INSTITUTO SUPERIOR TÉCNICO

DEEP LEARNING

1st SEMESTER - 2021/2022

Homework assignment 2

Group 41:
Tomás ANTUNES - 102112
Raquel FÉLIX - 90497
Vasco PEARSON - 97015

Professor:
André MARTINS
Francisco MELO
Ben PETERS

2nd February, 2022

Contents

1 Question 1

1.1 Question 1.1)

In this question we want to find out how many parameters there are in a convolutional neural network (CNN) that takes as input images of size 28x28x3 and has the following layers:

- One convolutional layer with 8 kernels of shape 5x5x3 with stride of 1, and a padding of zero (i.e., no padding).
- A max-pooling layer with kernel size 4x4 and stride of 2 (both horizontally and vertically).
- A linear output layer with 10 outputs followed by a softmax transformation.

Parameters are weights or biases that are learnt during training. They contribute to the model's predictive power and are updated using backpropagation. Lets start by finding out how many parameters there are in the convolutional layer.

As we said above, the convolutional layer has 8 kernels of shape 5x5x3, stride of 1 and a padding of 0. Each kernel will slide over the image spatially, computing dot products (between the kernel and a 5x5x3 chunk of the image) and will result in one number. This dot product involves 5*5*3=75 weights and 1 bias, meaning that we have 76 parameters for each kernel. Since we have 8 kernels, the total number of parameters in the convolutional layer is the one bellow:

$$((5 \times 5 \times 3) + 1) \times 8 = 608 \text{ parameters in the convolutional layer}$$

It is important to note that we did not consider the stride and padding to calculate the number of parameters. This is because the weights and bias in the convolutional layer are not influenced by stride or padding. These are hyperparameters that only affect the shape of the output, therefore we will consider them bellow when calculating the number of parameters for the linear output layer.

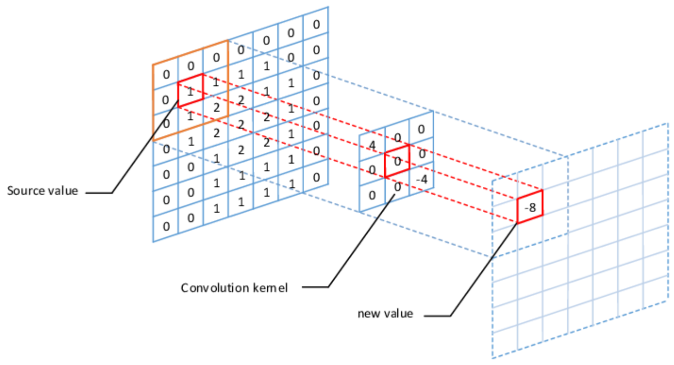

Figure 1: Convolution example, [6]

In the next layer we apply max-pooling, which is a type of pooling. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. In the case of max-pooling this is done by reporting the maximum output within a rectangular neighbourhood (in our case 4x4). Pooling helps to make the representation become approximately invariant to small translations of the input. This means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. However, pooling layers have no parameters, since all they do is calculate a specific number. Therefore the number of parameters in our max-pooling layer is zero.

0 parameters is the max-pooling layer

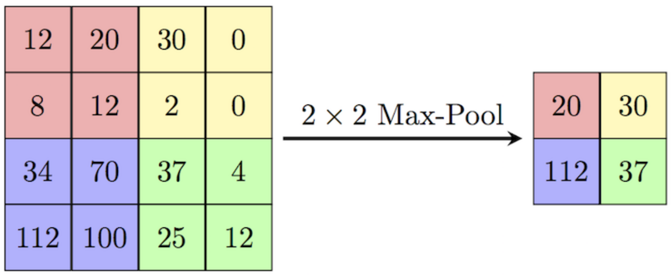

Figure 2: Max-pooling example, [5]

We now have our final linear output layer. This layer has 10 outputs and is followed by a softmax transformation. Since every neuron in the previous layer is connected to every other neuron in the output layer, this will be the layer with the highest number of parameters. The number of parameters will be the number of neurons in the output layer times the number of neurons in the previous layer (because the weight matrix will have these dimensions), plus the number of output neurons (because there will be a bias for each one). This means that we will have to find out how many neurons there are in the previous layer.

