3MD3020: Deep Learning
MVA Master

Assignment 1

Samuel Diai

Due: January 22, 2020

A. Computer Vision Part

0.1 Latent Variable Models

0.2 Decoder: The Generative Part of the VAE

Question 1 [5 points]
In our setting, the model can be represented by a probabilistic graphical model. As there are no interactions between the neighbours pixels, the model simply translates into an ensemble of n pairwise nodes. For each pair, the node z_n is linked to the node x_n. And we have

$$p(x_n, z_n) = p(x_n|z_n)p(z_n)$$

Thus, we can sample from this model with a simple method : the Ancestral sampling. We want to draw a sample from the joint distribution $p(x_n, z_n)$. This methods works in general in directed acyclic graph (DAG) and our model is a very simple case of a DAG. In our case, for each pixel n, we sample a variable $\hat{z_n} \sim \mathcal{N}(0, I_D)$, then we sample a variable $\hat{x_n} \sim p(.|z_n = \hat{z_n}) = \prod_{m=1}^{M} \mathcal{B}(f_\theta(\hat{z_n})_m)$.

Moreover, as the pairwise nodes are independent from each other, we can sample a whole image x by sampling in parallel n pixel values x_n.

Question 2 [5 points]
Sampling $\log(p(x_n))$ using monte carlo as :

$$\log(p(x_n)) = \log(\mathbb{E}_{p(z_n)}[p(x_n|z_n)]) \approx \log \frac{1}{L} \sum_{l=1}^{L} p(x_n|z_n^l), \text{ where } z_n^l \sim p(z_n)$$

is not efficient. Indeed we have to draw L samples z_n^l for each image n, and for each of these sample l and image n, we have to compute $p(x_n|z_n^l) = \prod_{m=1}^{M} \mathcal{B}(x_n^{(m)}|f_\theta(z_n^l)_m)$, so this computation is quite expensive, we have to compute $N \times L$ neural networks evaluation $f_\theta(z_n)$ for $n \in [|1, L|]$, and for each of these evaluation we have to compute M Bernoulli terms. The global complexity is $\mathcal{O}(N \times L \times M)$.

The latter will have a bad effect on the computation of $p(x_n|z_n^l) = \prod_{m=1}^{M} \mathcal{B}(x_n^{(m)}|f_\theta(z_n^l)_m)$, this term is a product of M probabilities (inferior to 1). Then as M increases, this term will more likely be near 0.

Thus putting it back in the computation of $\log(p(x_n)) = \frac{1}{L}\sum_{l=1}^{L} p(x_n|z_n^l)$, this term will tend to be close to zero as M increases. And M cannot be tuned since it is the shape of the input images and this number is quite high in practice $M \approx 500 \times 500$.

0.3 KL Divergence

Question 3 [5 points]
In the univariate case, we have with $q = \mathcal{N}(\mu_q, \sigma_q)$ and $p = \mathcal{N}(\mu_p, \sigma_p)$,

$$D_{KL}(q||p) = \log(\frac{\sigma_p}{\sigma_q}) + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2}$$

Let's assume $\mu_p = \mu_q$ and we denote $x = \frac{\sigma_q}{\sigma_p}$ this reduces as :

$$D_{KL}(q||p) = -\log(x) + \frac{1}{2}(x^2 - 1)$$

We can plot this function and see that for $x = 1$ ie $\sigma_q = \sigma_p$ the divergence is zero, and for $x << 1$ or $x >> 1$, the divergence is very high and tends to $+\infty$.

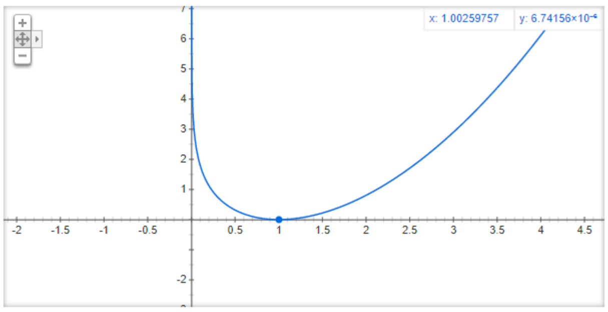


Figure 1: Plot of the function $x \mapsto -log(x) + \frac{1}{2}(x^2 - 1)$

In this example we recover that for any p and q distribution $D_{KL}(q||p) \geq 0$ with equality if $p = q$.

