Generative Models

Lecture 5

Variational Autoencoders

The *Variational Autoencoder* introduces randomness in both Encoder and Decoder.

The idea is that the data are samples from some conditional distribution

$$p(\mathbf{x}|\mathbf{z})$$

where $\mathbf{z}$ is some latent or hidden variable

The Encoder generates the distribution of the latent variable $\mathbf{z}$ dependent on the data points

$$q_\phi(\mathbf{z}|\mathbf{x})$$

The Decoder generates the reconstruction distribution

$$p_\theta(\mathbf{x}|\mathbf{z})$$

where

$$\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$$

is a sample from the latent variable distribution.

ϕ and θ denote the parameters of the Encoder and Decoder networks resp.

Here is how it works: we choose parametrized distributions q_θ and p_ϕ and we then optimize the parameters θ and ϕ such that for each $\mathbf{x}$ in the data set, we sample $\mathbf{z}$ from the distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and then maximize the log-likelihood

$$\log p_\theta(\mathbf{x}|\mathbf{z})$$

We view $q_\phi(\mathbf{z}|\mathbf{x})$ as the variational approximation to the posterior $p_\theta(\mathbf{z}|\mathbf{x})$

Then we can compute the KL-divergence as before

$$
\begin{aligned}
D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x}) &= -\mathbb{E}_{q_\phi}(\log\left(\frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})}\right) \\
&= -\int \log p_\theta(\mathbf{z}|\mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} + \int \log q_\phi(\mathbf{z}|\mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\int \log \frac{p_\theta(\mathbf{x}|\mathbf{z}) p_{prior}(\mathbf{z})}{p_\theta(\mathbf{x})} q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} + \int \log q_\phi(\mathbf{z}|\mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\mathbb{E}_{q_\phi}(\log p_\theta(\mathbf{x}|\mathbf{z})) - \mathbb{E}_{q\phi}(p_{prior}(\mathbf{z})) + \log p_\theta(\mathbf{x}) + \mathbb{E}_{q_\theta}(\log q_\phi(\mathbf{z}|\mathbf{x}))
\end{aligned}
$$

where $p_{prior}(\mathbf{z})$ is some suitable prior distribution of the latent variable $\mathbf{z}$

Rearranging terms we get

$$\begin{aligned}\log p_\theta(\mathbf{x}) &= \mathbb{E}_{q_\phi}(\log p_\theta(\mathbf{x}|\mathbf{z})) + \mathbb{E}_{q_\phi}(p_{prior}(\mathbf{z})) - \mathbb{E}_{q_\phi}(\log q_\phi(\mathbf{z}|\mathbf{x})) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x})) \\ &= \mathbb{E}_{q_\phi}(p_\theta(\mathbf{x}|\mathbf{z})) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_{prior}(\mathbf{z})) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x}))\end{aligned}$$

Thus to maximize the log likelihood $\log p_\theta(\mathbf{x})$ we have to maximize to $ELBO$,

$$\mathbb{E}_{q_\phi}(p_\theta(\mathbf{x}|\mathbf{z})) + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_{prior}(\mathbf{z})))$$

The first term is the reconstruction log-likelihood (we average over all latent samples).

The second term can be viewed as a regularization term that limits how far the approximation to the posterior can deviate from the prior (one often takes the posterior as $\mathcal{N}(0, I)$)

To do gradient ascent to maximize the $ELBO$ we need to estimate the gradients

$$\begin{aligned}\nabla_{\phi,\theta} ELBO &= \nabla_{\phi,\theta} \mathbb{E}_{q_\phi} \left(\log p_\theta(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z})\right) - \log q_\phi(\mathbf{z}|\mathbf{x})) \\ &= \nabla_{\phi,\theta} \mathbb{E}_{q_\phi} \left(\log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right)\end{aligned}$$

There is no problem computing ∇_θ since we can differentiate with respect to θ under $\mathbb{E}_{q_\phi}$

$$\nabla_\theta \mathbb{E}_{q_\phi} \left(\log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right) = \mathbb{E}_{q_\phi}(\nabla_\theta(\log p_\theta(\mathbf{x}, \mathbf{z})))$$

In order to differentiate with respect to ϕ we resort to the 'reparametrization trick'.

This means that we cannot arbitrarily choose the distribution output by the Encoder.

In practice this means that we let the Encoder output vectors $\mu_\phi(\mathbf{x})$ and $\sigma_\phi(\mathbf{x})$ and let the latent variable

$$\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_\phi(\mathbf{x}), \sigma^2_\phi(\mathbf{x})I)$$

i.e. q_ϕ is a multi-variate Gaussian with diagonal covariance matrix.

Then we can change variables

$$\mathbf{z} = \mu_\phi(\mathbf{x}) + \sigma_\phi(\mathbf{x})\,\underline{\varepsilon}$$

where $\underline{\varepsilon} \sim \mathcal{N}(0, I)$ so

$$\mathbb{E}_{q_\phi}(\log q_\phi(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{\mathcal{N}(0,I)}(\log q_\phi(\mu_\phi(\mathbf{x})) + \sigma_\phi(\mathbf{x})\,\underline{\varepsilon}))$$