	Activation Shape	Activation Size	Parameters
Input layer	(28, 28, 3)	2352	0
Conv layer (f=5, s=1)	(24, 24, 8)	4608	608
Max-pooling layer	(11, 11, 8)	968	0
Linear output layer	(10,1)	10	9690

Table 1: Structure of the model

The table above helps us understand the overall structure of our model, while also showing the number of parameters for each layer. It is usefull to keep track of the shape of the layers, which will help us figure out the shape of the layer that precedes our linear output layer. In the input layer we feed the images to our CNN. Since the images are of size 28x28x3 we have an activation shape of (28,28,3). There is no learning done in this layer, we are only feeding the inputs into our CNN, therefore there are no parameters. Note that the activation size of each layer corresponds to multiplying the values of every dimension of the activation shape.

Then we have the convolutional layer, where the activation shape is now (24, 24, 8). The first two dimensions are 24 because we cannot fit the center of a 5x5 kernel on the two edge pixels of each row/column. The third dimension is 8 because we have 8 kernels. In the theoretical classes we saw that given an $N \times N \times D$ image, $F \times F \times D$ filters, K channels and stride S, the resulting output will be of size $M \times M \times K$ where

$$M = \frac{N - F}{S} + 1$$

In our case $N = 28$, $D = 3$, $F = 5$, $K = 8$ and $S = 1$. This leads to $M = 24$ and therefore an activation shape of $(24, 24, 8)$

The max-pooling layer activation shape can be obtained using the same formula we used above. This time we have $N = 24$, $F = 4$ and $S = 2$ since we are using a stride of 2 with 4x4 pooling. This leads to $M = 11$ and therefore we have an activation shape of (11, 11, 8). Note that pooling does not alter the last dimension, that corresponds to the number of kernels in the previous convolutional layer.

Now, since we have the shape of the linear output layer and the shape of the layer that precedes it, we can compute the number of parameters as explained above:

$$((11 \times 11 \times 8) \times 10) + 10 = 9690 \text{ parameters in the linear output layer}$$

The softmax transformation does not influence the number of parameters. It is a parameter free activation function and does not need to be trained.

Adding the parameters from the three layers, we get a total of 10298 parameters.

1.2 Question 1.2)

Lets suppose now that we replace the convolutional and max-pooling layers above by a single feedforward layer with hidden size 100, keeping the linear output layer. How many parameters would we have?

The input layer is the same as we described in the previous question and therefore it

	Activation Shape	Activation Size	Parameters
Input layer	(28, 28, 3)	2352	0
Hidden layer (100 units)	(100, 1)	100	235300
Linear output layer	(10, 1)	10	1010

Table 2: Structure of the model

also has no parameters. The difference now is that we have a hidden layer after the input layer. This layer is a fully connected layer (every neuron in the previous layer is connected to all the neurons in the hidden layer), so the number of parameters is the number of neurons in the hidden layer multiplied by the number of neurons in the previous layer (this corresponds to the number of weights), plus the number of neurons in the hidden layer (this corresponds to the biases).

$$((28 \times 28 \times 3) \times 100) + 100 = 235300 \text{ parameters in the hidden layer}$$

We calculate the number of parameters in the output layer just as we did in the previous quesion (which is the same thing we did with the hidden layer).

$$(100 \times 10) + 10 = 1010 \text{ parameters in the linear output layer}$$

Adding the parameters from the two layers, we get a total of 236310 parameters. In this model we have almost 23 times as many parameters as we did in the previous model, where we had 10298 parameters. This is because in convolutional neural networks we have parameter sharing, which is when we force sets of parameters to be equal. Natural images have many statistical properties that are invariant to translation. For example, a photo of a dog remains a photo of a dog if it is translated one pixel to the right. CNNs take this property into account by sharing parameters across multiple image locations. The same feature (a hidden unit with the same weights) is computed over different locations in the input. This means that we can find a dog with the same dog detector whether the dog appears at column i or column $i + 1$ in the image.

