DD2434/FDD3434 Machine Learning, Advanced Course

Assignment 1B

Abdessamad BADAOUI

1.1 CAVI for Earth quakes

1.1.1 Question 1.1.1

Let's begin by formulating the probability distributions for the various variables in our problem:

- $Z_n|\pi \sim Cat(\pi)$ which means that $\mathbb{P}(Z_n = k) = \pi_k$
- $S_n|Z_n = k, \lambda_k \sim Pois(\lambda_k) = \frac{\lambda_k^{S_n}}{S_n!}\exp(-\lambda_k)$
- $X_n|Z_n = k, \mu_k, \tau \sim \mathcal{N}(\mu_k, \frac{1}{\tau}I) = \frac{\tau}{2\pi}\exp(-\frac{\tau}{2}\sum_{i=0}^{1}(x_{ni} - \mu_{ki})^2)$
- $\mu_k|\upsilon_0, \upsilon_1, \rho \sim \mathcal{N}(\upsilon, \frac{1}{\rho}) = \frac{\rho}{2\pi}\exp(-\frac{\rho}{2}\sum_{i=0}^{1}(\mu_{ki} - \upsilon_i)^2)$
- $\lambda_k|\alpha, \beta \sim Gamma(\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda_k^{\alpha-1}e^{-\beta\lambda_k}$

The DGM of the model is described in Figure 1, where red nodes are constants and blue ones are observed :

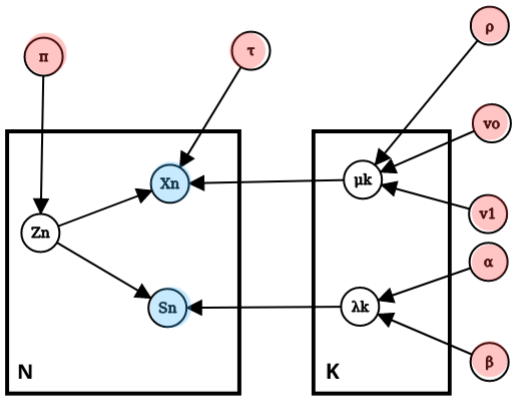


Figure 1: DGM for our model. The nodes in red represent constants, while those in blue represent observed variables.

1.1.2 Question 1.1.2

Let's simplify the expression of $\log p(X, S, Z, \lambda, \mu|\pi, \tau, \alpha, \beta, \upsilon, \rho)$

We first have : $p(X, S, Z, \lambda, \mu|\pi, \tau, \alpha, \beta, \upsilon, \rho) = p(X, S, Z, \lambda, \mu|\lambda, \mu)p(\lambda)p(\mu)$. We will omit writing the constant variables.

Let's first make the following notation :

- $Z_n = k \Leftrightarrow Z_{nk} = 1, Z_{nk'} = 0 \qquad \forall k \neq k'$

We can then express $p(X, Z, S|\lambda, \mu)$:

$$\begin{aligned} p(X, Z, S|\lambda, \mu) &= \prod_{n=1}^{N} p(X_n, S_n|Z_n, \lambda, \mu)p(Z_n|\pi) \\ &= \prod_{n,k}[p(X_n|\mu_k, \tau)p(S_n|\lambda_k)p(Z_{nk} = 1|\pi)]^{Z_{nk}} \\ &= \prod_{n,k}[\frac{\tau}{2\pi}\exp(-\frac{\tau}{2}\sum_{i=0}^{1}(x_{ni} - \mu_{ki})^2).\frac{\lambda_k^{S_n}}{S_n!}e^{-\lambda_k}\pi_k]^{Z_{nk}} \end{aligned} \tag{1}$$

We have $p(\mu)$ is given by:

$$p(\mu) = \prod_{k=1}^{K} p(\mu_k) = \prod_{k=1}^{K} \frac{\rho}{2\pi}\exp(-\frac{\rho}{2}\sum_{i=0}^{1}(\mu_{ki} - \upsilon_i)^2) \tag{2}$$

and $p(\lambda)$ is given by:

$$p(\lambda) = \prod_{k=1}^{K} p(\lambda_k) = \Big(\frac{\beta^\alpha}{\Gamma(\alpha)}\Big)^K \prod_{k=1}^{K} \lambda_k^{\alpha-1}e^{-\beta\lambda_k} \tag{3}$$

