INSTITUTO SUPERIOR TÉCNICO

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

DEEP LEARNING

Homework 2

Group 61 - Lab 10

Authors:	***ID:***
Manuel Ruivo	87061
Luís Coelho	90127
Carlos Gomes	98417

2nd of February of 2022

Contents

Question 1

Exercise 1.1 - CNN Parameters

To compute the number of parameters we must first understand that the convolutional neural network (CNN) is comprised of 3 layers:

- A convolutional layer;
- A max-pooling layer;
- An output layer.

To calculate the output width and height (marked as *par*, as in *parameter*, in the equation below) in the convolutional and max-pooling layers, as our filter/kernel are symmetric we know that the height is the same as the width, we use the following expression:

$$Output_{par} = \frac{Input_{par} - Kernel_{par} + 2 \cdot Padding_{par}}{stride} + 1 \tag{1}$$

Therefore we can calculate the output of the convolutional layer being:

$$Convolutional_{width} = \frac{28 - 5 + 2 \cdot 0}{1} + 1 = 24$$

$$Convolutional_{height} = \frac{28 - 5 + 2 \cdot 0}{1} + 1 = 24 \tag{2}$$

And the output of the max-pooling layer:

$$Max - Pooling_{width} = \frac{24 - 4 + 2 \cdot 0}{2} + 1 = 11$$

$$Max - Pooling_{height} = \frac{24 - 4 + 2 \cdot 0}{2} + 1 = 11 \tag{3}$$

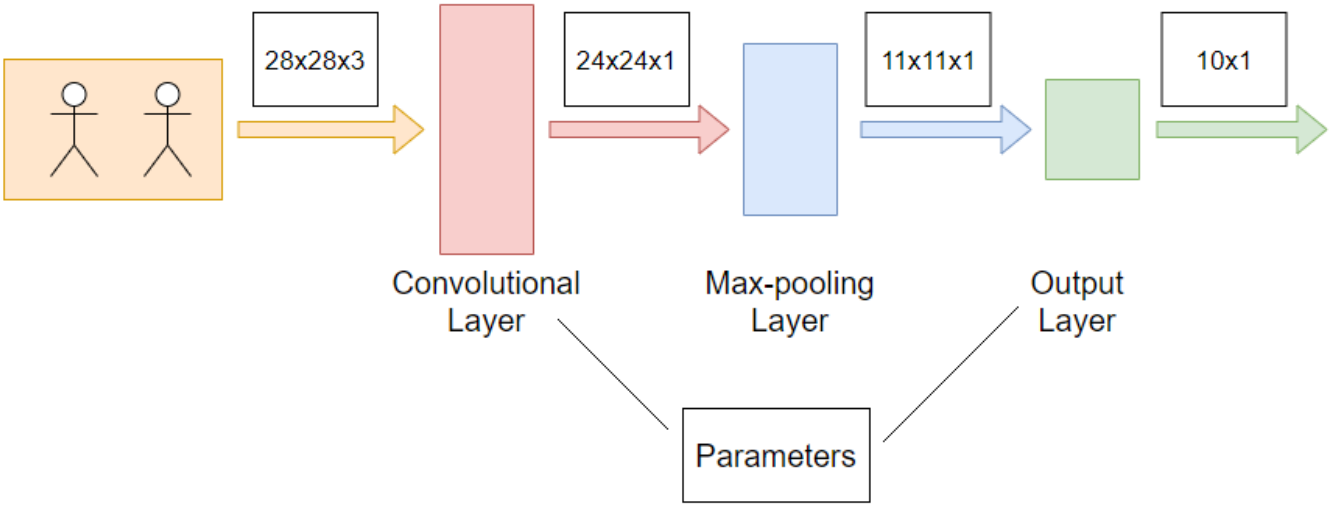

Figure 1: *Organization of the layers and respective outputs*

Now we can calculate the total number of parameters as being the sum of the parameters of each layer:

$$\begin{aligned} Total\ Number\ of\ Parameters = & Number\ of\ Convolutional\ Layer\ Parameters \\ & + Number\ of\ Max-Pooling\ Layer\ Parameters \\ & + Number\ of\ Output\ Layer\ Parameters \end{aligned} \tag{4}$$

The number of parameters in each layer can be calculated as being:

$$Number\ of\ Parameters = Number\ of\ filters \cdot ((kernel_{width} \cdot kernel_{height} \cdot Number\ of\ channels) + bias) \tag{5}$$

In the convolutional layer we have 608 trainable weights:

$$8 \cdot (5 \cdot 5 \cdot 3 + 1) = 608$$

In the Max-Pooling layer, there are no parameters (only hyperparameters).

In the Output layer we have 9 690 trainable weights:

$$10 \cdot (11 \cdot 11 \cdot 8 + 1) = 9\ 690$$

The total number of parameters is the sum of the parameters of each previous layer, thus, we have a grand total of 10 298 parameters.

$$608 + 0 + 9\ 690 = 10\ 298$$

Exercise 1.2 - Feedfoward Parameters

To calculate the total amount of parameters in this question we must first understand that the feedforward layer "replaces" the convolution and max-pooling layers seen in the previous exercise. Since we are working with a feedforward layer have to stretch/reshape the image, i.e., we do not preserve the spacial structure. Therefore, out fully connected layer presents an input of $2\ 352 \cdot 1$ stretched image (our unstretched was 28x28x3).

