Kungliga Tekniska högskolan

Machine Learning advanced course

Assignment 1B

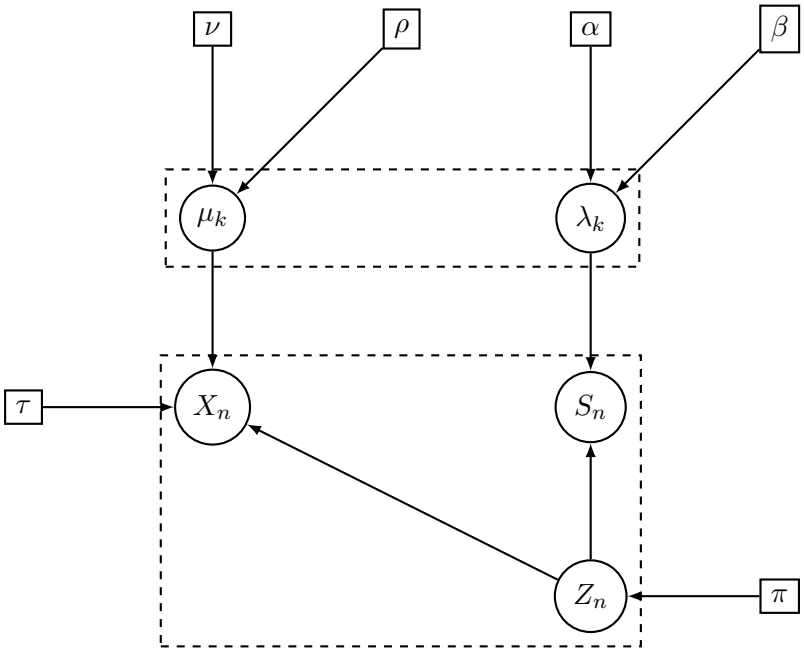
Student :
Yohan Pellerin

Course coordinator and teachers :
Jens Lagergren

December 10, 2023

1 CAVI for Earth quakes

Question 1.1.1



Question 1.1.2

Using conditional Independence, we can write :

$$p(X,S,Z,\lambda,\mu|\pi,\tau,\alpha,\beta,\nu,\rho) = \prod_n \left[p(X_n|Z_n,\mu,\tau)p(S_n|\lambda,Z_n)p(Z_n|\pi)\right] \prod_k \left[p(\mu_k|\beta,\alpha)p(\lambda|\nu,\rho)\right]$$

$$p(X,S,Z,\lambda,\mu|\pi,\tau,\alpha,\beta,\nu,\rho) = \prod_n \left[p(Z_n|\pi) \prod_k \left[p(X_n|Z_n=k,\mu_k,\tau)^{\mathbb{I}_{Z_n=k}} p(S_n|\lambda,Z_n=k)^{\mathbb{I}_{Z_n=k}}\right]\right] * \prod_k \left[p(\mu_k|\beta,\alpha)p(\lambda|\nu,\rho)\right]$$

$$\log p(X,S,Z,\lambda,\mu|\pi,\tau,\alpha,\beta,\nu,\rho) = \sum_n \left[\log p(Z_n|\pi) + \sum_k \mathbb{I}_{Z_n=k} \left[\log p(X_n|Z_n=k,\mu_k,\tau) + \log p(S_n|\lambda,Z_n=k)\right]\right] + \sum_k \log p(\mu_k|\beta,\alpha) + \log p(\lambda|\nu,\rho)$$

Question 1.1.3

The CAVI update equations are the followings.

$$\log q^*(\mu_i) \stackrel{+}{=} E_{-\mu_i}\left[P(X,S,Z,\lambda,\mu)\right]$$
$$\log q^*(Z_i) \stackrel{+}{=} E_{-Z_i}\left[P(X,S,Z,\lambda,\mu)\right]$$
$$\log q^*(\lambda_i) \stackrel{+}{=} E_{-\lambda_i}\left[P(X,S,Z,\lambda,\mu)\right]$$

