Variational Autoencoder

Marwan MASHRA - Abdurrahman SHAHID

April 30, 2022

1 Introduction

Let's start by introducing what an Autoencoder (AE) is. We denote D, a domain in which a data point x lives. We can further assume, the data is N dimensional, i.e. $x^N \in D$. An AE is composed of two parts, en Encoder and a Decoder. The Encoder gets a data point x^N as input, and outputs a latent representation of it, given by a vector $z^n \quad \text{with} \quad n << N$. This latent vector z^N is then given as input to the Decoder, which outputs a reconstructed $\hat{x}$ with $\hat{x}$ a N dimensional vector. The Encoder maps an input from the input domain D to a latent representation having fewer dimensions. The goal of the Encoder, is to retain as much useful information as possible. Afterward, the Decoder tries to reconstruct the input using the latent vector z. As the dimension of z is lower than the dimension of the initial input x, the Decoder cannot reconstruct exactly x. There is inevitably a loss of information during the encoding phase. Hence, the Decoder's output $\hat{x}$ is not equal to x. The goal of the AE, is to decrease the discrepancy between the input x and the reconstructed ouptut $\hat{x}$.

Now that we have the big picture of what an AE is, we will present 3 use cases. First, an AE can be used as a dimensionality reduction technique. Indeed, the latent representation z usually has a very low dimension compared to the input x. Interestingly, one can prove that, an AE with only linear transformation and activation function, and a squared error loss function, is equivalent to Principal Component Analysis (PCA). Also, an AE can be used to separate useful signal from noise. Indeed, there exists what is called *denoising auto-encoder*, which takes as input the data of interest x plus a noise (here Gaussian) $e \sim \mathcal{N}(0, 1)$. The Decoder then tries to reconstruct only x and eliminate the noise e. Last but not least, an AE can be used to generate new data points. The Decoder sample a vector from the latent space, and generates a new data point $\hat{x}$. Thanks to the training setting, the generated data points $\hat{x}$ are very similar to the original data points x from the domain D. However, if the Decoder samples from regions of the latent space that were not used during the trainig, the generated data point $\hat{x}$ will be very different from data points one can found in D, $\hat{x}$ can even be altogether meaningless. To adress this shortcoming, we can use *Variational Autoencoders (VAE)*.

VAE works exactly the same, except for the latent space. The Encoder outputs parameters of a probability distribution. In our work, the Encoder outputs parameters of several Gaussian distributions. We aim to learn a joint Gaussian distriubtion. We thus have two latent vectors, one containig the means z_μ, and the other containig the log of the variances $z_{log(\sigma^2)}$. Moreover, the covariance matrix is diagonal. Thus, the vectors z_μ and $z_{log(\sigma^2)}$ are enough to entirely describe the joint distribution. We then sample from this joint distribution a multidimensional vector z, then pass it as input to the Decoder. The Decoder then, as usual, outputs a vector $\hat{x}$ from the latent representation z.

2 Neural architecture

A variational autoencoder is made of two different components: an encoder, and a decoder.

2.1 Encoder

The encoder takes as an input a data point x, forwards it through a deep network of one or several hidden layers, and outputs two vectors:

1. mean vector $\mu(x)$
2. vector of standard deviations $\sigma(x)$

More formally, let $x \in \mathbb{R}^D$ be a data point, d be the size of the latent space $\mathbb{Z}$. $\mu(x) \in \mathbb{R}^d$ and $\sigma(x) \in \mathbb{R}^d$ be the output vectors (means vector and standard deviations vector). $z^{(i)}$ be the output of the hidden layer i, and $W^{(i)}$,$b^{(i)}$ be the parameters of the layer. Then, the input x gets forwarded through the hidden layers :

