Deep Learning Lab Exercise 4 Report

Julien Rolland
Alexandre Combeau
Gaëtan Serré
{first name.surname}@universite-paris-saclay.fr

Contents

1 Introduction

This project aims to implement a Variational Auto Encoder (VAE) in order to generate images very similar to those of the MNIST dataset[2].
In order to define what a VAE is, it is necessary to define an Auto Encoder (AE). An AE is a neural network which consists in projecting data towards a space called latent, often of a lower dimension, without loss of information. The purpose of an AE is to compress data. It is composed of two neural networks: an encoder and a decoder. The first one encodes the data to the latent space. The second one decodes the previously encoded data from the latent space to the original data space. It is used only during training to ensure that the AE does not lose information during encoding. A VAE is also composed of an encoder and a decoder. However, the goal is not at all the same. Indeed, a VAE is a generative model, its goal is thus to be able to generate data close to those on which it has been trained. To do this, the encoder of a VAE, instead of learning to project a data to a latent space, will learn the parameters ϕ of a distribution q (in our case, a multivariate Gaussian distribution). The decoder will learn the parameters θ of a distribution p which, given an element following q_ϕ, generates the original data with a high probability. To create new data, we can simply sample a value following the encoder's distribution and to pass it to the decoder.
For this project, we will use the Pytorch library[4] to create and train our VAE.

The overwhelming majority of the calculations and ideas come from the Deep Learning course taught by Caio Filippo Corro[1].

2 Formal description of the neural architectures

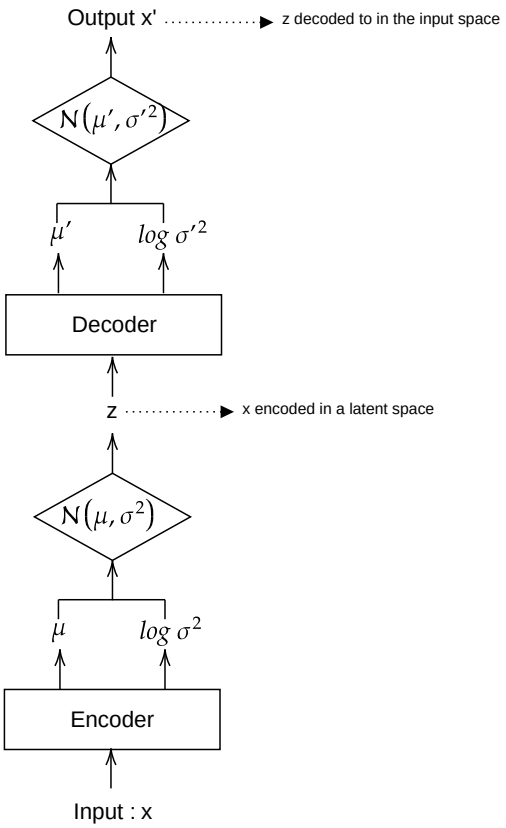


Figure 1: Formal view of our VAE architecture

The encoder and the decoder of a VAE have for objective to learn a couple of parameters μ and log σ^2 of a Gaussian distribution. They are neural networks with the same internal architecture. They can have any number of hidden layers, theses can be Linear or Convolutional for example. Both of networks have the same final output layers, 2 linear layers one for μ and the other for log σ^2. We use the μ and log σ^2 fitted by the encoder to sample z which will be the input for our decoder. Then, we use μ' and log σ'^2 to sample x' which will be the decoded data. We want x' to be close to the input data x. The μ and log σ^2 of the encoder and the decoder will be used to compute our loss. The main difference between the encoder and the decoder architecture is the shape of the data we give them, which in the end will give them an opposite objective.

