# DD2434 - Machine Learning, Advanced Course

Assignment 1B

R. Darous
darous@ug.kth.se

December 11, 2023

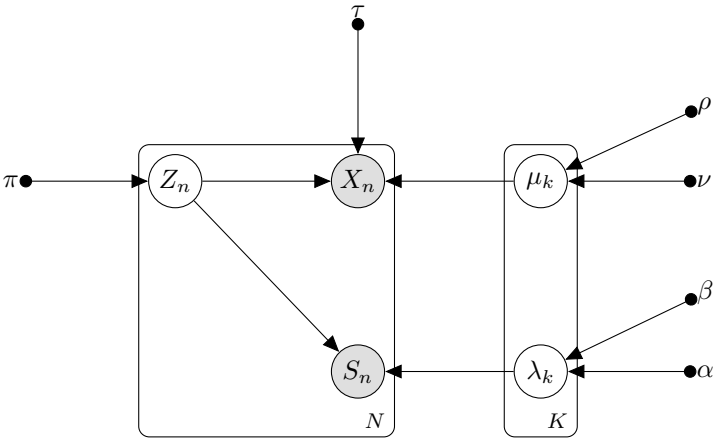
In this assignment, we will see an example of CAVI application for Earth quakes models. Then, we will take a look at VAE image generation

# Contents

# 1 CAVI for Earth quakes

In an area with frequent earthquakes emanating from $K$ super epicentra, we gather seismographic data on the strength, $S_n$, and the 2D-coordinates, $X_n$, of each outbreak. We introduce a class variable, $Z_n$, with a categorical distribution parameterized by $\pi$, which assigns the $n$th observation to a super epicentra. We model $S_n \mid Z_n = k, \lambda_k$ as a Poisson random variable with super epicentra specific intensity, $\lambda_k$, and $X_n \mid Z_n = k, \mu_k, \tau$ as a 2D Normal r.v. with super epicenter specific mean vector $\mu_k = (\mu_0, \mu_1)$ and precision matrix set to $\tau \cdot I$, where I is the 2D identity matrix. We set a 2D Normal prior on $\mu_k$ with mean vector $\nu = (\nu_0, \nu_1)$ and precision matrix $\rho \cdot I$. We set a Gamma prior on $\lambda_k$ with shape parameter $\alpha$ and rate parameter $\beta$. The remaining parameters are treated as constants.

## 1.1 DGM of the model




## 1.2 Joint distribution of the model

Let's compute the joint distribution of the model, given the DGM :

$$
\begin{aligned}
&p(X, S, Z, \lambda, \mu \mid \pi, \tau, \alpha, \beta, \nu, \rho) \\
&= \prod_{k=1}^{K} p(\mu_k \mid \nu, \rho) \prod_{k=1}^{K} p(\lambda_k \mid \alpha, \beta) \prod_{n=1}^{N} p(Z_n, X_n, S_n \mid \mu, \lambda, \pi, \tau) \\
&= \prod_{k=1}^{K} p(\mu_k \mid \nu, \rho) \prod_{k=1}^{K} p(\lambda_k \mid \alpha, \beta) \prod_{n=1}^{N} p(Z_n \mid \pi) p(X_n \mid Z_n, \mu, \tau) p(S_n \mid Z_n, \lambda)
\end{aligned}
$$

Hence, we get the following log-joint distribution for the model (hyper parameters are no longer noted for the sake of clarity) :

$$
\begin{aligned}
\log p(X, S, Z, \lambda, \mu) = &\sum_{k=1}^{K} (\log p(\mu_k) + \log p(\lambda_k)) \\
&+ \sum_{i=1}^{N} (\log p(Z_n) + \log p(X_n \mid Z_n, \mu) + \log p(S_n \mid Z_n, \lambda))
\end{aligned}
$$

## 1.3 Optimal variational distribution and parameters

Now, we will use the mean-field assumption $q(Z,\mu,\lambda) = \prod_n q(Z_n) \prod_k q(\mu_k) q(\lambda_k)$ and CAVI update equation to derive the optimal variational distributions and parameters of $q(Z_n), q(\mu_k)$ and $q(\lambda_k)$ .

First, let's compute the known densities of the joint.

### 1.3.1 Computing known densities

Let $n \in \{1,...,N\}$ and $k \in \{1,...,K\}$.

$$\begin{aligned}
\log p(Z_n) &= \log\left(\prod_{k=1}^{K} \pi_k^{Z_n^k}\right) = \sum_{k=1}^{K} Z_n^k \log(\pi_k) \text{ where } Z_n^k = \mathbf{1}_{Z_n=k} \\
\log p(X_n \mid Z_n, \mu) &= \log\left(\frac{\sqrt{\tau}}{\sqrt{2\pi}}\right) - \frac{\tau}{2}(X_n - \mu_{Z_n})^T(X_n - \mu_{Z_n}) \\
\log p(S_n \mid Z_n, \lambda) &= -\lambda_{Z_n} - \log(S_n!) + S_n \log(\lambda_{Z_n}) \\
\log p(\mu_k) &= \log\left(\frac{\sqrt{\rho}}{\sqrt{2\pi}}\right) - \frac{\rho}{2}(\mu_k - \nu)^T(\mu_k - \nu) \\
\log p(\lambda_k) &= \log\left(\frac{\beta^\alpha}{\Gamma(\alpha)}\right) + (\alpha - 1)\log(\lambda_k) - \beta\lambda_k
\end{aligned}$$

Now, using the mean field assumption and CAVI update equation, we will derive the optimal variational distribution and parameters of $q(Z_n), q(\mu_k)$ and $q(\lambda_k)$ .

Let $n \in \{1,...,N\}, k \in \{1,...,K\}$.

