Yifan Bai

20153885

Problem 1

Formula for part 1 to 4

1. Log likelihood for Bernoulli distribution

For ONE sample, the log likelihood for this sample given the target is:

$$L(\mu) = \sum_{i=1}^{n} (\log \mu_i * target_i + \log 1 - \mu_i * (1 - target_i))$$

where i = 1 to n is the index for each pixel. And we will sum this formula by the dimension of batch, which is 1.

2. Log likelihood for Normal distribution

result = np.log(2*np.pi) + logvar + (z - mu).pow(2) / logvar.exp() result = (-1. / 2.) * torch.sum(result, dim=1) # sum by batch Similarly, per sample, before we take on the log:

$$L(\mu) = \prod_i \left(\frac{1}{\sqrt{2\pi\sigma}} \exp\left(\frac{(\mu - target)^2}{2\sigma^2}\right) \right)$$

Now if we take the log:

$$Log_L(\mu) = \sum_i -\frac{1}{2} \left(\log(2\pi\sigma) + \frac{(\mu - target)^2}{2\sigma^2} \right)$$

3. Log mean exp

Just follow the instruction. Build array A by taking the max on y, the rest is trivial

4. KL divergence

The formula for KL divergence, when p, q are both assumed Gaussian, has a neat form:

$$D_{KL}(p(x)||q(x)) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}$$

Now given $p(x) -> \frac{1}{\sqrt{2\pi\sigma_p}} \exp\left((x - mu_p)^2 / 2\sigma_p^2\right)$ and $q(x) -> \frac{1}{\sqrt{2\pi\sigma_q}} \exp\left((x - mu_q)^2 / 2\sigma_q^2\right)$

$$p(x)/q(x) = \frac{\sqrt{\sigma_q}}{\sqrt{\sigma_p}} \exp\left[(x - mu_p)^2 / 2\sigma_p^2) - (x - mu_p)^2 / 2\sigma_p^2)\right]$$

$$\log p(x)/q(x) = \frac{1}{2} * (logvar_q - logvar_p) + (mu_p - mu_q)^2 - (logvar_p^2 - logvar_q^2)$$

And we simply need to sum this up following axis 1 (batch)

Training result

6.

I have trained the model described in this notebook, with the log loss I implemented in the `vae.ipynb` whose PDF is attached at the end of Appendix. The result is shown above. And I have achieved an average per-instance ELBO of **101.28** at the end of 20-th epoch.

7.

The log-likelihood in test samples are **95.54**

Problem 2

1.

According to Equation 7 in the paper (Nowozin et al. link: https://arxiv.org/pdf/1606.00709.pdf (https://arxiv.org/pdf/1606.00709.pdf)), the objective is given as:

$$F(\theta, \omega) = E_{x \sim P}[g_f(V_\omega(x))] + E_{x \sim Q_\theta}[-f^*(V_\omega(x))]$$

Where g_f is the ouput activation function and f^* is the Conjugate function for Squared Hellinger:

$$g_f(v) = 1 - \exp(-v)$$
$$f^*(t) = \frac{t}{1-t}$$

3.

The objective function of the Wasserstein distance is given as the Equation (5) in the paper by Petzka et al. (https://arxiv.org/pdf/1709.08894.pdf (https://arxiv.org/pdf/1709.08894.pdf)):

