3MD3020: Deep Learning
MVA Master

Assignment 1 - Margaux Törnqvist

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 order to sample from the statistic model we will use **ancestral sampling**. Let's assume that for each image n, the variables $(x_n^{(1)}, ..x_n^{(m)})$ are indexed in a topological order i.e from ancestor to descendant.

Algorithm 1 Ancestral Sampling

```
  Ordered variables (x$_n^{(1)}, ..x_n^{(m)}$)**in a topological order**
  **for** $i \in 1..M$ **do**
    **if** x$_n^i$has no parent values **then**
      Sample from prior distribution
      Draw $x_n^i$ from $p(f_\theta(z_n^{(i)}))$
  **else**
    Retrieve samples of ancestors
    Sample a value for $x_n^i$using the conditional values sampled at former steps.
    Draw $x_n^i$ from p($x_n^i$/pa($x_n^i$)) where pa(x) denotes for "parents" of x.
  **end if**
**return** (x$_n^{(1)}, ..x_n^{(m)}$)
```

This method allows us to sample from the joint distribution p(($x_n^{(1)}, ..x_n^{(m)}$)).

Question 2 [5 points]

Although Monte-Carlo Integration can be used to approximate $logp(x_n)$ with samples $(z_n^{(1)}, ..z_n^{(m)})$ from $p(z_n)$ from the latent space, it is inefficient in training VAEs. Indeed, several problems appears.

1. **How do we know if the samples $(z_n^{(1)}, ..z_n^{(m)})$ have contributed to $p(x_n)$ i.e that the distribution $p(x_n/z_n)$ is close to $p(x_n)$?**
If we sample from any z_i's, how can we ensure that sampled x_n is likely to resemble to the desired output ? We need to ensure that $p(x_n/z_n)$ gives a positive probability to any realistic generated image x_n and not a probability close to zero. If we do not ensure this, it won't be back propagated through the gradients during the training and the output could be very different from input x_n.

2. **High dimensions of the latent space: difficulty to sample**.
If the latent space is big, it may be difficult to sample z. We will need to do a lot of samplings to get even one z_i.

0.3 KL Divergence

Question 3 [5 points]

For two univariate gaussian distributions p and q, you can show that :

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

With $(\mu_q, \mu_p, \sigma_q, \sigma_p)$ = (10,0,1,1), the gaussians are far from each other and the resulting KL-divergence is KL(q,p) = 100 which is very large.
With $(\mu_q, \mu_p, \sigma_q, \sigma_p)$ = (2,1,1,1), the distributions are relatively close to each other and the resulting KL-divergence is KL(q,p) = $\frac{1}{2}$, which is very small.
The KL-divergence quantitatively mesures how similar two distributions are.

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

Question 4 [5 points]

$logp(x_n) - KL(q(Z|x_n) \parallel p(Z|x_n)) = E_{q_\phi(Z|x_n)}log(p(x_n|z_n)) - KL(q(Z|x_n) \parallel p(Z))$(16)
The right term of equation 16 can be seen as a lower bound of $\log p(x_n)$ because :
1. The log -likelihood of $p(x_n|z_n)$ with z_n ∼q(./ x$_n$) can't be higher than the true searched value $\log p(x_n)$.
2. The term in KL can be seen a regularization term which ensures that $q(z_n|x_n)$ is closed to the targeted distribution $p(z_n)$.
We have seen in question 2, that sampling from $p(x_n)$ from the latent distribution z_n can be burdensome and difficult. Indeed too many z_i won't be likely to contribute to $p(x_n)$. Therefore, we need to target $z_n^{(i)}$'s which are likely to generate x_n. We do this by introducing a distribution q, where $q(.|x_n)$ gives positive probabilities to $z_n^{(i)}$'s likely to produce an output x_n. As this distribution targets our samples $z_n^{(i)}$, x_n is easier to sample. Therefore, instead of sampling $\log p(x_n)$ we chose to sample it's lower bound which includes a term where $z_n^{(i)}$'s are selected by q in order to maximize the likelihood to have a realistic x_n.
However, if Monte-Carlo Integration ensures that our lower bound is tractable, it doesn't imply it is differentiable, a fundamental operation needed for back-propagating the gradients during training, and thus easy to maximize. For choices of multivariate gaussians for q, the term in KL divergence becomes analytically solvable and can be minimized. As sampling isn't a differentiable operation, the term implying the decoder won't be able to update the gradients.