Since our fully connected layer has an hidden size of 100, the amount of weights is going to be 100x2352 and the amount of bias 100. Therefore we have 235 300 parameters in this layer alone:

$$100 \cdot 2\ 352 + 100 = 235\ 300$$

In the output layer the rationale is the same but with 100 inputs and a size of 10. Therefore we are going to have 100x10 weights and 10 bias, to a total of 1010 parameters.

$$100 \cdot 10 + 10 = 1\ 010$$

The grand total of parameters is the sum of the parameters of each individual layer:

$$Par = (100 \cdot 2\ 352 + 100) + (100 \cdot 10 + 10) = 236\ 310\ parameters$$

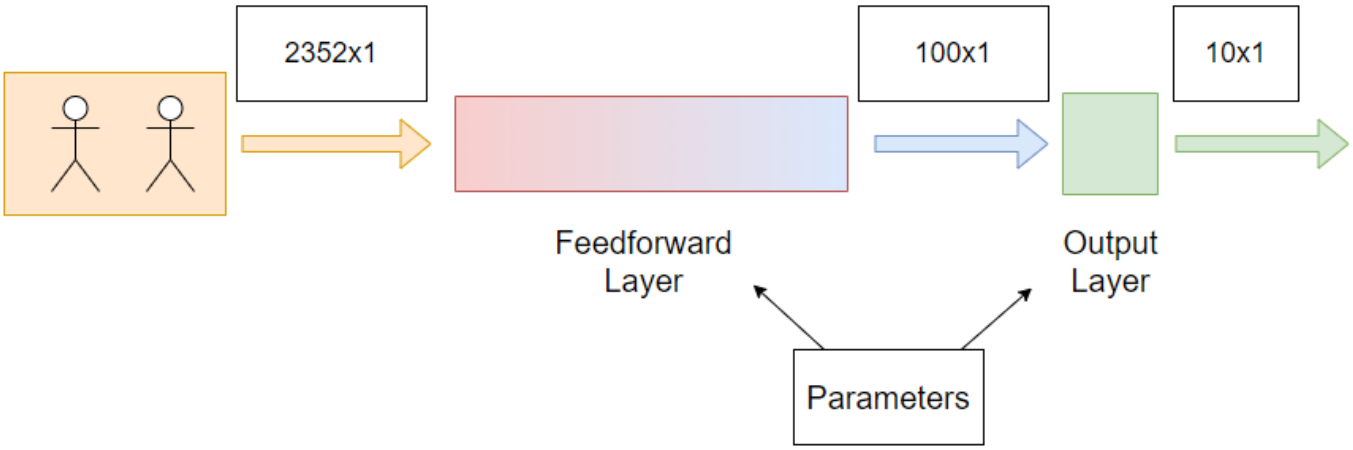

Figure 2: *Organization of the layers and respective outputs*

Exercise 1.3 - Self-Attention Heads Demonstrations

In this exercise we are going to work with self-attention. The reference used is the Lecture 11 of the class slides.

Exercise 1.3.1 - Self-Attention Probabilities for Each Attention Head

For a sequence of length L, independently of the attention head h, we know the expressions of the **Q**uery, **K**ey and **V**alue vectors:

$$\left.\begin{aligned}\mathbf{Q} &= X \cdot W_Q \\ \mathbf{K} &= X \cdot W_K \\ \mathbf{V} &= X \cdot W_V\end{aligned}\right\} \in \Re^{L\ x\ d} \tag{6}$$

We can compute the query-key affinity scores with this expression (computing the dot-product affinity between **Q**uery and **K**ey vectors, as an example):

$$S^h = \mathbf{Q}^h \cdot (\mathbf{K}^h)^T \in \Re^{L\ x\ L} \tag{7}$$

And finally, we can convert the scores to probabilities:

$$\begin{aligned} P^h = softmax(S^h) \Rightarrow P^h &= softmax(Q^h \cdot (K^h)^T) \\ &= softmax\Big((X \cdot W_Q^h) \cdot (X \cdot W_K^h)^T\Big) \\ &= softmax\big(X \cdot W_Q^h \cdot (W_K^h)^T \cdot X^T\big) \\ &= softmax\big(X \cdot A^h \cdot X^T\big)\ ,\ A^h = W_Q^h \cdot (W_K^h)^T \in \Re^{L\ x\ L} \end{aligned}$$

However, we still need to prove that the rank of matrix A is at most equal to d. To do so, we must start with the following expression:

$$A^h = W_Q^h \cdot (W_K^h)^T \in \Re^{L\ x\ L} \tag{8}$$

W_Q and W_K^T have the following dimensions:

$$W_Q \in \Re^{L\ x\ d}$$

$$W_K^T \in \Re^{d\ x\ L} \tag{9}$$

Since, for any matrix, the dimension of its row vectors is equal to the dimension of its columns vectors. Considering that L≥d, then both matrices have at most a rank of size d.

