponential

conditional_matrix_exponential(input_dim, context_dim, hidden_dims=None, iterations=8, normalization='none', bound=None)

A helper function to create a `ConditionalMatrixExponential` object for consistency with other helpers.

Parameters
• **input_dim** *(int)* – Dimension of input variable

• **context_dim** *(int)* – Dimension of context variable

• **hidden_dims** *(list[int])* – The desired hidden dimensions of the dense network. Defaults to using `[input_dim * 10, input_dim * 10]`

• **iterations** *(int)* – the number of terms to use in the truncated power series that approximates matrix exponentiation.

• **normalization** *(string)* – One of ['none', 'weight', 'spectral'] normalization that selects what type of normalization to apply to the weight matrix. *weight* corresponds to weight normalization (Salimans and Kingma, 2016) and *spectral* to spectral normalization (Miyato et al, 2018).

• **bound** *(float)* – a bound on either the weight or spectral norm, when either of those two types of regularization are chosen by the `normalization` argument. A lower value for this results in fewer required terms of the truncated power series to closely approximate the exact value of the matrix exponential.

### 4.5.11 conditional_neural_autoregressive

`conditional_neural_autoregressive(input_dim, context_dim, hidden_dims=None, activation='sigmoid', width=16)`

A helper function to create a `ConditionalNeuralAutoregressive` object that takes care of constructing an autoregressive network with the correct input/output dimensions.

**Parameters**

• **input_dim** *(int)* – Dimension of input variable

• **context_dim** *(int)* – Dimension of context variable

• **hidden_dims** *(list[int])* – The desired hidden dimensions of the autoregressive network. Defaults to using `[3*input_dim + 1]`

• **activation** *(string)* – Activation function to use. One of ‘ELU’, ‘LeakyReLU’, ‘sigmoid’, or ‘tanh’.

• **width** *(int)* – The width of the “multilayer perceptron” in the transform (see paper). Defaults to 16

### 4.5.12 conditional_planar

`conditional_planar(input_dim, context_dim, hidden_dims=None)`

A helper function to create a `ConditionalPlanar` object that takes care of constructing a dense network with the correct input/output dimensions.

**Parameters**

• **input_dim** *(int)* – Dimension of input variable

• **context_dim** *(int)* – Dimension of context variable

• **hidden_dims** *(list[int])* – The desired hidden dimensions of the dense network. Defaults to using `[input_dim * 10, input_dim * 10]`
4.5.13 conditional_radial

conditional_radial(input_dim, context_dim, hidden_dims=None)

A helper function to create a ConditionalRadial object that takes care of constructing a dense network with the correct input/output dimensions.

Parameters

• input_dim (int) – Dimension of input variable
• context_dim (int) – Dimension of context variable
• hidden_dims (list[int]) – The desired hidden dimensions of the dense network. Defaults to using [input_dim * 10, input_dim * 10]

4.5.14 conditional_spline

conditional_spline(input_dim, context_dim, hidden_dims=None, count_bins=8, bound=3.0, order='linear')

A helper function to create a ConditionalSpline object that takes care of constructing a dense network with the correct input/output dimensions.

Parameters

• input_dim (int) – Dimension of input variable
• context_dim (int) – Dimension of context variable
• hidden_dims (list[int]) – The desired hidden dimensions of the dense network. Defaults to using [input_dim * 10, input_dim * 10]
• count_bins (int) – The number of segments comprising the spline.
• bound (float) – The quantity $K$ determining the bounding box, $[-K, K] \times [-K, K]$, of the spline.
• order (str) – One of ['linear', 'quadratic'] specifying the order of the spline.

4.5.15 conditional_spline_autoregressive

conditional_spline_autoregressive(input_dim, context_dim, hidden_dims=None, count_bins=8, bound=3.0, order='linear')

A helper function to create a ConditionalSplineAutoregressive object that takes care of constructing an autoregressive network with the correct input/output dimensions.

Parameters

• input_dim (int) – Dimension of input variable
• context_dim (int) – Dimension of context variable
• hidden_dims (list[int]) – The desired hidden dimensions of the autoregressive network. Defaults to using [input_dim * 10, input_dim * 10]
• count_bins (int) – The number of segments comprising the spline.
• bound (float) – The quantity $K$ determining the bounding box, $[-K, K] \times [-K, K]$, of the spline.
• order (str) – One of ['linear', 'quadratic'] specifying the order of the spline.
4.5.16 elu

elu()
A helper function to create an ELUTransform object for consistency with other helpers.

4.5.17 generalized_channel_permute

generalized_channel_permute(**kwargs)
A helper function to create a GeneralizedChannelPermute object for consistency with other helpers.

Parameters channels (int) – Number of channel dimensions in the input.

4.5.18 householder

householder (input_dim, count_transforms=None)
A helper function to create a Householder object for consistency with other helpers.

Parameters

- input_dim (int) – Dimension of input variable
- count_transforms (int) – number of applications of Householder transformation to apply.

4.5.19 leaky_relu

leaky_relu()
A helper function to create a LeakyReLUTransform object for consistency with other helpers.

4.5.20 matrix_exponential

matrix_exponential (input_dim, iterations=8, normalization='none', bound=None)
A helper function to create a MatrixExponential object for consistency with other helpers.

Parameters

- input_dim (int) – Dimension of input variable
- iterations (int) – the number of terms to use in the truncated power series that approximates matrix exponentiation.
- normalization (string) – One of ['none', 'weight', 'spectral'] normalization that selects what type of normalization to apply to the weight matrix. weight corresponds to weight normalization (Salimans and Kingma, 2016) and spectral to spectral normalization (Miyato et al, 2018).
- bound (float) – a bound on either the weight or spectral norm, when either of those two types of regularization are chosen by the normalization argument. A lower value for this results in fewer required terms of the truncated power series to closely approximate the exact value of the matrix exponential.
4.5.21 neural_autoregressive

neural_autoregressive \((\text{input\_dim, hidden\_dims=\text{None}, activation=}'\text{sigmoid}', \text{width=}16)\)

A helper function to create a NeuralAutoregressive object that takes care of constructing an autoregressive network with the correct input/output dimensions.

Parameters

- **input_dim (int)** – Dimension of input variable
- **hidden_dims (list[int])** – The desired hidden dimensions of the autoregressive network. Defaults to using \([3*\text{input\_dim} + 1]\)
- **activation (string)** – Activation function to use. One of ‘ELU’, ‘LeakyReLU’, ‘sigmoid’, or ‘tanh’.
- **width (int)** – The width of the “multilayer perceptron” in the transform (see paper). Defaults to 16.

4.5.22 permute

permute \((\text{input\_dim, permutation=\text{None}, dim=}1)\)

A helper function to create a Permute object for consistency with other helpers.

Parameters

- **input_dim (int)** – Dimension(s) of input variable to permute. Note that when \(\text{dim} < -1\) this must be a tuple corresponding to the event shape.
- **dim (int)** – the tensor dimension to permute. This value must be negative and defines the event dim as \(\text{abs(dim)}\).

4.5.23 planar

planar \((\text{input\_dim})\)

A helper function to create a Planar object for consistency with other helpers.

Parameters **input_dim (int)** – Dimension of input variable

4.5.24 polynomial

polynomial \((\text{input\_dim, hidden\_dims=\text{None}})\)

A helper function to create a Polynomial object that takes care of constructing an autoregressive network with the correct input/output dimensions.

Parameters

- **input_dim (int)** – Dimension of input variable
- **hidden_dims** – The desired hidden dimensions of the autoregressive network. Defaults to using \([\text{input\_dim} * 10]\)
4.5.25 radial

`radial(input_dim)`  
A helper function to create a `Radial` object for consistency with other helpers.

**Parameters**
- `input_dim (int)` – Dimension of input variable

4.5.26 spline

`spline(input_dim, **kwargs)`  
A helper function to create a `Spline` object for consistency with other helpers.

**Parameters**
- `input_dim (int)` – Dimension of input variable

4.5.27 spline_autoregressive

`spline_autoregressive(input_dim, hidden_dims=None, count_bins=8, bound=3.0, order='linear')`  
A helper function to create an `SplineAutoregressive` object that takes care of constructing an autoregressive network with the correct input/output dimensions.

**Parameters**
- `input_dim (int)` – Dimension of input variable
- `hidden_dims (list[int])` – The desired hidden dimensions of the autoregressive network. Defaults to using `[3*input_dim + 1]`
- `count_bins (int)` – The number of segments comprising the spline.
- `bound (float)` – The quantity $K$ determining the bounding box, $[-K, K] \times [-K, K]$, of the spline.
- `order (string)` – One of ['linear', 'quadratic'] specifying the order of the spline.

4.5.28 spline_coupling

`spline_coupling(input_dim, split_dim=None, hidden_dims=None, count_bins=8, bound=3.0)`  
A helper function to create a `SplineCoupling` object for consistency with other helpers.

**Parameters**
- `input_dim (int)` – Dimension of input variable

4.5.29 sylvester

`sylvester(input_dim, count_transforms=None)`  
A helper function to create a `Sylvester` object for consistency with other helpers.

**Parameters**
- `input_dim (int)` – Dimension of input variable
- `count_transforms` – Number of Sylvester operations to apply. Defaults to `input_dim // 2 + 1`. :type count_transforms: int
4.6 Constraints

Pyro’s constraints library extends \texttt{torch.distributions.constraints}.

4.6.1 Constraint

alias of \texttt{torch.distributions.constraints.Constraint}

4.6.2 IndependentConstraint

\begin{verbatim}
class IndependentConstraint (base_constraint, reinterpreted_batch_ndims)
    Wraps a constraint by aggregating over reinterpreted_batch_ndims-many dims in check(), so that
    an event is valid only if all its independent entries are valid.

    Parameters
        • base_constraint (torch.distributions.constraints.Constraint) –
          A base constraint whose entries are incidentally independent.
        • reinterpreted_batch_ndims (int) – The number of extra event dimensions that
          will be considered dependent.
\end{verbatim}

4.6.3 boolean

alias of \texttt{torch.distributions.constraints.boolean}

4.6.4 cat

alias of \texttt{torch.distributions.constraints.cat}

4.6.5 corr_cholesky_constraint

\texttt{corr_cholesky_constraint}

4.6.6 dependent

alias of \texttt{torch.distributions.constraints.dependent}

4.6.7 dependent_property

alias of \texttt{torch.distributions.constraints.dependent_property}

4.6.8 greater_than

alias of \texttt{torch.distributions.constraints.greater_than}
4.6.9 greater_than_eq

alias of torch.distributions.constraints.greater_than_eq

4.6.10 half_open_interval

alias of torch.distributions.constraints.half_open_interval

4.6.11 integer

integer

4.6.12 integer_interval

alias of torch.distributions.constraints.integer_interval

4.6.13 interval

alias of torch.distributions.constraints.interval

4.6.14 is_dependent

alias of torch.distributions.constraints.is_dependent

4.6.15 less_than

alias of torch.distributions.constraints.less_than

4.6.16 lower_cholesky

alias of torch.distributions.constraints.lower_cholesky

4.6.17 lower_triangular

alias of torch.distributions.constraints.lower_triangular

4.6.18 nonnegative_integer

alias of torch.distributions.constraints.nonnegative_integer

4.6.19 ordered_vector

ordered_vector
4.6.20 positive
alias of torch.distributions.constraints.positive

4.6.21 positive_definite
alias of torch.distributions.constraints.positive_definite

4.6.22 positive_integer
alias of torch.distributions.constraints.positive_integer

4.6.23 real
alias of torch.distributions.constraints.real

4.6.24 real_vector
alias of torch.distributions.constraints.real_vector

4.6.25 simplex
alias of torch.distributions.constraints.simplex

4.6.26 stack
alias of torch.distributions.constraints.stack

4.6.27 unit_interval
alias of torch.distributions.constraints.unit_interval
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.

## 5.1 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`.
- in general parameters are associated with both constrained and unconstrained values. for example, under the hood a parameter that is constrained to be positive is represented as an unconstrained tensor in log space.
clear()
Clear the ParamStore

items()
Iterate over (name, constrained_param) pairs. Note that constrained_param is in the constrained (i.e. user-facing) space.

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:

```python
dict_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** *(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. Note that, in the event the parameter is constrained, unconstrained_value is in the unconstrained space implicitly used by the constraint.

get_all_param_names()

replace_param(param_name, new_param, old_param)

get_param(name, init_tensor=None, constraint=Real(), event_dim=None)
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
- **event_dim** *(int)* – (ignored)

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:

```python
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)
Neural Networks

The module `pyro.nn` provides implementations of neural network modules that are useful in the context of deep probabilistic programming.

6.1 Pyro Modules

Pyro includes a class `PyroModule`, a subclass of `torch.nn.Module`, whose attributes can be modified by Pyro effects. To create a poutine-aware attribute, use either the `PyroParam` struct or the `PyroSample` struct:

```python
my_module = PyroModule()
my_module.x = PyroParam(torch.tensor(1.), constraint=constraints.positive)
my_module.y = PyroSample(dist.Normal(0, 1))
```

### class PyroParam

Bases: `pyro.nn.module.PyroParam`

Declares a Pyro-managed learnable attribute of a `PyroModule`, similar to `pyro.param`.

This can be used either to set attributes of `PyroModule` instances:

```python
assert isinstance(my_module, PyroModule)
my_module.x = PyroParam(torch.zeros(4))  # eager
my_module.y = PyroParam(lambda: torch.randn(4))  # lazy
my_module.z = PyroParam(torch.ones(4),
                          constraint=constraints.positive,
                          event_dim=1)  # eager
```

or EXPERIMENTALLY as a decorator on lazy initialization properties:

```python
class MyModule(PyroModule):
    @PyroParam
    def x(self):
        return torch.zeros(4)
```

(continues on next page)
@PyroParam
def y(self):
    return torch.randn(4)

@PyroParam(constraint=constraints.real, event_dim=1)
def z(self):
    return torch.ones(4)

def forward(self):
    return self.x + self.y + self.z  # accessed like a @property

Parameters

- **init_value** *(torch.Tensor or callable returning a torch.Tensor or None)* – Either a tensor for eager initialization, a callable for lazy initialization, or None for use as a decorator.
- **constraint** *(Constraint)* – torch constraint, defaults to `constraints.real`.
- **event_dim** *(int)* – (optional) number of rightmost dimensions unrelated to batching. Dimension to the left of this will be considered batch dimensions; if the param statement is inside a subsampled plate, then corresponding batch dimensions of the parameter will be correspondingly subsampled. If unspecified, all dimensions will be considered event dims and no subsampling will be performed.

class PyroSample(prior)
Bases: pyro.nn.module.PyroSample

Declares a Pyro-managed random attribute of a PyroModule, similar to `pyro.sample`.

This can be used either to set attributes of PyroModule instances:

```python
assert isinstance(my_module, PyroModule)
my_module.x = PyroSample(Normal(0, 1))  # independent
my_module.y = PyroSample(lambda self: Normal(self.x, 1))  # dependent
```

or EXPERIMENTALLY as a decorator on lazy initialization methods:

```python
class MyModule(PyroModule):
    @PyroSample
def x(self):
        return Normal(0, 1)  # independent

    @PyroSample
def y(self):
        return Normal(self.x, 1)  # dependent

    def forward(self):
        return self.y  # accessed like a @property
```

Parameters **prior** – distribution object or function that inputs the PyroModule instance `self` and returns a distribution object.

class PyroModule(name="")
Bases: torch.nn.modules.module.Module
Subclass of `torch.nn.Module` whose attributes can be modified by Pyro effects. Attributes can be set using helpers `PyroParam` and `PyroSample`, and methods can be decorated by `pyro_method()`.

### Parameters

To create a Pyro-managed parameter attribute, set that attribute using either `torch.nn.Parameter` (for unconstrained parameters) or `PyroParam` (for constrained parameters). Reading that attribute will then trigger a `pyro.param` statement. For example:

```python
# Create Pyro-managed parameter attributes.
my_module = PyroModule()
my_module.loc = nn.Parameter(torch.tensor(0.))
my_module.scale = PyroParam(torch.tensor(1.),
                            constraint=constraints.positive)

# Read the attributes.
loc = my_module.loc  # Triggers a pyro.param statement.
scale = my_module.scale  # Triggers another pyro.param statement.
```

Note that, unlike normal `torch.nn.Module` s, `PyroModule` s should not be registered with `pyro.module` statements. `PyroModule` s can contain other `PyroModule` s and normal `torch.nn.Module` s. Accessing a normal `torch.nn.Module` attribute of a `PyroModule` triggers a `pyro.module` statement. If multiple `PyroModule` s appear in a single Pyro model or guide, they should be included in a single root `PyroModule` for that model.

`PyroModule` s synchronize data with the param store at each `setattr`, `getattr`, and `delattr` event, based on the nested name of an attribute:

- Setting `mod.x = _x_init` tries to read `x` from the param store. If a value is found in the param store, that value is copied into `mod` and `_x_init` is ignored; otherwise `_x_init` is copied into both `mod` and the param store.

- Reading `mod.x` tries to read `x` from the param store. If a value is found in the param store, that value is copied into `mod`; otherwise `mod`'s value is copied into the param store. Finally `mod` and the param store agree on a single value to return.

- Deleting `del mod.x` removes a value from both `mod` and the param store.

Note two `PyroModule` of the same name will both synchronize with the global param store and thus contain the same data. When creating a `PyroModule`, then deleting it, then creating another with the same name, the latter will be populated with the former’s data from the param store. To avoid this persistence, either `pyro.clear_param_store()` or call `clear()` before deleting a `PyroModule`.

`PyroModule` s can be saved and loaded either directly using `torch.save()` / `torch.load()` or indirectly using the param store’s `save()` / `load()`. Note that `torch.load()` will be overridden by any values in the param store, so it is safest to `pyro.clear_param_store()` before loading.

### Samples

To create a Pyro-managed random attribute, set that attribute using the `PyroSample` helper, specifying a prior distribution. Reading that attribute will then trigger a `pyro.sample` statement. For example:

```python
# Create Pyro-managed random attributes.
my_module.x = PyroSample(dist.Normal(0, 1))
my_module.y = PyroSample(lambda self: dist.Normal(self.loc, self.scale))

# Sample the attributes.
x = my_module.x  # Triggers a pyro.sample statement.
y = my_module.y  # Triggers one pyro.sample + two pyro.param statements.
```
Sampling is cached within each invocation of .\_\_call\_\_() or method decorated by pyro\_method(). Because sample statements can appear only once in a Pyro trace, you should ensure that traced access to sample attributes is wrapped in a single invocation of .\_\_call\_\_() or method decorated by pyro\_method().

To make an existing module probabilistic, you can create a subclass and overwrite some parameters with PyroSample:

```python
class RandomLinear(nn.Linear, PyroModule):  # used as a mixin
def __init__(self, in_features, out_features):
    super().__init__(in_features, out_features)
    self.weight = PyroSample(
        lambda self: dist.Normal(0, 1)
            .expand([self.out_features, self.in_features])
            .to_event(2))
```

**Mixin classes**

PyroModule can be used as a mixin class, and supports simple syntax for dynamically creating mixins, for example the following are equivalent:

```python
# Version 1. create a named mixin class
class PyroLinear(nn.Linear, PyroModule):
    pass

m.linear = PyroLinear(m, n)

# Version 2. create a dynamic mixin class
m.linear = PyroModule[nn.Linear](m, n)
```

This notation can be used recursively to create Bayesian modules, e.g.:

```python
model = PyroModule[nn.Sequential](
    PyroModule[nn.Linear](28 * 28, 100),
    PyroModule[nn.Linear](100, 100),
    PyroModule[nn.Linear](100, 10),
)
assert isinstance(model, nn.Sequential)
assert isinstance(model, PyroModule)

# Now we can be Bayesian about weights in the first layer.
model[0].weight = PyroSample(
    prior=dist.Normal(0, 1).expand([28 * 28, 100]).to_event(2))
guide = AutoDiagonalNormal(model)
```

Note that PyroModule[...] does not recursively mix in PyroModule to submodules of the input Module; hence we needed to wrap each submodule of the nn.Sequential above.

**Parameters**

- **name (str)** – Optional name for a root PyroModule. This is ignored in sub-PyroModules of another PyroModule.

- **add_module (name, module)**
  Adds a child module to the current module.

- **named_pyro_params (prefix=", recurse=True)**
  Returns an iterator over PyroModule parameters, yielding both the name of the parameter as well as the parameter itself.
Parameters

- **prefix (str)** – prefix to prepend to all parameter names.
- **recurse (bool)** – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Returns a generator which yields tuples containing the name and parameter

### pyro_method (fn)

Decorator for top-level methods of a PyroModule to enable pyro effects and cache pyro.sample statements.

This should be applied to all public methods that read Pyro-managed attributes, but is not needed for .forward().

### clear (mod)

Removes data from both a PyroModule and the param store.

- **mod (PyroModule)** – A module to clear.

### to_pyro_module_(m, recurse=True)

Converts an ordinary torch.nn.Module instance to a PyroModule in-place.

This is useful for adding Pyro effects to third-party modules: no third-party code needs to be modified. For example:

```python
model = nn.Sequential(
    nn.Linear(28 * 28, 100),
    nn.Sigmoid(),
    nn.Linear(100, 100),
    nn.Sigmoid(),
    nn.Linear(100, 10),
)
to_pyro_module_(model)
assert isinstance(model, PyroModule[nn.Sequential])
assert isinstance(model[0], PyroModule[nn.Linear])

# Now we can attempt to be fully Bayesian:
for m in model.modules():
    for name, value in list(m.named_parameters(recurse=False)):
        setattr(m, name, PyroSample(prior=dist.Normal(0, 1)
                                      .expand(value.shape)
                                      .to_event(value.dim())))

guide = AutoDiagonalNormal(model)
```

Parameters

- **m (torch.nn.Module)** – A module instance.
- **recurse (bool)** – Whether to convert submodules to PyroModules.

### 6.2 AutoRegressiveNN

#### class AutoRegressiveNN

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input_dim</strong></td>
</tr>
<tr>
<td><strong>hidden_dims</strong></td>
</tr>
<tr>
<td><strong>param_dims</strong>=[1, 1]</td>
</tr>
<tr>
<td><strong>permutation</strong>=None</td>
</tr>
<tr>
<td><strong>skip_connections</strong>=False</td>
</tr>
<tr>
<td><strong>nonlinearity</strong>=ReLU()</td>
</tr>
</tbody>
</table>

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

---

6.2. AutoRegressiveNN

Example usage:

```python
>>> 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 variable
- `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:


```python
def forward(x)
    The forward method
```

### 6.3 DenseNN

**class DenseNN** `input_dim, hidden_dims, param_dims=[1, 1], nonlinearity=ReLU()`

**Bases:** pyro.nn.dense_nn.ConditionalDenseNN

An implementation of a simple dense feedforward network, for use in, e.g., some conditional flows such as pyro.distributions.transforms.ConditionalPlanarFlow and other unconditional flows such as `pyro.distributions.transforms.AffineCoupling` that do not require an autoregressive network.

Example usage:

```python
>>> input_dim = 10
>>> context_dim = 5
>>> z = torch.rand(100, context_dim)
```
>>> nn = DenseNN(context_dim, [50], param_dims=[1, input_dim, input_dim])

>>> a, b, c = nn(z)  # parameters of size (100, 1), (100, 10), (100, 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 \((p_n,\text{input_dim})\) for \(p_n\) in \(\text{param_dims}\) when \(p_n > 1\) and dimension \((\text{input_dim})\) when \(p_n == 1\). The default is \([1, 1]\), i.e. output two parameters of dimension \((\text{input_dim})\).
- **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.

**forward** *(x)*

The forward method

### 6.4 ConditionalAutoRegressiveNN

**class** **ConditionalAutoRegressiveNN** *(input_dim, context_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 that can input an additional context variable. (See Reference [2] Section 3.3 for an explanation of how the conditional MADE architecture works.)

Example usage:

```python
>>> x = torch.randn(100, 10)
>>> y = torch.randn(100, 5)
>>> arn = ConditionalAutoRegressiveNN(10, 5, [50], param_dims=[1])
>>> p = arn(x, context=y)  # 1 parameters of size (100, 10)
>>> arn = ConditionalAutoRegressiveNN(10, 5, [50], param_dims=[1, 1])
>>> m, s = arn(x, context=y)  # 2 parameters of size (100, 10)
>>> arn = ConditionalAutoRegressiveNN(10, 5, [50], param_dims=[1, 5, 3])
>>> a, b, c = arn(x, context=y)  # 3 parameters of sizes, (100, 1, 10), (100, 5, 10), (100, 3, 10)
```

Parameters

- **input_dim** *(int)* – the dimensionality of the input variable
- **context_dim** *(int)* – the dimensionality of the context variable
- **hidden_dims** *(list[int])* – the dimensionality of the hidden units per layer
- **param_dims** *(list[int])* – shape the output into parameters of dimension \((p_n, \text{input_dim})\) for \(p_n\) in \(\text{param_dims}\) when \(p_n > 1\) and dimension \((\text{input_dim})\) when \(p_n == 1\). The default is \([1, 1]\), i.e. output two parameters of dimension \((\text{input_dim})\), 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:

```python
def forward(x, context=None)
    # The forward method

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

### 6.5 ConditionalDenseNN

```python
class ConditionalDenseNN(input_dim, context_dim, hidden_dims, param_dims=[1, 1], nonlinearity=ReLU())
```

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

An implementation of a simple dense feedforward network taking a context variable, for use in, e.g., some conditional flows such as `pyro.distributions.transforms.ConditionalAffineCoupling`.

Example usage:

```python
>>> input_dim = 10
>>> context_dim = 5
>>> x = torch.rand(100, input_dim)
>>> z = torch.rand(100, context_dim)
>>> nn = ConditionalDenseNN(input_dim, context_dim, [50], param_dims=[1, input_dim])
>>> a, b, c = nn(x, context=z)  # parameters of size (100, 1), (100, 10), (100, 10)
```

**Parameters**

- **input_dim** (`int`) – the dimensionality of the input
- **context_dim** (`int`) – the dimensionality of the context variable
- **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,) for p_n in param_dims when p_n > 1 and dimension () when p_n == 1. The default is [1, 1], i.e. output two parameters of dimension ().
- **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.

```python
forward(x, context)
```

The forward method
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.

### 7.1 Pyro Optimizers

```python
class PyroOptim(optim_constructor, optim_args, clip_args=None):
    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
    • clip_args – a dictionary of clip_norm and/or clip_value args or a callable that returns such dictionaries

__call__(params, *args, **kwargs)

    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` *(str)* – file name to save to

Save optimizer state to disk

```python
load(filename)
```

Parameters `filename` *(str)* – file name to load from

Load optimizer state from disk

**AdagradRMSProp** *(optim_args)*

Wraps `pyro.optim.adagrad_rmsprop.AdagradRMSProp` with `PyroOptim`.

**ClippedAdam** *(optim_args)*

Wraps `pyro.optim.clipped_adam.ClippedAdam` with `PyroOptim`.

**DCTAdam** *(optim_args)*

Wraps `pyro.optim.dct_adam.DCTAdam` with `PyroOptim`.

**class PyroLRScheduler**(scheduler_constructor, optim_args, clip_args=None)*

Bases: `pyro.optim.optim.PyroOptim`

A wrapper for `lr_scheduler` objects that adjusts learning rates for dynamically generated parameters.

Parameters

- **scheduler_constructor** – a `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
- **clip_args** – a dictionary of clip_norm and/or clip_value args or a callable that returns such dictionaries.

Example:

```python
optimizer = torch.optim.SGD
scheduler = pyro.optim.ExponentialLR({'optimizer': optimizer, 'optim_args': {'lr →': 0.01}, 'gamma': 0.1})
svi = SVI(model, guide, scheduler, loss=TraceGraph_ELBO())
for i in range(epochs):
    for minibatch in DataLoader(dataset, batch_size):
        svi.step(minibatch)
        scheduler.step()
```

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

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

Takes the same arguments as the PyTorch scheduler (e.g. optional loss for ReduceLROnPlateau)

**class AdagradRMSProp**(params, eta=1.0, delta=1e-16, t=0.1)*

Bases: `torch.optim.optimizer.Optimizer`

Implements a mash-up of the Adagrad algorithm and RMSProp. For the precise update equation see equations 10 and 11 in reference [1].


Arguments

Parameters
• **params** – iterable of parameters to optimize or dicts defining parameter groups
• **eta** *(float)* – sets the step size scale (optional; default: 1.0)
• **t** *(float)* – t, optional): momentum parameter (optional; default: 0.1)
• **delta** *(float)* – modulates the exponent that controls how the step size scales (optional: default: 1e-16)

    `share_memory()`

    `step(closure=None)`
    Performs a single optimization step.

    **Parameters**
    closure – A (optional) closure that reevaluates the model and returns the loss.

class **ClippedAdam** *(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, clip_norm=10.0, lrd=1.0)*
Bases: torch.optim.optimizer.Optimizer

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

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

Reference


    `step(closure=None)`
    Performs a single optimization step.

class **HorovodOptimizer** *(pyro_optim, **horovod_kwargs)*
Bases: pyro.optim.optim.PyroOptim

Distributed wrapper for a PyroOptim optimizer.

This class wraps a PyroOptim object similar to the way horovod.torch.DistributedOptimizer() wraps a torch.optim.Optimizer.

**Note:** This requires horovod.torch to be installed, e.g. via `pip install pyro[horovod]`. For details see [https://horovod.readthedocs.io/en/stable/install.html](https://horovod.readthedocs.io/en/stable/install.html)

    Param A Pyro optimizer instance.

    Parameters
    **horovod_kwargs** – Extra parameters passed to horovod.torch.DistributedOptimizer().

7.1. Pyro Optimizers
__call__ (params, *args, **kwargs)

7.2 PyTorch Optimizers

**Adadelta** (optim_args, clip_args=None)
Wraps torch.optim.Adadelta with PyroOptim.

**Adagrad** (optim_args, clip_args=None)
Wraps torch.optim.Adagrad with PyroOptim.

**Adam** (optim_args, clip_args=None)
Wraps torch.optim.Adam with PyroOptim.

**AdamW** (optim_args, clip_args=None)
Wraps torch.optim.AdamW with PyroOptim.

**SparseAdam** (optim_args, clip_args=None)
Wraps torch.optim.SparseAdam with PyroOptim.

**Adamax** (optim_args, clip_args=None)
Wraps torch.optim.Adamax with PyroOptim.

**ASGD** (optim_args, clip_args=None)
Wraps torch.optim.ASGD with PyroOptim.

**SGD** (optim_args, clip_args=None)
Wraps torch.optim.SGD with PyroOptim.

**Rprop** (optim_args, clip_args=None)
Wraps torch.optim.Rprop with PyroOptim.

**RMSprop** (optim_args, clip_args=None)
Wraps torch.optim.RMSprop with PyroOptim.

**LambdaLR** (optim_args, clip_args=None)
Wraps torch.optim.LambdaLR with PyroLRScheduler.

**MultiplicativeLR** (optim_args, clip_args=None)
Wraps torch.optim.MultiplicativeLR with PyroLRScheduler.

**StepLR** (optim_args, clip_args=None)
Wraps torch.optim.StepLR with PyroLRScheduler.

**MultiStepLR** (optim_args, clip_args=None)
Wraps torch.optim.MultiStepLR with PyroLRScheduler.

**ExponentialLR** (optim_args, clip_args=None)
Wraps torch.optim.ExponentialLR with PyroLRScheduler.

**CosineAnnealingLR** (optim_args, clip_args=None)
Wraps torch.optim.CosineAnnealingLR with PyroLRScheduler.

**ReduceLROnPlateau** (optim_args, clip_args=None)
Wraps torch.optim.ReduceLROnPlateau with PyroLRScheduler.

**CyclicLR** (optim_args, clip_args=None)
Wraps torch.optim.CyclicLR with PyroLRScheduler.

**CosineAnnealingWarmRestarts** (optim_args, clip_args=None)
Wraps torch.optim.CosineAnnealingWarmRestarts with PyroLRScheduler.
OneCycleLR(optim_args, clip_args=None)
Wraps torch.optim.OneCycleLR with PyroLRScheduler.

7.3 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:

```python
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.
**class TorchMultiOptimizer**(optim_constructor, optim_args)

Facade to wrap Optimizer objects in a MultiOptimizer interface.

