Pixyz Documentation

masa

Nov 06, 2020

Package Reference

Pixyz is a library for developing deep generative models in a more concise, intuitive and extendable way!

CHAPTER 1

pixyz.distributions (Distribution API)

1.1 Distribution

class `pixyz.distributions.distributions.`**`Distribution`**(*var*, *cond_var=[]*, *name='p'*, *features_shape=torch.Size([])*, *atomic=True*)

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

Distribution class. In Pixyz, all distributions are required to inherit this class.

Examples

```
>>> import torch
>>> from torch.nn import functional as F
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[64], name="p1")
>>> print(p1)
Distribution:
  p_{1}(x)
Network architecture:
  Normal(
    name=p_{1}, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([64])
    (loc): torch.Size([1, 64])
    (scale): torch.Size([1, 64])
  )
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[64], name="p2")
>>> print(p2)
```

(continues on next page)

(continued from previous page)

```
Distribution:
  p_{2}(x|y)
Network architecture:
  Normal(
    name=p_{2}, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.Size([64])
    (scale): torch.Size([1, 64])
  )
```

```
>>> # Conditional distribution (by neural networks)
>>> class P(Normal):
...     def __init__(self):
...         super().__init__(var=["x"],cond_var=["y"],name="p3")
...         self.model_loc = nn.Linear(128, 64)
...         self.model_scale = nn.Linear(128, 64)
...     def forward(self, y):
...         return {"loc": self.model_loc(y), "scale": F.softplus(self.model_
↪scale(y))}
>>> p3 = P()
>>> print(p3)
Distribution:
  p_{3}(x|y)
Network architecture:
  P(
    name=p_{3}, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.Size([])
    (model_loc): Linear(in_features=128, out_features=64, bias=True)
    (model_scale): Linear(in_features=128, out_features=64, bias=True)
  )
```

__init__(*var*, *cond_var=[]*, *name='p'*, *features_shape=torch.Size([])*, *atomic=True*)

Parameters

- **var** (list of str) – Variables of this distribution.
- **cond_var** (list of str, defaults to []) – Conditional variables of this distribution. In case that cond_var is not empty, we must set the corresponding inputs to sample variables.
- **name** (str, defaults to "p") – Name of this distribution. This name is displayed in *prob_text* and *prob_factorized_text*.
- **features_shape** (torch.Size or list, defaults to torch.Size())) – Shape of dimensions (features) of this distribution.

graph

distribution_name
Name of this distribution class.

Type str

name
Name of this distribution displayed in *prob_text* and *prob_factorized_text*.

Type str

var
Variables of this distribution.

Type list

cond_var
Conditional variables of this distribution.

Type list

input_var
Input variables of this distribution. Normally, it has same values as `cond_var`.

Type list

prob_text
Return a formula of the (joint) probability distribution.

Type str

prob_factorized_text
Return a formula of the factorized probability distribution.

Type str

prob_joint_factorized_and_text
Return a formula of the factorized and the (joint) probability distributions.

Type str

features_shape
Shape of features of this distribution.

Type torch.Size or list

sample (*x_dict={}*, *batch_n=None*, *sample_shape=torch.Size([])*, *return_all=True*, *reparam=False*)
Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
```

(continues on next page)

(continued from previous page)

```
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

sample_mean (*x_dict={}*)

Return the mean of the distribution.

Parameters **x_dict** (`dict`, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
```

(continues on next page)

(continued from previous page)

```
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> mean = p1.sample_mean()
>>> print(mean)
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> mean = p2.sample_mean({"y": sample_y})
>>> print(mean) # doctest: +SKIP
tensor([[-0.2189, -1.0310, -0.1917, -0.3085,  1.5190, -0.9037,  1.2559,  0.
↪1410,
          1.2810, -0.6681]])
```

sample_variance(*x_dict={}*)

Return the variance of the distribution.

Parameters **x_dict** (dict, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> var = p1.sample_variance()
>>> print(var)
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> var = p2.sample_variance({"y": sample_y})
>>> print(var) # doctest: +SKIP
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

get_log_prob(*x_dict, sum_features=True, feature_dims=None*)

Giving variables, this method returns values of log-pdf.

Parameters

- **x_dict** (*dict*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

get_entropy (*x_dict={}*, *sum_features=True*, *feature_dims=None*)
Giving variables, this method returns values of entropy.

Parameters

- **x_dict** (*dict, defaults to {}*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by feature_dims.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **entropy** – Values of entropy.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> entropy = p1.get_entropy()
>>> print(entropy)
tensor([14.1894])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> entropy = p2.get_entropy({"y": sample_y})
>>> print(entropy)
tensor([14.1894])
```

log_prob (*sum_features=True*, *feature_dims=None*)
Return an instance of *pixyz.losses.LogProb*.

Parameters

- **sum_features** (`bool`, defaults to True) – Whether the output is summed across some axes (dimensions) which are specified by `feature_dims`.
- **feature_dims** (`list` or `NoneType`, defaults to None) – Set axes to sum across the output.

Returns An instance of *pixyz.losses.LogProb*

Return type *pixyz.losses.LogProb*

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob().eval({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob().eval({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

prob (*sum_features=True*, *feature_dims=None*)

Return an instance of *pixyz.losses.Prob*.

Parameters

- **sum_features** (`bool`, defaults to True) – Choose whether the output is summed across some axes (dimensions) which are specified by `feature_dims`.
- **feature_dims** (`list` or `NoneType`, defaults to None) – Set dimensions to sum across the output. (Note: this parameter is not used for now.)

Returns An instance of *pixyz.losses.Prob*

Return type *pixyz.losses.Prob*

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> prob = p1.prob().eval({"x": sample_x})
>>> print(prob) # doctest: +SKIP
tensor([4.0933e-07])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> prob = p2.prob().eval({"x": sample_x, "y": sample_y})
>>> print(prob) # doctest: +SKIP
tensor([2.9628e-09])
```

forward(**args*, ***kwargs*)
When this class is inherited by DNNs, this method should be overrided.

replace_var(***replace_dict*)
Return an instance of *pixyz.distributions.ReplaceVarDistribution*.

Parameters **replace_dict** (*dict*) – Dictionary.

Returns An instance of *pixyz.distributions.ReplaceVarDistribution*

Return type *pixyz.distributions.ReplaceVarDistribution*

marginalize_var(*marginalize_list*)
Return an instance of *pixyz.distributions.MarginalizeVarDistribution*.

Parameters **marginalize_list** (list or other) – Variables to marginalize.

Returns An instance of *pixyz.distributions.MarginalizeVarDistribution*

Return type *pixyz.distributions.MarginalizeVarDistribution*

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

1.2 Exponential families

1.2.1 Normal

class pixyz.distributions.**Normal**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), loc=None, scale=None*)
Bases: pixyz.distributions.distributions.DistributionBase

Normal distribution parameterized by loc and scale.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Return the class of PyTorch distribution.

distribution_name
Name of this distribution class.

Type str

has_reparam

1.2.2 Laplace

class pixyz.distributions.**Laplace**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), loc=None, scale=None*)

Bases: `pixyz.distributions.distributions.DistributionBase`

Laplace distribution parameterized by `loc` and `scale`.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Return the class of PyTorch distribution.

distribution_name
Name of this distribution class.

Type str

has_reparam

1.2.3 Bernoulli

class pixyz.distributions.**Bernoulli**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), probs=None*)

Bases: `pixyz.distributions.distributions.DistributionBase`

Bernoulli distribution parameterized by `probs`.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Return the class of PyTorch distribution.

distribution_name
Name of this distribution class.

Type str

has_reparam

1.2.4 RelaxedBernoulli

class pixyz.distributions.**RelaxedBernoulli**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), temperature=tensor(0.1000), probs=None*)

Bases: `pixyz.distributions.exponential_distributions.Bernoulli`

Relaxed (re-parameterizable) Bernoulli distribution parameterized by `probs` and `temperature`.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Use relaxed version only when sampling

distribution_name
Name of this distribution class.

Type str

set_dist (*x_dict={}*, *batch_n=None*, *sampling=False*, ***kwargs*)
Set `dist` as PyTorch distributions given parameters.

This requires that `params_keys` and `distribution_torch_class` are set.

Parameters

- **x_dict** (`dict`, defaults to {}.) – Parameters of this distribution.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sampling** (`bool` defaults to False.) – If it is false, the distribution will not be relaxed to compute log_prob.
- ****kwargs** – Arbitrary keyword arguments.

sample (*x_dict={}*, *batch_n=None*, *sample_shape=torch.Size([])*, *return_all=True*, *reparam=False*)
Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
```

(continues on next page)

(continued from previous page)

```
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

1.2.5 FactorizedBernoulli

class pixyz.distributions.**FactorizedBernoulli**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), probs=None*)

Bases: `pixyz.distributions.exponential_distributions.Bernoulli`

Factorized Bernoulli distribution parameterized by `probs`.

References

[Vedantam+ 2017] Generative Models of Visually Grounded Imagination

distribution_name

Name of this distribution class.

Type str

get_log_prob (*x_dict*)

Giving variables, this method returns values of log-pdf.

Parameters

- **x_dict** (dict) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

1.2.6 Categorical

class pixyz.distributions.**Categorical** (*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), probs=None*)

Bases: pixyz.distributions.distributions.DistributionBase

Categorical distribution parameterized by probs.

params_keys

Return the list of parameter names for this distribution.

Type list

distribution_torch_class

Return the class of PyTorch distribution.

distribution_name

Name of this distribution class.

Type str

has_reparam

1.2.7 RelaxedCategorical

class pixyz.distributions.**RelaxedCategorical**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), temperature=tensor(0.1000), probs=None*)

Bases: pixyz.distributions.exponential_distributions.Categorical

Relaxed (re-parameterizable) categorical distribution parameterized by probs and temperature. Notes: a shape of temperature should contain the event shape of this Categorical distribution.

params_keys

Return the list of parameter names for this distribution.

Type list

distribution_torch_class

Use relaxed version only when sampling

distribution_name

Name of this distribution class.

Type str

set_dist(*x_dict={}, batch_n=None, sampling=False, **kwargs*)

Set dist as PyTorch distributions given parameters.

This requires that *params_keys* and *distribution_torch_class* are set.

Parameters

- **x_dict** (dict, defaults to {}.) – Parameters of this distribution.
- **batch_n** (int, defaults to None.) – Set batch size of parameters.
- **sampling** (bool defaults to False.) – If it is false, the distribution will not be relaxed to compute log_prob.
- ****kwargs** – Arbitrary keyword arguments.

sample(*x_dict={}, batch_n=None, sample_shape=torch.Size([]), return_all=True, reparam=False*)

Sample variables of this distribution. If cond_var is not empty, you should set inputs as dict.

Parameters

- **x_dict** (torch.Tensor, list, or dict, defaults to {}) – Input variables.
- **batch_n** (int, defaults to None.) – Set batch size of parameters.
- **sample_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
              -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
              -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
               0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
              -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
               0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

1.2.8 Beta

class pixyz.distributions.**Beta**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), concentration1=None, concentration0=None*)

Bases: `pixyz.distributions.distributions.DistributionBase`

Beta distribution parameterized by `concentration1` and `concentration0`.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Return the class of PyTorch distribution.

distribution_name
Name of this distribution class.

Type str

has_reparam

1.2.9 Dirichlet

class pixyz.distributions.**Dirichlet**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), concentration=None*)

Bases: `pixyz.distributions.distributions.DistributionBase`

Dirichlet distribution parameterized by `concentration`.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Return the class of PyTorch distribution.

distribution_name
Name of this distribution class.

Type str

has_reparam

1.2.10 Gamma

class pixyz.distributions.**Gamma**(*var=['x'], cond_var=[], name='p', features_shape=torch.Size([]), concentration=None, rate=None*)

Bases: `pixyz.distributions.distributions.DistributionBase`

Gamma distribution parameterized by `concentration` and `rate`.

params_keys
Return the list of parameter names for this distribution.

Type list

distribution_torch_class
Return the class of PyTorch distribution.

distribution_name
Name of this distribution class.

Type str

has_reparam

1.3 Complex distributions

1.3.1 MixtureModel

class pixyz.distributions.**MixtureModel**(*distributions*, *prior*, *name='p'*)
Bases: *pixyz.distributions.distributions.Distribution*

Mixture models.

$$p(x) = \sum_i p(x|z=i)p(z=i)$$

Examples

```
>>> from pixyz.distributions import Normal, Categorical
>>> from pixyz.distributions.mixture_distributions import MixtureModel
>>> z_dim = 3  # the number of mixture
>>> x_dim = 2  # the input dimension.
>>> distributions = []  # the list of distributions
>>> for i in range(z_dim):
...     loc = torch.randn(x_dim)  # initialize the value of location (mean)
...     scale = torch.empty(x_dim).fill_(1.)  # initialize the value of scale␣
↪(variance)
...     distributions.append(Normal(loc=loc, scale=scale, var=["x"], name="p_%d"
↪%i))
>>> probs = torch.empty(z_dim).fill_(1. / z_dim)  # initialize the value of␣
↪probabilities
>>> prior = Categorical(probs=probs, var=["z"], name="prior")
>>> p = MixtureModel(distributions=distributions, prior=prior)
>>> print(p)
Distribution:
  p(x) = p_{0}(x|z=0)prior(z=0) + p_{1}(x|z=1)prior(z=1) + p_{2}(x|z=2)prior(z=2)
Network architecture:
  MixtureModel(
    name=p, distribution_name=Mixture Model,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([])
    (distributions): ModuleList(
      (0): Normal(
        name=p_{0}, distribution_name=Normal,
        var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([2])
        (loc): torch.Size([1, 2])
        (scale): torch.Size([1, 2])
      )
      (1): Normal(
        name=p_{1}, distribution_name=Normal,
        var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([2])
        (loc): torch.Size([1, 2])
        (scale): torch.Size([1, 2])
```

(continues on next page)

(continued from previous page)