Question 5 [5 points]

When the lower bound is "pushed up", i.e maximized two things can happen:
1. $p(x_n)$ **is maximized**
2. **The KL distance between** $p(z_n|x_n)$ **and** $q(z_n|x_n)$ **is minimized**, i.e the true posterior $p(z_n|x_n)$ is increasingly better approached by $q(z_n|x_n)$.

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

Question 6 [5 points]

The loss can be written in two terms per-sample losses : a reconstruction and a regularization loss . The reconstruction loss ensures that x_n is well reconstructed from z, sampled from distribution $q(z|x_n)$.
The regularization loss ensures that for sample z, the distribution $q(z|x_n)$ is close to the targeted distribution $p(z)$.

Question 7 [5 points]

As equation (16) suggests, in order to minimize the gap between $logp(x_n)$ and the ELBO, we need to minimize the following D_{KL} term between the distribution q(Z—x$_n$)$andthetarget$p(Z—x$_n$) in f_θ, which is the neural network we are looking to parameterize in θ:

D_{KL}(q$_\phi(Z|x_n) \parallel p_\theta(Z|x_n))$

i.e we are looking to minimize in f_θ:

$$D_{KL}(q_\phi(Z|x_n) \parallel p_\theta(Z|x_n))$$
$$= E_{Z\sim q_\phi}logq_\phi(Z|x_n) - E_{Z\sim q_\phi}log(\frac{p(x_n|Z)p(Z)}{p(x_n)})$$
$$= E_{Z\sim q_\phi}log(q_\phi(Z|x_n)) - E_{Z\sim q_\phi}log(p_\theta(Z)) - E_{Z\sim q_\phi}log(p(x_n|Z)) + E_{Z\sim q_\phi}log(p(x_n))$$

Minimizing this expression is equivalent to minimize the following expression :

$$D_{KL}(q_\phi(Z|x_n) \parallel p_\theta(Z)) - E_{Z\sim q_\phi}log(p(x_n|Z))$$

Finally by noting :

$$L^{recon} = -E_{Z\sim q_\phi}(log(p(x_n|Z) \text{ and } L^{reg} = D_{KL}(q_\phi(Z|x_n) \parallel p_\theta(Z))$$

And averaging the loss over the N samples we have :

$$\text{L} = \frac{1}{N}\sum_{i=1}^{N}(L^{recon} + L^{reg})$$

We can go even further by developing the term L^{reg} in the loss.

We chose the following distribution for q and p:

$$\text{q}_\phi(z_n|x_n) = N(z_n/\mu_\phi(x_n, diag(\Sigma_\phi(x_n))$$
$$p(z_n) = N(0, I_D)$$

For two multivariate gaussians N$(\mu_q, \Sigma_q)N(\mu_p, \Sigma_p)$:
KL(N$(\mu_q, \Sigma_q), N(\mu_p, \Sigma_p)) = \frac{1}{2}(Tr(\Sigma_p^{-1}\Sigma_q) + (\mu_p - \mu_q)^T\Sigma_p^{-1}(\mu_p - \mu_q) - D + \log\frac{det\Sigma_p}{det\Sigma_q})$
with N$(\mu_p, \Sigma_p) = N(0, I_D)$
KL(N$(\mu_q, \Sigma_q), N(0, I_D)) = \frac{1}{2}(Tr(\Sigma_q) + \mu_q^T\mu_q - D - \log det\Sigma_q)$
So L$^{reg} = -\frac{1}{2}(Tr(\Sigma_q) + \mu_q^T\mu_q - D - \log det\Sigma_q)$