**class MixedMultiOptimizer**(parts)

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.

**class Newton**(trust_radii={})

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)
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.

## 8.1 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:

```python
>>> 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`:

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

We can also use handlers as decorators:

```python
>>> @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, *args, **kwargs)*

Convenient wrapper of *BlockMessenger*

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 = pyro.poutine.trace(fn)
>>> fn_outer = pyro.poutine.trace(pyro.poutine.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_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.
• **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.

• **hide_all**(bool) – hide all sites

• **expose_all**(bool) – expose all sites normally

• **hide**(list) – list of site names to hide

• **expose**(list) – list of site names to be exposed while all others hidden

• **hide_types**(list) – list of site types to be hidden

• **expose_types**(list) – list of site types to be exposed while all others hidden

**Returns** stochastic function decorated with a `BlockMessenger`

**broadcast**(fn=None, *args, **kwargs)

Convenient wrapper of `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`.

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.

```python
>>> 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
```

**collapse**(fn=None, *args, **kwargs)

Convenient wrapper of `CollapseMessenger`

EXPERIMENTAL Collapses all sites in the context by lazily sampling and attempting to use conjugacy relations. If no conjugacy is known this will fail. Code using the results of sample sites must be written to accept Funsors rather than Tensors. This requires `funsor` to be installed.

**condition**(fn=None, *args, **kwargs)

Convenient wrapper of `ConditionMessenger`

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:

```python
>>> def model(x):
...     s = pyro.param("s", torch.tensor(0.5))
```
Pyro Documentation

(continued from previous page)

...
...

z = pyro.sample("z", dist.Normal(x, s))
return z ** 2

To observe a value for site z, we can write
>>> conditioned_model = pyro.poutine.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, *args, **kwargs)
Convenient wrapper of DoMessenger
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 as if they were hard-coded to those values and introduce fresh sample
sites with the same names whose values do not propagate.
Composes freely with condition() to represent counterfactual distributions over potential outcomes. See
Single World Intervention Graphs [1] for additional details and theory.
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 = pyro.poutine.do(model, data={"z": torch.tensor(1.)})

This is equivalent to replacing z = pyro.sample(“z”, . . . ) with z = torch.tensor(1.) and introducing a fresh
sample site pyro.sample(“z”, . . . ) whose value is not used elsewhere.
References
[1] Single World Intervention Graphs: A Primer, Thomas Richardson, James Robins
Parameters
• fn – a stochastic function (callable containing Pyro primitive calls)
• data – a dict mapping sample site names to interventions
Returns stochastic function decorated with a DoMessenger
enum(fn=None, *args, **kwargs)
Convenient wrapper of EnumMessenger
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 or None.
156

Chapter 8. Poutine (Effect handlers)


escape (fn=None, *args, **kwargs)
Convenient wrapper of EscapeMessenger
Messenger that does a nonlocal exit by raising a util.NonlocalExit exception

infer_config (fn=None, *args, **kwargs)
Convenient wrapper of InferConfigMessenger
Given a callable fn that contains Pyro primitive calls and a callable config_fn 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, *args, **kwargs)
Convenient wrapper of LiftMessenger
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:

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

```python
>>> tr = pyro.poutine.trace(lifted_model).get_trace(0.0)
>>> tr.nodes["s"]["type"] == "sample"
True
>>> tr2 = pyro.poutine.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, dim=None, name=None)
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”
- **dim (int)** – An optional dimension to use for this independence index. Interface stub, behavior not yet implemented.
- **name (str)** – An optional unique name to help inference algorithms match `pyro.markov()` sites between models and guides. Interface stub, behavior not yet implemented.

`mask (fn=None, *args, **kwargs)`

Convenient wrapper of `MaskMessenger`

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.BoolTensor`) – 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

`reparam (fn=None, *args, **kwargs)`

Convenient wrapper of `ReparamMessenger`

Reparameterizes each affected sample site into one or more auxiliary sample sites followed by a deterministic transformation [1].

To specify reparameterizers, pass a `config` dict or callable to the constructor. See the `pyro.infer.reparam` module for available reparameterizers.

Note some reparameterizers can examine the `*args, **kwargs` inputs of functions they affect; these reparameterizers require using `poutine.reparam` as a decorator rather than as a context manager.
Parameters

- **config** (dict or callable) – Configuration, either a dict mapping site name to Reparameterizer, or a function mapping site to Reparameterizer or None.

### replay

```
replay(fn=None, *args, **kwargs)
```

Convenient wrapper of `ReplayMessenger`

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:

```python
>>> 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:

```python
>>> old_trace = pyro.poutine.trace(model).get_trace(1.0)
>>> replayed_model = pyro.poutine.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

```
scale(fn=None, *args, **kwargs)
```

Convenient wrapper of `ScaleMessenger`

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:

```python
>>> 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:

```python
>>> scaled_model = pyro.poutine.scale(model, scale=0.5)
>>> scaled_tr = pyro.poutine.trace(scaled_model).get_trace(0.0)
>>> unscaled_tr = pyro.poutine.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*

### seed

**seed** *(fn=None, *args, **kwargs)*

Convenient wrapper of *SeedMessenger*

Handler to set the random number generator to a pre-defined state by setting its seed. This is the same as calling `pyro.set_rng_seed()` before the call to `fn`. This handler has no additional effect on primitive statements on the standard Pyro backend, but it might intercept `pyro.sample` calls in other backends, e.g. the NumPy backend.

**Parameters**

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

• **rng_seed** *(int)* – rng seed.

### trace

**trace** *(fn=None, *args, **kwargs)*

Convenient wrapper of *TraceMessenger*

Return a handler that records the inputs and outputs of primitive calls and their dependencies.

Consider the following Pyro program:

```python
>>> 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.

```python
>>> 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]
```

**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*

### uncondition

**uncondition** *(fn=None, *args, **kwargs)*

Convenient wrapper of *UnconditionMessenger*

Messenger to force the value of observed nodes to be sampled from their distribution, ignoring observations.

### config Enumerate

**config Enumerate** *(guide=None, default='parallel', expand=False, num_samples=None, tmc='diagonal')*

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:

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

or as a decorator:

```python
@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.
- **tmc** *(string or None)* – “mixture” or “diagonal” strategies to use in Tensor Monte Carlo

Returns an annotated guide

Return type callable

8.2 Trace

```
class Trace(graph_type='flat')
```

Graph data structure denoting the relationships amongst different pyro primitives in the execution trace.

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:
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"]
```

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.

Parameters `graph_type` (`string`) – string specifying the kind of trace graph to construct

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

    Adds a site to the trace.

    Raises an error when attempting to add a duplicate node instead of silently overwriting.

Parameters `site_filter` (`function`) – computes 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.

Parameters `compute_score_parts` (`function`) – computes 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.

Parameters `copy` (`function`) – makes a shallow copy of self with nodes and edges preserved.

Parameters `detach_` (`function`) – Detach values (in-place) at each sample site of the trace.

Parameters `edges`
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

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

predecessors(site_name)

remove_node(site_name)

reparameterized_nodes

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

stochastic_nodes

Returns a list of names of sample sites

successors(site_name)

symbolize_dims(plate_to_symbol=None)

Assign unique symbols to all tensor dimensions.

topological_sort(reverse=False)

Return a list of nodes (site names) in topologically sorted order.

Parameters reverse (bool) – Return the list in reverse order.

Returns list of topologically sorted nodes (site names).

8.3 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.

Parameters msg – a message to be processed

Returns None

effectful (fn=None, type=None)
Wrapper for calling apply_stack() to apply any active effects.

Parameters
• fn – function or callable that performs an effectful computation
• type (str) – the type label of the operation, e.g. “sample”

8.4 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_factor(site)

Determines whether a trace site originated from a factor statement.

site_is_subsample(site)

Determines whether a trace site originated from a subsample statement inside an plate.

8.5 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.
8.5.1 Messenger

```python
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 effectful())
        • post (bool) – if True, use this operation as postprocess

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

        Example:
        ```python
        @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 effectful())

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

        Example:
        ```python
        SomeMessengerClass.unregister(some_function, "name")
        ```

    block_messengers
        (predicate)

        EXPERIMENTAL Context manager to temporarily remove matching messengers from the _PYRO_STACK. Note this does not call the .__exit__() and .__enter__() methods.

        This is useful to selectively block enclosing handlers.

        Parameters predicate (callable) – A predicate mapping messenger instance to boolean.

        This mutes all messengers m for which bool(predicate(m)) is True.

        Yields A list of matched messengers that are blocked.
```
8.5.2 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 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”:

```python
>>> def fn():
...     a = pyro.sample("a", dist.Normal(0., 1.))
...     return pyro.sample("b", dist.Normal(a, 1.))
>>> fn_inner = pyro.poutine.trace(fn)
>>> fn_outer = pyro.poutine.trace(pyro.poutine.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_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.
- **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.
- **hide_all** (bool) – hide all sites
- **expose_all** (bool) – expose all sites normally
- **hide** (list) – list of site names to hide
- **expose** (list) – list of site names to be exposed while all others hidden
- **hide_types** (list) – list of site types to be hidden
- **expose_types** (list) – list of site types to be exposed while all others hidden

Returns stochastic function decorated with a BlockMessenger
8.5.3 BroadcastMessenger

class BroadcastMessenger

Bases: pyro.poutine.messenger.Messenger

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.

```python
>>> 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
```

8.5.4 CollapseMessenger

class CollapseMessenger(*args, **kwargs)

Bases: pyro.poutine.trace_messenger.TraceMessenger

EXPERIMENTAL Collapses all sites in the context by lazily sampling and attempting to use conjugacy relations. If no conjugacy is known this will fail. Code using the results of sample sites must be written to accept Funsors rather than Tensors. This requires funsor to be installed.

8.5.5 ConditionMessenger

class ConditionMessenger(data)

Bases: pyro.poutine.messenger.Messenger

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:

```python
>>> 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

```python
>>> conditioned_model = pyro.poutine.condition(model, data={"z": torch.tensor(1.)})
```
This is equivalent to adding \textit{obs=value} as a keyword argument to \texttt{pyro.sample(“z”, …)} in \texttt{model}.

**Parameters**
- \texttt{fn} – a stochastic function (callable containing Pyro primitive calls)
- \texttt{data} – a dict or a \texttt{Trace}

**Returns** stochastic function decorated with a \texttt{ConditionMessenger}

### 8.5.6 DoMessenger

**class DoMessenger**(\texttt{data})

**Bases:** \texttt{pyro.poutine.messenger.Messenger}

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 as if they were hard-coded to those values and introduce fresh sample sites with the same names whose values do not propagate.

Composes freely with \texttt{condition()} to represent counterfactual distributions over potential outcomes. See Single World Intervention Graphs [1] for additional details and theory.

Consider the following Pyro program:

```python
>>> 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 \texttt{z}, we can write

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

This is equivalent to replacing \texttt{z = pyro.sample("z", …)} with \texttt{z = torch.tensor(1.)} and introducing a fresh sample site \texttt{pyro.sample("z", …)} whose value is not used elsewhere.

**References**


**Parameters**
- \texttt{fn} – a stochastic function (callable containing Pyro primitive calls)
- \texttt{data} – a dict mapping sample site names to interventions

**Returns** stochastic function decorated with a \texttt{DoMessenger}

### 8.5.7 EnumMessenger

**class EnumMessenger** (\texttt{first_available_dim=None})

**Bases:** \texttt{pyro.poutine.messenger.Messenger}

Enumerates in parallel over discrete sample sites marked \texttt{infer={"enumerate": "parallel"}}.

**Parameters**
- \texttt{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 or None.

**enumerate_site** (\texttt{msg})
8.5.8 EscapeMessenger

```python
class EscapeMessenger(escape_fn)
    Bases: pyro.poutine.messenger.Messenger
    Messenger that does a nonlocal exit by raising a util.NonlocalExit exception
```

8.5.9 IndepMessenger

```python
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:
    ```python
    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])
    ```
```

8.5.10 InferConfigMessenger

```python
class InferConfigMessenger(config_fn)
    Bases: pyro.poutine.messenger.Messenger
    Given a callable fn that contains Pyro primitive calls and a callable config_fn 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
```

8.5.11 LiftMessenger

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

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

```  

```python
>>> lifted_model = pyro.poutine.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))`:

```python
>>> tr = pyro.poutine.trace(lifted_model).get_trace(0.0)
>>> tr.nodes["s"]["type"] == "sample"
True
>>> tr2 = pyro.poutine.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`

### 8.5.12 MarkovMessenger

**class** MarkovMessenger(*history=1, keep=False, dim=None, name=None*)

**Bases:** pyro.poutine.reentrant_messenger.ReentrantMessenger

Markov dependency declaration.

This is a statistical equivalent of a memory management arena.

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 shared ancestors).
- **dim** (*int*) – An optional dimension to use for this independence index. Interface stub, behavior not yet implemented.
- **name** (*str*) – An optional unique name to help inference algorithms match `pyro.markov()` sites between models and guides. Interface stub, behavior not yet implemented.

**generator** (*iterable*)

8.5. Messengers
8.5.13 MaskMessenger

class MaskMessenger(mask)
   Bases: pyro.poutine.messenger.Messenger

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.BoolTensor) – a \{0,1\}-valued masking tensor (1 includes a site, 0
     excludes a site)

Returns stochastic function decorated with a MaskMessenger

8.5.14 PlateMessenger

class PlateMessenger(name=None, size=None, subsample_size=None, subsample=None, dim=None, use_cuda=None, device=None)
   Bases: pyro.poutine.subsample_messenger.SubsampleMessenger

Swiss army knife of broadcasting amazingness: combines shape inference, independence annotation, and sub-
sampling

block_plate(name=None, dim=None)
   EXPERIMENTAL Context manager to temporarily block a single enclosing plate.

This is useful for sampling auxiliary variables or lazily sampling global variables that are needed in a plated
context. For example the following models are equivalent:

Example:

```python
def model_1(data):
    loc = pyro.sample("loc", dist.Normal(0, 1))
    with pyro.plate("data", len(data)):
        with block_plate("data"):
            scale = pyro.sample("scale", dist.LogNormal(0, 1))
            pyro.sample("x", dist.Normal(loc, scale))

def model_2(data):
    loc = pyro.sample("loc", dist.Normal(0, 1))
    scale = pyro.sample("scale", dist.LogNormal(0, 1))
    with pyro.plate("data", len(data)):
        pyro.sample("x", dist.Normal(loc, scale))
```

Parameters
   - name (str) – Optional name of plate to match.
   - dim (int) – Optional dim of plate to match. Must be negative.

Raises ValueError if no enclosing plate was found.

8.5.15 ReentrantMessenger

class ReentrantMessenger
   Bases: pyro.poutine.messenger.Messenger
8.5.16 ReparamMessenger

class ReparamHandler(msngr, fn)
    Bases: object

Reparameterization poutine.

class ReparamMessenger(config)
    Bases: pyro.poutine.messenger.Messenger

Reparametrizes each affected sample site into one or more auxiliary sample sites followed by a deterministic
transformation [1].

To specify reparameterizers, pass a config dict or callable to the constructor. See the pyro.infer.
reparam module for available reparameterizers.

Note some reparameterizers can examine the *args,**kwargs inputs of functions they affect; these repara-
parameterizers require using poutine.reparam as a decorator rather than as a context manager.


Parameters

config (dict or callable) – Configuration, either a dict mapping site name to
Reparameterizer, or a function mapping site to Reparameterizer or None.

8.5.17 ReplayMessenger

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

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:

```python
>>> 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:

```python
>>> old_trace = pyro.poutine.trace(model).get_trace(1.0)
>>> replayed_model = pyro.poutine.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`
8.5.18 ScaleMessenger

class ScaleMessenger(scale)

Bases: pyro.poutine.messenger.Messenger

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:

```python
>>> 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:

```python
>>> scaled_model = pyro.poutine.scale(model, scale=0.5)
>>> scaled_tr = pyro.poutine.trace(scaled_model).get_trace(0.0)
>>> unscaled_tr = pyro.poutine.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

8.5.19 SeedMessenger

class SeedMessenger(rng_seed)

Bases: pyro.poutine.messenger.Messenger

Handler to set the random number generator to a pre-defined state by setting its seed. This is the same as calling pyro.set_rng_seed() before the call to fn. This handler has no additional effect on primitive statements on the standard Pyro backend, but it might intercept pyro.sample calls in other backends, e.g. the NumPy backend.

Parameters

- `fn` – a stochastic function (callable containing Pyro primitive calls).
- `rng_seed` (int) – rng seed.

8.5.20 SubsampleMessenger

class SubsampleMessenger(name, size=None, subsample_size=None, subsample=None, dim=None, use_cuda=None, device=None)

Bases: pyro.poutine.indep_messenger.IndepMessenger

Extension of IndepMessenger that includes subsampling.
8.5.21 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.

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

    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 = 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]

    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

    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.
8.5.22 UnconditionMessenger

```python
class UnconditionMessenger
    Bases: pyro.poutine.messenger.Messenger
```

Messenger to force the value of observed nodes to be sampled from their distribution, ignoring observations.
The `pyro.ops` module implements tensor utilities that are mostly independent of the rest of Pyro.

### 9.1 Utilities for HMC

**class DualAveraging** ($\text{prox\_center}=0$, $t0=10$, $kappa=0.75$, $gamma=0.05$)

**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**


**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**

Updates states of the scheme given a new statistic/subgradient $g$.

**Parameters**

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

**step**($g$)

**get_state**

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

**velocity_verlet**($z, r, potential_fn, kinetic_grad, step_size, num_steps=1, z_grads=None$)

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 $r$.
- $kinetic_grad$ (*callable*) – a function calculating gradient of kinetic energy w.r.t. momentum variable.
- $step_size$ (*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. $z_{next}$.

**potential_grad**($potential\_fn, z$)

Gradient of $potential\_fn$ w.r.t. parameters $z$.

**Parameters**

- **potential_fn** – python callable that takes in a dictionary of parameters and returns the potential energy.
- $z$ (*dict*) – dictionary of parameter values keyed by site name.

**Returns** tuple of ($z_{grads}, potential\_energy$), where $z_{grads}$ is a dictionary with the same keys as $z$ containing gradients and potential\_energy is a torch scalar.

**class WelfordCovariance**(*diagonal=True*)

Bases: **object**

Implements Welford’s online scheme for estimating (co)variance (see [1]). Useful for adapting diagonal and dense mass structures for HMC.

**References**

[1] *The Art of Computer Programming*, Donald E. Knuth

**reset**
update(sample)

get_covariance(regularize=True)

class WelfordArrowheadCovariance(head_size=0)
    Bases: object
    Likes WelfordCovariance but generalized to the arrowhead structure.

reset()

update(sample)

get_covariance(regularize=True)
    Gets the covariance in arrowhead form: (top, bottom_diag) where top = cov[:head_size] and bottom_diag = cov.diag()[head_size:].

9.2 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 useful 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:

```python
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
```


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 estimate 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 estimate 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 estimate 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 estimate of the covariance 3x3 matrix with cov.shape == x. shape[:-1] + (3,3).

Return type  tuple

9.3 Special Functions

safe_log(x)
Like torch.log() but avoids infinite gradients at log(0) by clamping them to at most 1 / finfo.eps.

log_beta(x, y, tol=0.0)
Computes log Beta function.
When tol >= 0.02 this uses a shifted Stirling’s approximation to the log Beta function. The approximation adapts Stirling’s approximation of the log Gamma function:

\[ \log(\Gamma(z)) = (z - 1/2) \log(z) - z + \log(2 \pi)/2 \]

to approximate the log Beta function:

\[ \log_{beta}(x, y) = (x-1/2) \log(x) + (y-1/2) \log(y) - (x+y-1/2) \log(x+y) + \log(2\pi)/2 \]

The approximation additionally improves accuracy near zero by iteratively shifting the log Gamma approximation using the recursion:

\[ \log(\Gamma(x)) = \log(\Gamma(x + 1)) - \log(x) \]

If this recursion is applied \( n \) times, then absolute error is bounded by \( \text{error} < 0.082 / n < \text{tol} \), thus we choose \( n \) based on the user provided \( \text{tol} \).

Parameters
- x (torch.Tensor) – A positive tensor.
- y (torch.Tensor) – A positive tensor.
- tol (float) – Bound on maximum absolute error. Defaults to 0.1. For very small \( \text{tol} \), this function simply defers to \( \log_{beta}() \).

Return type  torch.Tensor

log_binomial(n, k, tol=0.0)
Computes log binomial coefficient.
When tol >= 0.02 this uses a shifted Stirling’s approximation to the log Beta function via \( \log_{beta}() \).

Parameters
- n (torch.Tensor) – A nonnegative integer tensor.
- k (torch.Tensor) – An integer tensor ranging in [0, n].

Return type  torch.Tensor
9.4 Tensor Utilities

**block_diag_embed** *(mat)*  
Takes a tensor of shape \((..., B, M, N)\) and returns a block diagonal tensor of shape \((..., B \times M, B \times N)\).

**Parameters**  
- `mat` *(torch.Tensor)* – an input tensor with 3 or more dimensions

**Returns** torch.Tensor  
a block diagonal tensor with dimension \(m.dim() - 1\)

**block_diagonal** *(mat, block_size)*  
Takes a block diagonal tensor of shape \((..., B \times M, B \times N)\) and returns a tensor of shape \((..., B, M, N)\).

**Parameters**  
- `mat` *(torch.Tensor)* – an input tensor with 2 or more dimensions
- `block_size` *(int)* – the number of blocks \(B\).

**Returns** torch.Tensor  
a tensor with dimension \(mat.dim() + 1\)

**periodic_repeat** *(tensor, size, dim)*  
Repeat a period-sized tensor up to given size. For example:

```python
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6]])
>>> periodic_repeat(x, 4, 0)
tensor([[1, 2, 3],
        [4, 5, 6],
        [1, 2, 3],
        [4, 5, 6]])
>>> periodic_repeat(x, 4, 1)
tensor([[1, 2, 3, 1],
        [4, 5, 6, 4]])
```

This is useful for computing static seasonality in time series models.

**Parameters**  
- `tensor` *(torch.Tensor)* – A tensor of differences.
- `size` *(int)* – Desired size of the result along dimension `dim`.
- `dim` *(int)* – The tensor dimension along which to repeat.

**periodic_cumsum** *(tensor, period, dim)*  
Compute periodic cumsum along a given dimension. For example if `dim`=0:

```python
for t in range(period):
    assert result[t] == tensor[t]
for t in range(period, len(tensor)):
    assert result[t] == tensor[t] + result[t - period]
```

This is useful for computing drifting seasonality in time series models.

**Parameters**  
- `tensor` *(torch.Tensor)* – A tensor of differences.
- `period` *(int)* – The period of repetition.
- `dim` *(int)* – The tensor dimension along which to accumulate.

**periodic_features** *(duration, max_period=None, min_period=None, **options)*  
Create periodic (sin,cos) features from `max_period` down to `min_period`.

---

182 Chapter 9. Miscellaneous Ops
This is useful in time series models where long uneven seasonality can be treated via regression. When only \texttt{max\_period} is specified this generates periodic features at all length scales. When also \texttt{min\_period} is specified this generates periodic features at large length scales, but omits high frequency features. This is useful when combining regression for long seasonality with other techniques like \texttt{periodic\_repeat()} and \texttt{periodic\_cumsum()} for short time scales. For example, to combine regress yearly seasonality down to the scale of one week one could set \texttt{max\_period=365.25} and \texttt{min\_period=7}.

Parameters

- \texttt{duration (int)} – Number of discrete time steps.
- \texttt{max\_period (float)} – Optional max period, defaults to \texttt{duration}.
- \texttt{min\_period (float)} – Optional min period (exclusive), defaults to $2 = \text{Nyquist cutoff}$.
- **\texttt{options}** – Tensor construction options, e.g. \texttt{dtype} and \texttt{device}.

Returns A \((\text{duration}, 2 \times \text{ceil} (\text{max\_period} / \text{min\_period}) - 2)\)-shaped tensor of features normalized to lie in \([-1, 1]\).

Return type \texttt{Tensor}

\texttt{next\_fast\_len (size)}

Returns the next largest number \(n \geq size\) whose prime factors are all 2, 3, or 5. These sizes are efficient for fast fourier transforms. Equivalent to \texttt{scipy.fftpack.next\_fast\_len()}.

Parameters \texttt{size (int)} – A positive number.

Returns A possibly larger number.

Rtype \texttt{int}

\texttt{convolve (signal, kernel, mode='full')}

Computes the 1-d convolution of signal by kernel using FFTs. The two arguments should have the same rightmost dim, but may otherwise be arbitrarily broadcastable.

Parameters

- \texttt{signal (torch.Tensor)} – A signal to convolve.
- \texttt{kernel (torch.Tensor)} – A convolution kernel.
- \texttt{mode (str)} – One of: ‘full’, ‘valid’, ‘same’.

Returns A tensor with broadcasted shape. Letting \(m = \text{signal.size(-1)}\) and \(n = \text{kernel.size(-1)}\), the rightmost size of the result will be: \(m + n - 1\) if mode is ‘full’; \(\max (m, n) - \min (m, n) + 1\) if mode is ‘valid’; or \(\max (m, n)\) if mode is ‘same’.

Rtype \texttt{torch.Tensor}

\texttt{repeated\_matmul (M, n)}

Takes a batch of matrices \(M\) as input and returns the stacked result of doing the \(n\)-many matrix multiplications \(M, M^2, \ldots, M^n\). Parallel cost is logarithmic in \(n\).

Parameters

- \texttt{M (torch.Tensor)} – A batch of square tensors of shape \((\ldots, N, N)\).
- \texttt{n (int)} – The order of the largest product \(M^n\)

Returns \texttt{torch.Tensor} A batch of square tensors of shape \((n, \ldots, N, N)\)

\texttt{dct (x, dim=-1)}

Discrete cosine transform of type II, scaled to be orthonormal.

This is the inverse of \texttt{idct\_ii()}, and is equivalent to \texttt{scipy.fftpack.dct()} with \texttt{norm="ortho"}.
Parameters

- **x** *(Tensor)* – The input signal.
- **dim** *(int)* – Dimension along which to compute DCT.

Return type: Tensor

**dct** *(x, dim=-1)*

Inverse discrete cosine transform of type II, scaled to be orthonormal.

This is the inverse of **dct_ii()**, and is equivalent to **scipy.fftpack.idct**() with norm="ortho".

Parameters

- **x** *(Tensor)* – The input signal.
- **dim** *(int)* – Dimension along which to compute DCT.

Return type: Tensor

**idct** *(x, dim=-1)*

Inverse discrete cosine transform of type II, scaled to be orthonormal.

This is the inverse of **dct_ii()**, and is equivalent to **scipy.fftpack.idct**() with norm="ortho".

Parameters

- **x** *(Tensor)* – The input signal.
- **dim** *(int)* – Dimension along which to compute DCT.

Return type: Tensor

**haar_transform** *(x)*

Discrete Haar transform.

Performs a Haar transform along the final dimension. This is the inverse of **inverse_haar_transform()**.

Parameters: **x** *(Tensor)* – The input signal.

Return type: Tensor

**inverse_haar_transform** *(x)*

Performs an inverse Haar transform along the final dimension. This is the inverse of **haar_transform()**.

Parameters: **x** *(Tensor)* – The input signal.

Return type: Tensor

**cholesky** *(x)*

**cholesky_solve** *(x, y)*

**matmul** *(x, y)*

**matvecmul** *(x, y)*

**triangular_solve** *(x, y, upper=False, transpose=False)*

**precision_to_scale_tril** *(P)*

### 9.5 Tensor Indexing

**index** *(tensor, args)*

Indexing with nested tuples.

See also the convenience wrapper **Index**.

This is useful for writing indexing code that is compatible with multiple interpretations, e.g. scalar evaluation, vectorized evaluation, or reshaping.

For example suppose **x** is a parameter with **x.dim() == 2** and we wish to generalize the expression **x[...], t]** where **t** can be any of:

- a scalar **t=1** as in **x[...], 1]**;
- a slice **t=slice(None)** equivalent to **x[...], :]**; or
• a reshaping operation \( t=(\text{Ellipsis}, \text{None}) \) equivalent to \( x\.\text{unsqueeze}(-1) \).

While naive indexing would work for the first two, the third example would result in a nested tuple \( (\text{Ellipsis}, (\text{Ellipsis}, \text{None})) \). This helper flattens that nested tuple and combines consecutive \text{Ellipsis}.

**Parameters**

- **tensor** *(torch.Tensor)* – A tensor to be indexed.
- **args** *(tuple)* – An index, as args to \texttt{__getitem__}.