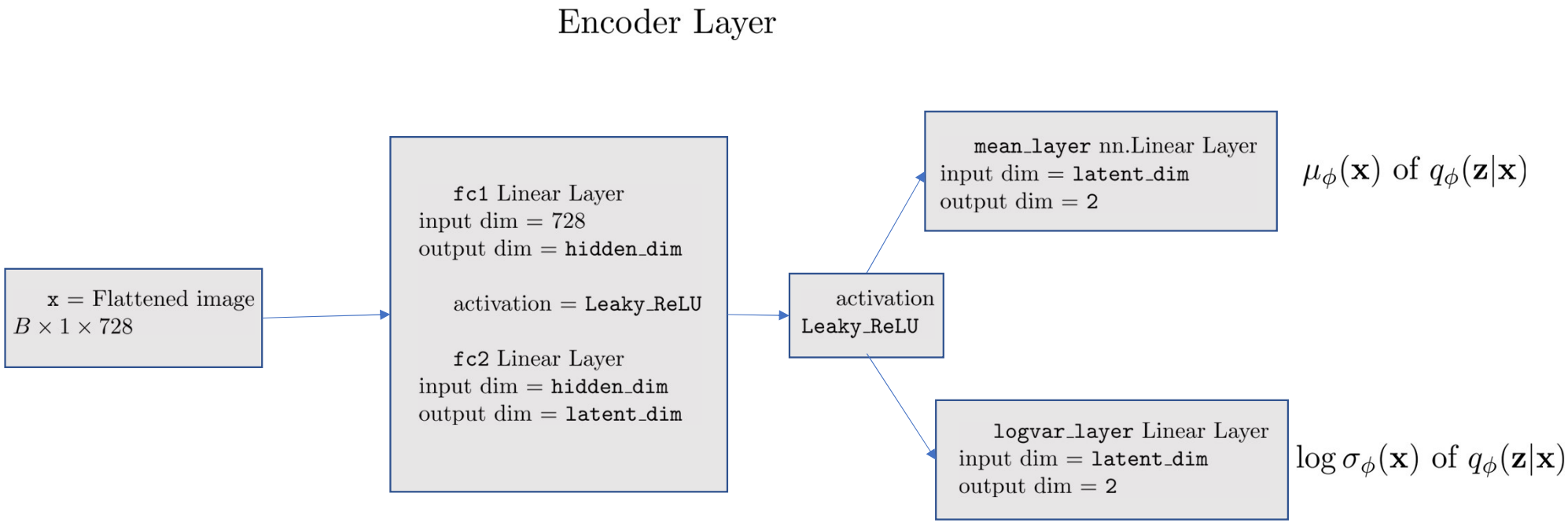
So we can estimate gradients by Monte Carlo and do gradient ascent to update the parameters θ and ϕ

This is done pretty much automatically by the pyro class `SVI`

We shall go through two VAE notebooks, that only uses Pytorch

First the imports

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
from torch.distributions.bernoulli import Bernoulli
from torch.optim import Adam
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
from mpl_toolkits.axes_grid1 import ImageGrid
from torchvision.utils import save_image, make_grid

```


Encoder Layer
x = Flattened image
$B \times 1 \times 728$
fc1 Linear Layer
input dim = 728
output dim = hidden_dim
activation = Leaky_ReLU
fc2 Linear Layer
input dim = hidden_dim
output dim = latent_dim
activation
Leaky_ReLU
mean_layer nn.Linear Layer
input dim = latent_dim
output dim = 2
$\mu_\phi(\mathbf{x})$ of $q_\phi(\mathbf{z}|\mathbf{x})$
logvar_layer Linear Layer
input dim = latent_dim
output dim = 2
$\log \sigma_\phi(\mathbf{x})$ of $q_\phi(\mathbf{z}|\mathbf{x})$

```python
class Encoder(nn.Module):

    def __init__(self,input_dim,hidden_dim,latent_dim):
        super().__init__()

        self.fc1 = nn.Linear(input_dim,hidden_dim)
        self.fc2 = nn.Linear(hidden_dim,latent_dim)

        self.mean_layer = nn.Linear(latent_dim,2)
        self.logvar_layer = nn.Linear(latent_dim,2)

    def forward(self,x):

        x = self.fc1(x)
        x = F.leaky_relu(x,0.2)
        x = self.fc2(x)
        x = F.leaky_relu(x,0.2)

        mean = self.mean_layer(x)
        logvar = self.logvar_layer(x)

        return mean, logvar

```

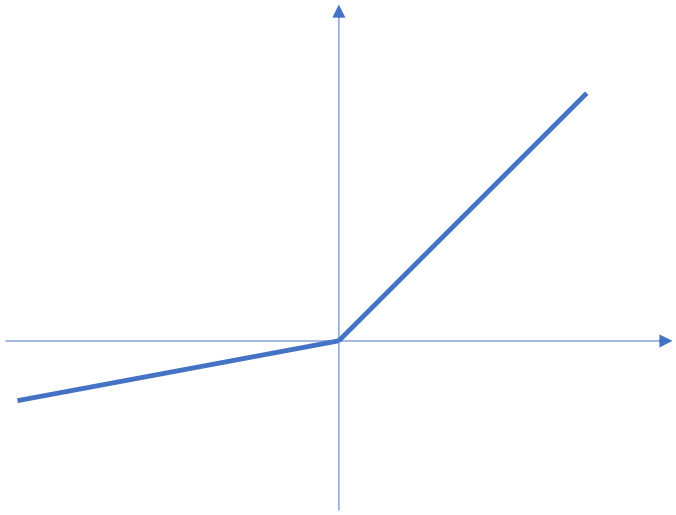
We sample from the latent distribution $\mathcal{N}(\mu_\phi(\mathbf{x}), \exp\log\sigma_\phi(\mathbf{x}))$ using the reparametrization trick i.e. we sample ε from $\mathcal{N}(0, I)$ and let

$$\mathbf{z} = \mu_\phi(\mathbf{x}) + \sqrt{\sigma_\phi(\mathbf{x})}\,\varepsilon$$

```
    def reparametrize(self,mean,var):
        epsilon = torch.randn_like(var)
        z = mean + torch.sqrt(var) * epsilon
        return z
```

The Decoder takes the sample $\mathbf{z}$ and applies several linear layers with `Leaky_ReLU` activations (the leaky_ relu function with parameter $\alpha > 0$ is given by

$$leaky\_relu(x, \alpha) = \begin{cases} x \text{ if } x \geq 0 \\ \alpha\, x \text{ if } x < 0 \end{cases}$$



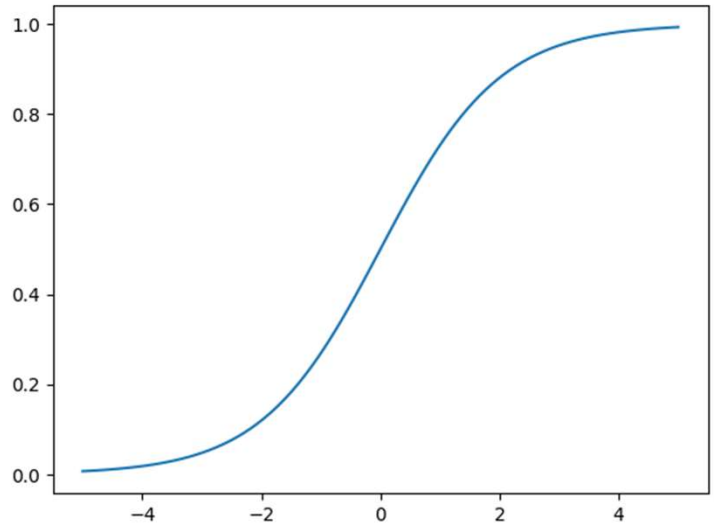
```python
class Decoder(nn.Module):

