Instituto Superior Técnico

Master Degree in Electrical and Computer Engineering

Deep Learning

2021/2022 - 2nd Quarter

Homework 2

Group 30

Tiago Franco Pinto nº93187 tiagofrancopinto@tecnico.ulisboa.pt
Tomás Pimenta nº93195 tomas.pimenta@tecnico.ulisboa.pt

Faculty: Mário Figueiredo

February 2nd 2021

Question 1

1.1

A convolutional neutral network (CNN) with an input image size 28x28x3, a convolutional layer with 8 kernel shape 5x5x3 with stride 1 and a padding of zero, a max-pooling layer with a kernel size 4x4 with stride 2 (horizontal and vertical) and a linear output with 10 units followed by a softmax transformation was considered.

The output width of the convolutional layer can be calculated by 1

$$\text{Output width} = \frac{\text{input width} - \text{kernel width} + 2 \times \text{padding width}}{\text{stride}} + 1 = \frac{28 - 5 + 2 \times 0}{1} + 1 = 24 \tag{1}$$

Since the input and kernel are symmetrical the output height = output width = 24

The output width of the max-pooling layer can be calculated by 2

$$\text{Output width/height} = \frac{\text{input width} - \text{kernel width}}{\text{stride}} + 1 = \frac{24 - 4}{2} + 1 = 11 \tag{2}$$

Once again, since the kernel used is symmetrical the output height = output width = 11

The input layer does not have any parameters because it is the input of the CNN. The max-pooling layer only reduces the convolution layer's output thus it does not have any parameters as well. For the convolutional layer the number of parameters can be calculated by 3

$$\begin{aligned}\text{Number of parameters} &= \text{number of filters} \times ((\text{kernel width} \times \text{kernel height} \times \text{kernel depth}) + \text{bias}) \\ &= 8 \times ((5 \times 5 \times 3) + 1) = 608.\end{aligned} \tag{3}$$

For the linear output layer the number of parameters can be calculated by 4

$$\text{Number of parameters} = \text{number of units}\ ((\text{input width} \times \text{input height} \times \text{input depth}) + \text{bias}) \tag{4}$$

The input width and height is the max-pooling layer output shape previously calculated consequently the number of parameters $= 10(11 \times 11 \times 8) + 1 = 9690$

In total this network has 608 + 9690 = 10298 parameters.

1.2

The convolutional and max-pooling layers were replaced by a feedfoward, fully connected layer with hidden size 100 and output size 10.

Just like in the previous question, the input layer does not have any parameters because it is the input of the CNN. For the hidden layer the number of parameters can be calculated by equation 4.

Number of parameters $= 100(28 \times 28 \times 3) + 1) = 235300$ For the output layer the layer the number of parameters can be calculated by 5

$$\begin{aligned}\text{Number of parameters } &= \text{ number of units } \times (\text{ number of hidden layer units } + \text{ bias }) \\ &= 10 \times (100 + 1) = 1010.\end{aligned} \tag{5}$$

In total this network has 235300 + 1010 = 236310 parameters.

As it can be seen, the feedfoward neural network has a lot more parameters compared to the CNN. This shows that the CNN is able to reduce significantly the total number of parameters. This is achieved while the performance remains the same. Less parameters means the computational costs are lower.

1.3

$\boldsymbol{X} \in \mathbb{R}^{L \times n}$ is and input matrix for a sequence of lenght L, where n is the emvedding size. The self-attention heads have matrices $\boldsymbol{W}_Q^{(h)}, \boldsymbol{W}_K^{(h)}, \boldsymbol{W}_V^{(h)} \in \mathbb{R}^{n \times d}$ for $h \in \{1, 2\}$, with $d \leq n$.