Parameter sharing has allowed CNNs to dramatically lower the number of unique model parameters and to significantly increase network sizes without requiring a corresponding increase in training data.

1.3 Question 1.3

1.3.1 a)

In this question we are looking at self-attention for a sequence. Let $X \in \mathbb{R}^{L\times n}$ be an input matrix for a sequence of length L, where n is the embedding size, and lets assume two self-attention heads have projection matrices $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{n\times d}$

for $h \in \{1, 2\}$, with $d < n$.
We want to show that the self-attention probabilities for each head can be written as the rows of a matrix of the form

$$P^{(h)} = \text{Softmax}(XA^{(h)}X^T)$$

where $A^{(h)} \in \mathbb{R}^{n\times n}$ has rank $\leq d$. Note that in the expression above, the Softmax transformation is applied row-wise.

In the theoretical lectures, [1], we saw that the self-attention probabilities are obtained by computing the softmax (row-wise) of the query-key affinity scores:

$$P = \text{Softmax}(QK^T) \in \mathbb{R}^{L\times L}$$

Here Q are the query vectors and K are the key vectors. We also saw in the theoretical lectures that these matrices are obtained by projecting the matrix $X \in \mathbb{R}^{L\times n}$ to a lower dimension:

$$Q = XW_Q$$
$$K = XW_K$$

Therefore we have the following expression for the self-attention probabilities:

$$P^{(h)} = \text{Softmax}(QK^T) = \text{Softmax}(XW_Q^{(h)}(XW_K^{(h)})^T) = \text{Softmax}(XW_Q^{(h)}W_K^{(h)T}X^T)$$

So $A^{(h)} = W_Q^{(h)}W_K^{(h)T} \in \mathbb{R}^{n\times n}$ (because $W_Q^{(h)}$ is an $n \times d$ matrix and $W_K^{(h)T}$ is a $d \times n$ matrix).

To show that $A^{(h)}$ has rank $\leq d$ we prove the following result:
$\text{rank}(MN) \leq \text{rank}(M)$ for any matrices $M \in \mathbb{R}^{m\times n}$, $N \in \mathbb{R}^{n\times l}$

Proof : Recall that the rank of a matrix M is defined to be the dimension of the range (or column space) of M.
In general, if a vector space V is a subset of a vector space W, then we have $dim(V) \leq dim(W)$.
Thus, we only need to show that the vector space $C(MN)$ is a subset of $C(M)$ (C is the column space).
Consider any vector $y \in C(MN)$. Then there exists a vector $x \in R^l$ such that $y = (MN)x$ by the definition of range.
Then we have $y = Mz$, where $z = Nx$.

Therefore vector y is in $C(M)$ and $C(M)$ is a subset of $C(MN)$ and we have

$$rank(MN) = \dim(C(MN)) \leq \dim(C(M)) = rank(M)$$

Using this result, we can say that $\text{rank}(A) = \text{rank}(W_Q^{(h)} W_K^{(h)T}) \leq \text{rank}(W_Q^{(h)})$. Since $\text{rank}(W_Q^{(h)}) \leq d$ we have $\text{rank}(A) \leq d$.