    def __init__(self,latent_dim,hidden_dim,input_dim):
        super().__init__()

        self.fc1 = nn.Linear(2,latent_dim)
        self.fc2 = nn.Linear(latent_dim,hidden_dim)
        self.fc3 = nn.Linear(hidden_dim,input_dim)

    def forward(self,z):

        z = self.fc1(z)
        z = F.leaky_relu(z,0.2)
        z = self.fc2(z)
        z = F.leaky_relu(z,0.2)
        z = self.fc3(z)

        z = F.sigmoid(z).to(torch.float)

        return z

```

Last output has dimension 784

The final linear layer outputs a 784 dimensional vector and we apply the sigmoid function σ

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

to each coordinate to get a 784 dimensional vector with coordinates between 0 and 1



The VAE itself is basically just combining the Encoder and Decoder.

The dataset consists of pairs `img,label`. For this model we are only interested in the `img` part, so we first take out the `img` part and then flatten the image from a $(1, 28, 28)$ tensor to a $1, 784)$ tensor

```
class VAE(nn.Module):

    def __init__(self,input_dim=784,hidden_dim=400,latent_dim=200):
        super().__init__()

        self.encoder = Encoder(input_dim,hidden_dim,latent_dim)
        self.decoder = Decoder(latent_dim,hidden_dim,input_dim)

    def reparametrize(self,mean,var):
        epsilon = torch.randn_like(var)
        z = mean + torch.sqrt(var) * epsilon
        return z

    def forward(self,x):

        x = x[0].reshape(-1,1,28*28)

        mean, logvar = self.encoder(x)

        u = self.reparametrize(mean,torch.exp(logvar))

        z = self.decoder(u)

        return z, mean, logvar
```

To train the model we need a loss function. The object of training is to maximize the $ELBO$ (or minimize $-ELBO$)

$$\mathbb{E}_{q_\phi}(\log p_\theta(\mathbf{x}|\mathbf{z})) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_{prior}(\mathbf{z}))$$

The distribution

$$p_\theta(\mathbf{x}|\mathbf{z}) = Bernoulli(\mathbf{z})(\mathbf{x})$$

where $\mathbf{z}$ is the output of the decoder

We have

$$\log Bernoulli(z)(x) = x \log z + (1 - x) \log(1 - z)$$

This is the negative `torch.nn.BCELoss()(z,x)` function

CLASS torch.nn.BCELoss(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

$$\log p_\theta(\mathbf{x}|\mathbf{z}) = -\texttt{torch.nn.BCELoss}(\texttt{reduction} = \text{`}\texttt{sum}')(\mathbf{z}, \mathbf{x})$$

To compute the KL-divergence, we have

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|mean(\mathbf{x}, \exp logvar(\mathbf{x}))$$

and

$$p_{prior}(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, I)$$

so we are computing the KL-divergence between two Gaussians

Let $mean(\mathbf{x}) = (\mu_1, \mu_2)$ and $var(\mathbf{x}) = (\sigma_1, \sigma_2))$

$$D_{KL}(\mathcal{N}\left((\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}\right) ||\mathcal{N}(0, I))$$

$$= \int \left(\log \mathcal{N}(\mathbf{z}|0, I) - \log \mathcal{N}(\mathbf{z}|\left((\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}\right)\right) \mathcal{N}(\mathbf{z}|\left((\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}\right) d\mathbf{z}$$

We have

$$\mathcal{N}(\mathbf{z}|0, I) = \frac{1}{2\pi} \exp(-\frac{1}{2}\mathbf{z}\mathbf{z}^T)$$

and

$$\mathcal{N}(\mathbf{z}|(\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}) = \frac{1}{2\pi\sqrt{\sigma_1\,\sigma_2}} \exp(-\frac{1}{2}(\mathbf{z}-(\mu_1, \mu_2)) \begin{pmatrix} \sigma_1^{-1} & 0 \\ 0 & \sigma_2^{-1} \end{pmatrix} (\mathbf{z}-(\mu_1, \mu_2))^T)$$

So

$$\begin{aligned} &\log \mathcal{N}(\mathbf{z}|(\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}) - \log \mathcal{N}(\mathbf{z}|0, I) \\ &= \frac{1}{2} \log \sigma_1\sigma_2 - \frac{1}{2}(\mathbf{z} - (\mu_1, \mu_2)) \begin{pmatrix} \sigma_1^{-1} & 0 \\ 0 & \sigma_2^{-1} \end{pmatrix} (\mathbf{z} - (\mu_1, \mu_2))^T) + \frac{1}{2}\mathbf{z}\mathbf{z}^T \end{aligned}$$

This expression equals

$$-\frac{1}{2}\log\sigma_1\sigma_1 - \frac{1}{2}\left(\frac{(z_1-\mu_1)^2}{\sigma_1} + \frac{(z_2-\mu_2)^2}{\sigma_2}\right) + \frac{1}{2}(z_1^2+z_2^2)$$

$$\int(-\frac{1}{2}\log\sigma_1\sigma_2\mathcal{N}(\mathbf{z}|\left((\mu_1,\mu_2),\begin{pmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{pmatrix}\right)d\mathbf{z} = -\frac{1}{2}\log\sigma_1\sigma_2$$

$$\int -\frac{1}{2}\left(\frac{(z_1-\mu_1)^2}{\sigma_1} + \frac{(z_2-\mu_2)^2}{\sigma_2}\right)\mathcal{N}(\mathbf{z}|\left((\mu_1,\mu_2),\begin{pmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{pmatrix}\right)d\mathbf{z} = -1$$

Finally

$$\int (z_1 - \mu_1)^2 \mathcal{N}(\mathbf{z}| \left((\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \right) d\mathbf{z} = \sigma_1$$

and

$$(z_1 - \mu_1)^2 = z_1^2 + \mu_1^2 - 2z_1\mu_1$$

so

$$\begin{aligned} &\int z_1^2 \mathcal{N}(\mathbf{z}| \left((\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \right) d\mathbf{z} \\ &= \sigma_1^2 - \mu_1^2 + 2\mu_1 \int z_1 \mathcal{N}(\mathbf{z}| \left((\mu_1, \mu_2), \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \right) d\mathbf{z} \\ &= \sigma_1^2 - \mu_1^2 + 2\mu_1^2 = \sigma_1^2 + \mu_1^2 \end{aligned}$$

Thus we finally get

$$D_{KL}(\mathcal{N}\left((\mu_1,\mu_2),\begin{pmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{pmatrix}\right)||\mathcal{N}(0,I)) = \frac{1}{2}(\mu_1^2+\mu_2^2+\sigma_1+\sigma_2-2-\log\sigma_1\sigma_2)$$

If `z,mean,logvar = vae(x)` we can code this as

