

---

# Pixyz Documentation

**masa**

**Jun 11, 2019**



---

## Package Reference

---

<b>1</b>	<b>pixyz.distributions (Distribution API)</b>	<b>3</b>
<b>2</b>	<b>pixyz.losses (Loss API)</b>	<b>51</b>
<b>3</b>	<b>pixyz.models (Model API)</b>	<b>73</b>
<b>4</b>	<b>pixyz.flows (Flow layers)</b>	<b>79</b>
<b>5</b>	<b>pixyz.utils</b>	<b>93</b>
<b>6</b>	<b>Indices and tables</b>	<b>97</b>
	<b>Python Module Index</b>	<b>99</b>
	<b>Index</b>	<b>101</b>



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



---

pixyz.distributions (Distribution API)

---

## 1.1 Distribution

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

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)
Distribution:
```

(continues on next page)

(continued from previous page)

```

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([]))
```

#### 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.

#### 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

**get\_params** (*params\_dict*={})

This method aims to get parameters of this distributions from constant parameters set in initialization and outputs of DNNs.

**Parameters** *params\_dict* (dict, defaults to {}) – Input parameters.

**Returns** *output\_dict* – Output parameters.

**Return type** dict

## Examples

```
>>> from pixyz.distributions import Normal
>>> dist_1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...                 features_shape=[1])
>>> print(dist_1)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([1])
    (loc): torch.Size([1, 1])
    (scale): torch.Size([1, 1])
  )
>>> dist_1.get_params()
{'loc': tensor([[0.]]) , 'scale': tensor([[1.]])}
```

```
>>> dist_2 = Normal(loc=torch.tensor(0.), scale="z", cond_var=["z"], var=["x
↪"])
>>> print(dist_2)
Distribution:
  p(x|z)
Network architecture:
```

(continues on next page)

(continued from previous page)

```

Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (loc): torch.Size([1])
)
>>> dist_2.get_params({"z": torch.tensor(1.)})
{'scale': tensor(1.), 'loc': tensor([0.])}

```

**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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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])

```

(continues on next page)

(continued from previous page)

```

>>> 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]])

```

**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")
```

(continues on next page)

(continued from previous page)

```
>>> 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.LogProb`.

#### 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 overridden.

**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 reimplement 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 (cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
```

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

### 1.2.2 Laplace

```
class pixyz.distributions.Laplace (cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
```

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

### 1.2.3 Bernoulli

```
class pixyz.distributions.Bernoulli (cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
    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
```

### 1.2.4 RelaxedBernoulli

```
class pixyz.distributions.RelaxedBernoulli (temperature=tensor(0.1000),
                                           cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
    Bases: pixyz.distributions.exponential_distributions.Bernoulli

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

    temperature

    distribution_torch_class
        Return the class of PyTorch distribution.

    relaxed_distribution_torch_class
        Use relaxed version only when sampling

    distribution_name
        Name of this distribution class.

        Type str

    set_dist (x_dict={}, sampling=True, batch_n=None, **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.
        • sampling (bool, defaults to False.) – Choose whether to use relaxed_* in PyTorch distribution.
        • batch_n (int, defaults to None.) – Set batch size of parameters.
        • **kwargs – Arbitrary keyword arguments.
```



## 1.2.5 FactorizedBernoulli

```
class pixyz.distributions.FactorizedBernoulli (cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
```

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 (cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
    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
```

## 1.2.7 RelaxedCategorical

```
class pixyz.distributions.RelaxedCategorical (temperature=tensor(0.1000), cond_var=[], var=['x'], name='p', features_shape=torch.Size([]), **kwargs)
    Bases: pixyz.distributions.exponential_distributions.Categorical
    Relaxed (re-parameterizable) categorical distribution parameterized by probs.

    temperature

    distribution_torch_class
        Return the class of PyTorch distribution.

    relaxed_distribution_torch_class
        Use relaxed version only when sampling

    distribution_name
        Name of this distribution class.
        Type str

    set_dict (x_dict={}, sampling=True, batch_n=None, **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.
- sampling (bool, defaults to False.) – Choose whether to use relaxed_* in PyTorch distribution.
- batch_n (int, defaults to None.) – Set batch size of parameters.
- **kwargs – Arbitrary keyword arguments.

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.]])
```

### 1.2.8 Beta

**class** pixyz.distributions.**Beta** (*cond\_var*=[], *var*=['x'], *name*='p', *features\_shape*=torch.Size([]), *\*\*kwargs*)

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

## 1.2.9 Dirichlet

```
class pixyz.distributions.Dirichlet (cond_var=[], var=['x'], name='p', features_shape=torch.Size([ ]), **kwargs)
```

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

## 1.2.10 Gamma

```
class pixyz.distributions.Gamma (cond_var=[], var=['x'], name='p', features_shape=torch.Size([ ]), **kwargs)
```

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

## 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([2])
    (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])
      )
      (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\_text**

Return a formula of the (joint) probability distribution.

**Type** *str*

**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** (*batch\_n=None, sample\_shape=torch.Size([ ]), 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.
- **sample\_shape** (*list or NoneType*, defaults to *torch.Size()*) – Shape of generating samples.
- **batch\_n** (*int*, defaults to *None*.) – Set batch size of parameters.
- **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:
```

(continues on next page)

(continued from previous page)

```

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]])}

```

**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([])*, *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  $p(z|y)$  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.]]),
 '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.],
              [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., ..., 1., 0., 0.]])
```

(continues on next page)



(continued from previous page)

```

    [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.

#### **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*)

This method aims to get parameters of this distributions from constant parameters set in initialization and outputs of DNNs.

**Parameters** **params\_dict** (dict, defaults to {}) – Input parameters.

**Returns** **output\_dict** – Output parameters.

**Return type** dict

### Examples

```

>>> from pixyz.distributions import Normal
>>> dist_1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...                 features_shape=[1])
>>> print(dist_1)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([1])
    (loc): torch.Size([1, 1])
    (scale): torch.Size([1, 1])
```

(continues on next page)

(continued from previous page)