**Returns** A flattened interpretation of \( tensor[args] \).

**Return type** torch.Tensor

```python
class Index(tensor):
   _bases: object

    Convenience wrapper around \texttt{index()}.

    The following are equivalent:

    ```python
    Index(x)[..., i, j, :]
    index(x, (Ellipsis, i, j, slice(None)))
    ```

    **Parameters** **tensor** *(torch.Tensor)* – A tensor to be indexed.

    **Returns** An object with a special \texttt{__getitem__} method.
```

```python
def vindex(tensor, args):
    Vectorized advanced indexing with broadcasting semantics.

    See also the convenience wrapper \texttt{Vindex}.

    This is useful for writing indexing code that is compatible with batching and enumeration, especially for selecting mixture components with discrete random variables.

    For example suppose \( x \) is a parameter with \( x\.\text{dim}() == 3 \) and we wish to generalize the expression \( x[i, :, j] \) from integer \( i,j \) to tensors \( i,j \) with batch dims and enum dims (but no event dims). Then we can write the generalize version using \texttt{Vindex}

    ```python
    xij = Vindex(x)[i, :, j]
    batch_shape = broadcast_shape(i.shape, j.shape)
    event_shape = (x.size(1),)
    assert xij.shape == batch_shape + event_shape
    ```

    To handle the case when \( x \) may also contain batch dimensions (e.g. if \( x \) was sampled in a plated context as when using vectorized particles), \texttt{vindex()} uses the special convention that \texttt{Ellipsis} denotes batch dimensions (hence \( ... \) can appear only on the left, never in the middle or in the right). Suppose \( x \) has event dim 3. Then we can write:

    ```python
    old_batch_shape = x.shape[:-3]
    old_event_shape = x.shape[-3:]
    xij = Vindex(x)[..., i, :, j]  # The ... denotes unknown batch shape.
    new_batch_shape = broadcast_shape(old_batch_shape, i.shape, j.shape)
    new_event_shape = (x.size(1),)
    assert xij.shape = new_batch_shape + new_event_shape
    ```
```

9.5. Tensor Indexing 185
Note that this special handling of `Ellipsis` differs from the NEP [1].

Formally, this function assumes:

1. Each arg is either `Ellipsis`, `slice(None)`, an integer, or a batched `torch.LongTensor` (i.e. with empty event shape). This function does not support Nontrivial slices or `torch.BoolTensor` masks. `Ellipsis` can only appear on the left as `args[0]`.

2. If `args[0]` is not `Ellipsis` then `tensor` is not batched, and its event dim is equal to `len(args)`.

3. If `args[0]` is `Ellipsis` then `tensor` is batched and its event dim is equal to `len(args[1:])`. Dims of `tensor` to the left of the event dims are considered batch dims and will be broadcasted with dims of tensor args.

Note that if none of the args is a tensor with `.dim() > 0`, then this function behaves like standard indexing:

```python
if not any(isinstance(a, torch.Tensor) and a.dim() for a in args):
    assert Vindex(x)[args] == x[args]
```

References

[1] https://www.numpy.org/neps/nep-0021-advanced-indexing.html introduces `vindex` as a helper for vectorized indexing. The Pyro implementation is similar to the proposed notation `x.vindex[]` except for slightly different handling of `Ellipsis`.

Parameters

- `tensor` (`torch.Tensor`) – A tensor to be indexed.
- `args` (`tuple`) – An index, as args to `__getitem__`.

Returns

A nonstandard interpretation of `tensor[args]`.

Return type `torch.Tensor`

class Vindex

Bases: object

Convenience wrapper around `vindex()`.

The following are equivalent:

```python
Vindex(x)[..., i, j, :]
vindex(x, (Ellipsis, i, j, slice(None))]
```

Parameters

- `tensor` (`torch.Tensor`) – A tensor to be indexed.

Returns

An object with a special `__getitem__()` method.

## 9.6 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.

**einsum** *(equation, *operands, **kwargs)*

Generalized plated sum-product algorithm via tensor variable elimination.

This generalizes `contract()` in two ways:

1. Multiple outputs are allowed, and intermediate results can be shared.
2. Inputs and outputs can be plated along symbols given in `plates`: reductions along `plates` are product reductions.

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

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

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

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

```python
assert len(x) == 3 and len(y) == 3
z = einsum('ab,ai,bi->b', w, x, y, plates='i')
z = contract('ab,a,a,b,b->b', w, *x, *y)
```

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

```python
assert len(x) == 3 and len(y) == 3
z = einsum('ai,ai->', x, y, plates='i')
z = contract('a,b,c,a,b,c->', *x, *y)
```

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

```python
assert len(x) == 3 and len(y) == 3
z0 = contract('ab,ac,ad,ab,ac,ad->b', *x, *y)
z1 = contract('ab,ac,ad,ab,ac,ad->c', *x, *y)
z2 = contract('ab,ac,ad,ab,ac,ad->d', *x, *y)
z = torch.stack([z0, z1, z2])
```

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

```python
xy = einsum('abc,acd->abd', x, y, plates='')
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.
- `plates (str)` – An optional string of plate symbols.
- `backend (str)` – An optional einsum backend, defaults to ‘torch’.
- `cache (dict)` – An optional `shared_intermediates()` cache.
- `modulo_total (bool)` – Optionally allow einsum to arbitrarily scale each result plate, which can significantly reduce computation. This is safe to set whenever each result plate denotes a nonnormalized probability distribution whose total is not of interest.

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

**Return type** tuple

**Raises**

- `ValueError` – if tensor sizes mismatch or an output requests a plated dim without that dim’s plates.
- `NotImplementedError` – if contraction would have cost exponential in the size of any input tensor.

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

Deprecated, use `einsum()` instead.

### 9.7 Gaussian Contraction

**class Gaussian (log_normalizer, info_vec, precision)**

Bases: object

Non-normalized Gaussian distribution.

This represents an arbitrary semidefinite quadratic function, which can be interpreted as a rank-deficient scaled Gaussian distribution. The precision matrix may have zero eigenvalues, thus it may be impossible to work directly with the covariance matrix.

**Parameters**

- `log_normalizer (torch.Tensor)` – a normalization constant, which is mainly used to keep track of normalization terms during contractions.
- `info_vec (torch.Tensor)` – information vector, which is a scaled version of the mean `info_vec = precision @ mean`. We use this representation to make gaussian contraction fast and stable.
- `precision (torch.Tensor)` – precision matrix of this gaussian.

```python
dim()
batch_shape
expand(batch_shape)
reshape(batch_shape)
```
__getitem__(index)
Index into the batch_shape of a Gaussian.

static cat(parts, dim=0)
Concatenate a list of Gaussians along a given batch dimension.

event_pad(left=0, right=0)
Pad along event dimension.

event_permute(perm)
Permute along event dimension.

__add__(other)
Adds two Gaussians in log-density space.

log_density(value)
Evaluate the log density of this Gaussian at a point value:

\[-0.5 \times \text{value}.T @ \text{precision} @ \text{value} + \text{value}.T @ \text{info_vec} + \log_{\text{normalizer}}\]

This is mainly used for testing.

rsample(sample_shape=torch.Size([]))
Reparameterized sampler.

condition(value)
Condition this Gaussian on a trailing subset of its state. This should satisfy:

\[g.\text{condition}(y).\text{dim}() = g.\text{dim}() - y.\text{size}(-1)\]

Note that since this is a non-normalized Gaussian, we include the density of \(y\) in the result. Thus \condition() is similar to a functools.partial binding of arguments:

\[
\begin{align*}
\text{left} &= x[\ldots, :n] \\
\text{right} &= x[\ldots, n:] \\
g.\text{log_density}(x) &= g.\text{condition}(\text{right}).\text{log_density}(\text{left})
\end{align*}
\]

left_condition(value)
Condition this Gaussian on a leading subset of its state. This should satisfy:

\[g.\text{condition}(y).\text{dim}() = g.\text{dim}() - y.\text{size}(-1)\]

Note that since this is a non-normalized Gaussian, we include the density of \(y\) in the result. Thus \condition() is similar to a functools.partial binding of arguments:

\[
\begin{align*}
\text{left} &= x[\ldots, :n] \\
\text{right} &= x[\ldots, n:] \\
g.\text{log_density}(x) &= g.\text{left_condition}(\text{left}).\text{log_density}(\text{right})
\end{align*}
\]

marginalize(left=0, right=0)
Marginalizing out variables on either side of the event dimension:

\[
\begin{align*}
g.\text{marginalize}(\text{left}=n).\text{event_logsumexp}() &= g.\text{logsumexp}() \\
g.\text{marginalize}(\text{right}=n).\text{event_logsumexp}() &= g.\text{logsumexp}()
\end{align*}
\]

and for data \(x\):

\[g.\text{condition}(x).\text{event_logsumexp}() = g.\text{marginalize}(\text{left}=g.\text{dim}() - x.\text{size}(-1)).\text{log_density}(x)\]

event_logsumexp()
Integrates out all latent state (i.e. operating on event dimensions).
class AffineNormal(matrix, loc, scale)

Bases: object

Represents a conditional diagonal normal distribution over a random variable $Y$ whose mean is an affine function of a random variable $X$. The likelihood of $X$ is thus:

$$\text{AffineNormal}(\text{matrix}, \text{loc}, \text{scale}).\text{condition}(y).\log_{\text{density}}(x)$$

which is equivalent to:

$$\text{Normal}(x \odot \text{matrix} + \text{loc}, \text{scale}).\text{to_event}(1).\log_{\text{prob}}(y)$$

Parameters

- **matrix** *(torch.Tensor)* – A transformation from $X$ to $Y$. Should have rightmost shape $(x_{\text{dim}}, y_{\text{dim}})$.
- **loc** *(torch.Tensor)* – A constant offset for $Y$’s mean. Should have rightmost shape $(y_{\text{dim}},)$.
- **scale** *(torch.Tensor)* – Standard deviation for $Y$. Should have rightmost shape $(y_{\text{dim}},)$.

batch_shape

condition(value)

left_condition(value)

If value.size(-1) == $x_{\text{dim}}$, this returns a Normal distribution with event_dim=1. After applying this method, the cost to draw a sample is $O(y_{\text{dim}})$ instead of $O(y_{\text{dim}} \times 3)$.

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

Reparameterized sampler.

to_gaussian()

expand(batch_shape)

reshape(batch_shape)

__getitem__(index)

event_permute(perm)

__add__(other)

marginalize(left=0, right=0)

mvn_to_gaussian(mvn)

Convert a MultivariateNormal distribution to a Gaussian.

Parameters

- **mvn** *(MultivariateNormal)* – A multivariate normal distribution.

Returns

An equivalent Gaussian object.

Return type

Gaussian

matrix_and_mvn_to_gaussian(matrix, mvn)

Convert a noisy affine function to a Gaussian. The noisy affine function is defined as:

$$y = x \odot \text{matrix} + \text{mvn}.\text{sample}()$$

Parameters
• **matrix** (*Tensor*) – A matrix with rightmost shape \((x_{\text{dim}}, y_{\text{dim}})\).

• **mvn** (*MultivariateNormal*) – A multivariate normal distribution.

Returns A Gaussian with broadcasted batch shape and \(\text{.dim()} == x_{\text{dim}} + y_{\text{dim}}\).

Return type *Gaussian*

### gaussian_tensordot \((x, y, \text{dims}=0)\)

Computes the integral over two gaussians:

\[
(x \@ y)(a,c) = \log(\text{integral}(\exp(x(a,b) + y(b,c)), b)),
\]

where \(x\) is a gaussian over variables \((a,b)\), \(y\) is a gaussian over variables \((b,c)\), \((a,b,c)\) can each be sets of zero or more variables, and \(\text{dims}\) is the size of \(b\).

Parameters

- **x** – a Gaussian instance
- **y** – a Gaussian instance
- **dims** – number of variables to contract

### 9.8 Statistical Utilities

#### gelman_rubin \((\text{input, chain_dim}=0, \text{sample_dim}=1)\)

Computes R-hat over chains of samples. It is required that \(\text{input.size(sample_dim)} >= 2\) and \(\text{input.size(chain_dim)} >= 2\).

Parameters

- **input** (*torch.Tensor*) – the input tensor.
- **chain_dim** (*int*) – the chain dimension.
- **sample_dim** (*int*) – the sample dimension.

Returns *torch.Tensor* R-hat of input.

#### split_gelman_rubin \((\text{input, chain_dim}=0, \text{sample_dim}=1)\)

Computes R-hat over chains of samples. It is required that \(\text{input.size(sample_dim)} >= 4\).

Parameters

- **input** (*torch.Tensor*) – the input tensor.
- **chain_dim** (*int*) – the chain dimension.
- **sample_dim** (*int*) – the sample dimension.

Returns *torch.Tensor* split R-hat of input.

#### autocorrelation \((\text{input, dim}=0)\)

Computes the autocorrelation of samples at dimension \(\text{dim}\).

Reference: [https://en.wikipedia.org/wiki/Autocorrelation#Efficient_computation](https://en.wikipedia.org/wiki/Autocorrelation#Efficient_computation)

Parameters

- **input** (*torch.Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to calculate autocorrelation.

Returns *torch.Tensor* autocorrelation of input.
\textbf{autocovariance} \((input, \text{dim}=0)\)

Computes the autocovariance of samples at dimension \(\text{dim}\).

\textbf{Parameters}

\begin{itemize}
    \item \textbf{input} (\texttt{torch.Tensor}) – the input tensor.
    \item \textbf{dim} (\texttt{int}) – the dimension to calculate autocorrelation.
\end{itemize}

\textbf{Returns} \texttt{torch.Tensor} autocorrelation of \(input\).

\textbf{effective_sample_size} \((input, \text{chain_dim}=0, \text{sample_dim}=1)\)

Computes effective sample size of \(input\).

\textbf{Reference:}

[1] \textit{Introduction to Markov Chain Monte Carlo}, Charles J. Geyer


\textbf{Parameters}

\begin{itemize}
    \item \textbf{input} (\texttt{torch.Tensor}) – the input tensor.
    \item \textbf{chain_dim} (\texttt{int}) – the chain dimension.
    \item \textbf{sample_dim} (\texttt{int}) – the sample dimension.
\end{itemize}

\textbf{Returns} \texttt{torch.Tensor} effective sample size of \(input\).

\textbf{resample} \((input, \text{num_samples}, \text{dim}=0, \text{replacement}=False)\)

Draws \text{num_samples} samples from \(input\) at dimension \(\text{dim}\).

\textbf{Parameters}

\begin{itemize}
    \item \textbf{input} (\texttt{torch.Tensor}) – the input tensor.
    \item \textbf{num_samples} (\texttt{int}) – the number of samples to draw from \(input\).
    \item \textbf{dim} (\texttt{int}) – dimension to draw from \(input\).
\end{itemize}

\textbf{Returns} \texttt{torch.Tensor} samples drawn randomly from \(input\).

\textbf{quantile} \((input, \text{probs}, \text{dim}=0)\)

Computes quantiles of \(input\) at \text{probs}. If \text{probs} is a scalar, the output will be squeezed at \text{dim}.

\textbf{Parameters}

\begin{itemize}
    \item \textbf{input} (\texttt{torch.Tensor}) – the input tensor.
    \item \textbf{probs} (\texttt{list}) – quantile positions.
    \item \textbf{dim} (\texttt{int}) – dimension to take quantiles from \(input\).
\end{itemize}

\textbf{Returns} \texttt{torch.Tensor} quantiles of \(input\) at \text{probs}.

\textbf{pi} \((input, \text{prob}, \text{dim}=0)\)

Computes percentile interval which assigns equal probability mass to each tail of the interval.

\textbf{Parameters}

\begin{itemize}
    \item \textbf{input} (\texttt{torch.Tensor}) – the input tensor.
    \item \textbf{prob} (\texttt{float}) – the probability mass of samples within the interval.
    \item \textbf{dim} (\texttt{int}) – dimension to calculate percentile interval from \(input\).
\end{itemize}

\textbf{Returns} \texttt{torch.Tensor} quantiles of \(input\) at \text{probs}.
hpdi(input, prob, dim=0)

Computes “highest posterior density interval” which is the narrowest interval with probability mass prob.

Parameters

- input(torch.Tensor) – the input tensor.
- prob(float) – the probability mass of samples within the interval.
- dim(int) – dimension to calculate percentile interval from input.

Returns torch.Tensor quantities of input at probs.

waic(input, log_weights=None, pointwise=False, dim=0)

Computes “Widely Applicable/Watanabe-Akaike Information Criterion” (WAIC) and its corresponding effective number of parameters.

Reference:
[1] WAIC and cross-validation in Stan, Aki Vehtari, Andrew Gelman

Parameters

- input(torch.Tensor) – the input tensor, which is log likelihood of a model.
- log_weights(torch.Tensor) – weights of samples along dim.
- dim(int) – the sample dimension of input.

Returns tuple tuple of WAIC and effective number of parameters.

fit_generalized_pareto(X)

Given a dataset X assumed to be drawn from the Generalized Pareto Distribution, estimate the distributional parameters k, sigma using a variant of the technique described in reference [1], as described in reference [2].


Parameters torch.Tensor – the input data X

Returns tuple tuple of floats (k, sigma) corresponding to the fit parameters

crps_empirical(pred, truth)

Computes negative Continuous Ranked Probability Score CRPS* [1] between a set of samples pred and true data truth. This uses an \( n \log(n) \) time algorithm to compute a quantity equal that would naively have complexity quadratic in the number of samples \( n \):

\[
\text{CRPS}^* = \mathbb{E}|\text{pred} - \text{truth}| - \frac{1}{2} \mathbb{E}|\text{pred} - \text{pred}'| \\
= (\text{pred} - \text{truth}).abs().mean(0) \\
- (\text{pred} - \text{pred}.unsqueeze(1)).abs().mean([0, 1]) / 2
\]

Note that for a single sample this reduces to absolute error.

References


Parameters

- pred(torch.Tensor) – A set of sample predictions batched on rightmost dim. This should have shape (num_samples,) + truth.shape.
- truth(torch.Tensor) – A tensor of true observations.
Returns A tensor of shape `truth.shape`.
Return type `torch.Tensor`

### 9.9 State Space Model and GP Utilities

class MaternKernel `(nu=1.5, num_gps=1, length_scale_init=None, kernel_scale_init=None)`
Bases: `pyro.nn.module.PyroModule`

Provides the building blocks for representing univariate Gaussian Processes (GPs) with Matern kernels as state space models.

**Parameters**

- `nu` (*float*) – The order of the Matern kernel (one of 0.5, 1.5 or 2.5)
- `num_gps` (*int*) – the number of GPs
- `length_scale_init` (*torch.Tensor*) – optional `num_gps`-dimensional vector of initializers for the length scale
- `kernel_scale_init` (*torch.Tensor*) – optional `num_gps`-dimensional vector of initializers for the kernel scale

**References**


**transition_matrix** *(dt)*

Compute the (exponentiated) transition matrix of the GP latent space. The resulting matrix has layout `(num_gps, old_state, new_state)`, i.e. this matrix multiplies states from the right.


**Parameters**

- `dt` (*float*) – the time interval over which the GP latent space evolves.

**Returns** `torch.Tensor` a 3-dimensional tensor of transition matrices of shape `(num_gps, state_dim, state_dim)`.

**stationary_covariance** *

Compute the stationary state covariance. See Eqn. 3.26 in reference [2].

**Returns** `torch.Tensor` a 3-dimensional tensor of covariance matrices of shape `(num_gps, state_dim, state_dim)`.

**process_covariance** *(A)*

Given a transition matrix `A` computed with `transition_matrix` compute the the process covariance as described in Eqn. 3.11 in reference [2].

**Returns** `torch.Tensor` a batched covariance matrix of shape `(num_gps, state_dim, state_dim)`

**transition_matrix_and_covariance** *(dt)*

Get the transition matrix and process covariance corresponding to a time interval `dt`.

**Parameters**

- `dt` (*float*) – the time interval over which the GP latent space evolves.

**Returns** `tuple` *(transition_matrix, process_covariance)* both 3-dimensional tensors of shape `(num_gps, state_dim, state_dim)`
CHAPTER 10

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`

The `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:

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

Example:

```python
>>> 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, \texttt{scope} will use the name of the input function as the prefix if no user-specified prefix is provided.

Example:

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

\texttt{name\_count} \texttt{(fn=}\texttt{None})

\texttt{name\_count} is a very simple autonaming scheme that simply appends a suffix “\_\_” plus a counter to any name that appears multiple times in an execution. Only duplicate instances of a name get a suffix; the first instance is not modified.

Example:

```python
>>> @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()
```

\texttt{name\_count} also composes with \texttt{scope()} by adding a suffix to duplicate scope entrances:

Example:

```python
>>> @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:

```python
>>> @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()

10.1 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 (str) – name of parameter
• init_tensor (torch.Tensor or callable) – initial tensor or lazy callable that returns a tensor. For large tensors, it may be cheaper to write e.g. lambda: torch.randn(100000), which will only be evaluated on the initial statement.
• constraint (torch.distributions.constraints.Constraint) – torch constraint, defaults to constraints.real.
• event_dim (int) – (optional) number of rightmost dimensions unrelated to batching. Dimension to the left of this will be considered batch dimensions; if the param statement is inside a subsampled plate, then corresponding batch dimensions of the parameter will be correspondingly subsampled. If unspecified, all dimensions will be considered event dims and no subsampling will be performed.

Returns parameter

Return type torch.Tensor
```

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.

```python
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.

### 10.2 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:

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

Example:

```python
>>> 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:

```python
>>> @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:

```python
>>> @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 times in an execution. Only duplicate instances of a name get a suffix; the first instance is not modified.

Example:

```python
>>> @name_count
... def model():
...     for i in range(3):
...         pyro.sample("x", dist.Bernoulli(0.5))
...
(continues on next page)```
name_count also composes with scope() by adding a suffix to duplicate scope entrances:

Example:

```python
>>> @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()
```
CHAPTER 11

Bayesian Neural Networks

11.1 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 distribution.

In effect, this distribution encodes the following generative process:

$A \sim \text{Normal}(A_{\text{mean}}, A_{\text{scale}})$ output $\sim \text{non-linearity}(AX)$

Parameters

- $X$ (torch.Tensor) – $B \times D$ dimensional mini-batch of inputs
- $A_{\text{mean}}$ (torch.Tensor) – $D \times H$ dimensional specifying weight mean
- $A_{\text{scale}}$ (torch.Tensor) – $D \times H$ 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.
- $A_{\text{prior}}$ (float or torch.Tensor) – the prior over the weights is assumed to be normal with mean zero and scale factor $A_{\text{prior}}$. 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:

This module implements the Causal Effect Variational Autoencoder [1], which demonstrates a number of innovations including:

- a generative model for causal effect inference with hidden confounders;
- a model and guide with twin neural nets to allow imbalanced treatment; and
- a custom training loss that includes both ELBO terms and extra terms needed to train the guide to be able to answer counterfactual queries.

The main interface is the `CEVAE` class, but users may customize by using components `Model`, `Guide`, `TraceCausalEffect_ELBO` and utilities.

References

https://github.com/AMLab-Amsterdam/CEVAE

12.1 CEVAE Class

class CEVAE (feature_dim, outcome_dist='bernoulli', latent_dim=20, hidden_dim=200, num_layers=3, num_samples=100)
Bases: torch.nn.modules.module.Module

Main class implementing a Causal Effect VAE [1]. This assumes a graphical model
where \( t \) is a binary treatment variable, \( y \) is an outcome, \( Z \) is an unobserved confounder, and \( X \) is a noisy function of the hidden confounder \( Z \).

Example:

```python
ccevae = CEVAE(feature_dim=5)
cevae.fit(x_train, t_train, y_train)
ite = cevae.ite(x_test)  # individual treatment effect
ate = ite.mean()  # average treatment effect
```

**Variables**

- **model (Model)** – Generative model.
- **guide (Guide)** – Inference model.

**Parameters**

- **feature_dim (int)** – Dimension of the feature space \( x \).
- **latent_dim (int)** – Dimension of the latent variable \( z \). Defaults to 20.
- **num_layers (int)** – Number of hidden layers in fully connected networks.
- **num_samples (int)** – Default number of samples for the `ite()` method. Defaults to 100.

```
fit(x, t, y, num_epochs=100, batch_size=100, learning_rate=0.001, learning_rate_decay=0.1, weight_decay=0.0001)
```

Train using SVI with the `TraceCausalEffect_ELBO` loss.
Parameters

- `x (Tensor)` –
- `t (Tensor)` –
- `y (Tensor)` –
- `num_epochs (int)` – Number of training epochs. Defaults to 100.
- `batch_size (int)` – Batch size. Defaults to 100.
- `learning_rate (float)` – Learning rate. Defaults to 1e-3.
- `learning_rate_decay (float)` – Learning rate decay over all epochs; the per-step decay rate will depend on batch size and number of epochs such that the initial learning rate will be `learning_rate` and the final learning rate will be `learning_rate * learning_rate_decay`. Defaults to 0.1.
- `weight_decay (float)` – Weight decay. Defaults to 1e-4.

Returns list of epoch losses

```python
ite(x, num_samples=None, batch_size=None)
```

Computes Individual Treatment Effect for a batch of data `x`.

\[
ITE(x) = \mathbb{E}[y \mid X = x, do(t = 1)] - \mathbb{E}[y \mid X = x, do(t = 0)]
\]

This has complexity \(O(len(x) \times num_samples ** 2)\).

Parameters

- `x (Tensor)` – A batch of data.
- `num_samples (int)` – The number of monte carlo samples. Defaults to self. `num_samples` which defaults to 100.
- `batch_size (int)` – Batch size. Defaults to `len(x)`.

Returns A `len(x)`-sized tensor of estimated effects.

Return type `Tensor`

```python
to_script_module()
```

Compile this module using `torch.jit.trace_module()`, assuming self has already been fit to data.

Returns A traced version of self with an `ite()` method.

Return type `torch.jit.ScriptModule`

### 12.2 CEVAE Components

class Model (config)

Bases: `pyro.nn.module.PyroModule`

Generative model for a causal model with latent confounder `z` and binary treatment `t`:

```r
z ~ p(z) # latent confounder
x ~ p(x|z) # partial noisy observation of z
t ~ p(t|z) # treatment, whose application is biased by z
y ~ p(y|t,z) # outcome
```
Each of these distributions is defined by a neural network. The \( y \) distribution is defined by a disjoint pair of neural networks defining \( p(y|t=0,z) \) and \( p(y|t=1,z) \); this allows highly imbalanced treatment.

**Parameters**

- **config** (*dict*) — A dict specifying `feature_dim`, `latent_dim`, `hidden_dim`, `num_layers`, and `outcome_dist`.

**forward** (*x*, *t=None*, *y=None*, *size=None*)

- \( y_{\text{mean}} \) (*x*, *t=None*)
- \( z_{\text{dist}} () \)
- \( x_{\text{dist}} (z) \)
- \( y_{\text{dist}} (t, z) \)
- \( t_{\text{dist}} (z) \)

**class Guide (config)**

**Bases:** `pyro.nn.module.PyroModule`

Inference model for causal effect estimation with latent confounder \( z \) and binary treatment \( t \):

\[
\begin{align*}
    t &\sim q(t|x) & \# \text{treatment} \\
    y &\sim q(y|t,x) & \# \text{outcome} \\
    z &\sim q(z|y,t,x) & \# \text{latent confounder, an embedding}
\end{align*}
\]

Each of these distributions is defined by a neural network. The \( y \) and \( z \) distributions are defined by disjoint pairs of neural networks defining \( p(-|t=0,\ldots) \) and \( p(-|t=1,\ldots) \); this allows highly imbalanced treatment.

**Parameters**

- **config** (*dict*) — A dict specifying `feature_dim`, `latent_dim`, `hidden_dim`, `num_layers`, and `outcome_dist`.

**forward** (*x*, *t=None*, *y=None*, *size=None*)

- \( t_{\text{dist}} (x) \)
- \( y_{\text{dist}} (t, x) \)
- \( z_{\text{dist}} (y, t, x) \)

**class TraceCausalEffect_ELBO** (*num_particles=1*, *

**Bases:** `pyro.infer.trace_elbo.Trace_ELBO`

Loss function for training a CVAE. From [1], the CVAE objective (to maximize) is:

\[-\text{loss} = \text{ELBO} + \log q(t|x) + \log q(y|t,x)\]

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

12.3 Utilities

**class FullyConnected** (*sizes, final_activation=None*)

**Bases:** `torch.nn.modules.container.Sequential`

Fully connected multi-layer network with ELU activations.

**append** (*layer*)
class DistributionNet
   Bases: torch.nn.modules.module.Module

   Base class for distribution nets.

   static get_class(dtype)
      Get a subclass by a prefix of its name, e.g.:

      assert DistributionNet.get_class("bernoulli") is BernoulliNet

class BernoulliNet(sizes)
   Bases: pyro.contrib.cevae.DistributionNet

   FullyConnected network outputting a single logits value.

   This is used to represent a conditional probability distribution of a single Bernoulli random variable conditioned on a sizes[0]-sized real value, for example:

   ```python
   net = BernoulliNet([3, 4])
   z = torch.randn(3)
   logits, = net(z)
   t = net.make_dist(logits).sample()
   ```

   forward(x)

   static make_dist(logits)

class ExponentialNet(sizes)
   Bases: pyro.contrib.cevae.DistributionNet

   FullyConnected network outputting a constrained rate.

   This is used to represent a conditional probability distribution of a single Normal random variable conditioned on a sizes[0]-size real value, for example:

   ```python
   net = ExponentialNet([3, 4])
   x = torch.randn(3)
   rate, = net(x)
   y = net.make_dist(rate).sample()
   ```

   forward(x)

   static make_dist(rate)

class LaplaceNet(sizes)
   Bases: pyro.contrib.cevae.DistributionNet

   FullyConnected network outputting a constrained loc, scale pair.

   This is used to represent a conditional probability distribution of a single Laplace random variable conditioned on a sizes[0]-size real value, for example:

   ```python
   net = LaplaceNet([3, 4])
   x = torch.randn(3)
   loc, scale = net(x)
   y = net.make_dist(loc, scale).sample()
   ```

   forward(x)

   static make_dist(loc, scale)

class NormalNet(sizes)
   Bases: pyro.contrib.cevae.DistributionNet
**FullyConnected** network outputting a constrained loc, scale pair.

This is used to represent a conditional probability distribution of a single Normal random variable conditioned on a sizes[0]-size real value, for example:

```python
net = NormalNet([3, 4])
x = torch.randn(3)
loc, scale = net(x)
y = net.make_dist(loc, scale).sample()
```

**forward** (x)

**static make_dist** (loc, scale)

**class StudentTNet** (sizes)
**Bases:** pyro.contrib.cevae.DistributionNet

**FullyConnected** network outputting a constrained df, loc, scale triple, with shared df > 1.

This is used to represent a conditional probability distribution of a single Student’s t random variable conditioned on a sizes[0]-size real value, for example:

```python
net = StudentTNet([3, 4])
x = torch.randn(3)
df, loc, scale = net(x)
y = net.make_dist(df, loc, scale).sample()
```

**forward** (x)

**static make_dist** (df, loc, scale)

**class DiagNormalNet** (sizes)
**Bases:** torch.nn.modules.module.Module

**FullyConnected** network outputting a constrained loc, scale pair.

This is used to represent a conditional probability distribution of a sizes[-1]-sized diagonal Normal random variable conditioned on a sizes[0]-size real value, for example:

```python
net = DiagNormalNet([3, 4, 5])
z = torch.randn(3)
loc, scale = net(z)
x = dist.Normal(loc, scale).sample()
```

This is intended for the latent z distribution and the prewhitened x features, and conservatively clips loc and scale values.