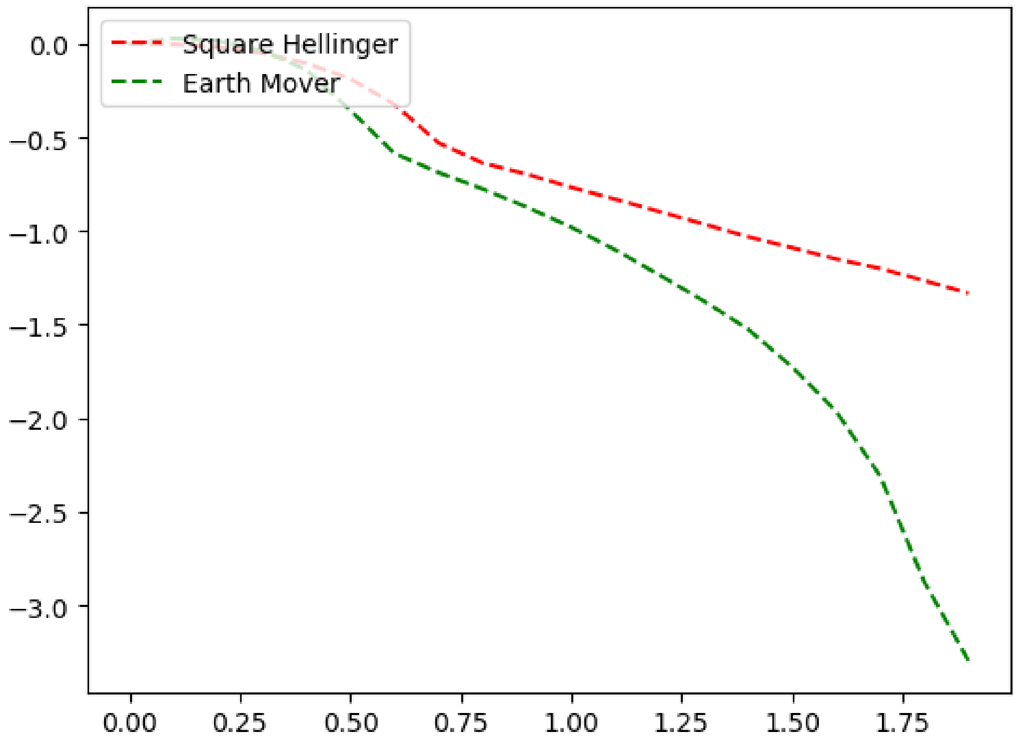
$$\min_{v} \max_{f \in Lip_1} E_{x \sim \mu}[f(x)] - E_{y \sim v}[f(y)]$$

Where the Lipschitz Penalty is defined as (equation 8 in the paper):

$$E_{y \sim v}[f(y)] - E_{x \sim \mu}[f(x)] + \lambda E_{\hat{x} \sim \tau}[(max\{0, ||\nabla f(\hat{x})|| - 1\})^2]$$

5.

In this part we trained a toy model to uncover the variable theta. For each of distance of our choosing, we trained the same model for 30 epochs.



The distance vs. Theta is shown above, for Squared Hellinger and Earth Mover.

Problem 3

Samples

I have completed the training and sampling process given the code templates. Below are generated images from a generator that is trained for 150 epochs:

Blurriness

From the sampled pictures, we can see that the clarity of them are not comparable to the authentic ones. The major flaws are:

- The picture doesn't have the fine textures one would expect from wall or door materials.
- Half of the number signs were not readable.

Further training could probably help, 150 epochs may not be enough for this model.

Diversity

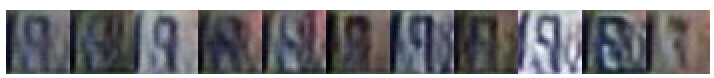
On the variety side, the generated samples did reflect a lot of differences in color and style.

Disentangled representation

I sampled a baseline seed z, and its image decoded were shown below:

Then for the 100 latent variable dimensions, I sampled every 10th of them and added 5.0 to the dimension, the generated images are shown below:

- the first picture is generated with baseline z
- the next ones are generated with some z's components added by 5.0

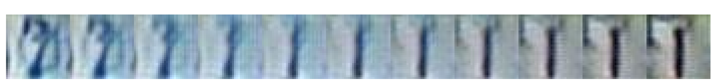


From the pictures we can conclude that the model learned disentangled representations. Modications to some components definitely changed the shape of writing.

Interpolating in two spaces

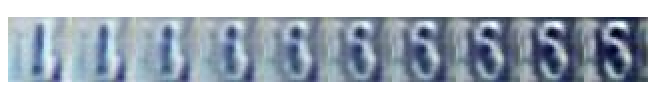
(a)

The images for alpha = 0, 0.1, 0.2, 0.3, 0.4, ... 1 are shown below. We can see the writing of 7 gradually changed to 1.



