Deep Learing

Homework 2

Stefanos Zafiris ist1109041
Črtomir Perharič ist1108973

December-January 2023/2024

1 Question 1

1.1 Self-attention computational complexity

In this task we have three matrices with the same size:

$$\mathbf{Q} \in \mathbb{R}^{L \times D}, \mathbf{K} \in \mathbb{R}^{L \times D}, \mathbf{V} \in \mathbb{R}^{L \times D}$$

The task is to determine the computational complexity of:

$$\mathbf{Z} = \texttt{Softmax}(\mathbf{Q}\mathbf{K}^T)\mathbf{V}$$

in terms of L. This is done by determining the computational complexity of each of the three steps. The first one is the matrix multiplication of $\mathbf{Q}\mathbf{K}^T$. The computational complexity of such an action is $O(L^2 \cdot D)$ as there is the element-wise multiplication and summation of the products. The resulting matrix will have the shape $L \times L$. The next step is to determine the computational complexity of the Softmax applied to the matrix achieved from the multiplication. Here you have to first calculate the exponential and after that a sum, which achieves a computational complexity of $O(L^2)$. The output matrix still has the shape $L \times L$. Last operation is again a matrix multiplication, but this time with a matrix of size of $L \times L$ `and` $L \times D$. The computational complexity of this operation is $O(L^2 \cdot D)$. So the overall computational complexity is:

$$O(L^2 \cdot D + L^2 + L^2 \cdot D)$$

The first and the last term dominate this function, so the final computational complexity:

$$O(L^2 \cdot D)$$

The computational complexity in terms of L is:

$$O(L^2)$$

A quadratic computational complexity can become really problematic with long sequences. This is because the complexity grows quadratically and can lead to problems in terms of time and memory. This is why one must be careful when creating an algorithm and think about its computational complexity.

1.2 Feature map

For the McLaurin series expansion

$$\exp(t) = 1 + t + \frac{t^2}{2} + \ldots,$$

using only the first three terms, we must construct a feature map $\phi : \mathbb{R}^D \to \mathbb{R}^M$ so that $\exp\{\boldsymbol{q}^T\boldsymbol{k}\} \approx \phi(\boldsymbol{q})^T\phi(\boldsymbol{k})$. We should start by writing out the McLaurin

series:

$$\begin{aligned}\exp\left(\boldsymbol{q}^T\boldsymbol{k}\right) &\approx 1 + \boldsymbol{q}^T\boldsymbol{k} + \frac{\left(\boldsymbol{q}^T\boldsymbol{k}\right)^2}{2} \\ &= 1 + \sum_{i=1}^{D} q_i k_i + \frac{1}{2}\left(\sum_{i=1}^{D} q_i k_i\right)^2,\end{aligned} \tag{1}$$

where we have simply rewritten the scalar product in terms of a sum of vector component products in the second line. This gives us an idea for the construction of our feature map. We clearly need a constant term 1 to get the first term in equation (1). Secondly in order to compute the second term, we need a linear term of the input vector $\boldsymbol{x}$ in $\phi(\boldsymbol{x})$. To reconstruct the final term, we should write it out, to see which terms we need:

$$\frac{1}{2}\left(\sum_{i=1}^{D} q_i k_i\right)^2 = \sum_{i=1}^{D} \frac{(q_i k_i)^2}{2} + \sum_{i=1}^{D-1}\sum_{j=i+1}^{D} q_i k_i q_j k_j.$$

The first term is simply the product of squares of each components, while the second term represents all the mixed product terms one gets, when squaring a multi term expression. We now have all the terms and can write the feature map as

$$\phi(\boldsymbol{x}) = \begin{bmatrix} 1 & \boldsymbol{x}^T & \frac{(\boldsymbol{x}\odot\boldsymbol{x})^T}{\sqrt{2}} & x_1x_2 & \dots & x_1x_D & x_2x_3 & \dots & x_{D-1}x_D \end{bmatrix}^T, \tag{2}$$

