## 1 Variational Autoencoders

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

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

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

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

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

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.

$\phi$ and $\theta$ denote the parameters of the Encoder and Decoder networks resp.

Here is how it works: we choose parametrized distributions $q_\theta$ and $p_\phi$ and we then optimize the parameters $\theta$ and $\phi$ 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

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

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

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

It follows that we can express the log-likelihood as Rearranging terms we get

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

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

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

Since the KL-divergence is always $\geq 0$ we have

$$\log p_\theta(\mathbf{x}) \geq ELBO$$

so we will want to maximize the $ELBO$ i.e. to find

$$\operatorname*{argmax}_{\theta,\phi} \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left(\log \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})}\right)$$

Rewriting

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

where $p_{prior}(\mathbf{z})$ is some prior distribution of $\mathbf{z}$ which we typically choose as $\mathcal{N}(0, I)$, we get

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

The $\log p_\theta(\mathbf{x}|\mathbf{z})$ can be viewed as a reconstruction log likelihood dependent on a specific value of the latent variable $\mathbf{z}$ and so we want to maximize the expectation over all the values of $\mathbf{z}$

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

The KL-divergence term

$$D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_{prior}(\mathbf{z}))$$

which we want to minimize can be viewed as a regularizer which tries to keep the latent distribution $q_\phi(\mathbf{z}|\mathbf{x})$ close to the prior $p_{prior}(\mathbf{z})$

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

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

We are now back in to the situation from section 1.2.

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

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

In order to differentiate with respect to $\phi$ 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. $q_\phi$ 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 $\theta$ and $\phi$

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

In the code example we again use the $MNIST$ data set.

The Encoder Layer takes as input a batch $B$ of images $B \times 1 \times 28 \times 28$ image, flattens to $B \times 1 \times 784$.
This goes through a linear Layer `fc1` and then a `Softplus` activation layer ($Softplus(\mathbf{u}) = \log(1 + \exp(\mathbf{u}))$)
Then the output of the activation layer is branched, one branch going to a linear layer, `fc21` which outputs the mean of the latent variable distribution $q_\theta(\mathbf{z}|\mathbf{x})$. The other branch going to another linear layer `fc22`, after applying exp , this is the diagonal in the covariance matrix of $q_\theta(\mathbf{z}|\mathbf{x})$.

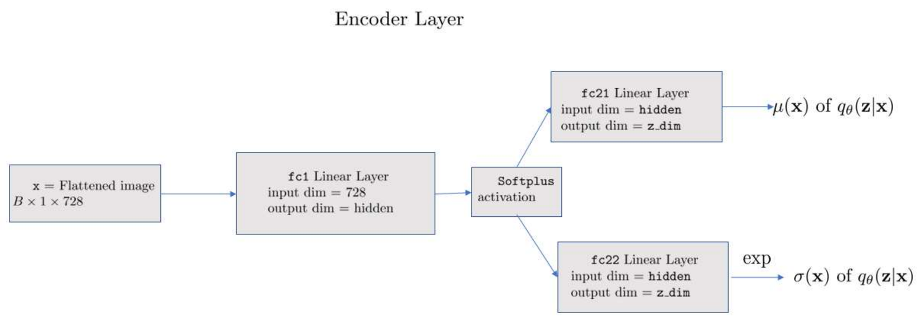
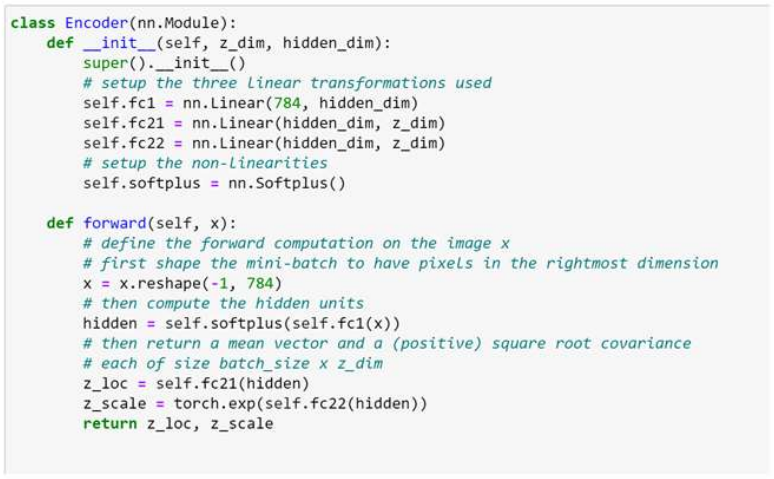


Figure 1:

