University of Paris-Saclay - UFR Sciences

Lab 1 report - Variational Autoencoder

Diyun LU - Marwan MASHRA - Yihan ZHONG

12 October 2022

1 Introduction

1.1 From AE to VAE

In order to introduce Variational Auto-Encoders(VAE), we will introduce The Auto-Encoders(AE) first. AE is consists of two parts: the Encoder and the Decoder. The encoder and decoder can be seen as two functions: 1). Encoder is for mapping high-dimensional inputs such as pictures to a latent representation of it; 2). Decoder will take the latent vector as inputs to create high-dimensional outputs such as the generated images. In deep learning, we usually use neural networks to learn these two functions.

The training process can be briefly described that we take one image X as input of Encoder, then we can obtain the latent vectors. We put the vector into Decoder to get a new generated image $\hat{X}$, then we minimize the difference between X and $\hat{X}$ (use MSE for example).

However, we can see that AE maps images to numeric encoding. The problem is that the model will be overfitting when the loss between the input and the output is continuously reduced during the training process. In addition, since the AE can only encode an image to a certain encoding, so when we input the encoding, we will always obtain the same image. If the encoding is from an unseen image, then we will obtain a substandard generated image, the generalization performance of AE is poor.

We can address the above problem by using VAE, which maps the input into a disribution instead of a numeric encoding. For example, we use Gaussian distribution, then the output of Encoder will be the means (μ_1,μ_2...μ_n) and standard deviations (σ_1,σ_2...σ_n). Then we sample from each normal distribution ($N(\mu_1$, ${\sigma_1}^2$), (N(μ_2, ${\sigma_2}^2$)...(N(μ_n, ${\sigma_n}^2$))$toget$(Z$_1$,Z_2...Z_n) which will be the input of Decoder.

VAE is an end to end learning, which means the input is the original data, and the output is the final result. There is no manual feature extraction requires. In the end-to-end network, the features can be learned by models automatically. Thus, the model will learn all the intermediate processes using the given data.

1.2 How VAE works

Assume that we have some samples, and we want to generate a new sample, the intuitive idea is to obtain the distribution $P(x)$ of the data. However, $P(x)$ is usually very complicated and really difficult to compute. Therefore, we assume it is a simple distribution such as a Gaussian distribution so that we can use maximum likelihood to calculate the parameters. However, the assumption is too naive in the real situation. Then we assume that $P(x)$ is a mixture of multiple Gaussian distributions (i.e.GMM) and a latent variable z satisfies the continuous Gaussian distributions. Thus, we can obtain the parameters μ, σ for $p(x|z)$. This is the essential idea of VAE. Since the true $p_\theta(x|z)$ is intractable, we let the variational approximate posterior be a multivariate Gaussian with a diagonal covariance structure:

$$log q_\phi(z|x^{(i)}) = log \mathcal{N}(z; \mu^{(i)}, \sigma^{2(i)} I) \tag{1}$$

The loss of VAE is consisted of two parts as shown below. The first part is a reconstruction loss that we would like to minimize the sample $\hat{x}$ and x; the second part is the KL divergence of distribution $q_\phi(z|x)$ and $p(z)$ as we want them to be as close as possible.

$$l(\theta, \phi) = -E_{z \sim q_\phi(z|x)}[\log q_\phi(x|z)] + KL[q_\phi(z|x)||p(z)] \tag{2}$$

Even though we have a loss function, our network still cannot be trained directly as two sampling operations are involved, and we cannot calculate gradients for sampling. The common mean-field theory approach needs analytical solution of expectations with respect to the approximate posterior. Therefore, the reparameterization trick is used. We will show more details in the implementation part.

In general, we assume that $p(z)$ is a standard normal distribution, and $q_\phi(z|x)$ is a normal distribution, the parameters are computed by the neural network. We can also use other distributions. We have used Bernoulli distribution in our experiment. However, it did not perform well compared to normal distribution. The details are shown in later parts.

1.3 Cons and pros of VAE

As generative models, VAE is usually compared with GAN. The advantage of VAE is that there is an explicit way to assess the quality of the model such as log-likelihood or lower bound estimation.