### 1.3.2 Deriving optimal $q^*(Z_n)$

Given CAVI update equation,

$$\begin{aligned}
\log q^*(Z_n) &= \mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[\log p(X,S,Z,\lambda,\mu)\right] \\
&\stackrel{+}{=} \mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[\log p(Z_n) + \log p(X_n \mid Z_n,\mu) + \log p(S_n \mid Z_n,\lambda)\right] \\
&= \log p(Z_n) + \mathbb{E}_{q(Z_{-n},\mu)}\left[\log\left(p(X_n \mid Z_n,\mu)\right)\right] + \mathbb{E}_{q(Z_{-n},\lambda)}\left[\log p(S_n \mid Z_n,\lambda)\right]
\end{aligned}$$

Moreover,

$$\begin{aligned}
\mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[\log\left(p(X_n \mid Z_n,\mu)\right)\right] &\stackrel{+}{=} -\frac{\tau}{2}\mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[(X_n - \mu_{Z_n})^\top (X_n - \mu_{Z_n})\right] \\
&\stackrel{+}{=} -\frac{\tau}{2}\mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[\sum_{k=1}^{k} Z_n^k (X_n - \mu_k)^\top (X_n - \mu_k)\right] \text{ where } Z_n^k = \mathbf{1}_{Z_n=k} \\
&\stackrel{+}{=} -\frac{\tau}{2}\sum_{k=1}^{k} Z_n^k \mathbb{E}_{q(\mu_k)}\left[(X_n - \mu_k)^\top (X_n - \mu_k)\right]
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[\log\left(p\left(S_n \mid Z_n,\lambda\right)\right)\right] &\stackrel{+}{=} \mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[-\lambda_{Z_n} + S_n \log\left(\lambda_{Z_n}\right)\right] \\
&\stackrel{+}{=} \mathbb{E}_{q(Z_{-n},\lambda,\mu)}\left[\sum_{k=1}^{k} Z_n^k\left(-\lambda_k + S_n \log\left(\lambda_k\right)\right)\right] \\
&\stackrel{+}{=} \sum_{k=1}^{K} Z_n^k \mathbb{E}_{q(\lambda_k)}\left[S_n \log(\lambda_k) - \lambda_k\right] \\
&\stackrel{+}{=} \sum_{k=1}^{K} Z_n^k \left(S_n \mathbb{E}_{q(\lambda_k)}\left[\log(\lambda_k)\right] - \mathbb{E}_{q(\lambda_k)}\left[\lambda_k\right]\right)
\end{aligned}$$

Injecting the computations in the variational distribution gives :

$$q^*\left(Z_n\right) \stackrel{+}{=} \sum_{k=1}^{K} Z_n^k \left(\log\left(\pi_k\right) - \frac{\tau}{2}\mathbb{E}_{q(\mu_k)}\left[\left(X_n - \mu_k\right)^\top \left(X_n - \mu_k\right)\right] + S_n \mathbb{E}_{q(\lambda_k)}\left[\log(\lambda_k)\right] - \mathbb{E}_{q(\lambda_k)}\left[\lambda_k\right]\right)$$

Thus, $Z_n \sim \mathrm{Cat}((\pi^*_{n,k})_{k\in\{1,\dots,K\}}$ where

$$\forall k \in \{1,...,K\}, \pi^*_{n,k} = \log\left(\pi_k\right) - \frac{\tau}{2}\mathbb{E}_{q(\mu_k)}\left[\left(X_n - \mu_k\right)^\top \left(X_n - \mu_k\right)\right] + S_n \mathbb{E}_{q(\lambda_k)}\left[\log(\lambda_k)\right] - \mathbb{E}_{q(\lambda_k)}\left[\lambda_k\right]$$

### 1.3.3 Deriving optimal $q^*(\lambda_k)$

Given CAVI update equation,

$$\begin{aligned}
\log q^*(\lambda_k) &= \mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[\log p(X,S,Z,\lambda,\mu)\right] \\
&\stackrel{+}{=} \mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[\log p\left(\lambda_k\right) + \sum_{n=1}^{N} \log p\left(S_n \mid Z_n,\lambda\right)\right] \\
&\stackrel{+}{=} \log p\left(\lambda_k\right) + \sum_{n=1}^{N} \mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[\log p\left(S_n \mid Z_n,\lambda\right)\right]
\end{aligned}$$

Moreover,

$$\begin{aligned}
\mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[\log p\left(S_n \mid Z_n,\lambda\right)\right] &\stackrel{+}{=} \mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[-\lambda_{Z_n} + S_n \log\left(\lambda_{Z_n}\right)\right] \\
&\stackrel{+}{=} \mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[\sum_{l=1}^{K} Z_n^l\left(-\lambda_l + S_n \log\left(\lambda_l\right)\right)\right] \\
&\text{(sum of constant terms w.r.t k but for m=k)} \\
&\stackrel{+}{=} \mathbb{E}_{q(Z,\lambda_{-k},\mu)}\left[Z_n^k\left(\left(-\lambda_k\right) + S_n \log(\lambda_k)\right)\right] \\
&\stackrel{+}{=} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right]\left(-\lambda_k + S_n \log(\lambda_k)\right)
\end{aligned}$$

Injecting the computations in the variational distribution gives :

$$\begin{aligned}
\log q^*(\lambda_k) &= (\alpha - 1)\log\lambda_k - \beta\lambda_k \\
&\quad - \lambda_k \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] + \log\lambda_k \sum_{n=1}^{N} S_n \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] \\
&= \left(\alpha + \sum_{n=1}^{N} S_n \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] - 1\right)\log\lambda_k - \left(\beta + \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right]\right)\lambda_k
\end{aligned}$$

Thus, $\lambda_k \sim \text{Gamma}(\alpha^*, \beta^*)$, where

$$\alpha^* = \alpha + \sum_{n=1}^{N} S_n \mathbb{E}_{q(Z_n)}\left[Z_n^k\right], \qquad \beta^* = \beta + \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right]$$

### 1.3.4 Deriving optimal $q^*(\mu_k)$

Given CAVI update equation,

$$\begin{aligned}
\log q^*(\mu_k) &= \mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[\log p(X, S, Z, \lambda, \mu)\right] \\
&\stackrel{+}{=} \mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[\log p(\mu_k) + \sum_{n=1}^{N} \log p(X_n \mid Z_n, \mu)\right] \\
&\stackrel{+}{=} \log p(\mu_k) + \sum_{n=1}^{N} \mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[\log p(X_n \mid Z_n, \mu)\right]
\end{aligned}$$

Moreover,

$$\begin{aligned}
\mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[\log p(X_n \mid Z_n, \mu)\right] &\stackrel{+}{=} \mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[-\frac{\tau}{2}(X_n - \mu_{Z_n})^T(X_n - \mu_{Z_n})\right] \\
&\stackrel{+}{=} -\frac{\tau}{2}\mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[\sum_{l=1}^{K} Z_n^l (X_n - \mu_l)^T(X_n - \mu_l)\right] \\
&\quad \text{(sum of constant terms w.r.t k but for l = k)} \\
&\stackrel{+}{=} -\frac{\tau}{2}\mathbb{E}_{q(Z,\lambda,\mu_{-k})}\left[Z_n^k (X_n - \mu_k)^T(X_n - \mu_k)\right] \\
&\stackrel{+}{=} -\frac{\tau}{2}(X_n - \mu_k)^T(X_n - \mu_k)\mathbb{E}_{q(Z_n)}\left[Z_n^k\right]
\end{aligned}$$

