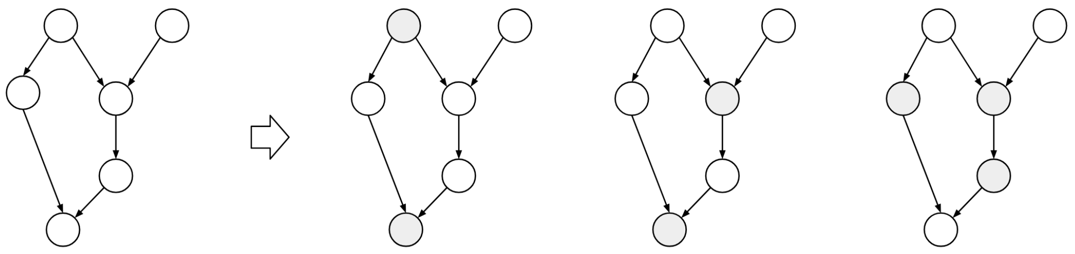
ZhuSuan Documentation

Release 0.0.1

ZhuSuan contributors

Dec 19, 2022

CONTENTS



ZhuSuan-PyTorch 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 PyTorch. Unlike existing deep learning libraries, which are mainly designed for deterministic neural networks and supervised tasks, ZhuSuan-PyTorch 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, etc.).
- MCMC samplers: Stochastic Gradient MCMC (sgmcmc), etc.

CHAPTER

ONE

INSTALLATION

ZhuSuan-PyTorch 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-PyTorch and its dependencies automatically. ZhuSuan-PyTorch is compatible with the lastest version of PyTorch.

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

After installation, open your python console and type:

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

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

1.1 Basic Concepts in ZhuSuan

1.1.1 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.

We can create a univariate Normal distribution in ZhuSuan by:

```
>>> import zhusuan as zs
>>> dist_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 torch
>>> dist_b = zs.distributions.Normal(mean=[[-1., 1.], [0., -2.]], std=[0., 1.])

>>> dist_b.sample().shape
torch.Size([2, 2])

>>> dist_b.sample(10).shape
torch.Size([10, 2, 2])
```

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:

```
>>> dist_c = zs.distributions.Normal(mean=[[-1., 1.], [0., -2.]], std=1., group_
↪ndims=1)

>>> dist_c.log_prob(torch.zeros([1]))
tensor([-2.837877  -3.8378773])

>>> dist_d = zs.distributions.Normal(mean=torch.zeros([2, 1, 3]), std=1.,
...                                  group_ndims=2)

>>> dist_d.log_prob(torch.zeros([5, 1, 1, 3])).shape
torch.Size([5,2,])
```

1.1.2 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 random vector and fix the hyper-parameters:

```
x = torch.rand([5])
alpha = 1.
beta = 0.1
```

To define the model, the first step is to define a subclass of `BayesianNet`:

```
from zhusuan.framework.bn import BayesianNet
class Net(BayesianNet):
    def __init__(self):
        # Initialize...
    def forward(self, observed):
        # Forward propagation...
```

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 = self.stochastic_node('Normal', name="w", mean=torch.zeros([x.shape[-1]]),␣
  ↪std=alpha)
  ```

  Alternatively, to prevent passing wrong parameter to distribution classes(`mean` and `std` are passed to Normal class in the above code), stochastic nodes can be also constructed by:

  ```
  from zhusuan.distributions import Normal

  normal = Normal(mean=torch.zeros([x.shape[-1]]), std=alpha)
  w = self.stochastic_node(normal, name="w")
  # or using alias of stochastic_node method
  w = self.sn(normal, name="w")
  ```

  Here `w` is a *StochasticTensor* that follows the *Normal* distribution, it will be registered to the `nodes` property of the class.

  ```
  >>> print(self.nodes['w'])
  <zhusuan.framework.stochastic_tensor.StochasticTensor object at ...
  ```

  For any distribution available in *zhusuan.distributions*, we can use the name of the distributions and the `stochastic_node` method of `BayesianNet` to create the corresponding stochastic node. The returned variables is an sample of stochastic_node, which means that you can mix them with any Torch operations, for example, the predicted mean of the linear regression is an inner product between `w` and the input `x`:

  ```
  y_mean = torch.sum(w * x, dim=-1)
  ```

- Deterministic nodes. As the above code shows, deterministic nodes can be constructed directly with Torch 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 `cache` property:

  ```
  self.cache['y_mean'] = y_mean
  ```

  This allows you to fetch the `y_mean` Var whenever you want it.

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

```
class bayesian_linear_regression(BayesianNet):
    def __init__(self, alpha, beta):
        super().__init__()
        self.alpha = alpha
        self.beta = beta

    def forward(self, observed):
        self.observe(observed)
        x = self.observed['x']
        w = self.stochastic_node('Normal', name="w", mean=torch.zeros([x.shape[-1]]),␣
↪std=alpha)
        y_mean = torch.sum(w * x, dim=-1)
        y = self.stochastic_node('Normal', name="y", mean=y_mean, std=beta)
        return self
```

Then we can construct an instance of the model:

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

In ZhuSuan-PyTorch, we use a dictionary variable *observed* and the method `observe()` to assign observations to certain stochastic nodes or pass training data to model, for example:

```
model({'w': w_obs, 'x': x})
```

will cause the random variable w to be observed as `w_obs`. The result is that `y_mean` is computed from the observed value of `w` (`w_obs`) and the training data `x` passed by the dictionary variable.

For stochastic nodes that are not given observations, their samples will be used when the corresponding `StochasticTensor` is involved in computation with Vars or fed into Torch operations. In this example it means that if we don't pass any observation of w to the model, the samples of `w` will be used to compute `y_mean`.

After construction, `BayesianNet` supports queries about the current state of the network, such as:

```
# get named node(s)
w = self.nodes['w'].tensor
y = self.nodes['y'].tensor

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

1.2 Variational Autoencoders

Variational Auto-Encoders (VAE) 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_mnist.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.