1.3.2 b)

Lets now assume that $W_Q^{(2)} = W_Q^{(1)} B$ and $W_K^{(2)} = W_K^{(1)} B^{-T}$. For the first attention head we have

$$P = \text{Softmax}(X W_Q^{(1)} W_K^{(1)T} X^T)$$

and for the second attention head we have

$$P = \text{Softmax}(X W_Q^{(2)} W_K^{(2)T} X^T) = \text{Softmax}(X W_Q^{(1)} B (W_K^{(1)} B^{-T})^T X^T) =$$

$$= \text{Softmax}(X W_Q^{(1)} B B^{-1} W_K^{(1)T} X^T) = \text{Softmax}(X W_Q^{(1)} W_K^{(1)T} X^T)$$

Therefore, we have the same values for the self-attention probabilities for the two attention heads.

2 Question 2

2.1 Question 2.1

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and **equivariant representations**. We have adressed the advantages of parameters sharing in the previous question. The main idea is that in convolutional neural networks parameters are shared across multiple image locations. This means that the same feature is computed over different locations in the input, meaning that we can find a specific pattern anywhere in the image.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance to translation**. A function is equivariant if a chage in the input leads to a change in the output in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, i.e., shifts it, then the convolution function is equivariant to g.

If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example if we want to detect a certain pattern in the first layer of a CNN and this pattern appears more or less everywhere in the image, it is practical to share parameters across the entire image. However in some cases, we may not wish to share parameters across the entire image. For example if we have cropped images of faces that are centered, we want the part of the network processing the top of the face to look for eyes and the part of the network processing the bottom of the face to look for a chin. As we said in the previous question, parameter sharing has allowed CNNs to dramatically lower the number of unique model parameters and to significantly increase network sizes without requiring a corresponding increase in training data.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms, such as pooling over the outputs of separately parametrized convolutions, are necessary for handling these kinds of transformations.

2.2 Question 2.2

In this question we implemented a simple convolutional network with two convolutional blocks and the structure specified in the homework. After implementing the network we trained our model for 15 epochs using SGD and three different learning rates (0.1, 0.01, 0.001). The highest validation accuracy was achieved for a learning rate of 0.01, therefore we considered this to be our best configuration. Bellow we have the plots for the training loss and the validation accuracy, both as a function of the epoch number.

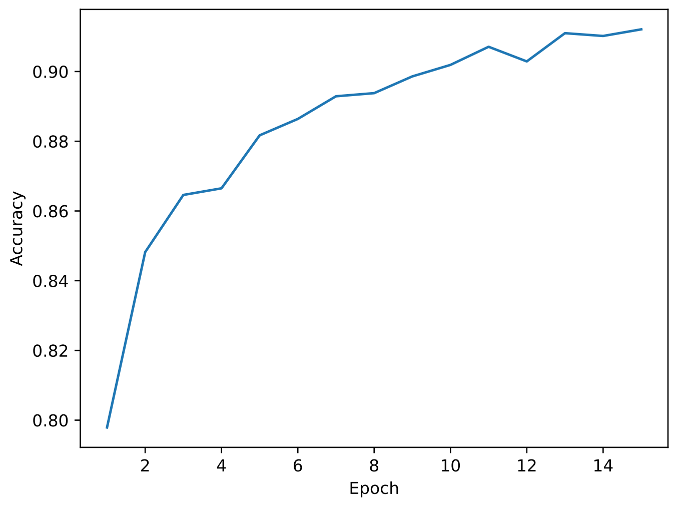

Figure 3: Accuracy plot for the validation set as function of the number of epochs for the CNN