1.3.1

The self-attention probabilities can by calculated by scaled dot product attention and applying soft max row-wise .

$$\boldsymbol{Q}^{(h)} = \boldsymbol{X}\boldsymbol{W}_Q^{(h)} \text{ and } \boldsymbol{K}^{(h)} = \boldsymbol{X}\boldsymbol{W}_K^{(h)}$$

$$\begin{aligned}\boldsymbol{P} = \text{Softmax}\left(\frac{\boldsymbol{Q}^{(h)}\boldsymbol{K}^{(h)\top}}{\sqrt{d}}\right) &= \text{Softmax}\left(\frac{\boldsymbol{X}\boldsymbol{W}_Q^{(h)}\left(\boldsymbol{X}\boldsymbol{W}_K^{(h)}\right)^\top}{\sqrt{d}}\right) \\ &= \text{Softmax}\left(\frac{\boldsymbol{X}\boldsymbol{W}_Q^{(h)}\boldsymbol{W}_K^{(h)\top}\boldsymbol{X}^\top}{\sqrt{d}}\right) \\ &= \text{Softmax}\left(\boldsymbol{X}\boldsymbol{A}^{(h)}\boldsymbol{X}^\top\right),\end{aligned} \tag{6}$$

Thus $\boldsymbol{A}^{(h)}$ is defined as $\boldsymbol{A}^{(h)} = \frac{\boldsymbol{W}_Q^{(h)}\boldsymbol{W}_K^{(h)\top}}{\sqrt{d}}$.

1.3.2

By substituting $\boldsymbol{W}_Q^{(2)} = \boldsymbol{W}_Q^{(1)}\boldsymbol{B}$ and $W_K^{(2)} = W_K^{(1)}B^{-\top}$, where $B \in \mathbb{R}^{d\times d}$ in $\boldsymbol{A}^{(2)}$ we get:

$$\begin{aligned}\boldsymbol{A}^{(2)} &= \frac{\boldsymbol{W}_Q^{(1)}\boldsymbol{B}\left(\boldsymbol{W}_K^{(1)}\boldsymbol{B}^{-\top}\right)^\top}{\sqrt{d}}\\ &= \frac{\boldsymbol{W}_Q^{(1)}\boldsymbol{B}\boldsymbol{B}^{-1}\boldsymbol{W}_K^{(1)\top}}{\sqrt{d}}\\ &= \frac{\boldsymbol{W}_Q^{(1)}\boldsymbol{I}\boldsymbol{W}_K^{(1)\top}}{\sqrt{d}}\\ &= \frac{\boldsymbol{W}_Q^{(1)}\boldsymbol{W}_K^{(1)\top}}{\sqrt{d}}\end{aligned} \tag{7}$$

These means $\boldsymbol{A}^{(2)} = \boldsymbol{A}^{(1)}$ thus $\boldsymbol{P}^{(2)} = \boldsymbol{P}^{(1)}$ meaning the self-attention probabilities are the exactly the same for the two attention heads.

Question 2

2.1 Convolutional layers' equivariances

Convolutional layers exhibit translational/shift equivariances. If these layers didn't have this property, it would be necessary to retrain the model for the same picture as there was an element translation, making different pixels within the image now contain a certain element (i.e, an animal), resulting in wastes of computation for models with no translational equivariance properties. This property makes that translations on an input result on the same translation on the feature mapping space. This way, the convolutional layers with translational equivariance allow the models to be better at maintaining the same output when an image is translated, making it a good solution for image classification.

2.2 Implementing a simple convolutional network

When implementing the required convolutional network, we divided the network in three different blocks. Two "convolutional blocks" and one output "fully connected block". Each convolutional block is composed of a convolutional layer, an activation function (ReLU) and a pooling layer. The convolutional layer is characterised by the following attributes:

- input channels: the number of channels that the layer receives;
- output channels: the number of filters that will perform the convolution in this layer;
- kernel size: the size of the filters;

- stride: the shift in pixels between two consecutive windows;
- padding: layers of zeros added to the input.

The ReLU adds non-linearity to the network. The pooling layer makes the representations smaller and more manageable. It operates over each activation map independently. This layer is also characterised by a kernel size and striding. The fully connected layer is made of an affine transformation, a ReLU, a dropout layer, another affine transformation, another ReLU and finally another affine transformation followed by an output LogSoftmax layer (in this order).