On the other hand, the disadvantage of VAE is that it directly calculates the loss between the generated image and the original image instead of adversarial learning like GAN. Also, with the injected noise and imperfect reconstruction, the generated images by VAE are slightly blurry.

2 Comparison between Sigmoid Belief Networks and VAE

2.1 Aspects from generative stories

The sigmoid belief network is a generative model for binary data. It's taking the observed variables from $[0, 1]^d$, and the latent random variable from $[0, 1]^k$. In order to generate the data via ancestral sampling for SBN, we need to first sample from the latent variable distribution which is parameterized as:

$$P(y; a) = \prod_i \frac{\exp(yiai)}{1 + \exp(ai)}, a \in \mathbb{R}^k \tag{3}$$

While for VAE model, we generate the latent random variables $y \sim P(Y)$ without the parameters, and y can take any values from R. When we derive a lower bound on the evidence by introduce a proposal distribution $q_\phi(y|x)$, by the ELBO gap, we know that the objective is maximized the proposal distribution so that it is equal to the posterior distribution. By the Bayes rule, we have:

$$q(y|x) = p(y|x) = \frac{p(y)p(x|y)}{\sum_{y'} p(y')p(x|y')} \tag{4}$$

Where it requires summing over 2^k terms in the denominator for SBN. Therefore, there is no closed form solution in the Expectation step and Maximization step of EM as for the GMM model and the use of gradient descent is also impossible in this case. Instead, we use mean field theory to approximate the proposal distribution $q_\phi(y|x)$ in order to calculate the ELBO, the proposal distribution is defined as follows:

$$q_\phi(y|x) = \prod_{i=1}^{k} (\phi_i^{(x)})^{y_i} (1 - \phi^{(x)})_i^{1-y_i} \tag{5}$$

where $\phi^{(x)} \in [0, 1]^k$ are the parameters of the proposal distribution associated with each data point (observation) x. We can observe that they are independent Bernoullis. In the case of VAE model, it also introduces proposal distributions in order to solve the intractability of the log-likelihood. However, instead of introducing a proposal distribution per sample, it uses an amortized proposal distribution which is predicted by a neural network. The parameters of the proposal distribution $q(y|x)$ are shared for all data points. Assume y is continuous, we also do a mean field assumption on $q(y|x)$, where $q(yi|x)$ follows a one dimension Gaussian

distribution. But in the case of VAE, the output of the latent variable is a single vector, therefore the MFT is applied on all the elements of the vector. Unlike the SBN model, where the latent variable is always binary, VAE model can have different latent variables discrete or continuous.

For the training part, SBN uses an algorithm with 2 steps and do the iteration, which first find the optimal condition for ϕ and do gradient ascent on the ELBO wrt model parameters θ use variational methods, which means it derives an approximation functions that makes the ELBO tractable. VAE is an end-to-end learning, it uses a single back propagation to optimize over the parameters ϕ and θ.

For the generation part, the VAE uses a fixed prior $z \sim \mathcal{N}(0, 1)$, the optimization objective is to make the proposal distribution similar to the fixed prior. Therefore, we end up sampling from a fixed prior, which makes the gradient descent possible. However, in SBN, we sample from what we learn, then generate $p(x|z)$ from the prior $p(z)$.

2.2 Distribution families of VAE model

The variational autoencoder has three possible distribution families:

1. The family of distributions for the encoder $q_\phi(z|x)$ parameterized by ϕ
2. The family of distributions for the decoder $p_\theta(x|z)$ parameterized by θ
3. The family of distributions for prior $p(z)$