```
KL_div = - 0.5 * torch.sum(1+ logvar - mean**2 - torch.exp(logvar))
```

The total loss function i.e. the $-ELBO$ is then

```python
def loss_function(z,x,mean,logvar):
    reconstruction_loss = F.binary_cross_entropy(x,z,reduction='sum')
    KL_div =  - 0.5 * torch.sum(1+ logvar - mean**2 - torch.exp(logvar))

    return reconstruction_loss + KL_div
```

We can now run the training loop

We use the Adam optimizer

```
optimizer = Adam(vae.parameters(),lr=1e-3)
```

```
epochs = 50
for e in range(epochs):
    overall_loss = 0.0
    for i,img in enumerate(train_loader):

        x = img[0].reshape(-1,28*28)

        z,mean,logvar = vae(x)

        optimizer.zero_grad()

        loss = loss_function(x,z,mean,logvar)

        overall_loss += loss.item()

        loss.backward()
        optimizer.step()

    print("\tEpoch", e + 1, "\tAverage Loss: ", overall_loss/(i*batch_s
```

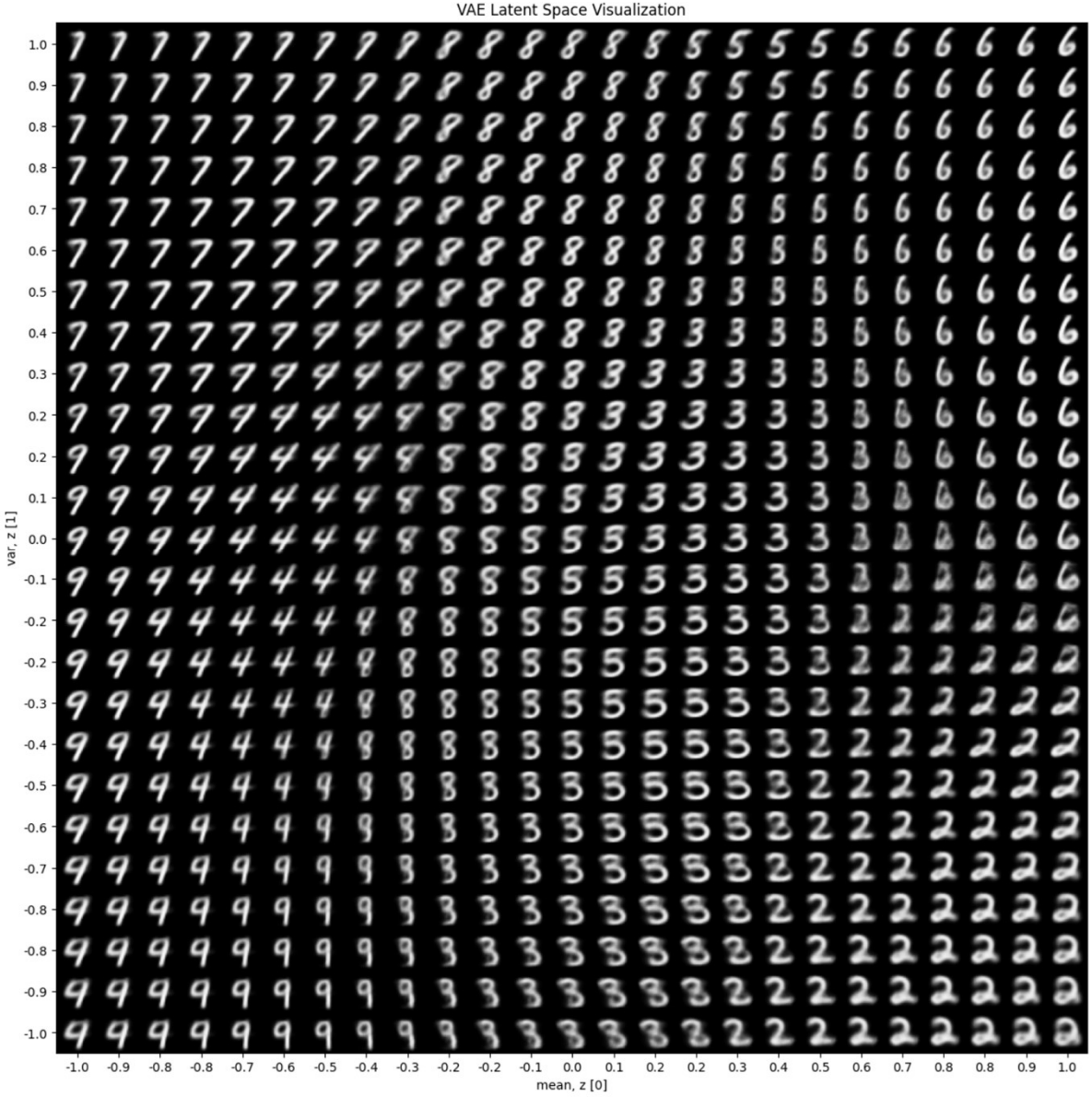
22