The first equation

Let's focus on the first equation, using the expression from the previous question, and only keeping the terms that depend on μ_i, we get :

$$\log q^*(\mu_i) \stackrel{+}{=} \sum_{n,k} E_{-\mu_i}\left[\mathbb{I}_{Z_n=k} \log p(X_n|Z_n=k,\mu_k,\tau)\right] + \sum_k E_{-\mu_i}\left[\mathbb{I}_{Z_n=k} \log p(\mu_k|\nu,\rho)\right]$$

Now, let's turn our attention to the first expectation. We know $X_n|Z_n=k,\mu,\tau$ follows a 2D Normal$(\mu_i,\tau.I)$
If $k=i$, $\log p(X_n|Z_n=k,\mu_i,\tau) \stackrel{+}{=} -\frac{1}{2\tau}(\mu_i - X_n)^T(\mu_i - X_n)$
But if $k \neq i$ this term doesn't depend on μ_i

$$\begin{aligned}\sum_{k,n} E_{-\mu_i}\left[\mathbb{I}_{Z_n=k} \log p(X_n|Z_n=k,\mu,\tau)\right] &\stackrel{+}{=} -\frac{1}{2\tau}\sum_n (\mu_i - X_n)^T(\mu_i - X_n) q(Z_n=i) \\ &\stackrel{+}{=} -\frac{1}{2\tau}\left[\mu_i^T\mu_i - \left(\sum_n q(Z_n=i)X_n^T\right)\mu_i - \mu_i^T\sum_n q(Z_n=i)X_n\right]\end{aligned}$$

For the second expectation, in a similar way, as we know $\lambda_k|\nu,\rho$ follows a 2D Normal$(\nu,\rho.I)$ we get :

$$\begin{aligned}\sum_k E_{-\mu_i}\left[\log p(\mu_k|\nu,\rho)\right] &\stackrel{+}{=} -\frac{1}{2\rho}(\mu_i-\nu)^T(\mu_i-\nu) \\ &\stackrel{+}{=} -\frac{1}{2\rho}\left[\mu_i^T\mu_i - \nu^T\mu_i - \mu_i^T\nu\right]\end{aligned}$$

Then, we rebuild the square and we get :

$$\log q^*(\mu_i) \stackrel{+}{=} -\frac{1}{2\tau_i^*}(\mu_i - \mu_i^*)^T(\mu_i - \mu_i^*)$$

with, $\frac{1}{\tau^*} = \frac{1}{\tau} + \frac{1}{\rho}$ and $\mu_i^* = \frac{\frac{\sum_n q(Z_n=i)X_n}{\tau} + \frac{\nu}{\rho}}{\frac{1}{\tau}+\frac{1}{\rho}}$
Then, we can say that the distribution (over q) of μ_i is a 2D Normal$(\mu_i^*,\tau^*.I)$

The second equation

Let's focus on the second equation, using the expression from the previous question, and only keeping the terms that depend on Z_i, we get :

$$\log q^*(Z_i) \stackrel{+}{=} \log p(Z_i|\pi) + \sum_k \mathbb{I}_{Z_i=k} E_{-Z_i}\left[\log p(X_i|Z_i=k,\mu_k,\tau) + \log p(S_i|\lambda_k,Z_i=k)\right]$$

As $Z_i|\pi$ is a Categorical, $\log p(Z_i|\pi) = \sum_k \mathbb{I}_{Z_i=k} log(\pi_k)$
As $X_i|Z_i=k,\mu,\tau$ follows a 2DNormal$(\mu_k,\tau.I)$

$$\log p(X_i|Z_i=k,\mu_k,\tau) = -\log(\tau^2 2\pi) - \frac{1}{2\tau}(\mu_k - X_i)^T(\mu_k - X_i)$$

$$E_{-Z_i}\left[\log p(X_i|Z_i=k,\mu,\tau)\right] = -\log(\tau^2 2\pi) - \frac{1}{2\tau}(E_{-Z_i}\left[\mu_k^T\mu_k\right] - X_i^T E_{-Z_i}\left[\mu_k\right] - E_{-Z_i}\left[\mu_k^T\right]X_i + X_i^T X_i)$$

$$E_{-Z_i}\left[\log p(X_i|Z_i=k,\mu,\tau)\right] = -\log(\tau^2 2\pi) - \frac{1}{2\tau}(E_{\mu_k}\left[\mu_k^T\mu_k\right] - X_i^T E_{\mu_k}\left[\mu_k\right] - E_{\mu_k}\left[\mu_k^T\right]X_i + X_i^T X_i)$$