where we have used the notation $\odot$ for element-wise multiplication of a vector. To express the dimensionality of the feature space M in terms of D, we must simply count the number of terms in the above expression with respect to D. The first term has of course dimension 1. The second term $\boldsymbol{x}^T$ has dimension D. Same goes for the third term, since it is still a D dimensional vector. For the mixed terms, however, it is a simple task of counting all the combinations of products, where the ordering doesn't matter. We can thus express it as a binomial coefficient $\binom{D}{2}$. Adding it all together, we get

$$M = 1 + D + D + \binom{D}{2} = 2D + 1 + \frac{D(D-1)}{2} = \frac{D^2 + 3D + 2}{2}. \tag{3}$$

To generalize the dimensionality expression for arbitrary number of terms K, we must make use of the expression for the number of terms in a power of a multinomial [1]

$$\binom{n+m-1}{m-1}.$$

Here n and m are the power and length of the multinomial, respectively. Applying this formula we can write the general expression as

$$M(D,K) = \sum_{k=0}^{K} \binom{K+D-1}{D-1}. \tag{4}$$

1.3 Softmax approximation

Now that we have constructed a feature map, we can approximate the softmax function by

$$\boldsymbol{Z} \approx \boldsymbol{D}^{-1}\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^T\boldsymbol{V}, \tag{5}$$

where

$$\boldsymbol{D} = \mathbf{Diag}\left(\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^T\mathbf{1}_L\right). \tag{6}$$

Lets prove this. We start by writing out the product

$$\boldsymbol{Q}\boldsymbol{K}^T = \begin{bmatrix}\boldsymbol{q}_1\\ \vdots \\ \boldsymbol{q}_D\end{bmatrix}\begin{bmatrix}\boldsymbol{k}_1 & \boldsymbol{k}_2 & \dots & \boldsymbol{k}_D\end{bmatrix} = \begin{bmatrix}\boldsymbol{q}_1^T\boldsymbol{k}_1 & \dots & \boldsymbol{q}_1^T\boldsymbol{k}_D\\ \vdots & \ddots & \vdots \\ \boldsymbol{q}_D^T\boldsymbol{k}_1 & \dots & \boldsymbol{q}_D^T\boldsymbol{k}_D\end{bmatrix},$$

where the indices now correspond to the matrix rows. Next we compute the softmax:

$$\begin{aligned}\sigma\left(\boldsymbol{Q}\boldsymbol{K}^T\right) &= \begin{bmatrix}\frac{\exp\left(\boldsymbol{q}_1^T\boldsymbol{k}_1\right)}{\sum_i^D \exp\left(\boldsymbol{q}_1^T\boldsymbol{k}_i\right)} & \dots & \frac{\exp\left(\boldsymbol{q}_1^T\boldsymbol{k}_D\right)}{\sum_i^D \exp\left(\boldsymbol{q}_1^T\boldsymbol{k}_i\right)}\\ \vdots & \ddots & \vdots \\ \frac{\exp\left(\boldsymbol{q}_D^T\boldsymbol{k}_1\right)}{\sum_i^D \exp\left(\boldsymbol{q}_D^T\boldsymbol{k}_i\right)} & \dots & \frac{\exp\left(\boldsymbol{q}_D^T\boldsymbol{k}_D\right)}{\sum_i^D \exp\left(\boldsymbol{q}_D^T\boldsymbol{k}_i\right)}\end{bmatrix}\\ &\approx \begin{bmatrix}\frac{\phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_1)}{\sum_i^D \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_i)} & \dots & \frac{\phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_D)}{\sum_i^D \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_i)}\\ \vdots & \ddots & \vdots \\ \frac{\phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_1)}{\sum_i^D \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_i)} & \dots & \frac{\phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_D)}{\sum_i^D \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_i)}\end{bmatrix}.\end{aligned} \tag{7}$$