```
Epoch 1         Average Loss:  169.05433931226523
Epoch 2         Average Loss:  160.5248147139764
Epoch 3         Average Loss:  156.35052939795492
Epoch 4         Average Loss:  153.65856944190838
Epoch 5         Average Loss:  151.8359131103923
Epoch 6         Average Loss:  150.75318972375834
Epoch 7         Average Loss:  149.655738600793
Epoch 8         Average Loss:  149.00604903028486
Epoch 9         Average Loss:  148.3741138225167
Epoch 10        Average Loss:  147.7428735718385
Epoch 11        Average Loss:  147.18598841493636
Epoch 12        Average Loss:  146.71981404554467
```

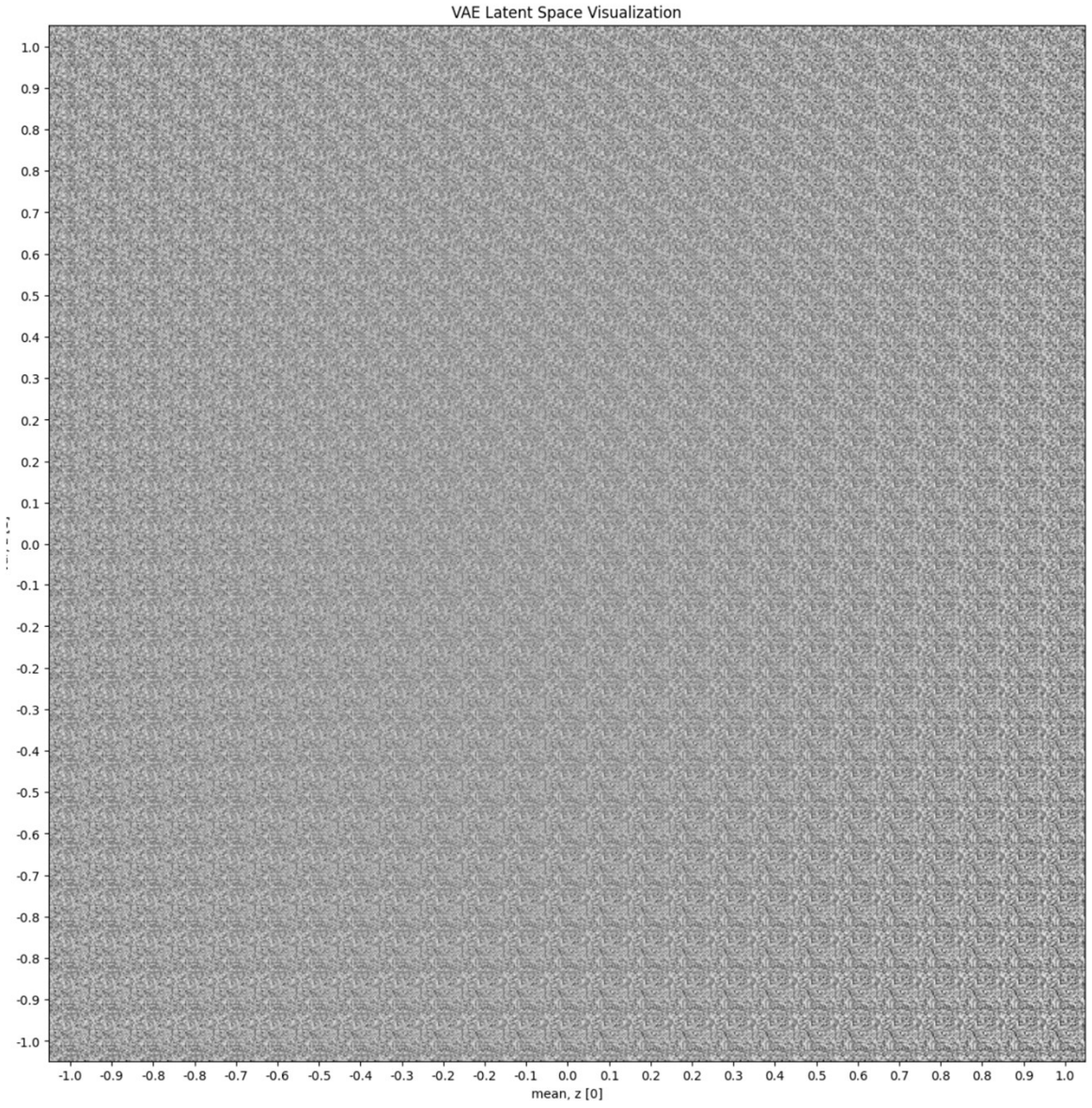
This function moves around in (mean,logvar) latent space and samples a new **z** for every `(mena,logvar)` and then shows the sampled image.

Remark how the images slowly change between different digits

```python
def plot_latent_space(model, scale=5.0, n=25, digit_size=28, figsize=15)
    # display a n*n 2D manifold of digits
    figure = np.zeros((digit_size * n, digit_size * n))

    # construct a grid
    grid_x = np.linspace(-scale, scale, n)
    grid_y = np.linspace(-scale, scale, n)[::-1]

    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            z_sample = torch.tensor([[xi, yi]], dtype=torch.float)
            x_decoded = vae.decoder(z_sample)
            digit = x_decoded[0].detach().cpu().reshape(digit_size, digi
            figure[i * digit_size : (i + 1) * digit_size, j * digit_size

    plt.figure(figsize=(figsize, figsize))
    plt.title('VAE Latent Space Visualization')
    start_range = digit_size // 2
    end_range = n * digit_size + start_range
    pixel_range = np.arange(start_range, end_range, digit_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("mean, z [0]")
    plt.ylabel("var, z [1]")
    plt.imshow(figure, cmap="Greys_r")
    plt.show()
```


VAE Latent Space Visualization
var, z [1]
1.0
0.9
0.8
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.2
0.1
0.0
-0.1
-0.2
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.8
-0.9
-1.0
-1.0 -0.9 -0.8 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.2 -0.1 0.0 0.1 0.2 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.8 0.9 1.0
mean, z [0]

Same function with an untrained model



Next we are going through another notebook, implementing a VAE using `pyro`

As usual, first the imports