Injecting the computations in the variational distribution gives :

$$\begin{aligned}
\log q^*(\mu_k) &\stackrel{+}{=} -\frac{\rho}{2}(\mu_k - \nu)^T(\mu_k - \nu) - \frac{\tau}{2}\sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right](X_n - \mu_k)^T(X_n - \mu_k) \\
&\stackrel{+}{=} -\frac{\rho}{2}(\mu_k^T\mu_k - \mu_k^T\nu - \nu^T\mu_k) \\
&\quad - \frac{\tau}{2}\sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right](\mu_k^T\mu_k - \mu_k^T X_n - X_n^T\mu_k) \\
&\stackrel{+}{=} -\frac{\rho}{2}(\mu_k^T\mu_k - \mu_k^T\nu - \nu^T\mu_k) \\
&\quad - \frac{\tau}{2}\left(\mu_k^T\mu_k \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] - \mu_k^T \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n - \sum_{n=1}^{N}\left(\mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n^T\right)\mu_k\right) \\
&\stackrel{+}{=} -\frac{1}{2}\left[\rho + \tau\sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right]\right]\mu_k^T\mu_k \\
&\quad - \frac{1}{2}\mu_k^T\left(-\rho\nu - \tau\sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n\right) \\
&\quad - \frac{1}{2}\left(-\rho\nu^T - \tau\sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n^T\right)\mu_k
\end{aligned}$$

Then,

$$
\begin{aligned}
\log q^*(\mu_k) &\stackrel{+}{=} -\frac{1}{2}\rho^* \mu_k^T \mu_k \\
&\quad - \frac{1}{2}\mu_k^T \left( -\rho\nu - \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right) \\
&\quad - \frac{1}{2}\left( -\rho\nu - \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right)^T \mu_k \qquad \text{with } \rho^* = \left[ \rho + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] \right] \\
&\stackrel{+}{=} -\frac{1}{2}\rho^* \left( \mu_k^T \mu_k - \mu_k^T \left( \rho\nu + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right) /\rho^* - \left( \rho\nu + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right)^T /\rho^* \mu_k \right) \\
&\stackrel{+}{=} -\frac{1}{2}\rho^* (\mu_k - \nu^*)^T (\mu_k - \nu^*)
\end{aligned}
$$

where $\mu^* = \left( \rho\nu + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right) /\rho^*$.

The last line is obtained by adding the constant $-\frac{1}{2}\rho^* \left( \rho\nu + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right)^T /\rho^* \cdot \left( \rho\nu + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right) /\rho^*$

Hence, $\mu_k \sim \mathcal{N}(\mu^*, \frac{1}{\rho^*}\mathbf{I})$, where :

$$
\begin{aligned}
\mu^* &= \left( \rho\nu + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] X_n \right) /\rho^* \\
\rho^* &= \left[ \rho + \tau \sum_{n=1}^{N} \mathbb{E}_{q(Z_n)}\left[Z_n^k\right] \right]
\end{aligned}
$$

### 1.3.5 Computing the parameters of the optimal distributions

To compute all those distributions, we need to precise the value of some expressions in the parameters derivations.

To compute $q^*(Z_n)$, we need, knowing that $\lambda_k \sim \text{Gamma}(\alpha^*, \beta*)$ and $\mu_k \sim \mathcal{N}(\nu^*, \frac{1}{\rho^*}\mathbf{I})$ :

$$
\begin{aligned}
&\mathbb{E}_{q(\lambda_k)}\left[\log(\lambda_k)\right] = \psi(\alpha^*) - \log\beta^* \\
&\mathbb{E}_{q(\lambda_k)}\left[\lambda_k\right] = \frac{\alpha^*}{\beta^*} \\
&\mathbb{E}_{q(\mu_k)}\left[ (X_n - \mu_k)^\top (X_n - \mu_k) \right] \\
&= X_n^T X_n - \mathbb{E}_{q(\mu_k)}\left[\mu_k\right]^T X_n - X_n^T \mathbb{E}_{q(\mu_k)}\left[\mu_k\right] + \mathbb{E}_{q(\mu_k)}\left[\mu_k^T \mu_k\right] \\
&= X_n^T X_n - {\nu^*}^T X_n - X_n^T \nu^* + {\nu^*}^T \nu^* + \frac{2}{\rho^*} \\
&= \frac{2}{\rho^*} + (X_n - \nu^*)^T (X_n - \nu^*)
\end{aligned}
$$

To compute $q^*(\lambda_k)$, we need :

$$
\begin{aligned}
\mathbb{E}_{q(Z_n)}\left[Z_n^k\right] &= \mathbb{E}_{q(Z_n)}\left[\mathbf{1}_{Z_n=k}\right] \\
&= q(Z_n = k) = \pi_{n,k}^*
\end{aligned}
$$

as $Z_n \sim \text{Cat}((\pi^*_{n,k})_k)$.

Finally, to compute $q^*(\mu_k)$, we need :

$$\mathbb{E}_{q(Z_n)}\left[Z_n^k\right] = \pi^*_{n,k}$$

as shown above.

Performing a first initialisation of the global hidden variables $\lambda$ and $\mu$ allows then to compute the distributions of the $Z_n$ and to implement the CAVI algorithm.

# 2 VAE image generation

The notebook "1B-VAE.ipynb" attached to the paper (whose code is given in the Annex) presents a VAE model to generate images. Here is a quick description of the model :

**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.

## 2.1 ELBO properties for the model

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

Let's show that ELBO can be rewritten as:

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

Using Bayes' rule, we have :

$$\begin{aligned} E_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right] &= E_{q(z|x)}\left[\log p(x \mid z)\right] + E_{q(z|x)}\left[\log p(z)\right] - E_{q(z|x)}\left[\log q(z \mid x)\right] \\ &= E_{q(z|x)}\left[\log p(x \mid z)\right] - E_{q(z|x)}\left[\log\left(\frac{q(z \mid x)}{p(z)}\right)\right] \\ &= E_{q(z|x)}\big(\log p(x|z)\big) - D_{KL}\big(q(z|x)||p(z)\big). \end{aligned}$$

We get de KL term just by using its definition. Hence, we get the result.

The first term can be approximated using Monte-Carlo integration, as explained in the notebook.

## 2.2 Close form of the KL divergence

Let's consider the second term: $-D_{KL}\big(q(z|x)||p(z)\big)$.

Kullback–Leibler divergence can be computed using the closed-form analytic expression when both the variational and the prior distributions are Gaussian.

Let's recall that :

- $z \sim N(0, I)$
- The variational distribution $q(z|x)$ is chosen to be multivariate Gaussian with a diagonal covariance matrix : $q(z|x) \sim q(\mu, \sigma^2)$, where $\mu, \sigma^2 = f(x)$.
- Let's denote $D$ the dimension of the latent space.