In order to do calculate the loss, we need to compute the gradient of the reconstruction term:

$$\nabla_\phi E_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] \tag{6}$$

We can notice that it is to calculate the gradient of decoder, however, taking the gradient wrt to the $q_\phi(z|x)$ is difficult. Since the expectation is taken wrt the distribution that we are trying to differentiate, we can't swap the gradient and the expectation. There are two methods that we have implemented in this lab in order to deal with the gradient of the proposal distribution. One is called reparameterization trick used for a Gaussian distribution, one is called score function gradient estimator used for a Bernoulli distribution of the latent space. We should also notice that the KL divergence is also calculated differently in these two cases. In the assumption of VAE, the prior $p(z)$ can be any distribution specified by the user as long as we can find a way to express the distribution in terms of the parameters emitted from the encoder. The transformation needs to be differentiable that's why we want to use reparameterization. We want to use the optimal distribution for generating the latent variables in order to generate observed data: $p_\theta(x, z) = p_\theta(x|z)p_\theta(z)$, with observed $x \in \mathbb{X}$, where $\mathbb{X}$ can be continuous or discrete.

3 Implementation

3.1 Neural Architecture

The general architecture of a VAE consists of two Neural Networks (Encoder, Decoder). Those two NN in VAE have different set of weights. , unlike in Auto-Encoders where weights sharing is common (the decoder's weights could be taken as the transpose of the encoder's weights, as we saw last year).

Generally, The encoder takes as input an image x from the dataset, in our case it's a binary vector from our observed binary space, with 0. for black pixels and 1. for white ones. And then outputs the parameters ϕ of the proposal distribution $q_\phi(y|x)$, depending on the type of latent space (continuous, binary), this could be one or several vectors. The decoder takes a latent vector sampled from the conditional distribution outputted by the encoder $y \sim q_\phi(y|x)$ as an input, learns the parameters θ of the approximated posterior distribution $P_\theta(X|y)$, and output the reconstructed logits x' (not the final reconstructed image).

For a more concrete and mathematical formulation of this architecture, we need to separate two cases of latent space :

3.1.1 Continuous latent space

In the case of a continuous latent space, both the prior and the proposal distributions are Gaussian distributions. This is a very common approach for a VAE. The encoder outputs two vectors, a vector of means $\mu \in \mathbb{R}^d$ and a vector of log of variance $log(\sigma^2) \in \mathbb{R}^d$.

Figure 1 illustrates visually the architecture of VAE with a continuous latent space.

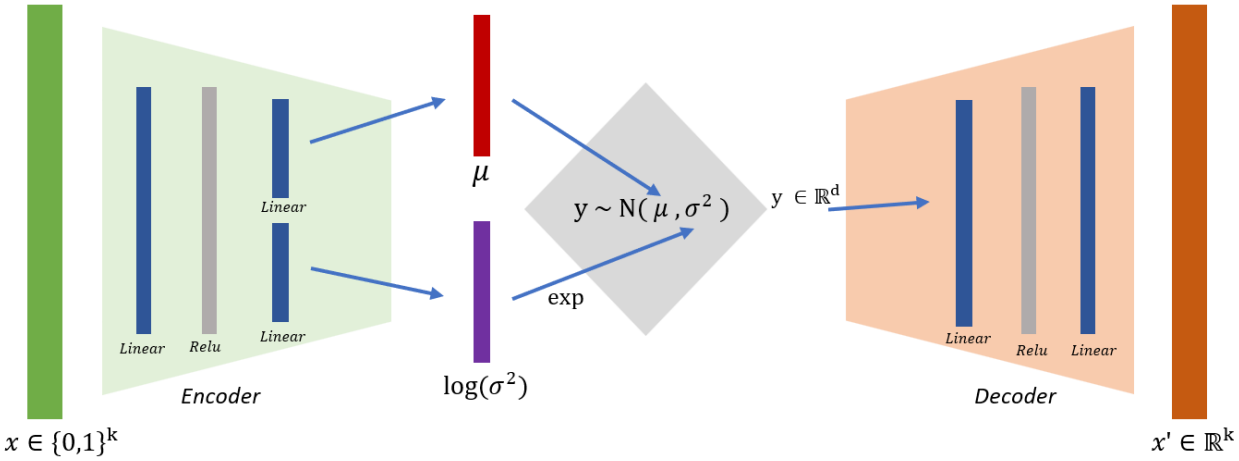


Figure 1: The architecture of VAE with a continuous latent space

Notice that we choose to output $log(\sigma^2)$ instead of σ^2. That's because the variance of a Gaussian distribution has several properties and constraints. To start, it has to be positive, we could use the *Relu* to enforce that, but its gradient is not well-defined around zero. In addition, the variance is usually very small values (between 0 and 1), so the optimization has to be done within a very small range of numbers. Choosing $log(\sigma^2)$ solves these problems and allows the encoder to output unconstrained vectors. We just need to remember to take the *exp* of it before sampling $y \sim \mathcal{N}(\mu, \sigma^2)$.