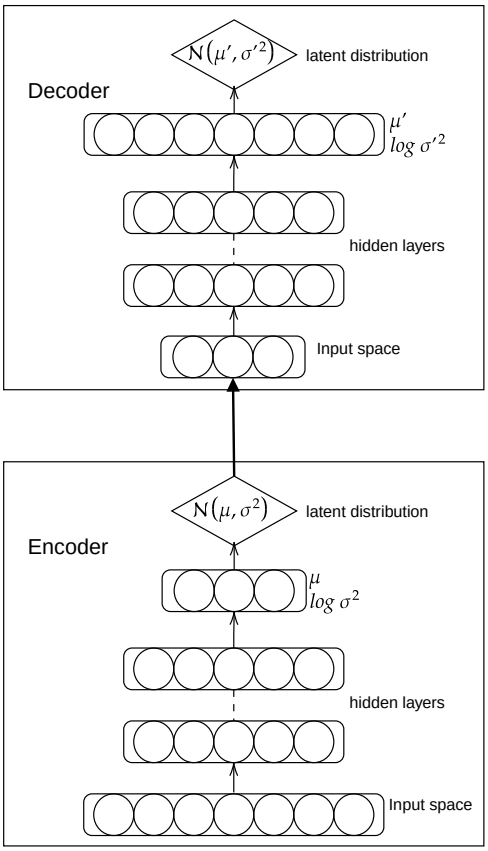


Figure 2: Zoom in the encoder/decoder of a VAE

Like any other neural network, we need to backpropagate the gradient of the loss to the decoder and the encoder of the VAE. The problem is that sampling a x' and a z are stochastic operations and therefore the gradient cannot be backpropagated. To solve this issue, we can use something called the "reparametrization trick".
It simply consists to say that:

$$z \sim \mathcal{N}\left(\mu, \sigma^2\right) \Leftrightarrow z = e \times \sigma + \mu \qquad \text{With } e \sim \mathcal{N}(0, 1)$$

By using this trick, we can now make the sampling of z and x' deterministic (given the same e) and backpropagate the gradient. The Figure 3 shows how the stochastic sampling node of the encoder is transformed using the reparametrization trick. We can use the same transformation for the sampling node of the decoder.

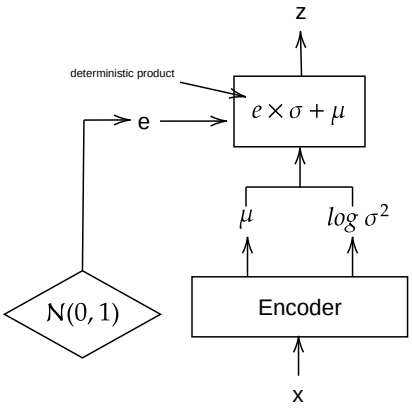


Figure 3: Reparametrization trick used with our Encoder

3 Objective function

To define the objective function of a VAE, we must start from the theory.
A VAE is a generative model so, given a dataset X, its goal is to approximate $p(X)$ using a distribution with parameters θ. Moreover, a VAE uses a latent distribution to encode the input data. Therefore $p_\theta(x)$ can be written:

$$p_\theta(x) = \mathbb{E}_{p_\theta}[p_\theta(z|x)] \tag{1}$$

With z representing a latent encoding of x.
However, $p_\theta(z|x)$ and consequently $p_\theta(x)$ are intractable. So we will have to make some approximations. First, we will introduce a prior distribution for $p_\theta(z)$. (which we will note $q(z)$) which will be, in our case, $\mathcal{N}(\vec{0}, \vec{1})$. Then, we will use the parameters ϕ of the VAE's encoder to approximate the posterior distribution $p_\theta(z|x)$:

$$q_\phi(z|x) \approx p_\theta(z|x)$$

The parameters θ of the VAE's decoder will be used to compute the likelihood distribution $p_\theta(x|z)$.
We can now define the two objectives of the cost function:

- Maximize $p_\theta(x)$ to ensure that the data sampled by this distribution is close to the to the original data.
- Minimize the Kullback-Leibler divergence between $q_\phi(z|x)$ and $p_\theta(z|x)$ in order to ensure that q_ϕ approximates the posterior distribution.

We can write the objective of a VAE as follows:

$$max_{\phi,\theta}\ \ \mathcal{L}(x, \phi, \theta) \tag{2}$$

Where

$$\mathcal{L}(x, \phi, \theta) = log\ p_\theta(x) - D_{KL}(q_\phi(z|x)\ ||\ p_\theta(z|x))$$

Also, we know that (proof 9):

$$D_{KL}(q_\phi(z|x)\ ||\ p_\theta(z|x)) = log\ p_\theta(x) - \mathbb{E}_{z\sim q_\phi}[log\ p_\theta(x|z)] + D_{KL}(q_\phi(z|x)\ ||\ q(z)) \tag{3}$$