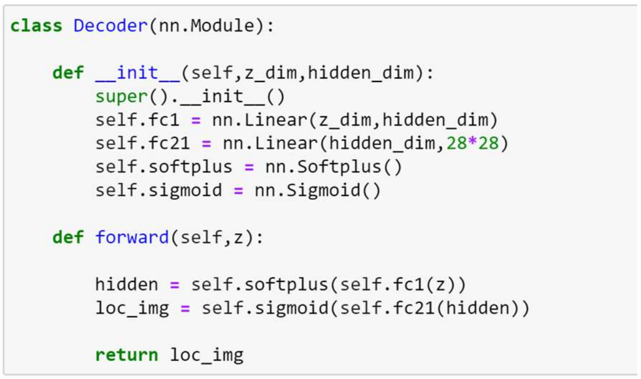


```python
class Encoder(nn.Module):
    def __init__(self, z_dim, hidden_dim):
        super().__init__()
        # setup the three linear transformations used
        self.fc1 = nn.Linear(784, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, z_dim)
        self.fc22 = nn.Linear(hidden_dim, z_dim)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

Figure 2:

The Decoder Layer takes a sample from the distribution of the latent variable $q_\theta(\mathbf{z}|\mathbf{x})$ and sends it through, first a linear layer `fc1`, an activation layer, `Softplus()`, another linear layer `fc 2` and finally applies a `sigmoid()` function, so the outputs are between 0 and 1



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

    def __init__(self,z_dim,hidden_dim):
        super().__init__()
        self.fc1 = nn.Linear(z_dim,hidden_dim)
        self.fc21 = nn.Linear(hidden_dim,28*28)
        self.softplus = nn.Softplus()
        self.sigmoid = nn.Sigmoid()

    def forward(self,z):

        hidden = self.softplus(self.fc1(z))
        loc_img = self.sigmoid(self.fc21(hidden))

        return loc_img
```

Figure 3:

The idea here is that the distribution $p_\phi(\mathbf{x}|\mathbf{z})$ is a product of Bernoulli distributions, one for each of the 784 pixels, and the output from the decoder, `loc_img`, is the vector of means for the these Bernoulli distributions.

The reconstructed image is the $28 \times 28$ tensor of samples from the Bernoulli distributions so a tensor of 0's and 1's

The reconstructed image actually looks better if instead of sampling the Bernoullis we just use the means as gray scale values.

We set the dimension of the latent variable to `z_dim` $= 50$.

The `model` registers the decoder object `decoder = Decoder(50,400.cuda()` in the `pyro.param_store` and defines the prior $p_{prior}(\mathbf{z}) = \mathcal{N}(0, I)$

Then we sample a $\mathbf{z}$ from the prior. The decoder outputs the means `loc_im` of the Bernoulli distributions, one for each of the 784 pixels. Su the likelihood is

$$p(\mathbf{x}|\mathbf{z}) = \prod_{x_i \in \mathbf{x}} Bernoulli(x_i; loc\_im_i)$$

The `guide` registers the `encoder = Encoder(50,400)` as a `pyro.module` and then defines the Variational approximation $q(\mathbf{z}|\mathbf{x})$ to the posterior

$$p_{post}(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p_{prior}(\mathbf{z})}{p(\mathbf{x})}$$

as

$$\mathcal{N}(\texttt{z\_loc}, \texttt{z\_scale})$$

where `z_loc,z_scale = encoder(x)`

Then we can run the training loop, minimizing $-ELBO$
The trained model can then be used to generate samples from the posterior i.e. the latent variables $\mathbf{z}$ and then generate images from the likelihood

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

Basically what we are doing is to generate new parameters $\mathbf{z}$ for the likelihood $p(\mathbf{x}|\mathbf{z})$ and then generate a sample from this distribution

We start with an image $\mathbf{x}$, sends it through the `encoder`, which outputs `z_loc,z_scale = encoder(x)`

Then generate a number of samples $\{\mathbf{z}_k\}$ from $q_\theta = \mathcal{N}(\texttt{z\_loc}, \texttt{z\_scale})$

```
samples = dist.Normal(z_loc, z_scale).sample(15)
loc_img = decoder(samples)
```

The `decoder` will send each of the $\mathbf{z}_k$'s to a 784 dimensional vector of means of Bernoulli distributions.

Sampling an image by sampling each pixel value (0 or 1) from the appropriate Bernoulli, generates an image.