```
    )
    (2): Normal(
      name=p_{2}, distribution_name=Normal,
      var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([2])
      (loc): torch.Size([1, 2])
      (scale): torch.Size([1, 2])
    )
  )
  (prior): Categorical(
    name=prior, distribution_name=Categorical,
    var=['z'], cond_var=[], input_var=[], features_shape=torch.Size([3])
    (probs): torch.Size([1, 3])
  )
)
```

__init__(*distributions*, *prior*, *name='p'*)

Parameters

- **distributions** (*list*) – List of distributions.
- **prior** (*pixyz.Distribution.Categorical*) – Prior distribution of latent variable (i.e., a contribution rate). This should be a categorical distribution and the number of its category should be the same as the length of `distributions`.
- **name** (`str`, defaults to "p") – Name of this distribution. This name is displayed in `prob_text` and *prob_factorized_text*.

hidden_var

Hidden variables of this distribution.

Type list

prob_factorized_text

Return a formula of the factorized probability distribution.

Type str

distribution_name

Name of this distribution class.

Type str

posterior(*name=None*)

sample(*x_dict={}*, *batch_n=None*, *sample_shape=torch.Size([])*, *return_all=True*, *return_hidden=False*, ***kwargs*)

Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
              -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
              -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
               0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
              -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
               0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

get_log_prob (*x_dict*, *return_hidden=False*, ***kwargs*)
Evaluate log-pdf, log p(x) (if return_hidden=False) or log p(x, z) (if return_hidden=True).

Parameters

- **x_dict** (`dict`) – Input variables (including *var*).
- **return_hidden** (`bool`, defaults to False) –

Returns

log_prob – The log-pdf value of x.

return_hidden = 0 : dim=0 : the size of batch

return_hidden = 1 : dim=0 : the number of mixture

dim=1 : the size of batch

Return type torch.Tensor

1.3.2 ProductOfNormal

class pixyz.distributions.**ProductOfNormal** (*p=[]*, *name='p'*, *features_shape=torch.Size([])*)
Bases: `pixyz.distributions.exponential_distributions.Normal`

Product of normal distributions.

$$p(z|x,y) \propto p(z)p(z|x)p(z|y)$$

In this model, $p(z|x)$ and $p(a|y)$ perform as *experts* and $p(z)$ corresponds a prior of *experts*.

References

[Vedantam+ 2017] Generative Models of Visually Grounded Imagination

[Wu+ 2018] Multimodal Generative Models for Scalable Weakly-Supervised Learning

Examples

```
>>> pon = ProductOfNormal([p_x, p_y]) # doctest: +SKIP
>>> pon.sample({"x": x, "y": y}) # doctest: +SKIP
{'x': tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]],),
 'y': tensor([[0., 0., 0.,  ..., 0., 0., 1.],
        [0., 0., 1.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 1., 0.],
        [1., 0., 0.,  ..., 0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
     [0., 0., 0.,  ..., 0., 0., 1.]]),
 'z': tensor([[ 0.6611,  0.3811,  0.7778,  ..., -0.0468, -0.3615, -0.6569],
     [-0.0071, -0.9178,  0.6620,  ..., -0.1472,  0.6023,  0.5903],
     [-0.3723, -0.7758,  0.0195,  ...,  0.8239, -0.3537,  0.3854],
     ...,
     [ 0.7820, -0.4761,  0.1804,  ..., -0.5701, -0.0714, -0.5485],
     [-0.1873, -0.2105, -0.1861,  ..., -0.5372,  0.0752,  0.2777],
     [-0.2563, -0.0828,  0.1605,  ...,  0.2767, -0.8456,  0.7364]])}
>>> pon.sample({"y": y}) # doctest: +SKIP
{'y': tensor([[0., 0., 0.,  ..., 0., 0., 0.],
     [0., 0., 0.,  ..., 0., 0., 1.],
     [0., 0., 0.,  ..., 1., 0., 0.],
     ...,
     [0., 0., 0.,  ..., 0., 0., 0.],
     [0., 1., 0.,  ..., 0., 0., 0.],
     [0., 0., 0.,  ..., 0., 0., 0.]]),
 'z': tensor([[-0.3264, -0.4448,  0.3610,  ..., -0.7378,  0.3002,  0.4370],
     [ 0.0928, -0.1830,  1.1768,  ...,  1.1808, -0.7226, -0.4152],
     [ 0.6999,  0.2222, -0.2901,  ...,  0.5706,  0.7091,  0.5179],
     ...,
     [ 0.5688, -1.6612, -0.0713,  ..., -0.1400, -0.3903,  0.2533],
     [ 0.5412, -0.0289,  0.6365,  ...,  0.7407,  0.7838,  0.9218],
     [ 0.0299,  0.5148, -0.1001,  ...,  0.9938,  1.0689, -1.1902]])}
>>> pon.sample()  # same as sampling from unit Gaussian. # doctest: +SKIP
{'z': tensor(-0.4494)}
```

`__init__`(*p=[], name='p', features_shape=torch.Size([])*)

Parameters

- **p** (`list` of *`pixyz.distributions.Normal`*.) – List of experts.
- **name** (`str`, defaults to "p") – Name of this distribution. This name is displayed in prob_text and prob_factorized_text.
- **features_shape** (`torch.Size` or `list`, defaults to torch.Size())) – Shape of dimensions (features) of this distribution.

Examples

```
>>> p_x = Normal(cond_var=['z'], loc='z', scale=torch.ones(1, 1))
>>> pon = ProductOfNormal([p_x])
>>> sample = pon.sample({'z': torch.zeros(1, 1)})
>>> sample  # doctest: +SKIP
```

`prob_factorized_text`

Return a formula of the factorized probability distribution.

Type str

`prob_joint_factorized_and_text`

Return a formula of the factorized probability distribution.

Type str

`get_params`(*params_dict={}, **kwargs*)

`log_prob`(*sum_features=True, feature_dims=None*)

Return an instance of *`pixyz.losses.LogProb`*.

Parameters

- **sum_features** (bool, defaults to True) – Whether the output is summed across some axes (dimensions) which are specified by feature_dims.
- **feature_dims** (list or NoneType, defaults to None) – Set axes to sum across the output.

Returns An instance of *pixyz.losses.LogProb*

Return type *pixyz.losses.LogProb*

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob().eval({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob().eval({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

prob (*sum_features=True*, *feature_dims=None*)

Return an instance of *pixyz.losses.Prob*.

Parameters

- **sum_features** (bool, defaults to True) – Choose whether the output is summed across some axes (dimensions) which are specified by feature_dims.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output. (Note: this parameter is not used for now.)

Returns An instance of *pixyz.losses.Prob*

Return type *pixyz.losses.Prob*

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> prob = p1.prob().eval({"x": sample_x})
>>> print(prob) # doctest: +SKIP
tensor([4.0933e-07])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> prob = p2.prob().eval({"x": sample_x, "y": sample_y})
>>> print(prob) # doctest: +SKIP
tensor([2.9628e-09])
```

get_log_prob (*x_dict*, *sum_features=True*, *feature_dims=None*)

Giving variables, this method returns values of log-pdf.

Parameters

- **x_dict** (*dict*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

1.3.3 ElementWiseProductOfNormal

class pixyz.distributions.**ElementWiseProductOfNormal** (*p*, *name='p'*, *features_shape=torch.Size([])*)

Bases: pixyz.distributions.poe.ProductOfNormal

Product of normal distributions. In this distribution, each element of the input vector on the given distribution is considered as a different expert.

$$p(z|x) = p(z|x_1, x_2) \propto p(z)p(z|x_1)p(z|x_2)$$

Examples

```
>>> pon = ElementWiseProductOfNormal(p) # doctest: +SKIP
>>> pon.sample({"x": x}) # doctest: +SKIP
{'x': tensor([[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]]),
 'z': tensor([[-0.3572, -0.0632,  0.4872,  0.2269, -0.1693, -0.0160, -0.0429,  0.
↪2017,
        -0.1589, -0.3380, -0.9598,  0.6216, -0.4296, -1.1349,  0.0901,  0.3994,
         0.2313, -0.5227, -0.7973,  0.3968,  0.7137, -0.5639, -0.4891, -0.1249,
         0.8256,  0.1463,  0.0801, -1.2202,  0.6984, -0.4036,  0.4960, -0.4376,
         0.3310, -0.2243, -0.2381, -0.2200,  0.8969,  0.2674,  0.4681,  1.6764,
         0.8127,  0.2722, -0.2048,  0.1903, -0.1398,  0.0099,  0.4382, -0.8016,
         0.9947,  0.7556, -0.2017, -0.3920,  1.4212, -1.2529, -0.1002, -0.0031,
         0.1876,  0.4267,  0.3622,  0.2648,  0.4752,  0.0843, -0.3065, -0.4922],
        [ 0.3770, -0.0413,  0.9102,  0.2897, -0.0567,  0.5211,  1.5233, -0.3539,
         0.5163, -0.2271, -0.1027,  0.0294, -1.4617,  0.1640,  0.2025, -0.2190,
         0.0555,  0.5779, -0.2930, -0.2161,  0.2835, -0.0354, -0.2569, -0.7171,
         0.0164, -0.4080,  1.1088,  0.3947,  0.2720, -0.0600, -0.9295, -0.0234,
         0.5624,  0.4866,  0.5285,  1.1827,  0.2494,  0.0777,  0.7585,  0.5127,
         0.7500, -0.3253,  0.0250,  0.0888,  1.0340, -0.1405, -0.8114,  0.4492,
         0.2725, -0.0270,  0.6379, -0.8096,  0.4259,  0.3179, -0.1681,  0.3365,
         0.6305,  0.5203,  0.2384,  0.0572,  0.4804,  0.9553, -0.3244,  1.5373]])}
>>> pon.sample({"x": torch.zeros_like(x)})  # same as sampling from unit Gaussian.
↪ # doctest: +SKIP
{'x': tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]]),
 'z': tensor([[-0.7777, -0.5908, -1.5498, -0.7505,  0.6201,  0.7218,  1.0045,  0.
↪8923,
        -0.8030, -0.3569,  0.2932,  0.2122,  0.1640,  0.7893, -0.3500, -1.0537,
        -1.2769,  0.6122, -1.0083, -0.2915, -0.1928, -0.7486,  0.2418, -1.9013,
         1.2514,  1.3035, -0.3029, -0.3098, -0.5415,  1.1970, -0.4443,  2.2393,
        -0.6980,  0.2820,  1.6972,  0.6322,  0.4308,  0.8953,  0.7248,  0.4440,
         2.2770,  1.7791,  0.7563, -1.1781, -0.8331,  0.1825,  1.5447,  0.1385,
        -1.1348,  0.0257,  0.3374,  0.5889,  1.1231, -1.2476, -0.3801, -1.4404,
        -1.3066, -1.2653,  0.5958, -1.7423,  0.7189, -0.7236,  0.2330,  0.3117],
        [ 0.5495,  0.7210, -0.4708, -2.0631, -0.6170,  0.2436, -0.0133, -0.4616,
        -0.8091, -0.1592,  1.3117,  0.0276,  0.6625, -0.3748, -0.5049,  1.8260,
        -0.3631,  1.1546, -1.0913,  0.2712,  1.5493,  1.4294, -2.1245, -2.0422,
         0.4976, -1.2785,  0.5028,  1.4240,  1.1983,  0.2468,  1.1682, -0.6725,
        -1.1198, -1.4942, -0.3629,  0.1325, -0.2256,  0.4280,  0.9830, -1.9427,
        -0.2181,  1.1850, -0.7514, -0.8172,  2.1031, -0.1698, -0.3777, -0.7863,
         1.0936, -1.3720,  0.9999,  1.3302, -0.8954, -0.5999,  2.3305,  0.5702,
        -1.0767, -0.2750, -0.3741, -0.7026, -1.5408,  0.0667,  1.2550, -0.5117]])}
```

__init__(*p*, *name='p'*, *features_shape=torch.Size([])*)

Parameters

- **p** (`pixyz.distributions.Normal`) – Each element of this input vector is considered as a different expert. When some elements are 0, experts corresponding to these elements are considered not to be specified. $p(z|x) = p(z|x_1, x_2 = 0) \propto p(z)p(z|x_1)$
- **name** (*str, defaults to "p"*) – Name of this distribution. This name is displayed in prob_text and prob_factorized_text.
- **features_shape** (`torch.Size` or `list`, defaults to torch.Size())) – Shape of dimensions (features) of this distribution.

1.4 Flow distributions

1.4.1 TransformedDistribution

class pixyz.distributions.**TransformedDistribution**(*prior*, *flow*, *var*, *name='p'*)

Bases: *pixyz.distributions.distributions.Distribution*

Convert flow transformations to distributions.

$$p(z = f_{flow}(x)),$$

where $x \sim p_{prior}(x)$.

Once initializing, it can be handled as a distribution module.

distribution_name

Name of this distribution class.

Type str

flow_input_var

Input variables of the flow module.

Type list

prob_factorized_text

Return a formula of the factorized probability distribution.

Type str

logdet_jacobian

Get log-determinant Jacobian.

Before calling this, you should run *forward* or update_jacobian methods to calculate and store log-determinant Jacobian.

sample(*x_dict={}*, *batch_n=None*, *sample_shape=torch.Size([])*, *return_all=True*, *reparam=False*, *compute_jacobian=True*)

Sample variables of this distribution. If cond_var is not empty, you should set inputs as dict.

Parameters

- **x_dict** (torch.Tensor, list, or dict, defaults to {}) – Input variables.
- **batch_n** (int, defaults to None.) – Set batch size of parameters.
- **sample_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

get_log_prob (*x_dict*, *sum_features=True*, *feature_dims=None*, *compute_jacobian=False*)
It calculates the log-likelihood for a given z. If a flow module has no inverse method, it only supports the

previously sampled z-values.

forward(*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in *logdet_jacobian*.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

1.4.2 InverseTransformedDistribution

class pixyz.distributions.**InverseTransformedDistribution**(*prior*, *flow*, *var*, *cond_var=[]*, *name='p'*)
Bases: *pixyz.distributions.distributions.Distribution*

Convert inverse flow transformations to distributions.

$$p(x = f^{-1}_{flow}(z)),$$

where $z \sim p_{prior}(z)$.

Once initializing, it can be handled as a distribution module.

Moreover, this distribution can take a conditional variable.

$$p(x = f^{-1}_{flow}(z, y)),$$

where $z \sim p_{prior}(z)$ and y is given.

distribution_name
Name of this distribution class.

Type str

flow_output_var

prob_factorized_text
Return a formula of the factorized probability distribution.

Type str

logdet_jacobian

Get log-determinant Jacobian.

Before calling this, you should run *forward* or `update_jacobian` methods to calculate and store log-determinant Jacobian.

sample (*x_dict={}*, *batch_n=None*, *sample_shape=torch.Size([])*, *return_all=True*, *reparam=False*, *return_hidden=True*)

Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
```

(continues on next page)

(continued from previous page)

```
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

inference (*x_dict*, *return_all=True*, *compute_jacobian=False*)

get_log_prob (*x_dict*, *sum_features=True*, *feature_dims=None*)

Giving variables, this method returns values of log-pdf.

Parameters

- **x_dict** (*dict*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

forward(*x*, *y=None*, *compute_jacobian=True*)

Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in *logdet_jacobian*.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)

Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

1.5 Special distributions

1.5.1 Deterministic

class pixyz.distributions.**Deterministic**(*var*, *cond_var=[]*, *name='p'*, ***kwargs*)

Bases: *pixyz.distributions.distributions.Distribution*

Deterministic distribution (or degeneration distribution)

Examples

```
>>> import torch
>>> class Generator(Deterministic):
...     def __init__(self):
...         super().__init__(var=["x"], cond_var=["z"])
...         self.model = torch.nn.Linear(64, 512)
...     def forward(self, z):
...         return {"x": self.model(z)}
>>> p = Generator()
```

(continues on next page)

(continued from previous page)

```
>>> print(p)
Distribution:
  p(x|z)
Network architecture:
  Generator(
    name=p, distribution_name=Deterministic,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (model): Linear(in_features=64, out_features=512, bias=True)
  )
>>> sample = p.sample({"z": torch.randn(1, 64)})
>>> p.log_prob().eval(sample) # log_prob is not defined.
Traceback (most recent call last):
 ...
NotImplementedError: Log probability of deterministic distribution is not defined.
```

distribution_name

Name of this distribution class.