0.4 The Encoder: $q_\phi(z_n|x_n)$ - Efficiently evaluating the integral

Question 4 [5 points]
As we have seen before, $D_{KL}(q(Z|x_n)||p(Z|x_n)) \geq 0$, thus we can deduce from the inequality (15) :

$$\log p(x_n) \geq \mathbb{E}_{q(z_n|x_n)}[\log p(x_n|z_n)] - D_{KL}(q(Z|x_n)||p(Z))$$

It's impossible to maximize the log-likelihood directly and thus we need to optimize the lower-bound. Indeed the direct computation of $p(x_n)$ is possible in theory but intractable. We only know the joint distribution $p(x_n, z_n)$, and in order to compute

$$p(x_n) = \int_{z_n} p(x_n, z_n) = \int_{z_n} \left[\mathcal{N}(z_n; 0, I_D) \prod_{m=1}^{M} \mathcal{B}(x_n^m, f_\theta(z_n)_m) \right]$$

, we need to know f_θ.

Question 5 [5 points]

On the left hand side, we first notice that $\log p(x_n) \leq 0$ (as it is a probability) and $D_{KL}(q(Z|x_n)||p(Z|x_n)) \geq 0$.

Thus if the lower-bound is pushed up, that means either $\log p(x_n)$ increases or either $D_{KL}(q(Z|x_n)||p(Z|x_n))$ decreases. If $\log p(x_n)$ increases, that means we are maximizing the loglikelihood and that our model fits more and more our data.

On the other side, if $D_{KL}(q(Z|x_n)||p(Z|x_n))$ decreases, that means the distance between the probability distributions become closer, that is $q(Z|x_n)$ is close to $p(Z|x_n)$.

Thus, this method allows us to do two things at the same time : optimize the likelihood and approximate $p(Z|x_n)$ with a simple distribution $q(Z|x_n)$.

0.5 Specifying the Encoder $q_\phi(z_n|x_n)$

Question 6 [5 points]
We approximate the expectation $\mathbb{E}_{q_\phi(z|x_n)}(p_\theta(x_n|Z))$ using one sample $\hat{z}_n \sim q_\phi(.|x_n)$:

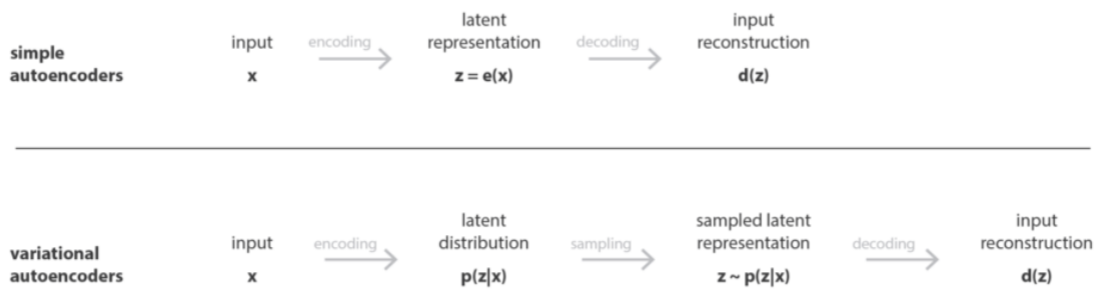
$$\mathbb{E}_{Z\sim q_\phi(z|x_n)}(p_\theta(x_n|Z)) \approx p_\theta(x_n|\hat{z}_n)$$

The loss become :

$$L = \frac{1}{N} \sum_{n=1}^{N} (L_n^{\text{recon}} + L_n^{\text{reg}}) = \frac{1}{N} \sum_{n=1}^{N} -p_\theta(x_n|\hat{z}_n) + D_{KL}(q(z_n|x_n)||p(z_n))$$

- For the reconstruction term, the goal of the probability $p_\theta(x_n|\hat{z}_n)$ is too measure how well x_n is reconstructed given the encoded probability distribution $q_\phi(.|x_n)$. That's why we want to maximize the expectation $\mathbb{E}_{Z\sim q_\phi(z|x_n)}(p_\theta(x_n|Z))$: we want that for every possible $Z \sim q_\phi(z|x_n)$, $p_\theta(x_n|Z)$ is high, that is given this Z, x_n is coherent.

- For the regularization term, the goal is to construct a representation of x in a latent space. In theory we want to construct a latent distribution $p(z|x)$ but as we have seen before, it is difficult to compute, therefore we use an approximate distribution $q(z|x)$.



simple autoencoders
input x
encoding
latent representation z = e(x)
decoding
input reconstruction d(z)
variational autoencoders
input x
encoding
latent distribution p(z|x)
sampling
sampled latent representation z ~ p(z|x)
decoding
input reconstruction d(z)

Figure 2: Difference between vaes and simple auto-encoders

As we can see on the picture above from this blog explaining the difference between VAE and standard auto-encoders, in simple auto-encoders, we encode the input a latent representation $z = e(x)$. The issue with this method is that the latent representation as **no regularity** : for two similar inputs, the corresponding latent representations can be far away. We solve this problem using VAEs by trying to learn a fixed distribution (Gaussian) instead of a scalar representation. That is why this term is called the regularization loss, when minimizing the KullBack Leibler element, $q(z_n|x_n)$ will become close to $p(z_n)$, and $p(z_n)$ is our fixed Gaussian distribution.