```

)
>>> dist_1.get_params()
{'loc': tensor([[0.]]) , 'scale': tensor([[1.]])}

>>> dist_2 = Normal(loc=torch.tensor(0.), scale="z", cond_var=["z"], var=["x
↪"])
>>> print(dist_2)
Distribution:
  p(x|z)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([1])
    (loc): torch.Size([1])
  )
>>> dist_2.get_params({"z": torch.tensor(1.)})
{'scale': tensor(1.), 'loc': tensor([0.])}

```

**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.LogProb*.

#### 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,

```

(continues on next page)

(continued from previous page)

```

-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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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])
```

(continues on next page)

(continued from previous page)

```

)
>>> 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]])}

```

**get\_log\_prob** (*x\_dict*, *sum\_features=True*, *feature\_dims=None*, *compute\_jacobian=False*)

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*

## 1.4.2 InverseTransformedDistribution

```
class pixyz.distributions.InverseTransformedDistribution(prior, flow, var,  
                                                    cond_var=[],  
                                                    name='p')
```

Bases: *pixyz.distributions.distributions.Distribution*

Convert inverse flow transformations to distributions.

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

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

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

Moreover, this distribution can take a conditional variable.

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

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)  
 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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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])
  )
```

(continues on next page)

(continued from previous page)

```

>>> 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]])}

```

**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 (\*\*kwargs)Bases: *pixyz.distributions.distributions.Distribution*

Deterministic distribution (or degeneration distribution)

**Examples**

```

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

```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError
```

**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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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)
```

(continues on next page)

(continued from previous page)

```

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]])

```

## 1.5.2 DataDistribution

**class** pixyz.distributions.DataDistribution (var, name='p\_{data}')

Bases: *pixyz.distributions.distributions.Distribution*

Data distribution.

Samples from this distribution equal given inputs.

### Examples

```
>>> import torch
>>> p = DataDistribution(var=["x"])
>>> print(p)
Distribution:
  p_{data}(x)
Network architecture:
  DataDistribution(
    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={}, \*\*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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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)
```

(continues on next page)

(continued from previous page)

```

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]])}

```

**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]])

```

**input\_var**

In `DataDistribution`, `input_var` is same as `var`.

### 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])
```

## 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:
  ReplaceVarDistribution(
    name=p, distribution_name=,
    var=['y'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (p): 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 overridden.

**get\_params** (*params\_dict*={})

This method aims to get parameters of this distributions from constant parameters set in initialization and outputs of DNNs.

**Parameters** **params\_dict** (*dict*, defaults to {}) – Input parameters.

**Returns** **output\_dict** – Output parameters.

**Return type** *dict*

## Examples

```
>>> from pixyz.distributions import Normal
>>> dist_1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...                 features_shape=[1])
>>> print(dist_1)
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([1])
    (loc): torch.Size([1, 1])
    (scale): torch.Size([1, 1])
  )
>>> dist_1.get_params()
{'loc': tensor([[0.]]) , 'scale': tensor([[1.]])}
```

```

>>> dist_2 = Normal(loc=torch.tensor(0.), scale="z", cond_var=["z"], var=["x
↳"])
>>> print(dist_2)
Distribution:
  p(x|z)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([1])
    (loc): torch.Size([1])
  )
>>> dist_2.get_params({"z": torch.tensor(1.)})
{'scale': tensor(1.), 'loc': tensor([0.] )}
```

**set\_dict** (*x\_dict*={}, *sampling*=False, *batch\_n*=None, *\*\*kwargs*)

**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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **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,
```

(continues on next page)

(continued from previous page)

```

    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]])}

```

**get\_log\_prob** (*x\_dict*, *\*\*kwargs*)

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\_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.]])

```

**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

## 1.6.2 MarginalizeVarDistribution

**class** `pixyz.distributions.MarginalizeVarDistribution` (*p*, *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:
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
>>> p_marg = MarginalizeVarDistribution(p_multi, ["y"])

```

(continues on next page)

(continued from previous page)

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

**\_\_init\_\_** (*p*, *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 overridden.

**get\_params** (*params\_dict*={})

This method aims to get parameters of this distributions from constant parameters set in initialization and outputs of DNNs.

**Parameters** *params\_dict* (dict, defaults to {}) – Input parameters.

**Returns** *output\_dict* – Output parameters.

**Return type** dict

## Examples

```
>>> from pixyz.distributions import Normal
>>> dist_1 = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"],
...                 features_shape=[1])
>>> print(dist_1)
Distribution:
  p(x)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=[], input_var=[], features_shape=torch.Size([1])
    (loc): torch.Size([1, 1])
    (scale): torch.Size([1, 1])
  )
>>> dist_1.get_params()
{'loc': tensor([[0.]]) , 'scale': tensor([[1.]])}
```

```
>>> dist_2 = Normal(loc=torch.tensor(0.), scale="z", cond_var=["z"], var=["x
↪"])
>>> print(dist_2)
Distribution:
```

(continues on next page)

(continued from previous page)

```

p(x|z)
Network architecture:
  Normal(
    name=p, distribution_name=Normal,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
    (loc): torch.Size([1])
  )
>>> dist_2.get_params({"z": torch.tensor(1.)})
{'scale': tensor(1.), 'loc': tensor([0.])}

```

**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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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]])}

```

**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.]])
```

**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

**prob\_factorized\_text**

Return a formula of the factorized probability distribution.