Now, we can write $\log p(X, S, Z, \lambda, \mu|\pi, \tau, \alpha, \beta, \upsilon, \rho)$ as :

$$\begin{aligned} \log p(X, S, Z, \lambda, \mu|\pi, \tau, \alpha, \beta, \upsilon, \rho) &= \log p(X, Z, S|\lambda, \mu) + logp(\mu) + logp(\lambda) \\ &= \sum_{n,k} Z_{nk}\big[(-\frac{\tau}{2}\sum_{i=0}^{1}(x_{ni} - \mu_{ki})^2) + log(\frac{\tau}{2\pi}) + log(\frac{\lambda_k^{S_n}}{S_n!} - \lambda_k + log(\pi_k)\big] \\ &+ \sum_{k=1}^{K}[\log(\frac{\rho}{2\pi}) - \frac{\rho}{2}\sum_{i=0}^{1}(\mu_{ki} - \upsilon_i)^2] \\ &+ K[\alpha\log(\beta) - \log(\Gamma(\alpha)] + \sum_{k=1}^{K}[(\alpha - 1)log(\lambda_k) - \beta\lambda_k] \end{aligned} \tag{4}$$

The first part of that expression can be written as follows :

$$\sum_{n,k} Z_{nk}\big[(-\frac{\tau}{2}\sum_{i=0}^{1}(x_{ni} - \mu_{ki})^2) + log(\frac{\tau}{2\pi}) + log(\frac{\lambda_k^{S_n}}{S_n!} - \lambda_k + log(\pi_k)\big] = \sum_{n,k} Z_{nk}\log(u_{nk}) \tag{5}$$

which represents the logarithm of a categorical distribution with parameter u (we will come back later on how to calculate $\log(u_{nk})$). Then the second term represents the sum of K logarithms of the normal distribution of parameters υ and ρ, and finally the sum of K logarithms of Gamma distribution with parameters α and β.

1.1.3 Question 1.1.3

We have the following variational inference assumption : $q(Z,\mu,\lambda) = \prod_{n=1}^{N} q(Z_n) \prod_{k=1}^{K} q(\mu_k)q(\lambda_k)$

Let's derive the CAVI updates equations for the different variables :

$$\begin{aligned}
\log q^*(Z_n) &= \mathbb{E}_{-Z_n,\mu,\lambda}\big[\log p(X,S,Z,\lambda,\mu)\big] \\
&\stackrel{+}{=} \mathbb{E}_{-Z_n,\mu,\lambda}\big[log(X,S,Z|\lambda,\mu)\big] \\
&\stackrel{+}{=} \mathbb{E}_{-Z_n,\mu,\lambda}\big[\sum_{n',k} Z_{n'k}\log(u_{n'k})\big] \\
&\stackrel{+}{=} \mathbb{E}_{-Z_n,\mu,\lambda}\big[\sum_{k} Z_{nk}\log(u_{nk})\big] \\
&\stackrel{+}{=} \sum_{k} Z_{nk}\mathbb{E}_{\mu,\lambda}\big[\log(u_{nk})\big] \\
&\stackrel{+}{=} \sum_{k} Z_{nk}\log(u'_{nk}) \\
\text{which leads to :}\quad q^*(Z_n) &\propto \prod_{k=1}^{K} {u'_{kn}}^{Z_{nk}} \\
&\propto \big[\frac{u'_{kn}}{\sum_k u'_{kn}}\big]^{Z_{nk}} \sim Cat(U''_n)
\end{aligned} \tag{6}$$

where $U''_n = \Big(\frac{u'_{nk}}{\sum_k u'_{nk}}\Big)_{1\le k\le K}$, and $\log(u'_{nk})$ is given by :

$$\begin{aligned}
\log(u'_{nk}) &= \mathbb{E}_{\mu,\lambda}\big[\log(u_{nk})\big] \\
&= \mathbb{E}_{\mu,\lambda}\big[-\frac{\tau}{2}\sum_{i=0}^{1}(x_{ni}-\mu_{ki})^2\big] + log(\frac{\tau}{2\pi}) + S_n\mathbb{E}_\lambda[\log\lambda_k] - log(S_n!) - \mathbb{E}_\lambda[\lambda_k] + \log(\pi_k)
\end{aligned} \tag{7}$$

The different terms in that expression can be calculated as follows :

- $\mathbb{E}_{\mu,\lambda}\big[-\frac{\tau}{2}\sum_{i=0}^{1}(x_{ni}-\mu_{ki})^2\big] = -\frac{\tau}{2}\sum_{i=0}^{1}\big[x_{ni}^2 - 2x_{ni}\upsilon^*_{ki} + \frac{1}{\rho^*} + \upsilon^{*2}_{ki}\big]$
- $\mathbb{E}_\lambda[\log(\lambda_k)] = -\log(\beta^*_k) + \psi(\alpha^*_k)$ where ψ is the digamma function.
- $\mathbb{E}_\lambda[\lambda_k] = \frac{\alpha^*_k}{\beta^*_k}$

where $\alpha^*_k, \beta^*_k, \rho^*$ and υ^*_k are the posteriors (we will calculate them after).
Once we calculate this quantity, we can simply exponentiate it to obtain the values of u'_{nk}.