Since it is now easier to show that equation (5) has the same form as the above expression we will rather conclude the proof in this direction. We start by writing the product

$$\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^T = \begin{bmatrix}\phi(\boldsymbol{q}_1)\\ \vdots \\ \phi(\boldsymbol{q}_D)\end{bmatrix}\begin{bmatrix}\phi(\boldsymbol{k}_1) & \phi(\boldsymbol{k}_2) & \dots & \phi(\boldsymbol{k}_D)\end{bmatrix} = \begin{bmatrix}\phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_1) & \dots & \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_D)\\ \vdots & \ddots & \vdots \\ \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_1) & \dots & \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_D)\end{bmatrix}.$$

Next we compute the diagonal matrix

$$\begin{aligned}\boldsymbol{D} &= \mathbf{Diag}\left(\begin{bmatrix}\phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_1) & \dots & \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_D)\\ \vdots & \ddots & \vdots \\ \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_1) & \dots & \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_D)\end{bmatrix}\begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}\right)\\ &= \mathbf{Diag}\left(\begin{bmatrix}\sum_i^D \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_i) & \dots & \sum_i^D \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_i)\end{bmatrix}^T\right)\\ &= \begin{bmatrix}\sum_i^D \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_i) & 0 & \dots & 0\\ 0 & \sum_i^D \phi(\boldsymbol{q}_2)^T\phi(\boldsymbol{k}_i) & \ddots & 0\\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & \sum_i^D \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_i)\end{bmatrix}.\end{aligned}$$

Since the inverse of a diagonal matrix is just a matrix with inverted diagonal elements we can now write the final multiplication as

$$\boldsymbol{D}^{-1}\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^T = \begin{bmatrix} \frac{\phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_1)}{\sum_i^D \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_i)} & \cdots & \frac{\phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_D)}{\sum_i^D \phi(\boldsymbol{q}_1)^T\phi(\boldsymbol{k}_i)} \\ \vdots & \ddots & \vdots \\ \frac{\phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_1)}{\sum_i^D \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_i)} & \cdots & \frac{\phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_D)}{\sum_i^D \phi(\boldsymbol{q}_D)^T\phi(\boldsymbol{k}_i)} \end{bmatrix}.$$

This matches the expression (7), thus we have completed the proof. Note that we didn't multiply by $\boldsymbol{V}$, but since it is written in both expressions on the right we really only had to show, that $\sigma((Q)\boldsymbol{K}^T) \approx \boldsymbol{D}^{-1}\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^T$.

1.4 Computational complexity of the approximation

The approximation determined in section 1.3 can be exploited to make the computational complexity smaller. Let's start with the calculation of computational complexity of

$$\mathbf{D} = \mathbf{Diag}(\phi(\mathbf{Q})\phi(\mathbf{K})^T\mathbf{1}_L).$$

The size of the matrices $\phi(\mathbf{Q})$ and $\phi(\mathbf{K})$ are $L \times M$, and $\mathbf{1}_L$ is a L dimensional vector. If we first do the multiplication of $\phi(\mathbf{K})^T\mathbf{1}_L$, the computational complexity is $O(ML)$, as it's a multiplication of a $M \times L$ matrix with a $L \times 1$ vector. The result is a M dimensional vector. Now the next step is a multiplication of matrix $\phi(\mathbf{Q})$ with the shape $L \times M$, with the vector of size M. The computational complexity this action has $O(LM)$ and the resulting output of it is a vector with of size L. The next step is to make this vector into a diagonal matrix called $\mathbf{D}$. The computational complexity of this action is O(L), and the resulting matrix has the shape of $L \times L$. The last step is the take the inverse of matrix $\mathbf{D}$ to achieve $\mathbf{D}^{-1}$. The computational complexity of inverting a diagonal matrix is O(L).
Now that we have determined the computational complexity of $\mathbf{D}^{-1}$ and it's size, we can compute the whole computational complexity of $\mathbf{Z}$ in

$$\mathbf{Z} \approx \mathbf{D}^{-1}\phi(\mathbf{Q})\phi(\mathbf{K})^T\mathbf{V}.$$