**forward** (x)
13.1 EasyGuide

class EasyGuide(model):
    Bases: `pyro.nn.module.PyroModule`

    Base class for “easy guides”, which are more flexible than AutoGuides, but are easier to write than raw Pyro guides.

    Derived classes should define a `guide()` method. This `guide()` method can combine ordinary guide statements (e.g. `pyro.sample` and `pyro.param`) with the following special statements:

    • `group = self.group(...)` selects multiple `pyro.sample` sites in the model. See `Group` for subsequent methods.
    • `with self.plate(...): ...` should be used instead of `pyro.plate`.
    • `self.map_estimate(...)` uses a Delta guide for a single site.

    Derived classes may also override the `init()` method to provide custom initialization for models sites.

    Parameters
    
    `model` (callable) – A Pyro model.

    `guide(*args, **kargs)`
        Guide implementation, to be overridden by user.

    `init(site)`
        Model initialization method, may be overridden by user.

        This should input a site and output a valid sample from that site. The default behavior is to draw a random sample:

        ```python
        return site["fn"]()
        ```

        For other possible initialization functions see http://docs.pyro.ai/en/stable/infer.autoguide.html#module-pyro.infer.autoguide.initialization
forward(*args, **kwargs)
    Runs the guide. This is typically used by inference algorithms.

plate(name, size=None, subsample_size=None, subsample=None, *args, **kwargs)
    A wrapper around pyro.plate to allow EasyGuide to automatically construct plates. You should use this rather than pyro.plate inside your guide() implementation.

group(match='.*')
    Select a Group of model sites for joint guidance.

    Parameters
    match (str) -- A regex string matching names of model sample sites.
    Returns
    A group of model sites.
    Return type
    Group

map_estimate(name)
    Construct a maximum a posteriori (MAP) guide using Delta distributions.

    Parameters
    name (str) -- The name of a model sample site.
    Returns
    A sampled value.
    Return type
    torch.Tensor

13.2 easy_guide

easy_guide(model)
    Convenience decorator to create an EasyGuide. The following are equivalent:

    # Version 1. Decorate a function.
    @easy_guide(model)
    def guide(self, foo, bar):
        return my_guide(foo, bar)

    # Version 2. Create and instantiate a subclass of EasyGuide.
    class Guide(EasyGuide):
        def guide(self, foo, bar):
            return my_guide(foo, bar)
    guide = Guide(model)

    Note @easy_guide wrappers cannot be pickled; to build a guide that can be pickled, instead subclass from EasyGuide.

    Parameters
    model (callable) -- a Pyro model.

13.3 Group

class Group(guide, sites)
    Bases: object
    An autoguide helper to match a group of model sites.

    Variables
    • event_shape (torch.Size) -- The total flattened concatenated shape of all matching sample sites in the model.
- **prototype_sites (list)** – A list of all matching sample sites in a prototype trace of the model.

**Parameters**
- **guide (EasyGuide)** – An easyguide instance.
- **sites (list)** – A list of model sites.

```python
guide
sample(guide_name, fn, infer=None)
```
Wrapper around `pyro.sample()` to create a single auxiliary sample site and then unpack to multiple sample sites for model replay.

**Parameters**
- **guide_name (str)** – The name of the auxiliary guide site.
- **fn (callable)** – A distribution with shape `self.event_shape`.
- **infer (dict)** – Optional inference configuration dict.

**Returns** A pair `(guide_z, model_zs)` where `guide_z` is the single concatenated blob and `model_zs` is a dict mapping site name to constrained model sample.

**Return type** tuple

```python
map_estimate()
```
Construct a maximum a posteriori (MAP) guide using Delta distributions.

**Returns** A dict mapping model site name to sampled value.

**Return type** dict
CHAPTER 14

Epidemiology

Warning: Code in pyro.contrib.epidemiology is under development. This code makes no guarantee about maintaining backwards compatibility.

pyro.contrib.epidemiology provides a modeling language for a class of stochastic discrete-time discrete-count compartmental models. This module implements black-box inference (both Stochastic Variational Inference and Hamiltonian Monte Carlo), prediction of latent variables, and forecasting of future trajectories.

For example usage see the following tutorials:

- Introduction
- Univariate models
- Regional models
- Inference via auxiliary variable HMC

14.1 Base Compartmental Model

class CompartmentalModel (compartments, duration, population, *, approximate=())
Bases: abc.ABC

Abstract base class for discrete-time discrete-value stochastic compartmental models.

Derived classes must implement methods initialize() and transition(). Derived classes may optionally implement global_model(), compute_flows(), and heuristic().

Example usage:

```python
# First implement a concrete derived class.
class MyModel (CompartmentalModel):
    def __init__(self, ...): ...
```

(continues on next page)
def global_model(self): ...
def initialize(self, params): ...
def transition(self, params, state, t): ...

# Run inference to fit the model to data.
model = MyModel(...)  
model.fit_svi(num_samples=100) # or .fit_mcmc(...)  
R0 = model.samples["R0"] # An example parameter.  
print("R0 = {:.3g} ± {:.3g}".format(R0.mean(), R0.std()))

# Predict latent variables.
samples = model.predict()

# Forecast forward.
samples = model.predict(forecast=30)

# You can assess future interventions (applied after `duration`) by
# storing them as attributes that are read by your derived methods.
model.my_intervention = False  
samples1 = model.predict(forecast=30)  
model.my_intervention = True  
samples2 = model.predict(forecast=30)  
effect = samples2["my_result"].mean() - samples1["my_result"].mean()  
print("average effect = {:.3g}".format(effect))

An example workflow is to use cheaper approximate inference while finding good model structure and priors, then move to more accurate but more expensive inference once the model is plausible.

1. Start with .fit_svi(guide_rank=0, num_steps=2000) for cheap inference while you search for a good model.

2. Additionally infer long-range correlations by moving to a low-rank multivariate normal guide via .fit_svi(guide_rank=None, num_steps=5000).

3. Optionally additionally infer non-Gaussian posterior by moving to the more expensive (but still approximate via moment matching) .fit_mcmc(num_quant_bins=1, num_samples=10000, num_chains=2).

4. Optionally improve fit around small counts by moving the the more expensive enumeration-based algorithm .fit_mcmc(num_quant_bins=4, num_samples=10000, num_chains=2) (GPU recommended).

Variables samples (dict) – Dictionary of posterior samples.

Parameters

• compartments (list) – A list of strings of compartment names.

• duration (int) – The number of discrete time steps in this model.

• population (int or torch.Tensor) – Either the total population of a single-region model or a tensor of each region’s population in a regional model.

• approximate (tuple) – Names of compartments for which pointwise approximations should be provided in transition(), e.g. if you specify approximate=("I") then the state["I_approx"] will be a continuous-valued non-enumerated point estimate of state["I"]. Approximations are useful to reduce computational cost. Approximations are continuous-valued with support (-0.5, population + 0.5).
time_plate
A pyro.plate for the time dimension.

region_plate
Either a pyro.plate or a trivial ExitStack depending on whether this model .is_regional.

full_mass
A list of a single tuple of the names of global random variables.

series
A frozenset of names of sample sites that are sampled each time step.

global_model()
Samples and returns any global parameters.

initialize(params)
Returns initial counts in each compartment.

Parameters
• params – The global params returned by global_model().

Returns
A dict mapping compartment name to initial value.

Return type
dict

transition(params, state, t)
Forward generative process for dynamics.

This inputs a current state and stochastically updates that state in-place.

Note that this method is called under multiple different interpretations, including batched and vectorized interpretations. During generate() this is called to generate a single sample. During heuristic() this is called to generate a batch of samples for SMC. During fit_mcmc() this is called both in vectorized form (vectorizing over time) and in sequential form (for a single time step); both forms enumerate over discrete latent variables. During predict() this is called to forecast a batch of samples, conditioned on posterior samples for the time interval [0:duration].

Parameters
• params – The global params returned by global_model().
• state(dict) – A dictionary mapping compartment name to current tensor value. This should be updated in-place.
• t(int or slice) – A time-like index. During inference t may be either a slice (for vectorized inference) or an integer time index. During prediction t will be integer time index.

finalize(params, prev, curr)
Optional method for likelihoods that depend on entire time series.

This should be used only for non-factorizable likelihoods that couple states across time. Factorizable likelihoods should instead be added to the transition() method, thereby enabling their use in heuristic() initialization. Since this method is called only after the last time step, it is not used in heuristic() initialization.

Warning: This currently does not support latent variables.

Parameters
• params – The global params returned by global_model().
Pyro Documentation

• **prev** *(dict)* –
• **curr** *(dict)* – Dictionaries mapping compartment name to tensor of entire time series. These two parameters are offset by 1 step, thereby making it easy to compute time series of fluxes. For quantized inference, this uses the approximate point estimates, so users must request any needed time series in **__init__()**, e.g. by calling `super().__init__(..., approximate=("I", "E"))` if likelihood depends on the I and E time series.

**compute_flows** *(prev, curr, t)*
Computes flows between compartments, given compartment populations before and after time step \( t \).

The default implementation assumes sequential flows terminating in an implicit compartment named “R”. For example if:

```
compartment_names = ("S", "E", "I")
```

the default implementation computes at time step \( t = 9 \):

- `flows["S2E_9"] = prev["S"] - curr["S"]`
- `flows["E2I_9"] = prev["E"] - curr["E"] + flows["S2E_9"]`
- `flows["I2R_9"] = prev["I"] - curr["I"] + flows["E2I_9"]`

For more complex flows (non-sequential, branching, looping, duplicating, etc.), users may override this method.

**Parameters**

• **state** *(dict)* – A dictionary mapping compartment name to current tensor value. This should be updated in-place.

  • **t** *(int or slice)* – A time-like index. During inference \( t \) may be either a slice (for vectorized inference) or an integer time index. During prediction \( t \) will be integer time index.

**Returns** A dict mapping flow name to tensor value.

**Return type** dict

**generate** *(fixed={})*
Generate data from the prior.

  • **Pram dict fixed** A dictionary of parameters on which to condition. These must be top-level parentless nodes, i.e. have no upstream stochastic dependencies.

**Returns** A dictionary mapping sample site name to sampled value.

**Return type** dict

**fit_svi** *(\*, num_samples=100, num_steps=2000, num_particles=32, learning_rate=0.1, learning_rate_decay=0.01, betas=(0.8, 0.99), haar=True, init_scale=0.01, guide_rank=0, jit=False, log_every=200, **options)***
Runs stochastic variational inference to generate posterior samples.

This runs **SVI**, setting the **.samples** attribute on completion.

This approximate inference method is useful for quickly iterating on probabilistic models.

**Parameters**

  • **num_samples** *(int)* – Number of posterior samples to draw from the trained guide. Defaults to 100.
• **num_steps** *(int)* – Number of SVI steps.

• **num_particles** *(int)* – Number of SVI particles per step.

• **learning_rate** *(int)* – Learning rate for the `ClippedAdam` optimizer.

• **learning_rate_decay** *(int)* – Learning rate for the `ClippedAdam` optimizer. Note this is decay over the entire schedule, not per-step decay.

• **betas** *(tuple)* – Momentum parameters for the `ClippedAdam` optimizer.

• **haar** *(bool)* – Whether to use a Haar wavelet reparameterizer.

• **guide_rank** *(int)* – Rank of the auto normal guide. If zero (default) use an `AutoNormal` guide. If a positive integer or None, use an `AutoLowRankMultivariateNormal` guide. If the string “full”, use an `AutoMultivariateNormal` guide. These latter two require more `num_steps` to fit.

• **init_scale** *(float)* – Initial scale of the `AutoLowRankMultivariateNormal` guide.

• **jit** *(bool)* – Whether to use a jit compiled ELBO.

• **log_every** *(int)* – How often to log svi losses.

• **heuristic_num_particles** *(int)* – Passed to `heuristic()` as `num_particles`. Defaults to 1024.

**Returns**  
Time series of SVI losses (useful to diagnose convergence).

**Return type**  
list

**fit_mcmc(** **options** **)**

Runs NUTS inference to generate posterior samples.

This uses the NUTS kernel to run MCMC, setting the `.samples` attribute on completion.

This uses an asymptotically exact enumeration-based model when `num_quant_bins > 1`, and a cheaper moment-matched approximate model when `num_quant_bins == 1`. Note that computational cost is exponential in `num_quant_bins`. Defaults to 1 for relaxed inference.

**Parameters**

• **options** – Options passed to MCMC. The remaining options are pulled out and have special meaning.

• **num_samples** *(int)* – Number of posterior samples to draw via mcmc. Defaults to 100.

• **max_tree_depth** *(int)* – (Default 5). Max tree depth of the NUTS kernel.

• **full_mass** – Specification of mass matrix of the NUTS kernel. Defaults to full mass over global random variables.

• **arrowhead_mass** *(bool)* – Whether to treat full_mass as the head of an arrowhead matrix versus simply as a block. Defaults to False.

• **num_quant_bins** *(int)* – If greater than 1, use asymptotically exact inference via local enumeration over this many quantization bins. If equal to 1, use continuous-valued relaxed approximate inference. Defaults to 1 for relaxed inference.

• **haar** *(bool)* – Whether to use a Haar wavelet reparameterizer. Defaults to True.

• **haar_full_mass** *(int)* – Number of low frequency Haar components to include in the full mass matrix. If haar=False then this is ignored. Defaults to 10.
• **heuristic Num Particles** *(int)* – Passed to `heuristic()` as `num_particles`. Defaults to 1024.

**Returns** An MCMC object for diagnostics, e.g. `MCMC.summary()`.

**Return type** *MCMC*

**predict** *(forecast=0)*
Predict latent variables and optionally forecast forward.

This may be run only after `fit_mcmc()` and draws the same `num_samples` as passed to `fit_mcmc()`.

**Parameters**
- **forecast** *(int)* – The number of time steps to forecast forward.

**Returns** A dictionary mapping sample site name (or compartment name) to a tensor whose first dimension corresponds to sample batching.

**Return type** *dict*

**heuristic** *(num_particles=1024, ess_threshold=0.5, retries=10)*
Finds an initial feasible guess of all latent variables, consistent with observed data. This is needed because not all hypotheses are feasible and HMC needs to start at a feasible solution to progress.

The default implementation attempts to find a feasible state using `SMCFilter` with proposals from the prior. However this method may be overridden in cases where SMC performs poorly e.g. in high-dimensional models.

**Parameters**
- **num_particles** *(int)* – Number of particles used for SMC.
- **ess_threshold** *(float)* – Effective sample size threshold for SMC.

**Returns** A dictionary mapping sample site name to tensor value.

**Return type** *dict*

### 14.2 Example Models

#### 14.2.1 Simple SIR

class **SimpleSIRModel** *(population, recovery_time, data)*

Susceptible-Infected-Recovered model.

To customize this model we recommend forking and editing this class.

This is a stochastic discrete-time discrete-state model with three compartments: “S” for susceptible, “I” for infected, and “R” for recovered individuals (the recovered individuals are implicit: R = population - S - I) with transitions S → I → R.

**Parameters**
- **population** *(int)* – Total population = S + I + R.
- **recovery_time** *(float)* – Mean recovery time (duration in state I). Must be greater than 1.
- **data** *(iterable)* – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S → I transitions. This allows false negative but no false positives.
14.2.2 Simple SEIR

```python
class SimpleSEIRModel (population, incubation_time, recovery_time, data)
    Susceptible-Exposed-Infected-Recovered model.

    To customize this model we recommend forking and editing this class.

    This is a stochastic discrete-time discrete-state model with four compartments: “S” for susceptible, “E” for exposed, “I” for infected, and “R” for recovered individuals (the recovered individuals are implicit: R = population - S - E - I) with transitions S -> E -> I -> R.

    Parameters
    • population (int) – Total population = S + E + I + R.
    • incubation_time (float) – Mean incubation time (duration in state E). Must be greater than 1.
    • recovery_time (float) – Mean recovery time (duration in state I). Must be greater than 1.
    • data (iterable) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S -> E transitions. This allows false negative but no false positives.
```

14.2.3 Simple SEIRD

```python
class SimpleSEIRDModel (population, incubation_time, recovery_time, mortality_rate, data)
    Susceptible-Exposed-Infected-Recovered-Dead model.

    To customize this model we recommend forking and editing this class.

    This is a stochastic discrete-time discrete-state model with four compartments: “S” for susceptible, “E” for exposed, “I” for infected, “D” for deceased individuals, and “R” for recovered individuals (the recovered individuals are implicit: R = population - S - E - I - D) with transitions S -> E -> I -> R and I -> D.

    Because the transitions are not simple linear succession, this model implements a custom compute_flows() method.

    Parameters
    • population (int) – Total population = S + E + I + R + D.
    • incubation_time (float) – Mean incubation time (duration in state E). Must be greater than 1.
    • recovery_time (float) – Mean recovery time (duration in state I). Must be greater than 1.
    • mortality_rate (float) – Portion of infections resulting in death. Must be in the open interval (0, 1).
    • data (iterable) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S -> E transitions. This allows false negative but no false positives.
```
14.2.4 Overdispersed SIR

class OverdispersedSIRModel (population, recovery_time, data)
Generalizes SimpleSIRModel with overdispersed distributions.

To customize this model we recommend forking and editing this class.

This adds a single global overdispersion parameter controlling overdispersion of the transition and observation distributions. See binomial_dist() and beta_binomial_dist() for distributional details. For prior work incorporating overdispersed distributions see [1,2,3,4].

References:


Parameters

• population (int) – Total population = S + I + R.
• recovery_time (float) – Mean recovery time (duration in state I). Must be greater than 1.
• data (iterable) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S → I transitions. This allows false negative but no false positives.

14.2.5 Overdispersed SEIR

class OverdispersedSEIRModel (population, incubation_time, recovery_time, data)
Generalizes SimpleSEIRModel with overdispersed distributions.

To customize this model we recommend forking and editing this class.

This adds a single global overdispersion parameter controlling overdispersion of the transition and observation distributions. See binomial_dist() and beta_binomial_dist() for distributional details. For prior work incorporating overdispersed distributions see [1,2,3,4].

References:


Parameters

- *population* (*int*) – Total population = S + E + I + R.
- *incubation_time* (*float*) – Mean incubation time (duration in state E). Must be greater than 1.
- *recovery_time* (*float*) – Mean recovery time (duration in state I). Must be greater than 1.
- *data* (*iterable*) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S \rightarrow E transitions. This allows false negative but no false positives.

14.2.6 Superspreading SIR

class SuperspreadingSIRModel (population, recovery_time, data)
Generalizes SimpleSIRModel by adding superspreading effects.

To customize this model we recommend forking and editing this class.

This model accounts for superspreading (overdispersed individual reproductive number) by assuming each infected individual infects BetaBinomial-many susceptible individuals, where the BetaBinomial distribution acts as an overdispersed Binomial distribution, adapting the more standard NegativeBinomial distribution that acts as an overdispersed Poisson distribution [1,2] to the setting of finite populations. To preserve Markov structure, we follow [2] and assume all infections by a single individual occur on the single time step where that individual makes an I \rightarrow R transition. That is, whereas the SimpleSIRModel assumes infected individuals infect Binomial(S, R/τ)-many susceptible individuals during each infected time step (over τ-many steps on average), this model assumes they infect BetaBinomial(k, ..., S)-many susceptible individuals but only on the final time step before recovering.

References


Parameters

- *population* (*int*) – Total population = S + I + R.
- *recovery_time* (*float*) – Mean recovery time (duration in state I). Must be greater than 1.
- *data* (*iterable*) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S \rightarrow I transitions. This allows false negative but no false positives.
14.2.7 Superspreading SEIR

class SuperspreadingSEIRModel(population, incubation_time, recovery_time, data, *, 
leaf_times=None, coal_times=None)

Generalizes SimpleSEIRModel by adding superspreading effects.

To customize this model we recommend forking and editing this class.

This model accounts for superspreading (overdispersed individual reproductive number) by assuming each infected individual infects BetaBinomial-many susceptible individuals, where the BetaBinomial distribution acts as an overdispersed Binomial distribution, adapting the more standard NegativeBinomial distribution that acts as an overdispersed Poisson distribution [1,2] to the setting of finite populations. To preserve Markov structure, we follow [2] and assume all infections by a single individual occur on the single time step where that individual makes an I -> R transition. That is, whereas the SimpleSEIRModel assumes infected individuals infect Binomial(S,R/tau)-many susceptible individuals during each infected time step (over tau-many steps on average), this model assumes they infect BetaBinomial(k,...,S)-many susceptible individuals but only on the final time step before recovering.

This model also adds an optional likelihood for observed phylogenetic data in the form of coalescent times. These are provided as a pair (leaf_times, coal_times) of times at which genomes are sequenced and lineages coalesce, respectively. We incorporate this data using the CoalescentRateLikelihood with base coalescence rate computed from the S and I populations. This likelihood is independent across time and preserves the Markov propert needed for inference.

References


Parameters

- population (int) – Total population = S + E + I + R.
- incubation_time (float) – Mean incubation time (duration in state E). Must be greater than 1.
- recovery_time (float) – Mean recovery time (duration in state I). Must be greater than 1.
- data (iterable) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of S -> E transitions. This allows false negative but no false positives.

14.2.8 Heterogeneous SIR

class HeterogeneousSIRModel(population, recovery_time, data)

Generalizes SimpleSIRModel by allowing Rt and rho to vary in time.

To customize this model we recommend forking and editing this class.

In this model, the response rate rho is piecewise constant with unknown value over three pieces. The reproductive number Rt is a product of a constant R0 with a factor beta that drifts via Brownian motion in log space. Both rho and Rt are available as time series.

Parameters
• **population** (*int*) – Total population = \( S + I + R \).

• **recovery_time** (*float*) – Mean recovery time (duration in state I). Must be greater than 1.

• **data** (*iterable*) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of \( S \to I \) transitions. This allows false negative but no false positives.

### 14.2.9 Sparse SIR

**class SparseSIRModel** (*population*, *recovery_time*, *data*, *mask*)

Generalizes **SimpleSIRModel** to allow sparsely observed infections.

To customize this model we recommend forking and editing this class.

This model allows observations of cumulative infections at uneven time intervals. To preserve Markov structure (and hence tractable inference) this model adds an auxiliary compartment \( O \) denoting the fully-observed cumulative number of observations at each time point. At observed times (when \( \text{mask}[t] == \text{True} \)) \( O \) must exactly match the provided data; between observed times \( O \) stochastically imputes the provided data.

This model demonstrates how to implement a custom `compute_flows()` method. A custom method is needed in this model because inhabitants of the \( S \) compartment can transition to both the \( I \) and \( O \) compartments, allowing duplication.

**Parameters**

• **population** (*int*) – Total population = \( S + I + R \).

• **recovery_time** (*float*) – Mean recovery time (duration in state I). Must be greater than 1.

• **data** (*iterable*) – Time series of cumulative observed infections. Whenever \( \text{mask}[t] == \text{True} \), \( \text{data}[t] \) corresponds to an observation; otherwise \( \text{data}[t] \) can be arbitrary, e.g. NAN.

• **mask** (*iterable*) – Boolean time series denoting whether an observation is made at each time step. Should satisfy \( \text{len(mask)} == \text{len(data)} \).

### 14.2.10 Unknown Start SIR

**class UnknownStartSIRModel** (*population*, *recovery_time*, *pre_obs_window*, *data*)

Generalizes **SimpleSIRModel** by allowing unknown date of first infection.

To customize this model we recommend forking and editing this class.

This model demonstrates:

1. How to incorporate spontaneous infections from external sources;

2. How to incorporate time-varying piecewise \( \rho \) by supporting forecasting in `transition()`.

3. How to override the `predict()` method to compute extra statistics.

**Parameters**

• **population** (*int*) – Total population = \( S + I + R \).

• **recovery_time** (*float*) – Mean recovery time (duration in state I). Must be greater than 1.
• **pre_obs_window** (*int*) – Number of time steps before beginning **data** where the initial infection may have occurred. Must be positive.

• **data** (*iterable*) – Time series of new observed infections. Each time step is Binomial distributed between 0 and the number of $S \rightarrow I$ transitions. This allows false negative but no false positives.

### 14.2.11 Regional SIR

**class RegionalSIRModel** (*population*, *coupling*, *recovery_time*, *data*)

Generalizes **SimpleSIRModel** to simultaneously model multiple regions with weak coupling across regions.

To customize this model we recommend forking and editing this class.

Regions are coupled by a coupling matrix with entries in $[0,1]$. The all ones matrix is equivalent to a single region. The identity matrix is equivalent to a set of independent regions. This need not be symmetric, but symmetric matrices are probably more physically plausible. The expected number of new infections each time step $S\rightarrow I$ is Binomial distributed with mean:

```
E[S\rightarrow I] = \frac{S (1 - \left(1 - \frac{R_0}{\text{population} \times \text{coupling}}\right) \times (I \times \text{coupling}))}{R_0 S (I \times \text{coupling}) \times (\text{population} \times \text{coupling})} \# \text{ for small } I
```

Thus in a nearly entirely susceptible population, a single infected individual infects approximately $R_0$ new individuals on average, independent of coupling.

This model demonstrates:

1. How to create a regional model with a population vector.
2. How to model both homogeneous parameters (here $R_0$) and heterogeneous parameters with hierarchical structure (here rho) using `self.region_plate`.
3. How to approximately couple regions in `transition()` using state["i_approx"].

**Parameters**

• **population** (*torch.Tensor*) – Tensor of per-region populations, defining $\text{population} = S + I + R$.

• **coupling** (*torch.Tensor*) – Pairwise coupling matrix. Entries should be in $[0,1]$.

• **recovery_time** (*float*) – Mean recovery time (duration in state $I$). Must be greater than 1.

• **data** (*iterable*) – Time x Region sized tensor of new observed infections. Each time step is vector of Binomials distributed between 0 and the number of $S \rightarrow I$ transitions. This allows false negative but no false positives.

### 14.2.12 Heterogeneous Regional SIR

**class HeterogeneousRegionalSIRModel** (*population*, *coupling*, *recovery_time*, *data*)

Generalizes **RegionalSIRModel** by allowing $R_t$ and rho to vary in time.

To customize this model we recommend forking and editing this class.

In this model, the response rate rho varies across time and region, whereas the reproductive number $R_t$ varies in time but is shared among regions. Both parameters drift according to transformed Brownian motion with learned drift rate.

This model demonstrates how to model hierarchical latent time series, other than compartmental variables.
Parameters

- **population** (*torch.Tensor*) – Tensor of per-region populations, defining population = S + I + R.
- **coupling** (*torch.Tensor*) – Pairwise coupling matrix. Entries should be in [0,1].
- **recovery_time** (*float*) – Mean recovery time (duration in state I). Must be greater than 1.
- **data** (*iterable*) – Time x Region sized tensor of new observed infections. Each time step is vector of Binomials distributed between 0 and the number of S -> I transitions. This allows false negative but no false positives.

### 14.3 Distributions

**set_approx_sample_thresh** (*thresh*)

EXPERIMENTAL Context manager / decorator to temporarily set the global default value of Binomial.approx_sample_thresh, thereby decreasing the computational complexity of sampling from Binomial, BetaBinomial, ExtendedBinomial, ExtendedBetaBinomial, and distributions returned by infection_dist().

This is useful for sampling from very large total_count.

This is used internally by **CompartmentalModel**.

**Parameters**

- **thresh** (*int or float.*) – New temporary threshold.

**set_approx_log_prob_tol** (*tol*)

EXPERIMENTAL Context manager / decorator to temporarily set the global default value of Binomial.approx_log_prob_tol and BetaBinomial.approx_log_prob_tol, thereby decreasing the computational complexity of scoring Binomial and BetaBinomial distributions.

This is used internally by **CompartmentalModel**.

**Parameters**

- **tol** (*int or float.*) – New temporary tolold.

**binomial_dist** (*total_count, probs, *, overdispersion=0.0*)

Returns a Beta-Binomial distribution that is an overdispersed version of a Binomial distribution, according to a parameter overdispersion, typically set in the range 0.1 to 0.5.

This is useful for (1) fitting real data that is overdispersed relative to a Binomial distribution, and (2) relaxing models of large populations to improve inference. In particular the overdispersion parameter lower bounds the relative uncertainty in stochastic models such that increasing population leads to a limiting scale-free dynamical system with bounded stochasticity, in contrast to Binomial-based SDEs that converge to deterministic ODEs in the large population limit.

This parameterization satisfies the following properties:

1. Variance increases monotonically in overdispersion.
2. overdispersion = 0 results in a Binomial distribution.
3. overdispersion lower bounds the relative uncertainty std_dev / (total_count * p * q), where probs = p = 1 - q, and serves as an asymptote for relative uncertainty as total_count → ∞. This contrasts the Binomial whose relative uncertainty tends to zero.
4. If \( X \sim \text{binomial_dist}(n, p, \text{overdispersion}=\sigma) \) then in the large population limit \( n \to \infty \), the scaled random variable \( \frac{X}{n} \) converges in distribution to LogitNormal(\( \log(p/(1-p)) \), \( \sigma \)).
To achieve these properties we set \( p = \text{probs}, q = 1 - p \), and:

\[
\text{concentration} = \frac{1}{(p \times q \times \text{overdispersion}^2)} - 1
\]

Parameters

- **total_count** (int or torch.Tensor) – Number of Bernoulli trials.
- **probs** (float or torch.Tensor) – Event probabilities.
- **overdispersion** (float or torch.tensor) – Amount of overdispersion, in the half open interval \([0,2)\). Defaults to zero.

**beta_binomial_dist** (concentration1, concentration0, total_count, *, overdispersion=0.0)

Returns a Beta-Binomial distribution that is an overdispersed version of a the usual Beta-Binomial distribution, according to an extra parameter overdispersion, typically set in the range 0.1 to 0.5.

Parameters

- **concentration1** (float or torch.Tensor) – 1st concentration parameter (alpha) for the Beta distribution.
- **concentration0** (float or torch.Tensor) – 2nd concentration parameter (beta) for the Beta distribution.
- **total_count** (float or torch.Tensor) – Number of Bernoulli trials.
- **overdispersion** (float or torch.tensor) – Amount of overdispersion, in the half open interval \([0,2)\). Defaults to zero.

**infection_dist** (*, individual_rate, num_infectious, num_susceptible=inf, population=inf, concentration=inf, overdispersion=0.0)

Create a Distribution over the number of new infections at a discrete time step.

This returns a Poisson, Negative-Binomial, Binomial, or Beta-Binomial distribution depending on whether population and concentration are finite. In Pyro models, the population is usually finite. In the limit population \(\rightarrow\infty\) and num_susceptible/population \(\rightarrow 1\), the Binomial converges to Poisson and the Beta-Binomial converges to Negative-Binomial. In the limit concentration \(\rightarrow\infty\), the Negative-Binomial converges to Poisson and the Beta-Binomial converges to Binomial.