Now, let's take a look at the architecture of the encoder and decoder more formally.

3.1.1.1 Encoder

Let $x \in \{0,1\}^k$ be a data point (for our dataset, $k = 784$), and $z^{(i)} \in \mathbb{R}^h$ be the output of the hidden layer i (in our case, we only have one hidden layer with $h = 400$), and $\mu \in \mathbb{R}^d$, $log(\sigma^2) \in \mathbb{R}^d$ two unconstrained latent variables (in our case, we take $d = 2$), and $\phi = (W^{(1)}, W^{(2)}, W^{(3)}, b^{(1)}, b^{(2)}, b^{(3)})$ the parameters of our encoder (we have three linear projections layers in total), then :

$$z^{(1)} = Relu(W^{(1)}x + b^{(1)}) \tag{7}$$

$$\mu = W^{(2)}z^{(1)} + b^{(2)} \tag{8}$$

$$\log(\sigma^2) = W^{(3)}z^{(1)} + b^{(3)} \tag{9}$$

3.1.1.2 Decoder

Let $x' \in \mathbb{R}^k$ be reconstructed logits of the input image x, and $\theta = (W^{(4)}, W^{(5)}b^{(4)}, b^{(5)})$ the parameters of our decoder (we have two linear projections layers in total), $y \in \mathbb{R}^d$ the sampled continuous latent vector, then :

$$\sigma^2 = exp(\log(\sigma^2)) \tag{10}$$
$$y \sim \mathcal{N}(\mu, \sigma^2) \tag{11}$$
$$z^{(2)} = Relu(W^{(4)}y + b^{(4)}) \tag{12}$$
$$x' = W^{(5)}z^{(2)} + b^{(5)} \tag{13}$$

3.1.1.3 Generating images

To generate images once our decoder has been trained, we start by sampling from our fixed prior distribution, which is a centered Gaussian with variance of 1 :

$$y \sim \mathcal{N}(0, 1) \tag{14}$$

Then, we forward y through our decoder to obtain the reconstructed logits x':

$$x' = Decoder(y) \tag{15}$$

We then need to apply *sigmoid* to obtain the conditional output distribution ρ:

$$\rho = sigmoid(x') \tag{16}$$

Finally, to generate images, we can either draw pixels (1 and 0) from a Bernoulli distribution parametrized by ρ :

$$Image = Bernoulli(\rho) \tag{17}$$

Or simply use a threshold of 0.5 over ρ, i.e. a pixel i is white if $\rho_i \geq 0.5$, and black otherwise.

3.1.2 Binary latent space

In the case of a binary latent space, both the prior and the proposal distributions are Bernoulli distributions. This is a less common approach for a VAE, is more difficult to implement and produces worse quality generated images overall, as we'll see later in the report. The encoder outputs the vector $p \in \mathbb{R}^d$.

Figure 2 illustrates visually the architecture of VAE with a binary latent space.