0.6 The Reparametrization Trick

Question 8 [5 points]

Let's compute the loss's gradient in ϕ :
$\nabla_\phi L = \frac{1}{N}\sum_{i=1}^{N}(\nabla_\phi L^{recon} + \nabla_\phi L^{reg})$
For both gradients we need to compute, we need to sample z with Monte-Carlo Integration from a distributin $q(z_n|x_n)$ Because of it's stochastic nature, this sampling method is a non-continous and non-differentiable function. This implies we can't compute the gradients and back-propagate them during the training of our network. Somehow, we know that stochastic gradient descent can handle stochastic inputs but not stochastic units i.e sampling transformations. What is called "the reparametrization trick" is a method which manages to passby this issue. It takes out the sampling operation out of the network, and introduce the samples to it as an input layer which can be used during stochastic gradient descent. We sample from $N(\mu_q, \Sigma_q)$ by first sampling from $N(0, \epsilon)$ with ϵ some noize, and then computing $z = \mu_(q) + \Sigma_q^{1/2} * \epsilon$. This trick allows us to work for a given noize ϵ with a lower bound which is deterministic, continous and differentiable. Back-propagation can now be done properly.

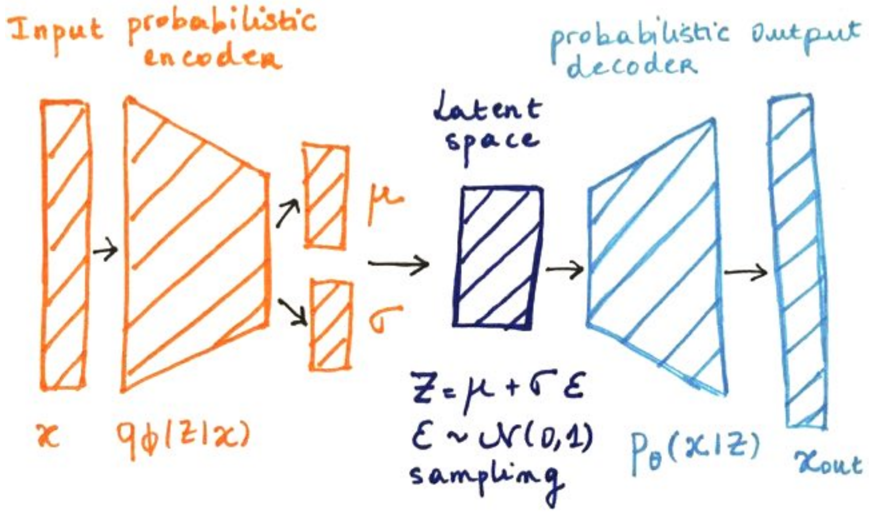


Figure 1: VAE architecture

0.7 Putting things together: Building a VAE

Question 9 [5 points]

VAE is constituted of an encoder which compresses information into a latent space dimension. A variable z is then sampled from this latent space. The decoder, tries to reconstruct the original input from the unique sample.