**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:
  DistributionBase(
    name=p, distribution_name=,
    var=['z'], cond_var=['y'], input_var=['y'], features_shape=torch.Size([])
  )
  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:
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
  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:
  DistributionBase(
    name=p, distribution_name=,
    var=['y'], cond_var=['a'], input_var=['a'], features_shape=torch.Size([])
  )
  DistributionBase(
    name=p, distribution_name=,
    var=['x'], cond_var=['z'], input_var=['z'], features_shape=torch.Size([])
  )
```

**\_\_init\_\_(a,b)**

### Parameters

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

### input\_var

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

### Type list

### prob\_factorized\_text

Return a formula of the factorized probability distribution.

**Type** str

**sample** (*x\_dict*={}, *batch\_n*=None, *return\_all*=True, *reparam*=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.
- **sample\_shape** (list or NoneType, defaults to torch.Size()) – Shape of generating samples.
- **batch\_n** (int, defaults to None.) – Set batch size of parameters.
- **return\_all** (bool, defaults to True) – Choose whether the output contains input variables.
- **reparam** (bool, defaults to False.) – Choose whether we sample variables with re-parameterized 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,
```

(continues on next page)

(continued from previous page)

```

    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]])}

```

**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])

```



## 2.1 Loss

**class** pixyz.losses.losses.**Loss** (*p, q=None, input\_var=None*)

Bases: object

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 StochasticReconstructionLoss, KullbackLeibler
...
>>> # Set distributions
>>> class Inference(Normal):
...     def __init__(self):
...         super().__init__(cond_var=["x"], var=["z"], 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__(cond_var=["z"], var=["x"], 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 = StochasticReconstructionLoss(q, p)
>>> kl = KullbackLeibler(q, prior)
>>> loss_cls = (reconst - kl).mean()
>>> print(loss_cls)
mean \left(- D_{\text{KL}} \left[ q(z|x) || p_{\text{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\_\_** (*p*, *q*=None, *input\_var*=None)

#### Parameters

- **p** (*pixyz.distributions.Distribution*) – Distribution.
- **q** (*pixyz.distributions.Distribution*, defaults to None) – Distribution.
- **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*

#### expectation (*p*, *input\_var*=None, *sample\_shape*=torch.Size([1]))

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

#### Parameters



- **p** (*pixyz.distributions.Distribution*) – Distribution for sampling.
- **input\_var** (*list*) – Input variables of this loss.
- **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*

**eval** (*x\_dict={}*, *return\_dict=False*, *\*\*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.

**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.

## 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])
```

### 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])
```

## 2.3 Expected value

### 2.3.1 Expectation

**class** pixyz.losses.**Expectation** (*p, f, input\_var=None, sample\_shape=torch.Size([])*)

Bases: `pixyz.losses.losses.Loss`

Expectation of a given function (Monte Carlo approximation).

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

where  $x_l \sim p(x)$ .

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

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

## Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> 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])
```

## 2.4 Entropy

### 2.4.1 CrossEntropy

**class** pixyz.losses.**CrossEntropy** (*p, q, input\_var=None*)

Bases: [pixyz.losses.losses.SetLoss](#)

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

$$H[p||q] = -\mathbb{E}_{p(x)}[\log q(x)] \approx -\frac{1}{L} \sum_{l=1}^L \log q(x_l),$$

where  $x_l \sim p(x)$ .

---

**Note:** This class is a special case of the [Expectation](#) class.

---

#### 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(0.), scale=torch.tensor(1.), var=["x"], features_
↳ shape=[64], name="q")
>>> loss_cls = CrossEntropy(p, q)
>>> print(loss_cls)
- \mathbb{E}_{\{p(x)\}} \left[ \log q(x) \right]
>>> loss = loss_cls.eval()
```

### 2.4.2 Entropy

**class** pixyz.losses.**Entropy** (*p, input\_var=None*)

Bases: [pixyz.losses.losses.SetLoss](#)

Entropy (Monte Carlo approximation).

$$H[p] = -\mathbb{E}_{p(x)}[\log p(x)] \approx -\frac{1}{L} \sum_{l=1}^L \log p(x_l),$$

where  $x_l \sim p(x)$ .

---

**Note:** This class is a special case of the [Expectation](#) class.

---

#### 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)
>>> print(loss_cls)
- \mathbb{E}_{p(x)} \left[ \log p(x) \right]
>>> loss = loss_cls.eval()
```

### 2.4.3 AnalyticalEntropy

**class** pixyz.losses.**AnalyticalEntropy** (*p, q=None, input\_var=None*)

Bases: *pixyz.losses.losses.Loss*

Entropy (analytical).

$$H[p] = -\mathbb{E}_{p(x)}[\log p(x)]$$

#### Examples

```
>>> import torch
>>> from pixyz.distributions import Normal
>>> p = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.), var=["x"], features_
↳shape=[64])
>>> loss_cls = AnalyticalEntropy(p)
>>> print(loss_cls)
- \mathbb{E}_{p(x)} \left[ \log p(x) \right]
>>> loss = loss_cls.eval()
```

### 2.4.4 StochasticReconstructionLoss

**class** pixyz.losses.**StochasticReconstructionLoss** (*encoder, decoder, input\_var=None*)

Bases: *pixyz.losses.losses.SetLoss*

Reconstruction Loss (Monte Carlo approximation).

$$-\mathbb{E}_{q(z|x)}[\log p(x|z)] \approx -\frac{1}{L} \sum_{l=1}^L \log p(x|z_l),$$

where  $z_l \sim q(z|x)$ .

---

**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], name="q") # q(z/x)
```

(continues on next page)

(continued from previous page)

```
>>> p = Normal(loc="z", scale=torch.tensor(1.), var=["x"], cond_var=["z"],
↳ features_shape=[64], name="p") # p(x/z)
>>> loss_cls = StochasticReconstructionLoss(q, p)
>>> print(loss_cls)
- \mathbb{E}_{q(z|x)} \left[ \log p(x|z) \right]
>>> loss = loss_cls.eval({"x": torch.randn(1, 64)})
```

## 2.5 Lower bound

### 2.5.1 ELBO

**class** pixyz.losses.**ELBO**(p, q, input\_var=None)

Bases: [pixyz.losses.losses.SetLoss](#)

The evidence lower bound (Monte Carlo approximation).

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

where  $z_l \sim q(z|x)$ .

---

**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

**class** pixyz.losses.**KullbackLeibler**(p, q, input\_var=None, dim=None)

Bases: [pixyz.losses.losses.Loss](#)

Kullback-Leibler divergence (analytical).

$$D_{KL}[p||q] = \mathbb{E}_{p(x)} \left[ \log \frac{p(x)}{q(x)} \right]$$

## 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 = Beta(concentration0=torch.tensor(1.), concentration1=torch.tensor(1.),
...          var=["z"], features_shape=[64], name="q")
>>> loss_cls = KullbackLeibler(p, q)
>>> print(loss_cls)
D_{KL} \left[p(z)||q(z) \right]
>>> loss = loss_cls.eval()
```