Question 7 [5 points]
We start by computing $D_{KL}(\mathcal{N}(\mu_1, \Sigma_1)||\mathcal{N}(\mu_2, \Sigma_2))$, the Kullback-Leibler divergence for two multivariate Gaussian distributions (of dimension d).
We have :

$$D_{KL}(\mathcal{N}(\mu_1, \Sigma_1)||\mathcal{N}(\mu_2, \Sigma_2)) = \int \left[\frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) + \frac{1}{2}(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)\right] \times p_1(x)dx$$
$$D_{KL}(\mathcal{N}(\mu_1, \Sigma_1)||\mathcal{N}(\mu_2, \Sigma_2)) = \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}\text{tr}\ \left\{\mathbb{E}_{p_1}[(x-\mu_1)(x-\mu_1)^T]\ \Sigma_1^{-1}\right\} + \frac{1}{2}\mathbb{E}_{p_1}[(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)]$$
$$D_{KL}(\mathcal{N}(\mu_1, \Sigma_1)||\mathcal{N}(\mu_2, \Sigma_2)) = \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}\text{tr}\ \{I_d\} + \frac{1}{2}(\mu_1-\mu_2)^T\Sigma_2^{-1}(\mu_1-\mu_2) + \frac{1}{2}\text{tr}\{\Sigma_2^{-1}\Sigma_1\}$$
$$D_{KL}(\mathcal{N}(\mu_1, \Sigma_1)||\mathcal{N}(\mu_2, \Sigma_2)) = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - d + \text{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right].$$

Thus in our case :

$$D_{KL}(q_\phi(Z|x_n)||p_\theta(Z)) = D_{KL}(\mathcal{N}(\mu_{\phi(x_n)}, \Sigma_{\phi(x_n)})||\mathcal{N}(0_D, \mathbb{1}_D))$$
$$D_{KL}(q_\phi(Z|x_n)||p_\theta(Z)) = \frac{1}{2}\left[\log\frac{|\mathbb{1}_D|}{|\Sigma_{\phi(x_n)}|} - D + \text{tr}\{\mathbb{1}_D^{-1}\Sigma_{\phi(x_n)}\} + (0_D - \mu_{\phi(x_n)})^T\mathbb{1}_D^{-1}(0_D - \mu_{\phi(x_n)})\right]$$
$$D_{KL}(q_\phi(Z|x_n)||p_\theta(Z)) = \frac{1}{2}\left[\log\frac{1}{|\Sigma_{\phi(x_n)}|} - D + \text{tr}\{\Sigma_{\phi(x_n)}\} + \mu_{\phi(x_n)}^T\mu_{\phi(x_n)}\right]$$

Using that $\Sigma_{\phi(x_n)}$ is diagonal and PSD :

$$D_{KL}(q_\phi(Z|x_n)||p_\theta(Z)) = \frac{1}{2}\left[\sum_{i=1}^{D}\{-\log\Sigma_{\phi(x_n)_i} + \Sigma_{\phi(x_n)_i}\} - D + \mu_{\phi(x_n)}^T\mu_{\phi(x_n)}\right]$$

We also have :

$$-\mathbb{E}_{q_\phi(.|x_n)}[\log p_\theta(x_n|z_n)] \approx -\log p_\theta(x_n|\hat{z}_n), \text{ where } \hat{z}_n \sim q_\phi(.|x_n) \text{ using one sample Monte Carlo Estimation.}$$

$$-\mathbb{E}_{q_\phi(.|x_n)}[\log p_\theta(x_n|z_n)] \approx -\log \prod_{m=1}^{M} \mathcal{B}(x_n^m, f_\theta(\hat{z}_n)_m)$$

$$-\mathbb{E}_{q_\phi(.|x_n)}[\log p_\theta(x_n|z_n)] \approx -\log \prod_{m=1}^{M} f_\theta(\hat{z}_n)_m^{x_n^m}(1 - f_\theta(\hat{z}_n)_m)^{1-x_n^m}$$

$$-\mathbb{E}_{q_\phi(.|x_n)}[\log p_\theta(x_n|z_n)] \approx -\sum_{m=1}^{M} x_n^m \log f_\theta(\hat{z}_n)_m + (1 - x_n^m)\log(1 - f_\theta(\hat{z}_n)_m)$$

$$-\mathbb{E}_{q_\phi(.|x_n)}[\log p_\theta(x_n|z_n)] \approx \text{BCE}(x_n, f_\theta(\hat{z}_n)), \text{ where BCE is the binary cross entropy loss}$$