1.2.1 Build the model

In ZhuSuan, a model is constructed using `BayesianNet`, which describes a directed graphical model, i.e., Bayesian networks.

```
import zhusuan as zs

class Generator(BayesianNet):
    def __init__(self, x_dim, z_dim, batch_size):
        # Initialize...
    def forward(self, observed):
        # Forward propagation...
```

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 `stochastic_node` method to generate samples of shape `[n, z_dim]`:

```
# z ~ N(z|0, I)
mean = torch.zeros([self.batch_size, self.z_dim])
std = torch.ones([self.batch_size, self.z_dim])

z = self.sn('Normal',
            name='z',
            mean=mean,
            std=std,
            reparameterize=False,
            reduce_mean_dims=[0],
            reduce_sum_dims=[1])
```

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 sample of `StochasticTensor`, which can be mixed with Vars and fed into any Torch operations.

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. 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-PyTorch, the way to achieve this is by setting `reduce_mean_dims` and `reduce_sum_dims`.

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)
# In __init__
self.fc1 = nn.Linear(z_dim, 500)
self.act1 = nn.Relu()
self.fc2 = nn.Linear(500, 500)
self.act2 = nn.Relu()
self.fc2_ = nn.Linear(500, x_dim)

# In forward
x_logits = self.fc2_(self.act2(self.fc2(self.act1(self.fc1(z)))))
```

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))
x_probs = nn.Sigmoid()(x_logits)
self.sn('Bernoulli',
        name='x',
        probs=x_probs,
        reduce_mean_dims=[0],
        reduce_sum_dims=[1])
```

Note: The *Bernoulli* distribution accepts log-odds of probabilities instead of probabilities. This is designed for numeric stability reasons.

Putting together, the code for constructing a VAE is:

```
class Generator(BayesianNet):
    def __init__(self, x_dim, z_dim, batch_size):
        super().__init__()
        self.x_dim = x_dim
        self.z_dim = z_dim
        self.batch_size = batch_size

        self.fc1 = nn.Linear(z_dim, 500)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(500, 500)
        self.act2 = nn.ReLU()

        self.fc2_ = nn.Linear(500, x_dim)
        self.act2_ = nn.Sigmoid()

    def forward(self, observed):
        self.observe(observed)
        mean = torch.zeros([self.batch_size, self.z_dim])
        std = torch.ones([self.batch_size, self.z_dim])

        z = self.sn('Normal',
                    name='z',
                    mean=mean,
                    std=std,
                    reparameterize=False,
                    reduce_mean_dims=[0],
                    reduce_sum_dims=[1])
        x_probs = self.act2_(self.fc2_(self.act2(self.fc2(self.act1(self.fc1(z))))))
        self.cache['x_mean'] = x_probs
        sample_x = self.sn('Bernoulli',
                        name='x',
                        probs=x_probs,
                        reduce_mean_dims=[0],
                        reduce_sum_dims=[1])
        return self

generator = Generator(x_dim, z_dim, batch_size)
```

1.2.2 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 Torch tensor 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:

```python
class Variational(BayesianNet):
    def __init__(self, x_dim, z_dim, batch_size):
        super().__init__()
        self.x_dim = x_dim
        self.z_dim = z_dim
        self.batch_size = batch_size

        self.fc1 = nn.Linear(x_dim, 500)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(500, 500)
        self.act2 = nn.ReLU()
```

(continues on next page)

(continued from previous page)

```
        self.fc3 = nn.Linear(500, z_dim)
        self.fc4 = nn.Linear(500, z_dim)

        self.dist = None

    def forward(self, observed):
        self.observe(observed)
        x = self.observed['x']
        z_logits = self.act2(self.fc2(self.act1(self.fc1(x))))

        z_mean = self.fc3(z_logits)
        z_std = torch.exp(self.fc4(z_logits))

        z = self.sn('Normal',
                    name='z',
                    mean=z_mean,
                    std=z_std,
                    reparameterize=True,
                    reduce_mean_dims=[0],
                    reduce_sum_dims=[1])
        return self

variational = Variational(x_dim, z_dim, batch_size)
```

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

```
model = zs.variational.ELBO(generator, variational)
```

The returned `lower_bound` is an `EvidenceLowerBoundObjective` instance, which is a derivatication of Torch's *Module*. However, optimizing the 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
lower_bound = model({'x': batch_x})
```

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. Torch 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 Torch's Adam optimizer to do the learning:

```python
optimizer = torch.optim.Adam(model.parameters(), lr)

# During each iter
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

1.2.3 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. We put the Var `x_mean` in the *cache* of `Generator` to keep track of it.

```python
class Generator(BayesianNet):
    def __init__(self, x_dim, z_dim, batch_size):
        ...

    def forward(self, observed):
        ...
        x_probs = self.act2_(self.fc2_(self.act2(self.fc2(self.act1(self.fc1(z))))))
        self.cache['x_mean'] = x_probs
        self.sn('Bernoulli',
                name='x',
                probs=x_probs,
                reduce_mean_dims=[0],
                reduce_sum_dims=[1])
        ...
```

so that we can easily access it from a `BayesianNet` instance. For random generations, no observation about the model is made, so we pass an empty observation to the model and get the generated sample by the `cache['x_mean']` of `Generator`:

```
cache = generator({}).cache
sample_gen = cache['x_mean']
```