Therefore

$$\mathcal{L}(x, \phi, \theta) = \mathbb{E}_{z \sim q_\phi}[log\ p_\theta(x|z)] - D_{KL}(q_\phi(z|x)\ ||\ q(z)) \tag{4}$$

Hence, the objective of a VAE is:

$$max_{\phi,\theta}\ \ \mathbb{E}_{z \sim q_\phi}[log\ p_\theta(x|z)] - D_{KL}(q_\phi(z|x)\ ||\ q(z)) \tag{5}$$

Maximizing $-D_{KL}(q_\phi(z|x)\ ||\ q(z))$ forces the distribution learned by the encoder to be close to $q(z)$, here $\mathcal{N}(\vec{0}, \vec{1})$. Maximizing $E_{z \sim p_\phi}[p_\theta(x|z)]$ maximizes the probability of decoding the original data x, given the encoder's q_ϕ distribution. This ensures that the decoder is able to recover the original data and that the encoding phase to the latent variable $z \sim q_\phi(z|x)$ does not cause any loss of information. This term is called the *reconstruction loss*.

4 Objective implementation

In our case, $q(z)$ is a multivariate Gaussian distribution $\mathcal{N}(\vec{0}, \vec{1})$. Thus, the ϕ and θ parameters learned respectively by the encoder and the decoder each represent a couple of parameters (μ, σ) of a multivariate Gaussian distribution. More precisely, the encoder and the decoder will learn a couple $(\mu, log\ \sigma^2)$ in order to avoid having a null variance (which would be the same as having a classical auto encoder). To compute $D_{KL}(q_{\mu,\sigma}(z|x)\ ||\ q(z))$, there is a closed form [3]:

$$D_{KL}(q_{\mu,\sigma}(z|x)\ ||\ q(z)) = -\frac{1}{2}\sum_{i=1}^{N} 1 + log(\sigma_i^2) - \mu_i^2 - \sigma_i^2 \tag{6}$$

However, there is no closed form for the reconstruction loss. We will therefore use a Monte Carlo approximation i.e. we will sample a z according to $q_{\mu,\sigma}(z|x)$ and say that:

$$\mathbb{E}_{z \sim q_{\mu,\sigma}}[log\ p_\theta(x|z)] \approx log\ p_\theta(x|z)$$

Moreover, instead of maximizing $log\ p_\theta(x|z)$, we will minimize the L2 loss function between the original data x and a sampled $\widehat{x}$ following $p_\theta(x|z)$:

$$L2(x, \widehat{x}) = \sum_{i=1}^{N} (x_i - \widehat{x}_i)^2$$

Thus, the function to be maximized becomes:

$$\mathcal{L}(x, \mu, \sigma, \theta) = -\sum_{i=1}^{N} (x_i - \widehat{x}_i)^2 + \frac{1}{2}\sum_{i=1}^{N} 1 + log(\sigma_i^2) - \mu_i^2 - \sigma_i^2 \tag{7}$$

This can be written with Pytorch:

```
reconstruction_loss = -((x.to(device) - x_hat) ** 2).sum() # -L2 loss
d_kl = -0.5 * (1 + log_sigma_squared - mu.pow(2) - log_sigma_squared.exp()).sum()
loss = -(reconstruction_loss - d_kl) # Pytorch only accepts loss function
```

In order to accelerate the training phase and to better generalize our VAE, we will train it with mini-batches.
This consists in giving a subset of data $X = \{x^{(i)}\}_{i=1}^{k} \subset Dataset$ to our VAE during the forward

pass. Then, the function to maximize during the backward pass becomes the average of the values of $\mathcal{L}$ over each $x \in X$:

$$\mathcal{L}(X, \mu, \sigma, \theta) = \frac{1}{k} \sum_{j=1}^{k} \mathcal{L}(x^{(j)}, \mu^{(j)}, \sigma^{(j)}, \theta^{(j)}) \tag{8}$$

That is, for each $x^{(i)} \in X$ corresponds a $\mu^{(i)}$, a $\sigma^{(i)}$ and a $\theta^{(i)}$. The cost function in Pytorch becomes:

```
reconstruction_loss = -((x.to(device) - x_hat) ** 2).sum(dim=1).mean() # -L2 loss
d_kl = -0.5 * (1 + log_sigma_squared - mu.pow(2) -
↪  log_sigma_squared.exp()).sum(dim=1).mean()
loss = -(reconstruction_loss - d_kl) # Pytorch only accepts loss function
```

Where *dim=1* means that the reconstruction loss and the Kullback-Leibler divergence is computed separately for each $x \in X$ and *.mean()* computes the average of these two terms over each $x \in X$.