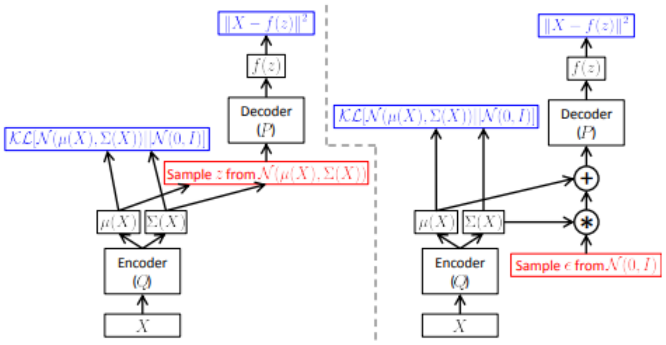
Finally :

$$L = \frac{1}{N}\sum_{n=1}^{N} L_n^{\text{recon}} + L_n^{\text{reg}}$$

$$L = \frac{1}{N}\sum_{n=1}^{N}\left[\text{BCE}(x_n, f_\theta(\hat{z}_n)) + \frac{1}{2}\left[\sum_{i=1}^{D}\{-\log \Sigma_{\phi(x_n)_i} + \Sigma_{\phi(x_n)_i}\} - D + \mu_{\phi(x_n)}^T \mu_{\phi(x_n)}\right]\right]$$

0.6 The Reparametrization Trick

Question 8 [5 points]



$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X),\Sigma(X))\|\mathcal{N}(0,I)]$
Sample z from $\mathcal{N}(\mu(X),\Sigma(X))$
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
X
$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X),\Sigma(X))\|\mathcal{N}(0,I)]$
$+$
$*$
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
Sample ϵ from $\mathcal{N}(0,I)$
X

Figure 3: figure from Carl Doersch tutorial

As described in the figure from Carl Doersch tutorial : we need to sample $\hat{z}_n \sim q_\phi(.|x_n)$ to compute the reconstruction loss. However this happen in the middle of the network and this operation is not differentiable. Indeed, in order to compute $\nabla_\phi L$, we need to compute $\nabla_\phi L_n^{\text{recon}}$ (the other terms does not depend on ϕ). And we have :

$$\nabla_\phi L_n^{\text{recon}} \approx \nabla_\phi \text{BCE}(x_n, f_\theta(\hat{z}_n)) \text{ where } \hat{z_n} \sim q_\phi(.|x_n)$$

$$\nabla_\phi L_n^{\text{recon}} \approx \frac{\partial BCE}{\partial x_2}\nabla_\phi(f_\theta(\hat{z}_n))$$

$$\nabla_\phi L_n^{\text{recon}} \approx \frac{\partial BCE}{\partial x_2}\frac{\partial f_\theta}{\partial x}\nabla_\phi((\hat{z_n} \sim q_\phi(.|x_n))) \text{ where the gradient is taken on the sampling action.}$$

We can clearly see that the latter gradient does not exist.

However, if we do $\hat{z_n} = \mu_{\phi(x_n)} + \Sigma_{\phi(x_n)}^{1/2}\epsilon$ where $\epsilon \sim \mathcal{N}(0_D, \mathbb{1}_D)$, we still have : $\hat{z_n} \sim q_\phi(.|x_n)$ And using the chain rule we have :

$$\nabla_\phi L_n^{\text{recon}} \approx \nabla_\phi \text{BCE}(x_n, f_\theta(\mu_{\phi(x_n)} + \Sigma_{\phi(x_n)}^{1/2}\epsilon))$$

$$\nabla_\phi L_n^{\text{recon}} \approx \frac{\partial BCE}{\partial x_2}\nabla_\phi(f_\theta(\mu_{\phi(x_n)} + \Sigma_{\phi(x_n)}^{1/2}\epsilon))$$

$$\nabla_\phi L_n^{\text{recon}} \approx \frac{\partial BCE}{\partial x_2}\frac{\partial f_\theta}{\partial x}[\nabla_\phi(\mu_{\phi(x_n)}) + \epsilon\nabla_\phi(\Sigma_{\phi(x_n)}^{1/2})]$$

We can compute analytically $\frac{\partial BCE}{\partial x_2}$ and we can compute $\frac{\partial f_\theta}{\partial x}$, $\nabla_\phi(\mu_{\phi(x_n)})$, $\nabla_\phi(\Sigma^{1/2}_{\phi(x_n)})$ using the neural network gradient descent.