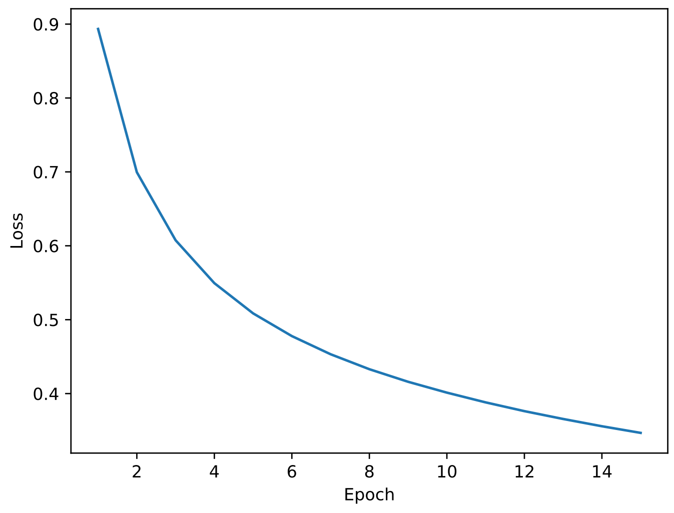

Figure 4: Loss plot for the training set as function of the number of epochs for the CNN

For this configuration, at epoch 15, we got a training loss of 0.3468, a validation accuracy of 0.9121 and a final test accuracy of 0.9072.

2.3 Question 2.3

In convolutional neural networks the bottom or lower layers take images closer to the input ones and look for edges and general features, usually small and local. As each pooling increases abstraction, the number of features per image reduces, but the remaining ones also become deeper. Top or higher layers analyse more specific and abstract features, usually specific to a certain class, which then are used to make predictions.

Below we plot the filters in the first and second convolutional layers of the model we implemented in the previous question. It is important to note that for this exercise, the image resolution is too low to notice interesting patterns in the filters.

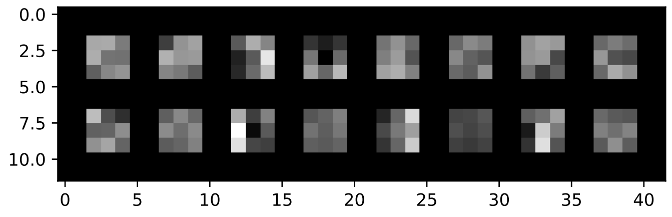

Figure 5: First layer filter

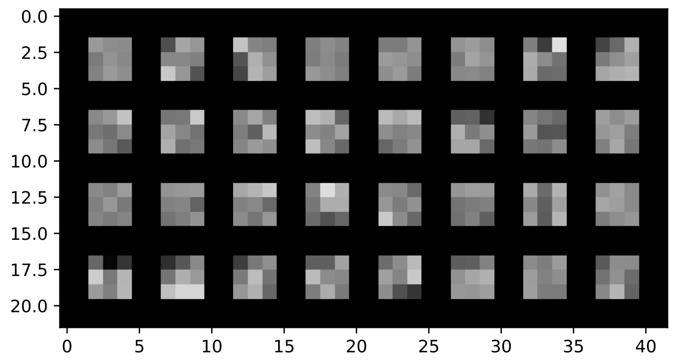

Figure 6: Second layer filter

3 Question 3

3.1 Question 3.1

In this question we were asked to perform image captioning, a process of generating textual description of an image. The task is addressed with an encoder-decoder model, where the encoder is a CNN model that processes and summarizes the input image into relevant feature representations, and the decoder is a LSTM model. An LSTM model is a type of recurrent neural network specially designed to learn long-term dependencies avoiding the long-term dependency problem. It is composed by cell units. The main purpose of the model is to filter the information that is given to the cell state by using 3 gates: input, forget and output gates.

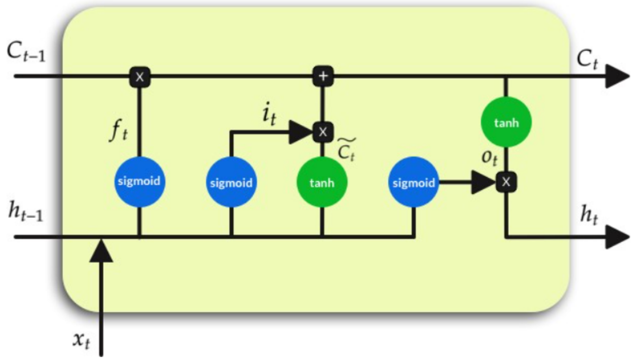

Figure 7: LSTM cell. The horizontal line in the top of the diagram, from C_{t-1} to C_t, corresponds to the cell state