Taking into consideration that every matrix multiplication is a linear combination of their rows/columns, then the rank of a product of two matrices can not be larger than the minimum rank of either matrix.

So, if L≥d, then:

$$\begin{aligned} rank(W_Q \cdot W_K^T) &\leq min(rank(W_Q), rank(W_K^T)) \\ rank(W_Q \cdot W_K^T) &\leq d \\ rank(A^h) &\leq d \end{aligned} \tag{10}$$

Exercise 1.3.2 - Self-Attention Probabilities for Two Heads

Considering now each of the two heads (h=1,2) we have:

$$P^1 = softmax(X \cdot W_Q^1 \cdot (W_K^1)^T \cdot X^T) \tag{11}$$

$$P^2 = softmax(X \cdot W_Q^2 \cdot (W_K^2)^T \cdot X^T) = softmax(X \cdot (W_Q^1 \cdot B) \cdot (W_K^1 \cdot B^{-T})^T \cdot X^T) \tag{12}$$

$$softmax(X \cdot W_Q^1 \cdot B \cdot B^{-1} \cdot (W_K^1)^T \cdot X^T) \tag{13}$$

$$softmax(X \cdot W_Q^1 \cdot (W_K^1)^T \cdot X^T) \tag{14}$$

Therefore, we prove that for $p^1 = p^2$ when $W_Q^2 = W_Q^1 \cdot B$ and $W_K^2 = W_K^1 \cdot B^{-T}$, having B as a invertible matrix in $\Re^{d\ x\ d}$.

Question 2

This question consisted in implementing a Convolutional Neural Network (CNN), to classify the Fashion-MNIST image dataset, with 10 classes. The Fashion-MNIST dataset could be downloaded with a given python file. The CNN was implemented by completing some of the methods in the **h2-q2.py** skeleton file, using the Pytorch automatic differentiation framework.

Exercise 2.1 - Types of Invariance Captured by CNNs

First a theoretical question was given regarding the kinds of invariances that can be captured with convolutional layers. In total, there are 3 types of invariance: translational, rotational and in scale. From these types of invariances, CNNs are only truly translation invariant since the filter slides the input from left-to-right and top-to-bottom anyways, meaning that it is indifferent to where in the input a given pattern is found, mattering only that **it is** found.

In terms of rotational and scale invariance, CNNs are not by default invariant in these cases, meaning that they can not identify a same set of patterns if they are scaled or rotated differently. This is only possible when we have a rich enough data set that can display every sample at every possible rotation angle and scale.

Exercise 2.2 - Implementation of CNN for Image Classification

For this question a CNN was to be implemented. This CNN should include two convolutional blocks, with one convolutional layers, a ReLU activation function and one Max Pooling layer, followed by a series of affine transformations with ReLU activation functions. After the first affine transformation there should be a dropout, with an optimal dropout probability, and the last activation function should be a LogSoftmax, instead of a ReLU.

The CNN was to be implemented by completing the methods **__init__()** (of the **CNN** class), **foward()** (of the **CNN** class) and **train_batch()**. The code written to do so was the following:

```
class CNN(nn.Module):

    def __init__(self, dropout):
        self.convblock1 = nn.Sequential(
            # Defining a 2D convolution layer
            # Expected shape: 28x28x1
            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1), # Padding = 1 -> out.shape =
                in.shape
            # Expected shape: 28x28x16
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Expected shape: 14x14x16
        )
        self.convblock2 = nn.Sequential(
            # Defining another 2D convolution layer
            # Expected shape: 14x14x16
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=0),
            # Expected shape: 12x12x32
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Expected shape: 6x6x32
        )
        self.affineblock = nn.Sequential(
            nn.Linear(1152, 600),
            nn.ReLU(),
            nn.Dropout(p=dropout),
            nn.Linear(600, 120),
```

```
            nn.ReLU(),
            nn.Linear(120, 10),
        )
        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        out_1 = self.convblock1(x)

        out_2 = self.convblock2(out_1)

        out_3 = np.zeros((out_2.shape[0],10))

        out_3 = torch.from_numpy(out_3)

        for i in range(out_2.shape[0]):
            out_3[i] = self.affineblock(torch.reshape(out_2[i],(1,1152)))

        out_3 = self.log_softmax(out_3)

        return out_3
        raise NotImplementedError


def train_batch(X, y, model, optimizer, criterion, **kwargs):
    # clear the gradients
    optimizer.zero_grad()
    # compute the model output
    #print(X.shape)
    yhat = model(X)
    #print(yhat.shape)
    # calculate loss
    loss = criterion(yhat, y)
    # credit assignment
    loss.backward()
    # update model weights
    optimizer.step()
    # return the numeric value of loss
    return loss.item()

    raise NotImplementedError
```

In total, the training of this model was to be done for 15 epoch using SGD. Both the learning rate and the aforementioned dropout probability were to be tuned. This tuning began with the dropout parameter, where the performance of a given number of dropout probabilities (0.3, 0.4, 0.5 and 0.8) was compared in order to deduce the best possible value for the model and the data in question. The performance of each model can be found in Table 1.