The overdispersed distributions (Negative-Binomial and Beta-Binomial returned when concentration \(\leq\infty\)) are useful for modeling superspreader individuals [1,2]. The finitely supported distributions Binomial and Negative-Binomial are useful in small populations and in probabilistic programming systems where truncation or censoring are expensive [3].

References


Parameters

- **individual_rate** – The mean number of infections per infectious individual per time step in the limit of large population, equal to \(R_0 / \tau\) where \(R_0\) is the basic reproductive number and \(\tau\) is the mean duration of infectiousness.
- **num_infectious** – The number of infectious individuals at this time step, sometimes $I$, sometimes $E+I$.

- **num_susceptible** – The number $S$ of susceptible individuals at this time step. This defaults to an infinite population.

- **population** – The total number of individuals in a population. This defaults to an infinite population.

- **concentration** – The concentration or dispersion parameter $k$ in overdispersed models of superspreaders [1,2]. This defaults to minimum variance concentration $= \infty$.

- **overdispersion** (*float or torch.tensor*) – Amount of overdispersion, in the half open interval $[0,2)$. Defaults to zero.

```python
class CoalescentRateLikelihood(leaf_times, coal_times, duration, *, validate_args=None)
Bases: object

EXPERIMENTAL This is not a Distribution, but acts as a transposed version of CoalescentTimesWithRate making the elements of rate_grid independent and thus compatible with plate and poutine.markov. For non-batched inputs the following are all equivalent likelihoods:

```python
# Version 1.
pyro.sample("coalescent",
    CoalescentTimesWithRate(leaf_times, rate_grid),
    obs=coal_times)

# Version 2. using pyro.plate
likelihood = CoalescentRateLikelihood(leaf_times, coal_times, len(rate_grid))
with pyro.plate("time", len(rate_grid)):
    pyro.factor("coalescent", likelihood(rate_grid))

# Version 3. using pyro.markov
likelihood = CoalescentRateLikelihood(leaf_times, coal_times, len(rate_grid))
for t in pyro.markov(range(len(rate_grid))):
    pyro.factor("coalescent_{}/".format(t), likelihood(rate_grid[t], t))
```

The third version is useful for e.g. SMCFilter where rate_grid might be computed sequentially.

**Parameters**

- **leaf_times** (*torch.Tensor*) – Tensor of times of sampling events, i.e. leaf nodes in the phylogeny. These can be arbitrary real numbers with arbitrary order and duplicates.

- **coal_times** (*torch.Tensor*) – A tensor of coalescent times. These denote sets of size $\text{leaf_times.size(-1)} - 1$ along the trailing dimension and should be sorted along that dimension.

- **duration** (*int*) – Size of the rate grid, $\text{rate_grid.size(-1)}$.

**__call__** (*rate_grid, t=slice(None, None, None))
Computes the likelihood of [1] equations 7-9 for one or all time points.

**References**


- **Parameters**
Pyro Documentation

- **rate_grid** *(torch.Tensor)* – Tensor of base coalescent rates (pairwise rate of coalescence). For example in a simple SIR model this might be \( \beta S / I \). The rightmost dimension is time, and this tensor represents a (batch of) rates that are piecewise constant in time.

- **time** *(int or slice)* – Optional time index by which the input was sliced, as in `rate_grid[..., t]`. This can be an integer for sequential models or `slice(None)` for vectorized models.

  **Returns** Likelihood \( p(\text{coal\_times} | \text{leaf\_times}, \text{rate\_grid}) \), or a part of that likelihood corresponding to a single time step.

  **Return type** `torch.Tensor`

---

**bio_phylo_to_times** *(tree, *, get_time=None)*

Extracts coalescent summary statistics from a phylogeny, suitable for use with `CoalescentRateLikelihood`.

**Parameters**


- **get_time** *(callable)* – Optional function to extract the time point of each sub-Clade. If absent, times will be computed by cumulative `branch_length`.

  **Returns** A pair of Tensors `(leaf\_times, coal\_times)` where `leaf\_times` are times of sampling events (leaf nodes in the phylogenetic tree) and `coal\_times` are times of coalescences (leaf nodes in the phylogenetic binary tree).

  **Return type** `tuple`
15.1 Datasets

15.1.1 Multi MNIST

This script generates a dataset similar to the Multi-MNIST dataset described in [1].


```python
imresize(arr, size)
sample_one(canvas_size, mnist)
sample_multi(num_digits, canvas_size, mnist)
mk_dataset(n, mnist, max_digits, canvas_size)
load_mnist(root_path)
load(root_path)
```

15.1.2 BART Ridership

```python
load_bart_od()
```

Load a dataset of hourly origin-destination ridership counts for every pair of BART stations during the years 2011-2019.

**Source** [https://www.bart.gov/about/reports/ridership](https://www.bart.gov/about/reports/ridership)

This downloads the dataset the first time it is called. On subsequent calls this reads from a local cached file `.pkl.bz2`. This attempts to download a preprocessed compressed cached file maintained by the Pyro team. On cache hit this should be very fast. On cache miss this falls back to downloading the original data source and preprocessing the dataset, requiring about 350MB of file transfer, storing a few GB of temp files, and taking upwards of 30 minutes.
Returns

a dataset is a dictionary with fields:

- "stations": a list of strings of station names
- "start_date": a `datetime.datetime` for the first observation
- "counts": a `torch.FloatTensor` of ridership counts, with shape `(num_hours, len(stations), len(stations))`.

load_fake_od()

Create a tiny synthetic dataset for smoke testing.

15.2 Utilities

get_data_loader(dataset_name, data_dir, batch_size=1, dataset_transforms=None, is_training_set=True, shuffle=True)

print_and_log(logger, msg)

get_data_directory(filepath=None)
pyro.contrib.forecast is a lightweight framework for experimenting with a restricted class of time series models and inference algorithms using familiar Pyro modeling syntax and PyTorch neural networks.

Models include hierarchical multivariate heavy-tailed time series of ~1000 time steps and ~1000 separate series. Inference combines subsample-compatible variational inference with Gaussian variable elimination based on the GaussianHMM class. Inference using Hamiltonian Monte Carlo sampling is also supported with HMCForecaster. Forecasts are in the form of joint posterior samples at multiple future time steps.

Hierarchical models use the familiar plate syntax for general hierarchical modeling in Pyro. Plates can be subsampled, enabling training of joint models over thousands of time series. Multivariate observations are handled via multivariate likelihoods like MultivariateNormal, GaussianHMM, or LinearHMM. Heavy tailed models are possible by using StudentT or Stable likelihoods, possibly together with LinearHMM and reparameterizers including StudentTReparam, StableReparam, and LinearHMMReparam.

Seasonality can be handled using the helpers periodic_repeat(), periodic_cumsum(), and periodic_features().

See pyro.contrib.timeseries for ways to construct temporal Gaussian processes useful as likelihoods.

For example usage see:

- The univariate forecasting tutorial
- The state space modeling tutorial
- The hierarchical forecasting tutorial
- The forecasting example

### 16.1 Forecaster Interface

class ForecastingModel
    Bases: pyro.nn.module.PyroModule

    Abstract base class for forecasting models.
Derived classes must implement the `model()` method.

```python
model(zero_data, covariates)
```

Generative model definition.

Implementations must call the `predict()` method exactly once.

Implementations must draw all time-dependent noise inside the `time_plate()`. The prediction passed to `predict()` must be a deterministic function of noise tensors that are independent over time. This requirement is slightly more general than state space models.

**Parameters**

- `zero_data` (*Tensor*) – A zero tensor like the input data, but extended to the duration of the `time_plate()`. This allows models to depend on the shape and device of data but not its value.

- `covariates` (*Tensor*) – A tensor of covariates with time dimension -2.

**Returns**

Return value is ignored.

```python
time_plate
```

Returns A plate named “time” with size `covariates.size(-2)` and `dim=-1`. This is available only during model execution.

**Return type**

`plate`

```python
predict(noise_dist, prediction)
```

Prediction function, to be called by `model()` implementations.

This should be called outside of the `time_plate()`.

This is similar to an observe statement in Pyro:

```python
pyro.sample("residual", noise_dist,
    obs=(data - prediction))
```

but with (1) additional reshaping logic to allow time-dependent `noise_dist` (most often a `GaussianHMM` or variant); and (2) additional logic to allow only a partial observation and forecast the remaining data.

**Parameters**

- `noise_dist` (*Distribution*) – A noise distribution with `.event_dim` in (0, 1, 2). `noise_dist` is typically zero-mean or zero-median or zero-mode or somehow centered.

- `prediction` (*Tensor*) – A prediction for the data. This should have the same shape as `data`, but broadcastable to full duration of the `covariates`.

```python
class Forecaster(model, data, covariates, *, guide=None, init_loc_fn=<function init_to_sample>, init_scale=0.1, create_plates=None, optim=None, learning_rate=0.01, betas=(0.9, 0.99), learning_rate_decay=0.1, clip_norm=10.0, time_reparam=None, dct_gradients=False, subsample_aware=False, num_steps=1001, num_particles=1, vectorize_particles=True, warm_start=False, log_every=100)
```

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

Forecaster for a `ForecastingModel` using variational inference.

On initialization, this fits a distribution using variational inference over latent variables and exact inference over the noise distribution, typically a `GaussianHMM` or variant.

After construction this can be called to generate sample forecasts.
Variables **losses** *(list)* – A list of losses recorded during training, typically used to debug convergence. Defined by `loss = -elbo / data.numel()`.

Parameters

- **model** *(ForecastingModel)* – A forecasting model subclass instance.
- **data** *(Tensor)* – A tensor dataset with time dimension -2.
- **covariates** *(Tensor)* – A tensor of covariates with time dimension -2. For models not using covariates, pass a shaped empty tensor `torch.empty(duration, 0)`.
- **guide** *(PyroModule)* – Optional guide instance. Defaults to a `AutoNormal`.
- **init_loc_fn** *(callable)* – A per-site initialization function for the `AutoNormal` guide. Defaults to `init_to_sample()`. See Initialization section for available functions.
- **init_scale** *(float)* – Initial uncertainty scale of the `AutoNormal` guide.
- **create_plates** *(callable)* – An optional function to create plates for subsampling with the `AutoNormal` guide.
- **optim** *(PyroOptim)* – An optional Pyro optimizer. Defaults to a freshly constructed `DCTAdam`.
- **learning_rate** *(float)* – Learning rate used by `DCTAdam`.
- **betas** *(tuple)* – Coefficients for running averages used by `DCTAdam`.
- **learning_rate_decay** *(float)* – Learning rate decay used by `DCTAdam`. Note this is the total decay over all `num_steps`, not the per-step decay factor.
- **clip_norm** *(float)* – Norm used for gradient clipping during optimization. Defaults to 10.0.
- **time_reparam** *(str)* – If not None (default), reparameterize all time-dependent variables via the Haar wavelet transform (if “haar”) or the discrete cosine transform (if “dct”).
- **dct_gradients** *(bool)* – Whether to discrete cosine transform gradients in `DCTAdam`. Defaults to False.
- **subsample_aware** *(bool)* – whether to update gradient statistics only for those elements that appear in a subsample. This is used by `DCTAdam`.
- **num_steps** *(int)* – Number of SVI steps.
- **num_particles** *(int)* – Number of particles used to compute the ELBO.
- **vectorize_particles** *(bool)* – If `num_particles > 1`, determines whether to vectorize computation of the ELBO. Defaults to True. Set to False for models with dynamic control flow.
- **warm_start** *(bool)* – Whether to warm start parameters from a smaller time window. Note this may introduce statistical leakage; usage is recommended for model exploration purposes only and should be disabled when publishing metrics.
- **log_every** *(int)* – Number of training steps between logging messages.

**__call__** *(data, covariates, num_samples, batch_size=None)*

Samples forecasted values of data for time steps in `[t1, t2)`, where `t1 = data.size(-2)` is the duration of observed data and `t2 = covariates.size(-2)` is the extended duration of covariates. For example to forecast 7 days forward conditioned on 30 days of observations, set `t1=30` and `t2=37`.

Parameters

---

16.1. Forecaster Interface 235
Python Documentation

- **data** (*Tensor*) - A tensor dataset with time dimension -2.
- **covariates** (*Tensor*) - A tensor of covariates with time dimension -2. For models not using covariates, pass a shaped empty tensor `torch.empty(duration, 0)`.
- **num_samples** (*int*) - The number of samples to generate.
- **batch_size** (*int*) - Optional batch size for sampling. This is useful for generating many samples from models with large memory footprint. Defaults to `num_samples`.

Returns: A batch of joint posterior samples of shape `(num_samples, 1, ..., 1) + data.shape[:-2] + (t2-t1, data.size(-1))`, where the 1's are inserted to avoid conflict with model plates.

Return type: Tensor

**class HMCForecaster**

```python
class HMCForecaster(model, data, covariates=None, *, num_warmup=1000, num_samples=1000,
num_chains=1, time_reparam=None, dense_mass=False, jit_compile=False,
max_tree_depth=10)
```

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

Forecaster for a `ForecastingModel` using Hamiltonian Monte Carlo.

On initialization, this will run NUTS sampler to get posterior samples of the model.

After construction, this can be called to generate sample forecasts.

Parameters:
- **model** (*ForecastingModel*) - A forecasting model subclass instance.
- **data** (*Tensor*) - A tensor dataset with time dimension -2.
- **covariates** (*Tensor*) - A tensor of covariates with time dimension -2. For models not using covariates, pass a shaped empty tensor `torch.empty(duration, 0)`.
- **num_warmup** (*int*) - number of MCMC warmup steps.
- **num_samples** (*int*) - number of MCMC samples.
- **num_chains** (*int*) - number of parallel MCMC chains.
- **dense_mass** (*bool*) - a flag to control whether the mass matrix is dense or diagonal. Defaults to False.
- **time_reparam** (*str*) - If not None (default), reparameterize all time-dependent variables via the Haar wavelet transform (if “haar”) or the discrete cosine transform (if “dct”).
- **jit_compile** (*bool*) - whether to use the PyTorch JIT to trace the log density computation, and use this optimized executable trace in the integrator. Defaults to False.
- **max_tree_depth** (*int*) - Max depth of the binary tree created during the doubling scheme of the NUTS sampler. Defaults to 10.

```python
__call__(data, covariates, num_samples, batch_size=None)
```

Samples forecasted values of data for time steps in `[t1, t2)` where `t1 = data.size(-2)` is the duration of observed data and `t2 = covariates.size(-2)` is the extended duration of covariates. For example to forecast 7 days forward conditioned on 30 days of observations, set `t1=30` and `t2=37`.

Parameters:
- **data** (*Tensor*) - A tensor dataset with time dimension -2.
- **covariates** (*Tensor*) - A tensor of covariates with time dimension -2. For models not using covariates, pass a shaped empty tensor `torch.empty(duration, 0)`.
- **num_samples** (*int*) - The number of samples to generate.
• **batch_size** (*int*) – Optional batch size for sampling. This is useful for generating many samples from models with large memory footprint. Defaults to `num_samples`.

**Returns** A batch of joint posterior samples of shape \((\text{num}_\text{samples}, 1, \ldots, 1) + \text{data}. \text{shape}[:-2] + (t2-t1, \text{data}.\text{size}(-1))\), where the 1's are inserted to avoid conflict with model plates.

**Return type** Tensor

### 16.2 Evaluation

**eval_mae** *(pred, truth)*
Evaluate mean absolute error, using sample median as point estimate.

**Parameters**

- **pred** (*torch.Tensor*) – Forecasted samples.
- **truth** (*torch.Tensor*) – Ground truth.

**Return type** float

**eval_rmse** *(pred, truth)*
Evaluate root mean squared error, using sample mean as point estimate.

**Parameters**

- **pred** (*torch.Tensor*) – Forecasted samples.
- **truth** (*torch.Tensor*) – Ground truth.

**Return type** float

**eval_crps** *(pred, truth)*
Evaluate continuous ranked probability score, averaged over all data elements.

**References**


**Parameters**

- **pred** (*torch.Tensor*) – Forecasted samples.
- **truth** (*torch.Tensor*) – Ground truth.

**Return type** float

**backtest** *

Backtest a forecasting model on a moving window of (train,test) data.

**Parameters**

- **data** (*Tensor*) – A tensor dataset with time dimension -2.
- **covariates** (*Tensor*) – A tensor of covariates with time dimension -2. For models not using covariates, pass a shaped empty tensor `torch.empty(duration, 0)`.
- **model_fn** (callable) – Function that returns an `ForecastingModel` object.
- **forecaster_fn** (*callable*) – Function that returns a forecaster object (for example, `Forecaster` or `HMCForecaster`) given arguments model, training data, training covariates and keyword arguments defined in `forecaster_options`.

- **metrics** (*dict*) – A dictionary mapping metric name to metric function. The metric function should input a forecast `pred` and ground truth and can output anything, often a number. Example metrics include: `eval_mae()`, `eval_rmse()`, and `eval_crps()`.

- **transform** (*callable*) – An optional transform to apply before computing metrics. If provided this will be applied as `pred, truth = transform(pred, truth)`.

- **train_window** (*int*) – Size of the training window. Be default trains from beginning of data. This must be None if forecaster is `Forecaster` and `forecaster_options["warm_start"]` is true.

- **min_train_window** (*int*) – If `train_window` is None, this specifies the min training window size. Defaults to 1.

- **test_window** (*int*) – Size of the test window. By default forecasts to end of data.

- **min_test_window** (*int*) – If `test_window` is None, this specifies the min test window size. Defaults to 1.

- **stride** (*int*) – Optional stride for test/train split. Defaults to 1.

- **seed** (*int*) – Random number seed.

- **num_samples** (*int*) – Number of samples for forecast. Defaults to 100.

- **batch_size** (*int*) – Batch size for forecast sampling. Defaults to `num_samples`.

- **forecaster_options** (*dict or callable*) – Options dict to pass to forecaster, or callable inputting time window `t0,t1,t2` and returning such a dict. See `Forecaster` for details.

**Returns** A list of dictionaries of evaluation data. Caller is responsible for aggregating the per-window metrics. Dictionary keys include: `train begin time “t0”`, `train/test split time “t1”`, `test end time “t2”`, `“seed”`, `“num_samples”`, `“train_walltime”`, `“test_walltime”`, and one key for each metric.

**Return type** list
17.1 Effect handlers

`enum`\(^{fn=\text{None}, ~\text{*args}, ~\text{**kwargs}}\)
Convenient wrapper of `EnumMessenger`
This version of `EnumMessenger` uses `to_data` to allocate a fresh enumeration dim for each discrete sample site.

`markov`\(^{fn=\text{None}, ~\text{*args}, ~\text{**kwargs}}\)
Convenient wrapper of `MarkovMessenger`
Handler for converting to/from funsors consistent with Pyro’s positional batch dimensions.

**Parameters**

- `history`\(^{\text{int}}\) – The number of previous contexts visible from the current context. Defaults to 1. If zero, this is similar to `pyro.plate`.

- `keep`\(^{\text{bool}}\) – 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 shared ancestors).

`named`\(^{fn=\text{None}, ~\text{*args}, ~\text{**kwargs}}\)
Convenient wrapper of `NamedMessenger`
Base effect handler class for the :func:`pyro.contrib.funsor.to_funsor` and :func:`pyro.contrib.funsor.to_data` primitives. Any effect handlers that invoke these primitives internally or wrap code that does should inherit from `NamedMessenger`.

This design ensures that the global name-dim mapping is reset upon handler exit rather than potentially persisting until the entire program terminates.
plate \((\text{fn}=\text{None}, \, *\text{args}, \, **\text{kwargs})\)

Convenient wrapper of PlateMessenger

Combines new IndepMessenger implementation with existing BroadcastMessenger. Should eventually be a drop-in replacement for pyro.plate.

replay \((\text{fn}=\text{None}, \, *\text{args}, \, **\text{kwargs})\)

Convenient wrapper of ReplayMessenger

This version of ReplayMessenger is almost identical to the original version, except that it calls to_data on the replayed funsor values. This may result in different unpacked shapes, but should produce correct allocations.

trace \((\text{fn}=\text{None}, \, *\text{args}, \, **\text{kwargs})\)

Convenient wrapper of TraceMessenger

Setting `pack_online=True` packs online instead of after the fact, converting all distributions and values to Funsors as soon as they are available.

Setting `pack_online=False` computes information necessary to do packing after execution. Each sample site is annotated with a “dim_to_name” dictionary, which can be passed directly to funsor.to_funsor.

class NamedMessenger (\(\text{first_available_dim}=\text{None}\))

Bases: pyro.poutine.reentrant_messenger.ReentrantMessenger

Base effect handler class for the :func:`pyro.contrib.funsor.to_funsor` and :func:`pyro.contrib.funsor.to_data` primitives. Any effect handlers that invoke these primitives internally or wrap code that does should inherit from NamedMessenger.

This design ensures that the global name-dim mapping is reset upon handler exit rather than potentially persisting until the entire program terminates.

class MarkovMessenger (\(\text{history}=1, \, \text{keep}=\text{False}\))

Bases: pyro.contrib.funsor.handlers.named_messenger.NamedMessenger

Handler for converting to/from funsors consistent with Pyro’s positional batch dimensions.

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 shared ancestors).

class GlobalNamedMessenger (\(\text{first_available_dim}=\text{None}\))

Bases: pyro.contrib.funsor.handlers.named_messenger.NamedMessenger

Base class for any new effect handlers that use the :func:`pyro.contrib.funsor.to_funsor` and :func:`pyro.contrib.funsor.to_data` primitives to allocate DimType.GLOBAL or DimType.VISIBLE dimensions.

Serves as a manual “scope” for dimensions that should not be recycled by :class:`MarkovMessenger`: global dimensions will be considered active until the innermost GlobalNamedMessenger under which they were initially allocated exits.

to_funsor \((x, \, \text{output}=\text{None}, \, \text{dim_to_name}=\text{None}, \, \text{dim_type}=<\text{DimType.LOCAL}: 0>)\)

to_data \((x, \, \text{name_to_dim}=\text{None}, \, \text{dim_type}=<\text{DimType.LOCAL}: 0>)\)

class StackFrame (\(\text{name_to_dim}, \, \text{dim_to_name}=1, \, \text{keep}=\text{False}\))

Bases: object
Consistent bidirectional mapping between integer positional dimensions and names. Can be queried like a dictionary (value = frame[key], frame[key] = value).

```python
class DimType
    Bases: enum.Enum

    Enumerates the possible types of dimensions to allocate
    LOCAL = 0
    GLOBAL = 1
    VISIBLE = 2

class DimRequest
    Args:
    value
    dim_type

    Bases: tuple

    dim_type  # Alias for field number 1
    value     # Alias for field number 0

class DimStack
    Bases: object

    Single piece of global state to keep track of the mapping between names and dimensions.
    Replaces the plate DimAllocator, the enum EnumAllocator, the stack in MarkovMessenger, _param_dims and _value_dims in EnumMessenger, and dim_to_symbol in msg['infer']

    MAX_DIM = -25
    DEFAULT_FIRST_DIM = -5

    set_first_available_dim(dim)

    push_global(frame)
    pop_global()
    push_iter(frame)
    pop_iter()
    push_local(frame)
    pop_local()

    global_frame
    local_frame

    current_write_env
    current_read_env

    allocate(key_to_value_request)

    names_from_batch_shape(batch_shape, dim_type=<DimType.LOCAL: 0>)
```

17.1. Effect handlers
Gaussian Processes

See the Gaussian Processes tutorial for an introduction.

**class** Parameterized

    **Bases:** pyro.nn.module.PyroModule

    A wrapper of PyroModule whose parameters can be set constraints, set priors.

    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.

    Example:

    ```python
>>> class Linear(Parameterized):
...     def __init__(self, a, b):
...         super().__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.a = PyroParam(torch.tensor(1.), constraints.positive)
    >>> linear.b = PyroSample(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())
    ```

    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_prior(name, prior)**

    Sets prior for a 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.infer.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**

### 18.1 Models

#### 18.1.1 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 multivariate 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

```python
>>> X = torch.tensor([[1., 5, 3], [4, 3, 7]])
>>> y = torch.tensor([2., 1])
>>> kernel = gp.kernels.RBF(input_dim=3)
>>> kernel.variance = pyro.nn.PyroSample(dist.Uniform(torch.tensor(0.5), torch.tensor(1.5)))
>>> kernel.lengthscale = pyro.nn.PyroSample(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:

```python
>>> hmc_kernel = HMC(gpr.model)
>>> mcmc = MCMC(hmc_kernel, num_samples=10)
>>> mcmc.run()
>>> ls_name = "kernel.lengthscale"
>>> posterior_ls = mcmc.get_samples()[ls_name]
```

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

```python
>>> 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`:

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

Reference:


**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 stabilize 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 $X_{\text{new}}$:

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

where $\theta$ are parameters of this model.

**Note:** Model’s parameters $\theta$ 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 $p(f^*(X_{\text{new}}))$

**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:

  ```python
  >>> 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:
```python
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:


**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.

### 18.1.2 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 multivariate 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 $\epsilon$ 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.
Pyro Documentation

Reference:

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 stabilize 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 $X_{\text{new}}$:

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

Note: The noise parameter noise($\epsilon$) 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 $p(f^\ast(X_{\text{new}}))$

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 $X_{\text{new}}$, 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

### 18.1.3 SparseGPRegression

The `SparseGPRegression` class is defined as follows:

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

**Basics:** `pyro.contrib.gp.models.model.GPModel`

Sparse Gaussian Process Regression model.

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

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

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

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

where $\epsilon$ 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):**
\[ q(f \mid X, X_u, u) = \mathcal{N}(m, \text{diag}(k(X, X) - Q)), \]

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

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

- 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:


[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 stabilize 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 \( X_{new} \):

\[ p(f^* \mid X_{new}, X, y, k, X_u, \epsilon) = \mathcal{N}(\text{loc}, \text{cov}). \]
Note: The noise parameter \( \text{noise} \), the inducing-point parameter \( X_u \), together with kernel's parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- \( X_{\text{new}} \) (torch.Tensor) – A input data for testing. Note that \( X_{\text{new}}.\text{shape}[1:] \) must be the same as \( \text{self.X.shape}[1:] \).
- \( \text{full_cov} \) (bool) – A flag to decide if we want to predict full covariance matrix or just variance.
- \( \text{noiseless} \) (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 \( p(f^*(X_{\text{new}})) \)

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

18.1.4 VariationalGP

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

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

\[
\begin{align*}
    f & \sim \mathcal{GP}(0, k(X, X)), \\
    y & \sim p(y) = p(y \mid f)p(f),
\end{align*}
\]

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 \( f_{\text{loc}} \) and \( f_{\text{scale_tril}} \), 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

- \( \text{X} \) (torch.Tensor) – A input data for training. Its first dimension is the number of data points.
- \( \text{y} \) (torch.Tensor) – An output data for training. Its last dimension is the number of data points.
- \( \text{kernel} \) (Kernel) – A Pyro kernel object, which is the covariance function \( k \).
- \( \text{Likelihood likelihood} \) (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 correspond to the number of classes.

• **whiten** (*bool*) – A flag to tell if variational parameters \(f\text{\_loc}\) and \(f\text{\_scale\_tril}\) are transformed by the inverse of \(Lff\), where \(Lff\) is the lower triangular decomposition of \(kern(X, X)\). 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 stabilize its Cholesky decomposition.

```python
model()
guide()
forward(Xnew, full_cov=False)
```

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

\[
p(f^* | X_{\text{new}}, X, y, k, f\text{\_loc}, f\text{\_scale\_tril}) = \mathcal{N}(\text{loc}, \text{cov}).
\]

**Note:** Variational parameters \(f\text{\_loc}\), \(f\text{\_scale\_tril}\), 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 \(p(f^*(X_{\text{new}}))\)

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

### 18.1.5 VariationalSparseGP

```python
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:

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

\[
y \sim p(y | f) = p(y | f)p(f),
\]

where \(p(y | f)\) is the likelihood.
We will use a variational approach in this model by approximating $q(f, u)$ to the posterior $p(f, u | y)$. Precisely, $q(f) = p(f | u)q(u)$, where $q(u)$ is a multivariate normal distribution with two parameters $u_{\text{loc}}$ and $u_{\text{scale\_tril}}$, 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 $O(NM^2)$ complexity for training, $O(M^3)$ complexity for testing. Here, $N$ is the number of train inputs, $M$ is the number of inducing inputs. Size of variational parameters is $O(M^2)$.

References:


**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 correspond 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 $u_{\text{loc}}$ and $u_{\text{scale\_tril}}$ are transformed by the inverse of $L_{uu}$, where $L_{uu}$ is the lower triangular decomposition of $\text{kernel}(X_u, X_u)$. 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 stabilize 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 $X_{new}$:

$$p(f^* | X_{new}, X, y, k, X_u, u_{\text{loc}}, u_{\text{scale\_tril}}) = \mathcal{N}(\text{loc}, \text{cov}).$$
Note: Variational parameters $u_{\text{loc}}, u_{\text{scale\_tril}}$, the inducing-point parameter $X_u$, together with kernel’s parameters have been learned from a training procedure (MCMC or SVI).

Parameters

- $X_{\text{new}}$ (torch.Tensor) – A input data for testing. Note that $X_{\text{new}}.\text{shape}[1:]$ 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 $p(f^*(X_{\text{new}}))$

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

### 18.1.6 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: $X_{\text{loc}}$ and $X_{\text{scale\_tril}}$.

For example, we can do dimensional reduction on Iris dataset as follows:

```python
>>> # 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.

>>> 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:


Parameters base_model (GPModel) – A Pyro Gaussian Process model object. Note that base_model.$X$ will be the initial value for the variational parameter $X_{\text{loc}}$.
guide()

```python
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.

## 18.2 Kernels

### 18.2.1 Kernel

```python
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:


**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.

```python
forward(X, Z=None, diag=False)
```

Calculates covariance matrix of inputs on active dimensionals.

**Parameters**

- `X (torch.Tensor)` – A 2D tensor with shape $N \times \text{input}_\text{dim}$.
- `Z (torch.Tensor)` – An (optional) 2D tensor with shape $M \times \text{input}_\text{dim}$.
- `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`

### 18.2.2 Brownian

```python
class Brownian(input_dim, variance=None, active_dims=None)
```

Bases: `pyro.contrib.gp.kernels.kernel.Kernel`

This kernel corresponds to a two-sided Brownian motion (Wiener process):

$$k(x, z) = \begin{cases} 
\sigma^2 \min(|x|, |z|), & \text{if } x \cdot z \geq 0 \\
0, & \text{otherwise}.
\end{cases}$$

18.2. Kernels
Note that the input dimension of this kernel must be 1.

Reference:

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

### 18.2.3 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.

### 18.2.4 Constant

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

\[
\kappa(x, z) = \sigma^2.
\]

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

### 18.2.5 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 \(k(x, z) = x^T(W W^T + D)z\) 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:

Parameters
- `input_dim` (*int*) – Number of feature dimensions of inputs.
- `rank` (*int*) – Optional rank. This is only used if `components` is unspecified. If neither `rank` nor `components` is specified, then `rank` defaults to `input_dim`. 

256 Chapter 18. Gaussian Processes
• **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.