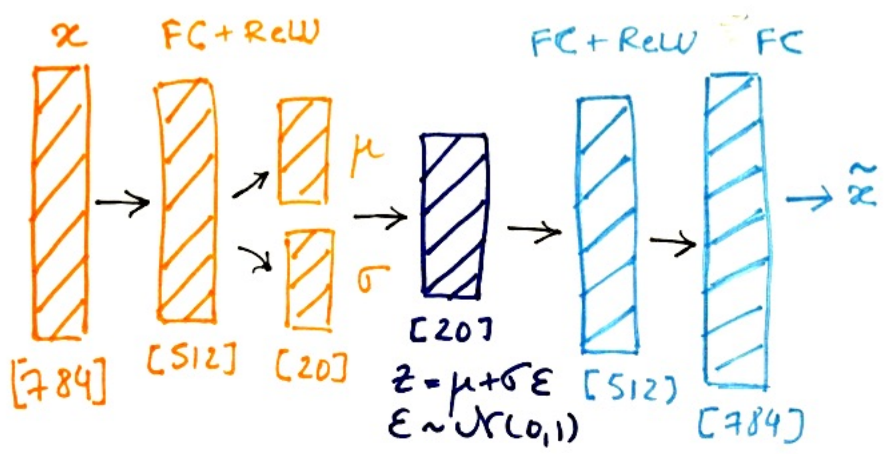


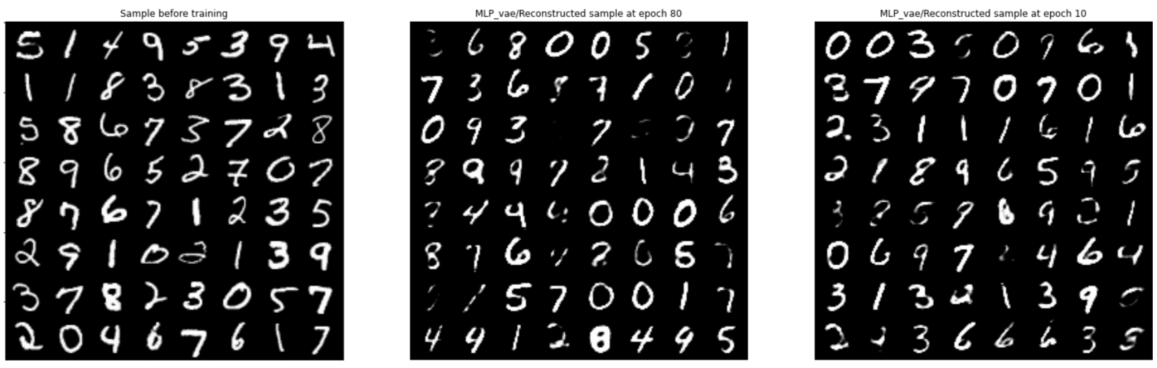
Figure 2: MLP VAE architecture

The MNIST dataset is composed of black and white images of size 28 x 28, which leads to an input of size 28 x 28 = 784. For the encoder, I used 1 fully connected layer with hidden layer $h_{dim1} = 512$, followed by a Relu activation layer. The latent space's dimension sizes 20. For the decoder, I used 2 fully connected layers with hidden layers h_{dim1} and output size 784. Only the first layer is followed by an activation Relu layer.

I trained the MLP vae by optimizing the loss which has a reconstruction term (binary crossentropy loss) and a regularization term (KL divergence) with Adam optimizer set with a initial learning rate of 1e-2.

I tried to augment the number of hidden layers, but this didn't improve performance.
Somehow, for a small network as ours, the results are still quite remarkable.

Question 10 [5 points]



Sample before training
MLP_vae/Reconstructed sample at epoch 80
MLP_vae/Reconstructed sample at epoch 10

Figure 3: Samples before and after reconstruction at epoch 10 and 80

In figure 1, is depicted samples at three points throughout training : before training, at epoch 10 and and at epoch 80. We notice that the reconstruction still presents alterations, and that there isn't a big difference between epoch 10 and epoch 80 which is surprising. I have tried to change learning rate but it hasn't improved results.

(Optional) Question 11 [10 points]

To build a convolutional vae, I designed a convolutional encoder and autoencoder. The encoder compresses information by feeding the input into two convolutional layers with kernel size 3 x 3, stride = 1, padding = 1 followed by a relu activation and a 2x2 maxpooling layer. The convolutional layers have respectively 32 and 64 channels. This convolutional block is followed by a fully connected layer which reduces dimensions first to 40 and then to 20, the latent space's dimensions.