Moreover, as we know that μ_k follows a 2DNormal$(\mu_k^*, \tau^*.I)$,
$E_{\mu_k}[\mu_k] = \mu_k^*$ and $E_{\mu_k}\left[\mu_k^T\right] = \mu_k^{*T}$
As $\tau^*.I$ is diagonal matrix, the first dimension and the second aren't correlated.
$\mu_k = \begin{bmatrix}\mu_{k1}\\ \mu_{k2}\end{bmatrix}$
$E_{\mu_k}\left[\mu_k^T\mu_k\right] = E_{\mu_{k1}}[\mu_{k1}^2] + E_{\mu_{k2}}[\mu_{k2}^2] = Var(\mu_{k2}) + E_{\mu_{k2}}[\mu_{k2}^2] + Var(\mu_{k1}) + E_{\mu_{k1}}[\mu_{k1}^2]$
$E_{\mu_k}\left[\mu_k^T\mu_k\right] = 2 * \tau^* + \mu_k^{*T}\mu_k^*$

As $S_i|\lambda, Z_i = k$ follows a Poisson(λ_k),

$$E_{-Z_i}[\log p(S_i|\lambda, Z_i = k)] = S_i E_{-Z_i}[log(\lambda_k)] - \log(S_i!) - E_{-Z_i}[\lambda_k]$$
$$E_{-Z_i}[\log p(S_i|\lambda, Z_i = k)] = S_i E_{\lambda_k}[log(\lambda_k)] - \log(S_i!) - E_{\lambda_k}[\lambda_k]$$

Moreover, we can also calculate the two expectations because the last equation will show that λ_k follows a Gamma distribution with $a_k^* and b_k^*$ as parameter.
$E_{\lambda_k}[log(\lambda_k)] = \psi(a_k^*) - \log(b_k^*)$ and $E_{\lambda_k}[\lambda_k] = \frac{a_k^*}{b_k^*}$
Then, after regrouping all the terms we get :

$$\log q^*(Z_i) \stackrel{+}{=} \sum_k \mathbb{I}_{Z_i=k}\Big[log(\pi_k) + log(\lambda_k) - \log(S_i!) - E_{\lambda_k}[\lambda_k] - \log(\tau^2 2\pi) - \frac{1}{2\tau}(E_{\mu_k}\left[\mu_k^T\mu_k\right] - X_i^T E_{\mu_k}[\mu_k] - E_{\mu_k}\left[\mu_k^T\right]X_i + X_i^T X_i)\Big]$$

Then, we can say that the distribution (over q) of Z_i is Categorical parameterized by π_i^*, where

$$log(\pi_{ki}^*) = log(\pi_k) + log(\lambda_k) - \log(S_i!) - E_{\lambda_k}[\lambda_k] - \log(\tau^2 2\pi) - \frac{1}{2\tau}(E_{\mu_k}\left[\mu_k^T\mu_k\right] - X_i^T E_{\mu_k}[\mu_k] - E_{\mu_k}\left[\mu_k^T\right]X_i + X_i^T X_i)$$

The last equation

Let's focus on the second equation, using the expression from the previous question, and only keeping the terms that depend on λ_i, we get :

$$\log q^*(\lambda_i) \stackrel{+}{=} \log p(\lambda_i|\alpha, \beta) + \sum_n E_{-\lambda_i}[\mathbb{I}_{Z_n=i} \log p(S_i|\lambda_i, Z_n = i)]$$

As $\lambda_i|\alpha, \beta$ follows a gamma distribution with parameters α and β
$\log p(\lambda_i|\alpha, \beta) \stackrel{+}{=} (\alpha - 1)\log(\lambda_i) - \beta\lambda_i$