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

$$\vdots \tag{2}$$

$$z^{(t)} = Relu(W^{(t)}z^{(t-1)} + b^{(t)}) \tag{3}$$

And the output vectors are the following:

$$\mu(x) = W^{(t+1)}z^{(t)} + b^{(t+1)} \tag{4}$$

$$\sigma(x) = W^{(t+2)}z^{(t)} + b^{(t+2)} \tag{5}$$

However, the standard deviation $\sigma(x)$ 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 standard deviation is usually very small values (between 0 and 1), so the optimization has to be done within a very small range of numbers. To deal with that, the encoder usually outputs the log-variance $log(\sigma^2(x))$ instead of the standard deviation. Taking the exponential will ensure that the value is positive. And having the square will allow for big values. So we rewrite 5 replacing the standard deviation by the log-variance:

$$log(\sigma^2(x)) = W^{(t+2)}z^{(t)} + b^{(t+2)} \tag{6}$$

2.2 Decoder

The decoder takes as an input a latent vector $z \in \mathbb{R}^d$ and outputs $\hat{x}$ a reconstructed version of x. Its goal is to have x and $\hat{x}$ be as close as possible, or in other words, minimize the reconstruction loss. It's also worth noting that the decoder shares the same parameters as the encoder, which makes the training much more efficient.

More formally, let $z \in \mathbb{R}^d$ be a latent vector sampled from the distribution $\mathcal{N}(\mu, \sigma^2)$ where μ and σ were outputted by the encoder. $\hat{x} \in \mathbb{R}^D$ be the output of the decoder (the decoded version of x). $\bar{z}(i)$ be the output of the hidden layer i. Then:

$$z \sim \mathcal{N}(\mu, \sigma^2) \tag{7}$$

$$\bar{z}^{(1)} = Relu(\bar{W}^{(1)}z + \bar{b}^{(1)}) \tag{8}$$

$$\vdots \tag{9}$$

$$\bar{z}^{(t)} = Relu(\bar{W}^{(t)}\bar{z}^{(t-1)} + \bar{b}^{(t)}) \tag{10}$$

And the output $\hat{x}$ is given by:

$$\hat{x} = \bar{W}^{(t+1)}\bar{z}^{(t)} + \bar{b}^{(t+1)} \tag{11}$$

However, since in our case, x and $\hat{x}$ are gray pixels of an image of the MNIST dataset, we want the values to be between 0 and 1 (gray scale pixels). So, we incorporate the sigmoid in 12 to obtain $\hat{x}$:

$$\hat{x} = Sigmoid(\bar{W}^{(t+1)}\bar{z}^{(t)} + \bar{b}^{(t+1)}) \tag{12}$$

3 Training method

In the training, variational Autoencoder could be seen as two deep neural networks stacked next to each other. Thus, we use backpropagation to train it (update the weights/bias to minimize the chosen loss function). The only problem that we might face using backpropagation is with the latent vector z. We said that z needs to be sampled from the distribution $\mathcal{N}(\mu, \sigma^2)$. However, the process of sampling from a distribution is not differentiable, and we can't compute the gradient of the loss with respect to z. To solve that probel, we use something called the Reparameterization Trick.

3.1 The Reparameterization Trick

Instead of sampling z from the distribution $\mathcal{N}(\mu, \sigma^2)$, we sample ϵ from a standard Gaussian distribution $\mathcal{N}(0, 1)$. Then z is just :

$$z = \mu + \sigma \cdot \epsilon \tag{13}$$

This trick allows us to have $z \sim \mathcal{N}(\mu, \sigma^2)$ while being able to compute the gradient, and thus, use backpropagation.

3.2 Loss

The only thing left to do is to choose the loss function. For VAE, we have two lasses:

3.2.1 Reconstruction loss

The main goal is for $\hat{x}$ the reconstructed (decoded) output to be as close as possible to the original input x. So, we're trying to minimize the difference between x and $\hat{x}$, also known as the reconstruction loss (let's call it here RL). We compute this loss using :

$$RL(x, \hat{x}) = \sum_{i=1}^{D} (x_i - \hat{x_i})^2 \tag{14}$$

However, since z was sampled from a distribution, we can't compute the exact reconstruction loss because we can get different $\hat{x}$ for the same x. Therefore, we approximate the loss using Monte-Carlo estimation. We do N iteration in which we compute $\hat{x}$ and compute the mean of the losses. Let's denote $\hat{x}^{(j)}$ the decoded output in iteration j, then the reconstruction loss is approximated as following :

$$RL(x, \hat{x}) = \frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{D} (x_i - \hat{x_i}^{(j)})^2 \tag{15}$$