After sampling mu and sigma from the latent space, we build a sample z which we want to look alike input x. Before feeding the sample to the decoder, we pass it through a fully connected layer and then reshape the tensor to (batch size, 64, 7, 7).

The decoder has 3 deconvolutional layers with kernel size 3x3, stride = 1, padding = 1 followed by a relu activation and 2x2 upsampling layer. The deconvolutional layers have respectively 32, 64, 32 and 1 channels.

Figure 2 illustrates the architecture of the vae.
I trained the convolutional vae by optimizing the loss which has a reconstruction term (binary crossentropy loss) and a regularization term (KL divergence) with Adam optimizer set with a initial learning rate of 1e-3.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
         MaxPool2d-2           [-1, 32, 14, 14]               0
            Conv2d-3           [-1, 64, 14, 14]          18,496
         MaxPool2d-4             [-1, 64, 7, 7]               0
           Flatten-5                 [-1, 3136]               0
            Linear-6                   [-1, 40]         125,480
            Linear-7                   [-1, 20]             820
            Linear-8                   [-1, 20]             820
            Linear-9                 [-1, 1568]          32,928
   ConvTranspose2d-10             [-1, 64, 7, 7]         18,496
          Upsample-11           [-1, 64, 14, 14]               0
   ConvTranspose2d-12           [-1, 32, 14, 14]          18,464
          Upsample-13           [-1, 32, 28, 28]               0
   ConvTranspose2d-14            [-1, 1, 28, 28]             289
================================================================
Total params: 216,113
Trainable params: 216,113
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.76
Params size (MB): 0.82
Estimated Total Size (MB): 1.59
----------------------------------------------------------------
```

Figure 4: Convolutional vae layer details

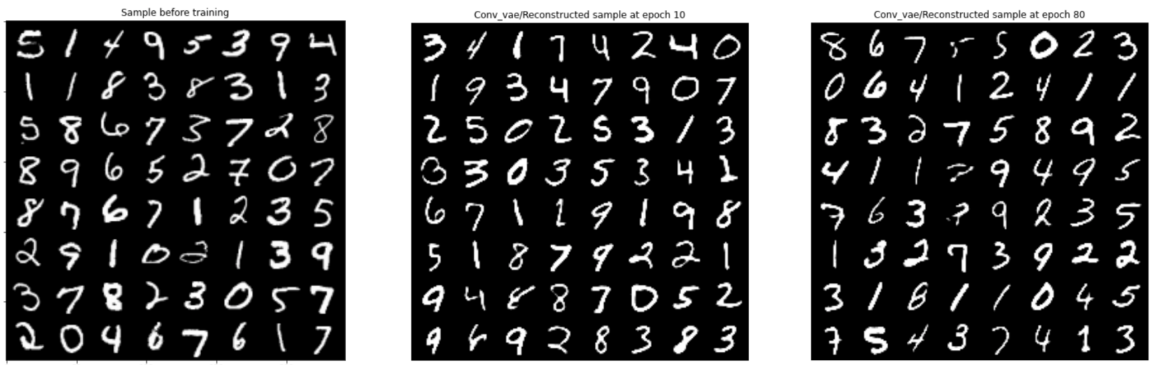

Sample before training
Conv_vae/Reconstructed sample at epoch 10
Conv_vae/Reconstructed sample at epoch 80

Figure 5: Samples before and after reconstruction at epoch 10 and 80

In figure 3, is depicted samples at three points throughout training : before training, at epoch 10 and and at epoch 80. We notice that the reconstruction is well better than for the MLP vae. Moreover, it still presents some alterations which shows that our vae isn't perfect. In order to improve our model, we could introduce noize at the sampling of z in order to help our model generalize and be more robust for reconstructing the input.

B. Natural Language Processing Part

0.8 RNNs

Question 12. [10 points]

After training the seq2seq model proposed by PyTorch's tutorial and testing it on a translation task, we'll analyze how attention is distributed along the tokens. In our model the attention layer is a vector of size maximum lenght of sequence 10. This implies that for short sentences the model will use only the first weights, while for longer sentences the model will use all of them.