(b)

The images by interpolating directly in the data space are shown below. The clarity is not as good as the first one because these images are simply overlayed.



Appendix

Image generation code (auxiliary)

In [ ]:

```python
# Auxiliary code for image generations

import torch
import numpy as np
from torchvision import transforms, datasets
from torch.utils.data.sampler import SubsetRandomSampler
from torch import nn
from torch import optim
import math
import matplotlib.pyplot as plt

device = 'cuda' if torch.cuda.is_available() else 'cpu'

generator = torch.load("generator3.pt")
critic = torch.load("critic3.pt")
generator.eval()
critic.eval()
```

In [ ]:

```python
import scipy
import math
import matplotlib.pyplot as plt

def generate_samples(model, size=36, plot=True, path='figures/'):
    with torch.no_grad():
        images = model.generate(torch.randn(size, 100).to(device)).
        images = np.moveaxis(images, 1, 3)
        rows = []
        how_many_per_row = int(np.sqrt(size))
        for i in range(how_many_per_row):
            rows.append(
                np.hstack([images[j] for j in range(i*how_many_per_
            )
        big_picture = np.vstack(rows)

        scipy.misc.imsave(f"{path}/big_picture2.jpg", big_picture)
# COMPLETE QUALITATIVE EVALUATION
generate_samples(generator, size=100)
```

In [ ]:

```python
baseline_z = torch.randn(1, 100).cpu().detach().numpy()
# get ten copies for perturb
baseline_z = np.repeat(baseline_z, 11, axis = 0)
for row, dim in enumerate(range(0,100,10)):
    print(row, dim)
    baseline_z[row+1][dim] += 5


def generate_z_axis(model, path="figures/"):
    with torch.no_grad():
        # generate_baseline_model
        images = model.generate(torch.from_numpy(baseline_z).to(dev
        images = np.moveaxis(images, 1, 3)
        scipy.misc.imsave(f"{path}/z_disentangled.jpg", np.hstack([

generate_z_axis(generator)
```

In [ ]:

```
baseline_z0 = torch.randn(2, 100).cpu().detach().numpy()

def generate_image_space_interp(model, path="figures/"):
    with torch.no_grad():
        # generate_baseline_model
        images = model.generate(torch.from_numpy(baseline_z0).to(de
        images = np.moveaxis(images, 1, 3)
        interp = [
            images[0] * (10-i)/10 + images[1] * (i/10)
            for i in range(10)
        ]
        scipy.misc.imsave(f"{path}/image_space.jpg", np.hstack(inte

generate_image_space_interp(generator)
```

vae.ipynb, completed

Solution template for the question 1.6-1.7. This template consists of following steps. Except the step 2, you don't need to modify it to answer the questions.

1. Initialize libraries
2. **Insert the answers for the questions 1.1~1.5 below (this is the part you need to fill)**
3. Define data loaders
4. Define VAE network architecture
5. Initialize the model and optimizer
6. Train the model
7. Save the model
8. Load the model
9. Evaluate the model with importance sampling

Initialize libraries

In [1]:

```
import math
from torchvision.datasets import utils
import torch.utils.data as data_utils
import torch
import os
import numpy as np
from torch import nn
from torch.nn.modules import upsampling
from torch.functional import F
from torch.optim import Adam
```