In addition, since we'll be using an SGD optimizer, the reconstruction loss will be computed as the mean of all reconstruction losses of elements of the batch. Let's say we have batches of size B, then the reconstruction loss is computed as following:

$$RL(x, \hat{x}) = \frac{1}{B} \frac{1}{N} \sum_{b=1}^{B} \sum_{j=1}^{N} \sum_{i=1}^{D} (x_i^{(b)} - \hat{x_i}^{(j)(b)})^2 \tag{16}$$

3.2.2 Kullback–Leibler divergence

Although our main goal is to have a distribution that allows us to reconstruct x with minimum loss, we also don't want the distribution computed by the encoder to be too far from a multivariate Gaussian distribution (our prior). But the question is why?

Let X be the data we're trying to model, $P(X)$ be the probability distribution of the data, z be the latent vector, $P(z)$ be the probability distribution of the latent vector, and $P(z|X)$ be the probability distribution that projects our data into latent space $\mathbb{Z}$. In a VAE, we're trying to infer $P(z)$ from $P(z|X)$. And since we don't have $P(z|X)$, we approximate it using $Q(z|X)$. So $P(z|X)$ is the well-behaved distribution (our prior), and $Q(z|X)$ is the distribution learned by the VAE that we want to stay close to our prior. As a consequence, we want to minimize the distance between the distribution computed by the encoder $Q(z|X)$ and our prior $P(z|X)$. Since the distance between two distribution can be computed through the KL divergence, we add it to our loss.

$$KL[Q(z|X)||P(z|X)] = E[\log Q(z|X) \log P(z|X)] \tag{17}$$

In our case, we take a multivariate Gaussian distribution as our prior, and then the KL divergence is given by :

$$KL[Q(z|X)||\mathcal{N}(0,1)] = -\frac{1}{2}\sum_{i=1}^{d}(1 + log(\sigma_i^2(x)) - \mu_i^2(x) - \sigma_i^2(x)) \tag{18}$$

since we'll be using an SGD optimizer, the KL divergence will be computed as the mean of all KL divergences of elements of the batch. Let's say we have batches of size B, then the KL divergence is computed as following:

$$KL = -\frac{1}{2}\frac{1}{B}\sum_{b=1}^{B}\sum_{i=1}^{d}(1 + log(\sigma_i^2(x_b)) - \mu_i^2(x_b) - \sigma_i^2(x_b)) \tag{19}$$

3.2.3 Loss function

Finally, the loss function will just be the sum of both the reconstruction loss and the KL divergence:

$$Loss = RL + KL \tag{20}$$

3.3 Gradient clipping

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 Experiments

We try to fine-tune three different hyperparameters:

4.1 Gradient clipping value

It seems that values between 0 and 1 are very commonly used. So, we wanted to test several values within that range and see how that effects the loss (Figure 1).

4.2 Gradient clipping norm

Gradient clipping could also be done by the norm instead of the value. So, we tried that and compared its results with clipping by value (Figure 2).