Type str

sample(*x_dict={}*, *return_all=True*, ***kwargs*)

Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
```

(continues on next page)

(continued from previous page)

```
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

sample_mean(*x_dict*)

Return the mean of the distribution.

Parameters **x_dict** (dict, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> mean = p1.sample_mean()
>>> print(mean)
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> mean = p2.sample_mean({"y": sample_y})
>>> print(mean) # doctest: +SKIP
tensor([[-0.2189, -1.0310, -0.1917, -0.3085,  1.5190, -0.9037,  1.2559,  0.
↪1410,
          1.2810, -0.6681]])
```

get_log_prob (*x_dict*, *sum_features=True*, *feature_dims=None*)
Giving variables, this method returns values of log-pdf.

Parameters

- **x_dict** (`dict`) – Input variables.
- **sum_features** (`bool`, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **feature_dims** (`list` or `NoneType`, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

has_reparam

1.5.2 EmpiricalDistribution

class pixyz.distributions.**EmpiricalDistribution** (*var*, *name='p_{data}'*)
Bases: *pixyz.distributions.distributions.Distribution*

Data distribution.

Samples from this distribution equal given inputs.

Examples

```
>>> import torch
>>> p = EmpiricalDistribution(var=["x"])
>>> print(p)
Distribution:
  p_{data}(x)
Network architecture:
  EmpiricalDistribution(
    name=p_{data}, distribution_name=Data distribution,
    var=['x'], cond_var=[], input_var=['x'], features_shape=torch.Size([])
  )
>>> sample = p.sample({"x": torch.randn(1, 64)})
```

distribution_name
Name of this distribution class.

Type str

sample(*x_dict={}*, *return_all=True*, ***kwargs*)
Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
```

(continues on next page)

(continued from previous page)

```
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

sample_mean(*x_dict*)

Return the mean of the distribution.

Parameters **x_dict** (dict, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> mean = p1.sample_mean()
>>> print(mean)
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
```

(continues on next page)

(continued from previous page)

```
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> mean = p2.sample_mean({"y": sample_y})
>>> print(mean) # doctest: +SKIP
tensor([[-0.2189, -1.0310, -0.1917, -0.3085,  1.5190, -0.9037,  1.2559,  0.
↪1410,
         1.2810, -0.6681]])
```

get_log_prob (*x_dict*, *sum_features=True*, *feature_dims=None*)
Giving variables, this method returns values of log-pdf.

Parameters

- **x_dict** (*dict*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

input_var
In EmpiricalDistribution, *input_var* is same as *var*.

has_reparam

1.5.3 CustomProb

class pixyz.distributions.**CustomProb** (*log_prob_function*, *var*, *distribution_name='Custom PDF'*, ***kwargs*)
Bases: *pixyz.distributions.distributions.Distribution*

This distribution is constructed by user-defined probability density/mass function.

Note that this distribution cannot perform sampling.

Examples

```
>>> import torch
>>> # banana shaped distribution
>>> def log_prob(z):
...     z1, z2 = torch.chunk(z, chunks=2, dim=1)
...     norm = torch.sqrt(z1 ** 2 + z2 ** 2)
...     exp1 = torch.exp(-0.5 * ((z1 - 2) / 0.6) ** 2)
...     exp2 = torch.exp(-0.5 * ((z1 + 2) / 0.6) ** 2)
...     u = 0.5 * ((norm - 2) / 0.4) ** 2 - torch.log(exp1 + exp2)
...     return -u
...
>>> p = CustomProb(log_prob, var=["z"])
>>> loss = p.log_prob().eval({"z": torch.randn(10, 2)})
```

`__init__`(*log_prob_function*, *var*, *distribution_name='Custom PDF'*, ***kwargs*)

Parameters

- **`log_prob_function`** (*`function`*) – User-defined log-probability density/mass function.
- **`var`** (*`list`*) – Variables of this distribution.
- **`distribution_name`** (`str`, optional) – Name of this distribution.
- **`+**kwargs`** – Arbitrary keyword arguments.

`log_prob_function`
User-defined log-probability density/mass function.

`input_var`
Input variables of this distribution. Normally, it has same values as `cond_var`.

Type list

`distribution_name`
Name of this distribution class.

Type str

`get_log_prob`(*x_dict*, *sum_features=True*, *feature_dims=None*)
Giving variables, this method returns values of log-pdf.

Parameters

- **`x_dict`** (*`dict`*) – Input variables.
- **`sum_features`** (`bool`, defaults to True) – Whether the output is summed across some dimensions which are specified by *feature_dims*.
- **`feature_dims`** (`list` or `NoneType`, defaults to None) – Set dimensions to sum across the output.

Returns **log_prob** – Values of log-probability density/mass function.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> sample_x = torch.randn(1, 10) # Psuedo data
>>> log_prob = p1.log_prob({"x": sample_x})
>>> print(log_prob) # doctest: +SKIP
tensor([-16.1153])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> log_prob = p2.log_prob({"x": sample_x, "y": sample_y})
>>> print(log_prob) # doctest: +SKIP
tensor([-21.5251])
```

sample (*x_dict={}*, *return_all=True*, ***kwargs*)

Sample variables of this distribution. If `cond_var` is not empty, you should set inputs as `dict`.

Parameters

- **x_dict** (`torch.Tensor`, `list`, or `dict`, defaults to {}) – Input variables.
- **batch_n** (`int`, defaults to None.) – Set batch size of parameters.
- **sample_shape** (`list` or `NoneType`, defaults to torch.Size()) – Shape of generating samples.
- **return_all** (`bool`, defaults to True) – Choose whether the output contains input variables.
- **reparam** (`bool`, defaults to False.) – Choose whether we sample variables with reparameterized trick.

Returns **output** – Samples of this distribution.

Return type dict

Examples

```
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10, 2])
>>> print(p)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([10, 2])
    (loc): torch.Size([1, 10, 2])
    (scale): torch.Size([1, 10, 2])
  )
```

(continues on next page)

(continued from previous page)

```
>>> p.sample()["x"].shape  # (batch_n=1, features_shape)
torch.Size([1, 10, 2])
>>> p.sample(batch_n=20)["x"].shape  # (batch_n, features_shape)
torch.Size([20, 10, 2])
>>> p.sample(batch_n=20, sample_shape=[40, 30])["x"].shape  # (sample_shape,␣
↪batch_n, features_shape)
torch.Size([40, 30, 20, 10, 2])
```

```
>>> # Conditional distribution
>>> p = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...            features_shape=[10])
>>> print(p)
Distribution:
  p(x|y)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['y'], input_var=['y'], features_shape=torch.
↪Size([10])
    (scale): torch.Size([1, 10])
  )
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> sample_a = torch.randn(1, 10) # Psuedo data
>>> sample = p.sample({"y": sample_y})
>>> print(sample) # input_var + var  # doctest: +SKIP
{'y': tensor([[-0.5182,  0.3484,  0.9042,  0.1914,  0.6905,
               -1.0859, -0.4433, -0.0255, 0.8198,  0.4571]]),
 'x': tensor([[-0.7205, -1.3996,  0.5528, -0.3059,  0.5384,
               -1.4976, -0.1480,  0.0841,0.3321,  0.5561]])}
>>> sample = p.sample({"y": sample_y, "a": sample_a}) # Redundant input ("a")
>>> print(sample) # input_var + var + "a" (redundant input)  # doctest: +SKIP
{'y': tensor([[ 1.3582, -1.1151, -0.8111,  1.0630,  1.1633,
                0.3855,  2.6324, -0.9357, -0.8649, -0.6015]]),
 'a': tensor([[-0.1874,  1.7958, -1.4084, -2.5646,  1.0868,
               -0.7523, -0.0852, -2.4222, -0.3914, -0.9755]]),
 'x': tensor([[-0.3272, -0.5222, -1.3659,  1.8386,  2.3204,
                0.3686,  0.6311, -1.1208, 0.3656, -0.6683]])}
```

has_reparam

1.6 Operators

1.6.1 ReplaceVarDistribution

class pixyz.distributions.**ReplaceVarDistribution**(*p*, *replace_dict*)

Bases: *pixyz.distributions.distributions.Distribution*

Replace names of variables in Distribution.

Examples

```
>>> p = DistributionBase(var=["x"],cond_var=["z"])
>>> print(p)
Distribution:
  p(x|z)
Network architecture:
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
>>> replace_dict = {'x': 'y'}
>>> p_repl = ReplaceVarDistribution(p, replace_dict)
>>> print(p_repl)
Distribution:
  p(y|z)
Network architecture:
  p(y|z) -> p(x|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
```

`__init__`(*p*, *replace_dict*)

Parameters

- **p** (`pixyz.distributions.Distribution` (not *`pixyz.distributions.MultiplyDistribution`*)) – Distribution.
- **replace_dict** (*`dict`*) – Dictionary.

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

When this class is inherited by DNNs, this method should be overrided.

`sample_mean`(*x_dict={}*)

Return the mean of the distribution.

Parameters **x_dict** (`dict`, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> mean = p1.sample_mean()
>>> print(mean)
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> mean = p2.sample_mean({"y": sample_y})
>>> print(mean) # doctest: +SKIP
tensor([[-0.2189, -1.0310, -0.1917, -0.3085,  1.5190, -0.9037,  1.2559,  0.
↪1410,
          1.2810, -0.6681]])
```

sample_variance(*x_dict={}*)

Return the variance of the distribution.

Parameters **x_dict** (dict, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> var = p1.sample_variance()
>>> print(var)
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> var = p2.sample_variance({"y": sample_y})
>>> print(var) # doctest: +SKIP
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

get_entropy(*x_dict={}*, *sum_features=True*, *feature_dims=None*)

Giving variables, this method returns values of entropy.

Parameters

- **x_dict** (*dict, defaults to {}*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by feature_dims.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **entropy** – Values of entropy.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> entropy = p1.get_entropy()
>>> print(entropy)
tensor([14.1894])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
```

(continues on next page)

(continued from previous page)

```
>>> entropy = p2.get_entropy({"y": sample_y})
>>> print(entropy)
tensor([14.1894])
```

distribution_name
Name of this distribution class.

Type str

1.6.2 MarginalizeVarDistribution

class pixyz.distributions.**MarginalizeVarDistribution**(*p: pixyz.distributions.distributions.Distribution, marginalize_list*)

Bases: *pixyz.distributions.distributions.Distribution*

Marginalize variables in Distribution.

$$p(x) = \int p(x,z)dz$$

Examples

```
>>> a = DistributionBase(var=["x"],cond_var=["z"])
>>> b = DistributionBase(var=["y"],cond_var=["z"])
>>> p_multi = a * b
>>> print(p_multi)
Distribution:
  p(x,y|z) = p(x|z)p(y|z)
Network architecture:
  p(y|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
  p(x|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
>>> p_marg = MarginalizeVarDistribution(p_multi, ["y"])
>>> print(p_marg)
Distribution:
  p(x|z) = \int p(x|z)p(y|z)dy
Network architecture:
  p(y|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
  p(x|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
```

__init__(*p: pixyz.distributions.distributions.Distribution*, *marginalize_list*)

Parameters

- **p** (`pixyz.distributions.Distribution` (not `pixyz.distributions.DistributionBase`)) – Distribution.
- **marginalize_list** (*list*) – Variables to marginalize.

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

When this class is inherited by DNNs, this method should be overrided.

sample_mean(*x_dict={}*)

Return the mean of the distribution.

Parameters **x_dict** (`dict`, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> mean = p1.sample_mean()
>>> print(mean)
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...             features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> mean = p2.sample_mean({"y": sample_y})
>>> print(mean) # doctest: +SKIP
tensor([[-0.2189, -1.0310, -0.1917, -0.3085,  1.5190, -0.9037,  1.2559,  0.
↪1410,
          1.2810, -0.6681]])
```

sample_variance(*x_dict={}*)

Return the variance of the distribution.

Parameters **x_dict** (`dict`, defaults to {}) – Parameters of this distribution.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...             features_shape=[10], name="p1")
>>> var = p1.sample_variance()
>>> print(var)
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
```

(continues on next page)

(continued from previous page)

```
...                features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> var = p2.sample_variance({"y": sample_y})
>>> print(var) # doctest: +SKIP
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

get_entropy(*x_dict={}*, *sum_features=True*, *feature_dims=None*)

Giving variables, this method returns values of entropy.

Parameters

- **x_dict** (*dict, defaults to {}*) – Input variables.
- **sum_features** (bool, defaults to True) – Whether the output is summed across some dimensions which are specified by feature_dims.
- **feature_dims** (list or NoneType, defaults to None) – Set dimensions to sum across the output.

Returns **entropy** – Values of entropy.