The values we used to characterise the two first blocks are be needed to perform input size calculations.The first block convolutional layer has 16 output channels, a kernel size of 3x3, stride of 1 and padding chosen to preserve the original image size. The max pooling layer has a kernel size of 2x2 and stride of 2. The second block differs on the number of output channels (32) and has a padding of zero.

Knowing the the number of input features is given by

$$input\,features = nr\,output\,channels\ \times\ output\,width\ \times\ output\,height, \tag{8}$$

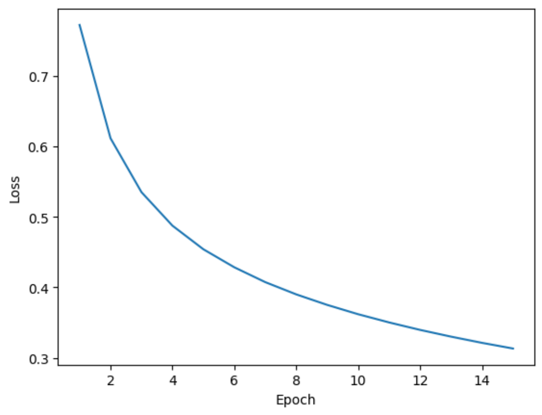
and knowing that depth corresponds to the number of channels and both the output height and width are the same (since kernel and padding are symmetric) and given by

$$output\,width\ =\ output\,height = \frac{input\,width\ -\ kernel\,width\ +\ 2\ \times\ padding\,width}{stride} + 1 \tag{9}$$

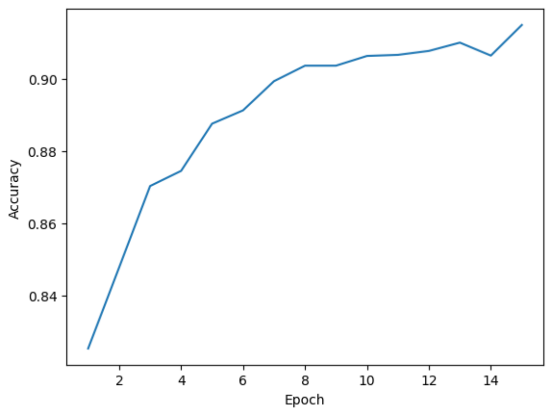
we can compute the number of input features of both the second block and the third block.

Using (9) and the values of the first block (a padding size of (kernel size - 1)/2 = 1), we can conclude that the first block's convolution layer produces an 28x28x16 output and the max pooling layer produces a 14x14x16 output. Using this last output dimensions as input dimensions for the second block and using (9) again, we get an 12x12x32 output from that block's convolutional layer and a 6x6x32 output from the max pooling layer. After using this, we can use (8) to compute the size of the last blocks input: 1152.

After implementing the *forward* and *trainbatch* methods, we training the model for 15 epochs using SGD for three different learning rates: 0.001, 0.01 and 0.1. By comparing the graphs of the training loss and validation accuracy (as we're tuning the learning rate hyperparameter), we noticed that the model that behaved the best was the learning rate of 0.01. As we can see in Figure 2, the loss gets down to near 0.3 and the validation accuracy starts stabilising at around 0.9.



(a) Loss



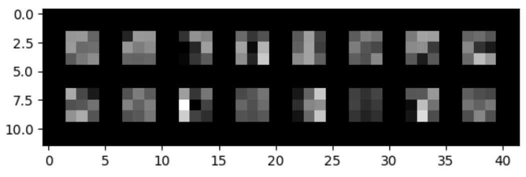
(b) Validation Accuracy

Figure 2: Metrics evolutions of the CNN with learning rate 0.01

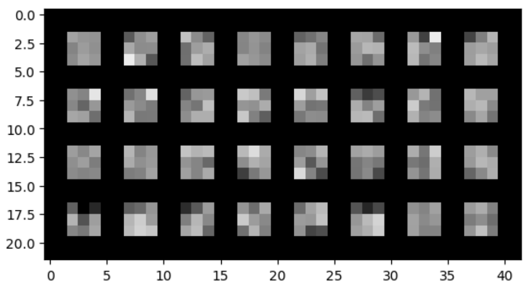
The accuracy on the final test (on the model with a learning rate of 0.01) was 0.9076.