Figure 4 depicts attention matrices from the tutorial for several inputs. Let's notice that there is an error in the construction of the attention matrices: there is a shift of 1 steps to the right after the first word. We take in consideration this shift for interpreting the matrices. First we notice that to translate the first word of the sequence, the model focuses only on the first word. This is due to the SOS character, given as input for the first instance, which specifies to the model that the sequence starts and that it can only focus on the first word to predict the next one.

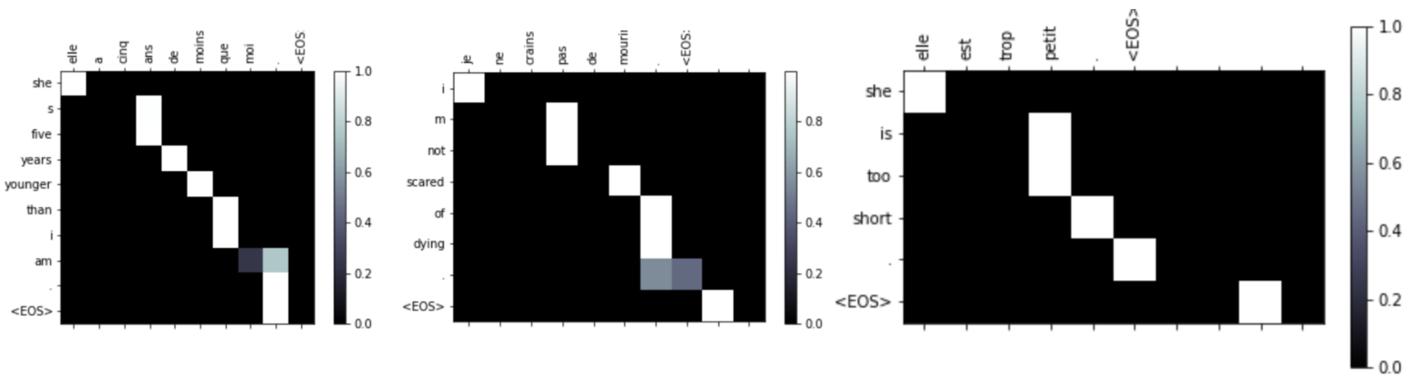


Figure 6: Attention matrix for inputs

We notice that the attention layer helps the model to understand the grammar of the sentence and not only translate word-to-word. Indeed, for instance in Figure in the first attention matrix, to translate the portion of sentence ”elle a cinq ”, the matrix attention shows that the distribution is centered around the word ”cinq”.Indeed, the model focuses on the word which gives most of the information, here ”cinq”. Then to translate ”ans” the network focuses on ”ans”, and so on. This illustrates well how attention mechanism helps the model to concentrate on the important part of the sequence.

Moreover, we notice that the weights for EOS character in the attention layer is always equal to one. If the sentence is shorter than the attention layer, all the weights after the EOS character are set to zero. This shows that EOS has the role of ”stopping” the recursive sequence-to-sequence process. It notifies to the model that the output is the final embedding.

As we have noticed from the attention layers, EOS character ables the model to learn distribution over sentences of different lenghts an to know when it has reached the end of the sequence. If we remove EOS, the model will continue the sequence-to-sequence process and won’t stop to predict new outputs.

Question 13. [5 points]

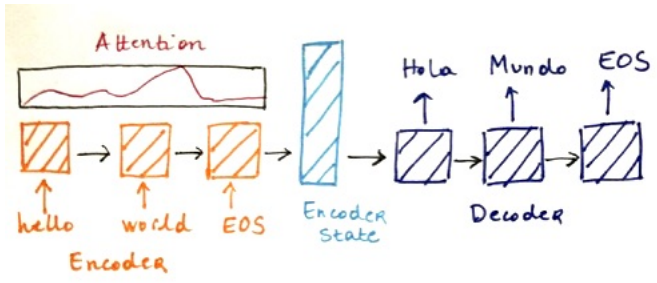


Figure 7: Transformer with attention layer

In a classic seq2seq architecture designed for translation, the encoder builds a **single context vector** from an input, an encoded sentence, and then feeds it to the decoder to predict an output, the translated sentence. The context vector sizes the same as the input vector. Somehow despite the GRU of our seq2seq model which regularizes the flow of information, for long sentences it can be **difficult for our model to memorize the entire context vector**. Imagine as a human to translate languages by memorizing entire sentences, it would be very burdensome. **Attention mechanism was inspired by human translation**. We translate a sentence word by word by focusing only on a small portion of the sentence. In the same way, **attention