Then, we have that :

$$q(z \mid x) = \frac{1}{(2\pi)^{-D/2}} \frac{1}{\prod_{i=1}^{D} \sigma_i} e^{-\frac{1}{2}\left((z-\mu)^T \Sigma^{-1} (z-\mu)\right)}$$

,

where $\Sigma$ is the covariance matrix of the variational distribution, and :

$$p(z) = \frac{1}{(2\pi)^{-D/2}} e^{-\frac{1}{2} z^T z}$$

And :

$$\begin{aligned}
\log \frac{q(z \mid x)}{p(z)} &= -\sum_{i=1}^{D} \log \sigma_i - \frac{1}{2}\left((z-\mu)^T \Sigma^{-1} (z-\mu)\right) + \frac{1}{2} z^T z \\
&= -\sum_{i=1}^{D} \log \sigma_i - \frac{1}{2} \sum_{i=1}^{D} \frac{(z_i - \mu_i)^2}{\sigma_i^2} + \frac{1}{2} \sum_{i=1}^{D} z_i^2
\end{aligned}$$

Then,

$$\begin{aligned}
D_{KL}\big(q(z|x)||p(z)\big) &= E_{q(z|x)}\left[\log\left(\frac{q(z \mid x)}{p(z)}\right)\right] \\
&= -\sum_{i=1}^{D} \log \sigma_i - \frac{1}{2} \sum_{i=1}^{D} E_{q(z|x)}\left[\frac{(z_i - \mu_i)^2}{\sigma_i^2}\right] + \frac{1}{2} \sum_{i=1}^{D} E_{q(z|x)}\left[z_i^2\right]
\end{aligned}$$

Let $j \in \{1, ..., D\}$.

$$
\begin{aligned}
E_{q(z|x)}\left[z_j^2\right] &= \int z_j^2 \frac{1}{(2\pi)^{-D/2}} \frac{1}{\prod_{i=1}^{D} \sigma_i} e^{-\frac{1}{2}\sum_{i=1}^{D} \frac{(z_i-\mu_i)^2}{\sigma_i^2}} \mathrm{d}z_1, ...\mathrm{d}z_D \\
&= \int z_j^2 \frac{1}{(\sqrt{2\pi})} \frac{1}{\sigma_j} e^{-\frac{1}{2}\frac{(z_j-\mu_j)^2}{\sigma_j^2}} \mathrm{d}z_j \\
&= E_{z_j \sim \mathcal{N}(\mu_j, \sigma_j^2)}\left[z_j^2\right] \\
&= \sigma_j^2 + \mu_j^2 \\
E_{q(z|x)}\left[z_j\right] &= \int z_j \frac{1}{(2\pi)^{-D/2}} \frac{1}{\prod_{i=1}^{D} \sigma_i} e^{-\frac{1}{2}\sum_{i=1}^{D} \frac{(z_i-\mu_i)^2}{\sigma_i^2}} \mathrm{d}z_1, ...\mathrm{d}z_D \\
&= \int z_j \frac{1}{(\sqrt{2\pi})} \frac{1}{\sigma_j} e^{-\frac{1}{2}\frac{(z_j-\mu_j)^2}{\sigma_j^2}} \mathrm{d}z_j \\
&= E_{z_j \sim \mathcal{N}(\mu_j, \sigma_j)}\left[z_j\right] \\
&= \mu_j \\
E_{q(z|x)}\left[\frac{(z_j-\mu_j)^2}{\sigma_j^2}\right] &= \frac{1}{\sigma_j^2}\left(E_{q(z|x)}\left[z_j^2\right] - 2\mu_j E_{q(z|x)}\left[z_j\right] + \mu_j^2\right) \\
&= \frac{1}{\sigma_j^2}\left(\sigma_j^2 + \mu_j^2 - 2\mu_j^2 + \mu_j^2\right) = 1
\end{aligned}
$$

Hence,

$$
\begin{aligned}
D_{KL}\big(q(z|x)||p(z)\big) &= E_{q(z|x)}\left[\log\left(\frac{q(z \mid x)}{p(z)}\right)\right] \\
&= -\frac{1}{2}\sum_{i=1}^{D} \log \sigma_i^2 - \frac{1}{2}\sum_{i=1}^{D} 1 + \frac{1}{2}\sum_{i=1}^{D} \sigma_i^2 + \mu_i^2 \\
&= -\frac{1}{2}\sum_{i=1}^{D} \log \sigma_i^2 + 1 - (\sigma_i^2 + \mu_i^2)
\end{aligned}
$$

## 2.3 Training the model

Using those computations, we can train VAE modell given in the notebbok. For the full code and results, check the Annex or the attached notebook file. Here are just shown a fex examples of the results of the trained model :

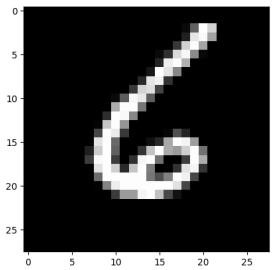



Figure 1: An image from the dataset

### 2.3.1 Image reconstruction

We can see that the reconstructed image is rather close from the original image.

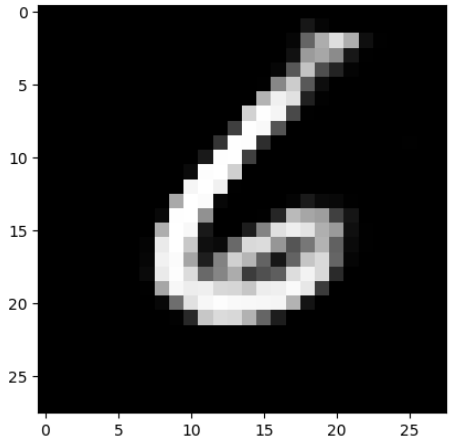



Figure 2: The reconstructed image when putting the image above as input of the VAE encoder

### 2.3.2 Image generated from noise

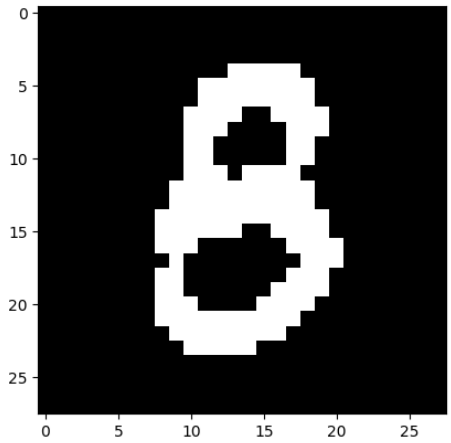



Figure 3: The reconstructed image when putting the image from Figure 1 as input of the VAE encoder

Even if that's not the case for every sample due to randomness of the noise, the model is able to generate images that look like number (here the number 8).