2.3 Filters

After implementing the CNN, we plotted the filters after the first and second convolutional layers (with the same 0.01 learning rate). As expected, the number of filters matched the number of output channels of the layer. We can see the plots on Figure 3.



(a) First Convolutional Layer



(b) Second Convolutional Layer

Figure 3: Filters

Knowing that the images used as input to our model have a really low resolution, we can't clearly notice the patterns on the filters. However, usually the top layers of CNNs capture low level representations, such as edges and corners, and the deeper ones identify more "semantic" representations, such as objects and scenes.

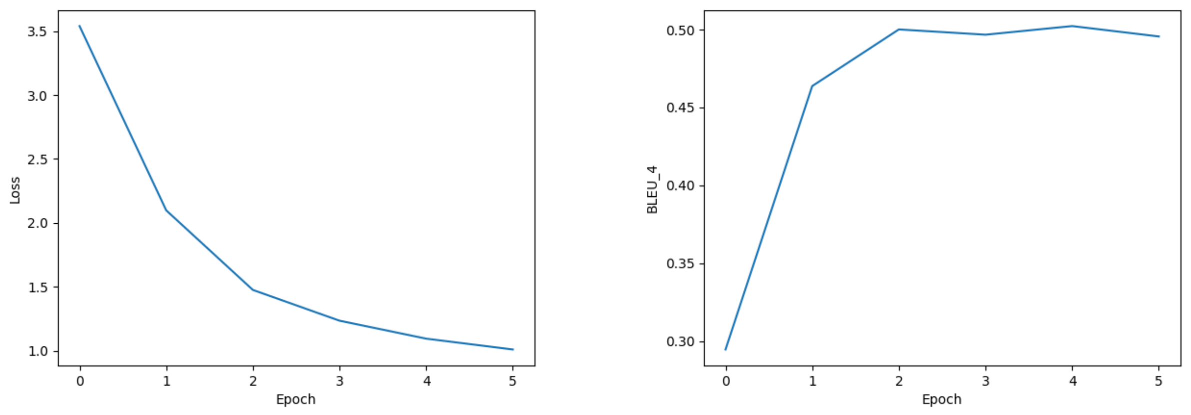
Question 3

3.1 a) RNN without attention

On this first problem, we implemented a vanilla image captioning model using an encoder-decoder architecture with an auto-regressive LSTM as the decoder. This network was used to solve a sequence generating problem, since our goal is to generate symbols sequentially with an auto-regressive model. We are then working with a sequence to sequence model with an encoder-decoder architecture, where the encoder encodes the source sentence into a vector state and the decoder (in this case, a LSTM) generates the target sentence conditioned on the vector state.

To implement this network, we needed to implement the *forward* method of the *Decoder* class. To do so, we passed the source word through an embedding layer before feeding it to the decoder along with the cell and hidden layer states (decoding). Finally, we added a dropout layer to improve generalization and passed its output through a fully connected layer that generates the output.

When running the model, we evaluated its performance based on the training loss and the validation's BLEU-4. As we can see in Figure 4, the loss gets down to around 1.0 and the BLEU-4 quickly rises to around 0.5 and then stabilizes.



(a) Loss (b) BLEU-4

Figure 4: RNN without attention Metrics

The BLEU-4 on the final test was 0.538943.

3.1 b) RNN with attention

Aiming to improve performance, we implemented an attention mechanism on the model. Attention is the idea of freeing the encoder-decoder from the fixed-lenght internal representation. To do so, the decoder searches for a set of positions in the source where the most relevant information is located. The model then uses this to predict a target word based on the context vectors associated with these source positions and all the target words that were previously generated.