We start by determining the computational complexity of

$$\phi(\mathbf{K})^T\mathbf{V}$$

Again the complexity of a matrix multiplication of matrices with size $M \times L$ and $L \times D$ is $O(LMD)$. This is because for each element in the result matrix, you need to perform L multiplication and L-1 additions, and there are $M \cdot D$ elements in the result matrix. As mentioned the outcome of this is a matrix with the shape $M \times D$. Now if we have a multiplication of $\phi(\mathbf{Q})$ with the obtained matrix, the computational complexity is still $O(LMD)$ and the output matrix has the shape $L \times D$. The last step is to compute the computational complexity of the

multiplication of $\mathbf{D}^{-1}$ with the resulting matrix that has the shape of $L \times D$. Since the matrix $\mathbf{D}^{-1}$ is a diagonal matrix, all of its entries are zero except for the ones in the main diagonal. Therefore when doing this multiplication you only need to perform element-wise multiplication between the non-zero diagonal entries of $\mathbf{D}^{-1}$ and the corresponding columns of result matrix. This reduces the computational complexity to O(LD) and the output shape of the matrix is $L \times D$.
The overall computational complexity of the whole calculation is

$$O(ML + LM + L + L + LMD + LMD + LD)$$

which simply has the complexity of

$$O(LMD).$$

The result has a computational complexity, which is linearly dependent on L, M and D.

2 Question 2

2.1 Simple convolutional network with max-pooling layers

In this section the goal is to implement a simple convolutional neural network. The structure of the network is showcased in Table 1. In the OCTMNIST dataset the images are of grayscale so the initial input shape is 28x28x1. The dataset also consists of 4 classes which is why the last affine tranformation layer has an output shape of 4. The first task of building this CNN was to determine how much the padding has to be in the initial convolutional layer so that the image keeps the shape of 28x28. By using the the formula of $\frac{Height - Kernel + 2 \cdot Padding}{Stride} + 1$, the padding size could be determined to be 1. The other task of building this CNN was to calculate the input shape of the initial affine tranformation layer which is 576, this is calculated by multiplying all the dimensions together in the last max pooling layers output shape ($6 \cdot 6 \cdot 16 = 576$).

Layer	Kernel size	Stride	Padding	Output shape
2D Convolutional	3x3	1	1	28x28x8
ReLu Activation	-	-	-	28x28x8
Max pooling	2x2	2	-	14x14x8
2D Convolutional	3x3	1	0	12x12x16
ReLu Activation	-	-	-	12x12x16
Max pooling	2x2	2	-	6x6x16
Flatten	-	-	-	576
Affine transformation	-	-	-	320
ReLu activation	-	-	-	320
Dropout	-	-	-	320
Affine tranformation	-	-	-	120
ReLu activation	-	-	-	120
Affine tranformation	-	-	-	4
LogSoftmax activation	-	-	-	4

Table 1: The structure of a simple convolutional neural network with max pooling layers

When training this CNN, the optimizer utilized is stochastic gradient descent, and training was done in 15 epochs. The learning rate was varied between runs for the following values: 0.1, 0.01, and 0.001, and the results of these runs is showcased in Table 2. All the configurations performed relatively well, but the one with learning rate set to 0.01 clearly stood out by having easily the best accuracy in validation and test. The plots of training loss and validation accuracy are showcased in Figure 1 for the best configuration (lr = 0.01).

Learnin rate	Final validation accuracy	Best validation accuracy	Final test accuracy
0.1	0.8429	0.8429	0.8072
0.01	**0.8755**	**0.8755**	**0.8204**
0.001	0.7984	0.7984	0.7977

Table 2: Comparison of configurations with different learning rates

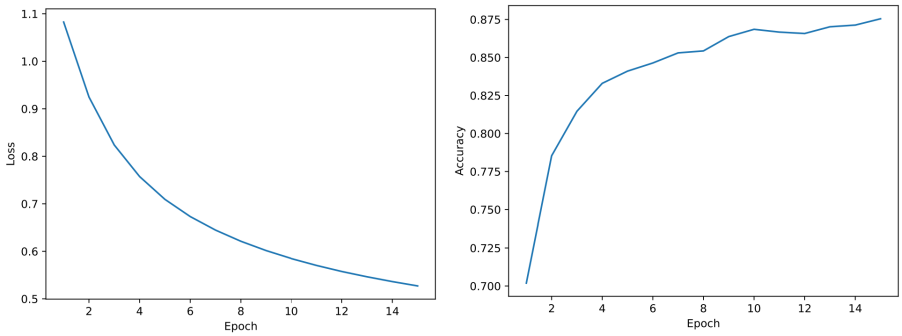

Figure 1: Plots of training loss (left) and validation accuracy (right) for a configuration with 0.01 learning rate