Table 1: *Recorded Convolutional Neural Network Model Testing Accuracies for different Dropout Probabilities*

Learning Rate	Dropout Probability	Testing Accuracy
0.01	0.3	90.50%
0.01	0.4	90.92%
0.01	0.5	90.68%
0.01	0.8	90.28%

As seen in Table 1, the best performing dropout probability was 0.4. Dropout corresponds to the percentage of neurons that are randomly switched-off during each training iteration of a given deep learning model. This procedure allows the neurons to be more independent, and, in turn, to capture more generic features, avoiding overfitting the model. However, if the dropout probability used is too large, there is a risk that the model ends up picking up features which are far too generic, ending up underfitting the model. Because of this, a balance should be kept in mind when picking a value for the dropout probability.

In the case of Table 1, there wasn't a significant variation in the performance of the model with the variation of the dropout probability, however, it can be noted that the largest dropout probability resulted in the least accurate model. In the end, the better performing dropout probability value was 0.4.

After finding the best performing dropout probability, a study on the impact of the learning rate in the performance of a CNN was made, using the learning rate values 0.1, 0.01 and 0.001, to ultimately find the best performing model. The results of this study can be found in Table 2

Table 2: *Recorded Convolutional Neural Network Model Testing Accuracies for different Learning Rates*

Learning Rate	Dropout Probability	Testing Accuracy
0.1	0.4	89.23%
0.01	0.4	90.92%
0.001	0.4	84.97%

The results in Table 2 show that the best performing learning rate was found to be 0.01, for a dropout probability of 0.4. This can be justified by taking into consideration the effect that the learning has in the model's loss function convergence. A large learning rate, i.e. 0.1, usually results in a far faster convergence when compared to smaller learning rates, but it also results in a sub-optimal and imprecise convergence value, possible not even to minimum value of the loss function.

On the hand, a smaller learning rate, i.e. 0.001, ends up resulting in a smaller convergence step and, in turn, in a slower convergence time. The small convergence step can be beneficial in finding the global minimum of the loss function, but a smaller than necessary step can end up making the model converge to a local minimum, instead of a global, causing a non-optimal classification performance. Besides that, the slower converge speed means that a larger number of epochs is needed for the model to fully converge, which can be a downside. In example, for Question 2, where a number of solely 15 epochs was given, the model with the smallest learning rate wasn't able to fully converge during the training, causing an worse performance when compared to the large learning rate.

The best practice to have when picking a learning rate of a model is a middle-term, i.e. 0.01. A learning rate not to large to the point of losing precision, but not so small to avoid converging to a local minimum or to even not converge at all.

This being said, the best performing CNN model was found to have a dropout probability of 0.4 and a learning rate of 0.01. The evolution of the **training loss** and the **validation accuracy** with the epoch number can be seen in Figure 3 and Figure 4, respectively.

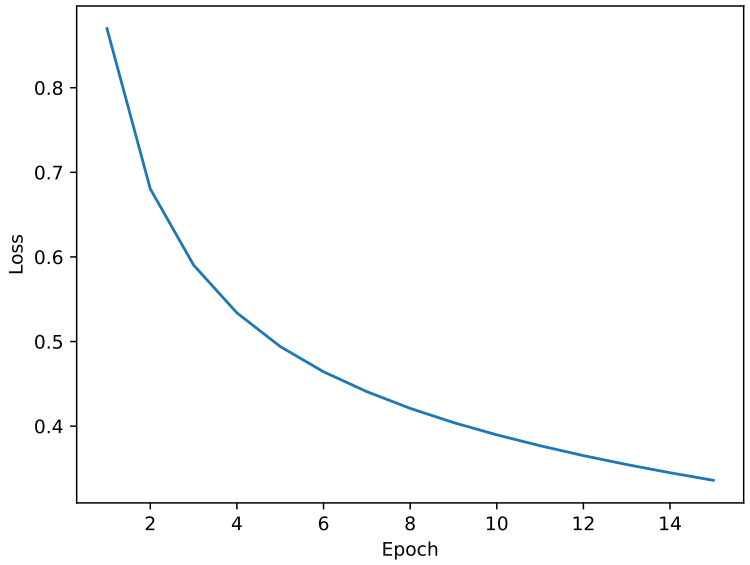

Figure 3: *Evolution of the training loss for the ideal Convolutional Neural Network during 15 epochs.*