```
import os

import numpy as np
import torch
from pyro.contrib.examples.util import MNIST
import torch.nn as nn
import torchvision.transforms as transforms

import pyro
import pyro.distributions as dist
import pyro.contrib.examples.util  # patches torchvision
from pyro.infer import SVI, Trace_ELBO
from pyro.optim import Adam
```

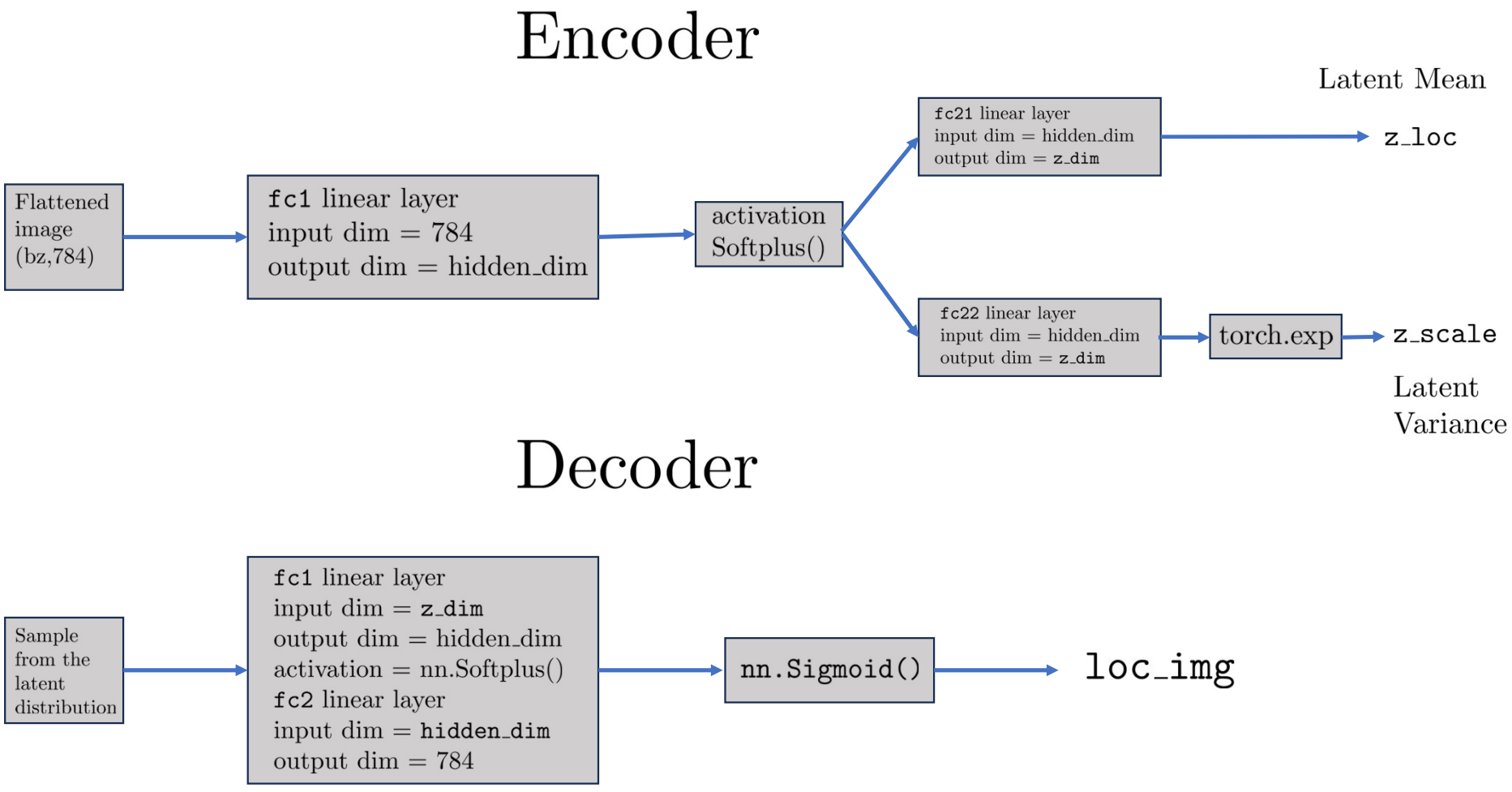
The data import and setting up the data loaders is the same as the previous notebook

```
transform = transforms.Compose([transforms.ToTensor()])

path = './data'

train_dataset = MNIST(path,transform=transform,download=True)
test_dataset = MNIST(path,transform=transform,download=True)

batch_size=100
train_loader = DataLoader(train_dataset,batch_size=batch_size,shuffle=True)
test_loader = DataLoader(test_dataset,batch_size=batch_size,shuffle=False)
```

We write an Encoder and a Decoder

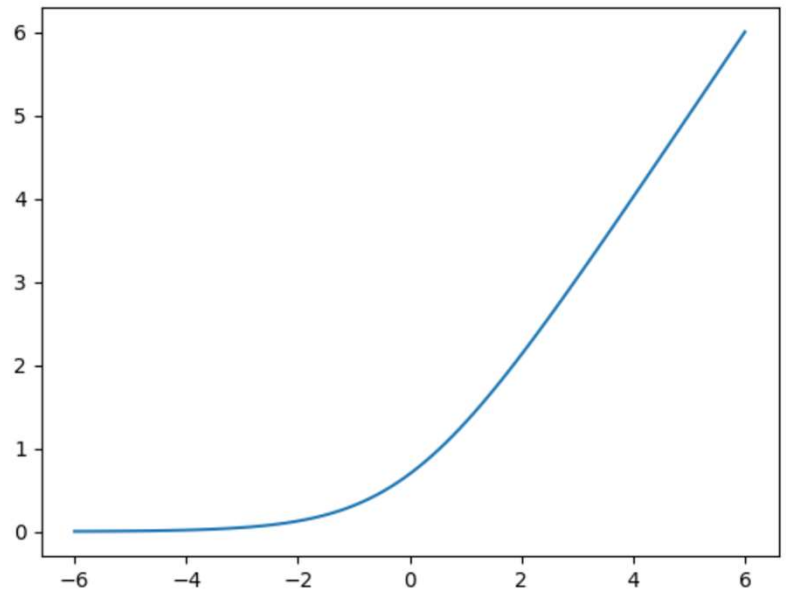
```python
class Encoder(nn.Module):
    def __init__(self, z_dim, hidden_dim):
        super().__init__()
        # setup the three linear transformations used
        self.fc1 = nn.Linear(784, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, z_dim)
        self.fc22 = nn.Linear(hidden_dim, z_dim)
        # setup the non-linearities
        self.softplus = nn.Softplus()

    def forward(self, x):
        # define the forward computation on the image x
        # first shape the mini-batch to have pixels in the rightmost dim
        x = x.reshape(-1, 784)
        # then compute the hidden units
        hidden = self.softplus(self.fc1(x))
        # then return a mean vector and a (positive) square root covaria
        # each of size batch_size x z_dim
        z_loc = self.fc21(hidden)
        z_scale = torch.exp(self.fc22(hidden))
        return z_loc, z_scale
```