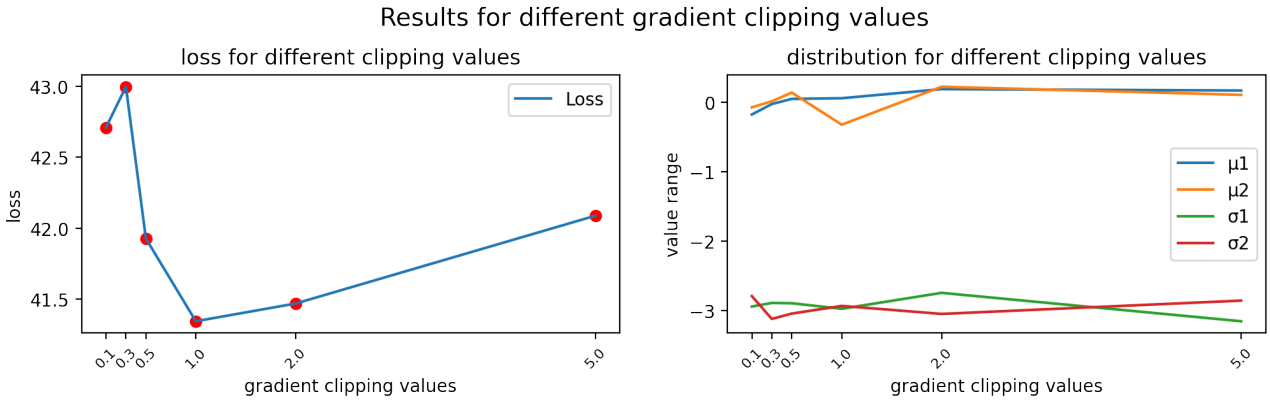

Figure 1: Experiment: Gradient clipping value

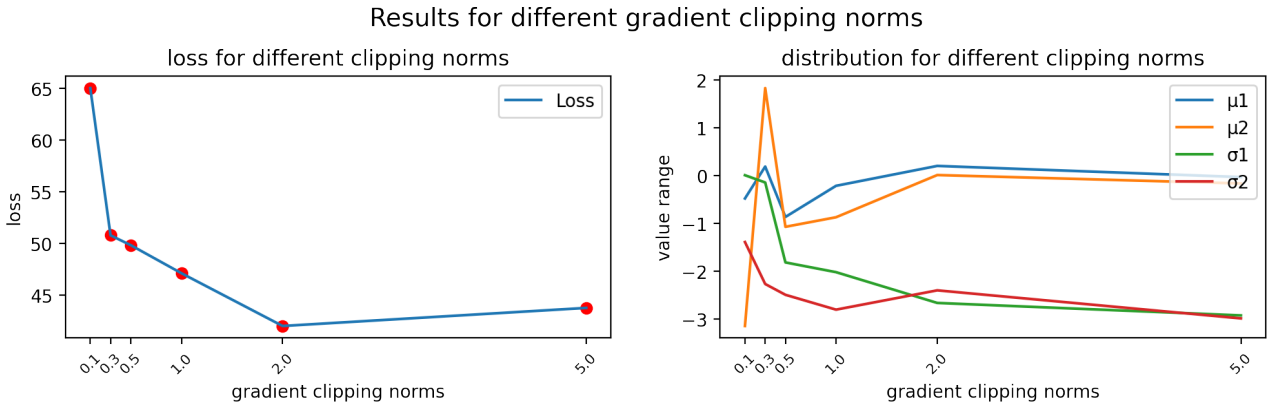

Figure 2: Experiment: Gradient clipping norm

4.3 Momentum

SGD with Momentum, is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent. This helps to converge faster. A typical value is around 0.9, but we wanted to test several values and see how does that affect the results (out of curiosity) (Figure 3).