0.7 Putting things together: Building a VAE

Question 9 [5 points]
To build a Variational Autoencoder, I used the following architecture for the encoder :

- The encoder takes the images as input, then pass into a linear layer of shape (x_{dim}, h^1_{dim})
- ReLU layer
- linear layer of shape (h^1_{dim}, h^2_{dim})
- ReLU layer
- linear layer of shape (h^2_{dim}, $2z_{dim}$)

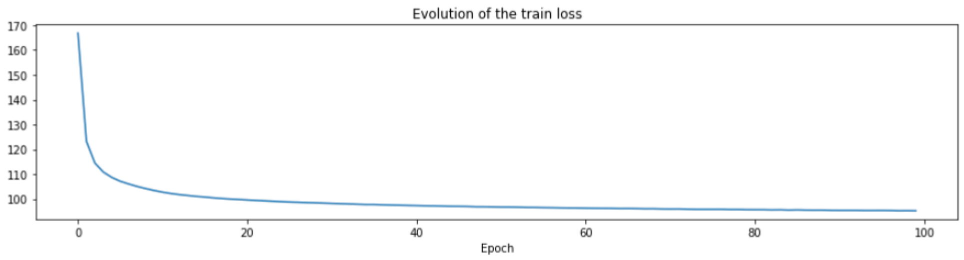
At the end of the encoder, the first part of the output (dimension z_{dim}) allows us to compute the mean μ_ϕ, and the second part of the output (dimension z_{dim}) allows us to compute the standard deviation (diagonal) $\Sigma\phi$.

For the decoder, I used the same reversed architecture :

- linear layer of shape of shape (z_{dim}, h^2_{dim})
- ReLU layer
- linear layer of shape of shape (h^2_{dim}, h^1_{dim})
- ReLU layer
- linear layer of shape of shape (h^1_{dim}, x_{dim})

The hyperparameters of my model are the dimensions h^1_{dim}, h^2_{dim}. I chose $h^1_{dim} = 512$, $h^2_{dim} = 256$. I trained on 100 epochs using the loss defined above : sum of the BCE loss and KL loss for the right multivariate gaussian distributions.

Question 10 [5 points]



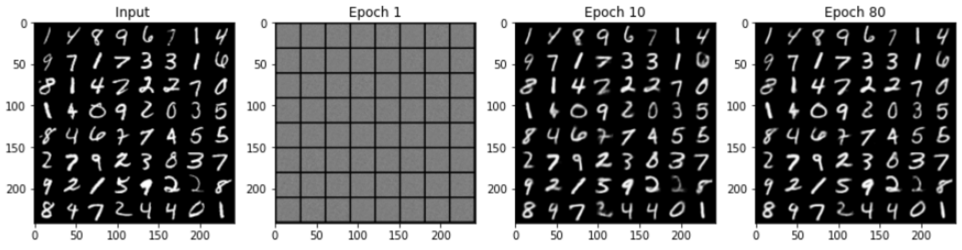


Figure 4: Results from the VAE-MLP model

As we can see on the picture, there is a significant improvement between the first epoch and the epoch 10. After the first epoch, the model is unable to draw good number representation whereas after the 10th epoch, we can see some clear digits

(9,1 ,5 etc ..). However the results are not perfect at the epoch 10. As we can see some digits are not well represented. For instance, the digit at the second row, last column does not represent a valid ”6” digit (as it should be) but rather a 0. But at the epoch 80, the same digit looks much like a 6. This means the training is not finished at the step 10, and some clear improvement can be made by continuing the epochs.We can notice more precisely the same phenomenon on the plot of the training loss.

After 100 epochs, the model reaches a training loss of 101.

(Optional) Question 11 [10 points] I also implemented a CNN VAE. As recommended in the U-Net architecture, I used transposed convolution for the decoder and convolution for the encoder. Transposed convolution is almost the same mathematical object than standard convolution, the difference is that it allows us to up scale the tensors instead of down scaling them (convolution).

To build a Variational Autoencoder, I used the following architecture for the encoder :

- The encoder takes the images as input, then pass into a 2d convolutional layer of shape (1, 16, ker dim = 3)
- Maxpool2d (size = 2)
- ReLU layer
- 2d convolutional layer of shape (16, 32, ker dim = 3)
- Maxpool2d (size = 2)
- ReLU layer
- 2d convolutional layer of shape (32, 64, ker dim = 3)
- Flatten layer

At the end of the encoder, the first part of the output (dimension z_{dim}) allows us to compute the mean μ_ϕ, and the second part of the output (dimension z_{dim}) allows us to compute the standard deviation (diagonal) $\Sigma\phi$.

For the decoder, I used the same reversed architecture :

- 2d transposed convolutional layer of shape (20, 16, ker dim = 3, stride = 2)
- ReLU layer
- 2d transposed convolutional layer of shape (16, 8, ker dim = 3, stride = 3)
- ReLU layer
- 2d transposed convolutional layer of shape (8, 4, ker dim = 3, stride = 2)
- ReLU layer
- 2d transposed convolutional layer of shape (4, 1, ker dim = 2, stride = 2)