# 3 Reparameterization and the score function

In modules 5 and 6, we were introduced to two estimators of the gradient of the ELBO: **one high variance estimator** which uses the score function (the REINFORCE estimator) and **one lower variance estimator** which uses the reparameterization trick (the reparameterized gradient estimator). In the In the paper *Sticking the Landing*, Roeder et al. show that the reparameterized gradient estimator also contains a score function, which effects the variance of the estimator.

## 3.1 Decomposing the gradient of the ELBO

Let's take a look at the gradient of the ELBO. We use the same notations as in the paper and use the reparameterization $z = t(\epsilon, \phi)$.

$$\begin{aligned}
\mathcal{L}(\phi) &= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{x} \mid \mathbf{z}) + \log p(\mathbf{z}) - \log q_\phi(\mathbf{z} \mid \mathbf{x})\right] \\
&= \mathbb{E}_{\mathbf{z}\sim q}\left[\log p(\mathbf{z} \mid \mathbf{x}) + \log p(\mathbf{x}) - \log q_\phi(\mathbf{z} \mid \mathbf{x})\right] \text{ (using Bayes' rule)} \\
&= \mathbb{E}_{\epsilon\sim q}\left[\log p(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x}) + \log p(\mathbf{x}) - \log q_\phi(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x})\right]
\end{aligned}$$

As $\epsilon$ does not depend on $\phi$, we can write, using the reparameterization trick :

$$\nabla\mathcal{L}(\phi) = \mathbb{E}_{\epsilon\sim q}\left[\nabla_\phi \left(\log p(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x}) + \log p(\mathbf{x}) - \log q_\phi(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x})\right)\right]$$

As done in the paper, we can decompose the total derivative ($\hat{TD}$) of the integrand of the estimator w.r.t. the trainable parameters $\phi$ as :

$$\begin{aligned}
\hat{\nabla}_{\mathrm{TD}}(\epsilon, \boldsymbol{\phi}) &= \nabla_\phi \left[\log p(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x}) + \log p(\mathbf{x}) - \log q_\phi(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x})\right] \\
&= \nabla_\phi \left[\log p(\mathbf{t}(\epsilon, \phi) \mid \mathbf{x}) + \log p(\mathbf{x}) - \log q_\phi(\mathbf{z} \mid \mathbf{x})\right] \\
&= \underbrace{\nabla_{\mathbf{z}} \left[\log p(\mathbf{z} \mid \mathbf{x}) - \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})\right] \nabla_{\boldsymbol{\phi}} t(\epsilon, \boldsymbol{\phi})}_{\text{path derivative}} - \underbrace{\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}_{\text{score function}},
\end{aligned}$$

The last line is a consequence of the reparameterization trick and of the chain rule. We can observe that **we indeed observe a score function**.

## 3.2 Showing that the expectation of the score function is zero

Even we have a score function, we can show that it's expectation is zero.

Indeed,

$$\begin{aligned}
\mathbb{E}_{\mathbf{z}\sim q}[\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})] &= \int q_\phi(z) \nabla_\phi \log q_\phi(z)\, dz \\
&= \int \nabla_\phi q_\phi(z)\, dz \quad \text{as } \nabla_\phi \log q_\phi(z) = \frac{1}{q_\phi(z)} \nabla_\phi q_\phi(z) \\
&= \nabla_\phi \int q_\phi(z) \mathrm{d}z \\
&= \nabla_\phi(1) \\
&= 0.
\end{aligned}$$

Hence, we get the result.

### 3.3 Handling the score function

As the expectation of the score function is zero, the authors suggest to remove the term from the estimator of the ELBO. It remains unbiased, which is necessary for having a stochastic gradient descent to converge.

### 3.4 Cases where the score function decreases the variance

The authors mention that for particular cases, the score function may actually decrease the variance. In those cases, the score function acts as a **control variate**.

# 4 Reparameterization and common distributions

Even though Gaussian distributions are easy to reparameterize and convenient to work with, they are not always the best fit for your model. For instance, your model may require the latent variable to be non-negative, or in $[0, 1]$. In order to apply VAE for those cases, we sometimes have to reparameterize some other distributions. Here, will apply the reparameterization trick to two different distributions (see the notebook "1B-Reparameterization.ipynb"):

## 4.1 Exponential distribution

In the paper *Auto-Encoding Variational Bayes* from Kingma and Welling is presented a method to reparameterize the exponential distribution.
Let $\epsilon \sim \mathcal{U}(0, 1)$, $X \sim \mathcal{E}(\lambda)$ and $F$ the CDF of $X$.

We have that $F^{-1}(\epsilon) \sim \mathcal{E}(\lambda)$.

We also have that :

- $F(x) = 1 - \mathrm{e}^{-\lambda x}$,
- $F^{-1}(y) = -\frac{\log(1-y)}{y}$

Hence, we can sample from the distribution of $\epsilon$, a reduced centered uniform distribution and then reparameterize by applying $F^{-1}$ to the samples.

### 4.1.1 Sampling from reparameterized exponential distribution

In the notebook is implemented the sampling of 1000 data points directly sampled from an exponential distribution, and using reparameterization. The histograms of the distributions are displayed here :

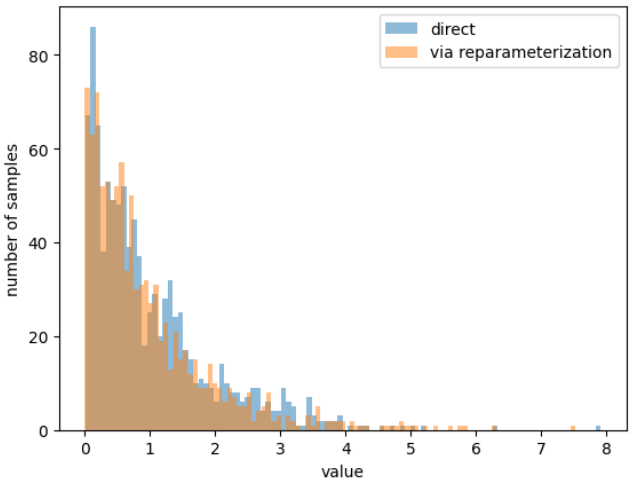



Figure 4: Sampling comparison : directly from the exponential distribution (blue histogram) or using reparameterization (orange)

We end up with a rather close distribution of the data points with the two methods.