2.2 Simple CNN without max-pooling layers

The goal of this section is to implement similar CNN as in section 2.1 but this time with no maxpooling layers. Because there are no maxpooling layers the convolutional layers had to be changed. Now the first convolutional layer has stride of 2 and padding of 1, and the second convolutional layer has a stride of 2 and padding of 0. All the other layers and values are kept the same as before. The structure of the CNN is showcased in Table 3.

Layer	Kernel size	Stride	Padding	Output shape
2D Convolutional	3x3	2	1	14x14x8
ReLu Activation	-	-	-	14x14x8
2D Convolutional	3x3	2	0	6x6x16
ReLu Activation	-	-	-	6x6x16
Flatten	-	-	-	576
Affine transformation	-	-	-	320
ReLu activation	-	-	-	320
Dropout	-	-	-	320
Affine tranformation	-	-	-	120
ReLu activation	-	-	-	120
Affine tranformation	-	-	-	4
LogSoftmax activation	-	-	-	4

Table 3: The structure of a simple convolutional neural netork with no max pooling layers

In section 2.1 we determined that the best configuration was the one with a learning rate of 0.01. The same configuration performed worse without the max-pooling layers, as can be seen from Table 4. The plots of training loss and validation accuracy are showcased in Figure 2 for the configuration with a learning rate of 0.01.

Model	Final validation accuracy	Best validation accuracy	Final test accuracy
With max-pooling	0.8755	0.8755	0.8204
Without max-pooling	0.8444	0.8444	0.8015

Table 4: Comparison of models with max-pooling vs without max-pooling

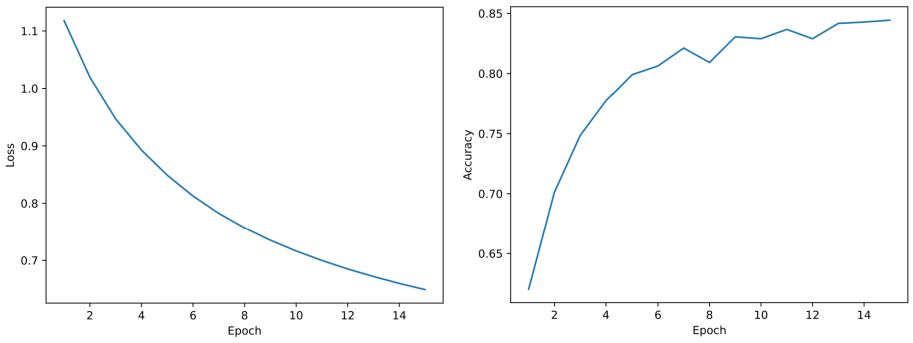

Figure 2: Plots of training loss (left) and validation accuracy (right) for a configuration with 0.01 learning rate and no max-pooling layers

2.3 Trainable parameters

The number of trainable parameters in both models with and without max-pooling layers is 224892. The similar number of trainable parameters in both models indicates that the difference in performance is not attributed to the model's capacity alone. The key factor contributing to the superior performance of the model with max-pooling layers lies in the way these layers operate.

Max-pooling layers are good at capturing and emphasizing the most important aspects in input data. Max-pooling assists the model in focusing on the most discriminative characteristics of the images by preserving only the highest values and discarding less significant information. This selective processing is very useful in image classification applications, where recognizing critical patterns or distinguishing characteristics is critical.

