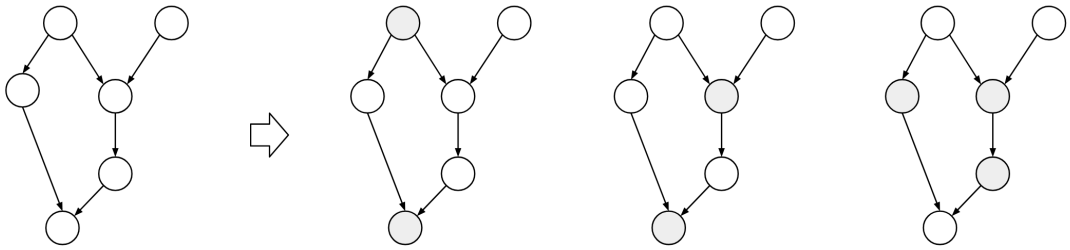
ZhuSuan Documentation

Release 0.4.0

ZhuSuan contributors

Aug 05, 2019

Contents



ZhuSuan is a python probabilistic programming library for **Bayesian deep learning**, which conjoins the complimentary advantages of Bayesian methods and deep learning. ZhuSuan is built upon Tensorflow. Unlike existing deep learning libraries, which are mainly designed for deterministic neural networks and supervised tasks, ZhuSuan provides deep learning style primitives and algorithms for building probabilistic models and applying Bayesian inference. The supported inference algorithms include:

- Variational inference with programmable variational posteriors, various objectives and advanced gradient estimators (SGVB, REINFORCE, VIMCO, etc.).
- Importance sampling for learning and evaluating models, with programmable proposals.
- Hamiltonian Monte Carlo (HMC) with parallel chains, and optional automatic parameter tuning.

CHAPTER 1

Installation

ZhuSuan is still under development. Before the first stable release (1.0), please clone the GitHub repository and run

```
pip install .
```

in the main directory. This will install ZhuSuan and its dependencies automatically. ZhuSuan also requires Tensorflow version 1.13.0 or later. Because users should choose whether to install the cpu or gpu version of Tensorflow, we do not include it in the dependencies. See Installing Tensorflow.

If you are developing ZhuSuan, you may want to install in an "editable" or "develop" mode. Please refer to the Contributing section in README.

After installation, open your python console and type:

```
>>> import zhusuan as zs
```

If no error occurs, you've successfully installed ZhuSuan.

1.1 Variational Autoencoders

Variational Auto-Encoders (VAE) [VAEKW13] is one of the most widely used deep generative models. In this tutorial, we show how to implement VAE in ZhuSuan step by step. The full script is at examples/variational_autoencoders/vae.py.

The generative process of a VAE for modeling binarized MNIST data is as follows:

$$\begin{aligned} z &\sim \mathrm{N}(z|0, I) \\ x_{logits} &= f_{NN}(z) \\ x &\sim \mathrm{Bernoulli}(x|\mathrm{sigmoid}(x_{logits})) \end{aligned}$$

This generative process is a stereotype for deep generative models, which starts with a latent representation (z) sampled from a simple distribution (such as standard Normal). Then the samples are forwarded through a deep neural network (f_{NN}) to capture the complex generative process of high dimensional observations such as images. Finally, some

noise is added to the output to get a tractable likelihood for the model. For binarized MNIST, the observation noise is chosen to be Bernoulli, with its parameters output by the neural network.

1.1.1 Build the model

In ZhuSuan, a model is constructed using *BayesianNet*, which describes a directed graphical model, i.e., Bayesian networks. The suggested practice is to wrap model construction into a function (we shall see the meanings of these arguments soon):

```
import zhusuan as zs

def build_gen(x_dim, z_dim, n, n_particles=1):
    bn = zs.BayesianNet()
```

Following the generative process, first we need a standard Normal distribution to generate the latent representations (z). As presented in our graphical model, the data is generated in batches with batch size `n`, and for each data, the latent representation is of dimension `z_dim`. So we add a stochastic node by `bn.normal` to generate samples of shape `[n, z_dim]`:

```
# z ~ N(z|0, I)
z_mean = tf.zeros([n, z_dim])
z = bn.normal("z", z_mean, std=1., group_ndims=1, n_samples=n_particles)
```

The method `bn.normal` is a helper function that creates a *Normal* distribution and adds a stochastic node that follows this distribution to the *BayesianNet* instance. The returned `z` is a *StochasticTensor*, which is Tensor-like and can be mixed with Tensors and fed into almost any Tensorflow primitives.

Note: To learn more about *Distribution* and *BayesianNet*. Please refer to *Basic Concepts in ZhuSuan*.

The shape of `z_mean` is `[n, z_dim]`, which means that we have `[n, z_dim]` independent inputs fed into the univariate *Normal* distribution. Because the input parameters are allowed to broadcast to match each other's shape, the standard deviation `std` is simply set to 1. Thus the shape of samples and probabilities evaluated at this node should be of shape `[n, z_dim]`. However, what we want in modeling MNIST data, is a batch of `[n]` independent events, with each one producing samples of `z` that is of shape `[z_dim]`, which is the dimension of latent representations. And the probabilities in every single event in the batch should be evaluated together, so the shape of local probabilities should be `[n]` instead of `[n, z_dim]`. In ZhuSuan, the way to achieve this is by setting *group_ndims*', as we do in the above model definition code. To help understand this, several other examples can be found in *Distribution* tutorial. `n_samples` is the number of samples to generate. It is None by default, in which case a single sample is generated without adding a new dimension.

Then we build a neural network of two fully-connected layers with z as the input, which is supposed to learn the complex transformation that generates images from their latent representations:

```
# x_logits = f_NN(z)
h = tf.layers.dense(z, 500, activation=tf.nn.relu)
h = tf.layers.dense(h, 500, activation=tf.nn.relu)
x_logits = tf.layers.dense(h, x_dim)
```

Next, we add an observation distribution (noise) that follows the Bernoulli distribution to get a tractable likelihood when evaluating the probability of an image:

```
# x ~ Bernoulli(x|sigmoid(x_logits))
bn.bernoulli("x", x_logits, group_ndims=1)
```

Note: The *Bernoulli* distribution accepts log-odds of probabilities instead of probabilities. This is designed for numeric stability reasons. Similar tricks are used in *Categorical* , which accepts log-probabilities instead of probabilities.

Putting together, the code for constructing a VAE is:

```
def build_gen(x_dim, z_dim, n, n_particles=1):
    bn = zs.BayesianNet()
    z_mean = tf.zeros([n, z_dim])
    z = bn.normal("z", z_mean, std=1., group_ndims=1, n_samples=n_particles)
    h = tf.layers.dense(z, 500, activation=tf.nn.relu)
    h = tf.layers.dense(h, 500, activation=tf.nn.relu)
    x_logits = tf.layers.dense(h, x_dim)
    bn.bernoulli("x", x_logits, group_ndims=1)
```

1.1.2 Reuse the model

Unlike common deep learning models (MLP, CNN, etc.), which is for supervised tasks, a key difficulty in designing programing primitives for generative models is their inner reusability. This is because in a probabilistic graphical model, a stochastic node can have two kinds of states, **observed or latent**. Consider the above case, if `z` is a tensor sampled from the prior, how about when you meet the condition that `z` is observed? In common practice of tensorflow programming, one has to build another computation graph from scratch and reuse the Variables (weights here). If there are many stochastic nodes in the model, this process will be really painful.

We provide a solution for this. To observe any stochastic nodes, pass a dictionary mapping of `(name, Tensor)` pairs when constructing *BayesianNet*. This will assign observed values to corresponding `StochasticTensor`s. For example, to observe a batch of images `x_batch`, write:

```
bn = zs.BayesianNet(observed={"x": x_batch})
```

Note: The observation passed must have the same type and shape as the *StochasticTensor*.

However, we usually need to pass different configurations of observations to the same *BayesianNet* more than once. To achieve this, ZhuSuan provides a new class called *MetaBayesianNet* to represent the meta version of *BayesianNet* which can repeatedly produce *BayesianNet* objects by accepting different observations. The recommended way to construct a *MetaBayesianNet* is by wrapping the function with a *meta_bayesian_net()* decorator:

```
@zs.meta_bayesian_net(scope="gen")
def build_gen(x_dim, z_dim, n, n_particles=1):
    ...
    return bn

model = build_gen(x_dim, z_dim, n, n_particles)
```

which transforms the function into returning a *MetaBayesianNet* instance:

```
>>> print(model)
<zhusuan.framework.meta_bn.MetaBayesianNet object at ...
```

so that we can observe stochastic nodes in this way:

```
# no observation
bn1 = model.observe()

# observe x
bn2 = model.observe(x=x_batch)
```

Each time the function is called, a different observation assignment is used to construct a *BayesianNet* instance. One question you may have in mind is that if there are Tensorflow Variables created in the above function, will them be reused across these bn s? The answer is no by default, but you can enable this by switching on the *reuse_variables* option in the decorator:

```
@zs.meta_bayesian_net(scope="gen", reuse_variables=True)
def build_gen(x_dim, z_dim, n, n_particles=1):
    ...
    return bn

model = build_gen(x_dim, z_dim, n, n_particles)
```

Then bn1 and bn2 will share the same set of Tensorflow Variables.

Note: This only shares Tensorflow Variables across different *BayesianNet* instances generated by the same *MetaBayesianNet* through the *observe()* method. Creating multiple *MetaBayesianNet* objects will recreate the tensorflow Variables, for example, in

```
m = build_gen(x_dim, z_dim, n, n_particles)
bn = m.observe()

m_new = build_gen(x_dim, z_dim, n, n_particles)
bn_new = m_new.observe()
```

bn and bn_new will use a different set of Tensorflow Variables.

Since reusing Tensorflow Variables in repeated function calls is a typical need, we provide another decorator *reuse_variables()* for the more general cases. Any function decorated by *reuse_variables()* will automatically create Tensorflow Variables the first time they are called and reuse them thereafter.

1.1.3 Inference and learning

Having built the model, the next step is to learn it from binarized MNIST images. We conduct Maximum Likelihood learning, that is, we are going to maximize the log likelihood of data in our model:

$$\max_{\theta} \log p_\theta(x)$$

where θ is the model parameter.

Note: In this variational autoencoder, the model parameter is the network weights, in other words, it's the Tensorflow Variables created in the fully_connected layers.

However, the model we defined has not only the observation (x) but also latent representation (z). This makes it hard for us to compute $p_\theta(x)$, which we call the marginal likelihood of x, because we only know the joint likelihood of the model:

$$p_\theta(x, z) = p_\theta(x|z)p(z)$$

while computing the marginal likelihood requires an integral over latent representation, which is generally intractable:

$$p_\theta(x) = \int p_\theta(x, z)\, dz$$

The intractable integral problem is a fundamental challenge in learning latent variable models like VAEs. Fortunately, the machine learning society has developed many approximate methods to address it. One of them is Variational Inference. As the intuition is very simple, we briefly introduce it below.

Because directly optimizing $\log p_\theta(x)$ is infeasible, we choose to optimize a lower bound of it. The lower bound is constructed as

$$\begin{aligned}\log p_\theta(x) &\geq \log p_\theta(x) - \mathrm{KL}(q_\phi(z|x)\|p_\theta(z|x)) \\ &= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x, z) - \log q_\phi(z|x)\right] \\ &= \mathcal{L}(\theta, \phi)\end{aligned}$$

where $q_\phi(z|x)$ is a user-specified distribution of z (called **variational posterior**) that is chosen to match the true posterior $p_\theta(z|x)$. The lower bound is equal to the marginal log likelihood if and only if $q_\phi(z|x) = p_\theta(z|x)$, when the Kullback–Leibler divergence between them ($\mathrm{KL}(q_\phi(z|x)\|p_\theta(z|x))$) is zero.

Note: In Bayesian Statistics, the process represented by the Bayes' rule

$$p(z|x) = \frac{p(z)(x|z)}{p(x)}$$

is called Bayesian Inference, where $p(z)$ is called the **prior**, $p(x|z)$ is the conditional likelihood, $p(x)$ is the marginal likelihood or **evidence**, and $p(z|x)$ is known as the **posterior**.

This lower bound is usually called Evidence Lower Bound (ELBO). Note that the only probabilities we need to evaluate in it is the joint likelihood and the probability of the variational posterior.

In variational autoencoder, the variational posterior ($q_\phi(z|x)$) is also parameterized by a neural network (g), which accepts input x, and outputs the mean and variance of a Normal distribution:

$$\begin{aligned}\mu_z(x;\phi), \log\sigma_z(x;\phi) &= g_{NN}(x) \\ q_\phi(z|x) &= \mathrm{N}(z|\mu_z(x;\phi), \sigma_z^2(x;\phi))\end{aligned}$$

In ZhuSuan, the variational posterior can also be defined as a *BayesianNet* . The code for above definition is:

```
@zs.reuse_variables(scope="q_net")
def build_q_net(x, z_dim, n_z_per_x):
    bn = zs.BayesianNet()
    h = tf.layers.dense(tf.cast(x, tf.float32), 500, activation=tf.nn.relu)
    h = tf.layers.dense(h, 500, activation=tf.nn.relu)
    z_mean = tf.layers.dense(h, z_dim)
    z_logstd = tf.layers.dense(h, z_dim)
    bn.normal("z", z_mean, logstd=z_logstd, group_ndims=1, n_samples=n_z_per_x)
    return bn

variational = build_q_net(x, z_dim, n_particles)
```

Having both `model` and `variational`, we can build the lower bound as:

```
lower_bound = zs.variational.elbo(
    model, {"x": x}, variational=variational, axis=0)
```

The returned `lower_bound` is an *EvidenceLowerBoundObjective* instance, which is also Tensor-like and can be evaluated directly. However, optimizing this lower bound objective needs special care. The easiest way is to do stochastic gradient descent (SGD), which is very common in deep learning literature. However, the gradient computation here involves taking derivatives of an expectation, which needs Monte Carlo estimation. This often induces large variance if not properly handled.

Note: Directly using auto-differentiation to compute the gradients of *EvidenceLowerBoundObjective* often gives you the wrong results. This is because auto-differentiation is not designed to handle expectations.

Many solutions have been proposed to estimate the gradient of some type of variational lower bound (ELBO or others) with relatively low variance. To make this more automatic and easier to handle, ZhuSuan has wrapped these gradient estimators all into methods of the corresponding variational objective (e.g., the *EvidenceLowerBoundObjective*). These functions don't return gradient estimates but a more convenient surrogate cost. Applying SGD on this surrogate cost with respect to parameters is equivalent to optimizing the corresponding variational lower bounds using the well-developed low-variance estimator.

Here we are using the **Stochastic Gradient Variational Bayes** (SGVB) estimator from the original paper of variational autoencoders [VAEKW13]. This estimator takes benefits of a clever reparameterization trick to greatly reduce the variance when estimating the gradients of ELBO. In ZhuSuan, one can use this estimator by calling the method *sgvb()* of the class:*~zhusuan.variational.exclusive_kl.EvidenceLowerBoundObjective* instance. The code for this part is:

```
# the surrogate cost for optimization
cost = tf.reduce_mean(lower_bound.sgvb())

# the lower bound value to print for monitoring convergence
lower_bound = tf.reduce_mean(lower_bound)
```

Note: For readers who are interested, we provide a detailed explanation of the *sgvb()* estimator used here, though this is not required for you to use ZhuSuan's variational functionality.

The key of SGVB estimator is a reparameterization trick, i.e., they reparameterize the random variable $z \sim q_\phi(z|x) = \mathrm{N}(z|\mu_z(x;\phi), \sigma_z^2(x;\phi))$, as

$$z = z(\epsilon; x, \phi) = \epsilon\sigma_z(x;\phi) + \mu_z(x;\phi),\ \epsilon \sim \mathrm{N}(0, I)$$

In this way, the expectation can be rewritten with respect to ϵ:

$$\begin{aligned}\mathcal{L}(\phi,\theta) &= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right] \\ &= \mathbb{E}_{\epsilon\sim\mathrm{N}(0,I)}\left[\log p_\theta(x, z(\epsilon;x,\phi)) - \log q_\phi(z(\epsilon;x,\phi)|x)\right]\end{aligned}$$

Thus the gradients with variational parameters ϕ can be directly moved into the expectation, enabling an unbiased low-variance Monte Carlo estimator:

$$\begin{aligned}\nabla_\phi L(\phi,\theta) &= \mathbb{E}_{\epsilon\sim\mathrm{N}(0,I)}\nabla_\phi\left[\log p_\theta(x, z(\epsilon;x,\phi)) - \log q_\phi(z(\epsilon;x,\phi)|x)\right] \\ &\approx \frac{1}{k}\sum_{i=1}^{k}\nabla_\phi\left[\log p_\theta(x, z(\epsilon_i;x,\phi)) - \log q_\phi(z(\epsilon_i;x,\phi)|x)\right]\end{aligned}$$

where $\epsilon_i \sim \mathrm{N}(0, I)$

Now that we have had the cost, the next step is to do the stochastic gradient descent. Tensorflow provides many advanced optimizers that improves the plain SGD, among which Adam [VAEKB14] is probably the most popular one in deep learning society. Here we are going to use Tensorflow's Adam optimizer to do the learning:

```
optimizer = tf.train.AdamOptimizer(0.001)
infer_op = optimizer.minimize(cost)
```

1.1.4 Generate images

What we've done above is to define and learn the model. To see how it performs, we would like to let it generate some images in the learning process. To improve the visual quality of generation, we remove the observation noise, i.e., the *Bernoulli* distribution. We do this by using the direct output of the neural network (`x_logits`):

```
@zs.meta_bayesian_net(scope="gen", reuse_variables=True)
def build_gen(x_dim, z_dim, n, n_particles=1):
    bn = zs.BayesianNet()
        ...
    x_logits = tf.layers.dense(h, x_dim)
        ...
```

and adding a sigmoid function to it to get a "mean" image. After that, we add a deterministic node in `bn` to keep track of the Tensor `x_mean`:

```
@zs.meta_bayesian_net(scope="gen", reuse_variables=True)
def build_gen(x_dim, z_dim, n, n_particles=1):
    bn = zs.BayesianNet()
        ...
    x_logits = tf.layers.dense(h, x_dim)
    bn.deterministic("x_mean", tf.sigmoid(x_logits))
        ...
```

so that we can easily access it from a *BayesianNet* instance. For random generations, no observation about the model is made, so we construct the corresponding *BayesianNet* by:

```
bn_gen = model.observe()
```

Then the generated samples can be fetched from the `x_mean` node of `bn_gen`:

```
x_gen = tf.reshape(bn_gen["x_mean"], [-1, 28, 28, 1])
```

1.1.5 Run gradient descent

Now, everything is good before a run. So we could just open the Tensorflow session, run the training loop, print statistics, and write generated images to disk:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch in range(1, epochs + 1):
        time_epoch = -time.time()
        np.random.shuffle(x_train)
        lbs = []
        for t in range(iters):
            x_batch = x_train[t * batch_size:(t + 1) * batch_size]
            _, lb = sess.run([infer_op, lower_bound],
                             feed_dict={x_input: x_batch,
                                        n_particles: 1,
```

(continues on next page)

(continued from previous page)

```
                                        n: batch_size})
            lbs.append(lb)
        time_epoch += time.time()
        print("Epoch {} ({:.1f}s): Lower bound = {}".format(
            epoch, time_epoch, np.mean(lbs)))


        if epoch % save_freq == 0:
            images = sess.run(x_gen, feed_dict={n: 100, n_particles: 1})
            name = os.path.join(result_path,
                                "vae.epoch.{}.png".format(epoch))
            save_image_collections(images, name)
```

Below is a sample image of random generations from the model. Keep watching them and have fun :)

References

1.2 Basic Concepts in ZhuSuan

1.2.1 Distribution

Distributions are basic functionalities for building probabilistic models. The *Distribution* class is the base class for various probabilistic distributions which support batch inputs, generating batches of samples and evaluate probabilities at batches of given values.

The list of all available distributions can be found on these pages:

- *univariate distributions*
- *multivariate distributions*

We can create a univariate Normal distribution in ZhuSuan by:

```
>>> import zhusuan as zs
>>> a = zs.distributions.Normal(mean=0., logstd=0.)
```

The typical input shape for a *Distribution* is like `batch_shape + input_shape`, where `input_shape` represents the shape of a non-batch input parameter; *batch_shape* represents how many independent inputs are fed into the distribution. In general, distributions support broadcasting for inputs.

Samples can be generated by calling *sample()* method of distribution objects. The shape is `([n_samples] + )batch_shape + value_shape`. The first additional axis is omitted only when passed *n_samples* is None (by default), in which case one sample is generated. `value_shape` is the non-batch value shape of the distribution. For a univariate distribution, its `value_shape` is [].

An example of univariate distributions (*Normal*):

```
>>> import tensorflow as tf
>>> _ = tf.InteractiveSession()

>>> b = zs.distributions.Normal([[-1., 1.], [0., -2.]], [0., 1.])