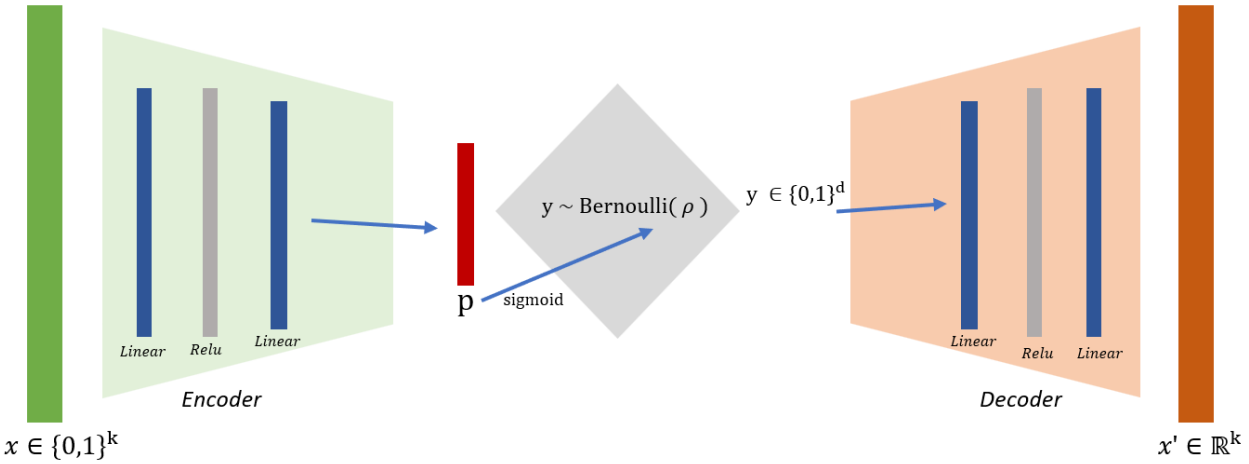


Figure 2: The architecture of VAE with a binary latent space

Notice that the vector p is unconstrained and thus, we'll need to apply *sigmoid* to obtain ρ in order to sample $y \sim Bernoulli(\rho)$.

Now, let's take a look at the architecture of the encoder and decoder more formally.

3.1.2.1 Encoder

Let $x \in \{0,1\}^k$ be a data point (for our dataset, $k = 784$), and $z^{(i)} \in \mathbb{R}^h$ be the output of the hidden layer i (in our case, we only have one hidden layer with $h = 400$), and $p \in \mathbb{R}^d$ two unconstrained latent variables (in our case, we take $d = 2$), and $\phi = (W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$ the parameters of our encoder (we have two linear projections layers in total), then :

$$z^{(1)} = Relu(W^{(1)}x + b^{(1)}) \tag{18}$$
$$p = W^{(2)}z^{(1)+}b^{(2)} \tag{19}$$
$$\tag{20}$$

3.1.2.2 Decoder

Let $x' \in \mathbb{R}^k$ be reconstructed logits of the input image x, $\rho \in [0,1]^h$ be a vector of probabilities, and $\theta = (W^{(4)}, W^{(5)}b^{(4)}, b^{(5)})$ the parameters of our decoder (we have two linear projections layers in total), $y \in \{0,1\}^d$ the sampled binary latent vector, then :

$$\rho = sigmoid(p) \tag{21}$$
$$y \sim Bernoulli(\rho) \tag{22}$$
$$z^{(2)} = Relu(W^{(4)}y + b^{(4)}) \tag{23}$$
$$x' = W^{(5)}z^{(2)} + b^{(5)} \tag{24}$$

3.1.2.3 Generating images

To generate images once our decoder has been trained, we start by sampling from our fixed prior distribution, which is a Bernoulli with a probability of 0.5 :

$$y \sim Bernoulli(0.5) \tag{25}$$

Then, we forward y through our decoder to obtain the reconstructed logits x':

$$x' = Decoder(y) \tag{26}$$

Once we have the reconstructed logits x', we can generate our images in the same way we did for the continuous latent space case.

3.2 Training

3.2.1 The Loss

Our goal during the training is to maximize the surrogate objective function

$$\max_{\theta,\phi} \frac{1}{|D|} \sum_{x \in D} ELBO(x, \theta, \phi) \tag{27}$$

Since we want to maximize the ELBO, our loss will be :

$$Loss = -ELBO(x, \theta, \phi) \tag{28}$$

The $ELBO(x, \theta, \phi)$ is defined as the difference between two terms :

$$ELBO(x, \theta, \phi) = \mathbb{E}_{y \sim q_\phi(y|x)}[logP_\theta(x|y)] - KL[q_\phi(y|x)||P(y)] \tag{29}$$