The forget gate, composed by the previous hidden state h_{t-1} and the current time step x_t, determines which information is discarded. The output is filtered by a sigmoid function, values near 0 are discarded and values near 1 are considered relevant information. The expression for the forget gate is:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

To determine the information to be part of the cell state, two main steps are performed by the model. First the input gate, composed by the previous hidden state h_{t-1} and by the current time step x_t, decides which values we'll update. The second step is to create the new candidate values by a tanh layer. The two combined create the information to update the cell state. The expressions for the input gate and new candidate values are respectively:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$\tilde{C_t} = \tanh(W_C[h_{t-1}, x_t] + b_C)$$

Finally, all the gates are combined to create the new cell state. This is done by multiplying the old state by f_t, forgetting the things we decided to forget earlier, and adding $i_t * \tilde{C_t}$. This is the new candidate values, scaled by how much we decided to update each state value. These operation are shown bellow.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t}$$

The new cell state along with the output gate create the final output, as the equation bellow shows. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_0)$$

$$h_t = o_t * tanh(C_t)$$

a)

In this exercise we implemented an image captioning model using an encoder-decoder architecture with an auto-regressive LSTM as the decoder. We were asked to implement the `forward` method of the LSTM decoder. After the implementation, we trained our model and obtained the following training loss and validation BLEU-4.

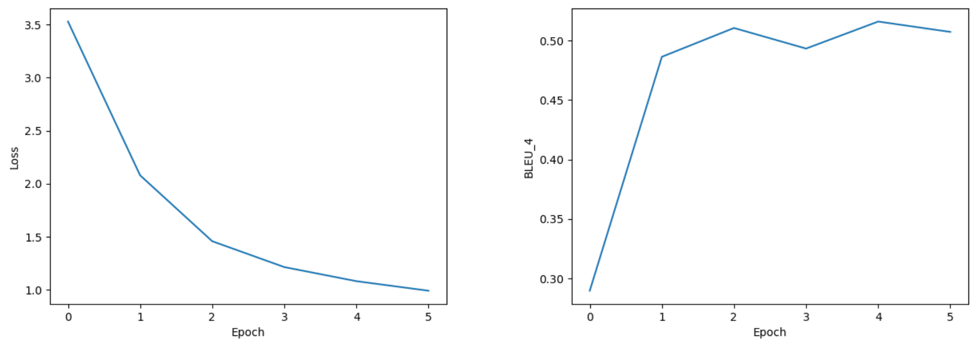

Figure 8: Training loss (left) and validation BLEU-4 (right) for the encoder-decoder

The figure for the loss is according to expected, since over epochs the loss decreases for the training set, reaching a value below 1. For the right figure, it is important to highlight that BLEU-4 considers precision of the overlap of 4-gram generated text output against the reference caption of the image, which is different from accuracy. The final BLEU-4 score in the test set was: 0.4981077727904016. The results are good taking into consideration that the dataset was small and non-diverse and also that the encoder was not fine-tuned.

b)

In this exercise we use the same strategy as before to implement an image captioning model for this task but adding an additive attention to the decoder. An attention mechanism allows a decoder to focus on certain parts of the encoder using a context vector. The context vector is a weighted average of encoder hidden states and is calculated as follows:

$$\mathbf{c}_i = \sum_j a_{ij}\mathbf{s}_j$$

Where a_{ij} are weights and s_j are encoder hidden states. The weights are calculated by combining the encoder and the decoder hidden states as follows:

$$\mathbf{a}_i = \text{softmax}(f_{att}(\mathbf{h}_i, \mathbf{s}_j))$$

Where s_j are encoder hidden states, h_i decoder hidden states and f_{att} the weighting function.
The weighting function in the case of the project is the additive attention that is calculated as follows:

$$f_{att}(\mathbf{h}_i, \mathbf{s}_j) = \mathbf{v}_a^T \tanh(\mathbf{W}_1\mathbf{h}_i + \mathbf{W}_2\mathbf{s}_j)$$

Where $\mathbf{W}_1$ and $\mathbf{W}_2$ are matrices corresponding to the linear layer and $\mathbf{v}_a$ is a scaling factor. In the project we used RELU instead of tanh.