1.2.4 Run gradient descent

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

```
for epoch in range(epoch_size):
    for step in range(num_batches):
        x = torch.as_tensor(x_train[step * batch_size:min((step + 1) * batch_size,␣
↪len_)])
        x = torch.reshape(x, [-1, x_dim])
        if x.shape[0] != batch_size:
            break
        loss = model({'x': x})
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (step + 1) % 100 == 0:
            print("Epoch[{}/{}], Step [{}/{}], Loss: {:.4f}".format(epoch + 1, epoch_
↪size, step + 1, num_batches,loss))

batch_x = x_test[0:64]

cache = generator({}).cache
sample_gen = cache['x_mean'].numpy()
```

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

References

1.3 Bayesian Neural Networks

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

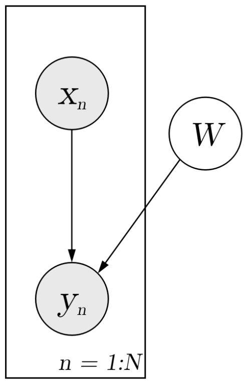
Recent years have seen neural networks' powerful abilities in fitting complex transformations, with successful applications on speech recognition, image classification, and machine translation, etc. However, typical training of neural networks requires lots of labeled data to control the risk of overfitting. 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):

```
class Net(BayesianNet):
    def __init__(self, layer_sizes, n_particles):
        super().__init__()
```

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 = x.repeat([self.n_particles, *len(x.shape) * [1]])
for i, (n_in, n_out) in enumerate(zip(self.layer_sizes[:-1], self.layer_sizes[1:])):
    w = self.sn('Normal',
                name='w' + str(i),
                mean=torch.zeros([n_out, n_in + 1]),
                std=torch.ones([n_out, n_in + 1]),
                group_ndims=2,
                n_samples=self.n_particles,
                reduce_mean_dims=[0])
```

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(self.layer_sizes[:-1], self.layer_sizes[1:])):
    w = self.sn('Normal',
                name='w' + str(i),
                mean=torch.zeros([n_out, n_in + 1]),
                std=torch.ones([n_out, n_in + 1]),
                group_ndims=2,
                n_samples=self.n_particles,
                reduce_mean_dims=[0])
    w = torch.unsqueeze(w, 1)
    w = w.repeat([1, batch_size, 1, 1])
    h = torch.cat((h, torch.ones([*h.shape[:-1], 1])), -1)
    h = torch.unsqueeze(h, -1)
    p = torch.sqrt(torch.as_tensor(h.shape[2], dtype=torch.float32))
    h = torch.matmul(w, h) / p
    h = torch.squeeze(h, -1)
    if i < len(self.layer_sizes) - 2:
        h = torch.nn.ReLU()(h)
```

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

```
y = self.observed['y']
y_pred = torch.mean(y_mean, 0)
self.cache['rmse'] = torch.sqrt(torch.mean((y - y_pred) ** 2))

self.sn('Normal',
        name='y',
        mean=y_mean,
```

(continues on next page)

(continued from previous page)

```
        logstd=self.y_logstd,
        reparameterize=True,
        reduce_mean_dims=[0, 1],
        multiplier=456)  # training data size
```

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

```
class Net(BayesianNet):
    def __init__(self, layer_sizes, n_particles):
        super().__init__()
        self.layer_sizes = layer_sizes
        self.n_particles = n_particles
        self.y_logstd = torch.nn.parameter.Parameter(torch.nn.init.constant_(torch.
↪empty([1], dtype = torch.float32), 0.0), requires_grad=True)

    def forward(self, observed):
        self.observe(observed)
        x = self.observed['x']
        h = x.repeat([self.n_particles, *len(x.shape) * [1]])

        batch_size = x.shape[0]

        for i, (n_in, n_out) in enumerate(zip(self.layer_sizes[:-1], self.layer_
↪sizes[1:])):
            w = self.sn('Normal',
                        name='w' + str(i),
                        mean=torch.zeros([n_out, n_in + 1]),
                        std=torch.ones([n_out, n_in + 1]),
                        group_ndims=2,
                        n_samples=self.n_particles,
                        reduce_mean_dims=[0])
            w = torch.unsqueeze(w, 1)
            w = w.repeat([1, batch_size, 1, 1])
            h = torch.cat((h, torch.ones([*h.shape[:-1], 1])), -1)
            h = torch.unsqueeze(h, -1)
            p = torch.sqrt(torch.as_tensor(h.shape[2], dtype=torch.float32))
            h = torch.matmul(w, h) / p
            h = torch.squeeze(h, -1)
            if i < len(self.layer_sizes) - 2:
                h = torch.nn.ReLU()(h)

        y_mean = torch.squeeze(h, 2)

        y = self.observed['y']
        y_pred = torch.mean(y_mean, 0)
        self.cache['rmse'] = torch.sqrt(torch.mean((y - y_pred) ** 2))

        self.sn('Normal',
                name='y',
                mean=y_mean,
                logstd=self.y_logstd,
                reparameterize=True,
                reduce_mean_dims=[0, 1],
                multiplier=456)  # training data size
        return self
```