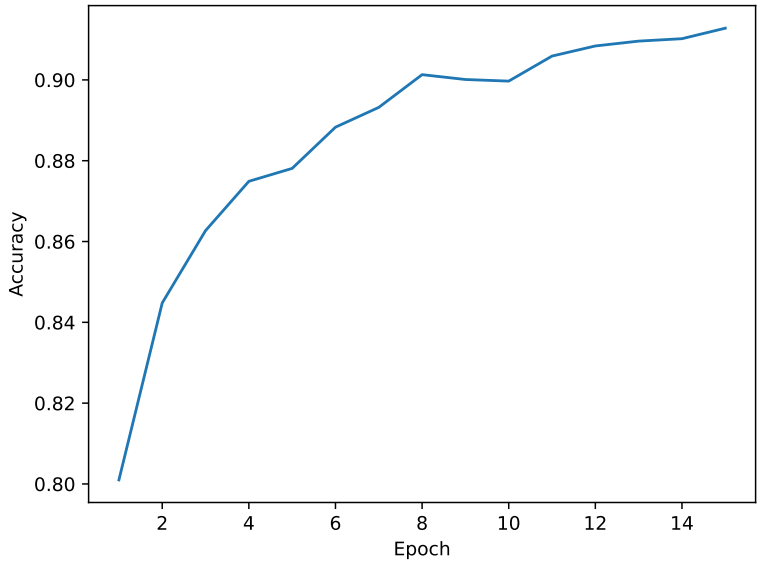

Figure 4: *Evolution of the validation accuracy for the ideal Convolutional Neural Network during 15 epochs.*

As a side note, comparing the ideal Convolutional Neural Networks' performance used in this Homework (90.92%) with the performance of ideal Feed-forward Neural Network found in the previous Homework (88.02%), it can be concluded that the CNNs perform better in image classification problems. This can be justified taking into consideration the fact that in the Feed-forward networks, the arrangement of pixels is done in an 1D vector, losing any spatial relationships within the data. In CNNs, this spatial relationship is maintained, through the use of 2D sliding windows that look into regions of the image to apply each convolutions and extract feature maps, which are used in order to store more rich and complex features to classify images, something that can't be done with Feed-forward networks. The more layers we add the more intricate this features become, starting from mere visuals patterns in the image, to some form of habitual shapes present in an image of a given class.

Exercise 2.3 - CNN Features Extraction

Convolution Neural Networks are commonly used to extract feature from image for both compression and classification purposes. When applying multiple convolutional layer it is know that the network will extract top-to-bottom features, i.e.the initial layer will look for more low-level features such as texture and simple patterns. When travelling through the network, the feature get gradually more and more meaningful. At the final stages, it is possible to observe complex patterns that represent part or complete shapes or objects. This can be easily seen with figure 5.

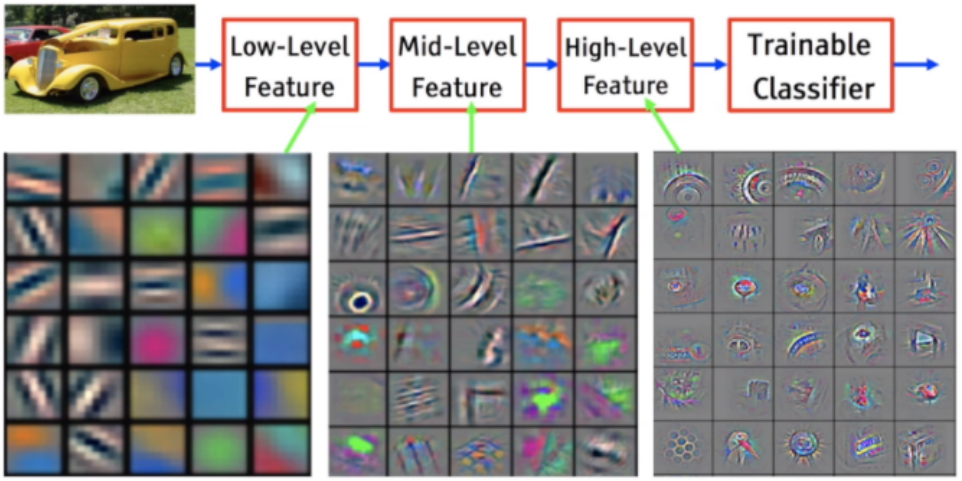

Figure 5: *Example of features extracted by a CNN*

The features are extracted by action of the kernels used within each convolutional layer. Once trained, our model learned the following kernels/filters:

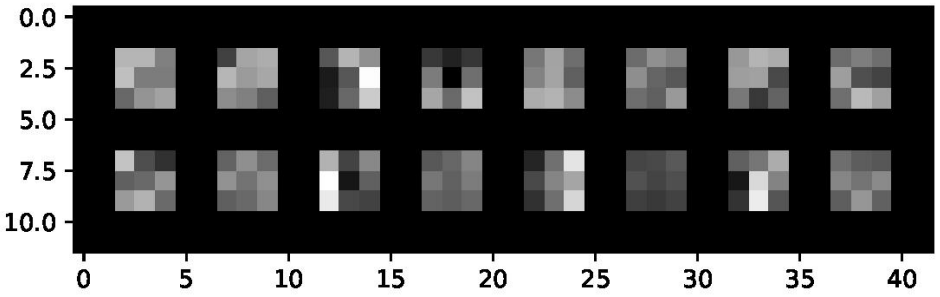

Figure 6: *Visual Representation of the 1st Convolutional block kernels.*

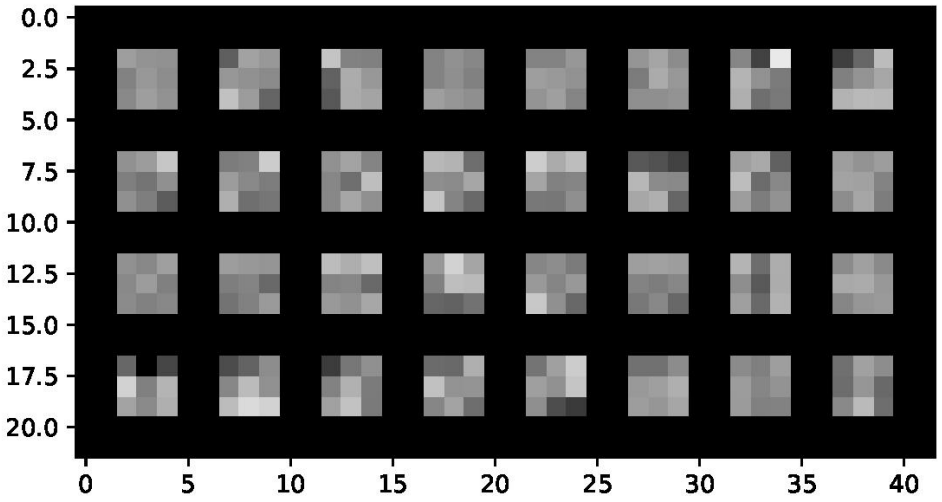

Figure 7: *Visual Representation of the 2nd Convolutional block kernels.*

As expected in Figure 6, we observe 16 kernels, one for each output channel, with dimension 3×3. Regarding Figure 7 we observe the presence of 32 kernels, again one per output channel, with the same dimensions as set before.Even tough these filters don't seem to have a major meaning when observing their visual representation as they are ultra-low resolution (3×3) they are enough to extract meaningful features that allow the CNN to classify correctly more than 90% of unseen data in the dataset.

Question 3

In this question, an image captioning LSTM model was to be implemented.

Exercise 3.1 - Image Captioning

This model should be able to automatically generate an appropriate image description. For this model, an encoder was already given, in the form of a CNN model that receives as input an image and outputs the most relevant features. However, the decoder part of this model was to be implemented, using an LTSM with and without an additional attention model. This decoder should be able to receive the features extracted by the CNN and output the desired image caption. The evaluation metric used to assess the performance of the developed model is the BLEU-4 metric and the model is to be implemented using the Python automatic differentiation toolkit Pytorch.

Exercise 3.1.a - LSTM

First, a LSTM-based decoder without an additional attention model was to be developed. This was done by implementing the method **forward()**, of class **Decoder**, located in the the **decoder.py** file. The code used to implement the LSTM decoder was the following:

```
class Decoder(nn.Module):

    def forward(self, word, decoder_hidden_state, decoder_cell_state, encoder_out=None):
        #text = [sent len, batch size] - one word at the time
        #embedded = [sent len, batch size , emb dim]
        embedded = self.dropout(self.embedding(word))
        #output = [sent len, batch size, hid dim * num directions]
        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]
        decoder_hidden_state, decoder_cell_state =
            self.decode_step(embedded,(decoder_hidden_state, decoder_cell_state))
        #hidden = [batch size, hid dim * num directions]
        hidden = self.dropout(decoder_hidden_state)
        scores = self.fc(hidden)
        return scores, decoder_hidden_state, decoder_cell_state
```

Then, the file **train.py** was to be run in order to train the developed decoder model and assess its performance. This file trained the model over 5 epochs and generated the desired training loss and validation BLEU-4 present in Figure 8 and in Figure 9, respectively.

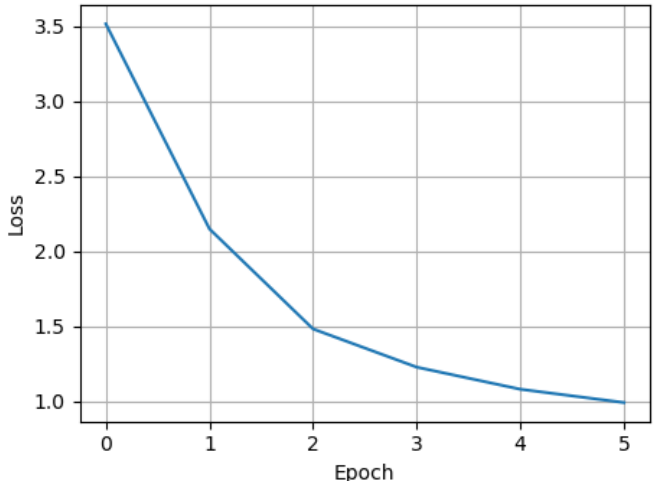

Figure 8: *Evolution of the training loss for the LSTM-based decoder without attention during 5 epochs.*