The results obtained by implementing both forward methods in the decoder with attention script are the following:

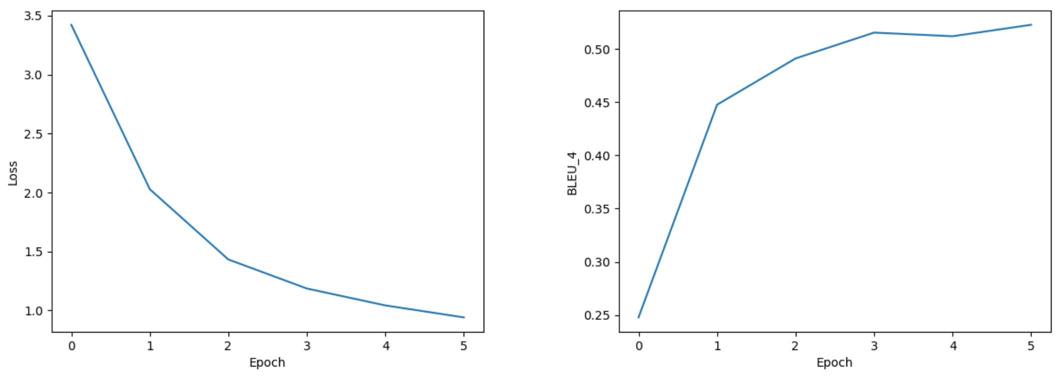

Figure 9: Training loss (left) and validation BLEU-4 (right) for the encoder-decoder with attention

Regarding the training loss, the results still follow the expected, i.e, the loss decreases over epochs reaching a value below 1. For the validation BLEU-4 it is possible to see a slightly better performance for the decoder with attention when compared to the decoder without attention in the previous exercise. The same can be observed in the final BLEU-4 score in the test set of 0.5264225434556401. This was expected given the purpose of the attention mechanism of selecting information from the encoder, the improvement in BLEU-4 was not higher since the dataset is very small and that there is a high complexity and number of regions in these images to attend.

c)

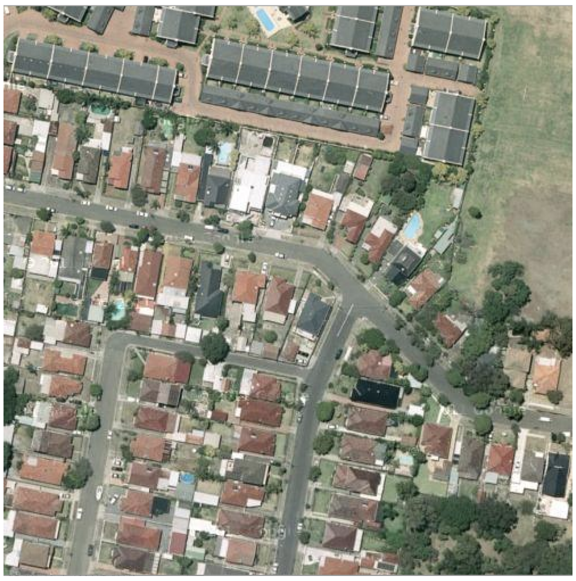

Figure 10: Image: 219.tiff

Caption generated for the image above: “a residential area with houses arranged neatly and some roads go through this area”.