Return type torch.Tensor

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> # Marginal distribution
>>> p1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...                features_shape=[10], name="p1")
>>> entropy = p1.get_entropy()
>>> print(entropy)
tensor([14.1894])
```

```
>>> # Conditional distribution
>>> p2 = Normal(loc="y", scale=torch.tensor(1.), var=["x"], cond_var=["y"],
...                features_shape=[10], name="p2")
>>> sample_y = torch.randn(1, 10) # Psuedo data
>>> entropy = p2.get_entropy({"y": sample_y})
>>> print(entropy)
tensor([14.1894])
```

distribution_name

Name of this distribution class.

Type str

1.6.3 MultiplyDistribution

class pixyz.distributions.**MultiplyDistribution**(*a*, *b*)

Bases: *pixyz.distributions.distributions.Distribution*

Multiply by given distributions, e.g, $p(x,y|z) = p(x|z,y)p(y|z)$. In this class, it is checked if two distributions can be multiplied.

p(x|z)p(z|y) -> Valid

p(x|z)p(y|z) -> Valid

p(x|z)p(y|a) -> Valid

p(x|z)p(z|x) -> Invalid (recursive)

p(x|z)p(x|y) -> Invalid (conflict)

Examples

```
>>> a = DistributionBase(var=["x"],cond_var=["z"])
>>> b = DistributionBase(var=["z"],cond_var=["y"])
>>> p_multi = MultiplyDistribution(a, b)
>>> print(p_multi)
Distribution:
  p(x,z|y) = p(x|z)p(z|y)
Network architecture:
  p(z|y):
  DistributionBase(
    name=p, distribution_name=,
    var=['z'], cond_var=['y'], input_var=['y'], features_shape=torch.Size([])
  )
  p(x|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
>>> b = DistributionBase(var=["y"],cond_var=["z"])
>>> p_multi = MultiplyDistribution(a, b)
>>> print(p_multi)
Distribution:
  p(x,y|z) = p(x|z)p(y|z)
Network architecture:
  p(y|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
  p(x|z):
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
>>> b = DistributionBase(var=["y"],cond_var=["a"])
>>> p_multi = MultiplyDistribution(a, b)
>>> print(p_multi)
Distribution:
  p(x,y|z,a) = p(x|z)p(y|a)
Network architecture:
  p(y|a):
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['a'], input_var=['a'], features_shape=torch.Size([])
  )
  p(x|z):
  DistributionBase(
    name=p, distribution_name=,
```

(continues on next page)

(continued from previous page)

```
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
)
```

__init__(*a*, *b*)

Parameters

- **a** (*pixyz.Distribution*) – Distribution.
- **b** (*pixyz.Distribution*) – Distribution.

CHAPTER 2

---

pixyz.losses (Loss API)

2.1 Loss

class `pixyz.losses.losses.Loss`(*input_var=None*)

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

Loss class. In Pixyz, all loss classes are required to inherit this class.

Examples

```
>>> import torch
>>> from torch.nn import functional as F
>>> from pixyz.distributions import Bernoulli, Normal
>>> from pixyz.losses import KullbackLeibler
...
>>> # Set distributions
>>> class Inference(Normal):
...     def __init__(self):
...         super().__init__(var=["z"],cond_var=["x"],name="q")
...         self.model_loc = torch.nn.Linear(128, 64)
...         self.model_scale = torch.nn.Linear(128, 64)
...     def forward(self, x):
...         return {"loc": self.model_loc(x), "scale": F.softplus(self.model_
↪scale(x))}
...
>>> class Generator(Bernoulli):
...     def __init__(self):
...         super().__init__(var=["x"],cond_var=["z"],name="p")
...         self.model = torch.nn.Linear(64, 128)
...     def forward(self, z):
...         return {"probs": torch.sigmoid(self.model(z))}
...
>>> p = Generator()
```

(continues on next page)

(continued from previous page)

```
>>> q = Inference()
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[64], name="p_{prior}")
...
>>> # Define a loss function (VAE)
>>> reconst = -p.log_prob().expectation(q)
>>> kl = KullbackLeibler(q,prior)
>>> loss_cls = (reconst - kl).mean()
>>> print(loss_cls)
mean \left(- D_{KL} \left[q(z|x)||p_{prior}(z) \right] - \mathbb{E}_{q(z|x)}
↪\left[\log p(x|z) \right] \right)
>>> # Evaluate this loss function
>>> data = torch.randn(1, 128)  # Pseudo data
>>> loss = loss_cls.eval({"x": data})
>>> print(loss)  # doctest: +SKIP
tensor(65.5939, grad_fn=<MeanBackward0>)
```

__init__(*input_var=None*)

Parameters **input_var** (list of str, defaults to None) – Input variables of this loss function. In general, users do not need to set them explicitly because these depend on the given distributions and each loss function.

input_var

Input variables of this distribution.

Type list

loss_text

abs()

Return an instance of *pixyz.losses.losses.AbsLoss*.

Returns An instance of *pixyz.losses.losses.AbsLoss*

Return type *pixyz.losses.losses.AbsLoss*

mean()

Return an instance of *pixyz.losses.losses.BatchMean*.

Returns An instance of pixyz.losses.BatchMean

Return type *pixyz.losses.losses.BatchMean*

sum()

Return an instance of *pixyz.losses.losses.BatchSum*.

Returns An instance of *pixyz.losses.losses.BatchSum*

Return type *pixyz.losses.losses.BatchSum*

detach()

Return an instance of *pixyz.losses.losses.Detach*.

Returns An instance of *pixyz.losses.losses.Detach*

Return type *pixyz.losses.losses.Detach*

expectation(*p*, *sample_shape=torch.Size([])*)

Return an instance of *pixyz.losses.Expectation*.

Parameters

- **p** (*pixyz.distributions.Distribution*) – Distribution for sampling.
- **sample_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.

Returns An instance of *pixyz.losses.Expectation*

Return type *pixyz.losses.Expectation*

constant_var (*constant_dict*)

Return an instance of pixyz.losses.ConstantVar.

Parameters **constant_dict** (*dict*) – constant variables.

Returns An instance of pixyz.losses.ConstantVar

Return type pixyz.losses.ConstantVar

eval (*x_dict={}*, *return_dict=False*, *return_all=True*, ***kwargs*)

Evaluate the value of the loss function given inputs (x_dict).

Parameters

- **x_dict** (dict, defaults to {}) – Input variables.
- **return_dict** (*bool, default to False.*) – Whether to return samples along with the evaluated value of the loss function.
- **return_all** (*bool, default to True.*) – Whether to return all samples, including those that have not been updated.

Returns

- **loss** (*torch.Tensor*) – the evaluated value of the loss function.
- **x_dict** (dict) – All samples generated when evaluating the loss function. If return_dict is False, it is not returned.

forward (*x_dict*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.2 Probability density function

2.2.1 LogProb

class pixyz.losses.**LogProb** (*p*, *sum_features=True*, *feature_dims=None*)

Bases: *pixyz.losses.losses.Loss*

The log probability density/mass function.

$$\log p(x)$$

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10])
>>> loss_cls = LogProb(p)  # or p.log_prob()
>>> print(loss_cls)
\log p(x)
>>> sample_x = torch.randn(2, 10) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor([12.9894, 15.5280])
```

forward(*x={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.2.2 Prob

class pixyz.losses.**Prob**(*p*, *sum_features=True*, *feature_dims=None*)

Bases: `pixyz.losses.pdf.LogProb`

The probability density/mass function.

$$p(x) = \exp(\log p(x))$$

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10])
>>> loss_cls = Prob(p)  # or p.prob()
>>> print(loss_cls)
p(x)
>>> sample_x = torch.randn(2, 10) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor([3.2903e-07, 5.5530e-07])
```

forward(*x={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.3 Expected value

2.3.1 Expectation

class pixyz.losses.**Expectation**(*p, f, sample_shape=torch.Size([1]), reparam=True*)

Bases: *pixyz.losses.losses.Loss*

Expectation of a given function (Monte Carlo approximation).

$$\mathbb{E}_{p(x)}[f(x)] \approx \frac{1}{L}\sum_{l=1}^{L} f(x_l), \quad \text{where} \quad x_l \sim p(x).$$

Note that f doesn't need to be able to sample, which is known as the law of the unconscious statistician (LOTUS).

Therefore, in this class, f is assumed to pixyz.Loss.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal, Bernoulli
>>> from pixyz.losses import LogProb
>>> q = Normal(loc="x", scale=torch.tensor(1.), var=["z"], cond_var=["x"],
...            features_shape=[10]) # q(z|x)
>>> p = Normal(loc="z", scale=torch.tensor(1.), var=["x"], cond_var=["z"],
...            features_shape=[10]) # p(x|z)
>>> loss_cls = LogProb(p).expectation(q) # equals to Expectation(q, LogProb(p))
>>> print(loss_cls)
\mathbb{E}_{p(z|x)} \left[\log p(x|z) \right]
>>> sample_x = torch.randn(2, 10) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor([-12.8181, -12.6062])
>>> loss_cls = LogProb(p).expectation(q,sample_shape=(5,))
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
>>> q = Bernoulli(probs=torch.tensor(0.5), var=["x"], cond_var=[], features_
↪shape=[10]) # q(x)
>>> p = Bernoulli(probs=torch.tensor(0.3), var=["x"], cond_var=[], features_
↪shape=[10]) # p(x)
>>> loss_cls = p.log_prob().expectation(q,sample_shape=[64])
>>> train_loss = loss_cls.eval()
>>> print(train_loss) # doctest: +SKIP
tensor([46.7559])
>>> eval_loss = loss_cls.eval(test_mode=True)
>>> print(eval_loss) # doctest: +SKIP
tensor([-7.6047])
```

forward(*x_dict={}, **kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.3.2 REINFORCE

`pixyz.losses.REINFORCE` (*p, f, b=0, sample_shape=torch.Size([1]), reparam=True*)

Surrogate Loss for Policy Gradient Method (REINFORCE) with a given reward function f and a given baseline b.

$$\mathbb{E}_{p(x)}[detach(f(x) - b(x)) \log p(x) + f(x) - b(x)].$$

in this function, f and b is assumed to `pixyz.Loss`.

Parameters

- **p** (`pixyz.distributions.Distribution`) – Distribution for expectation.
- **f** (`pixyz.losses.Loss`) – reward function
- **b** (`pixyz.losses.Loss` default to pixyz.losses.ValueLoss(0)) – baseline function
- **sample_shape** (`torch.Size` default to torch.Size([1])) – sample size for expectation
- **reparam** – using reparameterization in internal sampling

Returns **surrogate_loss** – policy gradient can be calcurated from a gradient of this surrogate loss.

Return type `pixyz.losses.Loss`

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal, Bernoulli
>>> from pixyz.losses import LogProb
>>> q = Bernoulli(probs=torch.tensor(0.5), var=["x"], cond_var=[], features_
↪shape=[10]) # q(x)
>>> p = Bernoulli(probs=torch.tensor(0.3), var=["x"], cond_var=[], features_
↪shape=[10]) # p(x)
>>> loss_cls = REINFORCE(q,p.log_prob(),sample_shape=[64])
>>> train_loss = loss_cls.eval(test_mode=True)
>>> print(train_loss) # doctest: +SKIP
tensor([46.7559])
>>> loss_cls = p.log_prob().expectation(q,sample_shape=[64])
>>> test_loss = loss_cls.eval()
>>> print(test_loss) # doctest: +SKIP
tensor([-7.6047])
```

2.4 Entropy

2.4.1 Entropy

`pixyz.losses.Entropy` (*p, analytical=True, sample_shape=torch.Size([1])*)

Entropy (Analytical or Monte Carlo approximation).

$$\begin{aligned} H(p) &= -\mathbb{E}_{p(x)}[\log p(x)] \qquad \text{(analytical)} \\ &\approx -\frac{1}{L}\sum_{l=1}^{L} \log p(x_l), \quad \text{where} \quad x_l \sim p(x) \quad \text{(MC approximation).} \end{aligned}$$

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"], features_
↪shape=[64])
>>> loss_cls = Entropy(p,analytical=True)
>>> print(loss_cls)
H \left[ {p(x)} \right]
>>> loss_cls.eval()
tensor([90.8121])
>>> loss_cls = Entropy(p,analytical=False,sample_shape=[10])
>>> print(loss_cls)
- \mathbb{E}_{p(x)} \left[\log p(x) \right]
>>> loss_cls.eval() # doctest: +SKIP
tensor([90.5991])
```

2.4.2 CrossEntropy

`pixyz.losses.CrossEntropy`(*p*, *q*, *analytical=False*, *sample_shape=torch.Size([1])*)

Cross entropy, a.k.a., the negative expected value of log-likelihood (Monte Carlo approximation or Analytical).

$$\begin{aligned} H(p,q) &= -\mathbb{E}_{p(x)}[\log q(x)] \qquad \text{(analytical)} \\ &\approx -\frac{1}{L}\sum_{l=1}^{L}\log q(x_l), \quad \text{where} \quad x_l \sim p(x) \quad \text{(MC approximation).} \end{aligned}$$

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"], features_
↪shape=[64], name="p")
>>> q = Normal(loc=torch.tensor(1.), scale=torch.tensor(1.), var=["x"], features_
↪shape=[64], name="q")
>>> loss_cls = CrossEntropy(p,q,analytical=True)
>>> print(loss_cls)
D_{KL} \left[p(x)||q(x) \right] + H \left[ {p(x)} \right]
>>> loss_cls.eval()
tensor([122.8121])
>>> loss_cls = CrossEntropy(p,q,analytical=False,sample_shape=[10])
>>> print(loss_cls)
- \mathbb{E}_{p(x)} \left[\log q(x) \right]
>>> loss_cls.eval() # doctest: +SKIP
tensor([123.2192])
```

2.5 Lower bound

2.5.1 ELBO

`pixyz.losses.ELBO(p, q, sample_shape=torch.Size([1]))`

The evidence lower bound (Monte Carlo approximation).

$$\mathbb{E}_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right] \approx \frac{1}{L}\sum_{l=1}^{L} \log p(x, z_l), \quad \text{where} \quad z_l \sim q(z|x).$$

Note: This class is a special case of the *Expectation* class.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> q = Normal(loc="x", scale=torch.tensor(1.), var=["z"], cond_var=["x"],␣
↪features_shape=[64]) # q(z|x)
>>> p = Normal(loc="z", scale=torch.tensor(1.), var=["x"], cond_var=["z"],␣
↪features_shape=[64]) # p(x|z)
>>> loss_cls = ELBO(p,q)
>>> print(loss_cls)
\mathbb{E}_{p(z|x)} \left[\log p(x|z) - \log p(z|x) \right]
>>> loss = loss_cls.eval({"x": torch.randn(1, 64)})
```

2.6 Statistical distance

2.6.1 KullbackLeibler

`pixyz.losses.KullbackLeibler(p, q, dim=None, analytical=True, sample_shape=torch.Size([1]))`

Kullback-Leibler divergence (analytical or Monte Carlo Apploximation).

$$\begin{aligned} D_{KL}[p||q] &= \mathbb{E}_{p(x)}\left[\log \frac{p(x)}{q(x)}\right] \qquad \text{(analytical)} \\ &\approx \frac{1}{L}\sum_{l=1}^{L} \log \frac{p(x_l)}{q(x_l)}, \quad \text{where} \quad x_l \sim p(x) \quad \text{(MC approximation).} \end{aligned}$$

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal, Beta
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["z"], features_
↪shape=[64], name="p")
>>> q = Normal(loc=torch.tensor(1.), scale=torch.tensor(1.), var=["z"], features_
↪shape=[64], name="q")
>>> loss_cls = KullbackLeibler(p,q,analytical=True)
```

(continues on next page)

(continued from previous page)

```
>>> print(loss_cls)
D_{KL} \left[p(z)||q(z) \right]
>>> loss_cls.eval()
tensor([32.])
>>> loss_cls = KullbackLeibler(p,q,analytical=False,sample_shape=[64])
>>> print(loss_cls)
\mathbb{E}_{p(z)} \left[\log p(z) - \log q(z) \right]
>>> loss_cls.eval() # doctest: +SKIP
tensor([31.4713])
```

2.6.2 WassersteinDistance

class pixyz.losses.**WassersteinDistance**(*p*, *q*, *metric=PairwiseDistance()*)

Bases: pixyz.losses.losses.Divergence

Wasserstein distance.

$$W(p, q) = \inf_{\Gamma \in \mathcal{P}(x_p\sim p, x_q\sim q)} \mathbb{E}_{(x_p, x_q) \sim \Gamma}[d(x_p, x_q)]$$

However, instead of the above true distance, this class computes the following one.

$$W'(p, q) = \mathbb{E}_{x_p\sim p, x_q \sim q}[d(x_p, x_q)].$$

Here, W' is the upper of W (i.e., $W\leq W'$), and these are equal when both p and q are degenerate (deterministic) distributions.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc="x", scale=torch.tensor(1.), var=["z"], cond_var=["x"],␣
↪features_shape=[64], name="p")
>>> q = Normal(loc="x", scale=torch.tensor(1.), var=["z"], cond_var=["x"],␣
↪features_shape=[64], name="q")
>>> loss_cls = WassersteinDistance(p, q)
>>> print(loss_cls)
W^{upper} \left(p(z|x), q(z|x) \right)
>>> loss = loss_cls.eval({"x": torch.randn(1, 64)})
```

forward(*x_dict*, ***kwargs*)