For the second term, as $S_i|\lambda_i, Z_n = i$ follows a Poisson(λ_i):

$$E_{-\lambda_i}[\mathbb{I}_{Z_n=i} \log p(S_i|\lambda_i, Z_n = i)] \stackrel{+}{=} q(Z_n = i)(S_n \log(\lambda_i) - \lambda_i)$$

Then regrouping the two terms, we get:

$$\log q^*(\lambda_i) \stackrel{+}{=} (a_i^* - 1)\log(\lambda_i) - \lambda_i(b_i^*)$$

With , $a_i^* = \sum_n [q(Z_n = i)S_n] + \alpha$
$b_i^* = \beta + \sum_n q(Z_n = i)$ Then, we can say that the distribution (over q) of λ_i is a gamma distribution parameterized by $a_i^* and b_i^*$

2 Reparametrization and the score function

Question 1.3.4

The expression of the ELBO is the following

$$L(\phi) = \mathbb{E}_{z\sim q}\left[\log p(x|z) + \log p(z) - \log q_\phi(z|x)\right]$$

Using the reparameterization trick, we can express a sample z from a parametric distribution $q_\phi(z)$ as a deterministic function of a random variable ϵ with some fixed distribution and the parameters ϕ of q_ϕ ,i.e.,$z = t(\epsilon, \phi)$. Under such a parameterization of z, we can decompose the total derivative (TD) of the integrand of the ELBO.

$$\begin{aligned}\hat{\nabla}_{TD}(\epsilon, \phi) &= \nabla_\phi \left[\log p(x|z) + \log p(z) - \log q_\phi(z|x)\right] \\ &= \nabla_\phi \left[\log p(z|x) + \log p(x) - \log q_\phi(z|x)\right] \\ &= \nabla_z \left[\log p(z|x) - \log q_\phi(z|x)\right] \nabla_\phi t(\epsilon, \phi) - \nabla_\phi \log q_\phi(z|x),\end{aligned}$$

Where, $\nabla_\phi \log q_\phi(z|x)$ is the score function, and $\nabla_z \left[\log p(z|x) - \log q_\phi(z|x)\right] \nabla_\phi t(\epsilon, \phi)$ is the path derivative.

Question 1.3.5

$$\begin{aligned}\mathbb{E}_{z\sim q}\left[\nabla_\phi \log q_\phi(z|x)\right] &= \int_z \left(\nabla_\phi \log q_\phi(z|x)\right) q_\phi(z|x) dz = \int_z \nabla_\phi q_\phi(z|x) dz \\ &= \nabla_\phi \left(\int_z q_\phi(z|x) dz\right) = \nabla_\phi(1) = 0\end{aligned}$$

The expectation of the score function is zero.

Question 1.3.6

The score function term has a high variance, then by removing this term, (because the expectation of the score function is zero) we can keep an an unbiased estimator of the true gradient which is needed for stochastic gradient descent to converge.

Question 1.3.7

The score function can act in some situations as a control variate.

3 Reparameterization of common distributions

Question 1.4.8

An exponential distribution as tractable inverse CDF. In this case,let ϵ follows a U(0,I), and let $g_\phi(\epsilon, x)$ be the inverse CDF of $q_\phi(z|x)$. The inverse of a exponential distribution is $f(y) = -\frac{\log(1-y)}{\lambda}$
Then $g_\phi(\epsilon, x) = -\frac{\log(1-\epsilon)}{\lambda}$, Where ϵ follows a U(0,I) and λ is the parameter. See the implementation in Q1 of the notebook.

Question 1.4.8

To reparameterize the categorical distribution, we approximating it by the Gumbel-Softmax distribution. First, we sample the Gumbel(0,1) distribution using inverse transform sampling by drawing u follows a Uniform(0, I) and computing $g = -\log(-\log(u))$. Then, we add $\log a$ the parameter of the categorical. Then, we apply the softmax function which then allows for differentiation during training, and using the argmax function for evaluation. See the implementation in Q2 of the notebook.

Appendix

Reparameterization of common distributions

We will work with Torch throughout this notebook.