The first term is called the reconstruction term, and the second one is the kl divergence. Let's take a deeper look at each one of them :

3.2.1.1 Reconstruction term

The reconstruction term evaluates how similar the reconstructed (decoded) image is to the original one. We want to maximize that term. If we look back to the term, and compare it to the cross entropy between the proposal distribution and the approximated posterior distribution :

$$H(q,p) \;=\; -\,\mathbb{E}_{y\sim q_\phi(y|x)}[logP_\theta(x|y)] \tag{30}$$

We notice that they're the same, but with a negative sign. Therefore, to compute the reconstruction term in the code, we just compute the negative of the cross entropy using PyTorch :

```
reconstruction_term = - F.binary_cross_entropy_with_logits(
            reconstruction_logits,
            batch,
            reduction="none"
        )
```

and we specify reduction="none" because we want the function to return a vector of the same size, instead of the mean of all reconstruction losses for example.

3.2.1.2 Kullback–Leibler divergence

The Kullback–Leibler divergence (or KL divergence for short) allows us to measure how different those distributions are. We see is the kl divergence between the proposal (latent) distribution $q_\phi(y|x)$ and the prior (fixed) distribution $P(y)$. We're trying to minimize this term, i.e. making the proposal distribution as close as possible to the prior distribution, which will allow us to generate new images by simply sampling from the prior distribution and decoding them.

The KL divergence can be computed as :

$$KL[q_\phi(y|x)||P(y)] \;=\; \mathbb{E}[\log q_\phi(y|x) - \log P(y)] \tag{31}$$

So the KL divergence will depend on the prior we choose which depends on the type of latent space we have (continuous, binary). For the continuous case, we have $q_\phi(y|x) = \mathcal{N}(\mu, \sigma^2)$ and $P(y) = \mathcal{N}(0,1)$, and thus, the KL divergence can be computed as a function of μ and σ^2 :

$$KL[\mathcal{N}(\mu,^2)||\mathcal{N}(0,1)] \;=\; -\frac{1}{2}\sum_{i=1}^{d}(1 + log(\sigma_i^2(x)) - \mu_i^2(x) - \sigma_i^2(x)) \tag{32}$$

And for the binary case, we have $q_\phi(y|x) = Bernoulli(\rho)$ and $P(y) = Bernoulli(0.5)$, and thus, the KL divergence can be computed as a function of ρ :

$$KL[Bernoulli(\rho)||Bernoulli(0.5)] \;=\; \rho\log\left(\frac{\rho}{0.5}\right) + (1-\rho)\log\left(\frac{1-\rho}{0.5}\right) \tag{33}$$

and by simplifying this term, we can rewrite it using the entropy of a Bernoulli distribution :

$$KL[Bernoulli(\rho)||Bernoulli(0.5)] \;=\; \log(2) - H(\rho) \tag{34}$$

3.2.2 Gradient updates

Unlike training using EM (Expectation-maximization) in GMM or SBN, we can train the VAE using EM by updating both parameters θ and ϕ in the same iteration. An update during the $t+1$ iteration will look

like :

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta^{(t)}} \sum_{x \in D} \frac{1}{|D|} ELBO(x, \theta^{(t)}, \phi^{(t)}) \tag{35}$$

$$\phi^{(t+1)} = \phi^{(t)} - \eta \nabla_{\phi^{(t)}} \sum_{x \in D} \frac{1}{|D|} ELBO(x, \theta^{(t)}, \phi^{(t)}) \tag{36}$$

And this could be done easily in PyTorch by passing both the parameters of the encoder and the decoder to the optimizer. However, there is one issue when using backpropagation for the VAE.

3.2.3 Backprobagation issue

During backpropagation, we compute the gradient of the loss function with respect to each parameter in θ and ϕ to update it. The gradient is computed using the chain rule. since we have a sampling process between our encoder and decoder $y \sim q_\phi(y|x)$, we can't compute the gradient of y and thus, we can't backpropagate over all the loss function.