>>> b.batch_shape.eval()
array([2, 2], dtype=int32)

>>> b.value_shape.eval()
array([], dtype=int32)

>>> tf.shape(b.sample()).eval()
array([2, 2], dtype=int32)

>>> tf.shape(b.sample(1)).eval()
array([1, 2, 2], dtype=int32)

>>> tf.shape(b.sample(10)).eval()
array([10,  2,  2], dtype=int32)
```

and an example of multivariate distributions (*OnehotCategorical*):

```
>>> c = zs.distributions.OnehotCategorical([[0., 1., -1.],
...                                         [2., 3., 4.]])

>>> c.batch_shape.eval()
array([2], dtype=int32)

>>> c.value_shape.eval()
array([3], dtype=int32)

>>> tf.shape(c.sample()).eval()
array([2, 3], dtype=int32)

>>> tf.shape(c.sample(1)).eval()
array([1, 2, 3], dtype=int32)

>>> tf.shape(c.sample(10)).eval()
array([10,  2,  3], dtype=int32)
```

There are cases where a batch of random variables are grouped into a single event so that their probabilities can be computed together. This is achieved by setting *group_ndims* argument, which defaults to 0. The last *group_ndims* number of axes in *batch_shape* are grouped into a single event. For example, `Normal(..., group_ndims=1)` will set the last axis of its *batch_shape* to a single event, i.e., a multivariate Normal with identity covariance matrix.

The log probability density (mass) function can be evaluated by passing given values to *log_prob()* method of distribution objects. In that case, the given Tensor should be broadcastable to shape `(... + )batch_shape + value_shape`. The returned Tensor has shape `(... + )batch_shape[:-group_ndims]`. For example:

```
>>> d = zs.distributions.Normal([[-1., 1.], [0., -2.]], 0.,
...                             group_ndims=1)

>>> d.log_prob(0.).eval()
array([-2.83787704, -3.83787727], dtype=float32)

>>> e = zs.distributions.Normal(tf.zeros([2, 1, 3]), 0.,
```

(continues on next page)

(continued from previous page)

```
...                                 group_ndims=2)

>>> tf.shape(e.log_prob(tf.zeros([5, 1, 1, 3]))).eval()
array([5, 2], dtype=int32)
```

1.2.2 BayesianNet

In ZhuSuan we support building probabilistic models as Bayesian networks, i.e., directed graphical models. Below we use a simple Bayesian linear regression example to illustrate this. The generative process of the model is

$$w \sim N(0, \alpha^2 I)$$
$$y \sim N(w^\top x, \beta^2)$$

where x denotes the input feature in the linear regression. We apply a Bayesian treatment and assume a Normal prior distribution of the regression weights w. Suppose the input feature has 5 dimensions. For simplicity we define the input as a placeholder and fix the hyper-parameters:

```
x = tf.placeholder(tf.float32, shape=[5])
alpha = 1.
beta = 0.1
```

To define the model, the first step is to construct a *BayesianNet* instance:

```
bn = zs.BayesianNet()
```

A Bayesian network describes the dependency structure of the joint distribution over a set of random variables as directed graphs. To support this, a *BayesianNet* instance can keep two kinds of nodes:

- Stochastic nodes. They are random variables in graphical models. The `w` node can be constructed as:

  ```
  w = bn.normal("w", tf.zeros([x.shape[-1]], std=alpha)
  ```

  Here `w` is a *StochasticTensor* that follows the *Normal* distribution:

  ```
  >>> print(w)
  <zhusuan.framework.bn.StochasticTensor object at ...
  ```

  For any distribution available in *zhusuan.distributions*, we can find a method of `BayesianNet` for creating the corresponding stochastic node. The returned *StochasticTensor* instances are Tensor-like, which means that you can mix them with almost any Tensorflow primitives, for example, the predicted mean of the linear regression is an inner product between `w` and the input `x`:

  ```
  y_mean = tf.reduce_sum(w * x, axis=-1)
  ```

- Deterministic nodes. As the above code shows, deterministic nodes can be constructed directly with Tensorflow operations, and in this way *BayesianNet* does not keep track of them. However, in some cases it's convenient to enable the tracking by the `deterministic()` method:

  ```
  y_mean = bn.deterministic("y_mean", tf.reduce_sum(w * x, axis=-1))
  ```

  This allows you to fetch the `y_mean` Tensor from `bn` whenever you want it.

The full code of building a Bayesian linear regression model is like:

```
def bayesian_linear_regression(x, alpha, beta):
    bn = zs.BayesianNet()
    w = bn.normal("w", mean=0., std=alpha)
    y_mean = tf.reduce_sum(w * x, axis=-1)
    bn.normal("y", y_mean, std=beta)
    return bn
```

A unique feature of graphical models is that stochastic nodes are allowed to have undetermined behaviour (i.e., being latent), and we can observe them at any time (then they are fixed to the observations). In ZhuSuan, the *BayesianNet* can be initialized with a dictionary argument *observed* to assign observations to certain stochastic nodes, for example:

```
bn = zs.BayesianNet(observed={"w": w_obs})
```

will cause the random variable w to be observed as `w_obs`. The result is that in `bn`, `y_mean` is computed from the observed value of `w` (`w_obs`). For stochastic nodes that are not given observations, their samples will be used when the corresponding *StochasticTensor* is involved in computation with Tensors or fed into Tensorflow operations. In this example it means that if we don't pass any observation to `bn`, the samples of `w` will be used to compute `y_mean`.

Although the above approach allows assigning observations to stochastic nodes, in most common cases, it is more convenient to first define the graphical model, and then pass observations whenever needed. Besides, the model should allow queries with different configurations of observations. To enable this workflow, we introduce a new class *MetaBayesianNet*. Conceptually we can view *MetaBayesianNet* instances as the original model and *BayesianNet* as the result of certain observations. As we shall see, *BayesianNet* instances can be lazily constructed from its meta class instance.

We made it very easy to define the model as a *MetaBayesianNet*. There is no change to the above code but just adding a decorator to the function:

```
@zs.meta_bayesian_net(scope="model")
def bayesian_linear_regression(x, alpha, beta):
    bn = zs.BayesianNet()
    w = bn.normal("w", mean=0., std=alpha)
    y_mean = tf.reduce_sum(w * x, axis=-1)
    bn.normal("y", y_mean, std=beta)
    return bn
```

The function decorated by `zs.meta_bayesian_net()` will return a *MetaBayesianNet* instead of the original *BayesianNet* instance:

```
>>> model = bayesian_linear_regression(x, alpha, beta)

>>> print(model)
<zhusuan.framework.meta_bn.MetaBayesianNet object at ...
```

As we have mentioned, *MetaBayesianNet* can allow different configurations of observations. This is achieved by its *observe()* method. We could pass observations as named arguments, and it will return a corresponding *BayesianNet* instance, for example:

```
bn = model.observe(w=w_obs)
```

will set `w` to be observed in the returned *BayesianNet* instance `bn`. Calling the above function with different named arguments instantiates the *BayesianNet* with different observations, which resembles the common behaviour of probabilistic graphical models.

Note: The observation passed must have the same type and shape as the *StochasticTensor*.

If there are tensorflow Variables created in a model construction function, you may want to reuse them for *BayesianNet* instances with different observations. There is another decorator in ZhuSuan named `reuse_variables()` to make this convenient. You could add it to any function that creates Tensorflow variables:

```
@zs.reuse_variables(scope="model")
def build_model(...):
    bn = zs.BayesianNet()
    ...
    return bn
```

or equivalently, switch on the *reuse_variables* option in the `zs.meta_bayesian_net()` decorator:

```
@zs.meta_bayesian_net(scope="model", reuse_variables=True)
def build_model(...):
    bn = zs.BayesianNet()
    ...
    return bn
```

Up to now we know how to construct a model and reuse it for different observations. After construction, *BayesianNet* supports queries about the current state of the network, such as:

```
# get named node(s)
w = bn["w"]
w, y = bn.get(["w", "y"])

# get log probabilities of stochastic nodes conditioned on the current
# value of other StochasticTensors.
log_pw, log_py = bn.cond_log_prob(["w", "y"])

# get log joint probability given the current values of all stochastic
# nodes
log_joint_value = bn.log_joint()
```

By default the log joint probability is computed by summing over conditional log probabilities at all stochastic nodes. This requires that the distribution batch shapes of all stochastic nodes are correctly aligned. If not, the returned value can be arbitrary. Most of the time you can adjust the *group_ndims* parameter of the stochastic nodes to fix this. If that's not the case, we still allow customizing the log joint probability function by rewriting it in the *MetaBayesianNet* instance like:

```
meta_bn = build_linear_regression(x, alpha, beta)

def customized_log_joint(bn):
    return tf.reduce_sum(
        bn.cond_log_prob("w"), axis=-1) + bn.cond_log_prob("y")

meta_bn.log_joint = customized_log_joint
```

then all *BayesianNet* instances constructed from this `meta_bn` will use the provided customized function to compute the result of `bn.log_joint()`.

1.3 Bayesian Neural Networks

Note: This tutorial assumes that readers have been familiar with ZhuSuan's *basic concepts*.

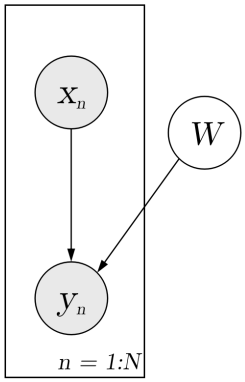
Recent years have seen neural networks' powerful abilities in fitting complex transformations, with successful applications on speech recognition, image classification, and machine translation, etc. However, typical training of neural networks requires lots of labeled data to control the risk of overfitting. And the problem becomes harder when it comes to real world regression tasks. These tasks often have smaller amount of training data to use, and the high-frequency characteristics of these data often makes neural networks easier to get trapped in overfitting.

A principled approach for solving this problem is **Bayesian Neural Networks** (BNN). In BNN, prior distributions are put upon the neural network's weights to consider the modeling uncertainty. By doing Bayesian inference on the weights, one can learn a predictor which both fits to the training data and reasons about the uncertainty of its own prediction on test data. In this tutorial, we show how to implement BNNs in ZhuSuan. The full script for this tutorial is at examples/bayesian_neural_nets/bnn_vi.py.

We use a regression dataset called Boston housing. This has $N = 506$ data points, with $D = 13$ dimensions. The generative process of a BNN for modeling multivariate regression is as follows:

$$
\begin{aligned}
W_i &\sim \mathrm{N}(W_i|0, I), \quad i = 1 \cdots L. \\
y_{mean} &= f_{NN}(x, \{W_i\}_{i=1}^{L}) \\
y &\sim \mathrm{N}(y|y_{mean}, \sigma^2)
\end{aligned}
$$

This generative process starts with an input feature (x), which is forwarded through a deep neural network (f_{NN}) with L layers, whose parameters in each layer (W_i) satisfy a factorized multivariate standard Normal distribution. With this forward transformation, the model is able to learn complex relationships between the input (x) and the output (y). Finally, some noise is added to the output to get a tractable likelihood for the model, which is typically a Gaussian noise in regression problems. A graphical model representation for bayesian neural network is as follows.



1.3.1 Build the model

We start by the model building function (we shall see the meanings of these arguments later):

```
@zs.meta_bayesian_net(scope="bnn", reuse_variables=True)
def build_bnn(x, layer_sizes, n_particles):
    bn = zs.BayesianNet()
```

Following the generative process, we need standard Normal distributions to generate the weights ($\{W_i\}_{i=1}^{L}$) in each layer. For a layer with `n_in` input units and `n_out` output units, the weights are of shape `[n_out, n_in + 1]` (one additional column for bias). To support multiple samples (useful in inference and prediction), a common practice is to set the *n_samples* argument to a placeholder, which we choose to be `n_particles` here:

```
h = tf.tile(x[None, ...], [n_particles, 1, 1])
for i, (n_in, n_out) in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):
    w = bn.normal("w" + str(i), tf.zeros([n_out, n_in + 1]), std=1.,
                  group_ndims=2, n_samples=n_particles)
```

Note that we expand `x` with a new dimension and tile it to enable computation with multiple particles of weight samples. To treat the weights in each layer as a whole and evaluate the probability of them together, `group_ndims` is set to 2. If you are unfamiliar with this property, see *Distribution* for details.

Then we write the feed-forward process of neural networks, through which the connection between output `y` and input `x` is established:

```
for i, (n_in, n_out) in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):
    w = bn.normal("w" + str(i), tf.zeros([n_out, n_in + 1]), std=1.,
                  group_ndims=2, n_samples=n_particles)
    h = tf.concat([h, tf.ones(tf.shape(h)[:-1])[..., None]], -1)
    h = tf.einsum("imk,ijk->ijm", w, h) / tf.sqrt(
        tf.cast(tf.shape(h)[2], tf.float32))
    if i < len(layer_sizes) - 2:
        h = tf.nn.relu(h)
```

Next, we add an observation distribution (noise) to get a tractable likelihood when evaluating the probability:

```
y_mean = bn.deterministic("y_mean", tf.squeeze(h, 2))
y_logstd = tf.get_variable("y_logstd", shape=[],
                           initializer=tf.constant_initializer(0.))
bn.normal("y", y_mean, logstd=y_logstd)
```

Putting together and adding model reuse, the code for constructing a BNN is:

```
@zs.meta_bayesian_net(scope="bnn", reuse_variables=True)
def build_bnn(x, layer_sizes, n_particles):
    bn = zs.BayesianNet()
    h = tf.tile(x[None, ...], [n_particles, 1, 1])
    for i, (n_in, n_out) in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):
        w = bn.normal("w" + str(i), tf.zeros([n_out, n_in + 1]), std=1.,
                      group_ndims=2, n_samples=n_particles)
        h = tf.concat([h, tf.ones(tf.shape(h)[:-1])[..., None]], -1)
        h = tf.einsum("imk,ijk->ijm", w, h) / tf.sqrt(
            tf.cast(tf.shape(h)[2], tf.float32))
        if i < len(layer_sizes) - 2:
            h = tf.nn.relu(h)

    y_mean = bn.deterministic("y_mean", tf.squeeze(h, 2))
    y_logstd = tf.get_variable("y_logstd", shape=[],
                               initializer=tf.constant_initializer(0.))
    bn.normal("y", y_mean, logstd=y_logstd)
    return bn
```

1.3.2 Inference

Having built the model, the next step is to infer the posterior distribution, or uncertainty of weights given the training data.

$$p(W|x_{1:N}, y_{1:N}) \propto p(W) \prod_{n=1}^{N} p(y_n|x_n, W)$$

Because the normalizing constant is intractable, we cannot directly compute the posterior distribution of network parameters ($\{W_i\}_{i=1}^L$). In order to solve this problem, we use Variational Inference, i.e., using a variational distribution $q_\phi(\{W_i\}_{i=1}^L) = \prod_{i=1}^L q_{\phi_i}(W_i)$ to approximate the true posterior. The simplest variational posterior ($q_{\phi_i}(W_i)$) we can specify is factorized (also called mean-field) Normal distribution parameterized by its mean and log standard deviation.

$$q_{\phi_i}(W_i) = \mathrm{N}(W_i|\mu_i, {\sigma_i}^2)$$

The code for above definition is:

```
@zs.reuse_variables(scope="variational")
def build_mean_field_variational(layer_sizes, n_particles):
    bn = zs.BayesianNet()
    for i, (n_in, n_out) in enumerate(zip(layer_sizes[:-1], layer_sizes[1:])):
        w_mean = tf.get_variable(
            "w_mean_" + str(i), shape=[n_out, n_in + 1],
            initializer=tf.constant_initializer(0.))
        w_logstd = tf.get_variable(
            "w_logstd_" + str(i), shape=[n_out, n_in + 1],
            initializer=tf.constant_initializer(0.))
        bn.normal("w" + str(i), w_mean, logstd=w_logstd,
                  n_samples=n_particles, group_ndims=2)
    return bn
```

In Variational Inference, to make $q_\phi(W)$ approximate $p(W|x_{1:N}, y_{1:N})$ well. We need to maximize a lower bound of the marginal log probability ($\log p(y|x)$):

$$\begin{aligned}\log p(y_{1:N}|x_{1:N}) &\geq \log p(y_{1:N}|x_{1:N}) - \mathrm{KL}(q_\phi(W)\|p(W|x_{1:N}, y_{1:N})) \\ &= \mathbb{E}_{q_\phi(W)}\left[\log(p(y_{1:N}|x_{1:N}, W)p(W)) - \log q_\phi(W)\right] \\ &\triangleq \mathcal{L}(\phi)\end{aligned}$$

The lower bound is equal to the marginal log likelihood if and only if $q_\phi(W) = p(W|x_{1:N}, y_{1:N})$, for i in $1 \cdots L$, when the Kullback–Leibler divergence between them ($\mathrm{KL}(q_\phi(W)\|p(W|x_{1:N}, y_{1:N}))$ is zero.

This lower bound is usually called Evidence Lower Bound (ELBO). Note that the only probabilities we need to evaluate in it is the joint likelihood and the probability of the variational posterior. The log conditional likelihood is

$$\log p(y_{1:N}|x_{1:N}, W) = \sum_{n=1}^N \log p(y_n|x_n, W)$$

Computing log conditional likelihood for the whole dataset is very time-consuming. In practice, we sub-sample a minibatch of data to approximate the conditional likelihood

$$\log p(y_{1:N}|x_{1:N}, W) \approx \frac{N}{M} \sum_{m=1}^M \log p(y_m|x_m, W)$$

Here $\{(x_m, y_m)\}_{m=1:M}$ is a subset including M random samples from the training set $\{(x_n, y_n)\}_{n=1:N}$. M is called the batch size. By setting the batch size relatively small, we can compute the lower bound above efficiently.

Note: Different from models like VAEs, BNN's latent variables $\{W_i\}_{i=1}^L$ are global for all the data, therefore we don't explicitly condition W on each data in the variational posterior.

We optimize this lower bound by stochastic gradient descent. As we have done in the *VAE tutorial*, the **Stochastic Gradient Variational Bayes** (SGVB) estimator is used. The code for this part is:

```
model = build_bnn(x, layer_sizes, n_particles)
variational = build_mean_field_variational(layer_sizes, n_particles)

def log_joint(bn):
    log_pws = bn.cond_log_prob(w_names)
    log_py_xw = bn.cond_log_prob('y')
    return tf.add_n(log_pws) + tf.reduce_mean(log_py_xw, 1) * n_train

model.log_joint = log_joint

lower_bound = zs.variational.elbo(
    model, {'y': y}, variational=variational, axis=0)
cost = lower_bound.sgvb()

optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
infer_op = optimizer.minimize(cost)
```

1.3.3 Evaluation

What we've done above is to define the model and infer the parameters. The main purpose of doing this is to predict about new data. The probability distribution of new data (y) given its input feature (x) and our training data (D) is

$$p(y|x, D) = \int_W p(y|x, W)p(W|D)$$

Because we have learned the approximation of $p(W|D)$ by the variational posterior $q(W)$, we can substitute it into the equation

$$p(y|x, D) \simeq \int_W p(y|x, W)q(W)$$

Although the above integral is still intractable, Monte Carlo estimation can be used to get an unbiased estimate of it by sampling from the variational posterior

$$p(y|x, D) \simeq \frac{1}{M}\sum_{i=1}^{M} p(y|x, W^i) \quad W^i \sim q(W)$$

We can choose the mean of this predictive distribution to be our prediction on new data

$$y^{pred} = \mathbb{E}_{p(y|x,D)}\ y \simeq \frac{1}{M}\sum_{i=1}^{M} \mathbb{E}_{p(y|x,W^i)}\ y \quad W^i \sim q(W)$$

The above equation can be implemented by passing the samples from the variational posterior as observations into the model, and averaging over the samples of `y_mean` from the resulting *`BayesianNet`*. The trick here is that the procedure of observing W as samples from $q(W)$ has been implemented when constructing the evidence lower bound, and we can fetch the intermediate *`BayesianNet`* instance by `lower_bound.bn`:

```
# prediction: rmse & log likelihood
y_mean = lower_bound.bn["y_mean"]
y_pred = tf.reduce_mean(y_mean, 0)
```

The predictive mean is given by `y_mean`. To see how this performs, we would like to compute some quantitative measurements including Root Mean Squared Error (RMSE) and log likelihood.

RMSE is defined as the square root of the predictive mean square error, smaller RMSE means better predictive accuracy:

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y_n^{pred} - y_n^{target})^2}$$

Log likelihood (LL) is defined as the natural logarithm of the likelihood function, larger LL means that the learned model fits the test data better:

$$\begin{aligned} LL &= \log p(y|x, D) \\ &\simeq \log \int_W p(y|x, W)q(W) \end{aligned}$$

This can also be computed by Monte Carlo estimation

$$LL \simeq \log \frac{1}{M}\sum_{i=1}^{M} p(y|x, W^i) \quad W^i \sim q(W)$$

To be noted, as we usually standardized the data to make them have unit variance at beginning (check the full script examples/bayesian_neural_nets/bnn_vi.py), we need to count its effect in our evaluation formulas. RMSE is proportional to the amplitude, therefore the final RMSE should be multiplied with the standard deviation. For log likelihood, it needs to be subtracted by a log term. All together, the code for evaluation is:

```
# prediction: rmse & log likelihood
y_mean = lower_bound.bn["y_mean"]
y_pred = tf.reduce_mean(y_mean, 0)
rmse = tf.sqrt(tf.reduce_mean((y_pred - y) ** 2)) * std_y_train
log_py_xw = lower_bound.bn.cond_log_prob("y")
log_likelihood = tf.reduce_mean(zs.log_mean_exp(log_py_xw, 0)) - tf.log(
    std_y_train)
```

1.3.4 Run gradient descent

Again, everything is good before a run. Now add the following codes to run the training loop and see how your BNN performs:

```
# Run the inference
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(1, epochs + 1):
        perm = np.random.permutation(x_train.shape[0])
        x_train = x_train[perm, :]
        y_train = y_train[perm]
        lbs = []
        for t in range(iters):
            x_batch = x_train[t * batch_size:(t + 1) * batch_size]
            y_batch = y_train[t * batch_size:(t + 1) * batch_size]
            _, lb = sess.run(
                [infer_op, lower_bound],
                feed_dict={n_particles: lb_samples,
                           x: x_batch, y: y_batch})
            lbs.append(lb)
        print('Epoch {}: Lower bound = {}'.format(epoch, np.mean(lbs)))
```

(continues on next page)

(continued from previous page)

```
        if epoch % test_freq == 0:
            test_rmse, test_ll = sess.run(
                [rmse, log_likelihood],
                feed_dict={n_particles: ll_samples,
                           x: x_test, y: y_test})
            print('>> TEST')
            print('>> Test rmse = {}, log_likelihood = {}'
                  .format(test_rmse, test_ll))
```

1.4 Logistic Normal Topic Models

The full script for this tutorial is at examples/topic_models/lntm_mcem.py.

1.4.1 An introduction to topic models and Latent Dirichlet Allocation

Nowadays it is much easier to get large corpus of documents. Even if there are no suitable labels with these documents, much information can be extracted. We consider designing a probabilistic model to generate the documents. Generative models can bring more benefits than generating more data. One can also fit the data under some specific structure through generative models. By inferring the parameters in the model (either return a most probable value or figure out its distribution), some valuable information may be discovered.

For example, we can model documents as arising from multiple topics, where a topic is defined to be a distribution over a fixed vocabulary of terms. The most famous model is **Latent Dirichlet Allocation** (LDA) [LNTMBNJ03]. First we describe the notations. Following notations differ from the standard notations in two places for consistence with our notations of LNTM: The topics is denoted $\vec{\phi}$ instead of $\vec{\beta}$, and the scalar Dirichlet prior of topics is δ instead of η. Suppose there are D documents in the corpus, and the dth document has N_d words. Let K be a specified number of topics, V the size of vocabulary, $\vec{\alpha}$ a positive K dimension-vector, and δ a positive scalar. Let $\mathrm{Dir}_K(\vec{\alpha})$ denote a K-dimensional Dirichlet with vector parameter $\vec{\alpha}$ and $\mathrm{Dir}_V(\delta)$ denote a V-dimensional Dirichlet with scalar parameter δ. Let $\mathrm{Catg}(\vec{p})$ be a categorical distribution with vector parameter $\vec{p} = (p_1, p_2, ..., p_n)^T$ ($\sum_{i=1}^n p_i = 1$) and support $\{1, 2, ..., n\}$.

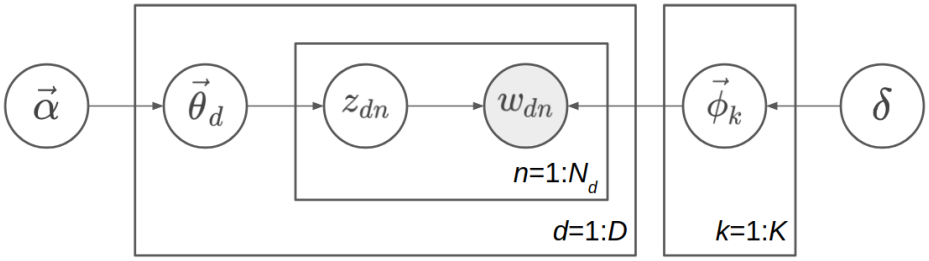
Note: Sometimes, the categorical and multinomial distributions are conflated, and it is common to speak of a "multinomial distribution" when a "categorical distribution" would be more precise. These two distributions are distinguished in ZhuSuan.

The generative process is:

$$
\begin{aligned}
\vec{\phi}_k &\sim \mathrm{Dir}_V(\delta), k = 1, 2, ..., K \\
\vec{\theta}_d &\sim \mathrm{Dir}_K(\vec{\alpha}), d = 1, 2, ..., D \\
z_{dn} &\sim \mathrm{Catg}(\vec{\theta}_d), d = 1, 2, ..., D, n = 1, 2, ..., N_d \\
w_{dn} &\sim \mathrm{Catg}(\vec{\phi}_{z_{dn}}), d = 1, 2, ..., D, n = 1, 2, ..., N_d
\end{aligned}
$$

In more detail, we first sample K **topics** $\{\vec{\phi}_k\}_{k=1}^K$ from the symmetric Dirichlet prior with parameter δ, so each topic is a K-dimensional vector, whose components sum up to 1. These topics are shared among different documents. Then for each document, suppose it is the dth document, we sample a **topic proportion** vector $\vec{\theta}_d$ from the Dirichlet prior with parameter $\vec{\alpha}$, indicating the topic proportion of this document, such as 70% topic 1 and 30% topic 2. Next we start to sample the words in the document. Sampling each word w_{dn} is a two-step process: first, sample the **topic assignment** z_{dn} from the categorical distribution with parameter $\vec{\theta}_d$; secondly, sample the word w_{dn} from the

categorical distribution with parameter $\vec{\phi}_{z_{dn}}$. The range of d is 1 to D, and the range of n is 1 to N_d in the dth document. The model is shown as a directed graphical model in the following figure.



Note: Topic $\{\phi_k\}$, topic proportion $\{\theta_d\}$, and topic assignment $\{z_{dn}\}$ have very different meaning. **Topic** means some distribution over the words in vocabulary. For example,a topic consisting of 10% "game", 5% "hockey", 3% "team", ..., possibly means a topic about sports. They are shared among different documents. A **topic proportion** belongs to a document, roughly indicating the probability distribution of topics in the document. A **topic assignment** belongs to a word in a document, indicating when sampling the word, which topic is sampled first, so the word is sampled from this assigned topic. Both topic, topic proportion, and topic assignment are latent variables which we have not observed. The only observed variable in the generative model is the words $\{w_{dn}\}$, and what Bayesian inference needs to do is to infer the posterior distribution of topic $\{\phi_k\}$, topic proportion $\{\theta_d\}$, and topic assignment $\{z_{dn}\}$.

The key property of LDA is conjugacy between the Dirichlet prior and likelihood. We can write the joint probability distribution as follows:

$$p(w_{1:D,1:N}, z_{1:D,1:N}, \vec{\theta}_{1:D}, \vec{\phi}_{1:K}; \vec{\alpha}, \delta) = \prod_{k=1}^{K} p(\vec{\phi}_k; \delta) \prod_{d=1}^{D} \{p(\vec{\theta}_d; \vec{\alpha}) \prod_{n=1}^{N_d} [p(z_{dn}|\vec{\theta}_d) p(w_{dn}|z_{dn}, \vec{\phi}_{1:K})]\}$$

Here $p(y|x)$ means conditional distribution in which x is a random variable, but $p(y; x)$ means distribution parameterized by x, while x is a fixed value.

We denote $\mathbf{\Theta} = (\vec{\theta}_1, \vec{\theta}_2, ..., \vec{\theta}_D)^T$, $\mathbf{\Phi} = (\vec{\phi}_1, \vec{\phi}_2, ..., \vec{\phi}_K)^T$. Then $\mathbf{\Theta}$ is a $D \times K$ matrix with each row representing topic proportion of one document, while $\mathbf{\Phi}$ is a $K \times V$ matrix with each row representing a topic. We also denote $\mathbf{z} = z_{1:D,1:N}$ and $\mathbf{w} = w_{1:D,1:N}$ for convenience.

Our goal is to do posterior inference from the joint distribution. Since there are three sets of latent variables in the joint distribution: $\mathbf{\Theta}$, $\mathbf{\Phi}$ and $\mathbf{z}$, inferring their posterior distribution at the same time will be difficult, but we can leverage the conjugacy between Dirichlet prior such as $p(\vec{\theta}_d; \vec{\alpha})$ and the multinomial likelihood such as $\prod_{n=1}^{N_d} p(z_{dn}|\vec{\theta}_d)$ (here the multinomial refers to a product of a bunch of categorical distribution, i.e. ignore the normalizing factor of multinomial distribution).

Two ways to leverage this conjugacy are:

(1) Iterate by fixing two sets of latent variables, and do conditional computing for the remaining set. The examples are Gibbs sampling and mean-field variational inference. For Gibbs sampling, each iterating step is fixing the value of samples of two sets, and sample from the conditional distribution of the remaining set. For mean-field variational inference, we often optimize by coordinate ascent: each iterating step is fixing the variational distribution of two sets, and updating the variational distribution of the remaining set based on the parameters of the variational distribution of the two sets. Thanks to the conjugacy, both conditional distribution in Gibbs sampling and conditional update of the variational distribution in variational inference are tractable.

(2) Alternatively, we can integrate out some sets of latent variable before doing further inference. For example, we can integrate out $\mathbf{\Theta}$ and $\mathbf{\Phi}$, remaining the joint distribution $p(\mathbf{w}, \mathbf{z}; \vec{\alpha}, \delta)$ and do Gibbs sampling or variational Bayes on $\mathbf{z}$. After having a estimation to $\mathbf{z}$, we can extract some estimation about $\mathbf{\Phi}$ as the topic information too. These methods are called respectively collapsed Gibbs sampling, and collapsed variational Bayesian inference.

However, conjugacy requires the model being designed carefully. Here, we use a more direct and general method to do Bayesian inference: Monte-Carlo EM, with HMC [LNTMN+11] as the Monte-Carlo sampler.

1.4.2 Logistic Normal Topic Model in ZhuSuan

Integrating out $\mathbf{\Theta}$ and $\mathbf{\Phi}$ requires conjugacy, or the integration is intractable. But integrating $\mathbf{z}$ is always tractable since $\mathbf{z}$ is discrete. Now we have:

$$p(w_{dn} = v|\vec{\theta}_d, \Phi) = \sum_{k=1}^{K} (\vec{\theta}_d)_k \Phi_{kv}$$

More compactly,

$$p(w_{dn}|\vec{\theta}_d, \Phi) = \mathrm{Catg}(\Phi^T \vec{\theta}_d)$$

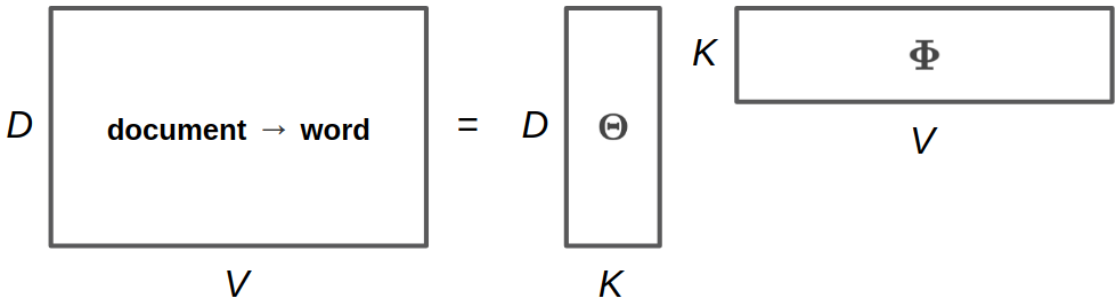
which means when sampling the words in the dth document, the word distribution is the weighted average of all topics, and the weights are the topic proportion of the document.

In LDA we implicitly use the bag-of-words model, and here we make it explicit. Let $\vec{x}_d$ be a V-dimensional vector, $\vec{x}_d = \sum_{n=1}^{N_d} \mathrm{one\_hot}(w_{dn})$. That is, for v from 1 to V, $(\vec{x}_d)_v$ represents the occurence count of the vth word in the document. Denote $\mathbf{X} = (\vec{x}_1, \vec{x}_2, ..., \vec{x}_D)^T$, which is a $D \times V$ matrix. You can verify the following concise formula:

$$\log p(\mathbf{X}|\mathbf{\Theta}, \mathbf{\Phi}) = -\mathrm{CE}(\mathbf{X}, \mathbf{\Theta\Phi})$$

Here, CE means cross entropy, which is defined for matrices as $\mathrm{CE}(\mathbf{A}, \mathbf{B}) = -\sum_{i,j} A_{ij} \log B_{ij}$. Note that $p(\mathbf{X}|\mathbf{\Theta}, \mathbf{\Phi})$ is not a proper distribution; It is a convenient term representing the likelihood of parameters. What we actually means is $\log p(w_{1:D,1:N}|\mathbf{\Theta}, \mathbf{\Phi}) = -\mathrm{CE}(\mathbf{X}, \mathbf{\Theta\Phi})$.

A intuitive demonstration of $\mathbf{\Theta}$, $\mathbf{\Phi}$ and $\mathbf{\Theta\Phi}$ is shown in the following picture. $\mathbf{\Theta}$ is the document-topic matrix, $\mathbf{\Phi}$ is the topic-word matrix, and then $\mathbf{\Theta\Phi}$ is the document-word matrix, which contains the word sampling distribution of each document.



As minimizing the cross entropy encourages $\mathbf{X}$ and $\mathbf{\Theta\Phi}$ to be similar, this may remind you of low-rank matrix factorization. It is natural since topic models can be interpreted as learning "document-topics" parameters and "topic-words" parameters. In fact one of the earliest topic models are solved using SVD, a standard algorithm for low-rank matrix factorization. However, as a probabilistic model, our model is different from matrix factorization by SVD (e.g. the loss function is different). Probabilistic model is more interpretable and can be solved by more algorithms, and Bayesian model can bring the benefits of incorporating prior knowledge and inferring with uncertainty.

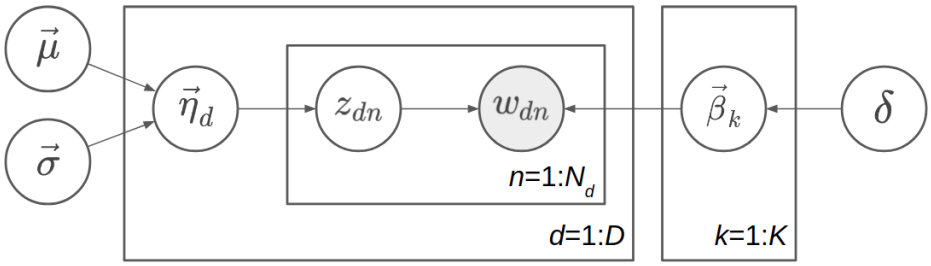
After integrating $\mathbf{z}$, only $\mathbf{\Theta}$ and $\mathbf{\Phi}$ are left, and there is no conjugacy any more. Even if we apply the "conditional computing" trick like Gibbs sampling, no closed-form updating process can be obtained. However, we can adopt the gradient-based method such as HMC and gradient ascent. Note that each row of $\mathbf{\Theta}$ and $\mathbf{\Phi}$ lies on a probability simplex, which is bounded and embedded. It is not common for HMC or gradient ascent to deal with constrained sampling or constrained optimzation. Since we do not nead conjugacy now, we replace the Dirichlet prior with **logistic normal** prior. Now the latent variables live in the whole space $\mathbb{R}^n$.

One may ask why to integrate the parameters $\mathbf{z}$ and lose the conjugacy. That is because our inference technique can also apply to other models which do not have conjugacy from the beginning, such as Neural Variational Document Model ([LNTMMYB16]).

The logistic normal topic model can be described as follows, where $\vec{\beta}_k$ is V-dimensional and $\vec{\eta}_d$ is K-dimensional:

$$\begin{aligned}
\vec{\beta}_k &\sim \mathcal{N}(\vec{0}, \delta^2\mathbf{I}), k = 1, 2, ..., K \\
\vec{\phi}_k &= \text{softmax}(\vec{\beta}_k), k = 1, 2, ..., K \\
\vec{\eta}_d &\sim \mathcal{N}(\vec{\mu}, \text{diag}(\vec{\sigma}^2)), d = 1, 2, ..., D \\
\vec{\theta}_d &= \text{softmax}(\vec{\eta}_d), d = 1, 2, ..., D \\
z_{dn} &\sim \text{Catg}(\vec{\theta}_d), d = 1, 2, ..., D, n = 1, 2, ..., N_d \\
w_{dn} &\sim \text{Catg}(\vec{\phi}_{z_{dn}}), d = 1, 2, ..., D, n = 1, 2, ..., N_d
\end{aligned}$$

The graphical model representation is shown in the following figure.



Since $\vec{\theta}_d$ is a deterministic function of $\vec{\eta}_d$, we can omit one of them in the probabilistic graphical model representation. Here $\vec{\theta}_d$ is omitted because $\vec{\eta}_d$ has a simpler prior. Similarly, we omit $\vec{\phi}_k$ and keep $\vec{\beta}_k$.

Note: Called *Logistic Normal Topic Model*, maybe this reminds you of correlated topic models. However, in our model the normal prior of $\vec{\eta}_d$ has a diagonal covariance matrix $\text{diag}(\vec{\sigma}^2)$, so it cannot model the correlations between different topics in the corpus. However, logistic normal distribution can approximate Dirichlet distribution (see [LNTMSS17]). Hence our model is roughly the same as LDA, while the inference techniques are different.

We denote $\mathbf{H} = (\vec{\eta}_1, \vec{\eta}_2, ..., \vec{\eta}_D)^T$, $\mathbf{B} = (\vec{\beta}_1, \vec{\beta}_2, ..., \vec{\beta}_K)^T$. Then $\mathbf{\Theta} = \text{softmax}(\mathbf{H})$, and $\mathbf{\Phi} = \text{softmax}(\mathbf{B})$. Recall our notation that $\mathbf{X} = (\vec{x}_1, \vec{x}_2, ..., \vec{x}_D)^T$ where $\vec{x}_d = \sum_{n=1}^{N_d} \text{one\_hot}(w_{dn})$. After integrating $\{z_{dn}\}$, the last two lines of the generating process:

$$z_{dn} \sim \text{Catg}(\vec{\theta}_d), w_{dn} \sim \text{Catg}(\vec{\phi}_{z_{dn}})$$

become $\log p(\mathbf{X}|\mathbf{\Theta}, \mathbf{\Phi}) = -\text{CE}(\mathbf{X}, \mathbf{\Theta}\mathbf{\Phi})$. So we can write the joint probability distribution as follows:

$$p(\mathbf{X}, \mathbf{H}, \mathbf{B}; \vec{\mu}, \vec{\sigma}, \delta) = p(\mathbf{B}; \delta)p(\mathbf{H}; \vec{\mu}, \vec{\sigma})p(\mathbf{X}|\mathbf{H}, \mathbf{B})$$

where both $p(\mathbf{B}; \delta)$ and $p(\mathbf{H}; \vec{\mu}, \vec{\sigma})$ are Gaussian distribution and $p(\mathbf{X}|\mathbf{H}, \mathbf{B}) = -\text{CE}(\mathbf{X}, \text{softmax}(\mathbf{H})\text{softmax}(\mathbf{B}))$.

In ZhuSuan, the code for constructing such a model is:

```
@zs.meta_bayesian_net(scope='lntm')
def lntm(n_chains, n_docs, n_topics, n_vocab, eta_mean, eta_logstd):
    bn = zs.BayesianNet()
    eta_mean = tf.tile(tf.expand_dims(eta_mean, 0), [n_docs, 1])
    eta = bn.normal('eta', eta_mean, logstd=eta_logstd, n_samples=n_chains,
                    group_ndims=1)
    theta = tf.nn.softmax(eta)
```

(continues on next page)

(continued from previous page)

```
    beta = bn.normal('beta', tf.zeros([n_topics, n_vocab]),
                     logstd=log_delta, group_ndims=1)
    phi = tf.nn.softmax(beta)
    # doc_word: Document-word matrix
    doc_word = tf.matmul(tf.reshape(theta, [-1, n_topics]), phi)
    doc_word = tf.reshape(doc_word, [n_chains, n_docs, n_vocab])
    bn.unnormalized_multinomial('x', tf.log(doc_word), normalize_logits=False,
                                dtype=tf.float32)
    return bn
```

where `eta_mean` is $\vec{\mu}$, `eta_logstd` is $\log \vec{\sigma}$, `eta` is $\mathbf{H}$ (H is the uppercase letter of η), `theta` is $\mathbf{\Theta} = \mathrm{softmax}(\mathbf{H})$, `beta` is $\mathbf{B}$ (B is the uppercase letter of β), `phi` is $\mathbf{\Phi} = \mathrm{softmax}(\mathbf{B})$, `doc_word` is $\mathbf{\Theta\Phi}$, `x` is $\mathbf{X}$.

Q: What does `UnnormalizedMultinomial` distribution means?

A: `UnnormalizedMultinomial` distribution is not a proper distribution. It means the likelihood of "bags of categorical". To understand this, let's talk about multinomial distribution first. Suppose there are k events $\{1, 2, ..., k\}$ with the probabilities $p_1, p_2, ..., p_k$, and we do n trials, and the count of result being i is x_i. Denote $\vec{x} = (x_1, x_2, ..., x_k)^T$ and $\vec{p} = (p_1, p_2, ..., p_k)^T$. Then $\vec{x}$ follows multinomial distribution: $p(\vec{x}; \vec{p}) = \frac{n!}{x_1!...x_k!} p_1^{x_1} ... p_k^{x_k}$, so $\log p(\vec{x}; \vec{p}) = \log \frac{n!}{x_1!...x_k!} - \mathrm{CE}(\vec{x}, \vec{p})$. However, when we want to optimize the parameter $\vec{p}$, we do not care the first term. On the other hand, if we have a *sequence* of results $\vec{w}$, and the result counts are summarized in $\vec{x}$. Then $\log p(\vec{w}; \vec{p}) = -\mathrm{CE}(\vec{x}, \vec{p})$. The normalizing constant also disappears. Since sometimes we only have access to $\vec{x}$ instead of the actual sequence of results, when we want to optimize w.r.t. the parameters, we can write $\vec{x} \sim \mathrm{UnnormalizedMultinomial}(\vec{p})$, although it is not a proper distribution and we cannot sample from it. `UnnormalizedMultinomial` just means $p(\vec{w}; \vec{p}) = -\mathrm{CE}(\vec{x}, \vec{p})$. In the example of topic models, the situation is also like this.

Q: The shape of `eta` in the model is `n_chains*n_docs*n_topics`. Why we need the first dimension to store its different samples?

A: After introducing the inference method, we should know `eta` is a latent variable which we need to integrate w.r.t. its distribution. In many cases the integration is intractable, so we replace the integration with Monte-Carlo methods, which requires the samples of the latent variable. Therefore we need to construct our model, calculate the joint likelihood and do inference all with the extra dimension storing different samples. In this example, the extra dimension is called "chains" because we utilize the extra dimension to initialize multiple chains and perform HMC evolution on each chain, in order to do parallel sampling and to get independent samples from the posterior.

1.4.3 Inference

Let's analyze the parameters and latent variables in the joint distribution. δ controls the sparsity of the words included in each topic, and larger δ leads to more sparsity. We leave it as a given tunable hyperparameter without the need to optimize. The parameters we need to optimize is $\vec{\mu}$ and $\vec{\sigma}^2$, whose element represents the mean and variance of topic proportion in documents; and $\mathbf{B}$, which represents the topics. For $\vec{\mu}$ and $\vec{\sigma}$, we want to find their **maximum likelihood (MLE)** solution. Unlike $\vec{\mu}$ and $\vec{\sigma}$, $\mathbf{B}$ has a prior, so we could treat it as a random variable and infer its posterior distribution. But here we just find its **maximum a posterior (MAP)** estimation, so we treat it as a parameter and optimize it by gradient ascent instead of inference via HMC. $\mathbf{H}$ is the latent variable, so we want to integrate it out before doing optimization.

Therefore, after integrating $\mathbf{H}$, our optimization problem is:

$$\max_{\mathbf{B}, \vec{\mu}, \vec{\sigma}} \log p(\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma}, \delta)$$

where

$$\begin{aligned}\log p(\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma}, \delta) &= \log p(\mathbf{X}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) + \log p(\mathbf{B}; \delta) \\ &= \log \int_{\mathbf{H}} p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) d\mathbf{H} + \log p(\mathbf{B}; \delta)\end{aligned}$$

The term $\log p(\mathbf{X}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) = \log \int_{\mathbf{H}} p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) d\mathbf{H}$ is **evidence** of the observed data $\mathbf{X}$, given the model with parameters $\mathbf{B}$, $\vec{\mu}$, $\vec{\sigma}$. Computing the integration is intractable, let alone maximize it w.r.t. the parameters. Fortunately, this is the standard form of which we can write an lower bound called **evidence lower bound (ELBO)**:

$$\begin{aligned}\log p(\mathbf{X}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) &\geq \log p(\mathbf{X}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) - \mathrm{KL}(q(\mathbf{H})||p(\mathbf{H}|\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma})) \\ &= \mathbb{E}_{q(\mathbf{H})}[\log p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) - \log q(\mathbf{H})] \\ &= \mathcal{L}(q, \mathbf{B}, \vec{\mu}, \vec{\sigma})\end{aligned}$$

Therefore,

$$\log p(\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma}, \delta) \geq \mathcal{L}(q, \mathbf{B}, \vec{\mu}, \vec{\sigma}) + \log p(\mathbf{B}; \delta)$$

When $q(\mathbf{H}) = p(\mathbf{H}|\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma})$, the lower bound is tight. To do optimization, we can apply coordinate ascent to the lower bound, i.e. **expectation-maximization (EM)** algorithm: We iterate between E-step and M-step.

In E-step, let

$$q(\mathbf{H}) \leftarrow \max_{q} \mathcal{L}(q, \mathbf{B}, \vec{\mu}, \vec{\sigma}) = p(\mathbf{H}|\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma})$$

In M-step, let

$$\begin{aligned}\mathbf{B}, \vec{\mu}, \vec{\sigma} &\leftarrow \max_{\mathbf{B}, \vec{\mu}, \vec{\sigma}} [\mathcal{L}(q, \mathbf{B}, \vec{\mu}, \vec{\sigma}) + \log p(\mathbf{B}; \delta)] \\ &= \max_{\mathbf{B}, \vec{\mu}, \vec{\sigma}} \{\mathbb{E}_{q(\mathbf{H})}[\log p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma})] + \log p(\mathbf{B}; \delta)\}\end{aligned}$$

However, both the posterior $p(\mathbf{H}|\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma})$ in the E step and the integration $\mathbb{E}_{q(\mathbf{H})}[\log p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma})]$ in the M step are intractable. It seems that we have turned an intractable problem into another intractable problem.

We have solutions indeed. Since the difficulty lies in calculating and using the posterior, we can use the whole set of tools in Bayesian inference. Here we use sampling methods, to draw a series of samples $\mathbf{H}^{(1)}, \mathbf{H}^{(2)}, ..., \mathbf{H}^{(S)}$ from $p(\mathbf{H}|\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma})$. Then we let $q(\mathbf{H})$ be the empirical distribution of these samples, as an approximation to the true posterior. Then the M-step becomes:

$$\begin{aligned}\mathbf{B}, \vec{\mu}, \vec{\sigma} &\leftarrow \max_{\mathbf{B}, \vec{\mu}, \vec{\sigma}} [\mathbb{E}_{q(\mathbf{H})}[\log p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma})] + \log p(\mathbf{B}; \delta)] \\ &= \max_{\mathbf{B}, \vec{\mu}, \vec{\sigma}} [\frac{1}{S} \sum_{s=1}^{S} \log p(\mathbf{X}, \mathbf{H}^{(s)}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) + \log p(\mathbf{B}; \delta)]\end{aligned}$$

Now the objective function is tractable to compute. This variant of EM algorithm is called **Monte-Carlo EM**.

We analyze the E-step and M-step in more detail. What sampling method should we choose in E-step? One of the workhorse sampling methods is **Hamiltonian Monte Carlo (HMC)** [LNTMN+11]. Unlike Gibbs sampling which needs a sampler of the conditional distribution, HMC is a black-box method which only requires access to the gradient of log joint distribution at any position, which is almost always tractable as long as the model is differentiable and the latent variable is unconstrained.

To use HMC in ZhuSuan, first define the HMC object with its parameters:

```
hmc = zs.HMC(step_size=1e-3, n_leapfrogs=20, adapt_step_size=True,
             target_acceptance_rate=0.6)
```

Then write the log joint probability $\log p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) = \log p(\mathbf{X}|\mathbf{B}, \mathbf{H}) + p(\mathbf{H}; \vec{\mu}, \vec{\sigma})$:

```
def e_obj(bn):
    return bn.cond_log_prob('eta') + bn.cond_log_prob('x')
```

Given the following defined tensor,

```
x = tf.placeholder(tf.float32, shape=[batch_size, n_vocab], name='x')
eta = tf.Variable(tf.zeros([n_chains, batch_size, n_topics]), name='eta')
beta = tf.Variable(tf.zeros([n_topics, n_vocab]), name='beta')
```

we can define the sampling operator of HMC:

```
model = lntm(n_chains, batch_size, n_topics, n_vocab, eta_mean, eta_logstd)
model.log_joint = e_obj
sample_op, hmc_info = hmc.sample(model,
                                 observed={'x': x, 'beta': beta},
                                 latent={'eta': eta})
```

When running the session, we can run `sample_op` to update the value of `eta`. Note that the first parameter of `hmc.sample` is a *MetaBayesianNet* instance corresponding to the generative model. It could also be a function accepting a Python dictionary containing values of both the observed and latent variables as its argument, and returning the log joint probability. `hmc_info` is a struct containing information about the sampling iteration executed by `sample_op`, such as the acceptance rate.

In the M-step, since $\log p(\mathbf{X}, \mathbf{H}|\mathbf{B}; \vec{\mu}, \vec{\sigma}) = \log p(\mathbf{X}|\mathbf{B}, \mathbf{H}) + p(\mathbf{H}; \vec{\mu}, \vec{\sigma})$, we can write the updating formula in more detail:

$$\vec{\mu}, \vec{\sigma} \leftarrow \max_{\vec{\mu}, \vec{\sigma}} [\frac{1}{S} \sum_{s=1}^{S} \log p(\mathbf{H}^{(s)}; \vec{\mu}, \vec{\sigma})]$$

$$\mathbf{B} \leftarrow \max_{\mathbf{B}} [\frac{1}{S} \sum_{s=1}^{S} \log p(\mathbf{X}|\mathbf{H}^{(s)}, \mathbf{B}) + \log p(\mathbf{B}; \delta)]$$

Then $\vec{\mu}$ and $\vec{\sigma}$ have closed solution by taking the samples of $\mathbf{H}$ as observed data and do maximum likelihood estimation of parameters in Gaussian distribution. $\mathbf{B}$, however, does not have a closed-form solution, so we do optimization using gradient ascent.

The gradient ascent operator of $\mathbf{B}$ can be defined as follows:

```
bn = model.observe(eta=eta, x=x, beta=beta)
log_p_beta, log_px = bn.cond_log_prob(['beta', 'x'])
log_p_beta = tf.reduce_sum(log_p_beta)
log_px = tf.reduce_sum(tf.reduce_mean(log_px, axis=0))
log_joint_beta = log_p_beta + log_px
learning_rate_ph = tf.placeholder(tf.float32, shape=[], name='lr')
optimizer = tf.train.AdamOptimizer(learning_rate_ph)
infer = optimizer.minimize(-log_joint_beta, var_list=[beta])
```

Since when optimizing $\mathbf{B}$, the samples of $\mathbf{H}$ is fixed, `var_list=[beta]` in the last line is necessary.

In the E-step, $p(\mathbf{H}|\mathbf{X}, \mathbf{B}; \vec{\mu}, \vec{\sigma})$ could factorise as $\prod_{d=1}^{D} p(\vec{\eta}_d|\vec{x}_d, \mathbf{B}; \vec{\mu}, \vec{\sigma})$, so we can do sampling for a mini-batch of data given some value of global parameters $\mathbf{B}$, $\vec{\mu}$, and $\vec{\sigma}$. Since the update of $\mathbf{B}$ requires calculating gradients and has a relatively large time cost, we use stochastic gradient ascent to optimize it. That is, after a mini-batch of latent variables are sampled, we do a step of gradient ascent as M-step for $\mathbf{B}$ using the mini-batch chosen in the E-step.

Now we have both the sampling operator for the latent variable `eta` and optimizing operator for the parameter `beta`, while the optimization w.r.t. `eta_mean` and `eta_logstd` is straightforward. Now we can run the EM algorithm.

First, the definition is as follows:

```
iters = X_train.shape[0] // batch_size
Eta = np.zeros((n_chains, X_train.shape[0], n_topics), dtype=np.float32)
Eta_mean = np.zeros(n_topics, dtype=np.float32)
Eta_logstd = np.zeros(n_topics, dtype=np.float32)

eta_mean = tf.placeholder(tf.float32, shape=[n_topics], name='eta_mean')
eta_logstd = tf.placeholder(tf.float32, shape=[n_topics],
                            name='eta_logstd')
eta_ph = tf.placeholder(tf.float32, shape=[n_chains, batch_size, n_topics],
                        name='eta_ph')
init_eta_ph = tf.assign(eta, eta_ph)
```

The key code in an epoch is:

```
time_epoch = -time.time()
lls = []
accs = []
for t in range(iters):
    x_batch = X_train[t*batch_size: (t+1)*batch_size]
    old_eta = Eta[:, t*batch_size: (t+1)*batch_size, :]

    # E step
    sess.run(init_eta_ph, feed_dict={eta_ph: old_eta})
    for j in range(num_e_steps):
        _, new_eta, acc = sess.run(
            [sample_op, hmc_info.samples['eta'],
             hmc_info.acceptance_rate],
            feed_dict={x: x_batch,
                       eta_mean: Eta_mean,
                       eta_logstd: Eta_logstd})
        accs.append(acc)
        # Store eta for the persistent chain
        if j + 1 == num_e_steps:
            Eta[:, t*batch_size: (t+1)*batch_size, :] = new_eta

    # M step
    _, ll = sess.run(
        [infer, log_px],
        feed_dict={x: x_batch,
                   eta_mean: Eta_mean,
                   eta_logstd: Eta_logstd,
                   learning_rate_ph: learning_rate})
    lls.append(ll)

# Update hyper-parameters
Eta_mean = np.mean(Eta, axis=(0, 1))
Eta_logstd = np.log(np.std(Eta, axis=(0, 1)) + 1e-6)

time_epoch += time.time()
print('Epoch {} ({:.1f}s): Perplexity = {:.2f}, acc = {:.3f}, '
      'eta mean = {:.2f}, logstd = {:.2f}'
      .format(epoch, time_epoch,
              np.exp(-np.sum(lls) / np.sum(X_train)),
              np.mean(accs), np.mean(Eta_mean),
              np.mean(Eta_logstd)))
```

We run `num_e_steps` times of E-step before M-step to make samples of HMC closer to the desired equilibrium distribution. We print the mean acceptance rate of HMC to diagnose whether HMC is working properly. If it is too

close to 0 or 1, the quality of samples will often be poor. Moreover, when HMC works properly, we can also tune the acceptance rate to a value for better performance, and the value is usually between 0.6 and 0.9. In the example we set `adapt_step_size=True` and `target_acceptance_rate=0.6` to HMC, so the outputs of actual acceptance rates should be close to 0.6.

Finally we can output the optimized value of `phi` = softmax(`beta`), `eta_mean` and `eta_logstd` to show the learned topics and their proportion in the documents of the corpus:

```
p = sess.run(phi)
for k in range(n_topics):
    rank = list(zip(list(p[k, :]), range(n_vocab)))
    rank.sort()
    rank.reverse()
    sys.stdout.write('Topic {}, eta mean = {:.2f} stdev = {:.2f}: '
                     .format(k, Eta_mean[k], np.exp(Eta_logstd[k])))
    for i in range(10):
        sys.stdout.write(vocab[rank[i][1]] + ' ')
    sys.stdout.write('\n')
```

1.4.4 Evaluation

The `log_likelihood` used to calculate the perplexity may be confusing. Typically, the "likelihood" should refer to the evidence of the observed data given some parameter value, i.e. $p(\mathbf{X}|\mathbf{B};\vec{\mu},\vec{\sigma})$, with the latent variable $\mathbf{H}$ integrated. However, it is even more difficult to compute the marginal likelihood than to do posterior inference. In the code, the likelihood is actually $p(\mathbf{X}|\mathbf{H},\mathbf{B})$, which is not the marginal likelihood; we should integrate it w.r.t. the prior of $\mathbf{H}$ to get marginal likelihood. Hence the perplexity output during the training process will be smaller than the actual value.

After training the model and outputing the topics, the script will run **Annealed Importance Sampling (AIS)** to estimate the marginal likelihood more accurately. It may take some time, and you could turn on the verbose mode of AIS to see its progress. Then our script will output the estimated perplexity which is relatively reliable. We do not introduce AIS here. Readers who are interested could refer to [LNTMNea01].

1.5 zhusuan.distributions

1.5.1 Base class

class Distribution(*dtype*, *param_dtype*, *is_continuous*, *is_reparameterized*, *use_path_derivative=False*, *group_ndims=0*, ***kwargs*)

Bases: `object`

The *Distribution* class is the base class for various probabilistic distributions which support batch inputs, generating batches of samples and evaluate probabilities at batches of given values.

The typical input shape for a *Distribution* is like `batch_shape + input_shape`. where `input_shape` represents the shape of non-batch input parameter, *batch_shape* represents how many independent inputs are fed into the distribution.

Samples generated are of shape `([n_samples]+ )batch_shape + value_shape`. The first additional axis is omitted only when passed *n_samples* is None (by default), in which case one sample is generated. *value_shape* is the non-batch value shape of the distribution. For a univariate distribution, its *value_shape* is [].

There are cases where a batch of random variables are grouped into a single event so that their probabilities should be computed together. This is achieved by setting *group_ndims* argument, which defaults to 0. The last *group_ndims* number of axes in *batch_shape* are grouped into a single event. For example, `Normal(...,`

`group_ndims=1)` will set the last axis of its *batch_shape* to a single event, i.e., a multivariate Normal with identity covariance matrix.

When evaluating probabilities at given values, the given Tensor should be broadcastable to shape `(... + )batch_shape + value_shape`. The returned Tensor has shape `(... + )batch_shape[:-group_ndims]`.

See also:

For more details and examples, please refer to *Basic Concepts in ZhuSuan*.

For both, the parameter *dtype* represents type of samples. For discrete, can be set by user. For continuous, automatically determined from parameter types.

The value type of *prob* and *log_prob* will be *param_dtype* which is deduced from the parameter(s) when initializating. And *dtype* must be among *int16*, *int32*, *int64*, *float16*, *float32* and *float64*.

When two or more parameters are tensors and they have different type, *TypeError* will be raised.

Parameters

- **dtype** – The value type of samples from the distribution.
- **param_dtype** – The parameter(s) type of the distribution.
- **is_continuous** – Whether the distribution is continuous.
- **is_reparameterized** – A bool. Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See above for more detailed explanation.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

1.5.2 Univariate distributions

class Normal(*mean=0.0*, *_sentinel=None*, *std=None*, *logstd=None*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Normal distribution. See *Distribution* for details.

Warning: The order of arguments *logstd/std* has changed to *std/logstd* since 0.3.1. Please use named arguments: `Normal(mean, std=..., ...)` or `Normal(mean, logstd=..., ...)`.

Parameters

- **mean** – A *float* Tensor. The mean of the Normal distribution. Should be broadcastable to match *std* or *logstd*.
- **_sentinel** – Used to prevent positional parameters. Internal, do not use.
- **std** – A *float* Tensor. The standard deviation of the Normal distribution. Should be positive and broadcastable to match *mean*.
- **logstd** – A *float* Tensor. The log standard deviation of the Normal distribution. Should be broadcastable to match *mean*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logstd
The log standard deviation of the Normal distribution.

mean
The mean of the Normal distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

std
The standard deviation of the Normal distribution.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class FoldNormal(*mean=0.0*, *_sentinel=None*, *std=None*, *logstd=None*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate FoldNormal distribution. See *Distribution* for details.

Warning: The order of arguments *logstd/std* has changed to *std/logstd* since 0.3.1. Please use named arguments: `FoldNormal(mean, std=..., ...)` or `FoldNormal(mean, logstd=..., ...)`.

Parameters

- **mean** – A *float* Tensor. The mean of the FoldNormal distribution. Should be broadcastable to match *std* or *logstd*.

- **_sentinel** – Used to prevent positional parameters. Internal, do not use.
- **std** – A *float* Tensor. The standard deviation of the FoldNormal distribution. Should be positive and broadcastable to match *mean*.
- **logstd** – A *float* Tensor. The log standard deviation of the FoldNormal distribution. Should be broadcastable to match *mean*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logstd
The log standard deviation of the FoldNormal distribution.

mean
The mean of the FoldNormal distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

std
The standard deviation of the FoldNormal distribution.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Bernoulli(*logits*, *dtype=tf.int32*, *group_ndims=0*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Bernoulli distribution. See *Distribution* for details.

Parameters

- **logits** – A *float* Tensor. The log-odds of probabilities of being 1.

$$\text{logits} = \log \frac{p}{1-p}$$

- **dtype** – The value type of samples from the distribution. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The log-odds of probabilities of being 1.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Categorical(*logits*, *dtype=tf.int32*, *group_ndims=0*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Categorical distribution. See *Distribution* for details.

Parameters

- **logits** – A N-D (N >= 1) *float32* or *float64* Tensor of shape (..., n_categories). Each slice *[i, j,..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **dtype** – The value type of samples from the distribution. Can be *float32*, *float64*, *int32*, or *int64*. Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

A single sample is a (N-1)-D Tensor with *tf.int32* values in range [0, n_categories).

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

Discrete
alias of *zhusuan.distributions.univariate.Categorical*

class Uniform(*minval=0.0*, *maxval=1.0*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Uniform distribution. See *Distribution* for details.

Parameters

- **minval** – A *float* Tensor. The lower bound on the range of the uniform distribution. Should be broadcastable to match *maxval*.
- **maxval** – A *float* Tensor. The upper bound on the range of the uniform distribution. Should be element-wise bigger than *minval*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

maxval
The upper bound on the range of the uniform distribution.

minval
The lower bound on the range of the uniform distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Gamma(*alpha*, *beta*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Gamma distribution. See *Distribution* for details.

Parameters

- **alpha** – A *float* Tensor. The shape parameter of the Gamma distribution. Should be positive and broadcastable to match *beta*.
- **beta** – A *float* Tensor. The inverse scale parameter of the Gamma distribution. Should be positive and broadcastable to match *alpha*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

alpha
The shape parameter of the Gamma distribution.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

beta
The inverse scale parameter of the Gamma distribution.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Beta(*alpha*, *beta*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Beta distribution. See *Distribution* for details.

Parameters

- **alpha** – A *float* Tensor. One of the two shape parameters of the Beta distribution. Should be positive and broadcastable to match *beta*.
- **beta** – A *float* Tensor. One of the two shape parameters of the Beta distribution. Should be positive and broadcastable to match *alpha*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

alpha
One of the two shape parameters of the Beta distribution.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

beta
One of the two shape parameters of the Beta distribution.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Poisson(*rate*, *dtype=tf.int32*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Poisson distribution. See *Distribution* for details.

Parameters

- **rate** – A *float* Tensor. The rate parameter of Poisson distribution. Must be positive.
- **dtype** – The value type of samples from the distribution. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution.

For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

rate
The rate parameter of Poisson.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Binomial(*logits*, *n_experiments*, *dtype=tf.int32*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Binomial distribution. See *Distribution* for details.

Parameters

- **logits** – A *float* Tensor. The log-odds of probabilities.

$$\text{logits} = \log \frac{p}{1-p}$$

- **n_experiments** – A 0-D *int32* Tensor. The number of experiments for each sample.
- **dtype** – The value type of samples from the distribution. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The log-odds of probabilities.

n_experiments
The number of experiments.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class InverseGamma(*alpha*, *beta*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate InverseGamma distribution. See *Distribution* for details.

Parameters

- **alpha** – A *float* Tensor. The shape parameter of the InverseGamma distribution. Should be positive and broadcastable to match *beta*.
- **beta** – A *float* Tensor. The scale parameter of the InverseGamma distribution. Should be positive and broadcastable to match *alpha*.

- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

alpha
The shape parameter of the InverseGamma distribution.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

beta
The scale parameter of the InverseGamma distribution.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)

Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative

Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape

The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Laplace(*loc*, *scale*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Laplace distribution. See *Distribution* for details.

Parameters

- **loc** – A *float* Tensor. The location parameter of the Laplace distribution. Should be broadcastable to match *scale*.
- **scale** – A *float* Tensor. The scale parameter of the Laplace distribution. Should be positive and broadcastable to match *loc*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape

The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype

The sample type of the distribution.

get_batch_shape()

Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

loc
The location parameter of the Laplace distribution.

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

scale
The scale parameter of the Laplace distribution.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class BinConcrete(*temperature*, *logits*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate BinConcrete distribution from (Maddison, 2016). It is the binary case of *Concrete*. See *Distribution* for details.

See also:

Concrete and *ExpConcrete*

Parameters

- **temperature** – A 0-D *float* Tensor. The temperature of the relaxed distribution. The temperature should be positive.
- **logits** – A *float* Tensor. The log-odds of probabilities of being 1.

$$\text{logits} = \log \frac{p}{1-p}$$

- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The log-odds of probabilities.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

temperature
The temperature of BinConcrete.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

BinGumbelSoftmax
alias of *zhusuan.distributions.univariate.BinConcrete*

1.5.3 Multivariate distributions

class MultivariateNormalCholesky(*mean*, *cov_tril*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.distributions.base.Distribution*

The class of multivariate normal distribution, where covariance is parameterized with the lower triangular matrix L in Cholesky decomposition $LL^T = \Sigma$.

See *Distribution* for details.

Parameters

- **mean** – An N-D *float* Tensor of shape *[..., n_dim]*. Each slice *[i, j, ..., k, :]* represents the mean of a single multivariate normal distribution.
- **cov_tril** – An (N+1)-D *float* Tensor of shape *[..., n_dim, n_dim]*. Each slice *[i, ..., k, :, :]* represents the lower triangular matrix in the Cholesky decomposition of the covariance of a single distribution.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

cov_tril
The lower triangular matrix in the cholosky decomposition of the covariance.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

mean
The mean of the normal distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class Multinomial(*logits*, *n_experiments*, *normalize_logits=True*, *dtype=tf.int32*, *group_ndims=0*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of Multinomial distribution. See *Distribution* for details.

Parameters

- **logits** – A N-D (N >= 1) *float* Tensor of shape *[..., n_categories]*. Each slice *[i, j, ..., k, :]* represents the log probabilities for all categories. By default (when *normal-*

ize_logits=True), the probabilities could be un-normalized.

$$\text{logits} \propto \log p$$

- **n_experiments** – A 0-D *int32* Tensor or *None*. When it is a 0-D *int32* integer, it represents the number of experiments for each sample, which should be invariant among samples. In this situation *_sample* function is supported. When it is *None*, *_sample* function is not supported, and when calculating probabilities the number of experiments will be inferred from *given*, so it could vary among samples.
- **normalize_logits** – A bool indicating whether *logits* should be normalized when computing probability. If you believe *logits* is already normalized, set it to *False* to speed up. Default is True.
- **dtype** – The value type of samples from the distribution. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

A single sample is a N-D Tensor with the same shape as logits. Each slice *[i, j, ..., k, :]* is a vector of counts for all categories.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

n_experiments
The number of experiments for each sample.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class UnnormalizedMultinomial(*logits*, *normalize_logits=True*, *dtype=tf.int32*, *group_ndims=0*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of UnnormalizedMultinomial distribution. UnnormalizedMultinomial distribution calculates probabilities differently from *Multinomial*: It considers the bag-of-words *given* as a statistics of an ordered result sequence, and calculates the probability of the (imagined) ordered sequence. Hence it does not multiply the term

$$\binom{n}{k_1, k_2, \ldots} = \frac{n!}{\prod_i k_i!}$$

See *Distribution* for details.

Parameters

- **logits** – A N-D (N >= 1) *float* Tensor of shape *[..., n_categories]*. Each slice *[i, j, ..., k, :]* represents the log probabilities for all categories. By default (when *normal-*

ize_logits=True), the probabilities could be un-normalized.

$$\text{logits} \propto \log p$$

- **normalize_logits** – A bool indicating whether *logits* should be normalized when computing probability. If you believe *logits* is already normalized, set it to *False* to speed up. Default is True.
- **dtype** – The value type of samples from the distribution. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

A single sample is a N-D Tensor with the same shape as logits. Each slice *[i, j, ..., k, :]* is a vector of counts for all categories.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

BagofCategoricals
alias of *zhusuan.distributions.multivariate.UnnormalizedMultinomial*

class OnehotCategorical(*logits*, *dtype=tf.int32*, *group_ndims=0*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of one-hot Categorical distribution. See *Distribution* for details.

Parameters

- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **dtype** – The value type of samples from the distribution. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

A single sample is a N-D Tensor with the same shape as logits. Each slice *[i, j, ..., k, :]* is a one-hot vector of the selected category.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new

sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

OnehotDiscrete
alias of *zhusuan.distributions.multivariate.OnehotCategorical*

class Dirichlet(*alpha*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of Dirichlet distribution. See *Distribution* for details.

Parameters

- **alpha** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the concentration parameter of a Dirichlet distribution. Should be positive.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

A single sample is a N-D Tensor with the same shape as alpha. Each slice *[i, j, ..., k, :]* of the sample is a vector of probabilities of a Categorical distribution *[x_1, x_2, ...]*, which lies on the simplex

$$\sum_i x_i = 1, 0 < x_i < 1$$

alpha
The concentration parameter of the Dirichlet distribution.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

n_categories
The number of categories in the distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

class ExpConcrete(*temperature*, *logits*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of ExpConcrete distribution from (Maddison, 2016), transformed from *Concrete* by taking logarithm. See *Distribution* for details.

See also:

BinConcrete and *Concrete*

Parameters

- **temperature** – A 0-D *float* Tensor. The temperature of the relaxed distribution. The temperature should be positive.
- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

temperature
The temperature of ExpConcrete.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

ExpGumbelSoftmax
alias of *zhusuan.distributions.multivariate.ExpConcrete*

class Concrete(*temperature*, *logits*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of Concrete (or Gumbel-Softmax) distribution from (Maddison, 2016; Jang, 2016), served as the continuous relaxation of the *OnehotCategorical*. See *Distribution* for details.

See also:

BinConcrete and *ExpConcrete*

Parameters

- **temperature** – A 0-D *float* Tensor. The temperature of the relaxed distribution. The temperature should be positive.
- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

logits
The un-normalized log probabilities.

n_categories
The number of categories in the distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

temperature
The temperature of Concrete.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

GumbelSoftmax
alias of *zhusuan.distributions.multivariate.Concrete*

class MatrixVariateNormalCholesky(*mean*, *u_tril*, *v_tril*, *group_ndims=0*, *is_reparameterized=True*, *use_path_derivative=False*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of matrix variate normal distribution, where covariances U and V are parameterized with the lower triangular matrix in Cholesky decomposition,

$$L_u \text{s.t.} L_u L_u^T = U,\ L_v \text{s.t.} L_v L_v^T = V$$

See *Distribution* for details.

Parameters

- **mean** – An N-D *float* Tensor of shape [..., n_row, n_col]. Each slice [i, j, ..., k, :, :] represents the mean of a single matrix variate normal distribution.
- **u_tril** – An N-D *float* Tensor of shape [..., n_row, n_row]. Each slice [i, j, ..., k, :, :] represents the lower triangular matrix in the Cholesky decomposition of the among-row covariance of a single matrix variate normal distribution.
- **v_tril** – An N-D *float* Tensor of shape [..., n_col, n_col]. Each slice [i, j, ..., k, :, :] represents the lower triangular matrix in the Cholesky decomposition of the among-column covariance of a single matrix variate normal distribution.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **use_path_derivative** – A bool. Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"
- **check_numerics** – Bool. Whether to check numeric issues.

batch_shape
The shape showing how many independent inputs (which we call batches) are fed into the distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`. We borrow this concept from *tf.contrib.distributions*.

dtype
The sample type of the distribution.

get_batch_shape()
Static *batch_shape*.

Returns A *TensorShape* instance.

get_value_shape()
Static *value_shape*.

Returns A *TensorShape* instance.

group_ndims
The number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. See *Distribution* for more detailed explanation.

is_continuous
Whether the distribution is continuous.

is_reparameterized
Whether the gradients of samples can and are allowed to propagate back into inputs, using the reparametrization trick from (Kingma, 2013).

log_prob(*given*)
Compute log probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate log probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

mean
The mean of the matrix variate normal distribution.

param_dtype
The parameter(s) type of the distribution.

path_param(*param*)
Automatically transforms a parameter based on *use_path_derivative*

prob(*given*)
Compute probability density (mass) function at *given* value.

Parameters **given** – A Tensor. The value at which to evaluate probability density (mass) function. Must be able to broadcast to have a shape of `(... + )batch_shape + value_shape`.

Returns A Tensor of shape `(... + )batch_shape[:-group_ndims]`.

sample(*n_samples=None*)
Return samples from the distribution. When *n_samples* is None (by default), one sample of shape `batch_shape + value_shape` is generated. For a scalar *n_samples*, the returned Tensor has a new sample dimension with size *n_samples* inserted at `axis=0`, i.e., the shape of samples is `[n_samples] + batch_shape + value_shape`.

Parameters **n_samples** – A 0-D *int32* Tensor or None. How many independent samples to draw from the distribution.

Returns A Tensor of samples.

u_tril
The lower triangular matrix in the Cholesky decomposition of the among-row covariance.

use_path_derivative
Whether when taking the gradients of the log-probability to propagate them through the parameters of the distribution (False meaning you do propagate them). This is based on the paper "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference"

v_tril
The lower triangular matrix in the Cholesky decomposition of the among-column covariance.

value_shape
The non-batch value shape of a distribution. For batch inputs, the shape of a generated sample is `batch_shape + value_shape`.

1.5.4 Distribution utils

log_combination(*n*, *ks*)
Compute the log combination function.

$$\log \binom{n}{k_1, k_2, \ldots} = \log n! - \sum_i \log k_i!$$

Parameters

- **n** – A N-D *float* Tensor. Can broadcast to match *tf.shape(ks)[:-1]*.
- **ks** – A (N + 1)-D *float* Tensor. Each slice *[i, j, . . . , k, :]* is a vector of *[k_1, k_2, . . .]*.

Returns A N-D Tensor of type same as *n*.

explicit_broadcast(*x*, *y*, *x_name*, *y_name*)
Explicit broadcast two Tensors to have the same shape.

Returns x, y after broadcast.

maybe_explicit_broadcast (*x*, *y*, *x_name*, *y_name*)
Explicit broadcast two Tensors to have the same shape if necessary.

Returns x, y after broadcast.

is_same_dynamic_shape (*x*, *y*)
Whether *x* and *y* has the same dynamic shape.

Parameters

- **x** – A Tensor.
- **y** – A Tensor.

Returns A scalar Tensor of *bool*.

1.6 zhusuan.framework

1.6.1 BayesianNet

class StochasticTensor (*bn*, *name*, *dist*, *observation=None*, ***kwargs*)
Bases: *zhusuan.utils.TensorArithmeticMixin*

The *StochasticTensor* class represents the stochastic nodes in a *BayesianNet*.

We can use any distribution available in *zhusuan.distributions* to construct a stochastic node in a *BayesianNet*. For example:

```
bn = zs.BayesianNet()
x = bn.normal("x", 0., std=1.)
```

will build a stochastic node in `bn` with the *Normal* distribution. The returned `x` will be a *StochasticTensor*. The second line is equivalent to:

```
dist = zs.distributions.Normal(0., std=1.)
x = bn.stochastic("x", dist)
```

StochasticTensor instances are Tensor-like, which means that they can be passed into any Tensorflow operations. This makes it easy to build Bayesian networks by mixing stochastic nodes and Tensorflow primitives.

See also:

For more information, please refer to *Basic Concepts in ZhuSuan*.

Parameters

- **bn** – A *BayesianNet*.
- **name** – A string. The name of the *StochasticTensor*. Must be unique in a *BayesianNet*.
- **dist** – A *Distribution* instance that determines the distribution used in this stochastic node.
- **observation** – A Tensor, which matches the shape of *dist*. If specified, then the *StochasticTensor* is observed and the *tensor* property will return the *observation*. This argument will overwrite the observation provided in *zhusuan.framework.meta_bn.MetaBayesianNet.observe()*.

- **n_samples** – A 0-D *int32* Tensor. Number of samples generated by this *StochasticTensor*.

bn
The *BayesianNet* where the *StochasticTensor* lives.

Returns A *BayesianNet* instance.

cond_log_p
The conditional log probability of the *StochasticTensor*, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the *StochasticTensor*.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the *StochasticTensor*.

Returns A *Distribution* instance.

dtype
The sample type of the *StochasticTensor*.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the *StochasticTensor* is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the *StochasticTensor*.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The *BayesianNet* where the *StochasticTensor* lives.

Returns A *BayesianNet* instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this *StochasticTensor*.

Returns A `TensorShape` instance.

tensor

The value of this *StochasticTensor*. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class BayesianNet(*observed=None*)

Bases: `zhusuan.framework.bn._BayesianNet`, `zhusuan.framework.utils.Context`

The *BayesianNet* class provides a convenient way to construct Bayesian networks, i.e., directed graphical models.

To start, we create a *BayesianNet* instance:

```
bn = zs.BayesianNet()
```

A *BayesianNet* keeps two kinds of nodes

- deterministic nodes: they are just Tensors, usually the outputs of Tensorflow operations.
- stochastic nodes: they are random variables in graphical models, and can be constructed like

```
w = bn.normal("w", 0., std=alpha)
```

Here `w` is a *StochasticTensor* that follows the *Normal* distribution. For any distribution available in *zhusuan.distributions*, we can find a method of *BayesianNet* for creating the corresponding stochastic node. If you define your own distribution class, then there is a general method *stochastic()* for doing this:

```
dist = CustomizedDistribution()
w = bn.stochastic("w", dist)
```

To observe any stochastic nodes in the network, pass a dictionary mapping of `(name, Tensor)` pairs when constructing *BayesianNet*. This will assign observed values to corresponding *StochasticTensor* s. For example:

```
bn = zs.BayesianNet(observed={"w": w_obs})
```

will set `w` to be observed.

Note: The observation passed must have the same type and shape as the *StochasticTensor*.

A useful case is that we often need to pass different observations more than once into the Bayesian network, for which we provide *meta_bayesian_net()* decorator and another abstract class *MetaBayesianNet*.

See also:

For more details and examples, please refer to *Basic Concepts in ZhuSuan*.

Parameters **observed** – A dictionary of (string, Tensor) pairs, which maps from names of stochastic nodes to their observed values.

bag_of_categoricals(*name*, *logits*, *normalize_logits=True*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the UnnormalizedMultinomial distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *UnnormalizedMultinomial* for more information about the other arguments.

Returns A *StochasticTensor* instance.

bernoulli(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Bernoulli distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Bernoulli* for more information about the other arguments.

Returns A *StochasticTensor* instance.

beta(*name*, *alpha*, *beta*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Beta distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Beta* for more information about the other arguments.

Returns A *StochasticTensor* instance.

bin_concrete(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the BinConcrete distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *BinConcrete* for more information about the other arguments.

Returns A *StochasticTensor* instance.

bin_gumbel_softmax(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the BinConcrete distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *BinConcrete* for more information about the other arguments.

Returns A *StochasticTensor* instance.

binomial(*name*, *logits*, *n_experiments*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Binomial distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Binomial* for more information about the other arguments.

Returns A *StochasticTensor* instance.

categorical(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Categorical distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Categorical* for more information about the other arguments.

Returns A *StochasticTensor* instance.

concrete(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Concrete distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Concrete* for more information about the other arguments.

Returns A *StochasticTensor* instance.

cond_log_prob(*name_or_names*)

The conditional log probabilities of stochastic nodes, evaluated at their current values (given by *StochasticTensor.tensor*).

Parameters **name_or_names** – A string or a list of strings. Name(s) of the stochastic nodes.

Returns A Tensor or a list of Tensors.

deterministic(*name*, *input_tensor*)

Add a named deterministic node in this *BayesianNet*.

Parameters

- **name** – The name of the deterministic node. Must be unique in a *BayesianNet*.
- **input_tensor** – A Tensor. The value of the deterministic node.

Returns A Tensor. The same as *input_tensor*.

dirichlet(*name*, *alpha*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Dirichlet distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Dirichlet* for more information about the other arguments.

Returns A *StochasticTensor* instance.

discrete(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the Categorical distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Categorical* for more information about the other arguments.

Returns A *StochasticTensor* instance.

exp_concrete(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the ExpConcrete distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *ExpConcrete* for more information about the other arguments.

Returns A *StochasticTensor* instance.

exp_gumbel_softmax(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the ExpConcrete distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *ExpConcrete* for more information about the other arguments.

Returns A *StochasticTensor* instance.

fold_normal(*name*, *mean=0.0*, *_sentinel=None*, *std=None*, *logstd=None*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the FoldNormal distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *FoldNormal* for more information about the other arguments.

Returns A *StochasticTensor* instance.

gamma(*name*, *alpha*, *beta*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the Gamma distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Gamma* for more information about the other arguments.

Returns A *StochasticTensor* instance.

get(*name_or_names*)
Get one or several nodes by name. For a single node, one can also use dictionary-like `bn[name]` to get the node.

Parameters **name_or_names** – A string or a tuple(list) of strings.

Returns A Tensor/*StochasticTensor* or a list of Tensor/*StochasticTensor* s.

classmethod get_context()

classmethod get_contexts()

gumbel_softmax(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the Concrete distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Concrete* for more information about the other arguments.

Returns A *StochasticTensor* instance.

inverse_gamma(*name*, *alpha*, *beta*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the InverseGamma distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *InverseGamma* for more information about the other arguments.

Returns A *StochasticTensor* instance.

laplace(*name*, *loc*, *scale*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Laplace distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Laplace* for more information about the other arguments.

Returns A *StochasticTensor* instance.

local_log_prob(*name_or_names*)

Note: Deprecated in 0.4, will be removed in 0.4.1.

---

log_joint()

The default log joint probability of this *BayesianNet*. It works by summing over all the conditional log probabilities of stochastic nodes evaluated at their current values (samples or observations).

Returns A Tensor.

matrix_variate_normal_cholesky(*name*, *mean*, *u_tril*, *v_tril*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the MatrixVariateNormalCholesky distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *MatrixVariateNormalCholesky* for more information about the other arguments.

Returns A *StochasticTensor* instance.

multinomial(*name*, *logits*, *n_experiments*, *normalize_logits=True*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Multinomial distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Multinomial* for more information about the other arguments.

Returns A *StochasticTensor* instance.

multivariate_normal_cholesky(*name*, *mean*, *cov_tril*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the MultivariateNormalCholesky distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *MultivariateNormalCholesky* for more information about the other arguments.

Returns A *StochasticTensor* instance.

nodes
The dictionary of all named nodes in this *BayesianNet*, including all *StochasticTensor* s and named deterministic nodes.

Returns A dict.

normal(*name*, *mean=0.0*, *_sentinel=None*, *std=None*, *logstd=None*, *group_ndims=0*, *n_samples=None*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the Normal distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Normal* for more information about the other arguments.

Returns A *StochasticTensor* instance.

onehot_categorical(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the OnehotCategorical distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *OnehotCategorical* for more information about the other arguments.

Returns A *StochasticTensor* instance.

onehot_discrete(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the OnehotCategorical distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *OnehotCategorical* for more information about the other arguments.

Returns A *StochasticTensor* instance.

outputs(*name_or_names*)

Note: Deprecated in 0.4, will be removed in 0.4.1.

---

poisson(*name*, *rate*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, *check_numerics=False*, ***kwargs*)
Add a stochastic node in this *BayesianNet* that follows the Poisson distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Poisson* for more information about the other arguments.

Returns A *StochasticTensor* instance.

query(*name_or_names*, *outputs=False*, *local_log_prob=False*)

Note: Deprecated in 0.4, will be removed in 0.4.1.

---

stochastic(*name*, *dist*, ***kwargs*)
Add a stochastic node in this *BayesianNet*.

Parameters

- **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.
- **dist** – The followed distribution.

- **kwargs** – Optional parameters to specify the sampling behaviors,
  - n_samples: A 0-D *int32* Tensor. Number of samples generated.

Returns A *StochasticTensor*.

uniform(*name*, *minval=0.0*, *maxval=1.0*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the Uniform distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *Uniform* for more information about the other arguments.

Returns A *StochasticTensor* instance.

unnormalized_multinomial(*name*, *logits*, *normalize_logits=True*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Add a stochastic node in this *BayesianNet* that follows the UnnormalizedMultinomial distribution.

Parameters **name** – The name of the stochastic node. Must be unique in a *BayesianNet*.

See *UnnormalizedMultinomial* for more information about the other arguments.

Returns A *StochasticTensor* instance.

1.6.2 MetaBayesianNet

class MetaBayesianNet(*f*, *args=None*, *kwargs=None*, *scope=None*, *reuse_variables=False*)

Bases: `object`

A lazy-constructed *BayesianNet*. Conceptually it's better to view *MetaBayesianNet* rather than *BayesianNet* as the model because it can accept different observations through the *observe()* method.

The suggested usage is through the *meta_bayesian_net()* decorator.

See also:

For more information, please refer to *Basic Concepts in ZhuSuan*.

Parameters

- **f** – A function that constructs and returns a *BayesianNet*.
- **args** – A list. Ordered arguments that will be passed into *f*.
- **kwargs** – A dictionary. Named arguments that will be passed into *f*.
- **scope** – A string. The scope name passed to tensorflow variable_scope().
- **reuse_variables** – A bool. Whether to reuse tensorflow Variables in repeated calls of *observe()*.

log_joint

The log joint function of this model. Can be overwritten as:

```
meta_bn = build_model(...)

def log_joint(bn):
    return ...

meta_bn.log_joint = log_joint
```

observe(***kwargs*)
Construct a *BayesianNet* given observations.

Parameters **kwargs** – A dictionary that maps from node names to their observed values.

Returns A *BayesianNet* instance.

meta_bayesian_net(*scope=None*, *reuse_variables=False*)
Transform a function that builds a *BayesianNet* into returning *MetaBayesianNet*.

The suggested usage is as a decorator:

```
@meta_bayesian_net(scope=..., reuse_variables=True)
def build_model(...):
    bn = zs.BayesianNet()
    ...
    return bn
```

The decorated function will return a *MetaBayesianNet* instance instead of a `BayesianNet` instance.

See also:

For more details and examples, please refer to *Basic Concepts in ZhuSuan*.

Parameters

- **scope** – A string. The scope name passed to tensorflow variable_scope().
- **reuse_variables** – A bool. Whether to reuse tensorflow Variables in repeated calls of *MetaBayesianNet.observe()*.

Returns The transformed function.

1.6.3 Utils

get_backward_ops(*seed_tensors*, *treat_as_inputs=None*)
Get backward ops from inputs to *seed_tensors* by topological order.

Parameters

- **seed_tensors** – A Tensor or list of Tensors, for which to get all preceding Tensors.
- **treat_as_inputs** – None or a list of Tensors that is treated as inputs during the search (where to stop searching the backward graph).

Returns A list of tensorflow *Operation* s in topological order.

reuse_variables(*scope*)
A decorator for transparent reuse of tensorflow Variables in a function. The decorated function will automatically create variables the first time they are called and reuse them thereafter.

Note: This decorator is internally implemented by tensorflow's `make_template()` function. See its doc for requirements on the target function.

Parameters **scope** – A string. The scope name passed to tensorflow variable_scope().

reuse(*scope*)
(Deprecated) Alias of *reuse_variables()*.

1.7 zhusuan.variational

1.7.1 Base class

class VariationalObjective(*meta_bn*, *observed*, *latent=None*, *variational=None*)

Bases: *zhusuan.utils.TensorArithmeticMixin*

The base class for variational objectives. You never use this class directly, but instead instantiate one of its subclasses by calling *elbo()*, *importance_weighted_objective()*, or *klpq()*.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

bn

The *BayesianNet* constructed by observing the *meta_bn* with samples from the variational posterior distributions. `None` if the log joint probability function is provided instead of *meta_bn*.

Note: This *BayesianNet* instance is useful when computing predictions with the approximate posterior distribution.

---

meta_bn

The inferred model. A *MetaBayesianNet* instance. `None` if instead log joint probability function is given.

tensor

Return the Tensor representing the value of the variational objective.

variational

The variational family. A *BayesianNet* instance. `None` if instead *latent* is given.

1.7.2 Exclusive KL divergence

elbo(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)

The evidence lower bound (ELBO) objective for variational inference. The returned value is a *EvidenceLowerBoundObjective* instance.

See *EvidenceLowerBoundObjective* for examples of usage.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are

mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.

- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If `None`, no dimension is reduced.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

Returns An *EvidenceLowerBoundObjective* instance.

class EvidenceLowerBoundObjective(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)

Bases: *zhusuan.variational.base.VariationalObjective*

The class that represents the evidence lower bound (ELBO) objective for variational inference. An instance of the class can be constructed by calling *elbo()*:

```
# lower_bound is an EvidenceLowerBoundObjective instance
lower_bound = zs.variational.elbo(
    meta_bn, observed, variational=variational, axis=0)
```

Here `meta_bn` is a *MetaBayesianNet* instance representing the model to be inferred. `variational` is a *BayesianNet* instance that defines the variational family. `axis` is the index of the sample dimension used to estimate the expectation when computing the objective.

Instances of *EvidenceLowerBoundObjective* are Tensor-like. They can be automatically or manually cast into Tensors when fed into Tensorflow Operators and doing computation with Tensors, or when the *tensor* property is accessed. It can also be evaluated like a Tensor:

```
# evaluate the ELBO
with tf.Session() as sess:
    print sess.run(lower_bound, feed_dict=...)
```

Maximizing the ELBO wrt. variational parameters is equivalent to minimizing $KL(q\|p)$, i.e., the KL-divergence between the variational posterior (q) and the true posterior (p). However, this cannot be directly done by calling Tensorflow optimizers on the *EvidenceLowerBoundObjective* instance because of the outer expectation in the true ELBO objective, while our ELBO value at hand is a single or a few sample estimates. The correct way for doing this is by calling the gradient estimator provided by *EvidenceLowerBoundObjective*. Currently there are two of them:

- *sgvb()*: The Stochastic Gradient Variational Bayes (SGVB) estimator, also known as "the reparameterization trick", or "path derivative estimator".
- *reinforce()*: The score function estimator with variance reduction, also known as "REINFORCE", "NVIL", or "likelihood-ratio estimator".

Thus the typical code for doing variational inference is like:

```
# choose a gradient estimator to return the surrogate cost
cost = lower_bound.sgvb()
# or
# cost = lower_bound.reinforce()
```

(continues on next page)

(continued from previous page)

```
# optimize the surrogate cost wrt. variational parameters
optimizer = tf.train.AdamOptimizer(learning_rate)
infer_op = optimizer.minimize(cost, var_list=variational_parameters)
with tf.Session() as sess:
    for _ in range(n_iters):
        _, lb = sess.run([infer_op, lower_bound], feed_dict=...)
```

Note: Don't directly optimize the *EvidenceLowerBoundObjective* instance wrt. variational parameters, i.e., parameters in q. Instead a proper gradient estimator should be chosen to produce the correct surrogate cost to minimize, as shown in the above code snippet.

On the other hand, the ELBO can be used for maximum likelihood learning of model parameters, as it is a lower bound of the marginal log likelihood of observed variables. Because the outer expectation in the ELBO is not related to model parameters, this time it's fine to directly optimize the class instance:

```
# optimize wrt. model parameters
learn_op = optimizer.minimize(-lower_bound, var_list=model_parameters)
# or
# learn_op = optimizer.minimize(cost, var_list=model_parameters)
# both ways are correct
```

Or we can do inference and learning jointly by optimize over both variational and model parameters:

```
# joint inference and learning
infer_and_learn_op = optimizer.minimize(
    cost, var_list=model_and_variational_parameters)
```

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If `None`, no dimension is reduced.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

bn

The *BayesianNet* constructed by observing the *meta_bn* with samples from the variational posterior distributions. `None` if the log joint probability function is provided instead of *meta_bn*.

Note: This *BayesianNet* instance is useful when computing predictions with the approximate posterior

distribution.

---

meta_bn
The inferred model. A *MetaBayesianNet* instance. None if instead log joint probability function is given.

reinforce(*variance_reduction=True*, *baseline=None*, *decay=0.8*)
Implements the score function gradient estimator for the ELBO, with optional variance reduction using moving mean estimate or "baseline". Also known as "REINFORCE" (Williams, 1992), "NVIL" (Mnih, 2014), and "likelihood-ratio estimator" (Glynn, 1990).

It works for all types of latent *StochasticTensor* s.

Note: To use the *reinforce()* estimator, the is_reparameterized property of each reparameterizable latent *StochasticTensor* must be set False.

Parameters

- **variance_reduction** – Bool. Whether to use variance reduction. By default will subtract the learning signal with a moving mean estimation of it. Users can pass an additional customized baseline using the baseline argument, in that way the returned will be a tuple of costs, the former for the gradient estimator, the latter for adapting the baseline.
- **baseline** – A Tensor that can broadcast to match the shape returned by *log_joint*. A trainable estimation for the scale of the elbo value, which is typically dependent on observed values, e.g., a neural network with observed values as inputs. This will be additional.
- **decay** – Float. The moving average decay for variance normalization.

Returns A Tensor. The surrogate cost for Tensorflow optimizers to minimize.

sgvb()
Implements the stochastic gradient variational bayes (SGVB) gradient estimator for the ELBO, also known as "reparameterization trick" or "path derivative estimator".

It only works for latent *StochasticTensor* s that can be reparameterized (Kingma, 2013). For example, Normal and Concrete.

Note: To use the *sgvb()* estimator, the is_reparameterized property of each latent *StochasticTensor* must be True (which is the default setting when they are constructed).

Returns A Tensor. The surrogate cost for Tensorflow optimizers to minimize.

tensor
Return the Tensor representing the value of the variational objective.

variational
The variational family. A *BayesianNet* instance. None if instead *latent* is given.

1.7.3 Inclusive KL divergence

klpq(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)
The inclusive KL objective for variational inference. The returned value is an *InclusiveKLObjective*

instance.

See *InclusiveKLObjective* for examples of usage.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If `None`, no dimension is reduced.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

Returns An *InclusiveKLObjective* instance.

class InclusiveKLObjective(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)

Bases: *zhusuan.variational.base.VariationalObjective*

The class that represents the inclusive KL objective ($KL(p\|q)$, i.e., the KL-divergence between the true posterior p and the variational posterior q). This is the opposite direction of the one ($KL(q\|p)$, or exclusive KL objective) that induces the ELBO objective.

An instance of the class can be constructed by calling *klpq()*:

```
# klpq_obj is an InclusiveKLObjective instance
klpq_obj = zs.variational.klpq(
    meta_bn, observed, variational=variational, axis=axis)
```

Here `meta_bn` is a *MetaBayesianNet* instance representing the model to be inferred. `variational` is a *BayesianNet* instance that defines the variational family. `axis` is the index of the sample dimension used to estimate the expectation when computing the gradients.

Unlike most *VariationalObjective* instances, the instance of *InclusiveKLObjective* cannot be used like a Tensor or evaluated, because in general this objective is not computable.

The only thing one could achieve with this objective is purely for inference, i.e., optimize it wrt. variational parameters (parameters in q). The way to perform this is by calling the supported gradient estimator and getting the surrogate cost to minimize. Currently there is

- *importance()*: The self-normalized importance sampling gradient estimator.

So the typical code for doing variational inference is like:

```
# call the gradient estimator to return the surrogate cost
cost = klpq_obj.importance()

# optimize the surrogate cost wrt. variational parameters
optimizer = tf.train.AdamOptimizer(learning_rate)
infer_op = optimizer.minimize(cost, var_list=variational_parameters)
with tf.Session() as sess:
```

(continues on next page)

(continued from previous page)

```
    for _ in range(n_iters):
        _, lb = sess.run([infer_op, lower_bound], feed_dict=...)
```

Note: The inclusive KL objective is only a criteria for variational inference but not model learning (Optimizing it doesn't do maximum likelihood learning like the ELBO objective does). That means, there is no reason to optimize the surrogate cost wrt. model parameters.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If `None`, no dimension is reduced.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

bn

The *BayesianNet* constructed by observing the *meta_bn* with samples from the variational posterior distributions. `None` if the log joint probability function is provided instead of *meta_bn*.

Note: This *BayesianNet* instance is useful when computing predictions with the approximate posterior distribution.

importance()

Implements the self-normalized importance sampling gradient estimator for variational inference. This was used in the Reweighted Wake-Sleep (RWS) algorithm (Bornschein, 2015) to adapt the proposal, or variational posterior in the importance weighted objective (See *ImportanceWeightedObjective*). Now this estimator is widely used for neural adaptive proposals in importance sampling.

It works for all types of latent *StochasticTensor* s.

Note: To use the *rws()* estimator, the `is_reparameterized` property of each reparameterizable latent *StochasticTensor* must be set False.

Returns A Tensor. The surrogate cost for Tensorflow optimizers to minimize.

meta_bn

The inferred model. A *MetaBayesianNet* instance. `None` if instead log joint probability function is given.

rws()
(Deprecated) Alias of *importance()*.

tensor
Return the Tensor representing the value of the variational objective.

variational
The variational family. A *BayesianNet* instance. None if instead *latent* is given.

1.7.4 Monte Carlo objectives

importance_weighted_objective(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)
The importance weighted objective for variational inference (Burda, 2015). The returned value is an *ImportanceWeightedObjective* instance.

See *ImportanceWeightedObjective* for examples of usage.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of (string, Tensor) pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of (string, Tensor) pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of (string, (Tensor, Tensor)) pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If None, no dimension is reduced.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

Returns An *ImportanceWeightedObjective* instance.

iw_objective(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)
The importance weighted objective for variational inference (Burda, 2015). The returned value is an *ImportanceWeightedObjective* instance.

See *ImportanceWeightedObjective* for examples of usage.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of (string, Tensor) pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of (string, Tensor) pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of (string, (Tensor, Tensor)) pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If None, no dimension is reduced.

- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

Returns An *ImportanceWeightedObjective* instance.

class ImportanceWeightedObjective(*meta_bn*, *observed*, *latent=None*, *axis=None*, *variational=None*)

Bases: *zhusuan.variational.base.VariationalObjective*

The class that represents the importance weighted objective for variational inference (Burda, 2015). An instance of the class can be constructed by calling *importance_weighted_objective()*:

```
# lower_bound is an ImportanceWeightedObjective instance
lower_bound = zs.variational.importance_weighted_objective(
    meta_bn, observed, variational=variational, axis=axis)
```

Here `meta_bn` is a *MetaBayesianNet* instance representing the model to be inferred. `variational` is a *BayesianNet* instance that defines the variational family. `axis` is the index of the sample dimension used to estimate the expectation when computing the objective.

Instances of *ImportanceWeightedObjective* are Tensor-like. They can be automatically or manually cast into Tensors when fed into Tensorflow operations and doing computation with Tensors, or when the *tensor* property is accessed. It can also be evaluated like a Tensor:

```
# evaluate the objective
with tf.Session() as sess:
    print sess.run(lower_bound, feed_dict=...)
```

The objective computes the same importance-sampling based estimate of the marginal log likelihood of observed variables as *is_loglikelihood()*. The difference is that the estimate now serves as a variational objective, since it is also a lower bound of the marginal log likelihood (as long as the number of samples is finite). The variational posterior here is in fact the proposal. As a variational objective, *ImportanceWeightedObjective* provides two gradient estimators for the variational (proposal) parameters:

- *sgvb()*: The Stochastic Gradient Variational Bayes (SGVB) estimator, also known as "the reparameterization trick", or "path derivative estimator".
- *vimco()*: The multi-sample score function estimator with variance reduction, also known as "VIMCO".

The typical code for joint inference and learning is like:

```
# choose a gradient estimator to return the surrogate cost
cost = lower_bound.sgvb()
# or
# cost = lower_bound.vimco()

# optimize the surrogate cost wrt. model and variational
# parameters
optimizer = tf.train.AdamOptimizer(learning_rate)
infer_and_learn_op = optimizer.minimize(
    cost, var_list=model_and_variational_parameters)
with tf.Session() as sess:
    for _ in range(n_iters):
        _, lb = sess.run([infer_op, lower_bound], feed_dict=...)
```

Note: Don't directly optimize the *ImportanceWeightedObjective* instance wrt. to variational parameters, i.e., parameters in q. Instead a proper gradient estimator should be chosen to produce the correct surrogate cost to minimize, as shown in the above code snippet.

Because the outer expectation in the objective is not related to model parameters, it's fine to directly optimize the class instance wrt. model parameters:

```
# optimize wrt. model parameters
learn_op = optimizer.minimize(-lower_bound,
                              var_list=model_parameters)
# or
# learn_op = optimizer.minimize(cost, var_list=model_parameters)
# both ways are correct
```

The above provides a way for users to combine the importance weighted objective with different methods of adapting proposals (q). In this situation the true posterior is a good choice, which indicates that any variational objectives can be used for the adaptation. Specially, when the *klpq()* objective is chosen, this reproduces the Reweighted Wake-Sleep algorithm (Bornschein, 2015) for learning deep generative models.

Parameters

- **meta_bn** – A *MetaBayesianNet* instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *variational* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If `None`, no dimension is reduced.
- **variational** – A *BayesianNet* instance that defines the variational family. *variational* and *latent* are mutually exclusive.

bn

The *BayesianNet* constructed by observing the *meta_bn* with samples from the variational posterior distributions. `None` if the log joint probability function is provided instead of *meta_bn*.

Note: This *BayesianNet* instance is useful when computing predictions with the approximate posterior distribution.

---

meta_bn

The inferred model. A *MetaBayesianNet* instance. `None` if instead log joint probability function is given.

sgvb()

Implements the stochastic gradient variational bayes (SGVB) gradient estimator for the objective, also known as "reparameterization trick" or "path derivative estimator". It was first used for importance weighted objectives in (Burda, 2015), where it's named "IWAE".

It only works for latent *StochasticTensor* s that can be reparameterized (Kingma, 2013). For example, `Normal` and `Concrete`.

Note: To use the *sgvb()* estimator, the `is_reparameterized` property of each latent *StochasticTensor* must be True (which is the default setting when they are constructed).

Returns A Tensor. The surrogate cost for Tensorflow optimizers to minimize.

tensor
Return the Tensor representing the value of the variational objective.

variational
The variational family. A *BayesianNet* instance. None if instead *latent* is given.

vimco()
Implements the multi-sample score function gradient estimator for the objective, also known as "VIMCO", which is named by authors of the original paper (Minh, 2016).

It works for all kinds of latent *StochasticTensor* s.

Note: To use the *vimco()* estimator, the is_reparameterized property of each reparameterizable latent *StochasticTensor* must be set False.

Returns A Tensor. The surrogate cost for Tensorflow optimizers to minimize.

1.8 zhusuan.hmc

class HMCInfo(*samples*, *acceptance_rate*, *updated_step_size*, *init_momentum*, *orig_hamiltonian*, *hamiltonian*, *orig_log_prob*, *log_prob*)
Bases: object

Contains information about a sampling iteration by *HMC*. Users can get fine control of the sampling process by monitoring these statistics.

Note: Attributes provided in this structure must be fetched together with the corresponding sampling operation and should not be fetched anywhere else. Otherwise you would get undefined behaviors.

Parameters

- **samples** – A dictionary of (string, Tensor) pairs. Samples generated by this HMC iteration.
- **acceptance_rate** – A Tensor. The acceptance rate in this iteration.
- **updated_step_size** – A Tensor. The updated step size (by adaptation) after this iteration.
- **init_momentum** – A dictionary of (string, Tensor) pairs. The initial momentum for each latent variable in this sampling iteration.
- **orig_hamiltonian** – A Tensor. The original hamiltonian at the beginning of the iteration.
- **hamiltonian** – A Tensor. The current hamiltonian at the end of the iteration.
- **orig_log_prob** – A Tensor. The log joint probability at the beginning position of the iteration.
- **log_prob** – A Tensor. The current log joint probability at the end position of the iteration.

class HMC(*step_size=1.0*, *n_leapfrogs=10*, *adapt_step_size=None*, *target_acceptance_rate=0.8*, *gamma=0.05*, *t0=100*, *kappa=0.75*, *adapt_mass=None*, *mass_collect_iters=10*, *mass_decay=0.99*)

Hamiltonian Monte Carlo (Neal, 2011) with adaptation for stepsize (Hoffman & Gelman, 2014) and mass. The usage is similar with a Tensorflow optimizer.

The *HMC* class supports running multiple MCMC chains in parallel. To use the sampler, the user first creates a (list of) tensorflow *Variable* storing the initial sample, whose shape is `chain axes + data axes`. There can be arbitrary number of chain axes followed by arbitrary number of data axes. Then the user provides a *log_joint* function which returns a tensor of shape `chain axes`, which is the log joint density for each chain. Finally, the user runs the operation returned by *sample()*, which updates the sample stored in the *Variable*.

Note: Currently we do not support invoking the *sample()* method multiple times per *HMC* class. Please declare one *HMC* class per each invoke of the *sample()* method.

Note: When the adaptations are on, the sampler is not reversible. To guarantee current equilibrium, the user should only turn on the adaptations during the burn-in iterations, and turn them off when collecting samples. To achieve this, the best practice is to set *adapt_step_size* and *adapt_mass* to be placeholders and feed different values (True/False) when needed.

Parameters

- **step_size** – A 0-D *float32* Tensor. Initial step size.
- **n_leapfrogs** – A 0-D *int32* Tensor. Number of leapfrog steps.
- **adapt_step_size** – A *bool* Tensor, if set, indicating whether to adapt the step size.
- **target_acceptance_rate** – A 0-D *float32* Tensor. The desired acceptance rate for adapting the step size.
- **gamma** – A 0-D *float32* Tensor. Parameter for adapting the step size, see (Hoffman & Gelman, 2014).
- **t0** – A 0-D *float32* Tensor. Parameter for adapting the step size, see (Hoffman & Gelman, 2014).
- **kappa** – A 0-D *float32* Tensor. Parameter for adapting the step size, see (Hoffman & Gelman, 2014).
- **adapt_mass** – A *bool* Tensor, if set, indicating whether to adapt the mass, adapt_step_size must be set.
- **mass_collect_iters** – A 0-D *int32* Tensor. The beginning iteration to change the mass.
- **mass_decay** – A 0-D *float32* Tensor. The decay of computing exponential moving variance.

sample(*meta_bn*, *observed*, *latent*)

Return the sampling *Operation* that runs a HMC iteration and the statistics collected during it, given the log joint function (or a *MetaBayesianNet* instance), observed values and latent variables.

Parameters

- **meta_bn** – A function or a *MetaBayesianNet* instance. If it is a function, it accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all *StochasticTensor* names in the model to their observed values. The function should

return a Tensor, representing the log joint likelihood of the model. More conveniently, the user can also provide a *MetaBayesianNet* instance instead of directly providing a log_joint function. Then a log_joint function will be created so that *log_joint(obs) = meta_bn.observe(**obs).log_joint()*.

- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed *StochasticTensor* s to their values.
- **latent** – A dictionary of `(string, Variable)` pairs. Mapping from names of latent *StochasticTensor* s to corresponding tensorflow *Variables* for storing their initial values and samples.

Returns A Tensorflow *Operation* that runs a HMC iteration.

Returns A *HMCInfo* instance that collects sampling statistics during an iteration.

1.9 zhusuan.sgmcmc

class SGMCMC

Bases: `object`

Base class for stochastic gradient MCMC (SGMCMC) algorithms.

SGMCMC is a class of MCMC algorithms which utilize stochastic gradients instead of the true gradients. To deal with the problems brought by stochasticity in gradients, more sophisticated updating scheme, such as SGHMC and SGNHT, were proposed. We provided four SGMCMC algorithms here: SGLD, PSGLD, SGHMC and SGNHT. For SGHMC and SGNHT, we support 2nd-order integrators introduced in (Chen et al., 2015).

The implementation framework is similar to that of *HMC* class. However, SGMCMC algorithms do not include Metropolis update, and typically do not include hyperparameter adaptation.

The usage is the same as that of *HMC* class. Running multiple SGMCMC chains in parallel is supported.

To use the sampler, the user first defines the sampling method and corresponding hyperparameters by calling the subclass *SGLD*, *PSGLD*, *SGHMC* or *SGNHT*. Then the user creates a (list of) tensorflow *Variable* storing the initial sample, whose shape is `chain axes + data axes`. There can be arbitrary number of chain axes followed by arbitrary number of data axes. Then the user provides a *log_joint* function which returns a tensor of shape `chain axes`, which is the log joint density for each chain. Alternatively, the user can also provide a *meta_bn* instance as a description of *log_joint*. Then the user runs the operation returned by *sample()*, which updates the sample stored in the *Variable*.

The typical code for SGMCMC inference is like:

```
sgmcmc = zs.SGHMC(learning_rate=2e-6, friction=0.2,
                  n_iter_resample_v=1000, second_order=True)
sample_op, sgmcmc_info = sgmcmc.make_grad_func(meta_bn,
    observed={'x': x, 'y': y}, latent={'w1': w1, 'w2': w2})