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:

```
class Variational(BayesianNet):
    def __init__(self, layer_sizes, n_particles):
        super().__init__()
        self.layer_sizes = layer_sizes
        self.n_particles = n_particles

        self.w_means = []
        self.w_logstds = []

        for i, (n_in, n_out) in enumerate(zip(self.layer_sizes[:-1], self.layer_
↪sizes[1:])):
            w_mean = torch.nn.init.constant_(torch.empty([n_out, n_in + 1], dtype =␣
↪torch.float32), 0)
            _name = 'w_mean_' + str(i)
            self.__dict__[_name] = w_mean
            w_logstd = torch.nn.init.constant_(torch.empty([n_out, n_in + 1], dtype =␣
↪torch.float32), 0)
            _name = 'w_logstd_' + str(i)
            self.__dict__[_name] = w_logstd
            w_mean = torch.nn.parameter.Parameter(w_mean, requires_grad=True)
            w_logstd = torch.nn.parameter.Parameter(w_logstd, requires_grad=True)
            self.w_means.append(w_mean)
            self.w_logstds.append(w_logstd)

        self.w_means = torch.nn.ParameterList(self.w_means)
        self.w_logstds = torch.nn.ParameterList(self.w_logstds)

    def forward(self, observed):
        self.observe(observed)
        for i, (n_in, n_out) in enumerate(zip(self.layer_sizes[:-1], self.layer_
↪sizes[1:])):
            self.sn('Normal',
                    name='w' + str(i),
                    mean=self.w_means[i],
                    logstd=self.w_logstds[i],
                    group_ndims=2,
                    n_samples=self.n_particles,
                    reparameterize=True,
                    reduce_mean_dims=[0])
        return self
```

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:

```
net = Net(layer_sizes, n_particles)
variational = Variational(layer_sizes, n_particles)

model = zs.variational.ELBO(net, variational)
```

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.

```
# prediction: rmse & log likelihood
# In Net
y_mean = torch.squeeze(h, 2)

y = self.observed['y']
y_pred = torch.mean(y_mean, 0)
self.cache['rmse'] = torch.sqrt(torch.mean((y - y_pred) ** 2))
# During training
lower_bound = model({'x': x, 'y': y})
```

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
rese = net.cache['rmse']
log_ll = model({'x': x, 'y': y})
```

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:

```
for epoch in range(epoch_size):
    perm = np.random.permutation(x_train.shape[0])
    x_train = x_train[perm, :]
    y_train = y_train[perm]

    for step in range(num_batches):
        x = torch.as_tensor(x_train[step * batch_size:(step + 1) * batch_size])
        y = torch.as_tensor(y_train[step * batch_size:(step + 1) * batch_size])
        lbs = model({'x': x, 'y': y})
        optimizer.zero_grad()
        lbs.backward()
        optimizer.step()

        if (step + 1) % num_batches == 0:
            rmse = net.cache['rmse'].clone().detach().numpy()
            print("Epoch[{}/{}], Step [{}/{}], Lower bound: {:.4f}, RMSE: {:.4f}".
↪format(epoch + 1, epoch_size,
                                                                              ␣
↪  step + 1,
                                                                              ␣
↪  num_batches,
                                                                              ␣
↪  float(lbs.clone().detach().numpy()),
                                                                              ␣
↪  float(rmse) * std_y_train))