The Loss function contains two terms, the KL divergence which updates the parameters of the encoder ϕ and is computed before the sampling process. Therefore, there is no issue when computing the gradient of it. And the reconstruction term, which is used to update both the parameters of the encoder ϕ and the decoder θ, and is computed after the sampling process. Therefore, computing the gradient of the reconstruction term with respect to θ is not an issue, but only with respect to ϕ. There are two solutions to this problem, either change the sampling process to make y differentiable (The Reparameterization Trick), or add a new term to the loss function to update the parameters ϕ without needing before the sampling process and without changing the value of the loss (The score function estimator).

3.2.4 The Reparameterization Trick

Instead of sampling $y \sim \mathcal{N}(\mu, \sigma^2)$, we sample $\epsilon \sim \mathcal{N}(0, 1)$. Then y is just :

$$y = \mu + \sigma \odot \epsilon \tag{37}$$

This simple trick allows us to have $y \sim \mathcal{N}(\mu, \sigma^2)$ while being able to compute the gradient, and thus, backpropagate through the reconstruction loss.

3.2.5 Score function estimator

Since the parameters of the encoder ϕ can't be updated using the reconstruction loss as explained, we need to introduce a new term to the loss function that is computed before the sampling process and for which the gradient depends only on ϕ and not θ. As seen in the class, we can rewrite the gradient of the reconstruction loss to depend only on the gradient of the log proposal distribution :

$$\nabla_\phi \, \mathbb{E}_{y \sim q_\phi(y|x)}[logP_\theta(x|y)] = \mathbb{E}_{y \sim q_\phi(y|x)}[(\log P_\theta(x|y)) \nabla_\phi \log q_\phi(y|x)] \tag{38}$$

This trick is popular in reinforcement learning and often called reinforce estimator. By adding this term to our loss, we can update the parameters of the encoder ϕ without having to backpropagate through $P_\theta(x|y)$ which depends on the reconstruction logits obtained after sampling. To tell PyTorch not to backpropagate through $P_\theta(x|y)$, we can use the function *detach*. Furthermore, to not change the actual value of our loss, we subtract the reinforce loss and detach it. However, a Monte-Calro estimation of the Score Function Estimator has a very big variance, and thus, we try to reduce that variance by introducing a baseline $v(x)$ that depends only on x. We can now rewrite the formula as following :

$$\nabla_\phi \, \mathbb{E}_{y \sim q_\phi(y|x)}[logP_\theta(x|y)] = \mathbb{E}_{y \sim q_\phi(y|x)}[(\log P_\theta(x|y) - v(x)) \nabla_\phi \log q_\phi(y|x)] \tag{39}$$

There are different ways to take $v(x)$. for instance, we can use a regression model to predict it. In our case, we use a simple version of $v(x)$ that consists of the average (running average) of all past values of $\log P_\theta(x|y)$.

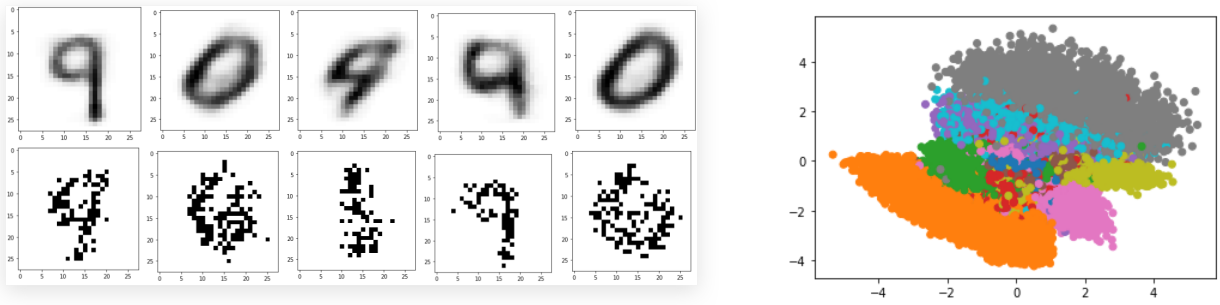
The final implementation looks like the following :

```
log_prob_p = reconstruction_loss.sum(dim=1) - running_avg
log_prob_q = -F.binary_cross_entropy(p, z, reduction='none').sum(dim=1, keepdims=False)
reinforce_loss = log_prob_p.detach()*log_prob_q - (log_prob_p*log_prob_q).detach()
loss = reconstruction_loss + kl_loss + reinforce_loss
....
n_updates += 1
running_avg += (reconstruction_loss.sum(dim=1).mean() - running_avg) / n_updates
```