In [1]:
```
import torch
from torch.distributions import Beta, Exponential, Categorical #, ...  import the distributions you need here
from torch.nn import functional as F
```

A helper function to visualize the generated samples:

In [2]:
```
import matplotlib.pyplot as plt
def compare_samples (samples_1, samples_2, bins=100, range=None):
  fig = plt.figure()
  if range is not None:
    plt.hist(samples_1, bins=bins, range=range)
    plt.hist(samples_2, bins=bins, range=range)
  else:
    plt.hist(samples_1, bins=bins)
    plt.hist(samples_2, bins=bins)
  plt.xlabel('value')
  plt.ylabel('number of samples')
  plt.legend(['direct','via reparameterization'])
  plt.show()
```

Q1. Exponential Distribution

Below write a function that generates N samples from $Exp(\lambda)$.

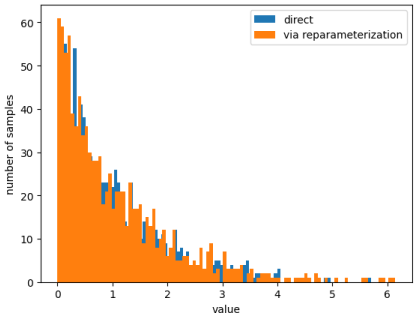
In [3]:
```
def exp_sampler(l, N):
  # insert your code
    rates = torch.full((N,), float(l) )
    samples = Exponential(rates).sample()
    return samples # should be N-by-1
```

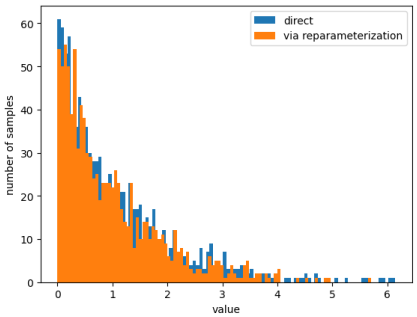
Now, implement the reparameterization trick:

In [4]:
```
def exp_reparametrize(l,N):
  # this function should return N samples via reparametrization,
  # insert your code
    e = torch.rand(N)
    samples = -torch.log(1-e)/l
    return samples
```

Generate samples for $\lambda = 1$ and compare:

In [5]:
```
l = 1    #lambda
N = 1000
direct_samples = exp_sampler(l, N)
reparametrized_samples = exp_reparametrize(l, N)
compare_samples(direct_samples, reparametrized_samples)
compare_samples(reparametrized_samples, direct_samples)
```





Q2. Categorical Distribution

Below write a function that generates N samples from Categorical (**a**), where **a** = $[a_0, a_1, a_2, a_3]$.

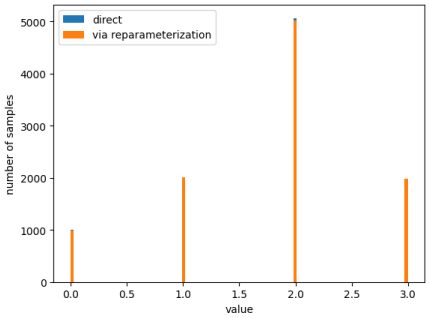
In [6]:
```
def categorical_sampler(a, N):
  # insert your code
    rates = a.repeat(N,1)
    samples = Categorical(rates).sample()
    return samples  # should be N-by-1
```

Now write a function that generates samples from Categorical (**a**) via reparameterization:

In [7]:
```
# Hint: approximate the Categorical distribution with the Gumbel-Softmax distribution
def categorical_reparametrize(a, N, temp=0.1, eps=1e-20):  # temp and eps are hyperparameters for Gumbel-Softmax
  # insert your code
    e = torch.rand(N, a.size(0))
    g = - torch.log(-torch.log(e+eps)+eps)

    logits = (torch.log(a) + g) / temp
    softmax_output = torch.nn.functional.softmax(logits, dim=1)

    # Apply the reparameterization trick
    samples = torch.argmax(softmax_output, dim=1)

    return samples # make sure that your implementation allows the gradient to backpropagate
```

Generate samples when $a = [0.1, 0.2, 0.5, 0.2]$ and visualize them:

In [8]:
```
a = torch.tensor([0.1,0.2,0.5,0.2])
N = 1000
direct_samples = categorical_sampler(a, N)
reparametrized_samples = categorical_reparametrize(a, N, temp=0.1, eps=1e-20)
compare_samples(direct_samples, reparametrized_samples)
```

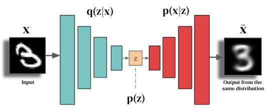

direct
via reparameterization
number of samples
value
0
1000
2000
3000
4000
5000
0.0
0.5
1.0
1.5
2.0
2.5
3.0

In [ ]:

VAE for image generation

Consider VAE model from *Auto-Encoding Variational Bayes (2014, D.P. Kingma et. al.)*.

We will implement a VAE model using Torch and apply it to the MNIST dataset.



Generative model: We model each pixel value $\in \{0, 1\}$ as a sample drawn from a Bernoulli distribution. Through a decoder, the latent random variable z_n associated with an image n is mapped to the success parameters of the Bernoulli distributions associated with the pixels of that image. Our generative model is described as follows:

$z_n \sim N(0, I)$

$\theta_n = g(z_n)$

$x_n \sim Bern(\theta_n)$

where g is the decoder. We choose the prior on z_n to be the standard multivariate normal distribution, for computational convenience.

Inference model: We infer the posterior distribution of z_n via variational inference. The variational distribution $q(z_n|x_n)$ is chosen to be multivariate Gaussian with a diagonal covariance matrix. The mean and covariance of this distribution are obtained by applying an encoder to x_n.

$q(z_n|x_n) \sim q(\mu_n, \sigma_n^2)$

where $\mu_n, \sigma_n^2 = f(x_n)$ and f is the encoder.

Implementation: Let's start with importing Torch and other necessary libraries:

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

from tqdm import tqdm
from torchvision.utils import save_image, make_grid
```

Step1: Model Hyperparameters

In [2]:
```
dataset_path = '~/datasets'

batch_size = 100

# Dimensions of the input, the hidden layer, and the latent space.
x_dim  = 784
hidden_dim = 400
latent_dim = 200

# Learning rate
lr = 1e-3

# Number of epoch
epochs = 15 # can try something greater if you are not satisfied with the results
```

Step2: Load Dataset

In [3]:
```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

mnist_transform = transforms.Compose([
        transforms.ToTensor(),
])


train_dataset = MNIST(dataset_path, transform=mnist_transform, train=True, download=True)
test_dataset  = MNIST(dataset_path, transform=mnist_transform, train=False, download=True)

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

Step3: Define the model

In [4]:
```
class Encoder(nn.Module):
    # encoder outputs the parameters of variational distribution "q"
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(Encoder, self).__init__()

        self.FC_enc1 = nn.Linear(input_dim, hidden_dim) # FC stands for a fully connected layer
        self.FC_enc2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_mean = nn.Linear(hidden_dim, latent_dim)
        self.FC_var  = nn.Linear (hidden_dim, latent_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2) # will use this to add non-linearity to our model

        self.training = True

    def forward(self, x):
        h_1     = self.LeakyReLU(self.FC_enc1(x))
        h_2     = self.LeakyReLU(self.FC_enc2(h_1))
        mean    = self.FC_mean(h_2)  # mean
        log_var = self.FC_var(h_2)   # log of variance

        return mean, log_var
```

In [5]:
```
class Decoder(nn.Module):
    # decoder generates the success parameter of each pixel
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.FC_dec1   = nn.Linear(latent_dim, hidden_dim)
        self.FC_dec2   = nn.Linear(hidden_dim, hidden_dim)
        self.FC_output = nn.Linear(hidden_dim, output_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2) # again for non-linearity

    def forward(self, z):
        h_out_1  = self.LeakyReLU(self.FC_dec1(z))
        h_out_2  = self.LeakyReLU(self.FC_dec2(h_out_1))

        theta = torch.sigmoid(self.FC_output(h_out_2))
        return theta
```

Q3.1 (2 points) Below implement the reparameterization function.