```python
forward(X, Z=None, diag=False)
```

### 18.2.6 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.

```python
forward(X, Z=None, diag=False)
```

### 18.2.7 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 \( x \cdot z \).

### 18.2.8 Exponent

**class Exponent**( *kern*)

Bases: `pyro.contrib.gp.kernels.kernel.Transforming`

Creates a new kernel according to

\[ k_{\text{new}}(x, z) = \exp(k(x, z)). \]

```python
forward(X, Z=None, diag=False)
```

### 18.2.9 Exponential

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

Bases: `pyro.contrib.gp.kernels.isotropic.Isotropy`

Implementation of Exponential kernel:

\[ k(x, z) = \sigma^2 \exp\left(\frac{|x-z|}{l}\right). \]

```python
forward(X, Z=None, diag=False)
```
18.2.10 Isotropy

class Isotropy (input_dim=None, 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.

18.2.11 Linear

class Linear (input_dim=None, 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 Constant kernel.

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

18.2.12 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)

18.2.13 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)
18.2.14 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)*

18.2.15 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:

\[
k(x, z) = \sigma^2 (bias + x \cdot z)^d.
\]

**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)*

18.2.16 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)*

18.2.17 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)*

### 18.2.18 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:

\[
k(x, z) = \sigma^2 \left(1 + 0.5 \times \frac{|x-z|^2}{\alpha \ell^2}\right)^{-\alpha}.
\]

Parameters

- **scale_mixture** *(torch.Tensor)* – Scale mixture (\(\alpha\)) parameter of this kernel. Should have size 1.

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

### 18.2.19 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)*

### 18.2.20 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.

### 18.2.21 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)*
### 18.2.22 Warping

```python
class Warping(kern, iwarping_fn=None, owarping_coef=None):
    Bases: pyro.contrib.gp.kernels.kernel.Transforming

    Creates a new kernel according to
    \[ k_{\text{new}}(x, z) = q(k(f(x), f(z))), \]
    where \( f \) is an function and \( q \) is a polynomial with non-negative coefficients \( \text{owarping_coef} \).

    We can take advantage of \( f \) to combine a Gaussian Process kernel with a deep learning architecture. For example:

    ```python
    >>> 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:

    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** \((X, Z=\text{None}, \text{diag}=\text{False})\)
```

### 18.2.23 WhiteNoise

```python
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 \( \delta \) is a Dirac delta function.

    **forward** \((X, Z=\text{None}, \text{diag}=\text{False})\)
```

### 18.3 Likelihoods

#### 18.3.1 Likelihood

```python
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** \((f_{\text{loc}}, f_{\text{var}}, \text{y=\text{None}})\)
    Samples \( y \) given \( f_{\text{loc}}, f_{\text{var}} \).
```
Parameters

- \( f_{\text{loc}}(\text{torch.Tensor}) \) – Mean of latent function output.
- \( f_{\text{var}}(\text{torch.Tensor}) \) – Variance of latent function output.
- \( y(\text{torch.Tensor}) \) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

18.3.2 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 \textit{Bernoulli} distribution, so the output of \texttt{response_function} should be in range \((0, 1)\). By default, we use \texttt{sigmoid} function.

Parameters

- \texttt{response_function}(callable) – A mapping to correct domain for Binary likelihood.

forward \((f_{\text{loc}}, f_{\text{var}}, y=None)\)

Samples \(y\) given \(f_{\text{loc}}, f_{\text{var}}\) according to

\[
\begin{align*}
  f &\sim N(f_{\text{loc}}, f_{\text{var}}), \\
  y &\sim \text{Bernoulli}(f).
\end{align*}
\]

Note: The log likelihood is estimated using Monte Carlo with 1 sample of \(f\).

Parameters

- \( f_{\text{loc}}(\text{torch.Tensor}) \) – Mean of latent function output.
- \( f_{\text{var}}(\text{torch.Tensor}) \) – Variance of latent function output.
- \( y(\text{torch.Tensor}) \) – Training output tensor.

Returns a tensor sampled from likelihood

Return type torch.Tensor

18.3.3 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 \textit{Normal} distribution.

Parameters

- \texttt{variance}(torch.Tensor) – A variance parameter, which plays the role of \textit{noise} in regression problems.
forward$(f_{loc}, f_{var}, y=None)$

Samples $y$ given $f_{loc}, f_{var}$ according to

\[
y \sim N(f_{loc}, f_{var} + \epsilon),
\]

where $\epsilon$ 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

### 18.3.4 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

\[
f \sim N(f_{loc}, f_{var}),
y \sim \text{Categorical}(f).
\]

**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
18.3.5 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

   \[ f \sim \mathcal{N}(f_{\text{loc}}, f_{\text{var}}), \]
   \[ y \sim \mathcal{P} \sim \exp(f). \]

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

18.4 Parameterized

class Parameterized
   Bases: pyro.nn.module.PyroModule

A wrapper of PyroModule whose parameters can be set constraints, set priors.

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.

Example:

```
>>> class Linear(Parameterized):
...     def __init__(self, a, b):
...         super().__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.a = PyroParam(torch.tensor(1.), constraints.positive)
>>> linear.b = PyroSample(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())

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_prior** (*name, prior*)
Sets prior for a parameter.

Parameters

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

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

### 18.5 Util

**conditional** (*Xnew, X, kernel, f_loc, f_scale_tril=None, Lff=None, full_cov=False, whiten=False, jitter=1e-06*)
Given *Xnew*, 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

\[ q(f \mid f_{loc}, f_{scale\_tril}) \sim p(f \mid X, y), \]
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 \( f = f_{\text{loc}} \) and computes
\[
p(f^*(X_{\text{new}}) | X, k, f).
\]

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


Parameters

- \( \text{Xnew} (\text{torch.Tensor}) \) – A new input data.
- \( \text{X} (\text{torch.Tensor}) \) – An input data to be conditioned on.
- \( \text{kernel} (\text{Kernel}) \) – A Pyro kernel object.
- \( f_{\text{loc}} (\text{torch.Tensor}) \) – Mean of \( q(f) \). In case \( f\_scale\_tril=None, f_{\text{loc}} = f \).
- \( f\_scale\_tril (\text{torch.Tensor}) \) – Lower triangular decomposition of covariance matrix of \( q(f) \)’s.
- \( L_{ff} (\text{torch.Tensor}) \) – Lower triangular decomposition of \( kernel(X, X) \) (optional).
- \( \text{full\_cov} (\text{bool}) \) – A flag to decide if we want to return full covariance matrix or just variance.
- \( \text{whiten} (\text{bool}) \) – A flag to tell if \( f_{\text{loc}} \) and \( f\_scale\_tril \) are already transformed by the inverse of \( L_{ff} \).
- \( \text{jitter} (\text{float}) \) – A small positive term which is added into the diagonal part of a covariance matrix to help stabilize its Cholesky decomposition.

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

Return type tuple(\text{torch.Tensor}, \text{torch.Tensor})

\( \text{train} (\text{gpmodule, optimizer=None, loss_fn=None, retain_graph=None, num_steps=1000}) \)
a helper to optimize parameters for a GP module.

Parameters

- \( \text{gpmodule} (\text{GPM}\text{odel}) \) – A GP module.
- \( \text{optimizer} (\text{Optimizer}) \) – A PyTorch optimizer instance. By default, we use Adam with \( lr=0.01 \).
- \( \text{loss\_fn} (\text{callable}) \) – A loss function which takes inputs are \( \text{gpmodule} \).
  \( \text{model}, \text{gpmodule.guide}, \) and returns ELBO loss. By default, \( \text{loss\_fn=TraceMeanField_ELBO().differentiable_loss} \).
- \( \text{retain\_graph} (\text{bool}) \) – An optional flag of \text{torch.autograd.backward}.
- \( \text{num\_steps} (\text{int}) \) – An number of steps to run SVI.

Returns a list of losses during the training procedure

Return type \text{list}
19.1 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.

```python
class Adam(optim_args)
    Bases: object
    __call__(params)

class JitTrace_ELBO(**kwargs)
    Bases: object
    __call__(model, guide, *args)

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)

Trace_ELBO(**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, constraint=Real(), event_dim=None)

plate(name, size, dim=None)

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

    process_message(msg)

class seed(fn=None, rng_seed=None)
    Bases: pyro.contrib.minipyro.Messenger

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

    get_trace(*args, **kwargs)

    postprocess_message(msg)
Tasks such as choosing the next question to ask in a psychology study, designing an election polling strategy, and deciding which compounds to synthesize and test in biological sciences are all fundamentally asking the same question: how do we design an experiment to maximize the information gathered? Pyro is designed to support automated optimal experiment design: specifying a model and guide is enough to obtain optimal designs for many different kinds of experiment scenarios. Check out our experimental design tutorials that use Pyro to design an adaptive psychology study that uses past data to select the next question, and design an election polling strategy that aims to give the strongest prediction about the eventual winner of the election.

Bayesian optimal experimental design (BOED) is a powerful methodology for tackling experimental design problems and is the framework adopted by Pyro. In the BOED framework, we begin with a Bayesian model with a likelihood $p(y|\theta, d)$ and a prior $p(\theta)$ on the target latent variables. In Pyro, any fully Bayesian model can be used in the BOED framework. The sample sites corresponding to experimental outcomes are the observation sites, those corresponding to latent variables of interest are the target sites. The design $d$ is the argument to the model, and is not a random variable.

In the BOED framework, we choose the design that optimizes the expected information gain (EIG) on the targets $\theta$ from running the experiment

$$\text{EIG}(d) = \mathbb{E}_{p(y|d)}[H[p(\theta)]]H[p(\theta|y, d)]$$

where $H[]$ represents the entropy and $p(\theta|y, d) \propto p(\theta)p(y|\theta, d)$ is the posterior we get from running the experiment with design $d$ and observing $y$. In other words, the optimal design is the one that, in expectation over possible future observations, most reduces posterior entropy over the target latent variables. If the predictive model is correct, this forms a design strategy that is (one-step) optimal from an information-theoretic viewpoint. For further details, see [1, 2].

The `pyro.contrib.oed` module provides tools to create optimal experimental designs for Pyro models. In particular, it provides estimators for the expected information gain (EIG).

To estimate the EIG for a particular design, we first set up our Pyro model. For example:

```python
def model(design):
    # This line allows batching of designs, treating all batch dimensions as independent
```

(continues on next page)
with pyro.plate_stack("plate_stack", design.shape):

    # We use a Normal prior for theta
    theta = pyro.sample("theta", dist.Normal(torch.tensor(0.0), torch.tensor(1.0)))

    # We use a simple logistic regression model for the likelihood
    logit_p = theta - design
    y = pyro.sample("y", dist.Bernoulli(logits=logit_p))

    return y

We then select an appropriate EIG estimator, such as:

eig = nmc_eig(model, design, observation_labels=["y"], target_labels=["theta"], N=2500, M=50)

It is possible to estimate the EIG across a grid of designs:

designs = torch.stack([design1, design2], dim=0)

to find the best design from a number of options.


### 20.1 Expected Information Gain

`laplace_eig` *(model, design, observation_labels, target_labels, guide, loss, optim, num_steps, final_num_samples, y_dist=None, eig=True, **prior_entropy_kwargs)*

Estimates the expected information gain (EIG) by making repeated Laplace approximations to the posterior.

**Parameters**

- `model (function)` – Pyro stochastic function taking `design` as only argument.
- `design (torch.Tensor)` – Tensor of possible designs.
- `observation_labels (list)` – labels of sample sites to be regarded as observables.
- `target_labels (list)` – labels of sample sites to be regarded as latent variables of interest, i.e. the sites that we wish to gain information about.
- `guide (function)` – Pyro stochastic function corresponding to `model`.
- `loss` – a Pyro loss such as `pyro.infer.Trace_ELBO().differentiable_loss`.
- `optim` – optimizer for the loss
- `num_steps (int)` – Number of gradient steps to take per sampled pseudo-observation.
- `final_num_samples (int)` – Number of y samples (pseudo-observations) to take.
- `y_dist` – Distribution to sample y from- if `None` we use the Bayesian marginal distribution.
• **eig** *(bool)* – Whether to compute the EIG or the average posterior entropy (APE). The EIG is given by \( EIG = prior\ entropy - APE \). If True, the prior entropy will be estimated analytically, or by Monte Carlo as appropriate for the model. If False the APE is returned.

• **prior_entropy_kwargs** *(dict)* – parameters for estimating the prior entropy: 
  *num_prior_samples* indicating the number of samples for a MC estimate of prior entropy, and 
  *mean_field* indicating if an analytic form for a mean-field prior should be tried.

**Returns** EIG estimate, optionally includes full optimization history

**Return type** torch.Tensor

vi_eig(*model, design, observation_labels, target_labels, vi_parameters, is_parameters, y_dist=None, eig=True, **prior_entropy_kwargs)*

Deprecated since version 0.4.1: Use **posterior_eig** instead.

Estimates the expected information gain (EIG) using variational inference (VI).

The APE 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 \( Y \)

• **eig** *(bool)* – Whether to compute the EIG or the average posterior entropy (APE). The EIG is given by \( EIG = prior\ entropy - APE \). If True, the prior entropy will be estimated analytically, or by Monte Carlo as appropriate for the model. If False the APE is returned.

• **prior_entropy_kwargs** *(dict)* – parameters for estimating the prior entropy:
  *num_prior_samples* indicating the number of samples for a MC estimate of prior entropy, and 
  *mean_field* indicating if an analytic form for a mean-field prior should be tried.

**Returns** EIG estimate, optionally includes full optimization history

**Return type** torch.Tensor
**nmc_eig** *(model, design, observation_labels=None, N=100, M=10, M_prime=None, independent_priors=False)*

Nested Monte Carlo estimate of the expected information gain (EIG). The estimate is, when there are not any random effects,

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

where \(\theta_n, y_n \sim p(\theta, y|d)\) and \(\theta_m \sim p(\theta)\). The estimate in the presence of random effects is

\[
\frac{1}{N} \sum_{n=1}^{N} \log \left( \frac{1}{M'} \sum_{m=1}^{M'} p(y_n|\theta_n, \tilde{\theta}_nm,d) \right) - \frac{1}{N} \sum_{n=1}^{N} \log \left( \frac{1}{M} \sum_{m=1}^{M} p(y_n|\theta_m, \tilde{\theta}_m,d) \right)
\]

where \(\tilde{\theta}\) are the random effects with \(\tilde{\theta}_{nm} \sim p(\tilde{\theta}|\theta = \theta_n)\) and \(\theta_m, \tilde{\theta}_m \sim p(\theta, \tilde{\theta})\). The latter form is used when \(M_prime \neq None\).

**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.
- **independent_priors** *(bool)* – Only used when `M_prime` is not `None`. Indicates whether the prior distributions for the target variables and the nuisance variables are independent. In this case, it is not necessary to sample the targets conditional on the nuisance variables.

**Returns** EIG estimate, optionally includes full optimization history

**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

\[
\sup_T E_{p(y,\theta|d)} [T(y, \theta)] - \log E_{p(y|d)p(\theta)} [\exp(T(y, \tilde{\theta}))]
\]

where \(T\) is any (measurable) function.

This method 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 optimization steps.


**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 optimization.

**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 optimization history

**Return type** torch.Tensor or tuple

---

**posterior_eig** (*model, design, observation_labels, target_labels, num_samples, num_steps, guide, optim, return_history=False, final_design=None, final_num_samples=None, eig=True, prior_entropy_kwargs={}, *args, **kwargs*)

Posterior estimate of expected information gain (EIG) computed from the average posterior entropy (APE) using \( EIG(d) = H[p(\theta)] - APE(d) \). See [1] for full details.

The posterior representation of APE is

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

where \( q \) is any distribution on \( \theta \).

This method optimises the loss over a given *guide* family 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 optimization steps.
- **guide** (*function*) – guide family for use in the (implicit) posterior estimation. The parameters of *guide* are optimised to maximise the posterior 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 optimization.

• `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`.

• `eig (bool)` – Whether to compute the EIG or the average posterior entropy (APE). The EIG is given by $EIG = prior \text{ entropy} - APE$. If True, the prior entropy will be estimated analytically, or by Monte Carlo as appropriate for the model. If False the APE is returned.

• `prior_entropy_kwargs (dict)` – parameters for estimating the prior entropy: `num_prior_samples` indicating the number of samples for a MC estimate of prior entropy, and `mean_field` indicating if an analytic form for a mean-field prior should be tried.

Returns EIG estimate, optionally includes full optimization history

Return type `torch.Tensor` or tuple

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

Estimate EIG by estimating the marginal entropy $p(y|d)$. See [1] for full details.

The marginal representation of EIG is

$$\inf_q E_p(y, \theta|d) \left[ \log \frac{p(y|\theta, d)}{q(y|d)} \right]$$

where $q$ is any distribution on $y$. A variational family for $q$ is specified in the `guide`.

Warning: This method does not estimate the correct quantity in the presence of random effects.


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 optimization steps.

• `guide (function)` – guide family for use in the marginal estimation. The parameters of `guide` are optimised to maximise the log-likelihood 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 optimization.

• `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 optimization history

**Return type**  torch.Tensor or tuple

**lfire_eig**(*model*, *design*, *observation_labels*, *target_labels*, *num_y_samples*, *num_theta_samples*, *num_steps*, *classifier*, *optim*, *return_history=False*, *final_design=None*, *final_num_samples=None*)

Estimates the EIG using the method of Likelihood-Free Inference by Ratio Estimation (LFIRE) as in [1]. LFIRE is run separately for several samples of $\theta$.


**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_y_samples** (*int*) – Number of samples to take in $y$ for each $\theta$.

• **num_steps** (*int*) – Number of optimization steps.

• **classifier** (*function*) – a Pytorch or Pyro classifier used to distinguish between samples of $y$ under $p(y|d)$ and samples under $p(y|\theta, d)$ for some $\theta$.

• **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 optimization.

• **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'.

**Param** int num_theta_samples: Number of initial samples in $\theta$ to take. The likelihood ratio is estimated by LFIRE for each sample.

**Returns**  EIG estimate, optionally includes full optimization history

**Return type**  torch.Tensor or tuple

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

Estimates the EIG using Variational Nested Monte Carlo (VNMC). The VNMC estimate [1] is

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

where $q(\theta|y)$ is the learned variational posterior approximation and $\theta_n, y_n \sim p(\theta, y|d), \theta_{mn} \sim q(\theta|y = y_n)$. As $N \to \infty$ this is an upper bound on EIG. We minimise this upper bound by stochastic gradient descent.
Warning: This method cannot be used in the presence of random effects.


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** (*tuple*) – Number of \((N, M)\) samples per iteration.
- **num_steps** (*int*) – Number of optimization steps.
- **guide** (*function*) – guide family for use in the posterior estimation. The parameters of `guide` are optimised to minimise the VNMC upper bound.
- **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 optimization.
- **final_design** (*torch.Tensor*) – The final design tensor to evaluate at. If None, uses `design`.
- **final_num_samples** (*tuple*) – The number of \((N, M)\) samples to use at the final evaluation, If None, uses `num_samples`.

Returns EIG estimate, optionally includes full optimization history

Return type torch.Tensor or tuple

### 20.2 Generalised Linear Mixed Models

Warning: This module will eventually be deprecated in favor of brmp

The *pyro.contrib.oed.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:

```python
from pyro.contrib.oed.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.))
```

(continues on next page)
A non-linear link function may be introduced, for instance:

```python
from pyro.contrib.oed.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.oed.glmm.lmer_model()`.
21.1 Random Variable

class RandomVariable(distribution)
    Bases: pyro.contrib.randomvariable.random_variable.RVMagicOps, pyro.contrib.
randomvariable.random_variable.RVChainOps

EXPERIMENTAL random variable container class around a distribution

Representation of a distribution interpreted as a random variable. Rather than directly manipulating a probability
density by applying pointwise transformations to it, this allows for simple arithmetic transformations of the
random variable the distribution represents. For more flexibility, consider using the transform method. Note
that if you perform a non-invertible transform (like abs(X) or X**2), certain things might not work properly.

Can switch between RandomVariable and Distribution objects with the convenient Distribution.rv and Random-
Variable.dist properties.

Supports either chaining operations or arithmetic operator overloading.

Example usage:

# This should be equivalent to an Exponential distribution.
RandomVariable(Uniform(0, 1)).log().neg().dist

# These two distributions Y1, Y2 should be the same
X = Uniform(0, 1).rv
Y1 = X.mul(4).pow(0.5).sub(1).abs().neg().dist
Y2 = {-abs((4*X)**(0.5) - 1)}.dist


dist
    Convenience property for exposing the distribution underlying the random variable.

    Returns The Distribution object underlying the random variable

    Return type Distribution
transform ($t$: torch.distributions.transforms.Transform)
Performs a transformation on the distribution underlying the RV.

**Parameters** $t$ (*Transform*) – The transformation (or sequence of transformations) to be applied to the distribution. There are many examples to be found in torch.distributions.transforms and pyro.distributions.transforms, or you can subclass directly from Transform.

**Returns** The transformed RandomVariable

**Return type** RandomVariable
The `pyro.contrib.timeseries` module provides a collection of Bayesian time series models useful for forecasting applications.

See the GP example for example usage.

### 22.1 Abstract Models

```python
class TimeSeriesModel(name=")
    Bases: pyro.nn.module.PyroModule
    Base class for univariate and multivariate time series models.

    log_prob(targets)
        Log probability function.

        Parameters
        ----------
        targets : torch.Tensor
            A 2-dimensional tensor of real-valued targets of shape (T, obs_dim), where T is the length of the time series and obs_dim is the dimension of the real-valued targets at each time step

        Returns
        -------
        torch.Tensor
            A 0-dimensional log probability for the case of properly multivariate time series models in which the output dimensions are correlated; otherwise returns a 1-dimensional tensor of log probabilities for batched univariate time series models.

    forecast(targets, dts)
        Parameters
        ----------
        targets : torch.Tensor
            A 2-dimensional tensor of real-valued targets of shape (T, obs_dim), where T is the length of the time series and obs_dim is the dimension of the real-valued targets at each time step. These represent the training data that are conditioned on for the purpose of making forecasts.

        dts : torch.Tensor
            A 1-dimensional tensor of times to forecast into the future, with zero corresponding to the time of the final target targets[-1].
```
**Returns torch.distributions.Distribution**

Returns a predictive distribution with batch shape $(S,)$ and event shape $(\text{obs\_dim},)$, where $S$ is the size of $\text{dts}$. That is, the resulting predictive distributions do not encode correlations between distinct times in $\text{dts}$.

**get_dist()**

Get a `Distribution` object corresponding to this time series model. Often this is a `GaussianHMM`.

### 22.2 Gaussian Processes

**class IndependentMaternGP**

```python
class IndependentMaternGP(nu=1.5, dt=1.0, obs_dim=1, length_scale_init=None, kernel_scale_init=None, obs_noise_scale_init=None):
    Bases: pyro.contrib.timeseries.base.TimeSeriesModel
```

A time series model in which each output dimension is modeled independently with a univariate Gaussian Process with a Matern kernel. The targets are assumed to be evenly spaced in time. Training and inference are logarithmic in the length of the time series $T$.

**Parameters**

- **nu** (*float*) – The order of the Matern kernel; one of 0.5, 1.5 or 2.5.
- **dt** (*float*) – The time spacing between neighboring observations of the time series.
- **obs_dim** (*int*) – The dimension of the targets at each time step.
- **length_scale_init** (*torch.Tensor*) – optional initial values for the kernel length scale given as a `obs_dim`-dimensional tensor
- **kernel_scale_init** (*torch.Tensor*) – optional initial values for the kernel scale given as a `obs_dim`-dimensional tensor
- **obs_noise_scale_init** (*torch.Tensor*) – optional initial values for the observation noise scale given as a `obs_dim`-dimensional tensor

**get_dist(duration=None)**

Get the `GaussianHMM` distribution that corresponds to $\text{obs\_dim}$-many independent Matern GPs.

**Parameters**

- **duration** (*int*) – Optional size of the time axis $\text{event\_shape}[0]$. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

**log_prob(targets)**

**Parameters**

- **targets** (*torch.Tensor*) – A 2-dimensional tensor of real-valued targets of shape $(T, \text{obs\_dim})$, where $T$ is the length of the time series and $\text{obs\_dim}$ is the dimension of the real-valued targets at each time step

**Returns**

`torch.Tensor` A 1-dimensional tensor of log probabilities of shape $(\text{obs\_dim},)$

**forecast(targets, dts)**

**Parameters**

- **targets** (*torch.Tensor*) – A 2-dimensional tensor of real-valued targets of shape $(T, \text{obs\_dim})$, where $T$ is the length of the time series and $\text{obs\_dim}$ is the dimension of the real-valued targets at each time step. These represent the training data that are conditioned on for the purpose of making forecasts.
- **dts** (*torch.Tensor*) – A 1-dimensional tensor of times to forecast into the future, with zero corresponding to the time of the final target $\text{targets}[-1]$. 

---

**Chapter 22. Time Series**
Returns `torch.distributions.Normal` Returns a predictive Normal distribution with batch shape \((S,)\) and event shape \((\text{obs\_dim},)\), where \(S\) is the size of \(\text{dts}\).

class LinearlyCoupledMaternGP

\[
\nu=1.5, \quad dt=1.0, \quad obs\_dim=2, \quad num\_gps=1,
\]
\[
\text{length\_scale\_init}=\text{None}, \quad \text{kernel\_scale\_init}=\text{None}, \quad \text{obs\_noise\_scale\_init}=\text{None}
\]

Bases: `pyro.contrib.timeseries.base.TimeSeriesModel`

A time series model in which each output dimension is modeled as a linear combination of shared univariate Gaussian Processes with Matern kernels.

In more detail, the generative process is:

\[
y_i(t) = \sum_j A_{ij} f_j(t) + \epsilon_i(t)
\]

The targets \(y_i\) are assumed to be evenly spaced in time. Training and inference are logarithmic in the length of the time series \(T\).

Parameters

- **nu** (*float*) – The order of the Matern kernel; one of 0.5, 1.5 or 2.5.
- **dt** (*float*) – The time spacing between neighboring observations of the time series.
- **obs_dim** (*int*) – The dimension of the targets at each time step.
- **num_gps** (*int*) – The number of independent GPs that are mixed to model the time series. Typical values might be \(\text{gp} \in \lfloor \text{obs}/2, \text{obs} \rfloor\)
- **length_scale_init** (*torch.Tensor*) – optional initial values for the kernel length scale given as a \(\text{num\_gps}\)-dimensional tensor
- **kernel_scale_init** (*torch.Tensor*) – optional initial values for the kernel scale given as a \(\text{num\_gps}\)-dimensional tensor
- **obs_noise_scale_init** (*torch.Tensor*) – optional initial values for the observation noise scale given as a \(\text{obs\_dim}\)-dimensional tensor

get_dist \((\text{duration}=\text{None})\)

Get the `GaussianHMM` distribution that corresponds to a `LinearlyCoupledMaternGP`.

Parameters **duration** (*int*) – Optional size of the time axis \(\text{event\_shape}[0]\). This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

log_prob \((\text{targets})\)

Parameters **targets** (*torch.Tensor*) – A 2-dimensional tensor of real-valued targets of shape \((T, \text{obs\_dim})\), where \(T\) is the length of the time series and \(\text{obs\_dim}\) is the dimension of the real-valued targets at each time step

Returns `torch.Tensor` a (scalar) log probability

forecast \((\text{targets}, \text{dts})\)

Parameters

- **targets** (*torch.Tensor*) – A 2-dimensional tensor of real-valued targets of shape \((T, \text{obs\_dim})\), where \(T\) is the length of the time series and \(\text{obs\_dim}\) is the dimension of the real-valued targets at each time step. These represent the training data that are conditioned on for the purpose of making forecasts.
- **dts** (*torch.Tensor*) – A 1-dimensional tensor of times to forecast into the future, with zero corresponding to the time of the final target \(\text{targets}[\text{-1}]\).
Returns `torch.distributions.MultivariateNormal` Returns a predictive MultivariateNormal distribution with batch shape \((S,)\) and event shape \((\text{obs\_dim},)\), where \(S\) is the size of \(\text{dts}\).

class DependentMaternGP  

\(\nu=1.5, \quad dt=1.0, \quad \text{obs\_dim}=1, \quad \text{linearly\_coupled}=\text{False},\)  
\(\text{length\_scale\_init}=\text{None}, \quad \text{obs\_noise\_scale\_init}=\text{None}\)  
Bases: `pyro.contrib.timeseries.base.TimeSeriesModel`

A time series model in which each output dimension is modeled as a univariate Gaussian Process with a Matern kernel. The different output dimensions become correlated because the Gaussian Processes are driven by a correlated Wiener process; see reference [1] for details. If, in addition, \(\text{linearly\_coupled}\) is True, additional correlation is achieved through linear mixing as in `LinearlyCoupledMaternGP`. The targets are assumed to be evenly spaced in time. Training and inference are logarithmic in the length of the time series \(T\).

Parameters
- **nu** *(float)* – The order of the Matern kernel; must be 1.5.
- **dt** *(float)* – The time spacing between neighboring observations of the time series.
- **obs\_dim** *(int)* – The dimension of the targets at each time step.
- **linearly\_coupled** *(bool)* – Whether to linearly mix the various gaussian processes in the likelihood. Defaults to False.
- **length\_scale\_init** *(torch.Tensor)* – optional initial values for the kernel length scale given as a \(\text{obs\_dim}\)-dimensional tensor
- **obs\_noise\_scale\_init** *(torch.Tensor)* – optional initial values for the observation noise scale given as a \(\text{obs\_dim}\)-dimensional tensor


get_dist *(duration=None)*

Get the `GaussianHMM` distribution that corresponds to a `DependentMaternGP`

log_prob *(targets)*

Parameters **targets** *(torch.Tensor)* – A 2-dimensional tensor of real-valued targets of shape \((T, \text{obs\_dim})\), where \(T\) is the length of the time series and \(\text{obs\_dim}\) is the dimension of the real-valued targets at each time step

Returns `torch.Tensor` A (scalar) log probability

forecast *(targets, dts)*

Parameters
- **targets** *(torch.Tensor)* – A 2-dimensional tensor of real-valued targets of shape \((T, \text{obs\_dim})\), where \(T\) is the length of the time series and \(\text{obs\_dim}\) is the dimension of the real-valued targets at each time step. These represent the training data that are conditioned on for the purpose of making forecasts.
- **dts** *(torch.Tensor)* – A 1-dimensional tensor of times to forecast into the future, with zero corresponding to the time of the final target \(\text{targets}[-1]\).