Insert **the answers for the questions 1.1~1.5 below**

In [2]:

```python
def log_likelihood_bernoulli(mu, target):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherw

    *** note. ***

    :param mu: (FloatTensor) - shape: (batch_size x input_size) - T
    :param target: (FloatTensor) - shape: (batch_size x input_size)
    :return: (FloatTensor) - shape: (batch_size,) - log-likelihood
    """
    # init
    batch_size = mu.size(0)
    mu = mu.view(batch_size, -1)
    target = target.view(batch_size, -1)
    result = target * torch.log(mu) + (1 - target) * torch.log(1 -
    result = torch.sum(result, dim=1)
    return result

def log_likelihood_normal(mu, logvar, z):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherw

    *** note. ***

    :param mu: (FloatTensor) - shape: (batch_size x input_size) - T
    :param logvar: (FloatTensor) - shape: (batch_size x input_size)
    :param z: (FloatTensor) - shape: (batch_size x input_size) - Ta
    :return: (FloatTensor) - shape: (batch_size,) - log probability
    """
    # init
    batch_size = mu.size(0)
    mu = mu.view(batch_size, -1)
    logvar = logvar.view(batch_size, -1)
    z = z.view(batch_size, -1)
    part_a = np.log(2 * np.pi) + logvar
    part_b = (z - mu).pow(2) / logvar.exp()
    result = (-1.0 / 2.0) * torch.sum(part_a + part_b, dim=1)  # su
    return result

def log_mean_exp(y):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherw

    *** note. ***

    :param y: (FloatTensor) - shape: (batch_size x sample_size) - V
    :return: (FloatTensor) - shape: (batch_size,) - Output for log_
    """
    # init
    batch_size = y.size(0)
    sample_size = y.size(1)

    # log_mean_exp

    a_array = y.max(dim=1, keepdim=True)[0]
    exp_y = (y - a_array).exp()
```

```python
    sum_exp = torch.sum(exp_y, dim=1)  # sum by batch size
    result = (1.0 / sample_size * sum_exp).log() + a_array.view(-1)
    return result


def kl_gaussian_gaussian_analytic(mu_q, logvar_q, mu_p, logvar_p):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherw

    *** note. ***

    :param mu_q: (FloatTensor) - shape: (batch_size x input_size) -
    :param logvar_q: (FloatTensor) - shape: (batch_size x input_siz
    :param mu_p: (FloatTensor) - shape: (batch_size x input_size) -
    :param logvar_p: (FloatTensor) - shape: (batch_size x input_siz
    :return: (FloatTensor) - shape: (batch_size,) - kl-divergence o
    """
    # init
    batch_size = mu_q.size(0)
    mu_q = mu_q.view(batch_size, -1)
    logvar_q = logvar_q.view(batch_size, -1)
    mu_p = mu_p.view(batch_size, -1)
    logvar_p = logvar_p.view(batch_size, -1)

    pq = (
        logvar_p
        - logvar_q
        - 1.0
        + (logvar_q.exp() / logvar_p.exp())
        + ((mu_q - mu_p).pow(2) / logvar_p.exp())
    )
    result = (1.0 / 2.0) * torch.sum(pq, dim=1)
    return result

def kl_gaussian_gaussian_mc(mu_q, logvar_q, mu_p, logvar_p, num_sam
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherw

    *** note. ***

    :param mu_q: (FloatTensor) - shape: (batch_size x input_size) -
    :param logvar_q: (FloatTensor) - shape: (batch_size x input_siz
    :param mu_p: (FloatTensor) - shape: (batch_size x input_size) -
    :param logvar_p: (FloatTensor) - shape: (batch_size x input_siz
    :param num_samples: (int) - shape: () - The number of sample fo
    :return: (FloatTensor) - shape: (batch_size,) - kl-divergence o
    """
    # init
    batch_size = mu_q.size(0)
    input_size = np.prod(mu_q.size()[1:])
    mu_q = (
        mu_q.view(batch_size, -1)
        .unsqueeze(1)
        .expand(batch_size, num_samples, input_size)
    )
    logvar_q = (
        logvar_q.view(batch_size, -1)
```

```
            .unsqueeze(1)
            .expand(batch_size, num_samples, input_size)
        )
        mu_p = (
            mu_p.view(batch_size, -1)
            .unsqueeze(1)
            .expand(batch_size, num_samples, input_size)
        )
        logvar_p = (
            logvar_p.view(batch_size, -1)
            .unsqueeze(1)
            .expand(batch_size, num_samples, input_size)
        )

        # ==
        # Monte carlo kld

        # define the normal distribution to sample from
        stdev_q = (0.5 * logvar_q).exp()
        normal = torch.distributions.Normal(mu_q.float(), stdev_q.float
        sampled_x = normal.sample()
        f_q = logvar_q + (sampled_x - mu_q).pow(2) / logvar_q.exp()
        f_p = logvar_p + (sampled_x - mu_p).pow(2) / logvar_p.exp()
        q_p = f_q - f_p
        sampled_x_KL = torch.sum(q_p, dim=2)
        kld = -0.5 / num_samples * torch.sum(sampled_x_KL, dim=1)
        return kld
```