Finally :

$$\boxed{\text{q}^*(Z_n) \sim Cat(U''_n) \quad \text{where } U''_n = \Big(\frac{u'_{nk}}{\sum_k u'_{nk}}\Big)_{1\le k\le K}}$$

Let's derive now the CAVI update equation for μ_k:

$$
\begin{aligned}
\log q^*(\mu_k) &= \mathbb{E}_{-\mu_k,Z,\lambda}\big[\log p(X,S,Z,\lambda,\mu)\big] \\
&\stackrel{+}{=} \mathbb{E}_{-\mu_k,Z,\lambda}\big[log(X,Z,S|\lambda,\mu)\big] + \log p(\mu_k) \\
&\stackrel{+}{=} \mathbb{E}_{-\mu_k,Z,\lambda}\big[\sum_{n=1}^{N} -\frac{\tau Z_{nk}}{2}\sum_{i=0}^{1}(x_{ni}-\mu_{ki})^2\big] - \frac{\rho}{2}\sum_{i=0}^{1}(\mu_{ki}-\upsilon_i)^2 \\
&\stackrel{+}{=} -\sum_{n=1}^{N}[\mathbb{E}_Z[Z_{nk}].\frac{\tau}{2}\sum_{i=1}^{1}(x_{ni}-\mu_{ki})^2] - \frac{\rho}{2}\sum_{i=0}^{1}(\mu_{ki}-\upsilon_i)^2 \\
&\stackrel{+}{=} -\frac{\tau}{2}\sum_{n=1}^{N}\sum_{i=0}^{1} q(Z_n=k)[x_{ni}^2 - 2x_{ni}\mu_{ki} + \mu_{ki}^2] - \frac{\rho}{2}\sum_{i=0}^{1}[\mu_{ki}^2 - 2\mu_{ki}\upsilon_i + \upsilon_i^2] \\
&\stackrel{+}{=} \Big[-\frac{\rho}{2} - \frac{\tau}{2}\sum_n q(Z_n=k)\Big]\sum_{i=0}^{1}\mu_{ki}^2 + \Big[\rho\upsilon_0 + \tau\sum_n q(Z_n=k)x_{n0}\Big]\mu_{k0} - \Big[\rho\upsilon_1 + \tau\sum_n q(Z_n=k)x_{n1}\Big]\mu_{k1} \\
&\stackrel{+}{=} \Big[-\frac{\rho}{2} - \frac{\tau}{2}\sum_n q(Z_n=k)\Big]\Big(\sum_{i=0}^{1}\mu_{ki}^2 + \frac{\rho\upsilon_0 + \tau\sum_n q(Z_n=k)x_{n0}}{\rho + \tau\sum_{n=1} q(Z_n=k)}\mu_{k0} + \frac{\rho\upsilon_1 + \tau\sum_n q(Z_n=k)x_{n1}}{\rho + \tau\sum_{n=1} q(Z_n=k)}\mu_{k1}\Big)
\end{aligned}
\tag{8}
$$

Completing the square over μ_{k0} and μ_{k1} we see that $\boxed{\mathrm{q}^*(\mu_k) \sim \mathcal{N}(\upsilon_k^*, \frac{1}{\rho_k^*}I)}$ where :

- $\upsilon_{k0}^* = \dfrac{\rho\upsilon_0 + \tau\sum_n q(Z_n=k)x_{n0}}{\rho + \tau\sum_{n=1} q(Z_n=k)}$
- $\upsilon_{k1}^* = \dfrac{\rho\upsilon_1 + \tau\sum_n q(Z_n=k)x_{n1}}{\rho + \tau\sum_{n=1} q(Z_n=k)}$
- $\rho_k^* = \rho + \tau\sum_{n=1}^{N} q(Z_n=k)$

Let's derive now the CAVI update equation for λ_k:

$$
\begin{aligned}
\log q^*(\lambda_k) &= \mathbb{E}_{-\lambda_k,Z,\mu}\big[\log p(X,S,Z,\lambda,\mu)\big] \\
&\stackrel{+}{=} \mathbb{E}_{-\lambda_k,Z,\mu}\big[\log p(X,Z,S|\lambda,\mu)\big] + \log p(\lambda_k) \\
&\stackrel{+}{=} \mathbb{E}_{-\lambda_k,Z,\mu}\big[\sum_{n=1}^{N} Z_{nk}\big(\log\big(\frac{\lambda_k^{S_n}}{S_n!}\big) - \lambda_k\big)\big] + \alpha\log(\beta) - \log(\Gamma(\alpha)) + (\alpha-1)\log(\lambda_k) - \beta\lambda_k \\
&\stackrel{+}{=} \sum_{n=1}^{N} q(Z_n=k)\big[S_n\log(\lambda_k) - \log(S_n!) - \lambda_k\big] + \alpha\log(\beta) - \log(\Gamma(\alpha)) + (\alpha-1)\log(\lambda_k) - \beta\lambda_k \\
&\stackrel{+}{=} \Big[\sum_{n=1}^{N} q(Z_n=k)S_n + \alpha - 1\Big]\log(\lambda_k) - \Big[\beta + \sum_{n=1}^{N} q(Z_n=k)\Big]\lambda_k
\end{aligned}
\tag{9}
$$