with tf.Session() as sess:
    for _ in range(n_iters):
        _, info = sess.run([sample_op, sgmcmc_info],
                           feed_dict=...)
        print("mean_k", info["mean_k"])    # For SGHMC and SGNHT,
                                           # optional
```

After getting the sample_op, the user can feed mini-batches to a data placeholder *observed* so that the gradient is a stochastic gradient. Then the user runs the sample_op like using HMC.

sample (*meta_bn*, *observed*, *latent*)

Return the sampling *Operation* that runs a SGMCMC iteration and the statistics collected during it, given the log joint function (or a `MetaBayesianNet` instance), observed values and latent variables.

Parameters

- **meta_bn** – A function or a `MetaBayesianNet` instance. If it is a function, it accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all *StochasticTensor* names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model. More conveniently, the user can also provide a `MetaBayesianNet` instance instead of directly providing a log_joint function. Then a log_joint function will be created so that *log_joint(obs) = meta_bn.observe(**obs).log_joint()*.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed *StochasticTensor* s to their values.
- **latent** – A dictionary of `(string, Variable)` pairs. Mapping from names of latent *StochasticTensor* s to corresponding tensorflow *Variables* for storing their initial values and samples.

Returns A Tensorflow *Operation* that runs a SGMCMC iteration, called *sample_op*.

Returns

A namedtuple that records some useful values, called *sgmcmc_info*. Suppose the list of keys of *latent* dictionary is `['w1', 'w2']`. Then the typical structure of *sgmcmc_info* is `SGMCMCInfo(attr1={'w1': some value, 'w2': some value}, attr2={'w1': some value, 'w2': some value}, ...)`. Hence, `sgmcmc_info.attr1` is a dictionary containing the quantity *attr1* corresponding to each latent variable in the *latent* dictionary.

sgmcmc_info returned by any SGMCMC algorithm has an attribute *q*, representing the updated values of latent variables. To check out other attributes, see the documentation for the specific subclass below.

class SGLD (*learning_rate*)

Bases: `zhusuan.sgmcmc.SGMCMC`

Subclass of SGMCMC which implements Stochastic Gradient Langevin Dynamics (Welling & Teh, 2011) (SGLD) update. The updating equation implemented below follows Equation (3) in the paper.

Attributes of returned *sgmcmc_info* in `SGMCMC.sample()`:

- **q** - The updated values of latent variables.

Parameters **learning_rate** – A 0-D *float32* Tensor. It can be either a constant or a placeholder for decaying learning rate.

class PSGLD (*learning_rate*, *preconditioner='rms'*, *preconditioner_hparams=None*)

Bases: `zhusuan.sgmcmc.SGLD`

Subclass of SGLD implementing preconditioned stochastic gradient Langevin dynamics, a variant proposed in (Li et al, 2015). We implement the RMSprop preconditioner (Equation (4-5) in the paper). Other preconditioners can be implemented similarly.

Attributes of returned *sgmcmc_info* in `SGMCMC.sample()`:

- **q** - The updated values of latent variables.

Parameters **learning_rate** – A 0-D *float32* Tensor. It can be either a constant or a placeholder for decaying learning rate.

class RMSPreconditioner

HParams
alias of RMSHParams

default_hps = RMSHParams(decay=0.9, epsilon=0.001)

class SGHMC(*learning_rate*, *friction=0.25*, *variance_estimate=0.0*, *n_iter_resample_v=20*, *second_order=True*)
Bases: *zhusuan.sgmcmc.SGMCMC*

Subclass of SGMCMC which implements Stochastic Gradient Hamiltonian Monte Carlo (Chen et al., 2014) (SGHMC) update. Compared to SGLD, it adds a momentum variable to the dynamics. Compared to naive HMC using stochastic gradient which diverges, SGHMC simultanenously adds (often the same amount of) friction and noise to make the dynamics have a stationary distribution. The updating equation implemented below follows Equation (15) in the paper. A 2nd-order integrator introduced in (Chen et al., 2015) is supported.

In the following description, we refer to Eq.(*) as Equation (15) in the SGHMC paper.

Attributes of returned *sgmcmc_info* in *SGMCMC.sample()*:

- **q** - The updated values of latent variables.
- **mean_k** - The mean kinetic energy of updated momentum variables corresponding to the latent variables. Each item is a scalar.

Parameters

- **learning_rate** – A 0-D *float32* Tensor corresponding to η in Eq.(*). Note that it does not scale the same as *learning_rate* in *SGLD* since $\eta = O(\epsilon^2)$ in Eq.(*) where ϵ is the step size. When NaN occurs, please consider decreasing *learning_rate*.
- **friction** – A 0-D *float32* Tensor corresponding to α in Eq.(*). A coefficient which simultaneously decays the momentum and adds an additional noise (hence here the name *friction* is not accurate). Larger *friction* makes the stationary distribution closer to the true posterior since it reduces the effect of stochasticity in the gradient, but slowers mixing of the MCMC chain.
- **variance_estimate** – A 0-D *float32* Tensor corresponding to β in Eq.(*). Just set it to zero if it is hard to estimate the gradient variance well. Note that *variance_estimate* must be smaller than *friction*.
- **n_iter_resample_v** – A 0-D *int32* Tensor. Each *n_iter_resample_v* calls to the sampling operation, the momentum variable will be resampled from the corresponding normal distribution once. Smaller *n_iter_resample_v* may lead to a stationary distribution closer to the true posterior but slowers mixing. If you do not want the momentum variable resampled, set the parameter to None or 0.
- **second_order** – A *bool* Tensor indicating whether to enable the 2nd-order integrator introduced in (Chen et al., 2015) or to use the ordinary 1st-order integrator.

class SGNHT(*learning_rate*, *variance_extra=0.0*, *tune_rate=1.0*, *n_iter_resample_v=None*, *second_order=True*, *use_vector_alpha=True*)
Bases: *zhusuan.sgmcmc.SGMCMC*

Subclass of SGMCMC which implements Stochastic Gradient Nosé-Hoover Thermostat (Ding et al., 2014) (SGNHT) update. It is built upon SGHMC, and it could tune the friction parameter α in SGHMC automati-

cally (here is an abuse of notation: in SGNHT α only refers to the friction coefficient, and the noise term is independent of it (unlike SGHMC)), i.e. it adds a new friction variable to the dynamics. The updating equation implemented below follows Algorithm 2 in the supplementary material of the paper. A 2nd-order integrator introduced in (Chen et al., 2015) is supported.

In the following description, we refer to Eq.(**) as the equation in Algorithm 2 in the SGNHT paper.

Attributes of returned *sgmcmc_info* in `SGMCMC.sample()`:

- **q** - The updated values of latent variables.
- **mean_k** - The mean kinetic energy of updated momentum variables corresponding to the latent variables. If *use_vector_alpha==True*, each item has the same shape as the corresponding latent variable; else, each item is a scalar.
- **alpha** - The values of friction variables α corresponding to the latent variables. If *use_vector_alpha==True*, each item has the same shape as the corresponding latent variable; else, each item is a scalar.

Parameters

- **learning_rate** – A 0-D *float32* Tensor corresponding to η in Eq.(**). Note that it does not scale the same as *learning_rate* in `SGLD` since $\eta = O(\epsilon^2)$ in Eq.(*) where ϵ is the step size. When NaN occurs, please consider decreasing *learning_rate*.
- **variance_extra** – A 0-D *float32* Tensor corresponding to a in Eq.(**), representing the additional noise added in the update (and the initial friction α will be set to this value). Normally just set it to zero.
- **tune_rate** – A 0-D *float32* Tensor. In Eq.(**), this parameter is not present (i.e. its value is implicitly set to 1), but a non-1 value is also valid. Higher *tune_rate* represents higher (multiplicative) rate of tuning the friction α.
- **n_iter_resample_v** – A 0-D *int32* Tensor. Each *n_iter_resample_v* calls to the sampling operation, the momentum variable will be resampled from the corresponding normal distribution once. Smaller *n_iter_resample_v* may lead to a stationary distribution closer to the true posterior but slowers mixing. If you do not want the momentum variable resampled, set the parameter to `None` or 0.
- **second_order** – A *bool* Tensor indicating whether to enable the 2nd-order integrator introduced in (Chen et al., 2015) or to use the ordinary 1st-order integrator.
- **use_vector_alpha** – A *bool* Tensor indicating whether to use a vector friction α. If it is true, then the friction has the same shape as the latent variable. That is, each component of the latent variable corresponds to an independently tunable friction. Else, the friction is a scalar.

1.10 zhusuan.evaluation

is_loglikelihood(*meta_bn*, *observed*, *latent=None*, *axis=None*, *proposal=None*)

Marginal log likelihood ($\log p(x)$) estimates using self-normalized importance sampling.

Parameters

- **meta_bn** – A `MetaBayesianNet` instance or a log joint probability function. For the latter, it must accepts a dictionary argument of `(string, Tensor)` pairs, which are mappings from all node names in the model to their observed values. The function should return a Tensor, representing the log joint likelihood of the model.

- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed stochastic nodes to their values.
- **latent** – A dictionary of `(string, (Tensor, Tensor))` pairs. Mapping from names of latent stochastic nodes to their samples and log probabilities. *latent* and *proposal* are mutually exclusive.
- **axis** – The sample dimension(s) to reduce when computing the outer expectation in the objective. If `None`, no dimension is reduced.
- **proposal** – A *BayesianNet* instance that defines the proposal distributions of latent nodes. *proposal* and *latent* are mutually exclusive.

Returns A Tensor. The estimated log likelihood of observed data.

1.11 zhusuan.transform

planar_normalizing_flow(*samples*, *log_probs*, *n_iters*)

Perform Planar Normalizing Flow along the last axis of inputs.

$$f(z_t) = z_{t-1} + h(z_{t-1} * w_t + b_t) * u_t$$

with activation function *tanh* as well as the invertibility trick from (Danilo 2016).

Parameters

- **samples** – A N-D (N>=2) *float32* Tensor of shape *[..., d]*, and planar normalizing flow will be performed along the last axis.
- **log_probs** – A (N-1)-D *float32* Tensor, should be of the same shape as the first N-1 axes of *samples*.
- **n_iters** – A int, which represents the number of successive flows.

Returns A N-D Tensor, the transformed samples.

Returns A (N-1)-D Tensor, the log probabilities of the transformed samples.

1.12 zhusuan.diagnostics

effective_sample_size(*samples*, *burn_in=100*)

Compute the effective sample size of a chain of vector samples, using the algorithm in Stan. Users should flatten their samples as vectors if not so.

Parameters

- **samples** – A 2-D numpy array of shape `(M, D)`, where `M` is the number of samples, and `D` is the number of dimensions of each sample.
- **burn_in** – The number of discarded samples.

Returns A 1-D numpy array. The effective sample size.

effective_sample_size_1d(*samples*)

Compute the effective sample size of a chain of scalar samples.

Parameters **samples** – A 1-D numpy array. The chain of samples.

Returns A float. The effective sample size.

1.13 zhusuan.utils

class TensorArithmeticMixin
Bases: object

Mixin class for implementing tensor arithmetic operations.

The derived class must support *tf.convert_to_tensor*, in order to inherit from this mixin class.

log_mean_exp(*x*, *axis=None*, *keepdims=False*)
Tensorflow numerically stable log mean of exps across the *axis*.

Parameters

- **x** – A Tensor.
- **axis** – An int or list or tuple. The dimensions to reduce. If *None* (the default), reduces all dimensions.
- **keepdims** – Bool. If true, retains reduced dimensions with length 1. Default to be False.

Returns A Tensor after the computation of log mean exp along given axes of x.

merge_dicts(**dict_args*)
Given any number of dicts, shallow copy and merge into a new dict, precedence goes to key value pairs in latter dicts.

1.14 zhusuan.legacy

1.14.1 Special

class Empirical(*dtype*, *batch_shape=None*, *value_shape=None*, *group_ndims=0*, *is_continuous=None*, ***kwargs*)
Bases: zhusuan.distributions.base.Distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of Empirical distribution. Distribution for any variables, which are sampled from an empirical distribution and have no explicit density. You can not sample from the distribution or calculate probabilities and log-probabilities. See Distribution for details.

Parameters

- **dtype** – The value type of samples from the distribution.
- **batch_shape** – A *TensorShape* describing the *batch_shape* of the distribution.
- **value_shape** – A *TensorShape* describing the *value_shape* of the distribution.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See Distribution for more detailed explanation.
- **is_continuous** – A bool or *None*. Whether the distribution is continuous or not. If *None*, will consider it continuous only if *dtype* is a float type.

class Implicit(*samples*, *value_shape=None*, *group_ndims=0*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of Implicit distribution. The distribution abstracts variables whose distribution have no explicit form. A common example of implicit variables are the generated samples from a GAN. See *Distribution* for details.

Parameters

- **samples** – A Tensor.
- **value_shape** – A *TensorShape* describing the *value_shape* of the distribution.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

1.14.2 Stochastic

class Normal(*name*, *mean=0.0*, *_sentinel=None*, *std=None*, *logstd=None*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Normal *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **_sentinel** – Used to prevent positional parameters. Internal, do not use.
- **mean** – A *float* Tensor. The mean of the Normal distribution. Should be broadcastable to match *logstd*.
- **std** – A *float* Tensor. The standard deviation of the Normal distribution. Should be positive and broadcastable to match *mean*.
- **logstd** – A *float* Tensor. The log standard deviation of the Normal distribution. Should be broadcastable to match *mean*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).

- **check_numerics** – Bool. Whether to check numeric issues.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype

The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()

Alias of *shape*.

Returns A TensorShape instance.

is_observed()

Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the StochasticTensor.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class FoldNormal(*name*, *mean=0.0*, *_sentinel=None*, *std=None*, *logstd=None*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate FoldNormal *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.

- **mean** – A *float* Tensor. The mean of the FoldNormal distribution. Should be broadcastable to match *std* or *logstd*.
- **_sentinel** – Used to prevent positional parameters. Internal, do not use.
- **std** – A *float* Tensor. The standard deviation of the FoldNormal distribution. Should be positive and broadcastable to match *mean*.
- **logstd** – A *float* Tensor. The log standard deviation of the FoldNormal distribution. Should be broadcastable to match *mean*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype

The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()

Alias of *shape*.

Returns A TensorShape instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape
Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor
The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Bernoulli(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Bernoulli *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **logits** – A *float* Tensor. The log-odds of probabilities of being 1.

$$\text{logits} = \log \frac{p}{1-p}$$

- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **dtype** – The value type of this *StochasticTensor*. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this StochasticTensor.

Returns A TensorShape instance.

tensor

The value of this StochasticTensor. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Categorical(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Categorical *StochasticTensor*. See StochasticTensor for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j,..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **dtype** – The value type of this *StochasticTensor*. Can be *float32*, *float64*, *int32*, or *int64*. Default is *int32*.

A single sample is a (N-1)-D Tensor with *tf.int32* values in range [0, n_categories).

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype
The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()
Alias of *shape*.

Returns A TensorShape instance.

is_observed()
Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the StochasticTensor.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class OnehotCategorical(*name*, *logits*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of one-hot Categorical *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **dtype** – The value type of this *StochasticTensor*. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.

A single sample is a N-D Tensor with the same shape as logits. Each slice *[i, j, ..., k, :]* is a one-hot vector of the selected category.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

Discrete

alias of *zhusuan.legacy.framework.stochastic.Categorical*

OnehotDiscrete

alias of *zhusuan.legacy.framework.stochastic.OnehotCategorical*

class Uniform(*name*, *minval=0.0*, *maxval=1.0*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Uniform *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.

- **minval** – A *float* Tensor. The lower bound on the range of the uniform distribution. Should be broadcastable to match *maxval*.
- **maxval** – A *float* Tensor. The upper bound on the range of the uniform distribution. Should be element-wise bigger than *minval*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype

The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()

Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()

Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Gamma(*name*, *alpha*, *beta*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Gamma *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **alpha** – A *float* Tensor. The shape parameter of the Gamma distribution. Should be positive and broadcastable to match *beta*.
- **beta** – A *float* Tensor. The inverse scale parameter of the Gamma distribution. Should be positive and broadcastable to match *alpha*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A TensorShape instance.

is_observed()
Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the StochasticTensor.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape
Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor
The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Beta(*name*, *alpha*, *beta*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Beta *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **alpha** – A *float* Tensor. One of the two shape parameters of the Beta distribution. Should be positive and broadcastable to match *beta*.
- **beta** – A *float* Tensor. One of the two shape parameters of the Beta distribution. Should be positive and broadcastable to match *alpha*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Poisson(*name*, *rate*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Poisson *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **rate** – A *float* Tensor. The rate parameter of Poisson distribution. Must be positive.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **dtype** – The value type of this *StochasticTensor*. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype

The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()

Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()

Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Binomial(*name*, *logits*, *n_experiments*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Binomial *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **logits** – A *float* Tensor. The log-odds of probabilities.

$$\text{logits} = \log \frac{p}{1-p}$$

- **n_experiments** – A 0-D *int32* Tensor. The number of experiments for each sample.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

- **dtype** – The value type of this *StochasticTensor*. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype

The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()

Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()

Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class InverseGamma(*name*, *alpha*, *beta*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate InverseGamma *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.

- **alpha** – A *float* Tensor. The shape parameter of the InverseGamma distribution. Should be positive and broadcastable to match *beta*.
- **beta** – A *float* Tensor. The scale parameter of the InverseGamma distribution. Should be positive and broadcastable to match *alpha*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype

The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()

Alias of *shape*.

Returns A TensorShape instance.

is_observed()

Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Laplace(*name*, *loc*, *scale*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate Laplace *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **loc** – A *float* Tensor. The location parameter of the Laplace distribution. Should be broadcastable to match *scale*.
- **scale** – A *float* Tensor. The scale parameter of the Laplace distribution. Should be positive and broadcastable to match *loc*.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class MultivariateNormalCholesky(*name*, *mean*, *cov_tril*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of multivariate normal *StochasticTensor*, where covariance is parameterized with the lower triangular matrix L in Cholesky decomposition $LL^T = \Sigma$.

See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **mean** – An N-D *float* Tensor of shape [..., n_dim]. Each slice [i, j, ..., k, :] represents the mean of a single multivariate normal distribution.
- **cov_tril** – An (N+1)-D *float* Tensor of shape [..., n_dim, n_dim]. Each slice [i, ..., k, :, :] represents the lower triangular matrix in the Cholesky decomposition of the covariance of a single distribution.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype

The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()

Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()

Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class MatrixVariateNormalCholesky(*name*, *mean*, *u_tril*, *v_tril*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of matrix variate normal *StochasticTensor*, where covariances U and V are parameterized with the lower triangular matrix in Cholesky decomposition,

$$L_u \text{s.t.} L_u L_u^T = U,\ L_v \text{s.t.} L_v L_v^T = V$$

See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **mean** – An N-D *float* Tensor of shape [..., n_row, n_col]. Each slice [i, j, ..., k, :, :] represents the mean of a single matrix variate normal distribution.
- **u_tril** – An N-D *float* Tensor of shape [..., n_row, n_row]. Each slice [i, j, ..., k, :, :] represents the lower triangular matrix in the Cholesky decomposition of the among-row covariance of a single matrix variate normal distribution.

- **v_tril** – An N-D *float* Tensor of shape [..., n_col, n_col]. Each slice [i, j, ..., k, :, :] represents the lower triangular matrix in the Cholesky decomposition of the among-column covariance of a single matrix variate normal distribution.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this distribution are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype

The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()

Alias of *shape*.

Returns A TensorShape instance.

is_observed()

Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Multinomial(*name*, *logits*, *n_experiments*, *normalize_logits=True*, *n_samples=None*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of Multinomial *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **logits** – A N-D (N >= 1) *float* Tensor of shape *[..., n_categories]*. Each slice *[i, j, ..., k, :]* represents the log probabilities for all categories. By default (when *normalize_logits=True*), the probabilities could be un-normalized.

$$\text{logits} \propto \log p$$

- **n_experiments** – A 0-D *int32* Tensor or *None*. When it is a 0-D *int32* integer, it represents the number of experiments for each sample, which should be invariant among samples. In this situation *_sample* function is supported. When it is *None*, *_sample* function is not supported, and when calculating probabilities the number of experiments will be inferred from *given*, so it could vary among samples.
- **normalize_logits** – A bool indicating whether *logits* should be normalized when computing probability. If you believe *logits* is already normalized, set it to *False* to speed up. Default is True.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **dtype** – The value type of this *StochasticTensor*. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.

A single sample is a N-D Tensor with the same shape as logits. Each slice *[i, j, ..., k, :]* is a vector of counts for all categories.

bn

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class UnnormalizedMultinomial(*name*, *logits*, *normalize_logits=True*, *group_ndims=0*, *dtype=tf.int32*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of UnnormalizedMultinomial *StochasticTensor*. UnnormalizedMultinomial distribution calculates probabilities differently from *Multinomial*: It considers the bag-of-words *given* as a statistics of an ordered result sequence, and calculates the probability of the (imagined) ordered sequence. Hence it does not multiply the term

$$\binom{n}{k_1, k_2, \ldots} = \frac{n!}{\prod_i k_i!}$$

See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **logits** – A N-D (N >= 1) *float* Tensor of shape *[..., n_categories]*. Each slice *[i, j, ..., k, :]* represents the log probabilities for all categories. By default (when *normalize_logits=True*), the probabilities could be un-normalized.

$$\text{logits} \propto \log p$$

- **normalize_logits** – A bool indicating whether *logits* should be normalized when computing probability. If you believe *logits* is already normalized, set it to *False* to speed up. Default is True.

- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **dtype** – The value type of this *StochasticTensor*. Can be int (*tf.int16*, *tf.int32*, *tf.int64*) or float (*tf.float16*, *tf.float32*, *tf.float64*). Default is *int32*.

A single sample is a N-D Tensor with the same shape as logits. Each slice *[i, j, ..., k, :]* is a vector of counts for all categories.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype

The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()

Alias of *shape*.

Returns A TensorShape instance.

is_observed()

Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape
Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor
The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

BagofCategoricals
alias of *zhusuan.legacy.framework.stochastic.UnnormalizedMultinomial*

class Dirichlet(*name*, *alpha*, *n_samples=None*, *group_ndims=0*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of Dirichlet *StochasticTensor*. See `StochasticTensor` for details.

Parameters

- **`name`** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **`alpha`** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the concentration parameter of a Dirichlet distribution. Should be positive.
- **`n_samples`** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **`group_ndims`** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *`Distribution`* for more detailed explanation.
- **`check_numerics`** – Bool. Whether to check numeric issues.

A single sample is a N-D Tensor with the same shape as alpha. Each slice *[i, j, ..., k, :]* of the sample is a vector of probabilities of a Categorical distribution *[x_1, x_2, ...]*, which lies on the simplex

$$\sum_i x_i = 1, 0 < x_i < 1$$

`bn`

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

`cond_log_p`

The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *`tensor`*).

Returns A Tensor.

`dist`

The distribution followed by the `StochasticTensor`.

Returns A *`Distribution`* instance.

`distribution`

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *`Distribution`* instance.

`dtype`

The sample type of the `StochasticTensor`.

Returns A `DType` instance.

`get_shape``()`

Alias of *`shape`*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape
Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor
The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class BinConcrete(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of univariate BinConcrete *StochasticTensor* from (Maddison, 2016). It is the binary case of *Concrete*. See `StochasticTensor` for details.

See also:

Concrete and *ExpConcrete*

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **temperature** – A 0-D *float* Tensor. The temperature of the relaxed distribution. The temperature should be positive.
- **logits** – A *float* Tensor. The log-odds of probabilities of being 1.

$$\text{logits} = \log \frac{p}{1-p}$$

- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype
The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()
Alias of *shape*.

Returns A TensorShape instance.

is_observed()
Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the StochasticTensor.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

BinGumbelSoftmax

alias of *zhusuan.legacy.framework.stochastic.BinConcrete*

class ExpConcrete(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of ExpConcrete *StochasticTensor* from (Maddison, 2016), transformed from *Concrete* by taking logarithm. See `StochasticTensor` for details.

See also:

BinConcrete and *Concrete*

Parameters

- **temperature** – A 0-D *float* Tensor. The temperature of the relaxed distribution. The temperature should be positive.
- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.

- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype

The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()

Alias of *shape*.

Returns A TensorShape instance.

is_observed()

Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape
Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor
The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

ExpGumbelSoftmax
alias of *zhusuan.legacy.framework.stochastic.ExpConcrete*

class Concrete(*name*, *temperature*, *logits*, *n_samples=None*, *group_ndims=0*, *is_reparameterized=True*, *check_numerics=False*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The class of Concrete (or Gumbel-Softmax) *StochasticTensor* from (Maddison, 2016; Jang, 2016), served as the continuous relaxation of the *OnehotCategorical*. See StochasticTensor for details.

See also:

BinConcrete and *ExpConcrete*

Parameters

- **temperature** – A 0-D *float* Tensor. The temperature of the relaxed distribution. The temperature should be positive.
- **logits** – A N-D (N >= 1) *float* Tensor of shape (..., n_categories). Each slice *[i, j, ..., k, :]* represents the un-normalized log probabilities for all categories.

$$\text{logits} \propto \log p$$

- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **is_reparameterized** – A Bool. If True, gradients on samples from this *StochasticTensor* are allowed to propagate into inputs, using the reparametrization trick from (Kingma, 2013).
- **check_numerics** – Bool. Whether to check numeric issues.

bn

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

cond_log_p

The conditional log probability of the StochasticTensor, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the StochasticTensor.

Returns A *Distribution* instance.

dtype
The sample type of the StochasticTensor.

Returns A DType instance.

get_shape()
Alias of *shape*.

Returns A TensorShape instance.

is_observed()
Whether the StochasticTensor is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the StochasticTensor.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The BayesianNet where the StochasticTensor lives.

Returns A BayesianNet instance.

prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape
Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor
The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

GumbelSoftmax
alias of *zhusuan.legacy.framework.stochastic.Concrete*

class Empirical(*name*, *dtype*, *batch_shape*, *n_samples=None*, *group_ndims=0*, *value_shape=None*, *is_continuous=None*, ***kwargs*)
Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of Empirical *StochasticTensor*. For any inference it is always required that the variables are observed. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **dtype** – The value type of samples from the distribution.
- **batch_shape** – A *TensorShape* describing the *batch_shape* of the distribution.
- **value_shape** – A *TensorShape* describing the *value_shape* of the distribution.
- **is_continuous** – A bool or *None*. Whether the distribution is continuous or not. If *None*, will consider it continuous only if *dtype* is a float type.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype

The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()

Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()

Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name

The name of the `StochasticTensor`.

Returns A string.

net

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

class Implicit(*name*, *samples*, *value_shape=None*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

Bases: *zhusuan.framework.bn.StochasticTensor*

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The class of Implicit *StochasticTensor*. This distribution always sample the implicit tensor provided. See `StochasticTensor` for details.

Parameters

- **name** – A string. The name of the *StochasticTensor*. Must be unique in the *BayesianNet* context.
- **samples** – A N-D (N >= 1) *float* Tensor.
- **value_shape** – A list or tuple describing the *value_shape* of the distribution. The entries of the list can either be int, Dimension or None.
- **group_ndims** – A 0-D *int32* Tensor representing the number of dimensions in *batch_shape* (counted from the end) that are grouped into a single event, so that their probabilities are calculated together. Default is 0, which means a single value is an event. See *Distribution* for more detailed explanation.
- **n_samples** – A 0-D *int32* Tensor or None. Number of samples generated by this *StochasticTensor*.

bn
The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

cond_log_p
The conditional log probability of the `StochasticTensor`, evaluated at its current value (given by *tensor*).

Returns A Tensor.

dist

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

distribution

Warning: Deprecated in 0.4, will be removed in 0.4.1.

The distribution followed by the `StochasticTensor`.

Returns A *Distribution* instance.

dtype
The sample type of the `StochasticTensor`.

Returns A `DType` instance.

get_shape()
Alias of *shape*.

Returns A `TensorShape` instance.

is_observed()
Whether the `StochasticTensor` is observed or not.

Returns A bool.

log_prob(*given*)

Warning: Deprecated in 0.4, will be removed in 0.4.1.

Compute the log probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The log probability value.

name
The name of the `StochasticTensor`.

Returns A string.

net

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

The `BayesianNet` where the `StochasticTensor` lives.

Returns A `BayesianNet` instance.

prob(*given*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Compute the probability density (mass) function of the underlying distribution at the *given* value.

Parameters **given** – A Tensor.

Returns A Tensor. The probability value.

sample(*n_samples*)

> **Warning:** Deprecated in 0.4, will be removed in 0.4.1.

Sample from the underlying distribution.

Parameters **n_samples** – A 0-D *int32* Tensor. The number of samples.

Returns A Tensor.

shape

Return the static shape of this `StochasticTensor`.

Returns A `TensorShape` instance.

tensor

The value of this `StochasticTensor`. If it is observed, then the observation is returned, otherwise samples are returned.

Returns A Tensor.

1.15 Contributing

We always welcome contributions to help make ZhuSuan better. If you would like to contribute, please check out the guidelines here. Below are an incomplete list of our contributors (find more on this page).

- Jiaxin Shi (thjashin)
- Jianfei Chen (cjf00000)
- Shengyang Sun (ssydasheng)
- Yucen Luo (xinmei9322)
- Yihong Gu (wmyw96)
- Yuhao Zhou (miskcoo)
- Ziyu Wang (meta-inf)

- Alexander Botev (botev)
- Shuyu Cheng (csy530216)
- Haowen Xu (korepwx)
- Huajun Wu (CaptainMushroom)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[VAEKW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[VAEKB14] Diederik Kingma and Jimmy Ba. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[LNTMBNJ03] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[LNTMN+11] Radford M Neal and others. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2011.

[LNTMMYB16] Yishu Miao, Lei Yu, and Phil Blunsom. Neural variational inference for text processing. In *International Conference on Machine Learning*, 1727–1736. 2016.

[LNTMSS17] Akash Srivastava and Charles Sutton. Autoencoding variational inference for topic models. *arXiv preprint arXiv:1703.01488*, 2017.

[LNTMNea01] Radford M Neal. Annealed importance sampling. *Statistics and computing*, 11(2):125–139, 2001.

Python Module Index

Z

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

Z