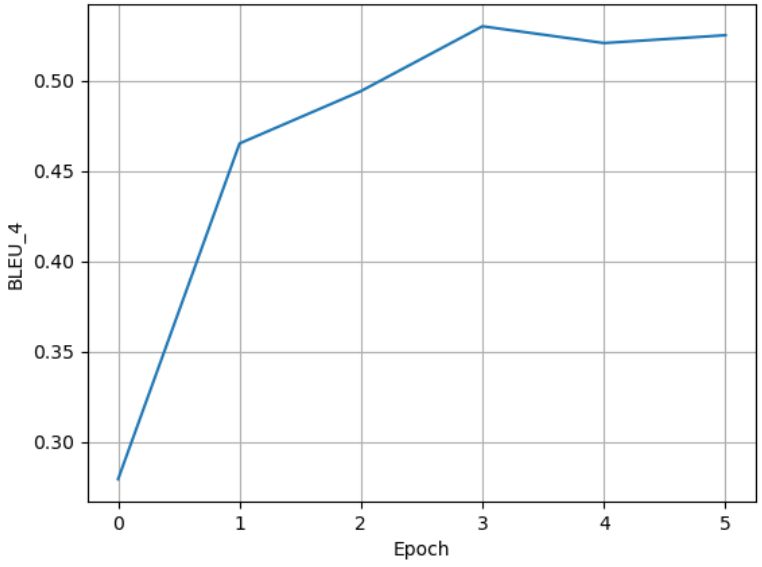

Figure 9: *Evolution of the validation BLEU-4 for the LSTM-based decoder without attention during 5 epochs.*

As expected, with each epoch, the training loss decreased as the models weights were being fine tuned in order to best represent the training set and minimize the cross-entropy loss function. This can be seen in Figure 8. On the other hand, Figure 9, displays an increase of the validation BLEU-4 performance with the training of the model, namely from around 0.47 after the first epoch to roughly 0.525 after the fifth epoch. After the training was concluded, a BLEU-4 test score of 0.5246 was registered.

Although the score was not ideal, taking into consideration that the adopted training dataset contained only 600 images with many similarities, the results were justified and deemed as acceptable. Perhaps by using a richer, more diverse and larger set of data, a better performing model could be attained.

Exercise 3.1.b - LSTM with Attention Model

Then, in the following question, another LSTM-based decoder was to be developed, this time using an additional attention model. To do so, the methods **forward()** of class **DecoderWithAttention** and **forward()** of class **Attention** were to be developed. Both of these classes were present in the **decoder_with_attention.py** file. The code used to implement the desired model with attention was the following:

```
class Attention(nn.Module):

    def forward(self, encoder_out, decoder_hidden):
        encoder = self.encoder_att(encoder_out)
        decoder = self.decoder_att(decoder_hidden).unsqueeze(1).repeat(1,256,1)
        out = self.relu(torch.cat((encoder,decoder),dim=1))
        attention_weighted_encoding = self.full_att(out).squeeze(2)
        attention_weighted_encoding = self.softmax(attention_weighted_encoding)
        # attention_weighted_encoding should be 8,512
        return attention_weighted_encoding

class DecoderWithAttention(nn.Module):

    def forward(self, word, decoder_hidden_state, decoder_cell_state, encoder_out):
        #text = [sent len, batch size] - one word at the time
        #embedded = [sent len, batch size , emb dim]
        embedded = self.dropout(self.embedding(word))
        attention = self.attention(encoder_out,decoder_hidden_state)
        attention_embedded = torch.cat((embedded,attention),dim=1)
        #output = [sent len, batch size, hid dim * num directions]
        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]
        decoder_hidden_state, decoder_cell_state =
            self.decode_step(attention_embedded,(decoder_hidden_state, decoder_cell_state))
        #hidden = [batch size, hid dim * num directions]
        hidden = self.dropout(decoder_hidden_state)
        scores = self.fc(hidden)
        return scores, decoder_hidden_state, decoder_cell_state
```

As in the previous question, the **train.py** file was used to train and assess the performance of the developed model. Like before, the model was trained over five epochs and the evolution of the training loss and of the validation BLEU-4 score can be observed in Figure 10 and in Figure 11, respectively.

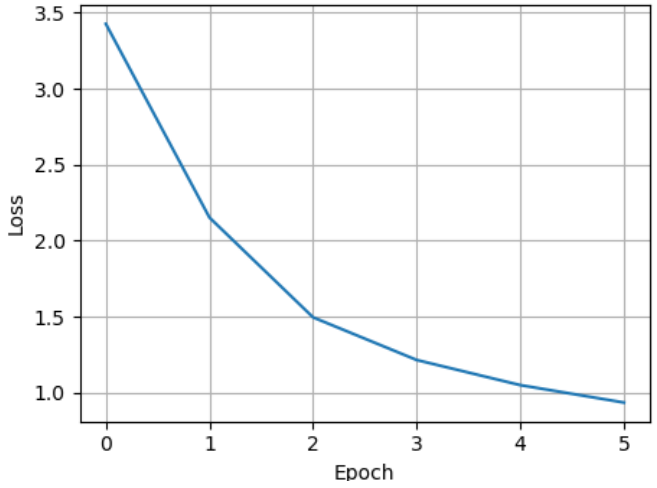

Figure 10: *Evolution of the training loss for the LSTM-based decoder with attention during 5 epochs.*