Completing the missing constants we see that $\boxed{\mathrm{q}^*(\lambda_k) \sim Gamma(\alpha_k^*, \beta_k^*)}$ where :

- $\alpha_k^* = \alpha + \sum_{n=1}^{N} q(Z_n=k)S_n$
- $\beta_k^* = \beta + \sum_{n=1}^{N} q(Z_n=k)$

2 VAE image generation

Answers are presented in the Notebook. (See the Appendix)

3 Reparametization and the score function

3.0.1 Question 1.3.4

Let's decompose the gradient of the ELBO using the reparameterization trick to a form where the score function appears as a term :

$$\begin{aligned}
ELBO &= \mathbb{E}_{z\sim q}\Big[\log(p(x|z)) + \log(p(z)) - \log(q_\phi(z|x))\Big] \\
\nabla_\phi ELBO &= \nabla_\phi \mathbb{E}_{z\sim q}\Big[\log p(x|z) + \log p(z) - \log q_\phi(z|x)\Big] \\
&= \nabla_\phi \mathbb{E}_{\mathcal{N}(\epsilon|0,I)}\Big[\log p(x|t(\epsilon,\phi)) + \log p(t(\epsilon,\phi)) - \log q_\phi(t(\epsilon,\phi)|x)\Big] \\
&= \nabla_\phi \int \mathcal{N}(\epsilon|0,I)\Big[\log p(x|t(\epsilon,\phi)) + \log p(t(\epsilon,\phi)) - \log q_\phi(t(\epsilon,\phi)|x)\Big]d\epsilon \\
&= \mathbb{E}_{\mathcal{N}(0,I)}\Big[\nabla_\phi\Big[\log p(x|t(\epsilon,\phi)) + \log p(t(\epsilon,\phi)) - \log q_\phi(t(\epsilon,\phi)|x)\Big]\Big] \\
&= \mathbb{E}_{z\sim q}\Big[\nabla_\phi\Big[\log p(x|z) + \log p(z) - \log q_\phi(z|x)\Big]\Big] \\
&= \mathbb{E}_{z\sim q}\Big[\hat{\nabla}_{TD}(\epsilon,\phi)\Big] \\
&= \mathbb{E}_{z\sim q}\Big[\nabla_z\Big[\log p(z|x) - \log q_\phi(z|x)\Big]\nabla_\phi t(\epsilon,\phi)\Big] - \mathbb{E}_{z\sim q}\Big[\nabla_\phi \log q_\phi(z|x)\Big]
\end{aligned} \tag{10}$$

Using the reparameterization trick and the equations about the total derivative (TD), we get the expression above containing the expectation of the score function $\nabla_\phi \log q_\phi(z|x)$.

3.0.2 Question 1.3.5

Let's show that the expectation of the score function is zero :

$$\begin{aligned}
\mathbb{E}_{z\sim q}\Big[\nabla_\phi \log q_\phi(z|x)\Big] &= \int_z q_\phi(z|x)\frac{\nabla_\phi q_\phi(z|x)}{q_\phi(z|x)}dz \\
&= \nabla_\phi \int_z q_\phi(z|x)dz = \nabla_\phi(1) = 0
\end{aligned} \tag{11}$$

3.0.3 Question 1.3.6

The proposed solution is to simply remove that high-variance score function term since its expectation is null. By doing so, we maintain an unbiased estimate of the true gradient.

3.0.4 Question 1.3.7

In such scenarios, the score function serves as a **control variate** by introducing a zero-expectation term into an estimator to reduce the variance.

4 Reparameterization of common distributions

4.0.1 Exponential Distribution

Let's show how to reparameterize the exponential distribution with parameter λ :

We have the CDF of the exponential distribution is given by : $y = F(x) = 1 - e^{-\lambda x}$, so :

$$\begin{aligned} y = F(x) &= 1 - e^{-\lambda x} \\ 1 - y &= e^{-\lambda x} \\ \lambda x &= -\ln(1-y) \\ x &= -\frac{1}{\lambda}\ln(1-y) \end{aligned} \tag{12}$$

Then we can conclude that $\boxed{\text{X} = \text{-}\ \frac{1}{\lambda}\ln(1-U) \sim Exp(\lambda) \text{ where } U \sim Uniform([0,1])}$

The plot in Figure 2 shows the result obtained in the notebook :