Furthermore, the addition of max-pooling layers gives the model some translation invariance. This characteristic enables the network to recognize patterns regardless of their exact placement in the input, making the model more resistant to spatial fluctuations within the images. This intrinsic translation invariance is very useful when working with photos containing objects or features in changing places.

3 Question 3

3.1 RNN based architecture

Lets first look at the loss function of the training and validation sets for the RNN based decoder architecture in figure 3. We see that it behaves like we would expect, as it decreases with the epoch number and it slightly plateaus. The validation set has a similar loss function. Next we plot all the different string similarity scores. In figure 4 we show the Jaccard similarity score, in figure 5 we show the Cosine similarity score, while in figure 6 the Damerau-Levenstein similarity can be seen. We can see that all the scores improve with the epoch number. The Cosine score is the highest, followed by the Jaccard score and finally the Damerau-Levenstein similarity score is the lowest. The final test scores and the loss function are compiled in table 5.

test set quantity	**loss**	**Jaccard score**	**Cosine score**	**Damerau-Levenstein score**
value	1.183	0.715	0.832	0.509

Table 5: Final test set value for the RNN decoder arcitecture.

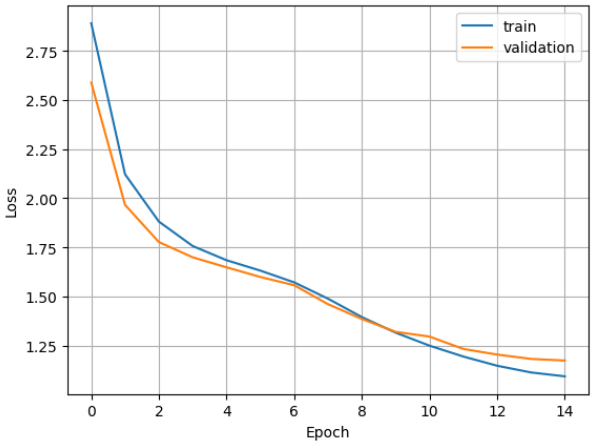

Figure 3: Training and validation loss function with respect to the epoch number for the RNN architecture.

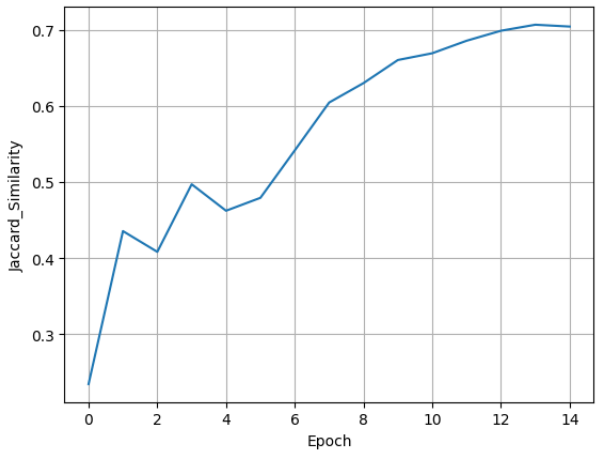

Figure 4: Jaccard string similarity score with respect to the epoch number for the RNN architecture.

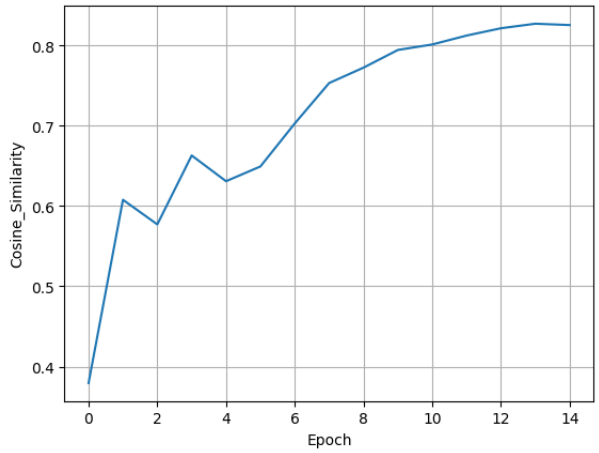

Figure 5: Cosine string similarity score with respect to the epoch number for the RNN architecture.