We can now use a classical Pytorch training loop to train our VAE. To visualize its performance during training, we can display the value of the cost function at each epoch or compare images of our dataset with the corresponding images decoded by our VAE.

5 Results

This section develops the practical application of the points discussed in the previous sections. In addition to implementing the networks, the loss and the training loop, we have written some functions to visualize the different results.

For example a function to visualize the original images with their reconstructions during the training to better visualize the improvement of the network. This is shown in the figure 4.

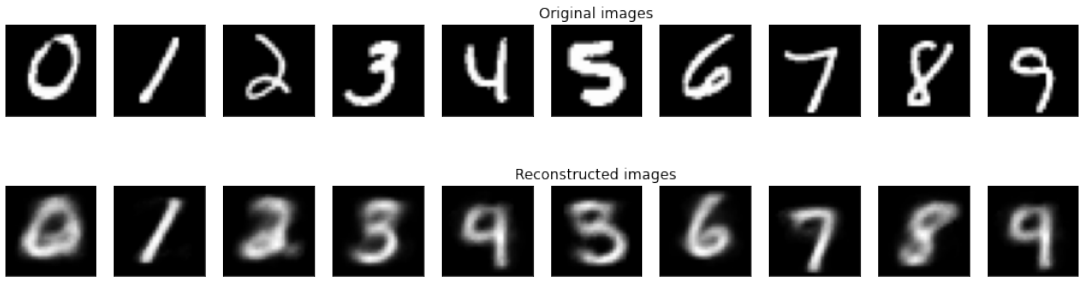
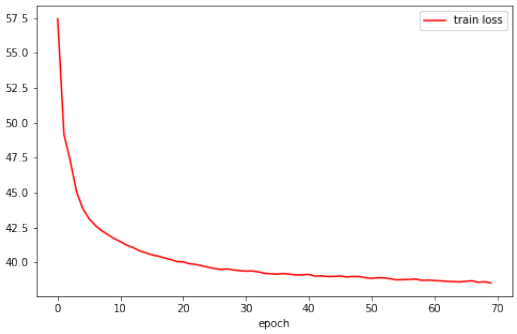


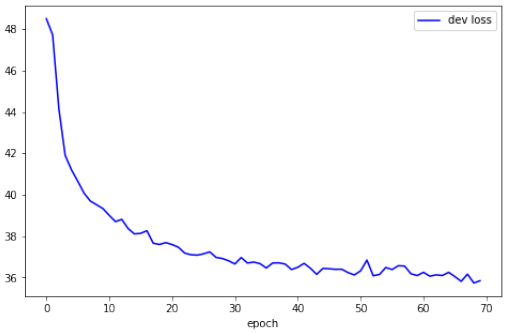
Figure 4: Original and reconstructed images

Although we have tried different networks and latent space sizes. The following figures have been realized with encoder/decoder having 3 hidden linear layers of 200 neurons with the *ReLu* activation function, and the dimension of the latent space fixed at 4. They were trained for 70 epochs with the Adam optimizer using a weight decay of 10^{-5}. We used 100 images per batch and a dropout of 0.3.

The evolution of the loss is as expected, its value is dropping on both the training and development set. Even though it does not seem to have stabilized completely after 70 epochs, which seems to indicate that we could still continue training to see more improvement. As shown in figure 5.



(a) Train loss



(b) Dev loss

Figure 5: Loss evolution over epochs

Once the network is trained, we can encode a large number of images in order to display the obtained vectors with a different color depending on their label. This can be done directly when the latent space has a small dimension, or by using the PCA of Scikit-learn [5] as it is the case for the figure 6. In both cases the results are similar. We can see that the different classes of the images are well separated in the latent space.