```python
class Decoder(nn.Module):
    def __init__(self, z_dim, hidden_dim):
        super().__init__()
        # setup the two linear transformations used
        self.fc1 = nn.Linear(z_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, 784)
        # setup the non-linearities
        self.softplus = nn.Softplus()
        self.sigmoid = nn.Sigmoid()

    def forward(self, z):
        # define the forward computation on the latent z
        # first compute the hidden units
        hidden = self.softplus(self.fc1(z))
        # return the parameter for the output Bernoulli
        # each is of size batch_size x 784
        loc_img = self.sigmoid(self.fc2(hidden))
        return loc_img
```


Encoder
Latent Mean
Flattened image (bz,784)
fc1 linear layer
input dim = 784
output dim = hidden_dim
activation Softplus()
fc21 linear layer
input dim = hidden_dim
output dim = z_dim
z_loc
fc22 linear layer
input dim = hidden_dim
output dim = z_dim
torch.exp
z_scale
Latent Variance
Decoder
Sample from the latent distribution
fc1 linear layer
input dim = z_dim
output dim = hidden_dim
activation = nn.Softplus()
fc2 linear layer
input dim = hidden_dim
output dim = 784
nn.Sigmoid()
loc_img

Softplus activation function

$$softplus(x) = \log(1 + \exp(x))$$



The Encoder and Decoder are basically the same as in the Pytorch VAE.

The differences come in the VAE class

Using the `pyro SVI` class we need a `model` which implements

$$p_\theta(\mathbf{x}|\mathbf{z})p_{prior}(\mathbf{z})$$

The prior is $\mathcal{N}(0, I)$ where I is the `z_dim` $\times$ `z_dim` identity matrix

```python
class VAE(nn.Module):
    # by default our latent space is 50-dimensional
    # and we use 400 hidden units
    def __init__(self, z_dim=50, hidden_dim=400, use_cuda=False):
        super().__init__()
        # create the encoder and decoder networks
        self.encoder = Encoder(z_dim, hidden_dim)
        self.decoder = Decoder(z_dim, hidden_dim)

        if use_cuda:
```

We need to register the decoder with `pyro`

The `z_loc` and `z_scale` will be updated through the `svi` updates

Sampling from the latent distribution and applying the `decoder`

The likelihood $p(\mathbf{x}|\mathbf{z})$ is the `Bernoulli` distribution with probabilities `loc_img`

```python
    # define the model p(x|z)p(z)
    def model(self, x):
        # register PyTorch module `decoder` with Pyro
        pyro.module("decoder", self.decoder)
        with pyro.plate("data", x.shape[0]):
            # setup hyperparameters for prior p(z)
            z_loc = x.new_zeros(torch.Size((x.shape[0], self.z_dim)))
            z_scale = x.new_ones(torch.Size((x.shape[0], self.z_dim)))
            # sample from prior (value will be sampled by guide when computing the ELBO)
            z = pyro.sample("latent", dist.Normal(z_loc, z_scale).to_event(1))
            # decode the latent code z
            loc_img = self.decoder(z)
            # score against actual images
            pyro.sample("obs", dist.Bernoulli(loc_img,validate_args=False).to_event(1), obs=x.reshape(-1, 784))
```

The `guide` is the Variational approximation to the posterior $p(\mathbf{z}|\mathbf{x})$.

In the VAE this is the latent distribution $q_\phi(\mathbf{z}|\mathbf{x})$ with the parameters mean = `z_loc` and variance = `z_scale`

The `encoder` has to be registered with `pyro`

The `encoder` outputs `z_loc` and `z_scale`, the mean and variance of the latent distribution

Sampling from the latent distribution

```
32      # define the guide (i.e. variational distribution) q(z|x)
33      def guide(self, x):
34          # register PyTorch module `encoder` with Pyro
35          pyro.module("encoder", self.encoder)
36          with pyro.plate("data", x.shape[0]):
37              # use the encoder to get the parameters used to define q(z|x)
38              z_loc, z_scale = self.encoder(x)
39              # sample the latent code z
40              pyro.sample("latent", dist.Normal(z_loc, z_scale).to_event(1))
41
```

Both the `model` and the `guide` are methods in the `VAE` class.

We can now write the training loop

We first instantiate the `VAE`, set the `optimizer` and instantiate the `SVI`

Instantiate the `VAE`