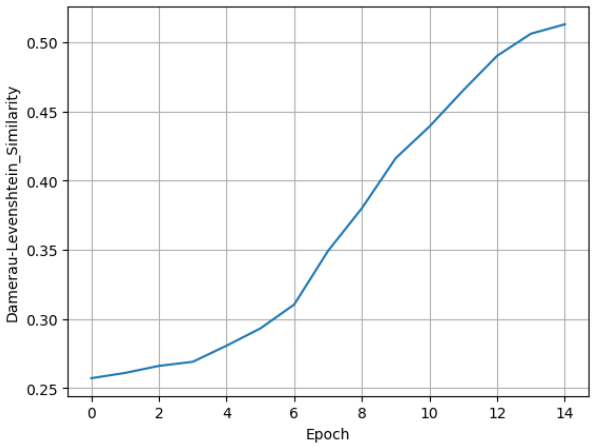

Figure 6: Damerau-Levenstein string similarity score with respect to the epoch number for the RNN architecture.

3.2 Attention based architecture

Now we switch the decoder to an attention based architecture. We again start by plotting the loss evolution with respect to the epoch number in figure 7. We see that it yet again has the typical loss function shape, both for the training and validation sets. Next we can observe the similarity scores in figures 8, 9 and 10 for the Jaccard, Cosine and Damerau-Levenstein scores, respectively. The cosine scores are again the highest, while the Damerau-Levenstein scores are the lowest. The final test set scores are shown in table 6

test set quantity	**loss**	**Jaccard score**	**Cosine score**	**Damerau-Levenstein score**
value	1.160	0.765	0.866	0.633

Table 6: Final test set values for the attention decoder arcitecture.

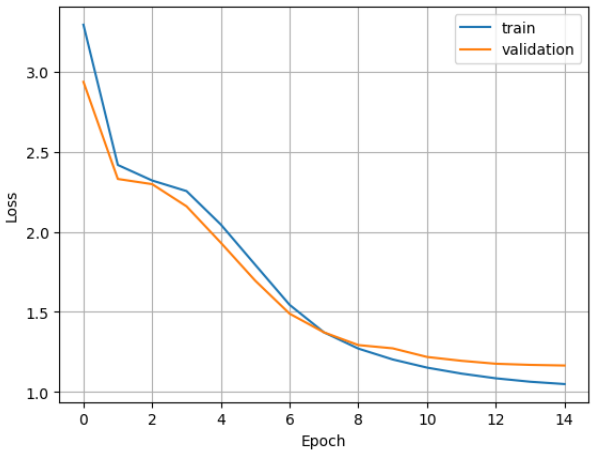

Figure 7: Training and validation loss function with respect to the epoch number for the attention architecture.

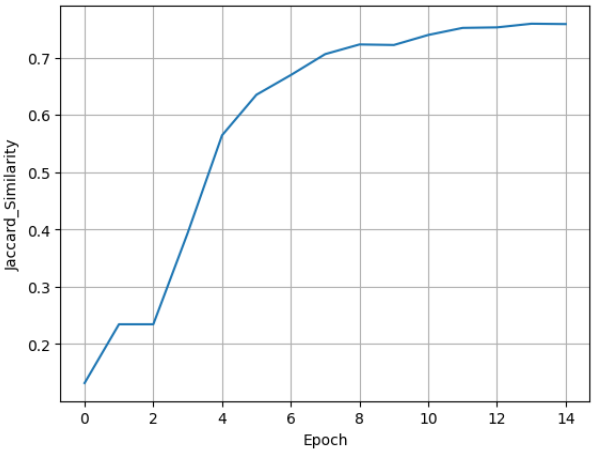

Figure 8: Jaccard string similarity score with respect to the epoch number for the attention architecture.