In [6]:
```
class Model(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder

    def reparameterization(self, mean, var):
        # insert your code here
        # dim of mean and var is (latent_dim)

        std = torch.sqrt(var)

    # Sample epsilon from N(0, 1)
        eps = torch.randn_like(std)

    # Reparameterize to get z
        z = mean + eps * std
        return z



    def forward(self, x):
        mean, log_var = self.Encoder(x)
        z = self.reparameterization(mean, torch.exp(log_var)) # takes exponential function (log var -> var)
        theta = self.Decoder(z)

        return theta, mean, log_var
```

Step4: Model initialization

In [7]:
```
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim, latent_dim=latent_dim)
decoder = Decoder(latent_dim=latent_dim, hidden_dim = hidden_dim, output_dim = x_dim)

model = Model(Encoder=encoder, Decoder=decoder)
```

Step5: Loss function and optimizer

Our objective function is ELBO: $E_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right]$

- **Q5.1 (1 point)** Show that ELBO can be rewritten as :

  $E_{q(z|x)}\big(\log p(x|z)\big) - D_{KL}\big(q(z|x)||p(z)\big)$

5.1 Your answer

. $ELBO = E_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right]$

. $ELBO = E_{q(z|x)}\left[\log p(x,z) - \log q(z|x)\right] = E_{q(z|x)}\left[\log p(x|z) + \log p(z) - \log q(z|x)\right]$

. $ELBO = E_{q(z|x)}\left[\log p(x|z)\right] + E_{q(z|x)}\left[\log p(z) - \log q(z|x)\right]$

. Moreover, $D_{KL}\big(q(z|x)||p(z)\big) = E_{q(z|x)}[logq(z \mid x) - logp(z)]$

.Then, $ELBO = E_{q(z|x)}\big(\log p(x|z)\big) - D_{KL}\big(q(z|x)||p(z)\big)$

Consider the first term: $E_{q(z|x)}\big(\log p(x|z)\big)$

$E_{q(z|x)}\big(\log p(x|z)\big) = \int q(z|x) \log p(x|z) dz$

We can approximate this integral by Monte Carlo integration as following:

$\approx \frac{1}{L}\sum_{l=1}^{L} \log p(x|z_l)$, where $z_l \sim q(z|x)$.

Now we can compute this term using the analytic expression for $p(x|z)$. (Remember we model each pixel as a sample drawn from a Bernoulli distribution).

Consider the second term: $-D_{KL}\big(q(z|x)||p(z)\big)$

- **Q5.2 (2 points)** Kullback–Leibler divergence can be computed using the closed-form analytic expression when both the variational and the prior distributions are Gaussian. Write down this KL divergence in terms of the parameters of the prior and the variational distributions. Your solution should consider a generic case where the latent space is K-dimensional.

5.2 Your answer

. $D_{KL}\big(q(z|x)||p(z)\big) = \frac{1}{2}\left[\log \frac{|\Sigma_2|}{|\Sigma_1|} - K + \mathrm{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2 - \mu_1)^T \Sigma_2^{-1}(\mu_2 - \mu_1)\right]$

. Where, q(.|x) and p are the pdfs of normal distributions with means μ_1 and μ_2 and variances Σ_1 and Σ_2

. In our case $\mu_2 = 0$ and $\Sigma_2 = I$, then $D_{KL}\big(q(z|x)||p(z)\big) = \frac{1}{2}\left[\mathrm{tr}\{\Sigma_1\} - \log|\Sigma_1| - K + \mu_1^T \mu_1\right]$

Q5.3 (5 points) Now use your findings to implement the loss function, which is the negative of ELBO:

In [8]:
```
from torch.optim import Adam

def loss_function(x, theta, mean, log_var): # should return the loss function (- ELBO)
    # insert your code here



    KLD = -0.5 * torch.sum(1 + log_var - mean.pow(2) - log_var.exp())

    elbo = torch.sum(x*torch.log(theta+1e-8)+(1-x)*torch.log(1-theta+1e-8)) - KLD

    return -elbo

# optimizer
optimizer = Adam(model.parameters(), lr=lr)
```