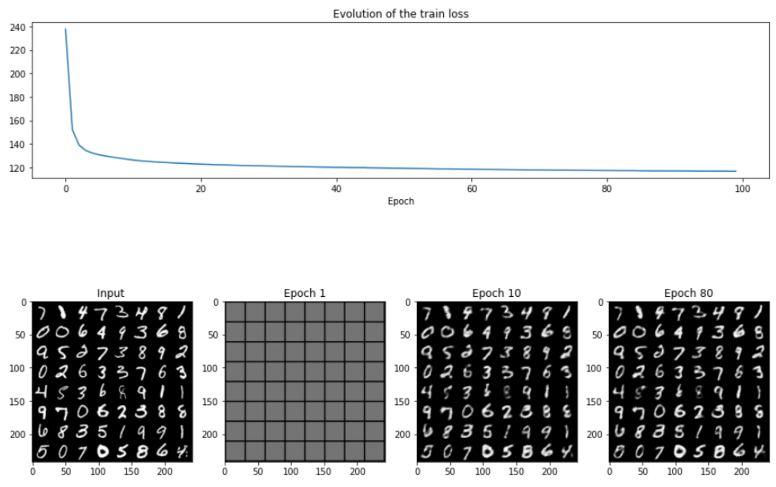


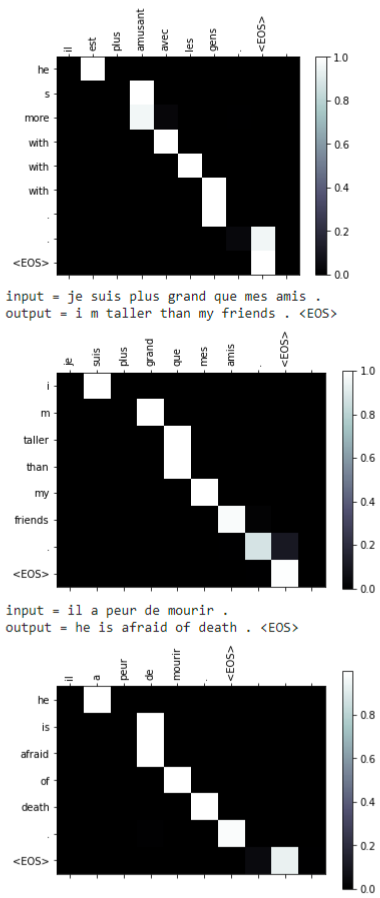
Figure 5: Results from the VAE-CNN model

The performance of this model is worse than the linear VAE model (127 vs 97 for the training loss). I think it's because the model is not complex enough to capture the latent probability distribution. With more layers, weights etc..., the performance should be much higher than the linear model, since convolutions are generally more suited for images.

B. Natural Language Processing Part

0.8 RNNs

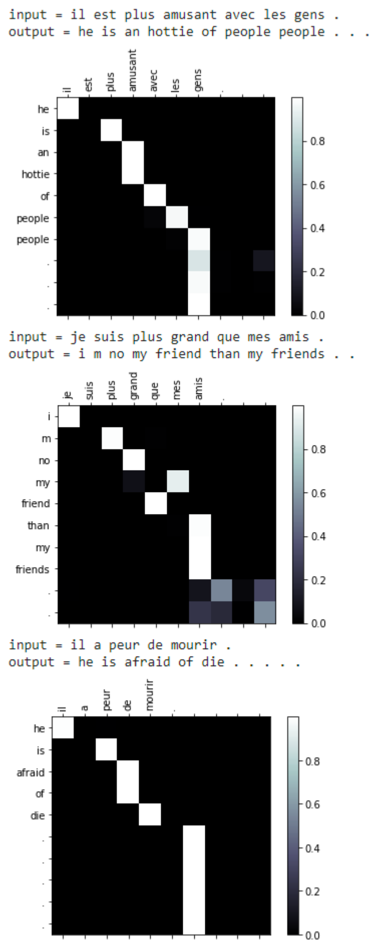
Question 12. [10 points]
I have run the pytorch tutorial and made two experiments : for the first one I use the code with the 'EOS' tokens and for the second one I remove the 'EOS' tokens from the training set. We can see bellow the results of the two experiments.



```
input = je suis plus grand que mes amis .
output = i m taller than my friends . <EOS>
```

```
input = il a peur de mourir .
output = he is afraid of death . <EOS>
```

Figure 6: With the EOS tokens



```
input = il est plus amusant avec les gens .
output = he is an hottie of people people . . .
```

```
input = je suis plus grand que mes amis .
output = i m no my friend than my friends . .
```

```
input = il a peur de mourir .
output = he is afraid of die . . . . .
```

Figure 7: Without the EOS tokens

For the experiment with the EOS token, the attention weights seem to not work properly. It is strange since the translation are correct for the second and the third example : "je suis plus grand que mes amis" > "i'm taller than my friends" and "il a peur de mourir" > "he is afraid of death". Indeed we see that the weight of the word "he" does not match the word "il" in the first example. There is a clear issue with the tutorial. However we can see that the tokens "EOS" are not systematically linked to each other. This is a good thing for sentences of different length. The shorter sentence's "EOS" is linked to a "." character. This allows the model to handle different input/output lengths.