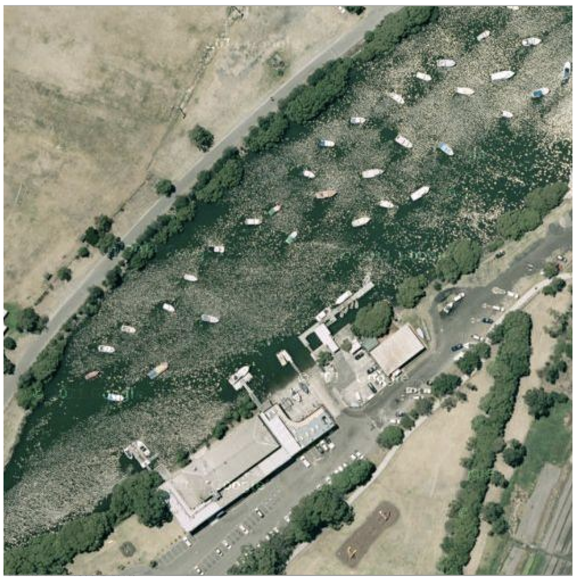

Figure 11: Image: 357.tif

Caption generated for the image above: “a curved river with some green bushes and a highway passed by”.

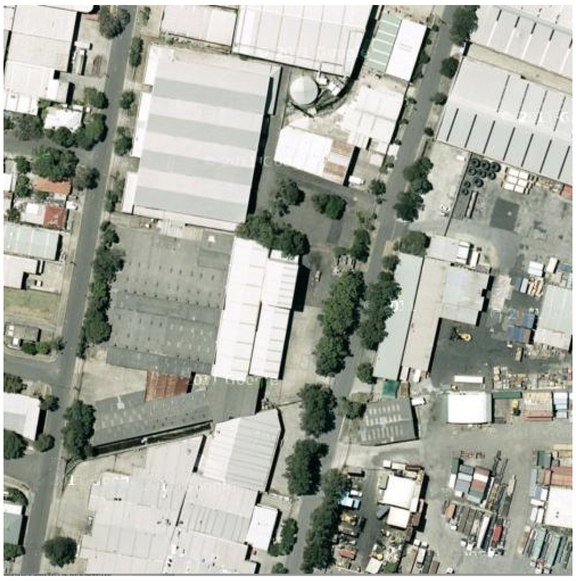

Figure 12: Image: 540.tif

Caption generated for the image above: "an industrial area with many white buildings and some roads go through this area".

4 Bibliography

- MARTINS, ANDRÉ; MELO, FRANCISCO; FIGUEIREDO, MÁRIO. (2021). Deep Learning Course: All lectures [PowerPoint presentation], [1]
- Artificial Intelligence Engines: A Tutorial Introduction to the Mathematics of Deep Learning: James Stone 2019 Sebtel Press, [2]
- Deep Learning: Ian Goodfellow, Yoshua Bengio, Aaron Courville, [3]
- Deep Learning with Python: François Chollet 2017 Manning Publications, [4]
- https://paperswithcode.com/method/max-pooling, accessed on 16th January, [5]
- https://www.researchgate.net/figure/Convolution-process_fig2_335878018, accessed on 16th January, [6]
- Osinga, E 2018, Deep Learning Cookbook, O'Reilly Media, Inc. [7]
- https://towardsdatascience.com/translational-invariance-vs-translational-equivariance-f9fbc8fca63a, Mishra, D 2020, Translational Invariance Vs Translational Equivariance, Towards Data Science, viewed 31 January 2022, [8]
- From a LSTM cell to a Multilayer LSTM Network with PyTorch: https://towardsdatascience a-lstm-cell-to-a-multilayer-lstm-network-with-pytorch-2899eb5696f3 (accessed in 01/02/2022), [9]
- Understanding LSTM Networks: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (accessed in 01/02/2022), [10]

- Implementing additive and multiplicative attention in PyTorch: https://tomekkorbak.com/2
attention-in-pytorch/ (accessed in 01/02/2022), [11]