Parameters x_dict (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.6.3 MMD

class pixyz.losses.**MMD**(*p*, *q*, *kernel='gaussian'*, ***kernel_params*)

Bases: pixyz.losses.losses.Divergence

The Maximum Mean Discrepancy (MMD).

$$D_{MMD^2}[p||q] = \mathbb{E}_{p(x),p(x')}[k(x,x')] + \mathbb{E}_{q(x),q(x')}[k(x,x')] - 2\mathbb{E}_{p(x),q(x')}[k(x,x')]$$

where $k(x,x')$ is any positive definite kernel.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc="x", scale=torch.tensor(1.), var=["z"], cond_var=["x"],
↪features_shape=[64], name="p")
>>> q = Normal(loc="x", scale=torch.tensor(1.), var=["z"], cond_var=["x"],
↪features_shape=[64], name="q")
>>> loss_cls = MMD(p, q, kernel="gaussian")
>>> print(loss_cls)
D_{MMD^2} \left[p(z|x)||q(z|x) \right]
>>> loss = loss_cls.eval({"x": torch.randn(1, 64)})
>>> # Use the inverse (multi-)quadric kernel
>>> loss = MMD(p, q, kernel="inv-multiquadratic").eval({"x": torch.randn(10, 64)})
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.7 Adversarial statistical distance

2.7.1 AdversarialJensenShannon

class pixyz.losses.**AdversarialJensenShannon** (*p*, *q*, *discriminator*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *inverse_g_loss=True*)

Bases: `pixyz.losses.adversarial_loss.AdversarialLoss`

Jensen-Shannon divergence (adversarial training).

$$D_{JS}[p(x)||q(x)] \leq 2 \cdot D_{JS}[p(x)||q(x)] + 2\log 2 = \mathbb{E}_{p(x)}[\log d^*(x)] + \mathbb{E}_{q(x)}[\log(1-d^*(x))],$$

where $d^*(x) = \arg\max_d \mathbb{E}_{p(x)}[\log d(x)] + \mathbb{E}_{q(x)}[\log(1-d(x))]$.

This class acts as a metric that evaluates a given distribution (generator). If you want to learn this evaluation metric itself, i.e., discriminator (critic), use the `train` method.

Examples

```
>>> import torch
>>> from pixyz.distributions import Deterministic, EmpiricalDistribution, Normal
>>> # Generator
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(var=["x"], cond_var=["z"], name="p")
...         self.model = nn.Linear(32, 64)
...     def forward(self, z):
...         return {"x": self.model(z)}
>>> p_g = Generator()
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[32], name="p_{prior}")
>>> p = (p_g*prior).marginalize_var("z")
>>> print(p)
Distribution:
  p(x) = \int p(x|z)p_{prior}(z)dz
Network architecture:
  p_{prior}(z):
  Normal(
    name=p_{prior}, distribution_name=Normal,
    var=['z'], cond_var=[], input_var=[], features_shape=torch.Size([32])
    (loc): torch.Size([1, 32])
    (scale): torch.Size([1, 32])
  )
  p(x|z):
  Generator(
    name=p, distribution_name=Deterministic,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (model): Linear(in_features=32, out_features=64, bias=True)
  )
>>> # Data distribution (dummy distribution)
>>> p_data = EmpiricalDistribution(["x"])
>>> print(p_data)
Distribution:
  p_{data}(x)
Network architecture:
  EmpiricalDistribution(
    name=p_{data}, distribution_name=Data distribution,
    var=['x'], cond_var=[], input_var=['x'], features_shape=torch.Size([])
  )
>>> # Discriminator (critic)
>>> class Discriminator(Deterministic):
...     def __init__(self):
...         super(Discriminator, self).__init__(var=["t"], cond_var=["x"], name="d
↪")
...         self.model = nn.Linear(64, 1)
...     def forward(self, x):
...         return {"t": torch.sigmoid(self.model(x))}
>>> d = Discriminator()
>>> print(d)
Distribution:
  d(t|x)
Network architecture:
  Discriminator(
    name=d, distribution_name=Deterministic,
    var=['t'], cond_var=['x'], input_var=['x'], features_shape=torch.Size([])
    (model): Linear(in_features=64, out_features=1, bias=True)
```

(continues on next page)

(continued from previous page)

```
    )
>>>
>>> # Set the loss class
>>> loss_cls = AdversarialJensenShannon(p, p_data, discriminator=d)
>>> print(loss_cls)
mean(D_{JS}^{Adv} \left[p(x)||p_{data}(x) \right])
>>>
>>> sample_x = torch.randn(2, 64) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor(1.3723, grad_fn=<AddBackward0>)
>>> # For evaluating a discriminator loss, set the `discriminator` option to True.
>>> loss_d = loss_cls.eval({"x": sample_x}, discriminator=True)
>>> print(loss_d) # doctest: +SKIP
tensor(1.4990, grad_fn=<AddBackward0>)
>>> # When training the evaluation metric (discriminator), use the train method.
>>> train_loss = loss_cls.loss_train({"x": sample_x})
```

References

[Goodfellow+ 2014] Generative Adversarial Networks

forward (*x_dict*, *discriminator=False*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

d_loss (*y_p*, *y_q*, *batch_n*)

Evaluate a discriminator loss given outputs of the discriminator.

Parameters

- **y_p** (*torch.Tensor*) – Output of discriminator given sample from p.
- **y_q** (*torch.Tensor*) – Output of discriminator given sample from q.
- **batch_n** (*int*) – Batch size of inputs.

Returns

Return type torch.Tensor

g_loss (*y_p*, *y_q*, *batch_n*)

Evaluate a generator loss given outputs of the discriminator.

Parameters

- **y_p** (*torch.Tensor*) – Output of discriminator given sample from p.
- **y_q** (*torch.Tensor*) – Output of discriminator given sample from q.
- **batch_n** (*int*) – Batch size of inputs.

Returns

Return type torch.Tensor

loss_train (*train_x_dict*, ***kwargs*)
Train the evaluation metric (discriminator).

Parameters

- **train_x_dict** (*dict*) – Input variables.
- ****kwargs** – Arbitrary keyword arguments.

Returns loss

Return type torch.Tensor

loss_test (*test_x_dict*, ***kwargs*)
Test the evaluation metric (discriminator).

Parameters

- **test_x_dict** (*dict*) – Input variables.
- ****kwargs** – Arbitrary keyword arguments.

Returns loss

Return type torch.Tensor

2.7.2 AdversarialKullbackLeibler

class pixyz.losses.**AdversarialKullbackLeibler** (*p*, *q*, *discriminator*, ***kwargs*)
Bases: pixyz.losses.adversarial_loss.AdversarialLoss

Kullback-Leibler divergence (adversarial training).

$$D_{KL}[p(x)||q(x)] = \mathbb{E}_{p(x)}\left[\log \frac{p(x)}{q(x)}\right] \approx \mathbb{E}_{p(x)}\left[\log \frac{d^*(x)}{1-d^*(x)}\right],$$

where $d^*(x) = \arg\max_d \mathbb{E}_{q(x)}[\log d(x)] + \mathbb{E}_{p(x)}[\log(1-d(x))]$.

Note that this divergence is minimized to close p to q.

Examples

```
>>> import torch
>>> from pixyz.distributions import Deterministic, EmpiricalDistribution, Normal
>>> # Generator
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(var=["x"], cond_var=["z"], name="p")
...         self.model = nn.Linear(32, 64)
...     def forward(self, z):
...         return {"x": self.model(z)}
>>> p_g = Generator()
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[32], name="p_{prior}")
>>> p = (p_g*prior).marginalize_var("z")
>>> print(p)
Distribution:
  p(x) = \int p(x|z)p_{prior}(z)dz
Network architecture:
  p_{prior}(z):
```

(continues on next page)

(continued from previous page)

```
  Normal(
    name=p_{prior}, distribution_name=Normal,
    var=['z'], cond_var=[], input_var=[], features_shape=torch.Size([32])
    (loc): torch.Size([1, 32])
    (scale): torch.Size([1, 32])
  )
  p(x|z):
  Generator(
    name=p, distribution_name=Deterministic,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (model): Linear(in_features=32, out_features=64, bias=True)
  )
>>> # Data distribution (dummy distribution)
>>> p_data = EmpiricalDistribution(["x"])
>>> print(p_data)
Distribution:
  p_{data}(x)
Network architecture:
  EmpiricalDistribution(
    name=p_{data}, distribution_name=Data distribution,
    var=['x'], cond_var=[], input_var=['x'], features_shape=torch.Size([])
  )
>>> # Discriminator (critic)
>>> class Discriminator(Deterministic):
...     def __init__(self):
...         super(Discriminator, self).__init__(var=["t"], cond_var=["x"], name="d
↪")
...         self.model = nn.Linear(64, 1)
...     def forward(self, x):
...         return {"t": torch.sigmoid(self.model(x))}
>>> d = Discriminator()
>>> print(d)
Distribution:
  d(t|x)
Network architecture:
  Discriminator(
    name=d, distribution_name=Deterministic,
    var=['t'], cond_var=['x'], input_var=['x'], features_shape=torch.Size([])
    (model): Linear(in_features=64, out_features=1, bias=True)
  )
>>>
>>> # Set the loss class
>>> loss_cls = AdversarialKullbackLeibler(p, p_data, discriminator=d)
>>> print(loss_cls)
mean(D_{KL}^{Adv} \left[p(x)||p_{data}(x) \right])
>>>
>>> sample_x = torch.randn(2, 64) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> # The evaluation value might be negative if the discriminator training is␣
↪incomplete.
>>> print(loss) # doctest: +SKIP
tensor(-0.8377, grad_fn=<AddBackward0>)
>>> # For evaluating a discriminator loss, set the `discriminator` option to True.
>>> loss_d = loss_cls.eval({"x": sample_x}, discriminator=True)
>>> print(loss_d) # doctest: +SKIP
tensor(1.9321, grad_fn=<AddBackward0>)
>>> # When training the evaluation metric (discriminator), use the train method.
```

(continues on next page)

(continued from previous page)

```
>>> train_loss = loss_cls.loss_train({"x": sample_x})
```

References

[Kim+ 2018] Disentangling by Factorising

forward (*x_dict*, *discriminator=False*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

g_loss (*y_p*, *batch_n*)

Evaluate a generator loss given an output of the discriminator.

Parameters

- **y_p** (*torch.Tensor*) – Output of discriminator given sample from p.
- **batch_n** (*int*) – Batch size of inputs.

Returns

Return type torch.Tensor

d_loss (*y_p*, *y_q*, *batch_n*)

Evaluate a discriminator loss given outputs of the discriminator.

Parameters

- **y_p** (*torch.Tensor*) – Output of discriminator given sample from p.
- **y_q** (*torch.Tensor*) – Output of discriminator given sample from q.
- **batch_n** (*int*) – Batch size of inputs.

Returns

Return type torch.Tensor

loss_train (*train_x_dict*, ***kwargs*)

Train the evaluation metric (discriminator).

Parameters

- **train_x_dict** (*dict*) – Input variables.
- ****kwargs** – Arbitrary keyword arguments.

Returns loss

Return type torch.Tensor

loss_test (*test_x_dict*, ***kwargs*)

Test the evaluation metric (discriminator).

Parameters

- **test_x_dict** (*dict*) – Input variables.
- ****kwargs** – Arbitrary keyword arguments.

Returns loss

Return type torch.Tensor

2.7.3 AdversarialWassersteinDistance

class pixyz.losses.**AdversarialWassersteinDistance**(*p*, *q*, *discriminator*, *clip_value=0.01*, ***kwargs*)

Bases: pixyz.losses.adversarial_loss.AdversarialJensenShannon

Wasserstein distance (adversarial training).

$$W(p,q) = \sup_{||d||_L \leq 1} \mathbb{E}_{p(x)}[d(x)] - \mathbb{E}_{q(x)}[d(x)]$$

Examples

```
>>> import torch
>>> from pixyz.distributions import Deterministic, EmpiricalDistribution, Normal
>>> # Generator
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(var=["x"], cond_var=["z"], name="p")
...         self.model = nn.Linear(32, 64)
...     def forward(self, z):
...         return {"x": self.model(z)}
>>> p_g = Generator()
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[32], name="p_{prior}")
>>> p = (p_g*prior).marginalize_var("z")
>>> print(p)
Distribution:
  p(x) = \int p(x|z)p_{prior}(z)dz
Network architecture:
  p_{prior}(z):
  Normal(
    name=p_{prior}, distribution_name=Normal,
    var=['z'], cond_var=[], input_var=[], features_shape=torch.Size([32])
    (loc): torch.Size([1, 32])
    (scale): torch.Size([1, 32])
  )
  p(x|z):
  Generator(
    name=p, distribution_name=Deterministic,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (model): Linear(in_features=32, out_features=64, bias=True)
  )
>>> # Data distribution (dummy distribution)
>>> p_data = EmpiricalDistribution(["x"])
>>> print(p_data)
Distribution:
  p_{data}(x)
Network architecture:
  EmpiricalDistribution(
    name=p_{data}, distribution_name=Data distribution,
    var=['x'], cond_var=[], input_var=['x'], features_shape=torch.Size([])
```

(continues on next page)

(continued from previous page)

```
  )
>>> # Discriminator (critic)
>>> class Discriminator(Deterministic):
...     def __init__(self):
...         super(Discriminator, self).__init__(var=["t"], cond_var=["x"], name="d
↪")
...         self.model = nn.Linear(64, 1)
...     def forward(self, x):
...         return {"t": self.model(x)}
>>> d = Discriminator()
>>> print(d)
Distribution:
  d(t|x)
Network architecture:
  Discriminator(
    name=d, distribution_name=Deterministic,
    var=['t'], cond_var=['x'], input_var=['x'], features_shape=torch.Size([])
    (model): Linear(in_features=64, out_features=1, bias=True)
  )
>>>
>>> # Set the loss class
>>> loss_cls = AdversarialWassersteinDistance(p, p_data, discriminator=d)
>>> print(loss_cls)
mean(W^{Adv} \left(p(x), p_{data}(x) \right))
>>>
>>> sample_x = torch.randn(2, 64) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor(-0.0060, grad_fn=<SubBackward0>)
>>> # For evaluating a discriminator loss, set the `discriminator` option to True.
>>> loss_d = loss_cls.eval({"x": sample_x}, discriminator=True)
>>> print(loss_d) # doctest: +SKIP
tensor(-0.3802, grad_fn=<NegBackward>)
>>> # When training the evaluation metric (discriminator), use the train method.
>>> train_loss = loss_cls.loss_train({"x": sample_x})
```

References

[Arjovsky+ 2017] Wasserstein GAN

d_loss (*y_p, y_q, *args, **kwargs*)

Evaluate a discriminator loss given outputs of the discriminator.