### 2.6.2 WassersteinDistance

**class** pixyz.losses.**WassersteinDistance** (*p, q, metric=PairwiseDistance(), input\_var=None*)  
 Bases: *pixyz.losses.losses.Loss*

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) })
```

### 2.6.3 MMD

**class** pixyz.losses.**MMD** (*p, q, input\_var=None, kernel='gaussian', \*\*kernel\_params*)  
 Bases: *pixyz.losses.losses.Loss*

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)})
```

## 2.7 Adversarial statistical distance

### 2.7.1 AdversarialJensenShannon

```
class pixyz.losses.AdversarialJensenShannon(p, q, discriminator, input_var=None, opti-
mizer=<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, DataDistribution, Normal
>>> # Generator
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(cond_var=["z"], var=["x"], 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:
```

(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])
)
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 = DataDistribution({"x"})
>>> print(p_data)
Distribution:
  p_{data}(x)
Network architecture:
  DataDistribution(
    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__(cond_var=["x"], var=["t"], 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 = 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.train({"x": sample_x})

```



## References

[Goodfellow+ 2014] Generative Adversarial Networks

**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

**train** (*train\_x\_dict*, *\*\*kwargs*)

Train the evaluation metric (discriminator).

### Parameters

- **train\_x\_dict** (*dict*) – Input variables.
- **\*\*kwargs** – Arbitrary keyword arguments.

### Returns

**Return type** torch.Tensor

**test** (*test\_x\_dict*, *\*\*kwargs*)

Test the evaluation metric (discriminator).

### Parameters

- **test\_x\_dict** (*dict*) – Input variables.
- **\*\*kwargs** – Arbitrary keyword arguments.

### Returns

**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).

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

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, DataDistribution, Normal
>>> # Generator
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(cond_var=["z"], var=["x"], 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:
  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])
  )
  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 = DataDistribution(["x"])
>>> print(p_data)
Distribution:
  p_{data}(x)
Network architecture:
  DataDistribution(
    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__(cond_var=["x"], var=["t"], name="d")
...         self.model = nn.Linear(64, 1)
...     def forward(self, x):
...         return {"t": torch.sigmoid(self.model(x))}
```

(continues on next page)

(continued from previous page)

```

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

```

## References

[Kim+ 2018] Disentangling by Factorising

**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*

**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

**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, DataDistribution, Normal
>>> # Generator
>>> class Generator(Deterministic):
...     def __init__(self):
...         super(Generator, self).__init__(cond_var=["z"], var=["x"], 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:
  Normal(
    name=p_{prior}, distribution_name=Normal,
    var=['z'], cond_var=[], input_var=[], features_shape=torch.Size([32])
    (loc): torch.Size([1, 32])
```

(continues on next page)

(continued from previous page)

```

        (scale): torch.Size([1, 32])
    )
    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 = DataDistribution(["x"])
>>> print(p_data)
Distribution:
  p_{data}(x)
Network architecture:
  DataDistribution(
    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__(cond_var=["x"], var=["t"], 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.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*

**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*

**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, input_var=None, series_var=None, update_value={}, slice_step=None, timestep_var=['t'])
```

Bases: *pixyz.losses.losses.Loss*

Iterative loss.

This class allows implementing an arbitrary model which requires iteration.

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

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
>>> x_dim = 128
>>> z_dim = 64
>>> h_dim = 32
>>>
>>> # p(x|z, h_{prev})
>>> class Decoder(Bernoulli):
...     def __init__(self):
...         super().__init__(cond_var=["z", "h_prev"], var=["x"], 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__(cond_var=["x", "h_prev"], var=["z"], 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__(cond_var=["x", "z", "h_prev"], var=["h"], name="f")
...         self.rnn_cell = torch.nn.GRUCell(x_dim + z_dim, h_dim)
...     def forward(self, x, z, h_prev):
...         return {"h": self.rnn_cell(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}_{p(h, z|x, h_{prev})} \left[\log p(x|z, h_{prev})\right] \right)
↪\right)
```

(continues on next page)

(continued from previous page)

```

>>> loss_cls = IterativeLoss(step_loss=step_loss_cls,
...                           series_var=["x"], update_value={"h": "h_prev"})
>>> print(loss_cls)
\sum_{t=1}^{t_{\max}} \text{mean} \left( \mathbb{E}_{p(h, z | x, h_{\text{prev}})} \left[ \log p(x | z, h_{\text{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)`

## 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

```

### 2.9.2 SetLoss

**class** pixyz.losses.losses.**SetLoss**(*loss*)

Bases: `pixyz.losses.losses.Loss`

## 2.10 Operators

### 2.10.1 LossOperator

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

Bases: `pixyz.losses.losses.Loss`



### 2.10.2 LossSelfOperator

```
class pixyz.losses.losses.LossSelfOperator (loss1)
    Bases: pixyz.losses.losses.Loss

    train (x_dict={}, **kwargs)

    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)
5
```

### 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)
-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)
2
```

### 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)
8
```

### 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)
0.5
>>> 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)
2.0
```

### 2.10.7 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
```

### 2.10.8 AbsLoss

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

Bases: *pixyz.losses.losses.LossSelfOperator*

Apply the *abs* operation to two losses.

## 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])
```

### 2.10.9 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])
```

(continues on next page)

(continued from previous page)

```

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

```

## 2.10.10 BatchSum

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

```

### 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 StochasticReconstructionLoss, KullbackLeibler
...
>>> # Set distributions (Distribution API)
>>> class Inference(Normal):
...     def __init__(self):
...         super().__init__(cond_var=["x"], var=["z"], 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__(cond_var=["z"], var=["x"], 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 = StochasticReconstructionLoss(q, p)
>>> kl = KullbackLeibler(q, prior)
>>> loss_cls = (reconst - kl).mean()
>>> print(loss_cls)
mean \left(- D_{\text{KL}} \left[q(z|x) || p_{\text{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, op-  
                      timizer=<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'>, op-
timizer_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.

```
__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`).

## 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`

**extra\_repr** ()

Set the extra representation of the module

To print customized extra information, you should reimplement 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
>>> 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 reimplement 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  $1 \times 1$  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.