Returns `torch.distributions.MultivariateNormal` Returns a predictive MultivariateNormal distribution with batch shape \((S,)\) and event shape \((\text{obs\_dim},)\), where \(S\) is the size of \(\text{dts}\).
22.3 Linear Gaussian State Space Models

```python
class GenericLGSSM(obs_dim=1, state_dim=2, obs_noise_scale_init=None, learnable_observation_loc=False)
Bases: pyro.contrib.timeseries.base.TimeSeriesModel

A generic Linear Gaussian State Space Model parameterized with arbitrary time invariant transition and observation dynamics. The targets are (implicitly) assumed to be evenly spaced in time. Training and inference are logarithmic in the length of the time series T.

Parameters

- **obs_dim (int)** – The dimension of the targets at each time step.
- **state_dim (int)** – The dimension of latent state at each time step.
- **learnable_observation_loc (bool)** – whether the mean of the observation model should be learned or not; defaults to False.

`get_dist(duration=None)`
Get the `GaussianHMM` distribution that corresponds to `GenericLGSSM`.

Parameters

- **duration (int)** – Optional size of the time axis `event_shape[0]`. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

`log_prob(targets)`

Parameters

- **targets (torch.Tensor)** – A 2-dimensional tensor of real-valued targets of shape `(T, obs_dim)`, where T is the length of the time series and `obs_dim` is the dimension of the real-valued targets at each time step.

Returns `torch.Tensor` A (scalar) log probability.

`forecast(targets, N_timesteps)`

Parameters

- **targets (torch.Tensor)** – A 2-dimensional tensor of real-valued targets of shape `(T, obs_dim)`, where T is the length of the time series and `obs_dim` is the dimension of the real-valued targets at each time step. These represent the training data that are conditioned on for the purpose of making forecasts.
- **N_timesteps (int)** – The number of timesteps to forecast into the future from the final target `targets[-1]`.

Returns `torch.distributions.MultivariateNormal` Returns a predictive `MultivariateNormal` distribution with batch shape `(N_timesteps,)` and event shape `(obs_dim,)`

```Python

class GenericLGSSMWithGPNoiseModel(obs_dim=1, state_dim=2, nu=1.5, obs_noise_scale_init=None, length_scale_init=None, kernel_scale_init=None, learnable_observation_loc=False)
Bases: pyro.contrib.timeseries.base.TimeSeriesModel

A generic Linear Gaussian State Space Model parameterized with arbitrary time invariant transition and observation dynamics together with separate Gaussian Process noise models for each output dimension. In more detail, the generative process is:

\[ y_i(t) = \sum_j A_{ij} z_j(t) + f_i(t) + \epsilon_i(t) \]

where the latent variables \( z(t) \) follow generic time invariant Linear Gaussian dynamics and the \( f_i(t) \) are Gaussian Processes with Matern kernels.
```
The targets are (implicitly) assumed to be evenly spaced in time. In particular a timestep of $dt = 1.0$ for the continuous-time GP dynamics corresponds to a single discrete step of the z-space dynamics. Training and inference are logarithmic in the length of the time series $T$.

**Parameters**

- **obs_dim** *(int)* – The dimension of the targets at each time step.
- **state_dim** *(int)* – The dimension of the z latent state at each time step.
- **nu** *(float)* – The order of the Matern kernel; one of 0.5, 1.5 or 2.5.
- **length_scale_init** *(torch.Tensor)* – optional initial values for the kernel length scale given as a `obs_dim`-dimensional tensor
- **kernel_scale_init** *(torch.Tensor)* – optional initial values for the kernel scale given as a `obs_dim`-dimensional tensor
- **obs_noise_scale_init** *(torch.Tensor)* – optional initial values for the observation noise scale given as a `obs_dim`-dimensional tensor
- **learnable_observation_loc** *(bool)* – whether the mean of the observation model should be learned or not; defaults to `False`.

**get_dist** *(duration=None)*

Get the GaussianHMM distribution that corresponds to `GenericLGSSMWithGPNoiseModel`.

**Parameters**

- **duration** *(int)* – Optional size of the time axis event_shape[0]. This is required when sampling from homogeneous HMMs whose parameters are not expanded along the time axis.

**log_prob** *(targets)*

**Parameters**

- **targets** *(torch.Tensor)* – A 2-dimensional tensor of real-valued targets of shape `(T, obs_dim)`, where $T$ is the length of the time series and `obs_dim` is the dimension of the real-valued targets at each time step

**Returns** `torch.Tensor` A (scalar) log probability.

**forecast** *(targets, N_timesteps)*

**Parameters**

- **targets** *(torch.Tensor)* – A 2-dimensional tensor of real-valued targets of shape `(T, obs_dim)`, where $T$ is the length of the time series and `obs_dim` is the dimension of the real-valued targets at each time step. These represent the training data that are conditioned on for the purpose of making forecasts.
- **N_timesteps** *(int)* – The number of timesteps to forecast into the future from the final target `targets[-1]`.

**Returns** `torch.distributions.MultivariateNormal` Returns a predictive MultivariateNormal distribution with batch shape `(N_timesteps,)` and event shape `(obs_dim,)`
CHAPTER 23

Tracking

23.1 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

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

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 probabilities 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 probabilities 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.


`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.


`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.


23.2 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)

23.3 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** \((x_1, x_0)\)

Compute and return the geodesic difference between 2 native states. This is a generalization of the Euclidean operation \(x_1 - x_0\).

**Parameters**

• **x1** – native state.

• **x0** – native state.

**Returns** Geodesic difference between native states \(x_1\) and \(x_2\).

**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** \((\text{dimension}, \text{dimension\_pv}, \text{num\_process\_noise\_parameters}=\text{None})\)

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** \((\text{dimension}, sv2)\)

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.

• \texttt{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.

\(\text{mean2pv}(x)\)
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.

\(\text{cov2pv}(P)\)
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.

\(\text{jacobian}(dt)\)
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 (\(f\)).

\(\text{process_noise_cov}(dt=0.0)\)
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 \texttt{NcpContinuous}(\texttt{dimension}, sv2)
NCP (Nearly-Constant Position) dynamic model with CWNV (Continuous White Noise Velocity).


Parameters

• \texttt{dimension} – native state dimension.

• \texttt{sv2} – variance of velocity. Usually chosen so that the standard deviation is roughly half of the max velocity one would ever expect to observe.

\(\text{process_noise_cov}(dt=0.0)\)
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 \texttt{NcvContinuous}(\texttt{dimension}, sa2)
NCV (Nearly-Constant Velocity) dynamic model with CWNA (Continuous 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.

**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 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:  

**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 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:  

**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 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.)

23.4 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
\( \text{Chi}^2 \) 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.

### 23.5 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:**

```python
>>> 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 to identify point.
- **point (torch.Tensor)** – Data, should be detached and on cpu.

**remove (key)**
Removes key and corresponding point from the 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 the difference of each element of their hashes is smaller than 2. In euclidean space, this corresponds to all points p where |p_k - (p_key)_k| < r, and some points (all points not guaranteed) where |p_k - (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)
Greedyly 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
23.6 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 \( z_1 - z_0 \).

    Parameters
    - z1 – measurement.
    - z0 – measurement.

    Returns Geodesic difference between \( z_1 \) and \( z_2 \).

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.

jacobian (x=None)
    Compute and return Jacobian (H) of measurement map (h) at target PV state \( x \).
Parameters \( \mathbf{x} \) – 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 24

Indices and tables

- genindex
- search
Python Module Index

p
pyro.contrib.autoname, 195
pyro.contrib.autoname.named, 197
pyro.contrib.bnn, 203
pyro.contrib.bnn.hidden_layer, 203
pyro.contrib.easyguide, 211
pyro.contrib.epidemiology, 215
pyro.contrib.epidemiology.compartmental, 215
pyro.contrib.epidemiology.distributions, 227
pyro.contrib.epidemiology.models, 220
pyro.contrib.examples.bart, 231
pyro.contrib.examples.multi_mnist, 231
pyro.contrib.examples.util, 232
pyro.contrib.forecast, 233
pyro.contrib.forecast.evaluate, 237
pyro.contrib.forecast.forecaster, 233
pyro.contrib.funsor, 239
pyro.contrib.funsor.handlers, 239
pyro.contrib.funsor.handlers.named_messenger, 240
pyro.contrib.funsor.handlers.primitives, 240
pyro.contrib.funsor.handlers.runtime, 240
pyro.contrib.gp, 243
pyro.contrib.gp.kernels, 255
pyro.contrib.gp.likelihoods, 261
pyro.contrib.gp.models.gplvm, 254
pyro.contrib.gp.models.gpr, 247
pyro.contrib.gp.models.model, 244
pyro.contrib.gp.models.sgpr, 249
pyro.contrib.gp.models.vgp, 251
pyro.contrib.gp.models.vsgp, 252
pyro.contrib.gp.parameterized, 264
pyro.contrib.gp.util, 265
pyro.contrib.minipyro, 267
pyro.contrib.oed, 269
pyro.contrib.oed.eig, 270
pyro.contrib.oed.glmm, 276
pyro.contrib.randomvariable, 279
pyro.contrib.timeseries, 281
pyro.contrib.timeseries.base, 281
pyro.contrib.timeseries.gp, 282
pyro.contrib.timeseries.lgssm, 285
pyro.contrib.timeseries.lgssmgp, 285
pyro.contrib.tracking, 287
pyro.contrib.tracking.assignment, 287
pyro.contrib.tracking.distributions, 290
pyro.contrib.tracking.dynamic_models, 290
pyro.contrib.tracking.extended kalman filter, 294
pyro.contrib.tracking.hashing, 296
pyro.contrib.tracking.measurements, 298
pyro.distributions.constraints, 129
pyro.distributions.torch, 53
pyro.infer.abstract_infer, 26
pyro.infer.autoguide, 36
pyro.infer.autoguide.initialization, 44
pyro.infer.discrete, 23
pyro.infer.elbo, 10
pyro.infer.energy_distance, 23
pyro.infer.importance, 17
pyro.infer.predictive, 25
pyro.infer.renyi_elbo, 15
pyro.infer.reparam, 45
pyro.infer.reparam.conjugate, 46
pyro.infer.reparam.discrete_cosine, 47
pyro.infer.reparam.haar, 47
pyro.infer.reparam.hmm, 49
pyro.infer.reparam.loc_scale, 46
pyro.infer.reparam.neutra, 50
pyro.infer.reparam.reparam, 45
pyro.infer.reparam.split, 50
pyro.infer.reparam.stable, 48
pyro.infer.reparam.studentt, 48
pyro.infer.reparam.transform, 47
pyro.infer.reparam.unit_jacobian, 48
pyro.infer.rws, 18
pyro.infer.smcfilter, 20
pyro.infer.svgd, 21
pyro.infer.svi, 9
pyro.infer.trace_elbo, 11
pyro.infer.trace_mean_field_elbo, 14
pyro.infer.trace_tail_adaptive_elbo, 15
pyro.infer.traceenum_elbo, 12
pyro.infer.tracegraph_elbo, 12
pyro.infer.tracetmc_elbo, 16
pyro.nn.module, 137
pyro.ops.dual_averaging, 177
pyro.ops.einsum, 186
pyro.ops.gaussian, 188
pyro.ops.indexing, 184
pyro.ops.integrator, 178
pyro.ops.newton, 179
pyro.ops.special, 181
pyro.ops.ssm_gp, 194
pyro.ops.stats, 191
pyro.ops.tensor_utils, 182
pyro.ops.welford, 178
pyro.optim.adagrad_rmsprop, 148
pyro.optim.clipped_adam, 149
pyro.optim.horovod, 149
pyro.optim.lr_scheduler, 148
pyro.optim.multi, 151
pyro.optim.optim, 147
pyro.optim.pytorch_optimizers, 150
pyro.params.param_store, 133
pyro.poutine.block_messenger, 167
pyro.poutine.broadcast_messenger, 168
pyro.poutine.collapse_messenger, 168
pyro.poutine.condition_messenger, 168
pyro.poutine.do_messenger, 169
pyro.poutine.enum_messenger, 169
pyro.poutine.escape_messenger, 170
pyro.poutine.handlers, 153
pyro.poutine.indep_messenger, 170
pyro.poutine.infer_config_messenger, 170
pyro.poutine.lift_messenger, 170
pyro.poutine.markov_messenger, 171
pyro.poutine.mask_messenger, 172
pyro.poutine.messenger, 166
pyro.poutine.plate_messenger, 172
pyro.poutine.reentrant_messenger, 172
pyro.poutine.reparam_messenger, 173
pyro.poutine.replay_messenger, 173
pyro.poutine.runtime, 163
pyro.poutine.scale_messenger, 174
pyro.poutine.seed_messenger, 174
pyro.poutine.subsample_messenger, 174
pyro.poutine.trace_messenger, 175
pyro.poutine.uncondition_messenger, 176
pyro.poutine.util, 164
pyro.primitives, 3

304 Python Module Index
arg_constraints (Delta attribute), 66
arg_constraints (DirichletMultinomial attribute), 67
arg_constraints (DiscreteHMM attribute), 68
arg_constraints (Empirical attribute), 69
arg_constraints (ExtendedBetaBinomial attribute), 69
arg_constraints (ExtendedBinomial attribute), 70
arg_constraints (GammaGaussianHMM attribute), 71
arg_constraints (GammaPoisson attribute), 72
arg_constraints (GaussianHMM attribute), 73
arg_constraints (GaussianMRF attribute), 75
arg_constraints (GaussianScaleMixture attribute), 76
arg_constraints (ImproperUniform attribute), 76
arg_constraints (IndependentHMM attribute), 77
arg_constraints (InverseGamma attribute), 77
arg_constraints (LinearHMM attribute), 78
arg_constraints (LKJCovCholesky attribute), 79
arg_constraints (MaskedDistribution attribute), 79
arg_constraints (MaskedMixture attribute), 80
arg_constraints (MixtureOfDiagNormals attribute), 81
arg_constraints (MixtureOfDiagNormalsSharedCovariance attribute), 82
arg_constraints (MultivariateStudentT attribute), 82
arg_constraints (OMTMultivariateNormal attribute), 83
arg_constraints (OrderedLogistic attribute), 83
arg_constraints (Rejection attribute), 85
arg_constraints (SpanningTree attribute), 86
arg_constraints (Stable attribute), 87
arg_constraints (TruncatedPolyaGamma attribute), 87
arg_constraints (Unit attribute), 88
arg_constraints (VonMises3D attribute), 88
arg_constraints (ZeroInflatedDistribution attribute), 89
arg_constraints (ZeroInflatedNegativeBinomial attribute), 89
arg_constraints (ZeroInflatedPoisson attribute), 90
ASGD () (in module pyro.optim.pytorch_optimizers), 90
AutoCallable (class in pyro.infer.autoguide), 90
AutoContinuous (class in pyro.infer.autoguide), 90
autocovariance() (in module pyro.ops.stats), 191
autocovariance() (in module pyro.ops.stats), 191
AutoDelta (class in pyro.infer.autoguide), 91
AutoDiagonalNormal (class in pyro.infer.autoguide), 91
AutoDiscreteParallel (class in pyro.infer.autoguide), 91
AutoGuide (class in pyro.infer.autoguide), 92
autoguide() (Parameterized method), 244, 265
AutoGuideList (class in pyro.infer.autoguide), 93
AutoLaplaceApproximation (class in pyro.infer.autoguide), 93
AutoLowRankMultivariateNormal (class in pyro.infer.autoguide), 93
AutoMultivariateNormal (class in pyro.infer.autoguide), 94
AutoNormal (class in pyro.infer.autoguide), 95
AutoNormalizingFlow (class in pyro.infer.autoguide), 96
autoregressive (AffineAutoregressive attribute), 97
autoregressive (BlockAutoregressive attribute), 98
autoregressive (GaussianAutoregressive attribute), 112
autoregressive (Polynomial attribute), 115
autoregressive (SplineAutoregressive attribute), 118
AutoRegressiveNN (class in pyro.nn.auto_reg.nn), 121
AVFMultivariateNormal (class in pyro.distributions), 122

B
backtest() (in module pyro.contrib.forecast.evaluate), 237
BackwardSampleMessenger (class in pyro.infer.traceenum_elbo), 12
bandwidth_factor (IMQSteinKernel attribute), 21
bandwidth_factor (RBFSteinKernel attribute), 21
batch() (in module pyro.primitives), 7
batch_shape (AffineNormal attribute), 190
batch_shape (Gaussian attribute), 192
BatchNorm (class in pyro.distributions.transforms), 96
batchnorm() (in module pyro.distributions.transforms), 122
Bernoulli (class in pyro.distributions), 53
BernoulliNet (class in pyro.contrib.cevae), 209
Beta (class in pyro.distributions), 53
beta_binomial_dist() (in module pyro.contrib.epidemiology.distributions), 228
BetaBinomial (class in pyro.distributions), 63
bijective (AffineAutoregressive attribute), 95
bijective (AffineCoupling attribute), 96
bijective (BatchNorm attribute), 97
bijective (BlockAutoregressive attribute), 98
bijective (ConditionalAffineAutoregressive attribute), 99
bijective (ConditionalAffineCoupling attribute), 101
bijective (ConditionalGeneralizedChannelPermute attribute), 102
bijective (ConditionalHouseholder attribute), 103
Index 307

bijective (ConditionalMatrixExponential attribute), 104
bijective (ConditionalNeuralAutoregressive attribute), 105
bijective (ConditionalPlanar attribute), 106
bijective (ConditionalRadial attribute), 107
bijective (ConditionalSpline attribute), 108
bijective (ConditionalSplineAutoregressive attribute), 109
bijective (CorrLCholeskyTransform attribute), 90
bijective (DiscreteCosineTransform attribute), 93
bijective (ELUTransform attribute), 91
bijective (GeneralizedChannelPermute attribute), 110
bijective (HaarTransform attribute), 91
bijective (Householder attribute), 111
bijective (LeakyReLUTransform attribute), 91
bijective (LowerCholeskyAffine attribute), 92
bijective (MatrixExponential attribute), 112
bijective (NeuralAutoregressive attribute), 113
bijective (OrderedTransform attribute), 92
bijective (Permute attribute), 93
bijective (Planar attribute), 114
bijective (Polynomial attribute), 115
bijective (Radial attribute), 116
bijective (Spline attribute), 117
bijective (SplineAutoregressive attribute), 118
bijective (SplineCoupling attribute), 119
bijective (Sylvester attribute), 120
Binary (class in pyro.contrib.gp_likelihoods), 262
Binomial (class in pyro.distributions), 53
binomial_dist() (in module pyro.contrib.epidemiology.distributions), 227
bio_phylo_to_times() (in module pyro.distributions.coalescent), 230
block (class in pyro.distributions.coalescent), 268
block() (in module pyro.poutine.handlers), 154
block_autoregressive() (in module pyro.distributions.transforms), 122
block_diag_embed() (in module pyro.ops.tensor_utils), 182
block_diagonal() (in module pyro.ops.tensor_utils), 182
block_messengers() (in module pyro.poutine.messenger), 166
block_plate() (in module pyro.poutine.messenger), 172
BlockAutoregressive (class in pyro.distributions.transforms), 97
BlockMassMatrix (class in pyro.infer.mcmc), 33
BlockMessenger (class in pyro.poutine.block_messenger), 167
broadcast() (in module pyro.poutine.handlers), 155
broadcastMessenger (class in pyro.poutine.broadcast_messenger), 168
Brownian (class in pyro.contrib.gp.kernels), 255

C
call() (AutoGuide method), 36
call() (Predictive method), 25
cat() (Gaussian static method), 189
Categorical (class in pyro.distributions), 53
Cauchy (class in pyro.distributions), 54
CEVAE (class in pyro.contrib.cevae), 205
Chi2 (class in pyro.distributions), 54
cholesky () (in module pyro.ops.tensor_utils), 184
cholesky_solve() (in module pyro.ops.tensor_utils), 184
cleanup() (HMC method), 31
cleanup() (MCMC Kernel method), 29
clear() (in module pyro.nn.module), 141
clear() (ParamStoreDict method), 133
clear_cache() (ConditionalTransformedDistribution method), 66
clear_cache() (HMC method), 31
clear_param_store() (in module pyro.primitives), 3
ClippedAdam (class in pyro.optim.clipped_adam), 149
ClippedAdam() (in module pyro.optim.optim), 148
CoalescentRateLikelihood (class in pyro.distributions), 229
CoalescentTimes (class in pyro.distributions), 64
CoalescentTimesWithRate (class in pyro.distributions), 65
codomain (AffineAutoregressive attribute), 95
codomain (AffineCoupling attribute), 96
codomain (BatchNorm attribute), 97
codomain (BlockAutoregressive attribute), 98
codomain (ConditionalAffineAutoregressive attribute), 99
codomain (ConditionalAffineCoupling attribute), 101
codomain (ConditionalGeneralizedChannelPermute attribute), 102
codomain (ConditionalHouseholder attribute), 103
codomain (ConditionalMatrixExponential attribute), 104
codomain (ConditionalNeuralAutoregressive attribute), 105
codomain (ConditionalPlanar attribute), 106
codomain (ConditionalRadial attribute), 107
codomain (ConditionalSpline attribute), 108
codomain (ConditionalSplineAutoregressive attribute), 109
codomain (CorrLCholeskyTransform attribute), 90
codomain (DiscreteCosineTransform attribute), 93
codomain (ELUTransform attribute), 91
codomain (GeneralizedChannelPermute attribute), 110
codomain (HaarTransform attribute), 91
codomain (Householder attribute), 111
codomain (LeakyReLUTransform attribute), 91
codomain (LowerCholeskyAffine attribute), 92
codomain (MatrixExponential attribute), 112
codomain (NeuralAutoregressive attribute), 113
codomain (OrderedTransform attribute), 92
codomain (Permute attribute), 93
codomain (Planar attribute), 114
codomain (Polynomial attribute), 115
codomain (Radial attribute), 116
codomain (Spline attribute), 117
codomain (SplineAutoregressive attribute), 118
codomain (SplineCoupling attribute), 119
codomain (Sylvester attribute), 120
collapse() (in module pyro.poutine.handlers), 155
CollapseMessenger (class in pyro.poutine.collapse_messenger), 168
Combination (class in pyro.contrib.gp.kernels), 256
CompartimentalModel (class in pyro.contrib.epidemiology.compartmental), 215
ComposeTransformModule (class in pyro.distributions.transforms), 120
compute_flows() (CompartimentalModel method), 218
compute_log_prob() (Trace method), 162
compute_marginals() (in module pyro.contrib.tracking.assignment), 289
compute_marginals() (TraceEnum_ELBO method), 13
compute_marginals_bp() (in module pyro.contrib.tracking.assignment), 289
compute_marginals_persistent() (in module pyro.contrib.tracking.assignment), 289
compute_marginals_persistent_bp() (in module pyro.contrib.tracking.assignment), 289
compute_score_parts() (Trace method), 162
compute_score_parts() (in module pyro.distributions.transforms), 289
centrer (DirichletMultinomial attribute), 67
concentration (GammaPoisson attribute), 72
concentration (InverseGamma attribute), 77
concentration (BetaBinomial attribute), 64
concentration (BetaBinomial attribute), 64
CondIndepStackFrame (class in pyro.poutine.indep_messenger), 170
condition() (AffineNormal method), 190
condition() (ConditionalAffineAutoregressive method), 99
condition() (ConditionalAffineCoupling method), 101
condition() (ConditionalDistribution method), 66
collapse() (ConditionalGeneralizedChannelPermute method), 102
condition() (ConditionalHouseholder method), 103
condition() (ConditionalMatrixExponential method), 104
condition() (ConditionalNeuralAutoregressive method), 105
condition() (ConditionalPlanar method), 106
condition() (ConditionalRadial method), 107
condition() (ConditionalSpline method), 108
condition() (ConditionalSplineAutoregressive method), 109
condition() (ConditionalTransform method), 90
condition() (ConditionalTransformedDistribution method), 66
condition() (Gaussian method), 189
condition() (in module pyro.poutine.handlers), 155
conditional() (in module pyro.contrib.gp.util), 265
conditional_affine_autoregressive() (in module pyro.distributions.transforms), 122
conditional_affine_coupling() (in module pyro.distributions.transforms), 123
conditional_generalized_channel_permute() (in module pyro.distributions.transforms), 123
conditional_householder() (in module pyro.distributions.transforms), 123
conditional_matrix_exponential() (in module pyro.distributions.transforms), 123
conditional_neural_autoregressive() (in module pyro.distributions.transforms), 124
conditional_planar() (in module pyro.distributions.transforms), 124
conditional_radial() (in module pyro.distributions.transforms), 125
conditional_spline() (in module pyro.distributions.transforms), 125
conditional_spline_autoregressive() (in module pyro.distributions.transforms), 125
ConditionalAffineAutoregressive (class in pyro.distributions.transforms), 98
ConditionalAffineCoupling (class in pyro.distributions.transforms), 100
ConditionalAutoRegressiveNN (class in pyro.nn.auto_reg_nn), 143
ConditionalDenseNN (class in pyro.nn.dense_nn), 144
ConditionalDistribution (class in pyro.distributions), 66
ConditionalGeneralizedChannelPermute (class in pyro.distributions.transforms), 101
ConditionalHouseholder (class in pyro.distributions.transforms), 102
ConditionalMatrixExponential (class in pyro.distributions.transforms), 103
pyro.contrib.forecast.evaluate

-mae()

-utils

eval_mae()

-utils

eval_mae()

-utils

evaluate_loss()

-utils

evaluate_mae()

-utils

evaluate_RMSE()

-utils

-escape_messenger

-utils

-escape_messenger

-utils

-escape_messenger

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils

-ons

-utils
Pyro Documentation

forecast() (GenericLGSSMWithGPNoiseModel method), 286
forecast() (IndependentMaternGP method), 282
forecast() (LinearlyCoupledMaternGP method), 283
forecast() (TimeSeriesModel method), 281
Forecaster (class in pyro.contrib.forecast.forecaster), 234
ForecastingModel (class in pyro.contrib.forecast.forecaster), 233
format_shapes() (Trace method), 162
forward() (AutoCallable method), 38
forward() (AutoContinuous method), 40
forward() (AutoDelta method), 39
forward() (AutoDiscreteParallel method), 44
forward() (AutoGuideList method), 37
forward() (AutoNormal method), 38
forward() (AutoRegressiveNN method), 142
forward() (BernoulliNet method), 209
forward() (Binary method), 262
forward() (Brownian method), 256
forward() (ConditionalAutoRegressiveNN method), 144
forward() (ConditionalDenseNN method), 145
forward() (Constant method), 256
forward() (Coregionalize method), 257
forward() (Cosine method), 257
forward() (DenseNN method), 143
forward() (DiagNormalNet method), 210
forward() (DynamicModel method), 290
forward() (EasyGuide method), 211
forward() (Exponent method), 257
forward() (Exponential method), 257
forward() (ExponentialNet method), 209
forward() (Gaussian method), 262
forward() (GPLVM method), 255
forward() (GPMultivariate method), 246
forward() (GPRegression method), 248
forward() (Guide method), 208
forward() (Kernel method), 255
forward() (LaplaceNet method), 209
forward() (Likelihood method), 261
forward() (Linear method), 258
forward() (Matern32 method), 258
forward() (Matern52 method), 258
forward() (Model method), 208
forward() (MultiClass method), 263
forward() (Ncp method), 292
forward() (Ncv method), 292
forward() (NormalNet method), 210
forward() (Periodic method), 259
forward() (Poisson method), 264
forward() (Polynomial method), 259
forward() (Predictive method), 25
forward() (Product method), 259
forward() (RationalQuadratic method), 260
forward() (RBF method), 260
forward() (SparseGPRegression method), 250
forward() (StudentTNet method), 210
forward() (Sum method), 260
forward() (VariationalGP method), 252
forward() (VariationalSparseGP method), 253
forward() (VerticalScaling method), 260
forward() (Warping method), 261
forward() (WhiteNoise method), 261
frame_num (EKFState attribute), 295
frame_num (Measurement attribute), 298
full_mass (CompartmentalModel attribute), 217
FullyConnected (class in pyro.contrib.cevae), 208

G
Gamma (class in pyro.distributions), 54
GammaGaussianHMM (class in pyro.distributions), 70
GammaPoisson (class in pyro.distributions), 72
gate (ZeroInflatedDistribution attribute), 89
gate_logs (ZeroInflatedDistribution attribute), 89
Gaussian (class in pyro.contrib.gp.likelihoods), 262
Gaussian (class in pyro.ops.gaussian), 188
gaussian_tensordot() (in module pyro.ops.gaussian), 191
GaussianHMM (class in pyro.distributions), 72
GaussianMRF (class in pyro.distributions), 74
GaussianScaleMixture (class in pyro.distributions), 75
gelman_rubin() (in module pyro.ops.stats), 191
generalized_channel_permute() (in module pyro.distributions.transforms), 126
GeneralizedChannelPermute (class in pyro.distributions.transforms), 110
generate() (CompartmentalModel method), 218
generator() (MarkovMessenger method), 171
GenericLGSSM (class in pyro.contrib.timeseries.lgssm), 285
GenericLGSSMWithGPNoiseModel (class in pyro.contrib.timeseries.lgssm_gp), 285
geodesic_difference() (DynamicModel method), 291
goedescic_difference() (Measurement method), 298
Geometric (class in pyro.distributions), 55
get_all_param_names() (ParamStoreDict method), 134
get_base_dist() (AutoContinuous method), 40
get_base_dist() (AutoDiagonalNormal method), 42
get_base_dist() (AutoMultivariateNormal method), 41
get_base_dist() (AutoNormalizingFlow method), 43
get_class() (DistributionNet static method), 209
get_covariance() (WelfordCovariance method), 179
get_covariance() (WelfordArrowheadCovariance method), 179
get_data_directory() (in module pyro.contrib.examples.util), 232
get_data_loader() (in module pyro.contrib.examples.util), 232
get_dist() (DependentMaternGP method), 284
get_dist() (GenericLGSSM method), 285
get_dist() (GenericLGSSMWithGPNoiseModel method), 286
get_dist() (IndependentMaternGP method), 282
get_dist() (LinearlyCoupledMaternGP method), 283
get_dist() (TimeSeriesModel method), 282
get_empirical() (SMCFilter method), 20
get_ESS() (Importance method), 17
get_log_normalizer() (Importance method), 17
get_named_particles() (SVGD method), 22
get_normalized_weights() (Importance method), 17
get_param() (ParamStoreDict method), 134
get_param_store() (in module pyro.contrib.minipyro), 268
get_permutation() (in module pyro.primitives), 3
get_posterior() (AutoContinuous method), 40
get_posterior() (AutoDiagonalNormal method), 42
get_posterior() (AutoLaplaceApproximation method), 44
get_posterior() (AutoLowRankMultivariateNormal method), 42
get_posterior() (AutoMultivariateNormal method), 41
get_posterior() (AutoNormalizingFlow method), 43
get_samples() (MCMC method), 28
get_samples() (Predictive method), 25
get_state() (DualAveraging method), 178
get_state() (ParamStoreDict method), 135
get_state() (PyroOptim method), 147
get_step() (MixedMultiOptimizer method), 152
get_step() (MultiOptimizer method), 151
get_step() (Newton method), 152
get_trace() (trace method), 268
get_trace() (TraceHandler method), 175
get_trace() (TraceMessenger method), 175
get_transform() (AutoContinuous method), 40
get_transform() (AutoDiagonalNormal method), 42
get_transform() (AutoMultivariateNormal method), 41
get_transform() (AutoNormalizingFlow method), 43
get_vectorized_trace() (Predictive method), 25
GLOBAL (DimType attribute), 241
global_frame (DimStack attribute), 241
global_model (CompartmentModel method), 217
GlobalNamedMessenger (class in pyro.contrib.funsor.handlers.named_messenger), 240
GPLVM (class in pyro.contrib.gp.models.gplvm), 254
GPModel (class in pyro.contrib.gp.models.model), 244
GPRegression (class in pyro.contrib.gp.models.gpr), 247
Group (class in pyro.contrib.cevae), 212
group() (EasyGuide method), 212
guide() (AutoContinuous method), 208
guide() (Group attribute), 213
guide() (EasyGuide method), 211
guide() (GPLVM method), 254
guide() (GPModel method), 246
guide() (GPRegression method), 248
guide() (SparseGPRegression method), 250
guide() (VariationalGP method), 252
guide() (VariationalSparseGP method), 253
Gumbel (class in pyro.distributions), 55