Parameters

- **y_p** (*torch.Tensor*) – Output of discriminator given sample from p.
- **y_q** (*torch.Tensor*) – Output of discriminator given sample from q.
- **batch_n** (*int*) – Batch size of inputs.

Returns

Return type torch.Tensor

g_loss (*y_p, y_q, *args, **kwargs*)

Evaluate a generator loss given outputs of the discriminator.

Parameters

- **y_p** (*torch.Tensor*) – Output of discriminator given sample from p.
- **y_q** (*torch.Tensor*) – Output of discriminator given sample from q.
- **batch_n** (*int*) – Batch size of inputs.

Returns

Return type torch.Tensor

loss_train (*train_x_dict*, ***kwargs*)
Train the evaluation metric (discriminator).

Parameters

- **train_x_dict** (*dict*) – Input variables.
- ****kwargs** – Arbitrary keyword arguments.

Returns loss

Return type torch.Tensor

loss_test (*test_x_dict*, ***kwargs*)
Test the evaluation metric (discriminator).

Parameters

- **test_x_dict** (*dict*) – Input variables.
- ****kwargs** – Arbitrary keyword arguments.

Returns loss

Return type torch.Tensor

2.8 Loss for sequential distributions

2.8.1 IterativeLoss

class pixyz.losses.**IterativeLoss** (*step_loss*, *max_iter=None*, *series_var=()*, *update_value={}*, *slice_step=None*, *timestep_var=()*)
Bases: *pixyz.losses.losses.Loss*

Iterative loss.

This class allows implementing an arbitrary model which requires iteration.

$$\mathcal{L} = \sum_{t=0}^{T-1} \mathcal{L}_{step}(x_t, h_t),$$

where $x_t = f_{slice\_step}(x, t)$.

Examples

```
>>> import torch
>>> from torch.nn import functional as F
>>> from pixyz.distributions import Normal, Bernoulli, Deterministic
>>>
>>> # Set distributions
```

(continues on next page)

(continued from previous page)

```
>>> x_dim = 128
>>> z_dim = 64
>>> h_dim = 32
>>>
>>> # p(x|z,h_{prev})
>>> class Decoder(Bernoulli):
...     def __init__(self):
...         super().__init__(var=["x"],cond_var=["z", "h_prev"],name="p")
...         self.fc = torch.nn.Linear(z_dim + h_dim, x_dim)
...     def forward(self, z, h_prev):
...         return {"probs": torch.sigmoid(self.fc(torch.cat((z, h_prev), dim=-
↪1)))}
...
>>> # q(z|x,h_{prev})
>>> class Encoder(Normal):
...     def __init__(self):
...         super().__init__(var=["z"],cond_var=["x", "h_prev"],name="q")
...         self.fc_loc = torch.nn.Linear(x_dim + h_dim, z_dim)
...         self.fc_scale = torch.nn.Linear(x_dim + h_dim, z_dim)
...     def forward(self, x, h_prev):
...         xh = torch.cat((x, h_prev), dim=-1)
...         return {"loc": self.fc_loc(xh), "scale": F.softplus(self.fc_
↪scale(xh))}
...
>>> # f(h|x,z,h_{prev}) (update h)
>>> class Recurrence(Deterministic):
...     def __init__(self):
...         super().__init__(var=["h"], cond_var=["x", "z", "h_prev"], name="f")
...         self.rnncell = torch.nn.GRUCell(x_dim + z_dim, h_dim)
...     def forward(self, x, z, h_prev):
...         return {"h": self.rnncell(torch.cat((z, x), dim=-1), h_prev)}
>>>
>>> p = Decoder()
>>> q = Encoder()
>>> f = Recurrence()
>>>
>>> # Set the loss class
>>> step_loss_cls = p.log_prob().expectation(q * f).mean()
>>> print(step_loss_cls)
mean \left(\mathbb{E}_{q(z,h|x,h_{prev})} \left[\log p(x|z,h_{prev}) \right]␣
↪\right)
>>> loss_cls = IterativeLoss(step_loss=step_loss_cls,
...                          series_var=["x"], update_value={"h": "h_prev"})
>>> print(loss_cls)
\sum_{t=0}^{t_{max} - 1} mean \left(\mathbb{E}_{q(z,h|x,h_{prev})} \left[\log␣
↪p(x|z,h_{prev}) \right] \right)
>>>
>>> # Evaluate
>>> x_sample = torch.randn(30, 2, 128) # (timestep_size, batch_size, feature_size)
>>> h_init = torch.zeros(2, 32) # (batch_size, h_dim)
>>> loss = loss_cls.eval({"x": x_sample, "h_prev": h_init})
>>> print(loss) # doctest: +SKIP
tensor(-2826.0906, grad_fn=<AddBackward0>)
```

slice_step_fn (*t, x*)

forward (*x_dict, **kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.9 Loss for special purpose

2.9.1 Parameter

class pixyz.losses.losses.**Parameter**(*input_var*)

Bases: *pixyz.losses.losses.Loss*

This class defines a single variable as a loss class.

It can be used such as a coefficient parameter of a loss class.

Examples

```
>>> loss_cls = Parameter("x")
>>> print(loss_cls)
x
>>> loss = loss_cls.eval({"x": 2})
>>> print(loss)
2
```

forward(*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.9.2 ValueLoss

class pixyz.losses.losses.**ValueLoss**(*loss1*)

Bases: *pixyz.losses.losses.Loss*

This class contains a scalar as a loss value.

If multiplying a scalar by an arbitrary loss class, this scalar is converted to the *ValueLoss*.

Examples

```
>>> loss_cls = ValueLoss(2)
>>> print(loss_cls)
2
>>> loss = loss_cls.eval()
>>> print(loss)
tensor(2.)
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.9.3 ConstantVar

class pixyz.losses.losses.**ConstantVar** (*base_loss*, *constant_dict*)

Bases: *pixyz.losses.losses.Loss*

This class is defined as a loss class that makes the value of a variable a constant before evaluation.

It can be used to fix the coefficient parameters of the loss class or to condition random variables.

Examples

```
>>> loss_cls = Parameter('x').constant_var({'x': 1})
>>> print(loss_cls)
x
>>> loss = loss_cls.eval()
>>> print(loss)
1
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10 Operators

2.10.1 LossOperator

class pixyz.losses.losses.**LossOperator** (*loss1*, *loss2*)

Bases: *pixyz.losses.losses.Loss*

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.2 LossSelfOperator

class pixyz.losses.losses.**LossSelfOperator**(*loss1*)

Bases: *pixyz.losses.losses.Loss*

loss_train(*x_dict={}*, ***kwargs*)

loss_test(*x_dict={}*, ***kwargs*)

2.10.3 AddLoss

class pixyz.losses.losses.**AddLoss**(*loss1*, *loss2*)

Bases: *pixyz.losses.losses.LossOperator*

Apply the *add* operation to the two losses.

Examples

```
>>> loss_cls_1 = ValueLoss(2)
>>> loss_cls_2 = Parameter("x")
>>> loss_cls = loss_cls_1 + loss_cls_2  # equals to AddLoss(loss_cls_1, loss_cls_
↪2)
>>> print(loss_cls)
x + 2
>>> loss = loss_cls.eval({"x": 3})
>>> print(loss)
tensor(5.)
```

forward(*x_dict={}*, ***kwargs*)

Parameters x_dict (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.10.4 SubLoss

class pixyz.losses.losses.**SubLoss**(*loss1*, *loss2*)

Bases: *pixyz.losses.losses.LossOperator*

Apply the *sub* operation to the two losses.

Examples

```
>>> loss_cls_1 = ValueLoss(2)
>>> loss_cls_2 = Parameter("x")
>>> loss_cls = loss_cls_1 - loss_cls_2  # equals to SubLoss(loss_cls_1, loss_cls_
↪2)
>>> print(loss_cls)
2 - x
>>> loss = loss_cls.eval({"x": 4})
>>> print(loss)
```

(continues on next page)

(continued from previous page)

```
tensor(-2.)
>>> loss_cls = loss_cls_2 - loss_cls_1  # equals to SubLoss(loss_cls_2, loss_cls_
↪1)
>>> print(loss_cls)
x - 2
>>> loss = loss_cls.eval({"x": 4})
>>> print(loss)
tensor(2.)
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.5 MulLoss

class pixyz.losses.losses.**MulLoss** (*loss1*, *loss2*)

Bases: *pixyz.losses.losses.LossOperator*

Apply the *mul* operation to the two losses.

Examples

```
>>> loss_cls_1 = ValueLoss(2)
>>> loss_cls_2 = Parameter("x")
>>> loss_cls = loss_cls_1 * loss_cls_2  # equals to MulLoss(loss_cls_1, loss_cls_
↪2)
>>> print(loss_cls)
2 x
>>> loss = loss_cls.eval({"x": 4})
>>> print(loss)
tensor(8.)
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.6 DivLoss

class pixyz.losses.losses.**DivLoss** (*loss1*, *loss2*)

Bases: *pixyz.losses.losses.LossOperator*

Apply the *div* operation to the two losses.

Examples

```
>>> loss_cls_1 = ValueLoss(2)
>>> loss_cls_2 = Parameter("x")
>>> loss_cls = loss_cls_1 / loss_cls_2  # equals to DivLoss(loss_cls_1, loss_cls_
↪2)
>>> print(loss_cls)
\frac{2}{x}
>>> loss = loss_cls.eval({"x": 4})
>>> print(loss)
tensor(0.5000)
>>> loss_cls = loss_cls_2 / loss_cls_1  # equals to DivLoss(loss_cls_2, loss_cls_
↪1)
>>> print(loss_cls)
\frac{x}{2}
>>> loss = loss_cls.eval({"x": 4})
>>> print(loss)
tensor(2.)
```

forward(*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.7 MinLoss

class pixyz.losses.losses.**MinLoss**(*loss1*, *loss2*)

Bases: *pixyz.losses.losses.LossOperator*

Apply the *min* operation to the loss.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> from pixyz.losses.losses import ValueLoss, Parameter, MinLoss
>>> loss_min= MinLoss(ValueLoss(3), ValueLoss(1))
>>> print(loss_min)
min \left(3, 1\right)
>>> print(loss_min.eval())
tensor(1.)
```

forward(*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.8 MaxLoss

class pixyz.losses.losses.**MaxLoss**(*loss1*, *loss2*)

Bases: *pixyz.losses.losses.LossOperator*

Apply the *max* operation to the loss.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> from pixyz.losses.losses import ValueLoss, MaxLoss
>>> loss_max= MaxLoss(ValueLoss(3), ValueLoss(1))
>>> print(loss_max)
max \left(3, 1\right)
>>> print(loss_max.eval())
tensor(3.)
```

forward(*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.10.9 NegLoss

class pixyz.losses.losses.**NegLoss**(*loss1*)

Bases: *pixyz.losses.losses.LossSelfOperator*

Apply the *neg* operation to the loss.

Examples

```
>>> loss_cls_1 = Parameter("x")
>>> loss_cls = -loss_cls_1  # equals to NegLoss(loss_cls_1)
>>> print(loss_cls)
- x
>>> loss = loss_cls.eval({"x": 4})
>>> print(loss)
-4
```

forward(*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.10.10 AbsLoss

class pixyz.losses.losses.**AbsLoss**(*loss1*)

Bases: *pixyz.losses.losses.LossSelfOperator*

Apply the *abs* operation to the loss.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> from pixyz.losses import LogProb
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10])
>>> loss_cls = LogProb(p).abs() # equals to AbsLoss(LogProb(p))
>>> print(loss_cls)
|\log p(x)|
>>> sample_x = torch.randn(2, 10) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor([12.9894, 15.5280])
```

forward(*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of pixyz.losses.Loss and dict
- *deterministically calcurated loss and updated all samples.*

2.10.11 BatchMean

class pixyz.losses.losses.**BatchMean**(*loss1*)

Bases: *pixyz.losses.losses.LossSelfOperator*

Average a loss class over given batch data.

$$\mathbb{E}_{p_{data}(x)}[\mathcal{L}(x)] \approx \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(x_i),$$

where $x_i \sim p_{data}(x)$ and $\mathcal{L}$ is a loss function.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> from pixyz.losses import LogProb
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10])
>>> loss_cls = LogProb(p).mean() # equals to BatchMean(LogProb(p))
>>> print(loss_cls)
mean \left(\log p(x) \right)
>>> sample_x = torch.randn(2, 10) # Psuedo data
```

(continues on next page)

(continued from previous page)

```
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor(-14.5038)
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.12 BatchSum

class pixyz.losses.losses.**BatchSum** (*loss1*)

Bases: *pixyz.losses.losses.LossSelfOperator*

Summation a loss class over given batch data.

$$\sum_{i=1}^{N} \mathcal{L}(x_i),$$

where $x_i \sim p_{data}(x)$ and $\mathcal{L}$ is a loss function.

Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> from pixyz.losses import LogProb
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...            features_shape=[10])
>>> loss_cls = LogProb(p).sum() # equals to BatchSum(LogProb(p))
>>> print(loss_cls)
sum \left(\log p(x) \right)
>>> sample_x = torch.randn(2, 10) # Psuedo data
>>> loss = loss_cls.eval({"x": sample_x})
>>> print(loss) # doctest: +SKIP
tensor(-31.9434)
```

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.13 Detach

class pixyz.losses.losses.**Detach** (*loss1*)

Bases: *pixyz.losses.losses.LossSelfOperator*

Apply the *detach* method to the loss.

forward (*x_dict={}*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

2.10.14 DataParalleledLoss

class `pixyz.losses.losses.`**DataParalleledLoss** (*loss*, *distributed=False*, ***kwargs*)

Bases: *pixyz.losses.losses.Loss*

Loss class wrapper of torch.nn.DataParallel. It can be used as the original loss class. *eval* & *forward* methods support data-parallel running.

Examples

```
>>> import torch
>>> from torch import optim
>>> from torch.nn import functional as F
>>> from pixyz.distributions import Bernoulli, Normal
>>> from pixyz.losses import KullbackLeibler, DataParalleledLoss
>>> from pixyz.models import Model
>>> used_gpu_i = set()
>>> used_gpu_g = set()
>>> # Set distributions (Distribution API)
>>> class Inference(Normal):
...     def __init__(self):
...         super().__init__(var=["z"],cond_var=["x"],name="q")
...         self.model_loc = torch.nn.Linear(128, 64)
...         self.model_scale = torch.nn.Linear(128, 64)
...     def forward(self, x):
...         used_gpu_i.add(x.device.index)
...         return {"loc": self.model_loc(x), "scale": F.softplus(self.model_
↪scale(x))}
>>> class Generator(Bernoulli):
...     def __init__(self):
...         super().__init__(var=["x"],cond_var=["z"],name="p")
...         self.model = torch.nn.Linear(64, 128)
...     def forward(self, z):
...         used_gpu_g.add(z.device.index)
...         return {"probs": torch.sigmoid(self.model(z))}
>>> p = Generator()
>>> q = Inference()
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[64], name="p_{prior}")
>>> # Define a loss function (Loss API)
>>> reconst = -p.log_prob().expectation(q)
>>> kl = KullbackLeibler(q,prior)
>>> batch_loss_cls = (reconst - kl)
>>> # device settings
>>> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
>>> device_count = torch.cuda.device_count()
>>> if device_count > 1:
```

(continues on next page)

(continued from previous page)

```
...     loss_cls = DataParalleledLoss(batch_loss_cls).mean().to(device)
... else:
...     loss_cls = batch_loss_cls.mean().to(device)
>>> # Set a model (Model API)
>>> model = Model(loss=loss_cls, distributions=[p, q],
...               optimizer=optim.Adam, optimizer_params={"lr": 1e-3})
>>> # Train and test the model
>>> data = torch.randn(2, 128).to(device)  # Pseudo data
>>> train_loss = model.train({"x": data})
>>> expected = set(range(device_count)) if torch.cuda.is_available() else {None}
>>> assert used_gpu_i==expected
>>> assert used_gpu_g==expected
```

forward(*x_dict*, ***kwargs*)

Parameters **x_dict** (*dict*) – Input variables.