3.2.6 Gradient clipping

This is an optimization trick and not mandatory. In VAE (and other deep networks), we sometimes face a problem called exploding gradient. It usually occurs when large error gradients accumulate, resulting in extremely large updates to our network weights during training. Very often, we use gradient clipping to deal with that problem. Clipping the gradient means setting a threshold. In addition, we can either clip by value, or by norm. In our case, we'll be experimenting with different values of clipping and observing the difference in the loss after several epochs.

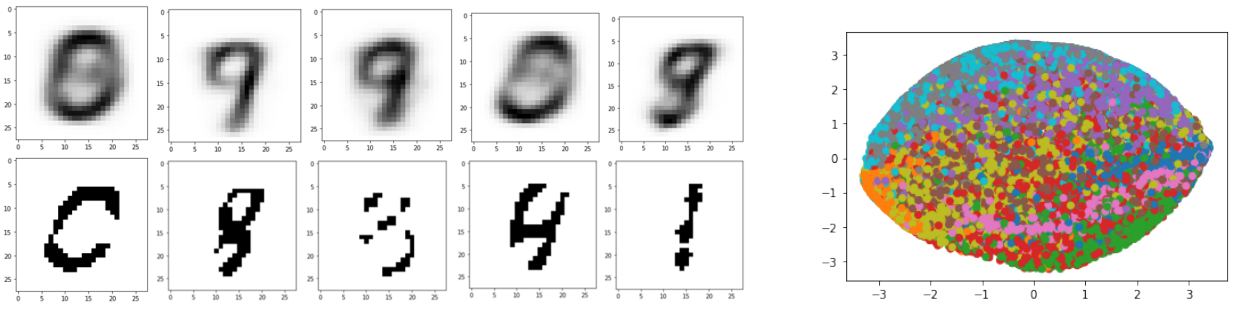
4 Results of our experiments



(a) The generated images

(b) The latent space

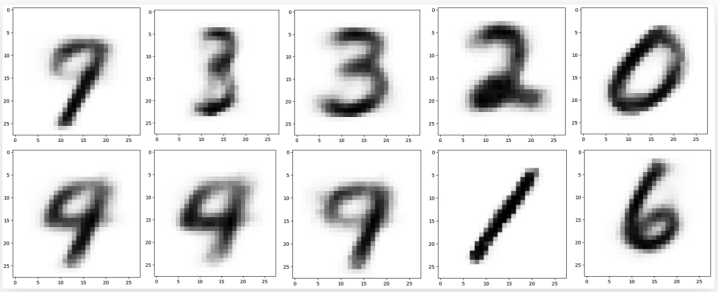
Figure 3: VAE with continuous latent space



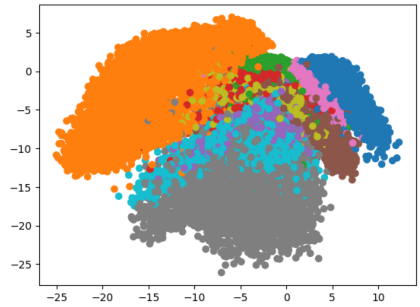
(a) The generated images

(b) The latent space (plotted using PCA)

Figure 4: VAE with binary latent space



(a) The generated images



(b) The latent space

Figure 5: Deterministic AE combined with GMM

From the result, we can notice that the distribution of the latent space of deterministic auto-encoder model looks similar to the one of VAE, with GMM, we can easily fit it with a 10 cluster and the new data that is generated from GMM model also has the similar quality compared to the one that is generated by VAE using Gaussian distribution.