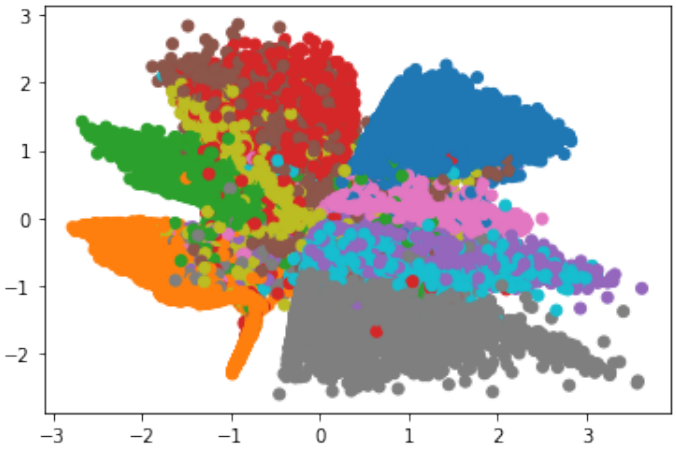


Figure 6: Latent space projected using PCA

With a trained network, we can also generate brand new images. We just have to take a random point of the latent space, typically a draw of $\mathcal{N}(\vec{0}, \vec{1})$, and decode it. Figure 7 contains 10 examples. We can see that the results are still very blurry but there are different "tricks" to make the images cleaner.

Figure 7: Randomly generated images

6 Experiments & Improvements ideas

We tried a lot of different values for the shape of the latent space, number of epochs, the number of hidden layers, their shape, the learning rate and the dropout. There was not a big difference between all of these configurations so we tried two things :

- We removed the sampling from the decoder to reduce the blur in the image. So instead of sampling an image using μ' and $log\ \sigma'^2$, we just returned μ'. As you can see in Figure 8, the images are a bit less blurry compared to the ones in Figure 7.

Figure 8: Generated images without sampling

- Since the encoder is manipulating images, we used convolutional layers in it. The loss is lower than when using an MLP (from 36 to 32 with 70 epochs). Also, as you can see in Figure 9, the resulting images are sharper to the ones in Figure 7 & 8. However, we used a implementation found on the internet (url available in the notebook) and we did not have the time to try other configurations.

Figure 9: Generated images using convolutional layers

We thought of several ideas that could improve our VAE:

- Try different optimizers
- Uses convolution hidden layers for the decoder
- Try different configurations for the convolutional layers (kernel size, padding, number of channels, pooling, ...)

7 Conclusion

The implementation of a variational auto encoder was our first experience with a generative model. It was a great challenge in both mathematics and computer science. Indeed, the mathematics behind VAEs is difficult to understand and the implementation of the loss in Pytorch is not trivial. However, thanks to our work and our knowledge in mathematics, deep learning and programming we were able to succeed in this project. It allowed us to understand how a VAE works and gave us the desire to learn more about generative models.

References

[1] Caio Filippo Corro. Deep learning course. 2022.

[2] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[3] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

A Appendix

Proof of 3

$$
\begin{aligned}
D_{KL}(q_\phi(z|x) \,||\, p_\theta(z|x)) &= \int_z q_\phi(z|x)\, log \frac{q_\phi(z|x)}{p_\theta(z|x)}\, dz \\
&= \int_z q_\phi(z|x)\, log \frac{q_\phi(z|x) p_\theta(x)}{p_\theta(z,x)}\, dz \\
&= \int_z q_\phi(z|x)\, log \frac{q_\phi(z|x)}{p_\theta(z,x)}\, dz + \int_z q_\phi(z|x)\, log\, p_\theta(x)\, dz \\
&= \int_z q_\phi(z|x)\, log \frac{q_\phi(z|x)}{p_\theta(x|z) p_\theta(z)}\, dz + log\, p_\theta(x) \\
&= \int_z q_\phi(z|x)\, log \frac{q_\phi(z|x)}{p_\theta(z)} - log\, p_\theta(x|z)\, dz + log\, p_\theta(x) \\
&= log\, p_\theta(x) + D_{KL}(q_\phi(z|x) \,||\, p_\theta(z)) - \mathbb{E}_{z\sim q_\phi}[log\, p_\theta(x|z)] \\
&= log\, p_\theta(x) + D_{KL}(q_\phi(z|x) \,||\, q(z)) - \mathbb{E}_{z\sim q_\phi}[log\, p_\theta(x|z)]
\end{aligned}
\tag{9}
$$