Returns

- a tuple of `pixyz.losses.Loss` and dict
- *deterministically calcurated loss and updated all samples.*

CHAPTER 3

pixyz.models (Model API)

3.1 Model

class pixyz.models.**Model**(*loss*, *test_loss=None*, *distributions=[]*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Bases: object

This class is for training and testing a loss class. It requires a defined loss class, distributions to train, and optimizer for initialization.

Examples

```
>>> import torch
>>> from torch import optim
>>> from torch.nn import functional as F
>>> from pixyz.distributions import Bernoulli, Normal
>>> from pixyz.losses import KullbackLeibler
...
>>> # Set distributions (Distribution API)
>>> class Inference(Normal):
...     def __init__(self):
...         super().__init__(var=["z"],cond_var=["x"],name="q")
...         self.model_loc = torch.nn.Linear(128, 64)
...         self.model_scale = torch.nn.Linear(128, 64)
...     def forward(self, x):
...         return {"loc": self.model_loc(x), "scale": F.softplus(self.model_
↪scale(x))}
...
>>> class Generator(Bernoulli):
...     def __init__(self):
...         super().__init__(var=["x"],cond_var=["z"],name="p")
...         self.model = torch.nn.Linear(64, 128)
```

(continues on next page)

(continued from previous page)

```
...     def forward(self, z):
...         return {"probs": torch.sigmoid(self.model(z))}
...
>>> p = Generator()
>>> q = Inference()
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[64], name="p_{prior}")
...
>>> # Define a loss function (Loss API)
>>> reconst = -p.log_prob().expectation(q)
>>> kl = KullbackLeibler(q,prior)
>>> loss_cls = (reconst - kl).mean()
>>> print(loss_cls)
mean \left(- D_{KL} \left[q(z|x)||p_{prior}(z) \right] - \mathbb{E}_{q(z|x)}␣
↪\left[\log p(x|z) \right] \right)
>>>
>>> # Set a model (Model API)
>>> model = Model(loss=loss_cls, distributions=[p, q],
...               optimizer=optim.Adam, optimizer_params={"lr": 1e-3})
>>> # Train and test the model
>>> data = torch.randn(1, 128)  # Pseudo data
>>> train_loss = model.train({"x": data})
>>> test_loss = model.test({"x": data})
```

__init__(*loss*, *test_loss=None*, *distributions=[]*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Parameters

- **loss** (*pixyz.losses.Loss*) – Loss class for training.
- **test_loss** (*pixyz.losses.Loss*) – Loss class for testing.
- **distributions** (*list*) – List of `pixyz.distributions.Distribution`.
- **optimizer** (*torch.optim*) – Optimization algorithm.
- **optimizer_params** (*dict*) – Parameters of optimizer
- **clip_grad_norm** (*float or int*) – Maximum allowed norm of the gradients.
- **clip_grad_value** (*float or int*) – Maximum allowed value of the gradients.

set_loss(*loss*, *test_loss=None*)

train(*train_x_dict={}*, ***kwargs*)

Train the model.

Parameters

- **train_x_dict** (*dict*) – Input data.
- ****kwargs** –

Returns **loss** – Train loss value

Return type torch.Tensor

test(*test_x_dict={}*, ***kwargs*)

Test the model.

Parameters

- **test_x_dict** (*dict*) – Input data

- ****kwargs** –

Returns **loss** – Test loss value

Return type torch.Tensor

3.2 Pre-implementation models

3.2.1 ML

class pixyz.models.**ML**(*p*, *other_distributions=[]*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=False*, *clip_grad_value=False*)

Bases: pixyz.models.model.Model

Maximum Likelihood (log-likelihood)

The negative log-likelihood of a given distribution (p) is set as the loss class of this model.

__init__(*p*, *other_distributions=[]*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=False*, *clip_grad_value=False*)

Parameters

- **p** (*torch.distributions.Distribution*) – Classifier (distribution).
- **optimizer** (*torch.optim*) – Optimization algorithm.
- **optimizer_params** (*dict*) – Parameters of optimizer
- **clip_grad_norm** (*float or int*) – Maximum allowed norm of the gradients.
- **clip_grad_value** (*float or int*) – Maximum allowed value of the gradients.

train(*train_x_dict={}*, ***kwargs*)

Train the model.

Parameters

- **train_x_dict** (*dict*) – Input data.
- ****kwargs** –

Returns **loss** – Train loss value

Return type torch.Tensor

test(*test_x_dict={}*, ***kwargs*)

Test the model.

Parameters

- **test_x_dict** (*dict*) – Input data
- ****kwargs** –

Returns **loss** – Test loss value

Return type torch.Tensor

3.2.2 VAE

class pixyz.models.**VAE**(*encoder*, *decoder*, *other_distributions=[]*, *regularizer=None*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Bases: `pixyz.models.model.Model`

Variational Autoencoder.

In VAE class, reconstruction loss on given distributions (encoder and decoder) is set as the default loss class. However, if you want to add additional terms, e.g., the KL divergence between encoder and prior, you need to set them to the *regularizer* argument, which defaults to None.

References

[Kingma+ 2013] Auto-Encoding Variational Bayes

__init__(*encoder*, *decoder*, *other_distributions=[]*, *regularizer=None*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Parameters

- **encoder** (*torch.distributions.Distribution*) – Encoder distribution.
- **decoder** (*torch.distributions.Distribution*) – Decoder distribution.
- **regularizer** (*torch.losses.Loss, defaults to None*) – If you want to add additional terms to the loss, set them to this argument.
- **optimizer** (*torch.optim*) – Optimization algorithm.
- **optimizer_params** (*dict*) – Parameters of optimizer
- **clip_grad_norm** (*float or int*) – Maximum allowed norm of the gradients.
- **clip_grad_value** (*float or int*) – Maximum allowed value of the gradients.

train(*train_x_dict={}*, ***kwargs*)

Train the model.

Parameters

- **train_x_dict** (*dict*) – Input data.
- ****kwargs** –

Returns **loss** – Train loss value

Return type torch.Tensor

test(*test_x_dict={}*, ***kwargs*)

Test the model.

Parameters

- **test_x_dict** (*dict*) – Input data
- ****kwargs** –

Returns **loss** – Test loss value

Return type torch.Tensor

3.2.3 VI

class pixyz.models.**VI**(*p*, *approximate_dist*, *other_distributions=[]*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Bases: `pixyz.models.model.Model`

Variational Inference (Amortized inference)

The ELBO for given distributions (p, approximate_dist) is set as the loss class of this model.

__init__(*p*, *approximate_dist*, *other_distributions=[]*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Parameters

- **p** (*torch.distributions.Distribution*) – Generative model (distribution).
- **approximate_dist** (*torch.distributions.Distribution*) – Approximate posterior distribution.
- **optimizer** (*torch.optim*) – Optimization algorithm.
- **optimizer_params** (*dict*) – Parameters of optimizer
- **clip_grad_norm** (*float or int*) – Maximum allowed norm of the gradients.
- **clip_grad_value** (*float or int*) – Maximum allowed value of the gradients.

train(*train_x_dict={}*, ***kwargs*)

Train the model.

Parameters

- **train_x_dict** (*dict*) – Input data.
- ****kwargs** –

Returns **loss** – Train loss value

Return type torch.Tensor

test(*test_x_dict={}*, ***kwargs*)

Test the model.

Parameters

- **test_x_dict** (*dict*) – Input data
- ****kwargs** –

Returns **loss** – Test loss value

Return type torch.Tensor

3.2.4 GAN

class pixyz.models.**GAN**(*p*, *discriminator*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *d_optimizer=<class 'torch.optim.adam.Adam'>*, *d_optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Bases: `pixyz.models.model.Model`

Generative Adversarial Network

(Adversarial) Jensen-Shannon divergence between given distributions (p_data, p) is set as the loss class of this model.

Examples

```
>>> import torch
>>> from torch import nn, optim
>>> from pixyz.distributions import Deterministic
>>> from pixyz.distributions import Normal
>>> from pixyz.models import GAN
>>> from pixyz.utils import print_latex
>>> x_dim = 128
>>> z_dim = 100
...
>>> # Set distributions (Distribution API)
...
>>> # generator model p(x|z)
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(var=["x"], cond_var=["z"], name="p")
...         self.model = nn.Sequential(
...             nn.Linear(z_dim, x_dim),
...             nn.Sigmoid()
...         )
...     def forward(self, z):
...         x = self.model(z)
...         return {"x": x}
...
>>> # prior model p(z)
>>> prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
...                var=["z"], features_shape=[z_dim], name="p_{prior}")
...
>>> # generative model
>>> p_g = Generator()
>>> p = (p_g*prior).marginalize_var("z")
...
>>> # discriminator model p(t|x)
>>> class Discriminator(Deterministic):
...     def __init__(self):
...         super(Discriminator, self).__init__(var=["t"], cond_var=["x"], name="d
↪")
...         self.model = nn.Sequential(
...             nn.Linear(x_dim, 1),
...             nn.Sigmoid()
...         )
...     def forward(self, x):
...         t = self.model(x)
...         return {"t": t}
...
>>> d = Discriminator()
>>> # Set a model (Model API)
>>> model = GAN(p, d, optimizer_params={"lr":0.0002}, d_optimizer_params={"lr":0.
↪0002})
>>> print(model)
Distributions (for training):
  p(x)
Loss function:
  mean(D_{JS}^{Adv} \left[p_{data}(x)||p(x) \right])
Optimizer:
  Adam (
```

(continues on next page)

(continued from previous page)

```
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 0.0002
    weight_decay: 0
)
>>> # Train and test the model
>>> data = torch.randn(1, x_dim)  # Pseudo data
>>> train_loss = model.train({"x": data})
>>> test_loss = model.test({"x": data})
```

__init__(*p*, *discriminator*, *optimizer=<class 'torch.optim.adam.Adam'>*, *optimizer_params={}*, *d_optimizer=<class 'torch.optim.adam.Adam'>*, *d_optimizer_params={}*, *clip_grad_norm=None*, *clip_grad_value=None*)

Parameters

- **p** (*torch.distributions.Distribution*) – Generative model (generator).
- **discriminator** (*torch.distributions.Distribution*) – Critic (discriminator).
- **optimizer** (*torch.optim*) – Optimization algorithm.
- **optimizer_params** (*dict*) – Parameters of optimizer
- **clip_grad_norm** (*float or int*) – Maximum allowed norm of the gradients.
- **clip_grad_value** (*float or int*) – Maximum allowed value of the gradients.

train(*train_x_dict={}*, *adversarial_loss=True*, ***kwargs*)
Train the model.

Parameters

- **train_x_dict** (*dict, defaults to {}*) – Input data.
- **adversarial_loss** (*bool, defaults to True*) – Whether to train the discriminator.
- ****kwargs** –

Returns

- **loss** (*torch.Tensor*) – Train loss value.
- **d_loss** (*torch.Tensor*) – Train loss value of the discriminator (if `adversarial_loss` is True).

test(*test_x_dict={}*, *adversarial_loss=True*, ***kwargs*)
Train the model.

Parameters

- **test_x_dict** (*dict, defaults to {}*) – Input data.
- **adversarial_loss** (*bool, defaults to True*) – Whether to return the discriminator loss.
- ****kwargs** –

Returns

- **loss** (*torch.Tensor*) – Test loss value.

- **d_loss** (*torch.Tensor*) – Test loss value of the discriminator (if `adversarial_loss` is True).

CHAPTER 4

pixyz.flows (Flow layers)

4.1 Flow

class pixyz.flows.**Flow**(*in_features*)
Bases: torch.nn.modules.module.Module

Flow class. In Pixyz, all flows are required to inherit this class.

__init__(*in_features*)

Parameters **in_features** (*int*) – Size of input data.

in_features

forward(*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in *logdet_jacobian*.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

logdet_jacobian
Get log-determinant Jacobian.

Before calling this, you should run *forward* or `update_jacobian` methods to calculate and store log-determinant Jacobian.

class `pixyz.flows.`**FlowList**(*flow_list*)
Bases: `pixyz.flows.flows.Flow`

__init__(*flow_list*)
Hold flow modules in a list.

Once initializing, it can be handled as a single flow module.

Notes

Indexing is not supported for now.

Parameters **flow_list** (*list*) –

forward(*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.2 Normalizing flow

4.2.1 PlanarFlow

class `pixyz.flows.`**PlanarFlow**(*in_features*, *constraint_u=False*)
Bases: `pixyz.flows.flows.Flow`

Planar flow.

$$f(\mathbf{x}) = \mathbf{x} + \mathbf{u}h(\mathbf{w}^T\mathbf{x} + \mathbf{b})$$

deriv_tanh (*x*)

reset_parameters ()

forward (*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

extra_repr ()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

4.3 Coupling layer

4.3.1 AffineCoupling

class pixyz.flows.**AffineCoupling** (*in_features*, *mask_type='channel_wise'*, *scale_net=None*, *translate_net=None*, *scale_translate_net=None*, *inverse_mask=False*)

Bases: `pixyz.flows.flows.Flow`

Affine coupling layer

$$\begin{aligned} \mathbf{y}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{y}_{d+1:D} &= \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d}) + t(\mathbf{x}_{1:d})) \end{aligned}$$

build_mask (*x*)

Parameters **x** (*torch.Tensor*) –

Returns mask

Return type torch.tensor

Examples

```
>>> scale_translate_net = lambda x: (x, x)
>>> f1 = AffineCoupling(4, mask_type="channel_wise", scale_translate_
↪net=scale_translate_net,
...                         inverse_mask=False)
>>> x1 = torch.randn([1,4,3,3])
>>> f1.build_mask(x1)
tensor([[[[1.]],
<BLANKLINE>
         [[1.]],
<BLANKLINE>
         [[0.]],
<BLANKLINE>
         [[0.]]]])
>>> f2 = AffineCoupling(2, mask_type="checkerboard", scale_translate_
↪net=scale_translate_net,
...                         inverse_mask=True)
>>> x2 = torch.randn([1,2,5,5])
>>> f2.build_mask(x2)
tensor([[[[0., 1., 0., 1., 0.],
          [1., 0., 1., 0., 1.],
          [0., 1., 0., 1., 0.],
          [1., 0., 1., 0., 1.],
          [0., 1., 0., 1., 0.]]]])
```

get_parameters (*x*, *y=None*)

Parameters

- **x** (*torch.tensor*) –
- **y** (*torch.tensor*) –

Returns

- **s** (*torch.tensor*)
- **t** (*torch.tensor*)

Examples

```
>>> # In case of using scale_translate_net
>>> scale_translate_net = lambda x: (x, x)
>>> f1 = AffineCoupling(4, mask_type="channel_wise", scale_translate_
↪net=scale_translate_net,
...                         inverse_mask=False)
>>> x1 = torch.randn([1,4,3,3])
>>> log_s, t = f1.get_parameters(x1)
>>> # In case of using scale_net and translate_net
```

(continues on next page)

(continued from previous page)

```
>>> scale_net = lambda x: x
>>> translate_net = lambda x: x
>>> f2 = AffineCoupling(4, mask_type="channel_wise", scale_net=scale_net,␣
↪translate_net=translate_net,
...                         inverse_mask=False)
>>> x2 = torch.randn([1,4,3,3])
>>> log_s, t = f2.get_parameters(x2)
```

forward(*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

4.4 Invertible layer

4.4.1 ChannelConv

class pixyz.flows.**ChannelConv**(*in_channels*, *decomposed=False*)
Bases: `pixyz.flows.flows.Flow`

Invertible 1×1 convolution.