## 4.2 Categorical distribution

In this section, we use the paper *Categorical Reparameterization with Gumbel-Softmax* from Jang, Gu, and Poole as a reference.
Here, we want to reparameterize $Z \sim \text{Cat}(\pi_1, ..., \pi_k)$.

The first simple model consists in sampling :

$$z = \text{one\_hot}\left(\arg\max_i \left[g_i + \log \pi_i\right]\right)$$

where $g_1 \dots g_k$ are i.i.d samples drawn from Gumbel$(0, 1)$.
However, because of the argmax function, this reparameterization does not allow differentation during training.

Hence, an alternative is suggested in the paper, that can split in two steps :

### 4.2.1 Approximating it by the Gumbel-Softmax distribution

We approximate the argmax function by the differentiable softmax function.

Then, we build the samples by generating a $k$-dimensional sample vectors $y$ where

$$y_i = \frac{\exp\left(\left(\log\left(\pi_i\right) + g_i\right)/\tau\right)}{\sum_{j=1}^{k} \exp\left(\left(\log\left(\pi_j\right) + g_j\right)/\tau\right)} \quad \text{for } i = 1, \dots, k$$

$\tau$ is the softmax temperature. In the paper is written that if the temperature $\tau$ approaches 0 , samples from the GumbelSoftmax distribution become one-hot and the Gumbel-Softmax distribution becomes identical to the categorical distribution above.

### 4.2.2 Using the argmax function for evaluation

Finally, to get a sample from the approximated reparameterized distribution, we just have to set :

$$z = \text{one\_hot}\left(\arg\max_i y_i\right)$$

where $y$ is defined above.

#### 4.2.3 Sampling from reparametrized categorical distribution

In the notebook is implemented the sampling of 1000 data points directly sampled from a categorical distribution, and using reparameterization. The histograms of the distributions are displayed here :

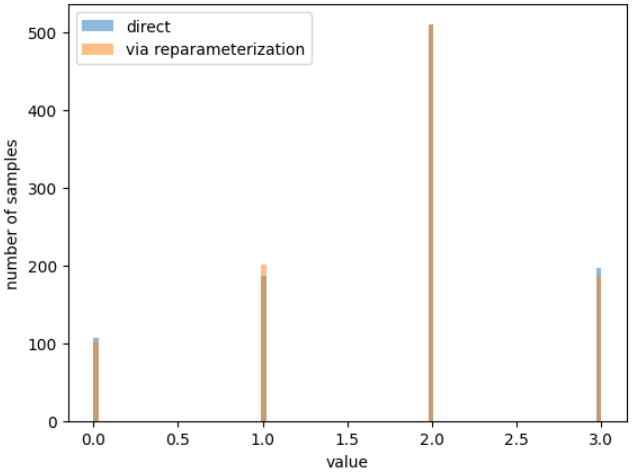



Figure 5: Sampling comparison : directly from a categorical distribution (blue histogram) or using reparameterization (orange)

We end up with a rather close distribution of the data points with the two methods.

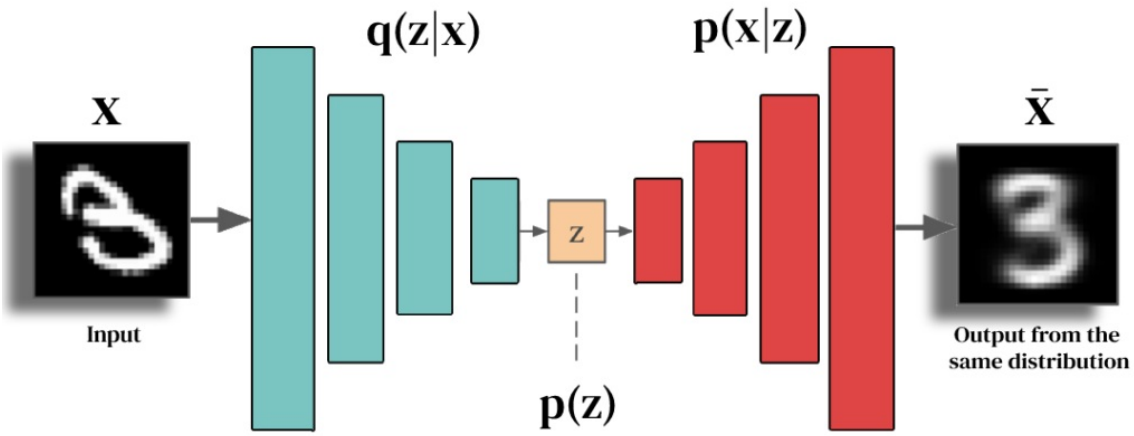
# 5 Annex

## 5.1 Code for Section 2 : VAE image generation

# *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 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=Tru
e)

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 connecte
d 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 mo
del
```

```
        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 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
        # sample epsilon from a standard normal distribution
        # return the sampled z
        epsilon = torch.randn(mean.shape)
        z = mean + torch.sqrt(var) * epsilon
        return z


    def forward(self, x):
        mean, log_var = self.Encoder(x)
        z = self.reparameterization(mean, torch.exp(log_var)) # takes exponential functi
on (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 Show that ELBO can be rewritten as:**

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

**Using Bayes' rule, we have :**

$$\begin{aligned}
E_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right] &= E_{q(z|x)}[\log p(x \mid z)] + E_{q(z|x)}[\log p(z)] - E_{q(z|x)}[\log q(z \mid x)] \\
&= E_{q(z|x)}[\log p(x \mid z)] - E_{q(z|x)}\left[\log \left(\frac{q(z \mid x)}{p(z)}\right)\right] \\
&= E_{q(z|x)}\big(\log p(x|z)\big) - D_{KL}\big(q(z|x)||p(z)\big).
\end{aligned}$$

**We get de KL term just by using its definition. Hence, we get the result.**

**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:**

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

**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:** $-D_{KL}\left(q(z|x)\right.$
$\left.||p(z)\right)$

- **Q5.2 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.**

**Let's recall that :**

- $z \sim N(0,$
  $I)$
- **The variational distribution** $q(z|x)$ **is chosen to be multivariate Gaussian with a diagonal covariance matrix :**
  $q(z|x)$ **, where** $\mu, \sigma^2$ **.**
  $\sim q(\mu,$ $= f(x)$
  $\sigma^2)$
- **Let's denote** $D$ **the dimension of the latent space.**

**Then, we have that :**

- $q(z \mid x)$ **, where** $\Sigma$ **is the covariance matrix of the variational distribution, and :**
  $-\frac{1}{2}$
  $\left((z-\mu)^T \Sigma^{-1}\right.$
  $= \frac{1}{(2\pi)^{-D/2}} \frac{1}{\prod_{i=1}^{D} \sigma_i} e^{\left.(z-\mu)\right)}$
- $p(z)$
  $= \frac{1}{(2\pi)^{-D/2}} e^{-\frac{1}{2} z^T z}$