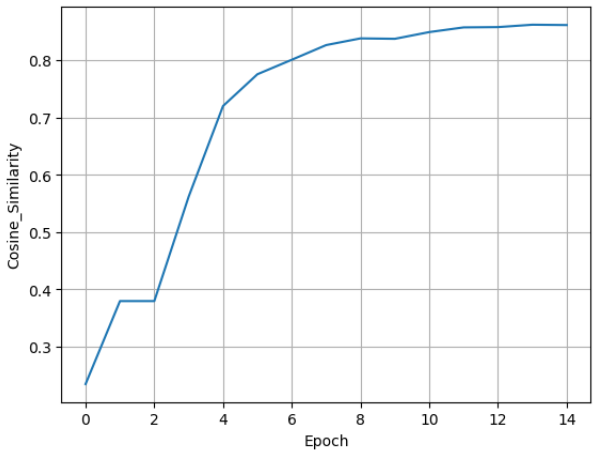

Figure 9: Cosine string similarity score with respect to the epoch number for the attention architecture.

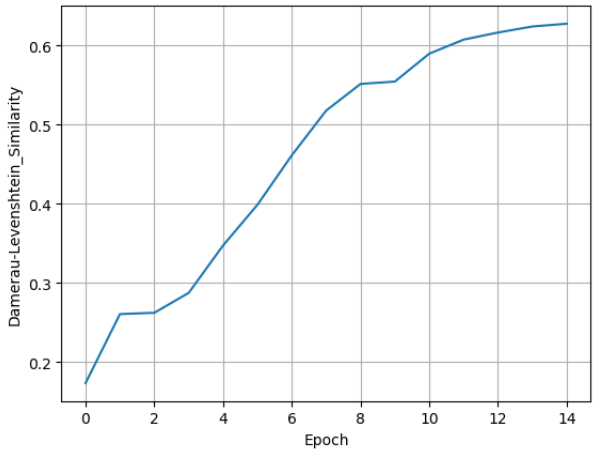

Figure 10: Damerau-Levenstein string similarity score with respect to the epoch number for the attention architecture.

3.3 Architecture comparison

As we can see from tables 5 and 6 the attention based architecture gives slightly better results compared to the RNN decoder. The LSTM processes texts sequentially. The attention mechanism, however, computes the scores for all tokens at the same time. From the lectures we have learned, that this allows the decoder to better focus on different parts of the input.
LSTMs suffer from the constraint, that all input sequences are forced to be

encoded into a fixed length vector. This can affect their performances, especially for longer sentences [2]. The attention mechanism, however, does not fall to the requirement of encoding a whole input sentence to a fixed length vector [3]. Thus the improvement in text encoding is especially noticeable in longer sentences. We could therefore conclude, that our dataset perhaps had longer sentences, since the attention based architecture showed a slight improvement in the predictions. Moreover, it is possible, that the test sentences were in average longer then the ones used to train the model, since that also impacts the performance of LSTMs [2]. It is worth noting, however, that the simultaneous processing of attention mechanisms is more expensive in terms of computational complexity.

3.4 String similarity scores comparison

The three string similarity scores also varied in both decoder architectures. The cosine string similarity score measures the cosine of the angle between two non zero vectors and expresses their orientation rather than magnitude [4]. The Jaccard similarity is quantified by comparison of the shared elements with respect to the total elements of two sets (in this case strings) combined [4]. The Damerau-Levenshtein similarity measures the number of operations needed to transform one string into another [4]. This means it compares strings based on characters rather than words, unlike the Jaccard similarity.
The difference in score numbers suggests that perhaps there weren't as many shared elements between the sets, since the Jaccard score is slightly lower compared to the cosine similarity score. The high cosine score would suggest that there was a high frequency of words. The lower Damerau-Levenshtein score suggests, that many transformations were required to compare the strings.

References

[1] Wikipedia: Multinomial theorem. Accessed 5.1.2024.

[2] J. Brownlee. Attention in long short-term memory recurrent neural networks, September 2022. Accessed 5.1.2024.

[3] D. Bahdanau and K. H. C. Y. Bengio. Neural machine translation by jointly learning to align and translate. arXiv:1409.0473 [cs.CL] (2016).

[4] Y. Khal. The complete guide to string similarity algorithms, August 2023. Accessed 6.1.2024.