```
1 vae = VAE()
```

Setting the optimizer

```
1 optimizer = Adam({"lr": 1.0e-3})
```

Instantiating the `SVI`

```
1 svi = SVI(vae.model, vae.guide, optimizer, loss=Trace_ELBO())
```

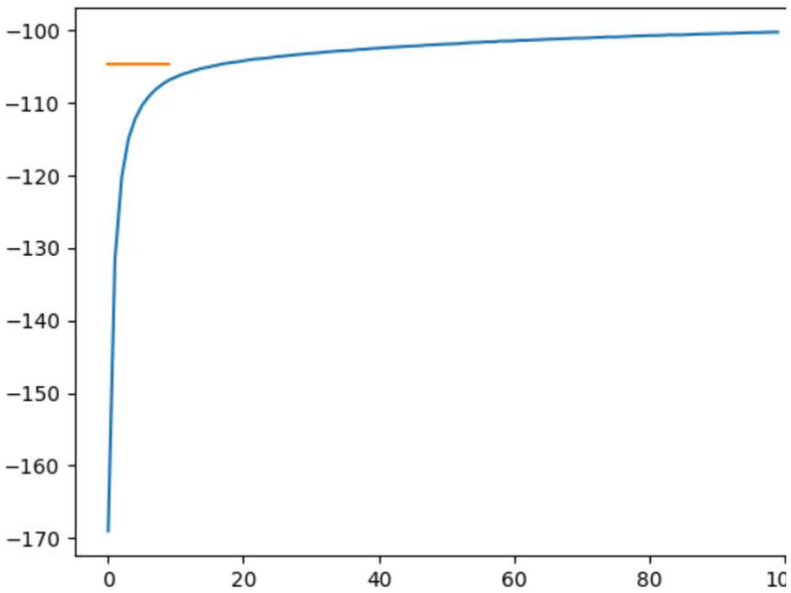
The training loop

The `svi` computes the loss (i.e. the $ELBO$), computes the gradients of the $ELBO$ and does the gradient step to update the parameters i.e. the parameters in the `encoder` and the `decoder`. This is why they have to be registered with `pyro`

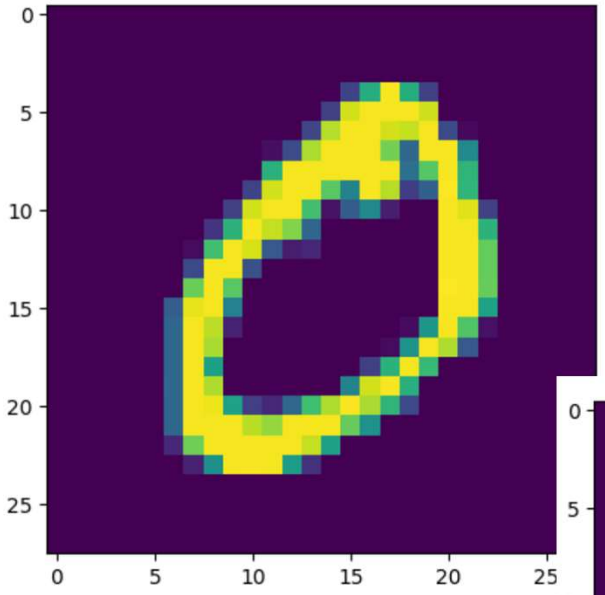
Here we evaluate the loss on the test set using the `svi.evaluate_loss` command

```python
epochs = 100

len_train_set = len(train_loader.dataset)
len_test_set = len(test_loader.dataset)

train_elbo = []
test_elbo = []
# training loop
for epoch in range(epochs):

    epoch_loss = 0.

    # do a training epoch over each mini-batch x returned
    # by the data loader
    for x, _ in train_loader:

        pyro.clear_param_store()

        loss = svi.step(x)

        epoch_loss += loss

    total_epoch_loss_train = epoch_loss / len_train_set

    train_elbo.append(-total_epoch_loss_train)

    print("[epoch %03d]  average training loss: %.4f" % (epoch, total_epoch_loss_train))

    if epoch % 10 == 0:

            test_loss = 0.
            # compute the loss over the entire test set
            for x, _ in test_loader:
                test_loss += svi.evaluate_loss(x)


            average_test_loss = test_loss / len_test_set
            test_elbo.append(-total_epoch_loss_test)
            print("[epoch %03d] average test loss: %.4f" % (epoch, total_epoch_loss_test))
```

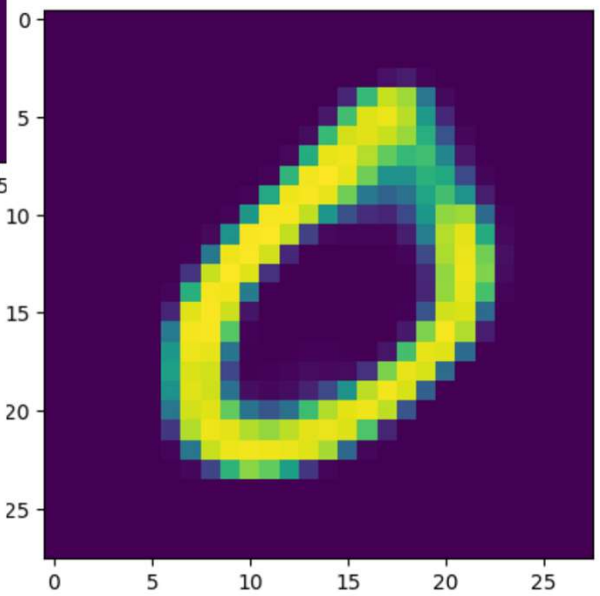
ELBO



Original



Reconstruction



Using a different dataset

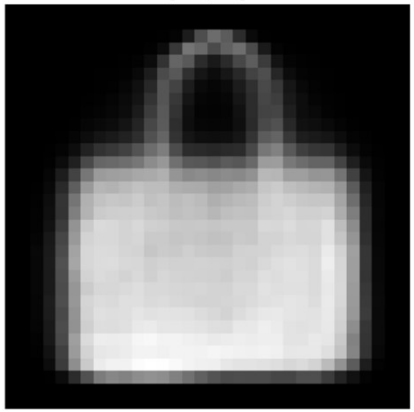
```
transform = transforms.Compose(
    [transforms.Pad(padding=2), transforms.ToTensor()]
)

path = './data'

train_dataset = FashionMNIST(
    path, train=True, download=True, transform=transform)
test_dataset = FashionMNIST(
    path, train=False, download=True, transform=transform)

batch_size=100
train_loader = DataLoader(train_dataset,batch_size=batch_size,shuffle=True)
test_loader = DataLoader(test_dataset,batch_size=batch_size,shuffle=False)

```

These images are 32×32 pixels

```
num_samples = 25
sample_images = [image[0][i,0] for i in range(num_samples)]

fig = plt.figure(figsize=(5,5))
grid = ImageGrid(fig,111,nrows_ncols=(5,5), axes_pad=0.1)

for ax, im in zip(grid,sample_images):
    ax.imshow(im,cmap='gray')
    ax.axis('off')
```

Generated Images

[0.0,1.0]



[1.0,1.0]