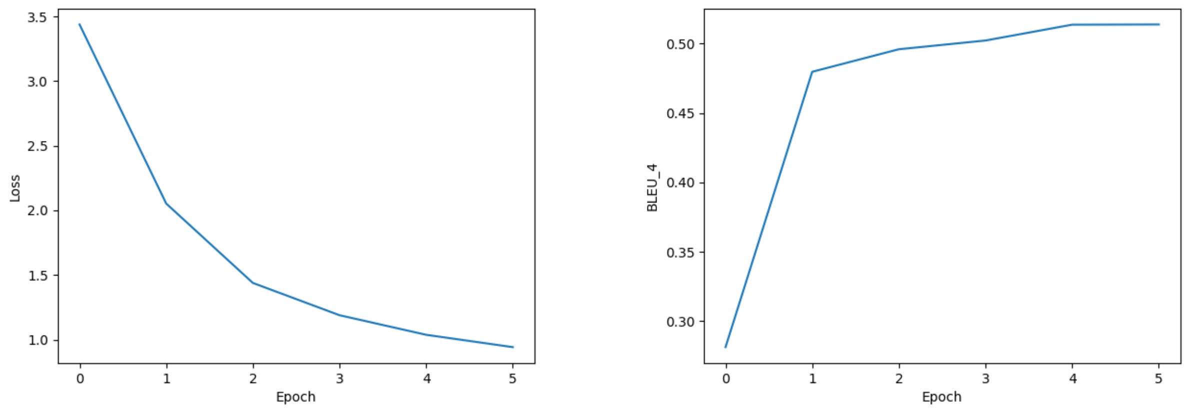
To compute the context vector, we start by using a vector $\boldsymbol{q}$ (*query*, the decoder state) and a vector of inputs $\boldsymbol{H} = [\boldsymbol{h_1}, ..., \boldsymbol{h_l}\ ]^T$. After that, we need to compute the affinity scores, and since we are applying additive attention, we use

$$s_i = \boldsymbol{u^T} ReLU(\boldsymbol{Ah_i} + \boldsymbol{Bq}) \tag{10}$$

and to compute the probabilities calculate the softmax of each score. Finally, to compute the weighted average (context vector) we use

$$\boldsymbol{c} = \boldsymbol{H^T p} = \sum_{i=1}^{L} p_i \boldsymbol{h_i} \tag{11}$$

Having this weighted average, we just need to use the same *forward* method as in the previous question, however, we need to concatenate this context vector to the output of the embedded layer before feeding it to the decoder. As we can see in Figure 5, the graphs behaved quite similarly to the graphs on Figure 4, only that the BLEU-4 score now stabilized slightly above the 0.5 mark.



(a) Loss (b) BLEU-4

Figure 5: RNN with attention metrics

The BLEU-4 score on the final test was 0.553886, proving to be slightly better that the model without attention. However, knowing that our dataset is small, the difference between this BLUE-4 scores is also small.

3.1 c) Image Captioning

After running the model, we plotted 3 images ("219.tif", "357.tif" and "540.tif") and the corresponding generated captions.

The caption generated for "219.tif" (Figure 6) was "*A residential area with houses arranged neatly and some roads go through this area.*".