Define data loaders

In [3]:

```
def get_data_loader(dataset_location, batch_size):
    URL = "http://www.cs.toronto.edu/~larocheh/public/datasets/bina
    # start processing
    def lines_to_np_array(lines):
        return np.array([[int(i) for i in line.split()] for line in
    splitdata = []
    for splitname in ["train", "valid", "test"]:
        filename = "binarized_mnist_%s.amat" % splitname
        filepath = os.path.join(dataset_location, filename)
        utils.download_url(URL + filename, dataset_location)
        with open(filepath) as f:
            lines = f.readlines()
        x = lines_to_np_array(lines).astype('float32')
        x = x.reshape(x.shape[0], 1, 28, 28)
        # pytorch data loader
        dataset = data_utils.TensorDataset(torch.from_numpy(x))
        dataset_loader = data_utils.DataLoader(x, batch_size=batch_
        splitdata.append(dataset_loader)
    return splitdata
```

In [4]:

```
train, valid, test = get_data_loader("binarized_mnist", 64)
```

```
Downloading http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/binarized_mnist_train.amat (http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/binarized_mnist_train.amat) to binarized_mnist/binarized_mnist_train.amat
```

78405632/? [00:20<00:00, 8440060.34it/s]

```
Downloading http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/binarized_mnist_valid.amat (http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/binarized_mnist_valid.amat) to binarized_mnist/binarized_mnist_valid.amat
```

15687680/? [00:20<00:00, 8121116.75it/s]

```
Downloading http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/binarized_mnist_test.amat (http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/binarized_mnist_test.amat) to binarized_mnist/binarized_mnist_test.amat
```

15687680/? [00:03<00:00, 5080611.19it/s]

Define VAE network architecture

In [5]:

```
class Encoder(nn.Module):
    def __init__(self, latent_size):
        super(Encoder, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(784, 300),
            nn.ELU(),
            nn.Linear(300, 300),
            nn.ELU(),
            nn.Linear(300, 2 * latent_size),
        )

    def forward(self, x):
        batch_size = x.size(0)
        z_mean, z_logvar = self.mlp(x.view(batch_size, 784)).chunk(
        return z_mean, z_logvar

class Decoder(nn.Module):
    def __init__(self, latent_size):
        super(Decoder, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(latent_size, 300),
            nn.ELU(),
            nn.Linear(300, 300),
            nn.ELU(),
            nn.Linear(300, 784),
        )

    def forward(self, z):
        return self.mlp(z) - 5.

class VAE(nn.Module):
    def __init__(self, latent_size):
        super(VAE, self).__init__()
        self.encode = Encoder(latent_size)
        self.decode = Decoder(latent_size)

    def forward(self, x):
        z_mean, z_logvar = self.encode(x)
        z_sample = z_mean + torch.exp(z_logvar / 2.) * torch.randn_
        x_mean = self.decode(z_sample)
        return z_mean, z_logvar, x_mean

    def loss(self, x, z_mean, z_logvar, x_mean):
        ZERO = torch.zeros(z_mean.size())
        #kl = kl_gaussian_gaussian_mc(z_mean, z_logvar, ZERO, ZERO,
        kl = kl_gaussian_gaussian_analytic(z_mean, z_logvar, ZERO, 
        recon_loss = -log_likelihood_bernoulli(
            torch.sigmoid(x_mean.view(x.size(0), -1)),
            x.view(x.size(0), -1),
        ).mean()
        return recon_loss + kl
```

Initialize a model and optimizer

In [6]:

```
vae = VAE(100)
params = vae.parameters()
optimizer = Adam(params, lr=3e-4)
print(vae)
```

```
VAE(
  (encode): Encoder(
    (mlp): Sequential(
      (0): Linear(in_features=784, out_features=300, bias=True)
      (1): ELU(alpha=1.0)
      (2): Linear(in_features=300, out_features=300, bias=True)
      (3): ELU(alpha=1.0)
      (4): Linear(in_features=300, out_features=200, bias=True)
    )
  )
  (decode): Decoder(
    (mlp): Sequential(
      (0): Linear(in_features=100, out_features=300, bias=True)
      (1): ELU(alpha=1.0)
      (2): Linear(in_features=300, out_features=300, bias=True)
      (3): ELU(alpha=1.0)
      (4): Linear(in_features=300, out_features=784, bias=True)
    )
  )
)
```