H
haar_transform() (in module pyro.ops.tensor_utils), 184
HaarReparam (class in pyro.infer.reparam.haar), 47
HaarTransform (class in pyro.distributions.transforms), 91
HalfCauchy (class in pyro.distributions), 55
HalfNormal (class in pyro.distributions), 55
has_enumerate_support (BetaBinomial attribute), 64
has_enumerate_support (Distribution attribute), 58
has_enumerate_support (Empirical attribute), 69
has_enumerate_support (MaskedDistribution attribute), 80
has_enumerate_support (SpanningTree attribute), 86
has_rsample (Delta attribute), 66
has_rsample (Distribution attribute), 58
has_rsample (GaussianHMM attribute), 74
has_rsample (GaussianScaleMixture attribute), 76
has_rsample (IndependentHMM attribute), 77
has_rsample (InverseGamma attribute), 77
has_rsample (LinearHMM attribute), 78
has_rsample (LKJCorrCholesky attribute), 79
has_rsample (MaskedDistribution attribute), 80
has_rsample (MaskedMixture attribute), 81
has_rsample (MixtureOfDiagNormals attribute), 82
has_rsample (MixtureOfDiagNormalsSharedCovariance attribute), 83
has_rsample (MultivariateStudentT attribute), 84
has_rsample (Rejection attribute), 85
has_rsample (Stable attribute), 86
has_rsample (TruncatedPolyaGamma attribute), 87
has_rsample_() (Distribution method), 59
HeterogeneousRegionalSIRModel (class in pyro.contrib.epidemiology.models), 226
HeterogeneousSIRModel (class in pyro.contrib.epidemiology.models), 227
heuristic() (CompartmentalModel method), 220
HiddenLayer (class in pyro.contrib.bnn.hidden_layer), 203
HMC (class in pyro.infer.mcmc), 30
HMCForecaster (class in pyro.contrib.forecast.forecaster), 236
HorovodOptimizer (class in pyro.optim.horovod), 149
Householder (class in pyro.distributions.transforms), 111
householder (in module pyro.distributions.transforms), 126
hpdi() (in module pyro.ops.stats), 192

iarange (class in pyro.primitives), 6
idct() (in module pyro.ops.tensor_utils), 184
identify_dense_edges() (in module pyro.poutine.trace_messenger), 175
Importance (class in pyro.infer.importance), 17
ImproperUniform (class in pyro.distributions.improper_uniform), 76
IMQSteinKernel (class in pyro.infer.svgd), 21
imresize() (in module pyro.contrib.examples.multi_mnist), 231
Independent (class in pyro.distributions), 55
independent() (TorchDistributionMixin method), 61
IndependentConstraint (class in pyro.distributions.constraints), 129
IndependentHMM (class in pyro.distributions), 76
IndependentMaternGP (class in pyro.contrib.timeseries_gp), 282
IndepMessenger (class in pyro.poutine.indep_messenger), 170
Index (class in pyro.ops.indexing), 185
index() (in module pyro.ops.indexing), 184
indices (IndepMessenger attribute), 170
inference_dist () (in module pyro.contrib.epidemiology.distributions), 228
infer_config() (in module pyro.poutine.handlers), 157
infer_discrete() (in module pyro.infer.discrete), 23
InferConfigMessenger (class in pyro.poutine.infer_config_messenger), 170
information_criterion() (TracePosterior method), 26
init() (EasyGuide method), 211
init() (SMCFilter method), 20
init_to_feasible() (in module pyro.infer.autoguide.initialization), 44
init_to_generated() (in module pyro.infer.autoguide.initialization), 45
init_to_mean() (in module pyro.infer.autoguide.initialization), 45
init_to_median() (in module pyro.infer.autoguide.initialization), 44
init_to_sample() (in module pyro.infer.autoguide.initialization), 44
init_to_uniform() (in module pyro.infer.autoguide.initialization), 45
init_to_value() (in module pyro.infer.autoguide.initialization), 45
initial_params (HMC attribute), 31
initial_params (MCMCKernel attribute), 29
initialize() (CompartmentalModel method), 217
initialize_model() (in module pyro.infer.mcmc.util), 35
InitMessenger (class in pyro.infer.autoguide.initialization), 45
innovation() (EKFState method), 295
integer (in module pyro.distributions.constraints), 130
inv_permutation (Permute attribute), 93
inverse_haar_transform() (in module pyro.distributions.transforms), 184
inverse_mass_matrix (BlockMassMatrix attribute), 34
inverse_mass_matrix (HMC attribute), 31
InvertGamma (class in pyro.distributions), 77
irange (class in pyro.primitives), 6
is_validation_enabled() (in module pyro.poutine.util), 165
Isotropy (class in pyro.contrib.gp.kernels), 258
ite() (CEVAE method), 207
items() (ParamStoreDict method), 134
iter_sample() (GPRegression method), 248
iter_stochastic_nodes() (Trace method), 163
iterated() (in module pyro.distributions.transforms), 120
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_partition_function</td>
<td>(SpanningTree attribute), 86</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(BetaBinomial method), 64</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(CoalescentTimes method), 64</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(CoalescentTimesWithRate method), 65</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(Delta method), 66</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(DependentMaternGP method), 284</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(DirichletMultinomial method), 67</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(DiscreteHMM method), 68</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(Distribution method), 58</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(EKFDistribution method), 290</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(Empirical method), 69</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(ExtendedBetaBinomial method), 69</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(ExtendedBinomial method), 70</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(FoldedDistribution method), 70</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GammaGaussianHMM method), 72</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GammaPoisson method), 72</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GaussianHMM method), 74</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GaussianMRF method), 75</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GaussianScaleMixture method), 76</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GenericLGSSM method), 285</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(GenericLGSSMWithGPNoiseModel method), 286</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(ImproperUniform method), 76</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(IndependentHMM method), 77</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(IndependentMaternGP method), 282</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(LinearHMM method), 79</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(LinearlyCoupledMaternGP method), 283</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(LKJCorrCholesky method), 79</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(MaskedDistribution method), 80</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(MaskedMixture method), 80</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(MixtureOfDiagNormals method), 81</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(MixtureOfDiagNormalsSharedCovariance method), 82</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(MultivariateStudentT method), 82</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(Rejector method), 85</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(RelaxedBernoulliStraightThrough method), 84</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(RelaxedOneHotCategoricalStraightThrough method), 84</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(SpanningTree method), 86</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(Stable method), 87</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(TimeSeriesModel method), 281</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(TruncatedPolyaGamma method), 88</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(Unit method), 88</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(VonMises3D method), 88</td>
</tr>
<tr>
<td>log_prob()</td>
<td>(ZeroInflatedDistribution method), 89</td>
</tr>
<tr>
<td>log_prob_sum()</td>
<td>(Trace method), 163</td>
</tr>
<tr>
<td>log_weights()</td>
<td>(Empirical attribute), 69</td>
</tr>
<tr>
<td>logging()</td>
<td>(HMC method), 32</td>
</tr>
<tr>
<td>logging()</td>
<td>(MCMCKernel method), 29</td>
</tr>
<tr>
<td>LogisticNormal</td>
<td>(class in pyro.distributions), 55</td>
</tr>
<tr>
<td>logits</td>
<td>(ZeroInflatedNegativeBinomial attribute), 89</td>
</tr>
<tr>
<td>LogNormal</td>
<td>(class in pyro.distributions), 55</td>
</tr>
<tr>
<td>loss()</td>
<td>(EnergyDistance method), 23</td>
</tr>
<tr>
<td>loss()</td>
<td>(RenyiELBO method), 16</td>
</tr>
<tr>
<td>loss()</td>
<td>(RewightedWakeSleep method), 19</td>
</tr>
<tr>
<td>loss()</td>
<td>(Trace_ELBO method), 11</td>
</tr>
<tr>
<td>loss()</td>
<td>(TraceCausalEffect_ELBO method), 208</td>
</tr>
<tr>
<td>loss()</td>
<td>(TraceEnum_ELBO method), 13</td>
</tr>
<tr>
<td>loss()</td>
<td>(TraceGraph_ELBO method), 12</td>
</tr>
<tr>
<td>loss()</td>
<td>(TraceMeanField_ELBO method), 14</td>
</tr>
<tr>
<td>loss()</td>
<td>(TraceTailAdaptive_ELBO method), 15</td>
</tr>
<tr>
<td>loss()</td>
<td>(TraceTMC_ELBO method), 16</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(JitTrace_ELBO method), 12</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(JitTraceEnum_ELBO method), 14</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(JitTraceGraph_ELBO method), 12</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(JitTraceMeanField_ELBO method), 12</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(RenyiELBO method), 16</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(RewightedWakeSleep method), 19</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(Trace_ELBO method), 11</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(TraceEnum_ELBO method), 13</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(TraceGraph_ELBO method), 12</td>
</tr>
<tr>
<td>loss_and_grads()</td>
<td>(TraceTMC_ELBO method), 16</td>
</tr>
<tr>
<td>loss_and_surrogate_loss()</td>
<td>(JitTrace_ELBO method), 11</td>
</tr>
<tr>
<td>LowerCholeskyAffine</td>
<td>(class in pyro.distributions.transforms), 92</td>
</tr>
<tr>
<td>LowRankMultivariateNormal</td>
<td>(class in pyro.distributions), 55</td>
</tr>
<tr>
<td>M</td>
<td>(class in pyro.contrib.tracking.hashing), 296</td>
</tr>
<tr>
<td>make_dist()</td>
<td>(BernoulliNet static method), 209</td>
</tr>
<tr>
<td>make_dist()</td>
<td>(ExponentialNet static method), 209</td>
</tr>
<tr>
<td>make_dist()</td>
<td>(LaplaceNet static method), 209</td>
</tr>
<tr>
<td>make_dist()</td>
<td>(NormalNet static method), 210</td>
</tr>
<tr>
<td>make_dist()</td>
<td>(StudentTNet static method), 210</td>
</tr>
<tr>
<td>map_estimate()</td>
<td>(EasyGuide method), 212</td>
</tr>
<tr>
<td>map_estimate()</td>
<td>(Group method), 213</td>
</tr>
<tr>
<td>marginal()</td>
<td>(TracePosterior method), 27</td>
</tr>
<tr>
<td>marginal()</td>
<td>(TracePredictive method), 27</td>
</tr>
<tr>
<td>marginal_eig()</td>
<td>(in module pyro.contrib.oed.eig), 274</td>
</tr>
<tr>
<td>MarginalAssignment</td>
<td>(class in pyro.contrib.tracking.assignment), 287</td>
</tr>
<tr>
<td>MarginalAssignmentPersistent</td>
<td>(class in pyro.contrib.tracking.assignment), 288</td>
</tr>
<tr>
<td>MarginalAssignmentSparse</td>
<td>(class in pyro.contrib.tracking.assignment), 287</td>
</tr>
<tr>
<td>marginalize()</td>
<td>(AffineNormal method), 190</td>
</tr>
<tr>
<td>marginalize()</td>
<td>(Gaussian method), 189</td>
</tr>
<tr>
<td>Marginals</td>
<td>(class in pyro.infer.abstract_infer), 26</td>
</tr>
</tbody>
</table>
markov() (in module pyro.contrib.funsor.handlers), 239
markov() (in module pyro.poutine.handlers), 157
MarkovMessenger (class in pyro.contrib.funsor.handlers.named_messenger), 240
MarkovMessenger (class in pyro.contrib.funsor.handlers.markov_messenger), 171
MaskedMixture (class in pyro.distributions), 61
MaskedDistribution (class in pyro.distributions), 79
MaskedMixt (class in pyro.distributions), 80
MaskMessenger (class in pyro.poutine.mask_messenger), 172
mass_matrix_adapter (HMC attribute), 32
mass_matrix_size (BlockMassMatrix attribute), 34
match() (in module ParamStoreDict), 134
Matern32 (class in pyro.contrib.gp.kernels), 258
Matern52 (class in pyro.contrib.gp.kernels), 258
MaternKernel (class in pyro.ops.ssm_gp), 194
matmul() (in module pyro.ops.tensor_utils), 184
matrix_and_mvn_to_gaussian() (in module pyro.ops.gaussian), 190
matrix_exponential() (in module pyro.distributions.transforms), 126
MatrixExponential (class in pyro.distributions.transforms), 111
matvecmul() (in module pyro.ops.tensor_utils), 184
MAX_DIM (DimStack attribute), 241
mc_extend() (in module pyro.poutine.util), 165
MCMC (class in pyro.infer.mcmc.api), 28
MCMCKernel (class in pyro.infer.mcmc.mcmkernel), 29
mean (BetaBinomial attribute), 64
mean (Delta attribute), 66
mean (DirichletMultinomial attribute), 67
mean (EKFState attribute), 295
mean (Empirical attribute), 69
mean (GammaPoisson attribute), 72
mean (MaskedDistribution attribute), 80
mean (MaskedMixture attribute), 80
mean (Measurement attribute), 298
mean (MultivariateStudentT attribute), 82
mean (Stable attribute), 87
mean (ZeroInflatedDistribution attribute), 89
mean2pv() (DynamicModel method), 291
mean2pv() (Ncp method), 292
mean2pv() (Ncv method), 293
mean_pv (EKFState attribute), 295
Measurement (class in pyro.contrib.tracking.measurements), 298
median() (AutoContinuous method), 40
median() (AutoDelta method), 39
median() (AutoGuide method), 36
median() (AutoGuideList method), 37
median() (AutoNormal method), 38
merge_points() (in module pyro.contrib.tracking.hashing), 297
Messenger (class in pyro.contrib.minipyro), 267
Messenger (class in pyro.contrib.poutine.messenger), 166
MixedMultiOptimizer (class in pyro.optim.multi), 152
MixtureOfDiagNormals (class in pyro.distributions), 81
MixtureOfDiagNormalsSharedCovariance (class in pyro.distributions), 81
MixtureSameFamily (class in pyro.distributions), 56
mk_dataset() (in module pyro.contrib.examples.multi_mnist), 231
mode (Parameterized attribute), 244, 265
model (AutoGuide attribute), 36
Model (class in pyro.contrib.cevae), 207
model (EasyGuide attribute), 211
model() (ForecastingModel method), 234
model() (GPLVM method), 254
model() (GPModel method), 246
model() (GPRegression method), 248
model() (SparseGPRegression method), 250
model() (VariationalGP method), 252
model() (VariationalSparseGP method), 253
module() (in module pyro.primitives), 7
module_from_param_with_module_name() (in module pyro.params.param_store), 135
MultiClass (class in pyro.contrib gp.likelihoods), 263
Multinomial (class in pyro.distributions), 56
MultiOptimizer (class in pyro.distributions), 151
MultiplicativeLR() (in module pyro.optim.pytorch_optimizers), 150
MultiStepLR() (in module pyro.optim.pytorch_optimizers), 150
MultivariateNormal (class in pyro.distributions), 56
MultivariateStudentT (class in pyro.distributions), 82
mvn_to_gaussian() (in module pyro.ops.gaussian), 190

N
name_count() (in module pyro.contrib.autoname), 196
name_count() (in module pyro.contrib.autoname.scoping), 200
NameCountMessenger (class in pyro.contrib.autoname.scoping), 199
named() (in module pyro.contrib.funsor.handlers), 239
named_parameters() (ParamStoreDict method), 134
Pyro Documentation

named_pyro_params() (PyroModule method), 140
NamedMessenger (class in pyro.contrib.funsor.handlers.named_messenger), 240
names_from_batch_shape() (DimStack method), 241
Ncp (class in pyro.contrib.tracking.dynamic_models), 291
NcpContinuous (class in pyro.contrib.tracking.dynamic_models), 293
NcpDiscrete (class in pyro.contrib.tracking.dynamic_models), 293
Ncv (class in pyro.contrib.tracking.dynamic_models), 292
NcvContinuous (class in pyro.contrib.tracking.dynamic_models), 293
NcvDiscrete (class in pyro.contrib.tracking.dynamic_models), 294
nearby() (LSH method), 297
NegativeBinomial (class in pyro.distributions), 56
neural_autoregressive() (in module pyro.distributions.transforms), 127
NeuralAutoregressive (class in pyro.distributions.transforms), 112
NeuTraReparam (class in pyro.infer.reparam.neutra), 50
Newton (class in pyro.optim.multi), 152
newton_step() (in module pyro.ops.newton), 179
newton_step_1d() (in module pyro.ops.newton), 180
newton_step_2d() (in module pyro.ops.newton), 180
newton_step_3d() (in module pyro.ops.newton), 180
next_context() (IndepMessenger method), 170
next_fast_len() (in module pyro.ops.tensor_utils), 183
nmc_eig() (in module pyro.contrib.oed.eig), 271
NonlocalExit, 163
nonreparam_stochastic_nodes (Trace attribute), 163
Normal (class in pyro.distributions), 56
NormalNet (class in pyro.contrib.cevae), 209
num_gamma_variates (TruncatedPolyaGamma attribute), 88
num_log_prob_terms (TruncatedPolyaGamma attribute), 88
num_process_noise_parameters (DynamicModel attribute), 290
num_steps (HMC attribute), 32
NUTS (class in pyro.infer.mcmc), 32
observation_nodes (Trace attribute), 163
OMTMultivariateNormal (class in pyro.distributions), 83
OneCycleLR() (in module pyro.optim.pytorch_optimizers), 150
OneHotCategorical (class in pyro.distributions), 56
ordered_vector (in module pyro.distributions.constraints), 130
OrderedLogistic (class in pyro.distributions), 83
OrderedTransform (class in pyro.distributions.transforms), 92
OverdispersedSEIRModel (class in pyro.contrib.epidemiology.models), 222
OverdispersedSIRModel (class in pyro.contrib.epidemiology.models), 222

P
pack_tensors() (Trace method), 163
param() (in module pyro.contrib.minipyro), 268
param() (in module pyro.primitives), 3
param_() (Object method), 198
param_name() (ParamStoreDict method), 135
param_nodes() (Trace attribute), 163
param_with_module_name() (in module pyro.params.param_store), 135
Parameterized (class in pyro.contrib.gp), 243
Parameterized (class in pyro.contrib.gp.parameterized), 264
ParamStoreDict (class in pyro.params.param_store), 133
Pareto (class in pyro.distributions), 56
Periodic (class in pyro.contrib.gp.kernels), 259
periodic_cumsum() (in module pyro.ops.tensor_utils), 182
periodic_features() (in module pyro.ops.tensor_utils), 182
periodic_repeat() (in module pyro.ops.tensor_utils), 182
Perrmute (class in pyro.distributions.transforms), 92
permute() (in module pyro.distributions.transforms), 127
pi() (in module pyro.ops.stats), 192
Planar (class in pyro.distributions.transforms), 113
planar() (in module pyro.distributions.transforms), 127
plate (class in pyro.primitives), 5
plate() (EasyGuide method), 212
plate() (in module pyro.contrib.funsor), 239
plate() (in module pyro.contrib.funsor.handlers), 239
plate() (in module pyro.contrib.minipyro), 268
plate_stack() (in module pyro.primitives), 6
PlateMessenger (class in pyro.contrib.minipyro), 267

O
Object (class in pyro.contrib.autoname.named), 197
pyro.poutine.replay_messenger (module), 173
pyro.poutine.runtime (module), 163
pyro.poutine.scale_messenger (module), 174
pyro.poutine.seed_messenger (module), 174
pyro.poutine.subsample_messenger (module), 174
pyro.poutine.trace_messenger (module), 175
pyro.poutine.uncondition_messenger (module), 176
pyro.poutine.runtime (module), 176
pyro.primitives.util (module), 164
pyro.poutine.scale_messenger (module), 175
pyro.poutine.replay_messenger (module), 176
PyroSample (class in pyro.nn.module), 138
PyroMultiOptimizer (class in pyro.optim.multi), 138
PyroOptim (class in pyro.optim.optim), 147
PyroParam (class in pyro.nn.module), 137
PyroSample (class in pyro.nn.module), 138
Q
Q() (Sylvester method), 119
quantile() (in module pyro.ops.stats), 192
quantiles() (AutoContinuous method), 41
quantiles() (AutoNormal method), 38
queue() (in module pyro.poutine.handlers), 158
R
R() (Sylvester method), 119
Radial (class in pyro.distributions.transforms), 115
radial() (in module pyro.distributions.transforms), 128
random_module() (in module pyro.primitives), 7
RandomVariable (class in pyro.contrib.randomvariable.random_variable), 279
rate (GammaPoisson attribute), 72
rate (InverseGamma attribute), 77
rate (ZeroInflatedPoisson attribute), 90
RationalQuadratic (class in pyro.contrib.gp.kernels), 260
RBF (class in pyro.contrib.gp.kernels), 259
RBFSteinKernel (class in pyro.infer.svdg), 21
ReduceLROnPlateau() (in module pyro.optim.pytorch_optimizers), 150
ReentrantMessenger (class in pyro.poutine.reentrant_messenger), 172
region_plate (CompartmentalModel attribute), 217
RegionalSIRModel (class in pyro.contrib.epidemiology.models), 226
register() (pyro.poutine.messenger.Messenger class method), 166
Rejector (class in pyro.distributions), 84
RelaxedBernoulli (class in pyro.distributions), 57
RelaxedBernoulliStraightThrough (class in pyro.distributions), 83
RelaxedOneHotCategorical (class in pyro.distributions), 57
RelaxedOneHotCategoricalStraightThrough (class in pyro.distributions), 84
remove() (LSH method), 297
remove_node() (Trace method), 163
RenyiELBO (class in pyro.infer.renyi_elbo), 15
Reparam (class in pyro.infer.reparam.reparam), 45
reparam() (in module pyro.poutine.handlers), 158
reparam() (NeuTraReparam method), 51
reparameterized_nodes (Trace attribute), 163
ReparamHandler (class in pyro.poutine.reparam_messenger), 173
ReparamMessenger (class in pyro.poutine.reparam_messenger), 173
repeated_matmul() (in module pyro.ops.tensor_utils), 183
replace_param() (ParamStoreDict method), 134
replay (class in pyro.contrib.minipyro), 268
replay() (in module pyro.contrib.functor_handlers), 240
replay() (in module pyro.poutine.handlers), 159
ReplayMessenger (class in pyro.poutine.replay_messenger), 173
resample() (in module pyro.ops.stats), 192
reset() (DualAveraging method), 178
reset() (WelfordArrowheadCovariance method), 179
reset() (WelfordCovariance method), 178
reset_parameters() (Householder method), 111
reset_parameters() (MatrixExponential method), 112
reset_parameters() (Planar method), 114
reset_parameters() (Polynomial method), 115
reset_parameters() (Radial method), 116
reset_parameters2() (Sylvester method), 120
reset_stack() (NonlocalExit method), 163
reshape() (AffineNormal method), 190
reshape() (Gaussian method), 188
reshape() (TorchDistributionMixin method), 61
RewightedWakeSleep (class in pyro.infer.rws), 18
RMSprop (in module pyro.optim.pytorch_optimizers), 150
Rprop (in module pyro.optim.pytorch_optimizers), 150
rsample() (AffineNormal method), 190
rsample() (AVFMultivariateNormal method), 63
rsample() (Delta method), 66
rsample() (Gaussian method), 189
rsample() (GaussianHMM method), 74
rsample() (GaussianScaleMixture method), 76
Index
time_plate (CompartmentalModel attribute), 216
time_plate (ForecastingModel attribute), 234
TimeSeriesModel (class in pyro.contrib.timeseries.base), 281
to_data () (in module pyro.contrib.funsor.handlers.primitives), 240
to_event () (TorchDistributionMixin method), 61
to_funsor () (in module pyro.contrib.funsor.handlers.primitives), 240
to_gaussian () (AffineNormal method), 190
to_pyro_module () (in module pyro.nn.module), 141
to_script_module () (CEVAE method), 207
topological_sort () (Trace method), 163
TorchDistribution (class in pyro.distributions), 62
TorchDistributionMixin (class in pyro.distributions.torch_distribution), 60
TorchMultiOptimizer (class in pyro.optim.multi), 152
total_count (ZeroInflatedNegativeBinomial attribute), 89
trace (class in pyro.contrib.minipyro), 268
Trace (class in pyro.poutine), 161
trace (TraceHandler attribute), 175
trace () (in module pyro.contrib.funsor.handlers), 240
trace () (in module pyro.ops.jit), 7
trace () (in module pyro.poutine.handlers), 160
Trace_ELBO (class in pyro.infer.trace_elbo), 11
Trace_ELBO () (in module pyro.mini.pyro), 267
TraceCausalEffect_ELBO (class in pyro.contrib.cevae), 208
TraceEnum_ELBO (class in pyro.infer.traceenum_elbo), 13
TraceEnumSample_ELBO (class in pyro.infer.discrete), 24
TraceGraph_ELBO (class in pyro.infer.tracegraph_elbo), 12
TraceHandler (class in pyro.poutine.trace_messenger), 175
TraceMeanField_ELBO (class in pyro.infer.trace_mean_field_elbo), 14
TraceMessenger (class in pyro.poutine.trace_messenger), 175
TracePosterior (class in pyro.infer.abstract_infer), 26
TracePredictive (class in pyro.infer.abstract_infer), 27
TraceTailAdaptive_ELBO (class in pyro.infer.trace_tail_adaptive_elbo), 15
TraceTMC_ELBO (class in pyro.infer.tracetmc_elbo), 16
train () (in module pyro.contrib.gp.util), 266
transform () (RandomVariable method), 279
transform_sample () (NeuTraReparam method), 51
TransformedDistribution (class in pyro.distributions), 57
Transforming (class in pyro.contrib.gp.kernels), 260
TransformModule (class in pyro.distributions), 120
TransformReparam (class in pyro.infer.reparam.transform), 47
transition () (CompartmentalModel method), 217
transition_matrix () (MaternKernel method), 194
transition_matrix_and_covariance () (MaternKernel method), 194
triangular_solve () (in module pyro.ops.tensor_utils), 184
TruncatedPolyaGamma (class in pyro.distributions), 87
truncation_point (TruncatedPolyaGamma attribute), 88
try_add () (ApproxSet method), 297

U
ubersum () (in module pyro.ops.contract), 188
uncondition () (in module pyro.poutine.handlers), 160
UnconditionMessenger (class in pyro.poutine.uncondition_messenger), 176
Uniform (class in pyro.distributions), 57
Unit (class in pyro.distributions), 88
UnitJacobianReparam (class in pyro.infer.reparam.unit_jacobian), 48
UnknownStartSIRModel (class in pyro.contrib.epidemiology.models), 225
unregister () (pyro.poutine.messenger.Messenger class method), 166

in
unscale () (BlockMassMatrix method), 34
update () (BlockMassMatrix method), 34
update () (EKFState method), 296
update () (WelfordArrowheadCovariance method), 179
update () (WelfordCovariance method), 178
user_param_name () (in module pyro.params.param_store), 135

V
validate_edges () (SpanningTree method), 86
validation_enabled () (in module pyro.primitives), 7
value (DimRequest attribute), 241
values () (ParamStoreDict method), 134
variance (BetaBinomial attribute), 64
variance (Delta attribute), 66
variance (DirichletMultinomial attribute), 67
variance (Empirical attribute), 69
variance (GammaPoisson attribute), 72
variance (MaskedDistribution attribute), 80
variance (MaskedMixture attribute), 81
variance (MultivariateStudentT attribute), 82
variance (Stable attribute), 87
variance (ZeroInflatedDistribution attribute), 89
VariationalGP (class in pyro.contrib.gp.models.vgp), 251
VariationalSparseGP (class in pyro.contrib.gp.models.vsgp), 252
vectorize() (in module pyro.infer.svgd), 22
vectorized (CondIndepStackFrame attribute), 170
vectorized_importance_weights() (in module pyro.infer.importance), 17
velocity_verlet() (in module pyro.ops.integrator), 178
VerticalScaling (class in pyro.contrib.gp.kernels), 260
vi_eig() (in module pyro.contrib.oed.eig), 271
Vindex (class in pyro.ops.indexing), 186
vindex() (in module pyro.ops.indexing), 185
VISIBLE (DimType attribute), 241
vnmc_eig() (in module pyro.contrib.oed.eig), 275
volume_preserving (Householder attribute), 111
volume_preserving (LowerCholeskyAffine attribute), 92
volume_preserving (Permute attribute), 93
VonMises (class in pyro.distributions), 57
VonMises3D (class in pyro.distributions), 88

W
waic() (in module pyro.ops.stats), 193
Warping (class in pyro.contrib.gp.kernels), 261
Weibull (class in pyro.distributions), 57
WelfordArrowheadCovariance (class in pyro.ops.welford), 179
WelfordCovariance (class in pyro.ops.welford), 178
WhiteNoise (class in pyro.contrib.gp.kernels), 261
with_cache() (DiscreteCosineTransform method), 94
with_cache() (HaarTransform method), 91
with_cache() (LowerCholeskyAffine method), 92
with_cache() (Permute method), 93

X
x_dist() (Model method), 208

Y
y_dist() (Guide method), 208
y_dist() (Model method), 208
y_mean() (Model method), 208

Z
z_dist() (Guide method), 208
z_dist() (Model method), 208