Using this source, I found an attention map that comes from a properly trained model.

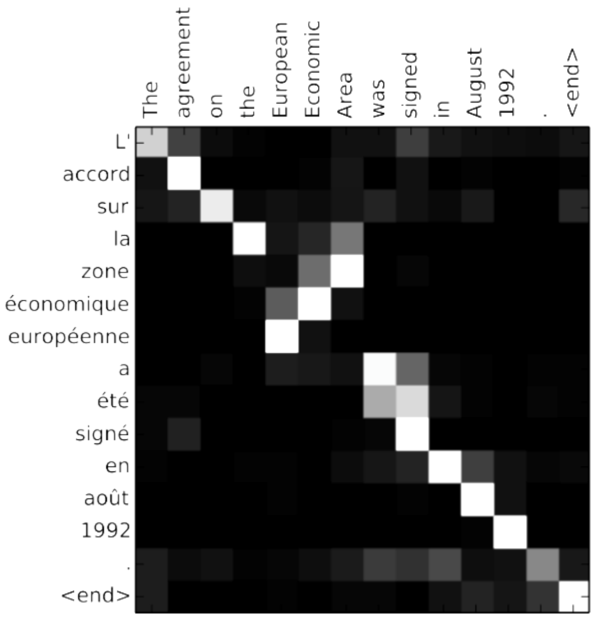


Figure 8: Proper Attention weights

This example seems to be much more accurate. As we can see, the weights are mainly diagonal since almost each word of the input in French corresponds to a word in English. However, we can notice that for the three words "zone économique européenne" which maps to "European Economic Area", the weights does not form a diagonal. Indeed, the order is reversed from french to English so that totally makes sense.

If we remove the "EOS" tokens, we can see the result of this experiment on figure 7. We can see that the translation are worse than the first experiment : "he is afraid of die" instead of "he is afraid of death". We can also notice that the decoder always puts the length of the output sentence to the max length $= 10$ (in the code). This comes from the lack of information of the decoder which does not know when to end the sentence.

Question 13. [5 points]
Attention is essential in this model as it helps the model to focus in some parts of the input sequence to predict the next word. Indeed, by definition, the attention mechanism allows the model to use all the hidden states of the input sequence, and make it select which ones will be reliable to use for the next prediction. Without attention, the decoder will only have access to the last hidden state, thus, it would reduce the performance of the model since it will be difficult for it to accurately process long input sequences.

Question 14. [5 points]
Teacher forcing is a training strategy consisting of using the real target output words as an input of the decoder instead of using the decoder's prediction. During training, this allow the model to train faster, since at the early stages of training, the predictions of the model are usually very bad (the model is not trained yet). If we do not use Teacher Forcing, at the beginning of the training, the hidden states of the model will be updated by a sequence of wrong predictions and errors will accumulate. This will lead to a longer convergence. However, this method may imply poor performance or instability. Indeed, during testing, true data is unavailable and the RNN model will need to feed its own previous prediction back to itself for the next prediction. Therefore there is a discrepancy between training and testing, and this might lead to poor model performance and instability.

0.9 Transformers

Question 15. [15 points]
We can derive analytically the impact of the difference sequence of layers. Let's denote L the number of encoders, x_l for $l \in [|0, L|]$ the data available at the layer l. In particular, x_0 is the input (after embedding) of the network. We have using the chain rule :

$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L}\frac{\partial x_L}{\partial x_{L-1}}\cdots\frac{\partial x_1}{\partial x_0}$$
$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L}\prod_{l=0}^{L-1}\frac{\partial x_{l+1}}{\partial x_l}$$

In case if we do the Layer Norm after the Self Attention Layer (usual framework as described in the paper "Attention is all you need"), we have :

$$x_{l+1} = \text{LayerNorm}(x_l + \mathcal{A}(x_l))$$

So we have :

$$\frac{\partial x_{l+1}}{\partial x_l} = \frac{\partial \text{LayerNorm}}{\partial x}(x_l + \mathcal{A}(x_l)) * (1 + \frac{\partial \mathcal{A}}{\partial x}(x_l))$$

So we have for this method :

$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L} \prod_{l=0}^{L-1} \left[\frac{\partial \text{LayerNorm}}{\partial x}(x_l + \mathcal{A}(x_l)) * (1 + \frac{\partial \mathcal{A}}{\partial x}(x_l)) \right]$$

In case if we do the non-standard method ie :

$$x_{l+1} = x_l + \mathcal{A}(\text{LayerNorm}(x_l))$$

We have :

$$\frac{\partial x_{l+1}}{\partial x_l} = 1 + \frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l)) \frac{\partial \text{LayerNorm}}{\partial x}(x_l)$$

So we have :

$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L} \prod_{l=0}^{L-1} \left[1 + \frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l)) \frac{\partial \text{LayerNorm}}{\partial x}(x_l) \right]$$