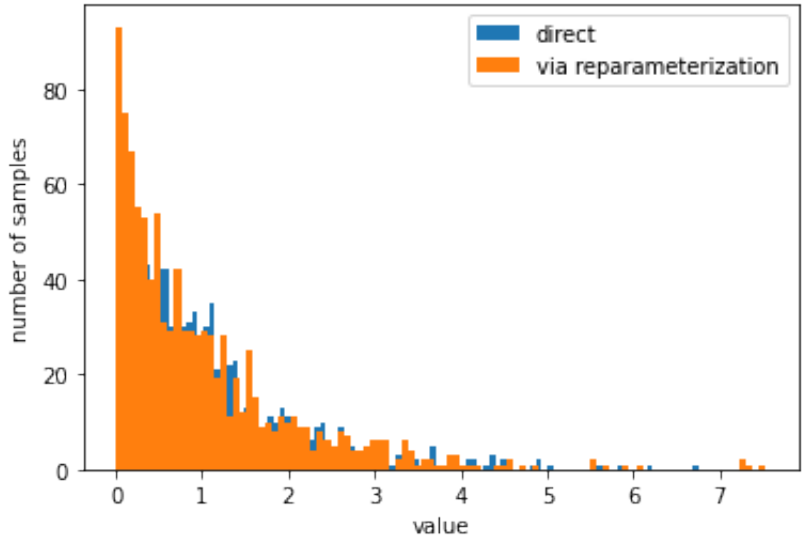


Figure 2: Direct samples vs Via reparameterization for the exponential distribution with $\lambda = 1$

4.0.2 Categorical Distribution

Let's suppose that we have $z \sim Cat(\pi_1, \ldots, \pi_K)$, a random variable following the categorical distribution. According to the paper, we can draw samples from z, using the Gumbel-softmax distribution:

$$z = argmax_{k \in \{1,..,K\}} \Big(softmax\big((g_k + log(\pi_k))/\tau\big) \Big) \tag{13}$$

where $g1, \ldots, g_K \sim Gumbel(0,1)$

The softmax is used to ensure having a differentiable function, allowing the gradient to backpropagate. The softmax produces probabilities, and argmax aids in evaluation by selecting the value with the highest probability.

Let's show how to reparameterize the $Gumbel(0,1)$ distribution:

We have the CDF of $Gumbel(0,1)$ distribution is given by : $y = F(x) = e^{-e^{-x}}$, so :

$$\begin{aligned} y = F(x) &= e^{-e^{-x}} \\ -e^{-x} &= \ln(y) \\ -x &= \ln(-\ln(y)) \\ x &= -\ln(-\ln(y)) \end{aligned} \tag{14}$$

Then we can conclude that $\boxed{\text{X = - ln}(-\ln(U)) \sim Gumbel(0,1) \text{ where } U \sim Uniform([0,1])}$

We then have a way to sample from a categorical distribution Z $\sim$Cat$(\pi_1, \ldots, \pi_K)$ given by:

$$\boxed{\text{Z} = \text{argmax}_{k \in \{1,..,K\}} \Big(softmax\big((-\ln(-\ln(U_k)) + log(\pi_k))/\tau\big) \Big) \text{ where } U_k \sim Uniform([0,1]) \quad \forall k \in 1, \ldots, K}$$

The plot in Figure 3 shows the result obtained in the notebook :

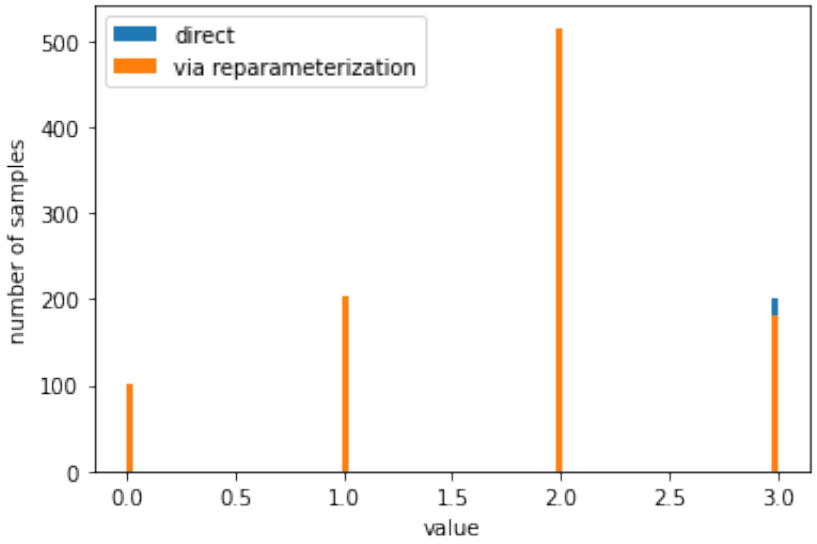


Figure 3: Direct samples vs Via reparameterization for the categorical distribution with parameters $a = [0.1, 0.2, 0.5, 0.2]$

1B-VAE-v2

December 7, 2023

1 *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:

[1]:
```
import torch
import torch.nn as nn

import numpy as np

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

1.0.1 *Step1: Model Hyperparameters*

[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 = 20 # can try something greater if you are not satisfied with the results
```

1.0.2 *Step2: Load Dataset*

[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)
```

1.0.3 *Step3: Define the model*

[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
```

[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.