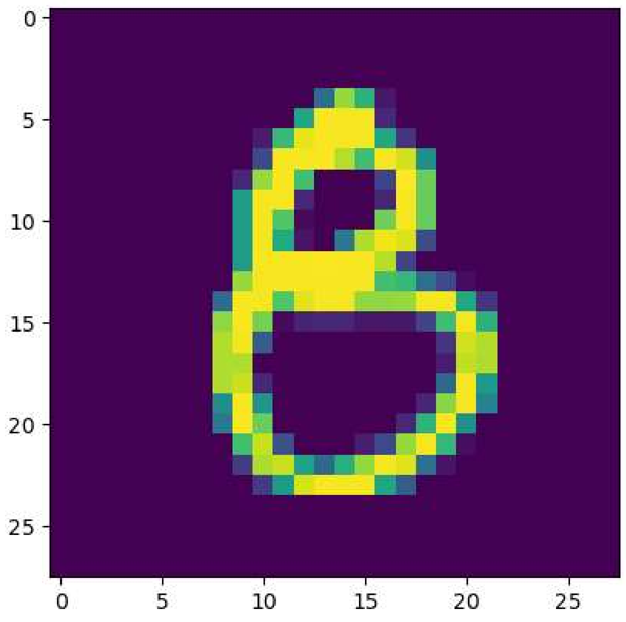


Figure 4: Input Image

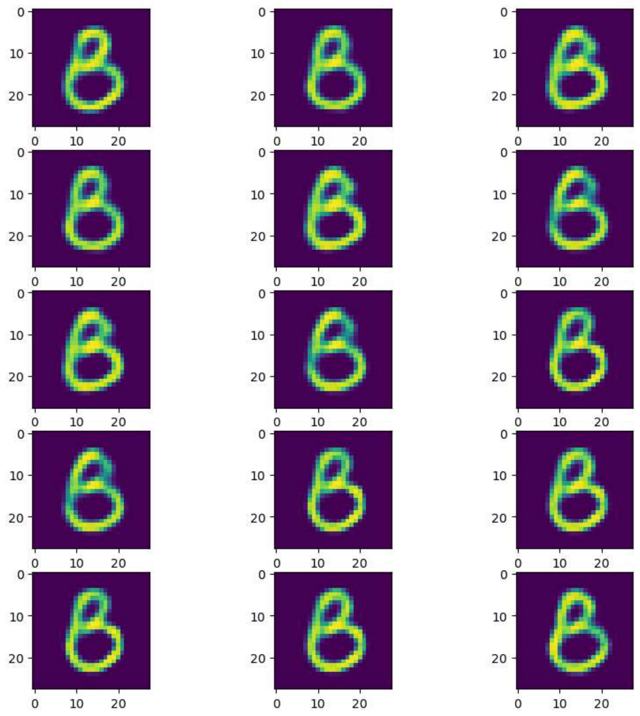


Figure 5: Generated Samples

If we sample from the Bernoullis we get purely black and white images

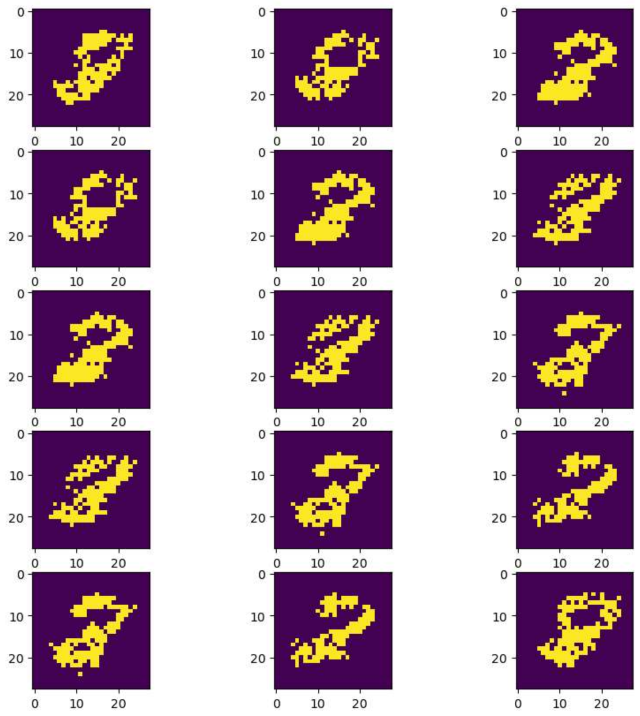


Figure 6: Samples from the Bernoulli Distributions

We can visualize a $VAE$ by fig. 31

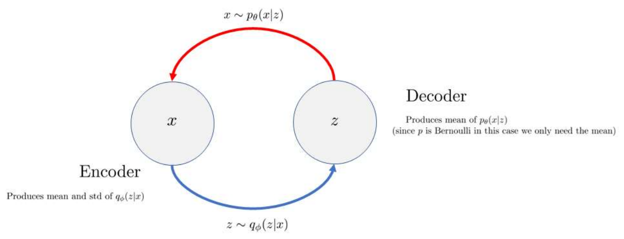

$x \sim p_\theta(x|z)$
$x$
$z$
Decoder
Produces mean of $p_\theta(x|z)$
(since $p$ is Bernoulli in this case we only need the mean)
Encoder
Produces mean and std of $q_\phi(z|x)$
$z \sim q_\phi(z|x)$

Figure 7: VAE