mechanisms create shortcuts between the input and the context vector by distributing weights along it**: high weights to the word to focus on, and smaller ones to the ones not considered to bring information to the translation context. By multiplying the context vector by the attention vector, the context vector is reduced and the network learns to memorize small proportions of sentences to translate instead of a long context vector. Thus as Figure 5 illustrates, as optimization is more efficient, convergence is faster with teacher forcing.

If the attention layer was removed in the model, the neural network would be forced to learn entire context vectors which could lead to a very long training and overfitting over the data. Not only do attention mechanism **helps managing the networks memory consumption, but also helps generalize** and learn new sentences by combining small portions of sentences it has learned to translate by the past.

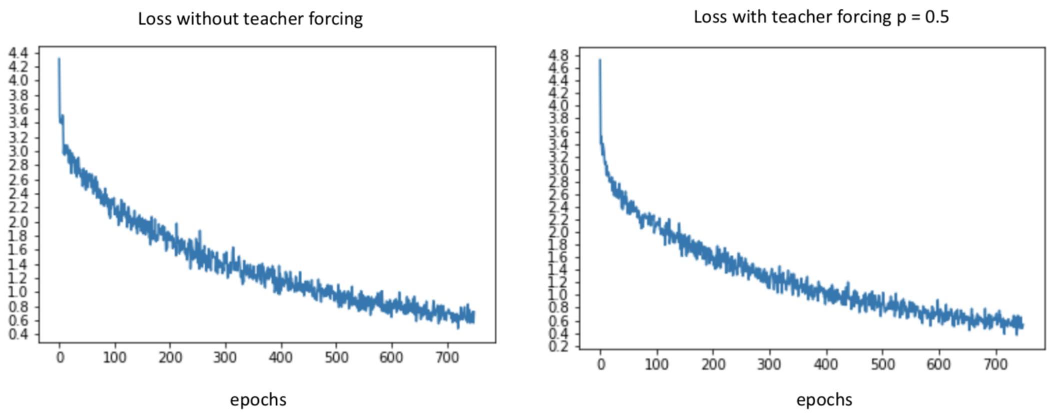


Figure 8: Loss with and without teacher forcing introduced to the network with probability 0.5

Question 14. [5 points]

As it's name suggests, **seq2seq model learns the current state sequence** from the output sequence of the previous state: it's a "sequence to sequence" architecture. If this recursive process can be very efficient in translation tasks, it can **sometimes lead to slow convergence and instability**. For instance, if the model is given the input "Cats love dogs" and that "Cats" is mistranslated by "they", as the next input will be "they", the word "love" will be impacted by the error and so on. Teacher forcing is an approach to propose a solution to such problems. Instead of feeding the network with the mistranslated term "they", you feed the network with the correct translation "Cats", and then "love", "dogs"... In short, **teacher forcing consists in giving to the decoder as input at each step the ground truth target and not the model's guess**. Therefore, the model will learn the correct sequence translation and it's statistical properties.

0.9 Transformers

Question 15. [15 points]

In earlier versions of transfomers, the normalization layer denoted LN was used to be placed after the addition of the self-attention layer and the residual connexion :

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

We can see equation (1) as post-processing step of the output i.e f(x) = LN(x).
Let's denote e the error and x_L the final layer. By applying the chain rule we can express the back propagation of the error e at any layer l.

$$\frac{\partial x_{l+1}}{\partial x_l} = \frac{\partial LayerNorm(x_l+A(x_l))}{\partial x_l} = \frac{\partial LayerNorm(x_l+A(x_l))}{\partial x_l}\frac{\partial(x_l+A(x_l))}{\partial x_l} = \frac{\partial LayerNorm(x_l+A(x_l))}{\partial x_l}(1+\frac{\partial A(x_l)}{\partial x_l})$$