[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
        std = torch.sqrt(var)
        eps = torch.randn_like(std)
        z = mean + std*eps
```

```
        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
```

1.0.4 *Step4: Model initialization*

[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)
```

1.0.5 *Step5: Loss function and optimizer*

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

- **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

$$
\begin{aligned}
\mathbb{E}_{q(z|x)}\Big[\log \frac{p(x,z)}{q(z|x)}\Big] &= \mathbb{E}_{q(z|x)}\Big[\log \frac{p(x|z)p(z)}{q(z|x)}\Big] \\
&= \mathbb{E}_{q(z|x)}\Big[\log p(x|z)\Big] + \mathbb{E}_{q(z|x)}\Big[\log \frac{p(z)}{q(z|x)}\Big] \\
&= \mathbb{E}_{q(z|x)}\Big[\log p(x|z)\Big] - D_{KL}\big(q(z|x)||p(z)\big)
\end{aligned} \tag{1}
$$

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

We have : $q(z|x) \sim \mathcal{N}(\mu, \Sigma)$ $p(z) \sim \mathcal{N}(0, I)$

where μ is of dimension K (dimension of the latent space) and Σ is the diagonal covariance matrix of size K*K

$$\begin{aligned} D_{KL}(q(z|x)||p(z)) &= \frac{1}{2}\Big[\log(\frac{1}{|\Sigma|}) - K + \mu^T I^{-1}\mu + tr\{I^{-1}.\Sigma\}\Big] \\ &= \frac{1}{2}\Big[-\log(\prod_{k=1}^{K}\sigma_k^2) - K + \mu^T\mu + \sum_{k=1}^{K}\sigma_k^2\Big] \\ -D_{KL}(q(z|x)||p(z)) &= \frac{1}{2}\sum_{k=1}^{K}(1 + \log((\sigma_k)^2) - (\mu_k)^2 - (\sigma_k)^2) \end{aligned} \tag{2}$$

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

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

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

    loss = 0
    var = torch.exp(log_var)
    monte_carlo_estimate = torch.zeros(len(theta))

    for l in range(20):
        z_l = model.reparameterization(mean,var)
        theta = decoder.forward(z_l)

        # print("theta =",theta)

        # p_x_given_z_l = torch.prod(theta,dim=1)
        # print("p_x_given_z_l =",p_x_given_z_l)
        # monte_carlo_estimate += torch.log(p_x_given_z_l) / 10
        eps = 0.00001
        log_pixel_probs = x * torch.log(theta + eps) + (1 - x) * torch.log(1 - theta + eps)
        monte_carlo_estimate += torch.sum(log_pixel_probs,dim=1) / 20

        # print("monte_carlo_estimate = ",monte_carlo_estimate)
```

```
    for row in range(len(theta)):
        D_KL = 0.5 * (-torch.sum(1 + torch.log(var[row]**2) - (mean[row])**2 -␣
↪(var[row])**2))

        loss += - ( monte_carlo_estimate[row] - D_KL )

    return loss

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

1.0.6 *Step6: Train the model*

[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)

        # print("mean = ",mean)
        # print("var_ = ",log_var)

        loss = loss_function(x, theta, mean, log_var)
        # print("loss = ",loss.item())

        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:  166.44393517190107
```

```
        Epoch 2 complete!       Average Loss:  121.72199050826899
        Epoch 3 complete!       Average Loss:  111.0623219526033
        Epoch 4 complete!       Average Loss:  106.77539851575543
        Epoch 5 complete!       Average Loss:  104.45536607236018
        Epoch 6 complete!       Average Loss:  102.86833505060518
        Epoch 7 complete!       Average Loss:  101.63271375143468
        Epoch 8 complete!       Average Loss:  100.61815701951168
        Epoch 9 complete!       Average Loss:  99.80982209933222
        Epoch 10 complete!      Average Loss:  99.14960523398372
        Epoch 11 complete!      Average Loss:  98.6174614102932
        Epoch 12 complete!      Average Loss:  98.12324210598393
        Epoch 13 complete!      Average Loss:  97.76596914453778
        Epoch 14 complete!      Average Loss:  97.4043195038606
        Epoch 15 complete!      Average Loss:  97.0933828353245
        Epoch 16 complete!      Average Loss:  96.81429069608201
        Epoch 17 complete!      Average Loss:  96.59282402637209
        Epoch 18 complete!      Average Loss:  96.35951025146076
        Epoch 19 complete!      Average Loss:  96.16844832533388
        Epoch 20 complete!      Average Loss:  95.94142824303526
Finish!!
```

1.0.7 *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

[16]:
```
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
        #
        #
        x = torch.round(x)
        theta,_,_ = model(x)