$c * s * s \rightarrow 4c * s/2 * s/2$

#### 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>
          [[ 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.

$c * s * s \rightarrow c/4 * 2s * 2s$

### 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]],
        <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],
```

(continues on next page)

(continued from previous page)

```

        [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],

```

(continues on next page)

(continued from previous page)

```

        [ 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)).data
>>> diff < 0.1
tensor(1, dtype=torch.uint8)
```

**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)).data
>>> diff < 0.1
tensor(1, dtype=torch.uint8)
```

### 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*





`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(dict, 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})
```

`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()
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`



### p

- `pixyz.distributions`, 3
- `pixyz.flows`, [79](#)
- `pixyz.losses`, [51](#)
- `pixyz.models`, [73](#)
- `pixyz.utils`, [93](#)





## Symbols

`__init__()` (*pixyz.distributions.CustomProb* method), 36  
`__init__()` (*pixyz.distributions.ElementWiseProductOfNormal* method), 25  
`__init__()` (*pixyz.distributions.MarginalizeVarDistribution* method), 43  
`__init__()` (*pixyz.distributions.MixtureModel* method), 17  
`__init__()` (*pixyz.distributions.MultiplyDistribution* method), 47  
`__init__()` (*pixyz.distributions.ProductOfNormal* method), 21  
`__init__()` (*pixyz.distributions.ReplaceVarDistribution* method), 38  
`__init__()` (*pixyz.distributions.distributions.Distribution* method), 4  
`__init__()` (*pixyz.flows.Flow* method), 79  
`__init__()` (*pixyz.flows.FlowList* method), 80  
`__init__()` (*pixyz.losses.losses.Loss* method), 52  
`__init__()` (*pixyz.models.GAN* method), 77  
`__init__()` (*pixyz.models.ML* method), 75  
`__init__()` (*pixyz.models.Model* method), 74  
`__init__()` (*pixyz.models.VAE* method), 76  
`__init__()` (*pixyz.models.VI* method), 77

## A

`abs()` (*pixyz.losses.losses.Loss* method), 52  
`AbsLoss` (class in *pixyz.losses.losses*), 71  
`ActNorm2d` (class in *pixyz.flows*), 90  
`AddLoss` (class in *pixyz.losses.losses*), 69  
`AdversarialJensenShannon` (class in *pixyz.losses*), 59  
`AdversarialKullbackLeibler` (class in *pixyz.losses*), 61  
`AdversarialWassersteinDistance` (class in *pixyz.losses*), 64  
`AffineCoupling` (class in *pixyz.flows*), 81  
`AnalyticalEntropy` (class in *pixyz.losses*), 56

## B

`BatchMean` (class in *pixyz.losses.losses*), 71  
`BatchNorm1d` (class in *pixyz.flows*), 88  
`BatchNorm2d` (class in *pixyz.flows*), 89  
`BatchSum` (class in *pixyz.losses.losses*), 72  
`Bernoulli` (class in *pixyz.distributions*), 12  
`Beta` (class in *pixyz.distributions*), 15  
`build_mask()` (*pixyz.flows.AffineCoupling* method), 81

## C

`Categorical` (class in *pixyz.distributions*), 14  
`ChannelConv` (class in *pixyz.flows*), 83  
`cond_var` (*pixyz.distributions.distributions.Distribution* attribute), 4  
`convert_latex_name()` (in module *pixyz.utils*), 96  
`CrossEntropy` (class in *pixyz.losses*), 55  
`CustomProb` (class in *pixyz.distributions*), 36

## D

`d_loss()` (*pixyz.losses.AdversarialJensenShannon* method), 61  
`d_loss()` (*pixyz.losses.AdversarialKullbackLeibler* method), 63  
`d_loss()` (*pixyz.losses.AdversarialWassersteinDistance* method), 65  
`DataDistribution` (class in *pixyz.distributions*), 34  
`delete_dict_values()` (in module *pixyz.utils*), 94  
`deriv_tanh()` (*pixyz.flows.PlanarFlow* method), 81  
`detach_dict()` (in module *pixyz.utils*), 94  
`Deterministic` (class in *pixyz.distributions*), 31  
`Dirichlet` (class in *pixyz.distributions*), 16  
`Distribution` (class in *pixyz.distributions.distributions*), 3  
`distribution_name` (*pixyz.distributions.Bernoulli* attribute), 12  
`distribution_name` (*pixyz.distributions.Beta* attribute), 16

distribution_name ( <a href="#">pixyz.distributions.Categorical</a> attribute), <a href="#">14</a>	distribution_torch_class ( <a href="#">pixyz.distributions.Dirichlet</a> attribute), <a href="#">16</a>
distribution_name ( <a href="#">pixyz.distributions.CustomProb</a> attribute), <a href="#">37</a>	distribution_torch_class ( <a href="#">pixyz.distributions.Gamma</a> attribute), <a href="#">16</a>
distribution_name ( <a href="#">pixyz.distributions.DataDistribution</a> attribute), <a href="#">34</a>	distribution_torch_class ( <a href="#">pixyz.distributions.Laplace</a> attribute), <a href="#">11</a>
distribution_name ( <a href="#">pixyz.distributions.Deterministic</a> attribute), <a href="#">32</a>	distribution_torch_class ( <a href="#">pixyz.distributions.Normal</a> attribute), <a href="#">11</a>
distribution_name ( <a href="#">pixyz.distributions.Dirichlet</a> attribute), <a href="#">16</a>	distribution_torch_class ( <a href="#">pixyz.distributions.RelaxedBernoulli</a> attribute), <a href="#">12</a>