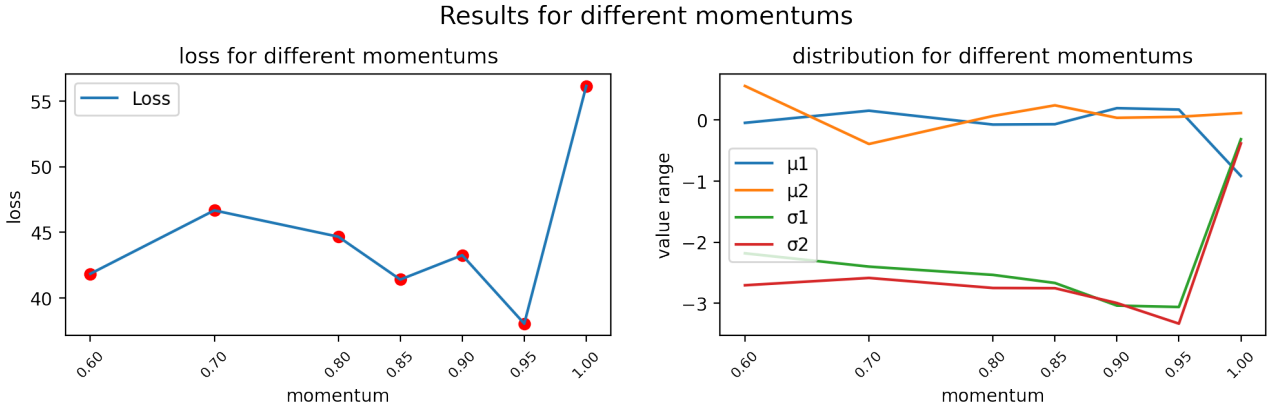

Figure 3: Experiment: Momentum

5 Results analysis

Looking at the results of the three previous experiments, we can conclude that gradient clipping by the value of 1 gave the best results among all others in experiment 1 and 2. In addition, sticking to a momentum around 0.9 seems to be the way to go. Although there are small differences between momentum in range 0.8 and 0.95, we stick to 0.9 to not overfit our data. Figure 4 shows some images generated by our best model.

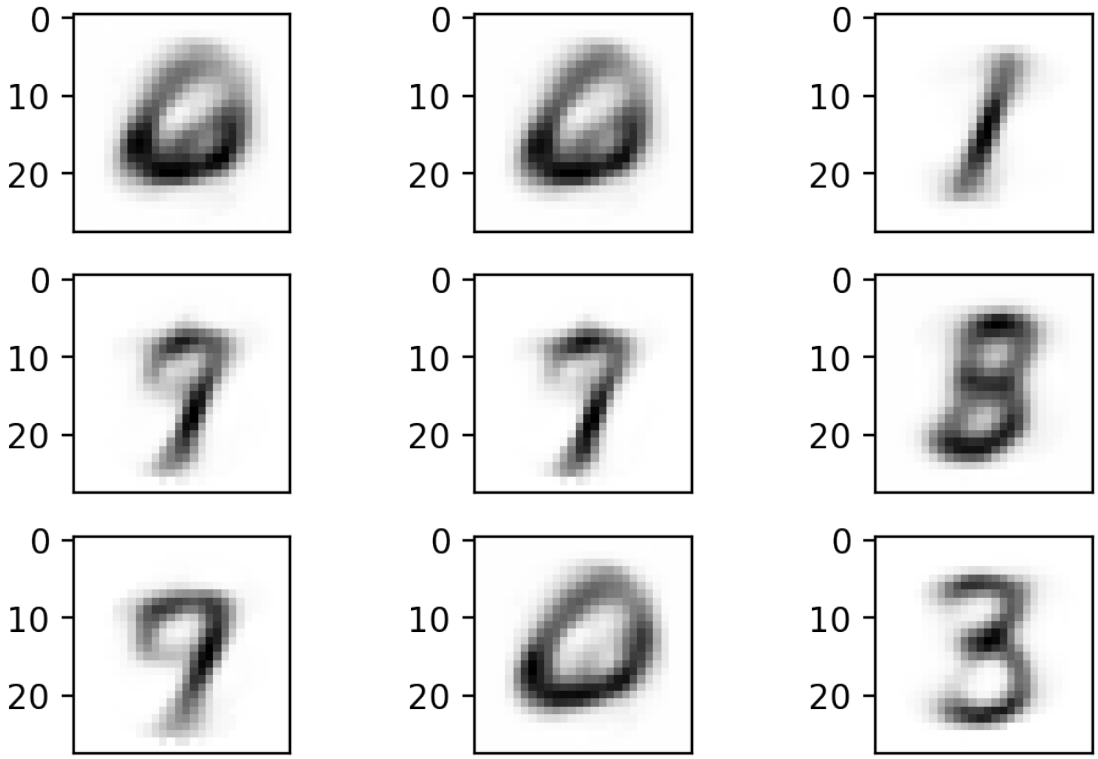

Figure 4: Generated images by our best model