print(theta)
```

```
100%|| 100/100 [00:00<00:00, 109.21it/s]

tensor([[1.8711e-09, 2.6298e-09, 1.5002e-09,  ..., 1.6320e-09, 1.6530e-09,
```

```
 1.8394e-09],
[6.0860e-08, 9.6870e-08, 1.6711e-07,  ..., 9.9031e-08, 1.0837e-07,
 4.6442e-08],
[3.5603e-10, 1.7039e-10, 2.2757e-10,  ..., 1.8943e-10, 1.3466e-10,
 1.1561e-10],
...,
[6.6744e-08, 5.0407e-08, 1.1255e-07,  ..., 8.4685e-08, 1.4221e-07,
 7.7155e-08],
[1.0527e-08, 1.0256e-08, 1.6202e-08,  ..., 1.2781e-08, 1.6378e-08,
 7.4796e-09],
[8.8011e-08, 4.7647e-08, 4.8835e-08,  ..., 6.8165e-08, 1.0139e-07,
 2.7916e-08]])
```

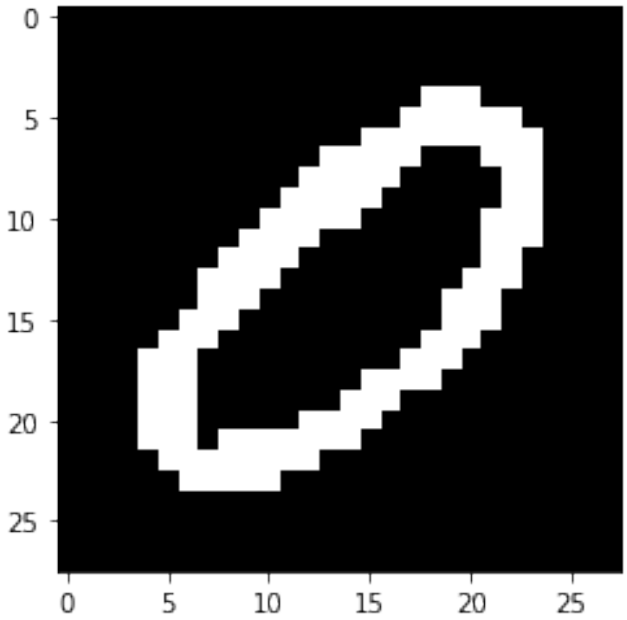
A helper function to display images:

[17]:
```
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')
```

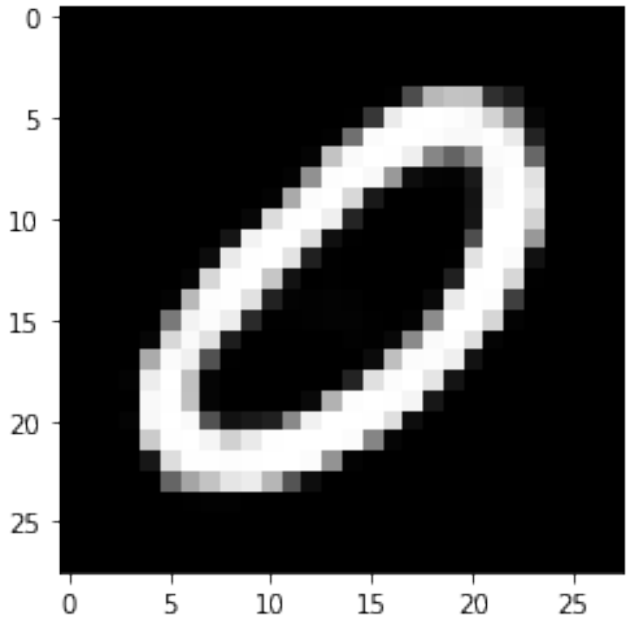
First display an image from the test dataset,

[38]:
```
show_image(x, idx=11) # try different indices as well
```



Now display its reconstruction and compare:

[37]:
```
show_image(theta, idx=11)
```



1.0.8 *Step8: Generate images from noise*

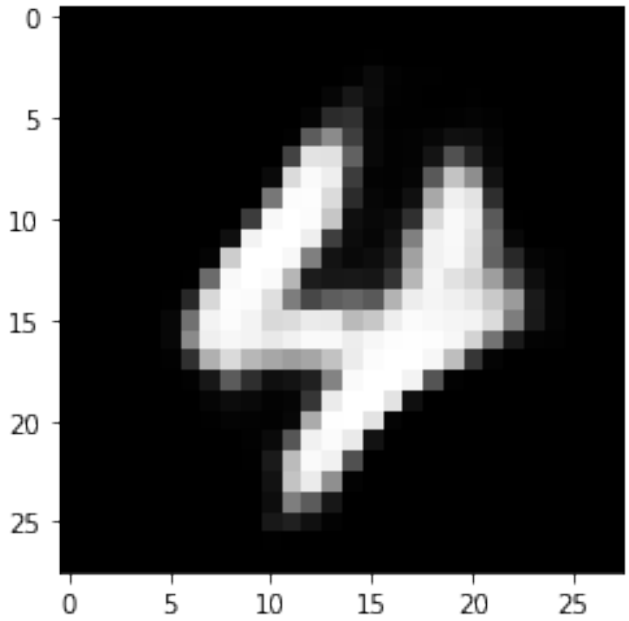
In the previous step, we sampled latent vector z from $q(z|x)$. However, we know that the KL term in our loss function enforced $q(z|x)$ to be close to $N(0, I)$. Therefore, we can sample z directly from noise $N(0, I)$, and pass it to the decoder $p(x|z)$.

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

[ ]:
```
with torch.no_grad():
    # insert your code here to create images from noise (it is enough to create␣
↪theta value for each pixel)
    noise = torch.randn(batch_size, latent_dim)
    generated_images = decoder.forward(noise)
```

Display a couple of generated images:

[32]:
```
show_image(generated_images, idx=31)
```


0
5
10
15
20
25
0
5
10
15
20
25

1B-Reparameterization

December 7, 2023

1 *Reparameterization of common distributions*

We will work with Torch throughout this notebook.

[1]:
```
import torch
from torch.distributions import Beta, exponential #, ...  import the␣
 ↪distributions you need here
from torch.nn import functional as F
```

A helper function to visualize the generated samples:

[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()
```