    # eval
    if epoch % test_freq == 0:
        x_t = torch.as_tensor(x_test)
        y_t = torch.as_tensor(y_test)
        lbs = model({'x': x_t, 'y': y_t})
        rmse = net.cache['rmse'].clone().detach().numpy()
        print('>> TEST')
        print('>> Test Lower bound: {:.4f}, RMSE: {:.4f}'.format(float(lbs.clone().
↪detach().numpy()), float(rmse) * std_y_train))
```

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.

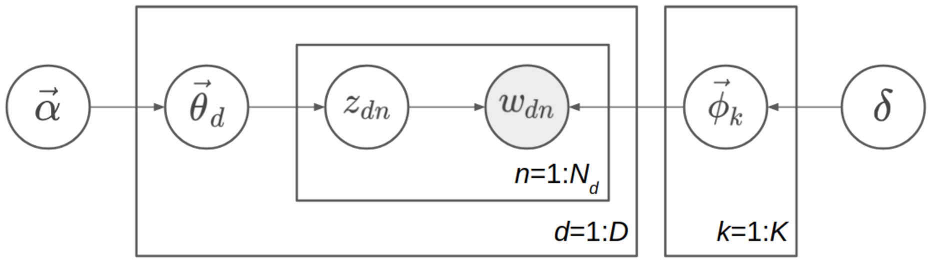
For example, we can model documents as arising from multiple topics, where a topic is defined to be a distribution over a fixed vocabulary of terms. The most famous model is **Latent Dirichlet Allocation** (LDA) [LNTMBNJ03]. First we describe the notations. Following notations differ from the standard 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,

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

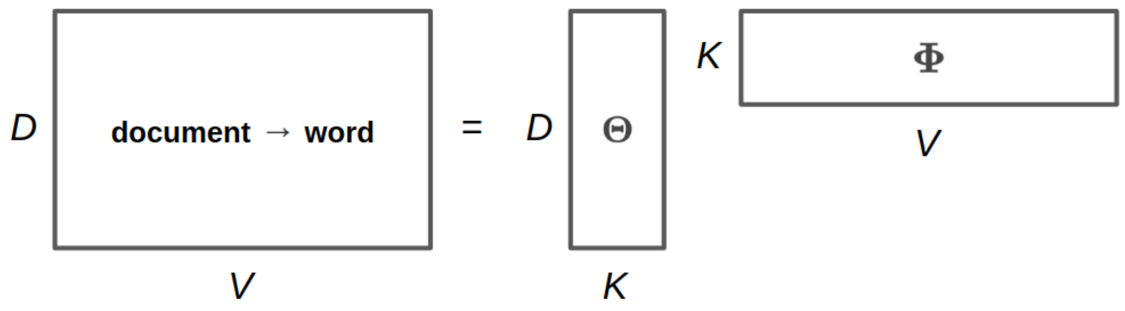
which means when sampling the words in the dth document, the word distribution is the weighted average of all topics, and the weights are the topic proportion of the document.

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, $\vec{x}_d = \sum_{n=1}^{N_d} \text{one\_hot}(w_{dn})$. 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:

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

Here, CE means cross entropy, which is defined for matrices as $\text{CE}(\mathbf{A}, \mathbf{B}) = -\sum_{i,j} A_{ij} \log B_{ij}$. 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 $\log p(w_{1:D,1:N}|\mathbf{\Theta}, \mathbf{\Phi}) = -\text{CE}(\mathbf{X}, \mathbf{\Theta}\mathbf{\Phi})$.

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



As minimizing the cross entropy encourages $\mathbf{X}$ and $\mathbf{\Theta}\mathbf{\Phi}$ 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.

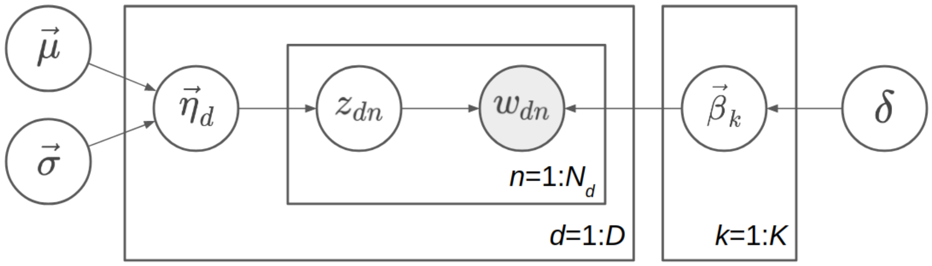
After integrating $\mathbf{z}$, only $\mathbf{\Theta}$ and $\mathbf{\Phi}$ are left, and there is no conjugacy any more. Even if we apply the "conditional computing" trick like Gibbs sampling, no closed-form updating process can be obtained. However, we can adopt the gradient-based method such as HMC 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 $\mathrm{diag}(\vec{\sigma}^2)$, 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 $\mathbf{\Theta} = \mathrm{softmax}(\mathbf{H})$, and $\mathbf{\Phi} = \mathrm{softmax}(\mathbf{B})$. Recall our notation that $\mathbf{X} = (\vec{x}_1, \vec{x}_2, ..., \vec{x}_D)^T$ where $\vec{x}_d = \sum_{n=1}^{N_d} \mathrm{one\_hot}(w_{dn})$. After integrating $\{z_{dn}\}$, the last two lines of the generating process:

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

become $\log p(\mathbf{X}|\mathbf{\Theta}, \mathbf{\Phi}) = -\mathrm{CE}(\mathbf{X}, \mathbf{\Theta}\mathbf{\Phi})$. 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 $p(\mathbf{X}|\mathbf{H}, \mathbf{B}) = -\mathrm{CE}(\mathbf{X}, \mathrm{softmax}(\mathbf{H})\mathrm{softmax}(\mathbf{B}))$.

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)
    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 $\mathbf{\Theta}\mathbf{\Phi}$, `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, :]
```

(continues on next page)