distribution_name ( <a href="#">pixyz.distributions.distributions.Distribution</a> attribute), <a href="#">4</a>	distribution_torch_class ( <a href="#">pixyz.distributions.RelaxedCategorical</a> attribute), <a href="#">14</a>
distribution_name ( <a href="#">pixyz.distributions.FactorizedBernoulli</a> attribute), <a href="#">13</a>	DivLoss (class in <a href="#">pixyz.losses.losses</a> ), <a href="#">70</a>
distribution_name ( <a href="#">pixyz.distributions.Gamma</a> attribute), <a href="#">16</a>	<b>E</b>
distribution_name ( <a href="#">pixyz.distributions.InverseTransformedDistribution</a> attribute), <a href="#">28</a>	ELBO (class in <a href="#">pixyz.losses</a> ), <a href="#">57</a>
distribution_name ( <a href="#">pixyz.distributions.Laplace</a> attribute), <a href="#">11</a>	ElementWiseProductOfNormal (class in <a href="#">pixyz.distributions</a> ), <a href="#">24</a>
distribution_name ( <a href="#">pixyz.distributions.MarginalizeVarDistribution</a> attribute), <a href="#">46</a>	Entropy (class in <a href="#">pixyz.losses</a> ), <a href="#">55</a>
distribution_name ( <a href="#">pixyz.distributions.MixtureModel</a> attribute), <a href="#">18</a>	epsilon() (in module <a href="#">pixyz.utils</a> ), <a href="#">93</a>
distribution_name ( <a href="#">pixyz.distributions.Normal</a> attribute), <a href="#">11</a>	eval() ( <a href="#">pixyz.losses.losses.Loss</a> method), <a href="#">53</a>
distribution_name ( <a href="#">pixyz.distributions.RelaxedBernoulli</a> attribute), <a href="#">12</a>	Expectation (class in <a href="#">pixyz.losses</a> ), <a href="#">54</a>
distribution_name ( <a href="#">pixyz.distributions.RelaxedCategorical</a> attribute), <a href="#">14</a>	expectation() ( <a href="#">pixyz.losses.losses.Loss</a> method), <a href="#">52</a>
distribution_name ( <a href="#">pixyz.distributions.ReplaceVarDistribution</a> attribute), <a href="#">42</a>	extra_repr() ( <a href="#">pixyz.distributions.distributions.Distribution</a> method), <a href="#">11</a>
distribution_name ( <a href="#">pixyz.distributions.TransformedDistribution</a> attribute), <a href="#">25</a>	extra_repr() ( <a href="#">pixyz.flows.AffineCoupling</a> method), <a href="#">83</a>
distribution_torch_class ( <a href="#">pixyz.distributions.Bernoulli</a> attribute), <a href="#">12</a>	extra_repr() ( <a href="#">pixyz.flows.PlanarFlow</a> method), <a href="#">81</a>
distribution_torch_class ( <a href="#">pixyz.distributions.Beta</a> attribute), <a href="#">16</a>	<b>F</b>
distribution_torch_class ( <a href="#">pixyz.distributions.Categorical</a> attribute), <a href="#">14</a>	FactorizedBernoulli (class in <a href="#">pixyz.distributions</a> ), <a href="#">13</a>
	features_shape( <a href="#">pixyz.distributions.distributions.Distribution</a> attribute), <a href="#">5</a>
	Flatten (class in <a href="#">pixyz.flows</a> ), <a href="#">88</a>
	Flow (class in <a href="#">pixyz.flows</a> ), <a href="#">79</a>
	flow_input_var( <a href="#">pixyz.distributions.TransformedDistribution</a> attribute), <a href="#">25</a>
	flow_output_var( <a href="#">pixyz.distributions.InverseTransformedDistribution</a> attribute), <a href="#">28</a>
	FlowList (class in <a href="#">pixyz.flows</a> ), <a href="#">80</a>
	forward() ( <a href="#">pixyz.distributions.distributions.Distribution</a> method), <a href="#">10</a>
	forward() ( <a href="#">pixyz.distributions.InverseTransformedDistribution</a> method), <a href="#">30</a>
	forward() ( <a href="#">pixyz.distributions.MarginalizeVarDistribution</a> method), <a href="#">43</a>
	forward() ( <a href="#">pixyz.distributions.ReplaceVarDistribution</a> method), <a href="#">38</a>
	forward() ( <a href="#">pixyz.distributions.TransformedDistribution</a> method), <a href="#">27</a>
	forward() ( <a href="#">pixyz.flows.ActNorm2d</a> method), <a href="#">90</a>

[forward\(\)](#) ([pixyz.flows.AffineCoupling method](#)), 82  
[forward\(\)](#) ([pixyz.flows.BatchNorm1d method](#)), 89  
[forward\(\)](#) ([pixyz.flows.ChannelConv method](#)), 83  
[forward\(\)](#) ([pixyz.flows.Flatten method](#)), 88  
[forward\(\)](#) ([pixyz.flows.Flow method](#)), 79  
[forward\(\)](#) ([pixyz.flows.FlowList method](#)), 80  
[forward\(\)](#) ([pixyz.flows.Permutation method](#)), 87  
[forward\(\)](#) ([pixyz.flows.PlanarFlow method](#)), 81  
[forward\(\)](#) ([pixyz.flows.Preprocess method](#)), 91  
[forward\(\)](#) ([pixyz.flows.Squeeze method](#)), 84  
[forward\(\)](#) ([pixyz.flows.Unsqueeze method](#)), 86

## G

[g\\_loss\(\)](#) ([pixyz.losses.AdversarialJensenShannon method](#)), 61  
[g\\_loss\(\)](#) ([pixyz.losses.AdversarialKullbackLeibler method](#)), 63  
[g\\_loss\(\)](#) ([pixyz.losses.AdversarialWassersteinDistance method](#)), 66  
[Gamma](#) ([class in pixyz.distributions](#)), 16  
[GAN](#) ([class in pixyz.models](#)), 77  
[get\\_dict\\_values\(\)](#) ([in module pixyz.utils](#)), 93  