Notes

This is implemented with reference to the following code. https://github.com/chaiyujin/glow-pytorch/blob/master/glow/modules.py

get_parameters(*x*, *inverse*)

forward (*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*x*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5 Operation layer

4.5.1 Squeeze

class pixyz.flows.**Squeeze**
Bases: `pixyz.flows.flows.Flow`

Squeeze operation.

c * s * s -> 4c * s/2 * s/2

Examples

```
>>> import torch
>>> a = torch.tensor([i+1 for i in range(16)]).view(1,1,4,4)
>>> print(a)
tensor([[[[ 1,  2,  3,  4],
          [ 5,  6,  7,  8],
          [ 9, 10, 11, 12],
          [13, 14, 15, 16]]]])
>>> f = Squeeze()
>>> print(f(a))
tensor([[[[ 1,  3],
          [ 9, 11]],
<BLANKLINE>
         [[ 2,  4],
          [10, 12]],
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
         [[ 5,  7],
          [13, 15]],
<BLANKLINE>
         [[ 6,  8],
          [14, 16]]]])
```

```
>>> print(f.inverse(f(a)))
tensor([[[[ 1,  2,  3,  4],
          [ 5,  6,  7,  8],
          [ 9, 10, 11, 12],
          [13, 14, 15, 16]]]])
```

forward (*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5.2 Unsqueeze

class pixyz.flows.**Unsqueeze**
Bases: `pixyz.flows.operations.Squeeze`

Unsqueeze operation.

c * s * s -> c/4 * 2s * 2s

Examples

```
>>> import torch
>>> a = torch.tensor([i+1 for i in range(16)]).view(1,4,2,2)
>>> print(a)
tensor([[[[ 1,  2],
          [ 3,  4]],
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
         [[ 5,  6],
          [ 7,  8]],
<BLANKLINE>
         [[ 9, 10],
          [11, 12]],
<BLANKLINE>
         [[13, 14],
          [15, 16]]]])
>>> f = Unsqueeze()
>>> print(f(a))
tensor([[[[ 1,  5,  2,  6],
          [ 9, 13, 10, 14],
          [ 3,  7,  4,  8],
          [11, 15, 12, 16]]]])
>>> print(f.inverse(f(a)))
tensor([[[[ 1,  2],
          [ 3,  4]],
<BLANKLINE>
         [[ 5,  6],
          [ 7,  8]],
<BLANKLINE>
         [[ 9, 10],
          [11, 12]],
<BLANKLINE>
         [[13, 14],
          [15, 16]]]])
```

forward(*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5.3 Permutation

class pixyz.flows.**Permutation**(*permute_indices*)

Bases: pixyz.flows.flows.Flow

Examples

```
>>> import torch
>>> a = torch.tensor([i+1 for i in range(16)]).view(1,4,2,2)
>>> print(a)
tensor([[[[ 1,  2],
          [ 3,  4]],
<BLANKLINE>
         [[ 5,  6],
          [ 7,  8]],
<BLANKLINE>
         [[ 9, 10],
          [11, 12]],
<BLANKLINE>
         [[13, 14],
          [15, 16]]]])
>>> perm = [0,3,1,2]
>>> f = Permutation(perm)
>>> f(a)
tensor([[[[ 1,  2],
          [ 3,  4]],
<BLANKLINE>
         [[13, 14],
          [15, 16]],
<BLANKLINE>
         [[ 5,  6],
          [ 7,  8]],
<BLANKLINE>
         [[ 9, 10],
          [11, 12]]]])
>>> f.inverse(f(a))
tensor([[[[ 1,  2],
          [ 3,  4]],
<BLANKLINE>
         [[ 5,  6],
          [ 7,  8]],
<BLANKLINE>
         [[ 9, 10],
          [11, 12]],
<BLANKLINE>
         [[13, 14],
          [15, 16]]]])
```

forward(*x*, *y=None*, *compute_jacobian=True*)

Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and

store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5.4 Shuffle

class pixyz.flows.**Shuffle** (*in_features*)
Bases: `pixyz.flows.operations.Permutation`

4.5.5 Reverse

class pixyz.flows.**Reverse** (*in_features*)
Bases: `pixyz.flows.operations.Permutation`

4.5.6 Flatten

class pixyz.flows.**Flatten** (*in_size=None*)
Bases: `pixyz.flows.flows.Flow`

forward (*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5.7 BatchNorm1d

class pixyz.flows.**BatchNorm1d**(*in_features*, *momentum=0.0*)

Bases: `pixyz.flows.flows.Flow`

A batch normalization with the inverse transformation.

Notes

This is implemented with reference to the following code. https://github.com/ikostrikov/pytorch-flows/blob/master/flows.py#L205

Examples

```
>>> x = torch.randn(20, 100)
>>> f = BatchNorm1d(100)
>>> # transformation
>>> z = f(x)
>>> # reconstruction
>>> _x = f.inverse(f(x))
>>> # check this reconstruction
>>> diff = torch.sum(torch.abs(_x-x)).item()
>>> diff < 0.1
True
```

forward(*x*, *y=None*, *compute_jacobian=True*)

Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse(*z*, *y=None*)

Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5.8 BatchNorm2d

class `pixyz.flows.`**`BatchNorm2d`**(*in_features*, *momentum=0.0*)

Bases: `pixyz.flows.normalizations.BatchNorm1d`

A batch normalization with the inverse transformation.

Notes

This is implemented with reference to the following code. https://github.com/ikostrikov/pytorch-flows/blob/master/flows.py#L205

Examples

```
>>> x = torch.randn(20, 100, 35, 45)
>>> f = BatchNorm2d(100)
>>> # transformation
>>> z = f(x)
>>> # reconstruction
>>> _x = f.inverse(f(x))
>>> # check this reconstruction
>>> diff = torch.sum(torch.abs(_x-x)).item()
>>> diff < 0.1
True
```

4.5.9 ActNorm2d

class `pixyz.flows.`**`ActNorm2d`**(*in_features*, *scale=1.0*)

Bases: `pixyz.flows.flows.Flow`

Activation Normalization Initialize the bias and scale with a given minibatch, so that the output per-channel have zero mean and unit variance for that. After initialization, *bias* and *logs* will be trained as parameters.

Notes

This is implemented with reference to the following code. https://github.com/chaiyujin/glow-pytorch/blob/master/glow/modules.py

`initialize_parameters`(*x*)

`forward`(*x*, *y=None*, *compute_jacobian=True*)

Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*x*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

4.5.10 Preprocess

class pixyz.flows.**Preprocess**
Bases: `pixyz.flows.flows.Flow`

static logit (*x*)

forward (*x*, *y=None*, *compute_jacobian=True*)
Forward propagation of flow layers.

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.
- **compute_jacobian** (*bool, defaults to True*) – Whether to calculate and store log-determinant Jacobian. If true, calculated Jacobian values are stored in `logdet_jacobian`.

Returns z

Return type torch.Tensor

inverse (*z*, *y=None*)
Backward (inverse) propagation of flow layers. In this method, log-determinant Jacobian is not calculated.

Parameters

- **z** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor, defaults to None*) – Data for conditioning.

Returns x

Return type torch.Tensor

CHAPTER 5

pixyz.utils

pixyz.utils.**set_epsilon**(*eps*)

Set a *epsilon* parameter.

Parameters **eps** (*int or float*) –

Examples

```
>>> from unittest import mock
>>> with mock.patch('pixyz.utils._EPSILON', 1e-07):
...     set_epsilon(1e-06)
...     epsilon()
1e-06
```

pixyz.utils.**epsilon**()

Get a *epsilon* parameter.

Returns

Return type int or float

Examples

```
>>> from unittest import mock
>>> with mock.patch('pixyz.utils._EPSILON', 1e-07):
...     epsilon()
1e-07
```

pixyz.utils.**get_dict_values**(*dicts*, *keys*, *return_dict=False*)

Get values from *dicts* specified by *keys*.

When *return_dict* is True, return values are in dictionary format.

Parameters

- **dicts** (*dict*) –
- **keys** (*list*) –
- **return_dict** (*bool*) –

Returns

Return type dict or list

Examples

```
>>> get_dict_values({"a":1,"b":2,"c":3}, ["b"])
[2]
>>> get_dict_values({"a":1,"b":2,"c":3}, ["b", "d"], True)
{'b': 2}
```

pixyz.utils.**delete_dict_values**(*dicts*, *keys*)

Delete values from *dicts* specified by *keys*.

Parameters

- **dicts** (*dict*) –
- **keys** (*list*) –

Returns **new_dicts**

Return type dict

Examples

```
>>> delete_dict_values({"a":1,"b":2,"c":3}, ["b","d"])
{'a': 1, 'c': 3}
```

pixyz.utils.**detach_dict**(*dicts*)

Detach all values in *dicts*.

Parameters **dicts** (*dict*) –

Returns

Return type dict

pixyz.utils.**replace_dict_keys**(*dicts*, *replace_list_dict*)

Replace values in *dicts* according to *replace_list_dict*.

Parameters

- **dicts** (*dict*) – Dictionary.
- **replace_list_dict** (*dict*) – Dictionary.

Returns **replaced_dicts** – Dictionary.

Return type dict

Examples

```
>>> replace_dict_keys({"a":1,"b":2,"c":3}, {"a":"x","b":"y"})
{'x': 1, 'y': 2, 'c': 3}
>>> replace_dict_keys({"a":1,"b":2,"c":3}, {"a":"x","e":"y"})  # keys of `replace_
↪list_dict`
{'x': 1, 'b': 2, 'c': 3}
```

`pixyz.utils.`**`replace_dict_keys_split`**(*dicts*, *replace_list_dict*)

Replace values in *dicts* according to `replace_list_dict`.

Replaced dict is splitted by `replaced_dict` and `remain_dict`.

Parameters

- **dicts** (*dict*) – Dictionary.
- **replace_list_dict** (*dict*) – Dictionary.

Returns

- **replaced_dict** (*dict*) – Dictionary.
- **remain_dict** (*dict*) – Dictionary.

Examples

```
>>> replace_list_dict = {'a': 'loc'}
>>> x_dict = {'a': 0, 'b': 1}
>>> print(replace_dict_keys_split(x_dict, replace_list_dict))
({'loc': 0}, {'b': 1})
```

class `pixyz.utils.`**`FrozenSampleDict`**(*dict_*)

Bases: `object`

`pixyz.utils.`**`lru_cache_for_sample_dict`**(*maxsize=0*)

Memoize the calculation result linked to the argument of sample dict. Note that dictionary arguments of the target function must be sample dict.

Parameters **maxsize** (*cache size prepared for the target method*) –

Returns

Return type decorator function

Examples

```
>>> import time
>>> import torch.nn as nn
>>> import pixyz.utils as utils
>>> # utils.CACHE_SIZE = 2  # you can also use this module option to enable all␣
↪memoization of distribution
>>> import pixyz.distributions as pd
>>> class LongEncoder(pd.Normal):
...     def __init__(self):
...         super().__init__(var=['x'], cond_var=['y'])
...         self.nn = nn.Sequential(*(nn.Linear(1,1) for i in range(10000)))
...     def forward(self, y):
```

(continues on next page)

(continued from previous page)

```
...         return {'loc': self.nn(y), 'scale': torch.ones(1,1)}
...     @lru_cache_for_sample_dict(maxsize=2)
...     def get_params(self, params_dict={}, **kwargs):
...         return super().get_params(params_dict, **kwargs)
>>> def measure_time(func):
...     start = time.time()
...     func()
...     elapsed_time = time.time() - start
...     return elapsed_time
>>> le = LongEncoder()
>>> y = torch.ones(1, 1)
>>> t_sample1 = measure_time(lambda:le.sample({'y': y}))
>>> print ("sample1:{0}".format(t_sample1) + "[sec]") # doctest: +SKIP
>>> t_log_prob = measure_time(lambda:le.get_log_prob({'x': y, 'y': y}))
>>> print ("log_prob:{0}".format(t_log_prob) + "[sec]") # doctest: +SKIP
>>> t_sample2 = measure_time(lambda:le.sample({'y': y}))
>>> print ("sample2:{0}".format(t_sample2) + "[sec]") # doctest: +SKIP
>>> assert t_sample1 > t_sample2, "processing time increases: {0}".format(t_
↪sample2 - t_sample1)
```

pixyz.utils.**tolist**(*a*)

Convert a given input to the dictionary format.

Parameters **a** (*list or other*) –

Returns

Return type list

Examples

```
>>> tolist(2)
[2]
>>> tolist([1, 2])
[1, 2]
>>> tolist([])
[]
```

pixyz.utils.**sum_samples**(*samples*)

Sum a given sample across the axes.

Parameters **samples** (*torch.Tensor*) – Input sample. The number of this axes is assumed to be 4 or less.

Returns Sum over all axes except the first axis.

Return type torch.Tensor

Examples

```
>>> a = torch.ones([2])
>>> sum_samples(a).size()
torch.Size([2])
>>> a = torch.ones([2, 3])
>>> sum_samples(a).size()
```

(continues on next page)

(continued from previous page)

```
torch.Size([2])
>>> a = torch.ones([2, 3, 4])
>>> sum_samples(a).size()
torch.Size([2])
```

pixyz.utils.**print_latex**(*obj*)
Print formulas in latex format.

Parameters **obj** (pixyz.distributions.distributions.Distribution, pixyz.losses.losses.Loss *or* *pixyz.models.model.Model.*) –

pixyz.utils.**convert_latex_name**(*name*)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W