**And :**

$$\log \frac{q(z \mid x)}{p(z)} = -\sum_{i=1}^{D} \log \sigma_i - \frac{1}{2} \left((z-\mu)^T \Sigma^{-1} (z-\mu)\right) + \frac{1}{2} z^T z = -\sum_{i=1}^{D} \log \sigma_i - \frac{1}{2} \sum^{D} \frac{(z_i-\mu_i)^2}{-2}$$

$$\overline{i=1} \quad \sigma_i^-$$

$$+\frac{1}{2}\sum_{i=1}^{D} z_i^2$$

**Then,**

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

$$= -\sum_{i=1}^{D}\log$$

$$\sigma_i - \frac{1}{2}$$

$$\sum_{i=1}^{D} E_{q(z|x)}$$

$$\left[\frac{(z_i-\mu_i)^2}{\sigma_i^2}\right]$$

$$+\frac{1}{2}$$

$$\sum_{i=1}^{D} E_{q(z|x)}\ \left[z_i^2\right]$$

**Let** $j \in \{1,\ldots,D$ **.**
$\}$

$$E_{q(z|x)}\ \left[z_j^2\right] = \int z_j^2 \frac{1}{(2\pi)^{-D/2}} \frac{1}{\prod_{i=1}^{D}\sigma_i} e^{-\frac{1}{2}\sum_{i=1}^{D}\frac{(z_i-\mu_i)^2}{\sigma_i^2}}\ \mathrm{d}z_1,\ldots \mathrm{d}z_D$$

$$= \int z_j^2 \frac{1}{(\sqrt{2\pi})} \frac{1}{\sigma_j} e^{-\frac{1}{2}\frac{(z_j-\mu_j)^2}{\sigma_j^2}}\ \mathrm{d}z_j$$

$$= E_{z_j\sim\mathcal{N}(\mu_j,\sigma_j^2)}\left[z_j^2\right]$$

$$= \sigma_j^2 + \mu_j^2$$

$$E_{q(z|x)}\ [z_j] = \int z_j \frac{1}{(2\pi)^{-D/2}} \frac{1}{\prod_{i=1}^{D}\sigma_i} e^{-\frac{1}{2}\sum_{i=1}^{D}\frac{(z_i-\mu_i)^2}{\sigma_i^2}}\ \mathrm{d}z_1,\ldots \mathrm{d}z_D$$

$$= \int z_j \frac{1}{(\sqrt{2\pi})} \frac{1}{\sigma_j} e^{-\frac{1}{2}\frac{(z_j-\mu_j)^2}{\sigma_j^2}}\ \mathrm{d}z_j$$

$$= E_{z_j\sim\mathcal{N}(\mu_j,\sigma_j)}[z_j]$$

$$= \mu_j$$

$$E_{q(z|x)} = \frac{1}{\sigma_j^2}$$

$$\left[(z_j-\mu_j)^2\right]$$

$$\left[\frac{\qquad}{\sigma_j^2}\right] \qquad \left(E_{q(z|x)}\left[z_j^2\right] - 2\mu_j E_{q(z|x)}\left[z_j\right] + \mu_j^2\right) = \frac{1}{\sigma_j^2}\left(\sigma_j^2 + \mu_j^2 - 2\mu_j^2 + \mu_j^2\right) = 1$$

**Hence,**

$$D_{KL}\left(q(z|x)||p(z)\right) = E_{q(z|x)}\left[\log\left(\frac{q(z\mid x)}{p(z)}\right)\right] = -\frac{1}{2}\sum_{i=1}^{D}\log\sigma_i^2 - \frac{1}{2}\sum_{i=1}^{D}1 + \frac{1}{2}\sum_{i=1}^{D}\sigma_i^2 + \mu_i^2 = -\frac{1}{2}\sum_{i=1}^{D}\log\sigma_i^2 + 1 - (\sigma_i^2 + \mu_i^2)$$

**Q5.3 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
    # return the loss function (- ELBO)

    # left term of ELBO
    loss = - torch.sum(x * torch.log(theta + 1e-8) + (1 - x) * torch.log(1 - theta + 1e-
8), dim=1)
    # KL divergence
    loss += - 0.5 * torch.sum(1 + log_var - mean.pow(2) - log_var.exp(), dim=1)
    loss = torch.sum(loss)

    return loss

# 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_i
dx*batch_size))

print("Finish!!")
```

```
Start training VAE...
 Epoch 1 complete!  Average Loss:  171.54891246152442
 Epoch 2 complete!  Average Loss:  119.5578116685361
 Epoch 3 complete!  Average Loss:  104.07018823690525
 Epoch 4 complete!  Average Loss:  97.83490304478819
 Epoch 5 complete!  Average Loss:  94.52309987674771
 Epoch 6 complete!  Average Loss:  92.00130045844637
 Epoch 7 complete!  Average Loss:  90.20876027917623
 Epoch 8 complete!  Average Loss:  88.88070362224802
 Epoch 9 complete!  Average Loss:  87.72827453307596
 Epoch 10 complete!  Average Loss:  86.67465288012573
 Epoch 11 complete!  Average Loss:  85.87502073768782
 Epoch 12 complete!  Average Loss:  85.26220632206021
 Epoch 13 complete!  Average Loss:  84.707846565565
 Epoch 14 complete!  Average Loss:  84.18755778674092
 Epoch 15 complete!  Average Loss:  83.77216710467968
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 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
        mean, log_var = model.Encoder(x)
        z = model.reparameterization(mean, torch.exp(log_var))
        theta = model.Decoder(z)
```

```
100%|██████████| 100/100 [00:01<00:00, 69.68it/s]
```

**A helper function to display images:**

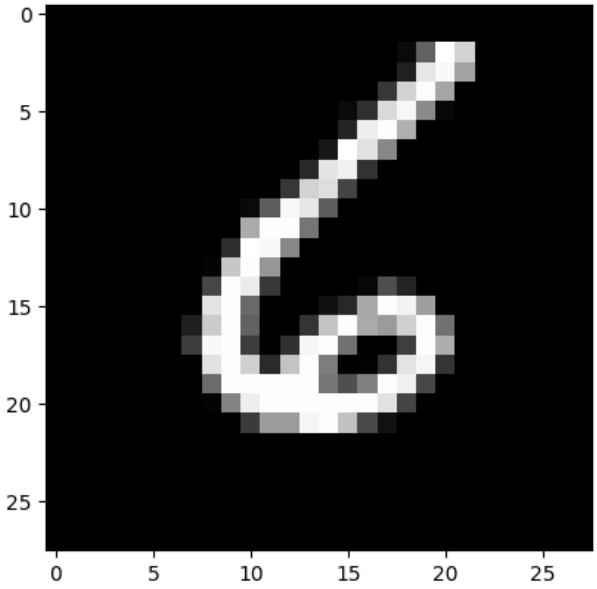
In [11]:

```
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 [12]:

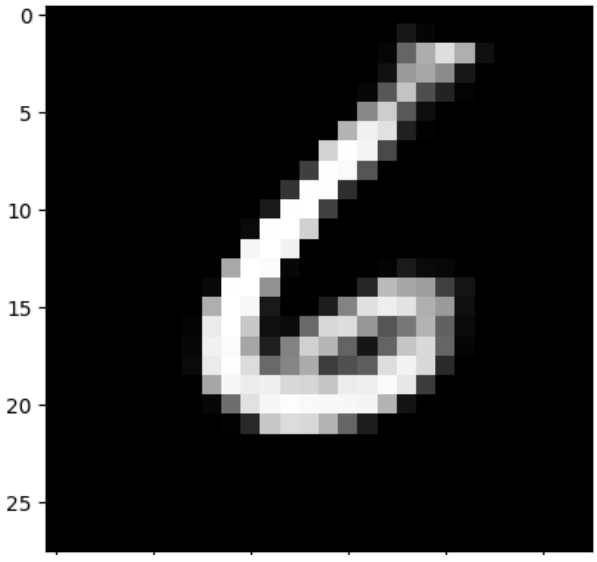
```
show_image(x, idx=8) # try different indices as well
```




**Now display its reconstruction and compare:**

In [13]:

```
show_image(theta, idx=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 Create images from noise and display.**

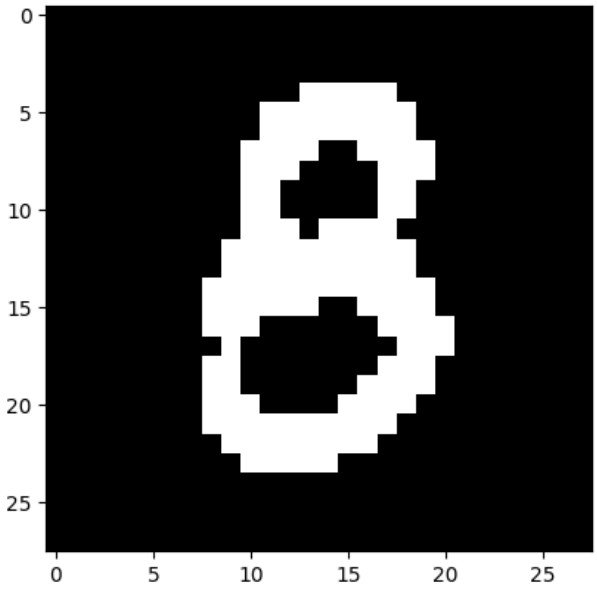
In [14]:

```
with torch.no_grad():
    # insert your code here to create images from noise (it is enough to create theta val
ue for each pixel)
    z = torch.randn(batch_size, latent_dim)
    theta = model.Decoder(z)
    generated_images = torch.round(theta) # should be a matrix ( batch_size-by-x_dim )
```

**Display a couple of generated images:**

In [16]:

```
show_image(generated_images, idx=13)
```



## 5.2 Code for Section 4 : Reparameterization

# *Reparameterization of common distributions*

We will work with Torch throughout this notebook.

```
import torch
from torch.distributions import Beta, Exponential, Uniform, Gumbel,
Categorical #, ...  import the distributions you need here
from torch.nn import functional as F
```

A helper function to visualize the generated samples:

```
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, alpha=0.5)
    plt.hist(samples_2, bins=bins, range=range, alpha=0.5)
  else:
    plt.hist(samples_1, bins=bins, alpha=0.5)
    plt.hist(samples_2, bins=bins, alpha=0.5)
  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 $Exp(\lambda)$.

```
def exp_sampler(l, N):
  samples = Exponential(l).sample((N,))
  return samples # should be N-by-1
```

Now, implement the reparameterization trick:

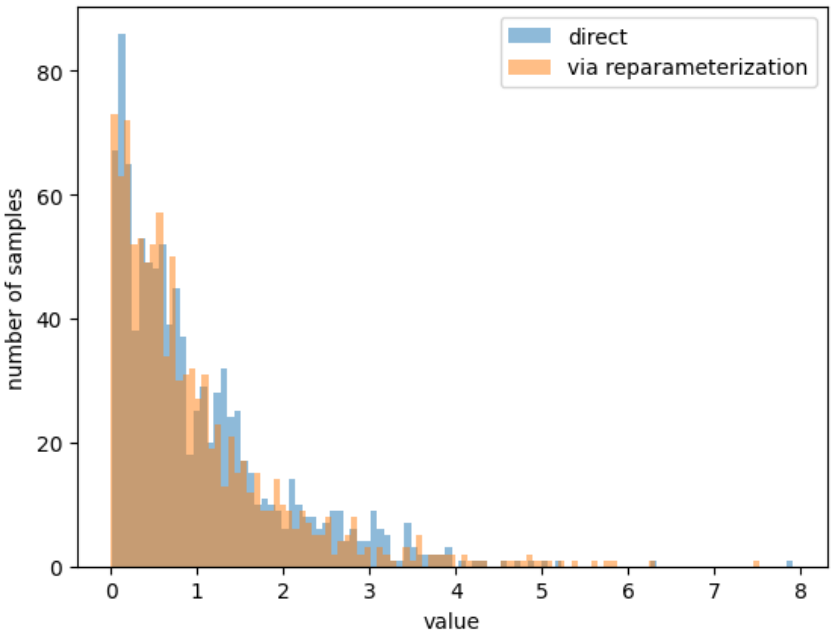
```
def inv_cdf_exp(l, samples) :
  samples = -torch.log(1-samples)/l
  return samples

def exp_reparametrize(l,N):
  # this function should return N samples via reparametrization,
  samples = Uniform(0,1).sample((N,))

  # reparametrization
  samples = inv_cdf_exp(l, samples)
  return samples
```

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

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




## *Q2. Categorical Distribution*

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

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

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

```
# 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
  samples = Gumbel(0,1).sample((N, a.shape[0]))

  # Add eps to prevent numerical instability in the log operation
  a = a + eps

  # reparametrization
  samples = F.softmax((torch.log(a)+samples)/temp, 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:

```
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)
reparametrized_samples = torch.argmax(reparametrized_samples, dim=1)
compare_samples(direct_samples, reparametrized_samples)
```