[get\\_entropy\(\)](#) ([pixyz.distributions.distributions.Distribution method](#)), 8  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.CustomProb method](#)), 37  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.distributions.Distribution method](#)), 8  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.FactorizedBernoulli method](#)), 13  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.InverseTransformedDistribution method](#)), 30  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.MixtureModel method](#)), 19  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.MultiplyDistribution method](#)), 49  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.ProductOfNormal method](#)), 23  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.ReplaceVarDistribution method](#)), 40  
[get\\_log\\_prob\(\)](#) ([pixyz.distributions.TransformedDistribution method](#)), 27  
[get\\_parameters\(\)](#) ([pixyz.flows.AffineCoupling method](#)), 82  
[get\\_parameters\(\)](#) ([pixyz.flows.ChannelConv method](#)), 83  
[get\\_params\(\)](#) ([pixyz.distributions.distributions.Distribution method](#)), 5  
[get\\_params\(\)](#) ([pixyz.distributions.MarginalizeVarDistribution method](#)), 43  
[get\\_params\(\)](#) ([pixyz.distributions.ProductOfNormal method](#)), 21  
[get\\_params\(\)](#) ([pixyz.distributions.ReplaceVarDistribution method](#)), 38

## H

[hidden\\_var](#) ([pixyz.distributions.MixtureModel attribute](#)), 18

## I

[in\\_features](#) ([pixyz.flows.Flow attribute](#)), 79  
[inference\(\)](#) ([pixyz.distributions.InverseTransformedDistribution method](#)), 30  
[initialize\\_parameters\(\)](#) ([pixyz.flows.ActNorm2d method](#)), 90  
[input\\_var](#) ([pixyz.distributions.CustomProb attribute](#)), 37  
[input\\_var](#) ([pixyz.distributions.DataDistribution attribute](#)), 36  
[input\\_var](#) ([pixyz.distributions.distributions.Distribution attribute](#)), 5  
[input\\_var](#) ([pixyz.distributions.MarginalizeVarDistribution attribute](#)), 46  
[input\\_var](#) ([pixyz.distributions.MultiplyDistribution attribute](#)), 47  
[input\\_var](#) ([pixyz.distributions.ReplaceVarDistribution attribute](#)), 42  
[input\\_var](#) ([pixyz.losses.losses.Loss attribute](#)), 52  
[inverse\(\)](#) ([pixyz.distributions.InverseTransformedDistribution method](#)), 31  
[inverse\(\)](#) ([pixyz.distributions.TransformedDistribution method](#)), 28  
[inverse\(\)](#) ([pixyz.flows.ActNorm2d method](#)), 90  
[inverse\(\)](#) ([pixyz.flows.AffineCoupling method](#)), 83  
[inverse\(\)](#) ([pixyz.flows.BatchNorm1d method](#)), 89  
[inverse\(\)](#) ([pixyz.flows.ChannelConv method](#)), 84  
[inverse\(\)](#) ([pixyz.flows.Flatten method](#)), 88  
[inverse\(\)](#) ([pixyz.flows.Flow method](#)), 79  
[inverse\(\)](#) ([pixyz.flows.FlowList method](#)), 80  
[inverse\(\)](#) ([pixyz.flows.Permutation method](#)), 87  
[inverse\(\)](#) ([pixyz.flows.Preprocess method](#)), 91  
[inverse\(\)](#) ([pixyz.flows.Squeeze method](#)), 85  
[inverse\(\)](#) ([pixyz.flows.Unsqueeze method](#)), 86  
[InverseTransformedDistribution](#) ([class in pixyz.distributions](#)), 28  
[IterativeLoss](#) ([class in pixyz.losses](#)), 66

## K

[KullbackLeibler](#) ([class in pixyz.losses](#)), 57

## L

[Laplace](#) ([class in pixyz.distributions](#)), 11  
[log\\_prob\(\)](#) ([pixyz.distributions.distributions.Distribution method](#)), 9  
[log\\_prob\(\)](#) ([pixyz.distributions.ProductOfNormal method](#)), 22  
[log\\_prob\\_function](#) ([pixyz.distributions.CustomProb attribute](#)), 36

[logdet\\_jacobian \(pixyz.distributions.InverseTransformedDistribution attribute\), 28](#)  
[logdet\\_jacobian \(pixyz.distributions.TransformedDistribution attribute\), 25](#)  
[logdet\\_jacobian \(pixyz.flows.Flow attribute\), 80](#)  
[logit \(\) \(pixyz.flows.Preprocess static method\), 91](#)  
[LogProb \(class in pixyz.losses\), 53](#)  
[Loss \(class in pixyz.losses.losses\), 51](#)  
[loss\\_text \(pixyz.losses.losses.Loss attribute\), 52](#)  
[LossOperator \(class in pixyz.losses.losses\), 68](#)  
[LossSelfOperator \(class in pixyz.losses.losses\), 69](#)

## M

[marginalize\\_var \(\) \(pixyz.distributions.distributions.Distribution method\), 11](#)  
[MarginalizeVarDistribution \(class in pixyz.distributions\), 42](#)  
[mean \(\) \(pixyz.losses.losses.Loss method\), 52](#)  
[MixtureModel \(class in pixyz.distributions\), 16](#)  
[ML \(class in pixyz.models\), 75](#)  
[MMD \(class in pixyz.losses\), 58](#)  
[Model \(class in pixyz.models\), 73](#)  
[MulLoss \(class in pixyz.losses.losses\), 70](#)  
[MultiplyDistribution \(class in pixyz.distributions\), 46](#)

## N

[name \(pixyz.distributions.distributions.Distribution attribute\), 4](#)  
[NegLoss \(class in pixyz.losses.losses\), 70](#)  
[Normal \(class in pixyz.distributions\), 11](#)

## P

[Parameter \(class in pixyz.losses.losses\), 68](#)  
[params\\_keys \(pixyz.distributions.Bernoulli attribute\), 12](#)  
[params\\_keys \(pixyz.distributions.Beta attribute\), 15](#)  
[params\\_keys \(pixyz.distributions.Categorical attribute\), 14](#)  
[params\\_keys \(pixyz.distributions.Dirichlet attribute\), 16](#)  
[params\\_keys \(pixyz.distributions.Gamma attribute\), 16](#)  
[params\\_keys \(pixyz.distributions.Laplace attribute\), 11](#)  
[params\\_keys \(pixyz.distributions.Normal attribute\), 11](#)  
[Permutation \(class in pixyz.flows\), 86](#)  
[pixyz.distributions \(module\), 3](#)  
[pixyz.flows \(module\), 79](#)  
[pixyz.losses \(module\), 51](#)  