(continued from previous page)

```
        # 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 Distribution

class Distribution(*dtype*, *is_continuous*, *is_reparameterized*, *use_path_derivative=False*, *group_ndims=0*, *device=device(type='cpu')*, ***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.

dtype must be among *torch.int16*, *torch.int32*, *torch.int64*, *torch.float16*, *torch.float32* and *torch.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.
- **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.

property 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`.

property device
The device this distribution lies at.

Returns torch.device

property dtype
The sample type of the distribution.

property 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 Var. 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 Var of shape `(... + )batch_shape[:-group_ndims]`.

prob(*given*)

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 Var 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 Var of samples.

1.5.2 Normal

class Normal(*mean=0.0*, *std=None*, *logstd=None*, *dtype=None*, *is_continuous=True*, *is_reparameterized=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

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

Parameters

- **`mean`** – A *float* Var. The mean of the Normal distribution. Should be broadcastable to match *std* or *logstd*.
- **`std`** – A *float* Var. The standard deviation of the Normal distribution. Should be positive and broadcastable to match *mean*.
- **`logstd`** – A *float* Var. The log standard deviation of the Normal distribution. Should be broadcastable to match *mean*.
- **`group_ndims`** – A 0-D *int32* Var 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"

property logstd
The log standard deviation of the Normal distribution.

property mean
The mean of the Normal distribution.

property std
The standard deviation of the Normal distribution.

1.5.3 Bernoulli

class Bernoulli(*logits=None*, *probs=None*, *dtype=None*, *is_continuous=False*, *group_ndims=0*, *device=device(type='cpu')*, ***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.

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

- **`probs`** – A 'float' Tensor. The p param of bernoulli distribution
- **`dtype`** – The value type of samples from the distribution. Can be int (*torch.int16*, *torch.int32*, *torch.int64*) or float (*torch.float16*, *torch.float32*, *torch.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.

`property logits`

`property probs`

1.5.4 Beta

`class Beta`(*alpha*, *beta*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

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

Parameters

- **`alpha`** – A 'float' Var. One of the two shape parameters of the Beta distribution.
- **`beta`** – A 'float' Var. One of the two shape parameters of the Beta distribution.

`property alpha`

One of the two shape parameters of the Beta distribution.

`property beta`

One of the two shape parameters of the Beta distribution.

1.5.5 Exponential

`class Exponential`(*rate*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

The class of univariate Exponential distribution See `Distribution` for details.

Parameters **`rate`** – A 'float' Var. Rate parameter of the Exponential distribution.

`property rate`

Shape parameter of the Exponential distribution.

1.5.6 Gamma

`class Gamma`(*alpha*, *beta*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

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

Parameters

- **`alpha`** – A 'float' Var. Shape parameter of the Gamma distribution.
- **`beta`** – A 'float' Var. Rate parameter of the Gamma distribution.

`property alpha`

Shape parameter of the Gamma distribution.

property beta
Rate parameter of the Gamma distribution.

1.5.7 Laplace

class Laplace(*loc*, *scale*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

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

Parameters

- **loc** – A 'float' Var. Mean of the Laplace distribution.
- **scale** – A 'float' Var. Scale of the Laplace distribution.

property loc
Mean of the Laplace distribution.

property scale
Scale of the Laplace distribution.

1.5.8 Logistic

class Logistic(*loc*, *scale*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

The class of univariate Logistic distribution, always using the reparametrization trick from (Kingma, 2013). See *Distribution* for details.

Parameters

- **loc** – A 'float' Var. The location term acting on standard Logistic distribution.
- **scale** – A 'float' Var. The scale term acting on standard Logistic distribution.

property loc

property scale

1.5.9 Poisson

class Poisson(*rate*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)
Bases: *zhusuan.distributions.base.Distribution*

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

Parameters rate – A 'float' Var. Rate parameter of the Poisson distribution.Must be positive.

property rate
Shape parameter of the Poisson distribution.

1.5.10 StudentT

class StudentT(*df*, *loc=0.0*, *scale=1.0*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

The class of univariate StudentT distribution See `Distribution` for details.

Parameters

- **df** – A 'float' Var. Degrees of freedom.
- **loc** – A 'float' Var. Mean of the StudentT distribution.
- **scale** – A 'float' Var. Scale of the StudentT distribution.

property df

Degrees of freedom.

property loc

Mean of the Laplace distribution.

property scale

Scale of the Laplace distribution.

1.5.11 Uniform

class Uniform(*low*, *high*, *dtype=None*, *is_continuous=True*, *is_reparameterized=True*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

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

Parameters

- **low** – A 'float' Var. Lower range (inclusive).
- **high** – A 'float' Var. Upper range (exclusive).

property high

Upper range (exclusive) of the Uniform distribution.

property low

Lower range (inclusive) of the Uniform distribution.

1.5.12 FlowDistribution

class FlowDistribution(*latents*, *transformation*, *flow_kwargs=None*, *dtype=torch.float32*, *group_ndims=0*, *device=device(type='cpu')*, ***kwargs*)

Bases: `zhusuan.distributions.base.Distribution`

A class for sample from Flow networks by provide the latent distribution and the flow network, when calling *sample* method, it returns the sample from flow network, when calling *log_prob* method it return the loss item of flow network.

Parameters

- **latents** – An instance of *Distribution* class, as the prioror the latent variableof FlowDistrubution
- **transformation** – A RevNet instance, the Flow net work built by user

- **`flow_kwargs`** – additional info to be recode
- **`dtype`** – data type
- **`device`** – device of Distribution

1.5.13 utils

`assert_same_dtype_in`(*tensors_with_name*, *dtypes=None*)
Whether all types of tensors in *tensors_with_name* are the same and in the allowed *dtypes*.

Parameters

- **`tensors_with_name`** – A list of (tensor, tensor_name).
- **`dtypes`** – A list of allowed dtypes. If *None*, then all dtypes are allowed.

Returns The dtype of *tensors*.

`assert_same_float_dtype`(*tensors_with_name*)
Whether all tensors in *tensors_with_name* have the same floating type.

Parameters **`tensors_with_name`** – A list of (tensor, tensor_name).

Returns The type of *tensors*.

`assert_same_log_float_dtype`(*tensors_with_name*)
Whether all tensors in *tensors_with_name* have the same floating type, which also support log/exp operations.

Parameters **`tensors_with_name`** – A list of (tensor, tensor_name).

Returns The type of *tensors*.

`check_broadcast`(*mean*, *std*)
check whether mean and std broadcast match

1.6 zhusuan.framework

1.6.1 BayesianNet

`class BayesianNet`(*observed=None*, *device=device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

`bernoulli`(*name*, *logits=None*, *probs=None*, *dtype=None*, *is_continuous=False*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

`beta`(*name*, *alpha*, *beta*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

`property cache`
The dictionary of all named deterministic nodes in this `BayesianNet`.

Returns A dict.

`property device`
The device this module lies at.

Returns torch.device

`exponential`(*name*, *rate*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

gamma(*name*, *alpha*, *beta*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

laplace(*name*, *loc*, *scale*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

log_joint(*use_cache=False*)

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 Var.

logistic(*name*, *loc*, *scale*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

property nodes

The dictionary of all named stochastic nodes in this `BayesianNet`.

Returns A dict.

normal(*name*, *mean=0.0*, *std=None*, *logstd=None*, *dtype=None*, *is_continuous=True*, *is_reparameterized=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

observe(*observed*)

Assign the nodes and values to be observed in this `BayesianNet`.

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

property observed

The dictionary of all observed nodes in this `BayesianNet`.

Returns A dict.

poisson(*name*, *rate*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

sn(*dist*, *name*, *n_samples=None*, ***kwargs*)

Short cut for method `stochastic_node()`

snode(**args*, ***kwargs*)

Short cut for method `stochastic_node()`

stochastic_node(*distribution*, *name*, *n_samples=None*, ***kwargs*)

Add a stochastic node in this `BayesianNet` that follows the distribution assigned by the `name` parameter.

Parameters

- **distribution** – The distribution which the node follows.
- **name** – The unique name of the node.
- **n_samples** – number of samples per sample process
- **kwargs** – Parameters of the distribution which the node builds with.

Returns A instance(sample) of the node.

studentT(*name*, *df*, *loc=0.0*, *scale=1.0*, *dtype=None*, *is_continuous=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

to(*device*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved *`device`*, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```
>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

training: bool

uniform(*name*, *low*, *high*, *dtype=None*, *is_continuous=True*, *is_reparameterized=True*, *group_ndims=0*, *n_samples=None*, ***kwargs*)

1.6.2 StochasticTensor

class StochasticTensor(*bn*, *name: str*, *dist:* zhusuan.distributions.base.Distribution, *observation=None*, *n_samples=None*, ***kwargs*)

Bases: `object`

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:

```
class Net(BayesianNet):
    def __init__(self):
        self.stochastic_node('Normal', name='x', mean=0., std=1.)
```

will build a stochastic node in `Net` with the *Normal* distribution. The returned `x` will be a instance of *StochasticTensor*.

StochasticTensor instances are Vars, which means that they can be passed into any Jittor operations. This makes it easy to build Bayesian networks by mixing stochastic nodes and Jittor 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 Var, which matches the shape of *dist*. If specified, then the *StochasticTensor* is observed and the *tensor* property will return the *observation*.
- **n_samples** – A 0-D integer. Number of samples generated by this *StochasticTensor*.

property bn

The *BayesianNet* where the *StochasticTensor* lives.

Returns A *BayesianNet* instance.

property dist
The distribution followed by the *StochasticTensor*.

Returns A *Distribution* instance.

property 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(*sample=None*)

property name
The name of the *StochasticTensor*.

Returns A string.

sample(*force=False*)
The value of this *StochasticTensor*. If it is observed, then the observation is returned, otherwise samples are returned. :param force: force to sample, disregard the observed value, default as False :return: A Var.

property shape
Return the static shape of this *StochasticTensor*.

Returns A `torch.Size` instance.

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

Returns A Var.

1.7 zhusuan.variational

1.7.1 ELBO

class ELBO(*generator*, *variational*, *estimator='sgvb'*, *transform=None*, *transform_var=[]*, *auxillary_var=[]*)
Bases: `torch.nn.modules.module.Module`

The class that represents the evidence lower bound (ELBO) objective for variational inference. It can be constructed like a Jittor's *Module* by passing 2 *BayesianNet* instances. For example, the generator network and the variational inference network in VAE. The model can calculate the ELBO's value with observations passed.

See also:

For more details and examples, please refer to *Variational Autoencoders* and *Bayesian Neural Networks*

Parameters

- **generator** – A :class'~zhusuan.framework.BayesianNet` 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.
- **variational** – A *BayesianNet* instance that defines the variational family.
- **estimator** – gradient estimate method, including `sgvb` and `reinforce`
- **transform** – A `RevNet` instance that transform Specified variables,

returns the transformed variable and the log_det_J i.e log-determinant of transition Jacobian matrix :param transform_var: a list of names of variable to be transformed, all tensor that correspond to these names will be placed into tuple by order and feed to the transform network :param auxillary_var: auxillary variable name list that need to be passed to transform network

forward(*observed*, *reduce_mean=True*, ***kwargs*)
observe nodes, transform latent variables, return evidence lower bound :return: evidence lower bound

log_joint(*nodes*)
The default log joint probability function. It works by summing over all the conditional log probabilities of stochastic nodes evaluated at their current values (samples or observations).

Returns A Var.

reinforce(*logpxz*, *logqz*, *reduce_mean=True*, *baseline=None*, *variance_reduction=True*, *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

- **logpxz** – log joint of generator
- **logqz** – log joint of variational
- **reduce_mean** – whether reduce to a scalar by mean operation
- **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.
- **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.
- **decay** – Float. The moving average decay for variance normalization.

Returns A Tensor. The surrogate cost for optimizers to minimize.

sgvb(*logpxz*, *logqz*, *reduce_mean=True*, *log_det=None*)
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 optimizers to minimize.

`training: bool`

1.7.2 ImportanceWeightedObjective

`class ImportanceWeightedObjective`(*generator*, *variational*, *axis=None*, *estimator='sgvb'*)

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

The class that represents the importance weighted objective for variational inference (Burda, 2015)

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".

Parameters

- **`generator`** – generator part of importance weighted objective
- **`variational`** – variational part of importance weighted objective
- **`axis`** – The sample dimension(s) to reduce when computing the outer expectation in the objective.
- **`estimator`** – the estimator, a str in either 'sgvb' or 'vimco'

`forward`(*observed*, *reduce_mean=True*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`log_joint`(*nodes*)

`sgvb`(*logpxz*, *logqz*, *reduce_mean=True*)

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 optimizers to minimize.

training: bool

vimco (*logpxz*, *logqz*, *reduce_mean=True*)

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 optimizers to minimize.

1.8 zhusuan.mcmc

1.8.1 SGMCMC

class SGMCMC

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

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, SGHMC.

The typical code for SGMCMC inference is like:

```
sgmcmc = zs.mcmc.SGLD(learning_rate=lr)
net = BayesianNet()
for epoch in range(epoch_size):
    for step in range(num_batches):
        w_samples = model.sample(net, {'x': x, 'y': y})

        for i, (k, w) in enumerate(w_samples.items()):
            # Utilize stochastic gradients by samples and update parameters.
            ...
```

forward (*bn*, *observed*, *resample=False*, *step=1*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

initialize()

sample(*bn*, *observed*, *resample=False*, *step=1*)
Running one sgmcmc iteration.

Parameters

- **bn** – A instance of *BayesianNet*.
- **observed** – A dictionary of `(string, Tensor)` pairs. Mapping from names of observed *StochasticTensor* s to their values.
- **resample** – Flag indicates if the sampler need get the var list of the *BayesianNet* instance, usually set to True on first sgmcmc iteration.

Returns A list of Var, samples generated by sgmcmc iteration.

training: bool

1.8.2 SGLD

class SGLD(*learning_rate*)
Bases: *zhusuan.mcmc.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.

- **var_list** - The updated values of latent variables.

Parameters **learning_rate** – A 0-D *float32* Var.

property device
The device this module lies at.

Returns torch.device

to(*device*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved *device*, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

`training: bool`

1.8.3 PSGLD

class PSGLD(*learning_rate*, *decay=0.9*, *epsilon=0.001*)

Bases: `zhusuan.mcmc.SGLD.SGLD`

PSGLD with RMSprop preconditioner, "Preconditioned stochastic gradient Langevin dynamics for deep neural networks"

training: bool

1.8.4 SGHMC

class SGHMC(*learning_rate*, *friction=0.25*, *variance_estimate=0.0*, *n_iter_resample_v=20*, *second_order=True*)

Bases: `zhusuan.mcmc.SGMCMC.SGMCMC`

training: bool

1.9 zhusuan.invertible

1.9.1 RevNet

class RevNet

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

An abc of reversible networkevery subclass should implement both `_forward` and `_inverse` abstract method. return value of `_forward` and `_inverse` is like `(y, log_det_J)`, in which `y` is the transformed tensor and *log_det_J`* is log-determinant of Jacobian.

forward(**inputs*, *reverse=False*, ***kwargs*)

when using `model.forward(x, reverse=False)` process going with `_forward(x)`, when using `model.forward(x, reverse=True)` process going with `_inverse(x)`.

training: bool

1.9.2 RevSequential

class RevSequential(*layers*)

Bases: `zhusuan.invertible.base.RevNet`

the RevSequential provide a invertible transform which contain a list of instance of RevNet. when forward passing with `reverse=False`, the input `x` goes through every RevNet in the list also with `reverse=False` *from begin to end* , when forward passing with `reverse=True`, input `x` goes through every in the list also with `reverse=True` *from end to begin.*

Parameters layers – a list of RevNet instance.

training: bool

1.9.3 Coupling

class Coupling(*in_out_dim*, *mid_dim*, *hidden*, *mask_config*)

Bases: `zhusuan.invertible.base.RevNet`

coupling layer class

Parameters

- **in_out_dim** – input/output dimensions.
- **mid_dim** – number of units in a hidden layer.
- **hidden** – number of hidden layers.
- **mask_config** – 1 if transform odd units, 0 if transform even units.

training: bool

1.9.4 MaskCoupling

class MaskCoupling(*in_out_dim=- 1*, *mid_dim=- 1*, *hidden=- 1*, *mask=None*, *inner_nn=None*)

Bases: `zhusuan.invertible.base.RevNet`

A coupling layer Identify if keep same or do transform by mask

Parameters

- **in_out_dim** – input/output dimensions.
- **mid_dim** – number of units in a hidden layer
- **hidden** – number of hidden layers
- **mask** – mask given by the user, often generated by function:*~zhusuan.invertible.coupling.get_coupling_mask*

training: bool

1.9.5 Scaling

class Scaling(*dim*)

Bases: `zhusuan.invertible.base.RevNet`

Initialize a (log-)scaling layer. when Forward pass, given class:`x` as input tensor, it returns (`y`, `log_det_J`) where `y` is transformed tensor by `y=x*exp(log_scale)` and `log_det_J` is log-determinant of Jacobian.

Parameters **dim** – input/output dimensions.

training: bool

1.9.6 MaskedLinear

class MaskedLinear(*input_size*, *n_outputs*, *mask*, *cond_label_size=None*)

Bases: `torch.nn.modules.linear.Linear`

MADE building block layer

forward(*x*, *cond_y=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

in_features: int

out_features: int

weight: torch.Tensor

1.9.7 MADE

class MADE(*input_size*, *hidden_size*, *n_hidden*, *cond_label_size=None*, *input_order='sequential'*, *input_degrees=None*, *activation='relu'*)

Bases: `zhusuan.invertible.base.RevNet`

MADE class

Parameters

- **input_size** – a scalar; dim of inputs
- **hidden_size** – a scalar; dim of hidden layers
- **n_hidden** – a scalar; number of hidden layers
- **activation** – a str; activation function to use
- **input_order** – a str or tensor; variable order for creating the autoregressive masks (sequentiallrandom) or the order flipped from the previous layer in a stack of mades
- **conditional** – a bool; whether model is conditional

static create_mask(*input_size*, *hidden_size*, *n_hidden*, *input_order='sequential'*, *input_degrees=None*)

Mask generator for MADE & MAF (see MADE paper sec 4:https://arxiv.org/abs/1502.03509)

Parameters

- **input_size** – dim of inputs
- **hidden_size** – dim of hidden layers
- **n_hidden** – number of hidden layers
- **input_order** – variable order for creating the autoregressive masks (sequentiallrandom)
- **input_degrees** – degrees provide by user

Returns: List of masks

```
training: bool
```

1.10 Contributing

We always welcome contributions to help make ZhuSuan-Torch 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).

ZhuSuan-PyTorch

- Zhengyi Wang (thuwzy)
- ChenDong Xiang (Xiang-cd)

ZhuSuan on other platforms

- Guande He (rubbybbs)
- Yong Ren (McGrady00H)
- Jianfei Chen (cjf00000)
- Jiaxin Shi (thjashin)
- Wen jie Line (awakebacker)
- Cheng Lu (LuChengTHU)

CHAPTER

TWO

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

INDEX