$$\frac{\partial e}{\partial x_l} = \frac{\partial e}{\partial x_L}\prod_{i=l}^{L-1}\frac{\partial LayerNorm(x_i+A(x_i))}{\partial x_i}\prod_{i=l}^{L-1}(1+\frac{\partial A(x_i))}{\partial x_i})\ (2)$$

Here $\prod_{i=l}^{L-1}\frac{\partial LayerNorm(x_i+A(x_i))}{\partial x_i}$ denotes the backward pass through the normalization layer and $\prod_{i=l}^{L-1}(1+\frac{\partial A(x_i))}{\partial x_i})$ the backward pass through the sub-layer with the residual connection.
A pre-LN transformer introduce a normalization layer only on the attention layer and before it's addition to the residual connexion:

$$x_l = x_l + A(LayerNorm(x_l))(3)$$

Equation (3) regards the LN layer as a part of the sub-layer, and does intervene in the post-processing of the residual connection i.e. this step can be seen directly as a function of the output f(x) = x. As for the post-LN transformer, We can back propagate the gradient of the error e at any layer l:

$$\frac{\partial e}{\partial x_l} = \frac{\partial e}{\partial x_L}(1+\sum_{i=l}^{L-1}\frac{\partial A(LayerNorm(x))}{\partial x_i})\ (4)$$

Question 16. [15 points]

Both of these methods are efficient to train transformers. Somehow we can show that pre-Norm transformers are preferable than post-Norm transformers when training very deep networks. Let's analyze the computed gradients of e for post-Norm and pre-Norm in equations (2) and (4).

We notice in expression (4), that the error gradient $\frac{\partial e}{\partial x_l}$ is directly passed from the bottom to the top layer. In the right side of the equation we also notice that the **number of product terms is independent of the depth of the network** which is interesting in terms of computational efficiency. Moreover, **gradient vanishing will occur only if the** $\sum_{i=l}^{L-1} \frac{\partial A(LayerNorm(x_l))}{\partial x_i}$ **is equal to -1** which rarely happens.

In contrast, in equation (2) we notice that the **number of product terms is linear with the depth of network**. Thus, the deeper the network will be, the heavier it will be to backpropagate the gradient of e. Moreover, when backpropagating from the last layer L back to l, **the multiplicative term** $\prod_{i=l}^{L-1} \frac{\partial LayerNorm(x_i+A(x_i))}{\partial x_i}$ **can cause exploding or vanishing of gradients**. Finally, the **information brought by the skip connections is at each layer altered by the LN layer**. This impedes information propagation through the network and augments risks of vanishing gradients.

To put it in a nutshell, Pre-LN transformer manages to eliminate the vanishing gradient problem by keeping gradients norms constant in each layer. Indeed, normalization applies only on the attention layer. As this term will rarely be equal to -1, the the term $1 + \frac{\partial LayerNorm(A(x_l))}{\partial x_i}$ **will nearly always be non equal to zero. In contrast, Post-LN transformer suffers from unbalanced gradients i.e vanishing gradients as the index of the layer decreases which causes training instability.**