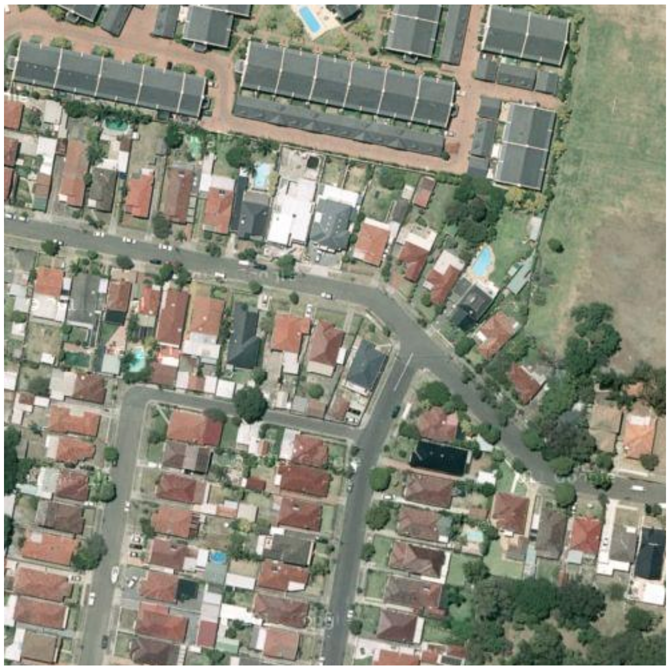


Figure 6: Image 219

The caption generated for ”357.tif” (Figure 7) was ”*A big river with some green bushes and white bunkers on the roadside.*”.

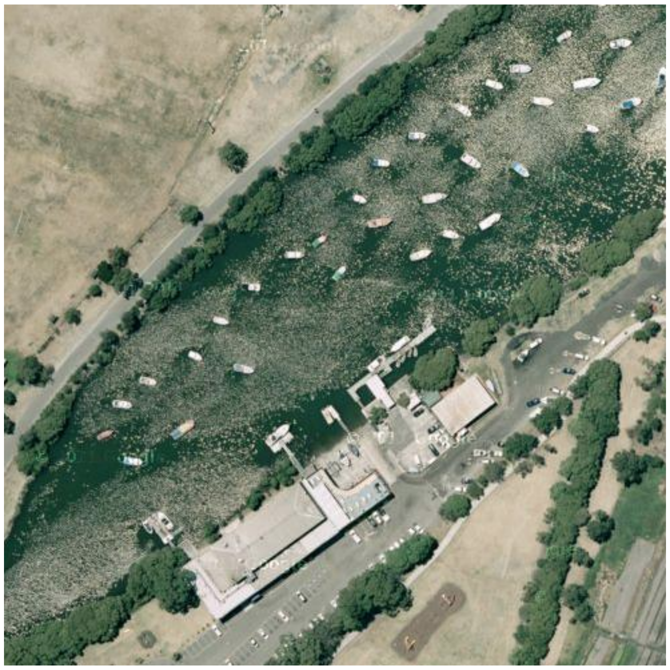


Figure 7: Image 357

The caption generated for ”540.tif” (Figure 8) was ”*An industrial area with many white buildings and some roads go through this area.*”.

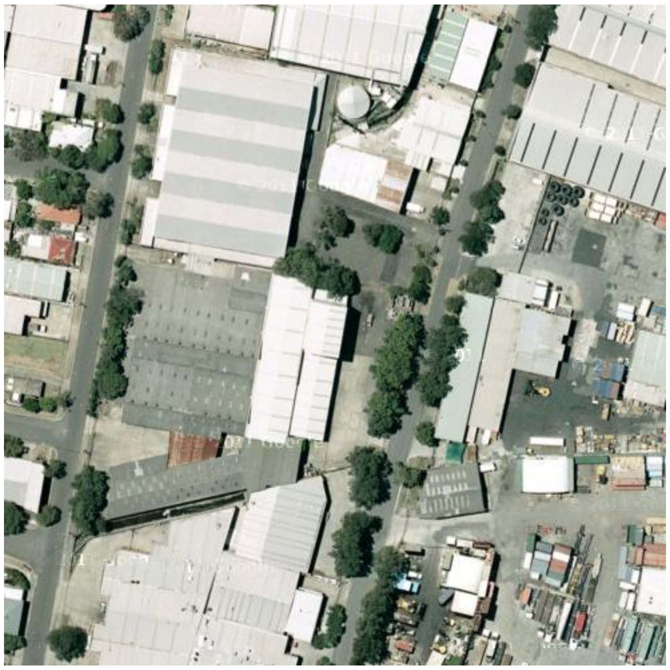


Figure 8: Image 540

As we can see by observing the images, the captions, even though simplistic, are actually pretty accurate, proving that the model behaved properly.