Train the model

In [7]:

```
for i in range(20):
    # train
    for x in train:
        optimizer.zero_grad()
        z_mean, z_logvar, x_mean = vae(x)
        loss = vae.loss(x, z_mean, z_logvar, x_mean)
        loss.backward()
        optimizer.step()

    # evaluate ELBO on the valid dataset
    with torch.no_grad():
        total_loss = 0.
        total_count = 0
        for x in valid:
            total_loss += vae.loss(x, *vae(x)) * x.size(0)
            total_count += x.size(0)
        print('-elbo: ', (total_loss / total_count).item())
```

```
-elbo:  167.884521484375
-elbo:  142.44168090820312
-elbo:  129.18482971191406
-elbo:  121.11119079589844
-elbo:  115.97421264648438
-elbo:  113.12680053710938
-elbo:  110.89364624023438
-elbo:  109.18376159667969
-elbo:  107.71327209472656
-elbo:  106.6328353881836
-elbo:  105.85040283203125
-elbo:  105.10814666748047
-elbo:  104.2016830444336
-elbo:  103.8462905883789
-elbo:  102.88982391357422
-elbo:  102.59537506103516
-elbo:  102.35488891601562
-elbo:  101.95604705810547
-elbo:  101.50086212158203
-elbo:  101.28077697753906
```

Save the model

In [8]:

```
torch.save(vae, 'model.pt')
```

```
/home/andybai/anaconda3/lib/python3.7/site-packages/torch/serializatio
n.py:360: UserWarning: Couldn't retrieve source code for container of
type VAE. It won't be checked for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/andybai/anaconda3/lib/python3.7/site-packages/torch/serializatio
n.py:360: UserWarning: Couldn't retrieve source code for container of
type Encoder. It won't be checked for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/andybai/anaconda3/lib/python3.7/site-packages/torch/serializatio
n.py:360: UserWarning: Couldn't retrieve source code for container of
type Decoder. It won't be checked for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
```

Load the model

In [9]:
```python
vae = torch.load('model.pt')
```

Evaluate the $\log p_\theta(x)$ of the model on test by using importance sampling

In [11]:
```python
total_loss = 0.
total_count = 0
with torch.no_grad():
    #x = next(iter(test))
    for x in test:
        # init
        K = 200
        M = x.size(0)

        # Sample from the posterior
        z_mean, z_logvar = vae.encode(x)
        eps = torch.randn(z_mean.size(0), K, z_mean.size(1))
        z_samples = z_mean[:, None, :] + torch.exp(z_logvar / 2.)[:

        # Decode samples
        z_samples_flat = z_samples.view(-1, z_samples.size(-1)) # F
        x_mean_flat = vae.decode(z_samples_flat) # Push it through

        # Reshape images and posterior to evaluate probabilities
        x_flat = x[:, None].repeat(1, K, 1, 1, 1).reshape(M*K, -1)
        z_mean_flat = z_mean[:, None, :].expand_as(z_samples).resha
        z_logvar_flat =  z_logvar[:, None, :].expand_as(z_samples).
        ZEROS = torch.zeros(z_mean_flat.size())

        # Calculate all the probabilities!
        log_p_x_z = log_likelihood_bernoulli(torch.sigmoid(x_mean_f
        log_q_z_x = log_likelihood_normal(z_mean_flat, z_logvar_fla
        log_p_z = log_likelihood_normal(ZEROS, ZEROS, z_samples_fla

        # Recombine them.
        w = log_p_x_z + log_p_z - log_q_z_x
        log_p = log_mean_exp(w)

        # Accumulate
        total_loss += log_p.sum()
        total_count += M
    print('log p(x):', (total_loss / total_count).item())
```

```
log p(x): -95.54447174072266
```