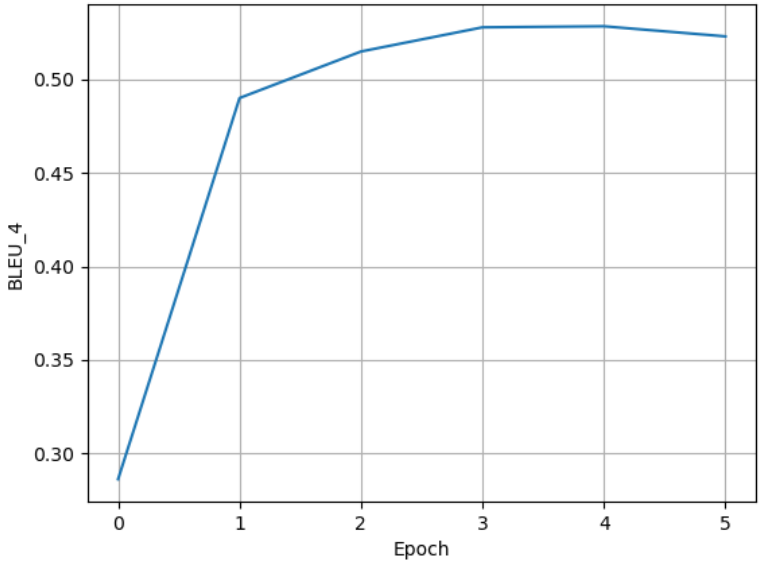

Figure 11: *Evolution of the validation BLEU-4 for the LSTM-based decoder without attention during 5 epochs.*

Although an attention mechanism was added to the decoder, as seen in Figure 10 and in Figure 11, the performance of the trained model was found similar to the performance of the decoder without attention. The same phenomenon was verified when taking into consideration that the recorded BLEU-4 test score for the decoder model with attention was 0.5189, differing only by 1% in relation to the score obtained without attention.

In theory, this should not happen since attention allows the model to identify the most important features, ignoring other features and, in turn, resulting in a better performing model. However, since the dataset only contains around 600 images, without much caption variation, and the complexity of the images, with many regions to attend, these benefits of using attention were not verified. If by chance, a larger dataset with more diverse caption and fewer elements to attend was used, then an increase in performance was to be expected.

Exercise 3.1.c - Generated Image Captions

Finally, we present the caption generated with both our models with and without attention, even though it was only required for the attention model. Three images were chosen by the instructors to be used here as an example, these being ”219.tif”, ”540.tif” and ”357.tif”, which can be found in Figure 12, Figure 14 and Figure 13, respectively. Lets now examine the captions generated.

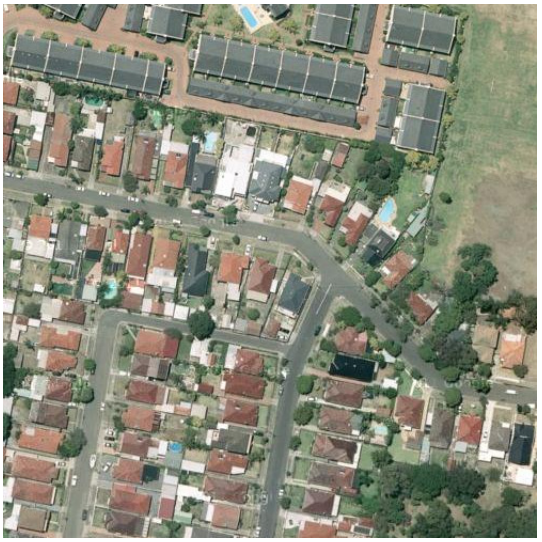

Figure 12: *Image from 219.tif*

For this image the following captions were generated: a residential area with houses arranged neatly and some roads go through this area. In this case, for both models (without attention); a residential area with houses arranged neatly and divided into rectangles by some roads (with attention). It is interesting how it is capable of recognizing a neighbourhood pattern even though it is not 100% order, as it is not a regular grid. It comes to show how powerful these models are. Rather than that the caption is pretty much spot on. Regarding the caption generated with the attention model we observe a little bit of more detail, particularly, on the pattern created by the roads.

Figure 13: *Image from 357.tif*

For this image the following captions were generated:A big river with some green bushes and white bunkers

on it while a highway passed by (model without attention); A wide river with some green bushes and a lawn beside (model with attention). The captions are actually very distinct. The model without attention seem to describe the bunkers (probably the white buildings in the image), and miss the classification of the road by saying it is a highway. However when considering the attention model, a better caption is generated mentioning the large empty planes beside the river instead of the small roads that cross them. This comes to show how attention really has an impact on the quality of the caption itself, despite the similar accuracy.

Figure 14: *Image from 540.tif*

Regarding the last picture, the caption generate was the same for both models - An industrial area with many white buildings and some roads go through this area. The caption converged in both models and it is coherent as that is truly a good description of what can be seen in the image.

Overall, this comes to prove for the conditions in which the models were trained, the attention model obtain smaller improvements over the regular model, when evaluating the quality of the captions itself. That does mean for other data sets, namely larger and more diverse, the attention model can not have a deeper effect on image captioning.