1.0.1 *Q1. Exponential Distribution*

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

[3]:
```
def exp_sampler(l, N):
  # insert your code
  exp_dist = torch.distributions.exponential.Exponential(rate=l)
  samples = exp_dist.sample((N,1))
  return samples.numpy() # should be N-by-1
```

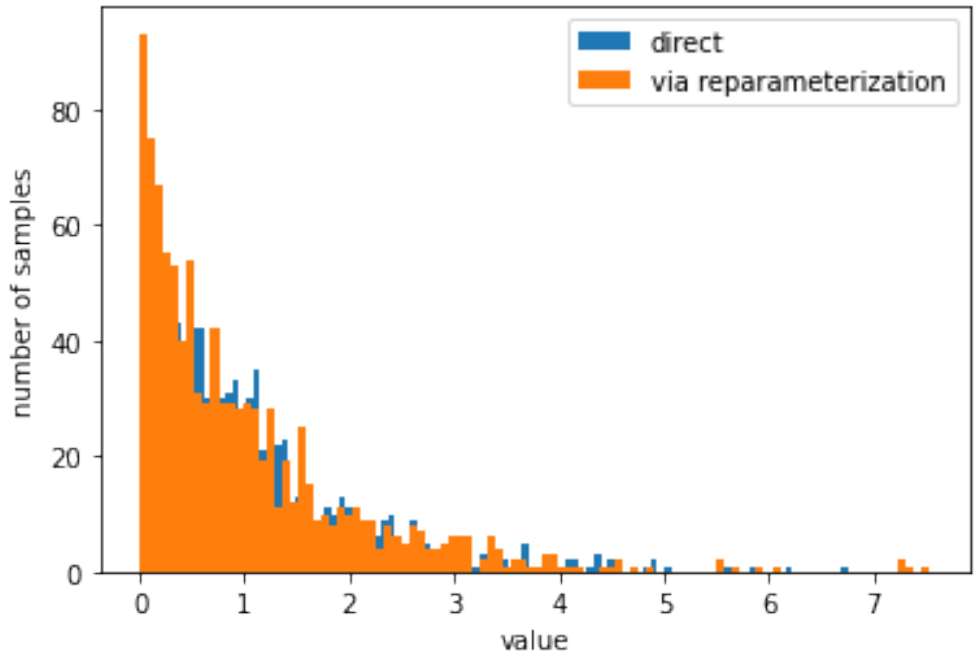
Now, implement the reparameterization trick:

[4]:
```
def exp_reparametrize(l,N):
  # this function should return N samples via reparametrization,
```

```
  # insert your code
  u = torch.rand((N,1))
  samples = (-1/l)*torch.log(1 - u)
  return samples.numpy()
```

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

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



1.0.2 *Q2. Categorical Distribution*

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

[11]:
```
def categorical_sampler(a, N):
  # insert your code
  cat_dist = torch.distributions.Categorical(probs=a)
  samples = cat_dist.sample((N,))
  return samples.numpy()  # should be N-by-1
```

Now write a function that generates samples from Categorical ($\mathbf{a}$) via reparameterization:

[22]:
```
# 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

  gumbel_samples = -torch.log(-torch.log(torch.rand((N,len(a)))))

  logits = torch.log(a + eps) + gumbel_samples

  y = torch.nn.functional.softmax(logits/temp,dim=-1)

  samples = torch.argmax(y,dim=-1)

  return samples.numpy() # 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:

[24]:
```
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)
```