[pixyz.models \(module\), 73](#)  
[pixyz.utils \(module\), 93](#)

[RelaxedBernoulli \(class in pixyz.distributions\), 80](#)  
[posterior \(\) \(pixyz.distributions.MixtureModel method\), 18](#)  
[Preprocess \(class in pixyz.flows\), 91](#)  
[print\\_latex \(\) \(in module pixyz.utils\), 96](#)  
[Prob \(class in pixyz.losses\), 53](#)  
[prob \(\) \(pixyz.distributions.distributions.Distribution method\), 10](#)  
[prob \(\) \(pixyz.distributions.ProductOfNormal method\), 22](#)  
[prob\\_factorized\\_text \(pixyz.distributions.distributions.Distribution attribute\), 5](#)  
[prob\\_factorized\\_text \(pixyz.distributions.InverseTransformedDistribution attribute\), 28](#)  
[prob\\_factorized\\_text \(pixyz.distributions.MarginalizeVarDistribution attribute\), 46](#)  
[prob\\_factorized\\_text \(pixyz.distributions.MixtureModel attribute\), 18](#)  
[prob\\_factorized\\_text \(pixyz.distributions.MultiplyDistribution attribute\), 47](#)  
[prob\\_factorized\\_text \(pixyz.distributions.ProductOfNormal attribute\), 21](#)  
[prob\\_factorized\\_text \(pixyz.distributions.TransformedDistribution attribute\), 25](#)  
[prob\\_joint\\_factorized\\_and\\_text \(pixyz.distributions.distributions.Distribution attribute\), 5](#)  
[prob\\_joint\\_factorized\\_and\\_text \(pixyz.distributions.ProductOfNormal attribute\), 21](#)  
[prob\\_text \(pixyz.distributions.distributions.Distribution attribute\), 5](#)  
[prob\\_text \(pixyz.distributions.MixtureModel attribute\), 18](#)  
[ProductOfNormal \(class in pixyz.distributions\), 20](#)

## R

[relaxed\\_distribution\\_torch\\_class \(pixyz.distributions.RelaxedBernoulli attribute\), 12](#)  
[relaxed\\_distribution\\_torch\\_class \(pixyz.distributions.RelaxedCategorical attribute\), 14](#)  
[RelaxedBernoulli \(class in pixyz.distributions\), 12](#)  
[RelaxedCategorical \(class in pixyz.distributions\), 14](#)  
[replace\\_dict\\_keys \(\) \(in module pixyz.utils\), 94](#)

`replace_dict_keys_split()` (in module `pixyz.utils`), 95  
`replace_var()` (`pixyz.distributions.distributions.Distribution` method), 10  
`ReplaceVarDistribution` (class in `pixyz.distributions`), 37  
`reset_parameters()` (`pixyz.flows.PlanarFlow` method), 81  
`Reverse` (class in `pixyz.flows`), 88

## S

`sample()` (`pixyz.distributions.DataDistribution` method), 34  
`sample()` (`pixyz.distributions.Deterministic` method), 32  
`sample()` (`pixyz.distributions.distributions.Distribution` method), 6  
`sample()` (`pixyz.distributions.InverseTransformedDistribution` method), 29  
`sample()` (`pixyz.distributions.MarginalizeVarDistribution` method), 44  
`sample()` (`pixyz.distributions.MixtureModel` method), 18  
`sample()` (`pixyz.distributions.MultiplyDistribution` method), 48  
`sample()` (`pixyz.distributions.ReplaceVarDistribution` method), 39  
`sample()` (`pixyz.distributions.TransformedDistribution` method), 26  
`sample_mean()` (`pixyz.distributions.DataDistribution` method), 35  
`sample_mean()` (`pixyz.distributions.Deterministic` method), 33  
`sample_mean()` (`pixyz.distributions.distributions.Distribution` method), 7  
`sample_mean()` (`pixyz.distributions.MarginalizeVarDistribution` method), 45  
`sample_mean()` (`pixyz.distributions.RelaxedCategorical` method), 14  
`sample_mean()` (`pixyz.distributions.ReplaceVarDistribution` method), 41  
`sample_variance()` (`pixyz.distributions.distributions.Distribution` method), 7  
`sample_variance()` (`pixyz.distributions.MarginalizeVarDistribution` method), 45  
`sample_variance()` (`pixyz.distributions.RelaxedCategorical` method), 15  
`sample_variance()` (`pixyz.distributions.ReplaceVarDistribution` method), 41

## T

`temperature` (`pixyz.distributions.RelaxedBernoulli` attribute), 12  
`temperature` (`pixyz.distributions.RelaxedCategorical` attribute), 14  
`test()` (`pixyz.losses.AdversarialJensenShannon` method), 61  
`test()` (`pixyz.losses.AdversarialKullbackLeibler` method), 64  
`test()` (`pixyz.losses.AdversarialWassersteinDistance` method), 66  
`test()` (`pixyz.losses.losses.LossSelfOperator` method), 69  
`test()` (`pixyz.models.GAN` method), 78  
`test()` (`pixyz.models.ML` method), 75  
`test()` (`pixyz.models.Model` method), 74  
`test()` (`pixyz.models.VAE` method), 76  
`test()` (`pixyz.models.VI` method), 77  
`tolist()` (in module `pixyz.utils`), 95  
`train()` (`pixyz.losses.AdversarialJensenShannon` method), 61  
`train()` (`pixyz.losses.AdversarialKullbackLeibler` method), 63  
`train()` (`pixyz.losses.AdversarialWassersteinDistance` method), 66  
`train()` (`pixyz.losses.losses.LossSelfOperator` method), 69  
`train()` (`pixyz.models.GAN` method), 78  
`train()` (`pixyz.models.ML` method), 75  
`train()` (`pixyz.models.Model` method), 74  
`train()` (`pixyz.models.VAE` method), 76  
`train()` (`pixyz.models.VI` method), 77  
`TransformedDistribution` (class in `pixyz.distributions`), 25

## U

Unsqueeze (*class in pixyz.flows*), [85](#)

## V

VAE (*class in pixyz.models*), [76](#)

var (*pixyz.distributions.distributions.Distribution*  
attribute), [4](#)

VI (*class in pixyz.models*), [77](#)

## W

WassersteinDistance (*class in pixyz.losses*), [58](#)