Step6: Train the model

In [9]:
```
print("Start training VAE...")
model.train()

for epoch in range(epochs):
    overall_loss = 0
    for batch_idx, (x, _) in enumerate(train_loader):
        x = x.view(batch_size, x_dim)
        x = torch.round(x)

        optimizer.zero_grad()

        theta, mean, log_var = model(x)
        loss = loss_function(x, theta, mean, log_var)

        overall_loss += loss.item()

        loss.backward()
        optimizer.step()

    print("\tEpoch", epoch + 1, "complete!", "\tAverage Loss: ", overall_loss / (batch_idx*batch_size))

print("Finish!!")
```

```
Start training VAE...
        Epoch 1 complete!       Average Loss:  170.70426180678734
        Epoch 2 complete!       Average Loss:  118.6383636627452
        Epoch 3 complete!       Average Loss:  103.6609433365505
        Epoch 4 complete!       Average Loss:  98.06722452459829
        Epoch 5 complete!       Average Loss:  94.35654803904946
        Epoch 6 complete!       Average Loss:  91.63974482209933
        Epoch 7 complete!       Average Loss:  89.88429380999582
        Epoch 8 complete!       Average Loss:  88.6002942811587
        Epoch 9 complete!       Average Loss:  87.52469352403746
        Epoch 10 complete!      Average Loss:  86.58153398567926
        Epoch 11 complete!      Average Loss:  85.73857355846985
        Epoch 12 complete!      Average Loss:  84.97660721971515
        Epoch 13 complete!      Average Loss:  84.40863705948716
        Epoch 14 complete!      Average Loss:  83.88314405030519
        Epoch 15 complete!      Average Loss:  83.52071974612636
Finish!!
```

Step7: Generate images from test dataset

With our model trained, now we can start generating images.

First, we will generate images from the latent representations of test data.

Basically, we will sample z from $q(z|x)$ and give it to the generative model (i.e., decoder) $p(x|z)$. The output of the decoder will be displayed as the generated image.

Q7.1 (2 points) Write a code to get the reconstructions of test data, and then display them using the show_image function

In [10]:
```
model.eval()
# below we get decoder outputs for test data
with torch.no_grad():
    for batch_idx, (x, _) in enumerate(tqdm(test_loader)):
        x = x.view(batch_size, x_dim)
        # insert your code below to generate theta from x
        #
        #
        theta, _ , _ = model(x)
```

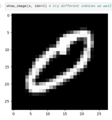
```
100%|████████████████████████████████████████| 100/100 [00:02<00:00, 37.41it/s]
```

A helper function to display images:

In [11]:
```
import matplotlib.pyplot as plt
def show_image(theta, idx):
    x_hat = theta.view(batch_size, 28, 28)
    #x_hat = Bernoulli(x_hat).sample() # sample pixel values (you can also try this, and observe how the generated images look)
    fig = plt.figure()
    plt.imshow(x_hat[idx].cpu().numpy(), cmap='gray')
```

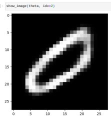
First display an image from the test dataset,

In [27]:
```
show_image(x, idx=2) # try different indices as well
```



Now display its reconstruction and compare:

In [28]:
```
show_image(theta, idx=2)
```



Step8: Generate images from noise

In the previous step, we sampled latent vector z from $q(z|x)$. However, we know that the KL term in our loss function enforced $q(z|x)$ to be close to $N(0, I)$. Therefore, we can sample z directly from noise $N(0, I)$, and pass it to the decoder $p(x|z)$.

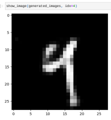
Q8.1 (3 points) Create images from noise and display.

In [31]:
```
model.eval()
with torch.no_grad():
    eps = torch.randn((batch_size,latent_dim))
    generated_images = model.Decoder(eps)
    print(generated_images.size())
    # insert your code here to create images from noise (it is enough to create theta value for each pixel)
    #
    #
    # generated_images = ....  # should be a matrix ( batch_size-by-x_dim )
```

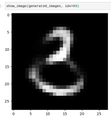
```
torch.Size([100, 784])
```

Display a couple of generated images:

In [35]:
```
show_image(generated_images, idx=4)
```



In [37]:
```
show_image(generated_images, idx=80)
```



In [38]:
```
show_image(generated_images, idx=88)
```