Question 16. [15 points]

When using the equation (21) ie $x_{l+1} = \text{LayerNorm}(x_l + \mathcal{A}(x_l))$, we get using the previous question the following equation for the back-propagation :

$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L} \prod_{l=0}^{L-1} \left[\frac{\partial \text{LayerNorm}}{\partial x}(x_l + \mathcal{A}(x_l)) * (1 + \frac{\partial \mathcal{A}}{\partial x}(x_l)) \right]$$

The issue with this equation is in the product, each term $\frac{\partial \text{LayerNorm}}{\partial x}(x_l + \mathcal{A}(x_l))$ can become very small and hence, we have globally that $\frac{\partial e}{\partial x_0}$ can become very small. This is a vanishing gradient problem. This problem is exacerbated if we increase the number of layers of encoders. Thus, for very deep network, this can become a big issue because the weights will freeze after some epochs, since the gradients are very small.

On the other hand, if we use the method of using the LayerNorm **before** the activation, ie equation (21) : $x_{l+1} = x_l + \mathcal{A}(\text{LayerNorm}(x_l))$, we get the following equation for the back-propagation :

$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L} \prod_{l=0}^{L-1} \left[1 + \frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l)) \frac{\partial \text{LayerNorm}}{\partial x}(x_l) \right]$$

As we can see in the product, each term is $1 + \frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l)) \frac{\partial \text{LayerNorm}}{\partial x}(x_l)$. Thus even if $\frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l))$ or $\frac{\partial \text{LayerNorm}}{\partial x}(x_l)$ become very small, we still have the $+1$ term.

So if a single term $\frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l))$ or $\frac{\partial \text{LayerNorm}}{\partial x}(x_l)$ is close to zero, $1 + \frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l)) \frac{\partial \text{LayerNorm}}{\partial x}(x_l)$ is close to one, and the global product we will be :

$$\frac{\partial e}{\partial x_0} = \frac{\partial e}{\partial x_L} \prod_{l' \neq l} \left[1 + \frac{\partial \mathcal{A}}{\partial x}(\text{LayerNorm}(x_l)) \frac{\partial \text{LayerNorm}}{\partial x}(x_l) \right]$$

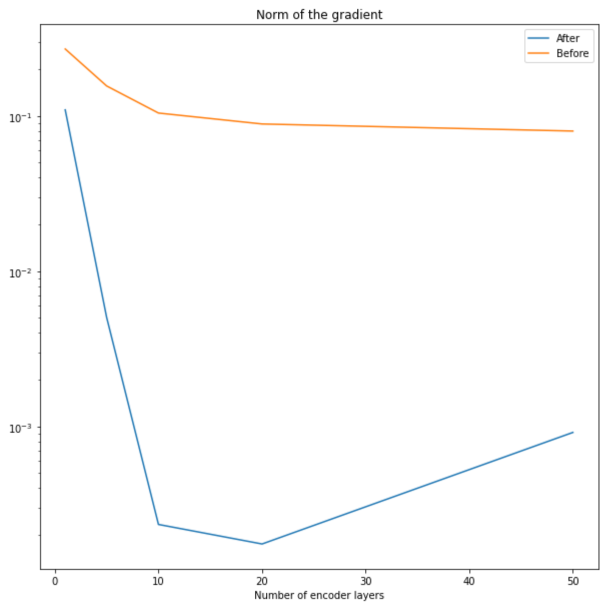
We will be able to effectively update the weights of the model, with non zero gradients.

(Optional) Question 17. [20 points]

For this question, I checked numerically the above result. In order, to do that, I edited the pytorch transformer tutorial, to work with the **before** or **after** normalization methods.

This tutorial presents a seq2seq basic transformer modeling network. In the tutorial, they train a TransformerEncoder model on a language modeling task. The language modeling task is to assign a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words. A sequence of tokens are passed to the embedding layer first, followed by a positional encoding layer to account for the order of the word. The TransformerEncoder consists of multiple layers of TransformerEncoderLayer. Along with the input sequence, a square attention mask is required because the self-attention layers in TransformerEncoder are only allowed to attend the earlier positions in the sequence. For the language modeling task, any tokens on the future positions should be masked. To have the actual words, the output of TransformerEncoder model is sent to the final Linear layer, which is followed by a log-Softmax function.

I edited the source code at the level of the definition of TransformerEncoderLayer to work with the **before** or **after** normalization methods. For each method, I compute the norm of the gradient at the level of the embedding. I plot the evolution of this norm as a function of the number of encoding layers.



As we can see on the graph, with the method where I apply the LayerNorm after the activation, the value of the norm of the gradient is very small ($\sim 10^{-4}$). On the other hand, with the method where I apply the LayerNorm before the activation, the magnitude is much higher ($\sim 10^{-1}$).