TÉCNICO LISBOA

Deep Learning

Electrical and Computer Engineering

Homework 2- Report

Students:

Alexandre Reis (100123) alexandre.leite.reis@tecnico.ulisboa.pt
Duarte Morais (100163) duarte.morais@tecnico.ulisboa.pt

Group 49

2023/2024 – First Semester, Q2

Contribution of each member of the group in this project: Question 2 was solved entirely (code, experimental results and report) by Duarte Morais (student 100163); Question 3 was solved entirely (code, experimental results and report) by Alexandre Reis (student 100123); Question 1 was solved partially by both members of the group.

1 Question 1

1.1 Question 1.1

The computational complexity of the self-attention operation $Z = Softmax(QK^T)V$ is $O(L^2D)$ where L is the sequence length and D is the hidden size dimension which is the size of the vector that represents each word. This quadratic complexity happens because the dot product between Q and K^T involves $L \times D$ elements for each of the L words, repeated for each word in the sequence, leading to $L \times D$ operations.

Since we only want the computational complexity of Z in terms of the sequence length then the the complexity is $O(L^2)$.

For long sequences, this becomes a problem because the number of required operations increase quickly (quadraticly) as the sequence length increases. This can lead to significant computational demands, slower processing times, and increased memory usage, all of which are impractical for real-world applications that may require the processing of very long sequences.

1.2 Question 1.2

Given the McLaurin series expansion for the exponential function:

$$\exp(t) = \sum_{n=0}^{\infty} \frac{t^n}{n!} = 1 + t + \frac{t^2}{2} + \frac{t^3}{6} + \dots$$

Using only the first three terms, we approximate $\exp(t) \approx 1 + t + \frac{t^2}{2}$.

For arbitrary vectors $q, k \in \mathbb{R}^D$, we have $\exp(q^T k) \approx \phi(q)^T \phi(k)$, where the feature map $\phi : \mathbb{R}^D \to \mathbb{R}^M$ can be defined as:

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ \frac{1}{\sqrt{2}} \odot x^2 \end{bmatrix} \tag{2}$$

due to the fact that:

$$\phi(q)^T \phi(k) = \begin{bmatrix} 1 & q^T & \frac{1}{\sqrt{2}} q^{T2} \end{bmatrix} \begin{bmatrix} 1 \\ k \\ \frac{1}{\sqrt{2}} q^{T2} \end{bmatrix} = 1 + q^T k + \frac{1}{2}(q^T k)^2$$

In the equation 2 x^2 denotes the element-wise square of x, and $\odot$ is the element-wise product. The dimensionality M of the feature space where $\phi(x)$ lives is $1 + D + D = 2D + 1$ because we have one constant term, D linear terms, and D quadratic terms.

Extending the series to the K-th order, M would be the sum of D terms for each power up to K , plus the constant term, resulting in the follow:

$$M = 1 + D + D + \cdots + D = 1 + DK$$

This represents the total number of features generated from the original D-dimensional space by including all terms up to the K-th power in the series expansion.

1.3 Question 1.3

To show that $Z \approx D^{-1}\Phi(Q)\Phi(K)^T V$, it's possible to start by applying the approximation from 1.2 to the original self-attention operation:

$$\text{softmax}(QK^T)V = \begin{bmatrix} \text{softmax}(q_1k_1^T) & \text{softmax}(q_1k_2^T) & \dots & \text{softmax}(q_1k_L^T) \\ \text{softmax}(q_2k_1^T) & \text{softmax}(q_2k_2^T) & \dots & \text{softmax}(q_2k_L^T) \\ \vdots & \vdots & \ddots & \vdots \\ \text{softmax}(q_Lk_1^T) & \text{softmax}(q_Lk_2^T) & \dots & \text{softmax}(q_Lk_L^T) \end{bmatrix} V \tag{3}$$

Continuing to develop the softmax matrix leads to:

$$\begin{bmatrix} \frac{e^{q_1k_1^T}}{\sum_{j=1}^{L} e^{q_1k_j^T}} & \cdots & \frac{e^{q_1k_L^T}}{\sum_{j=1}^{L} e^{q_1k_j^T}} \\ \vdots & \ddots & \vdots \\ \frac{e^{q_Lk_1^T}}{\sum_{j=1}^{L} e^{q_Lk_j^T}} & \cdots & \frac{e^{q_Lk_L^T}}{\sum_{j=1}^{L} e^{q_Lk_j^T}} \end{bmatrix} \tag{4}$$

Applying the approximation $\exp(q^Tk) \approx \phi(q)^T\phi(k)$ as $\exp(qk^T) \approx \phi(q)\phi(k)^T$:

$$\begin{bmatrix} \frac{\phi(q_1)\phi(k_1)^T}{\sum_{j=1}^{L} \phi(q_1)\phi(k_j)^T} & \cdots & \frac{\phi(q_1)\phi(k_j)^T}{\sum_{j=1}^{L} \phi(q_1)\phi(k_j)^T} \\ \vdots & \ddots & \vdots \\ \frac{\phi(q_L)\phi(k_1)^T}{\sum_{j=1}^{L} \phi(q_L)\phi(k_j)^T} & \cdots & \frac{\phi(q_L)\phi(k_L)^T}{\sum_{j=1}^{L} \phi(q_L)\phi(k_j)^T} \end{bmatrix} \tag{5}$$

Using the vector $\Phi(Q)\Phi(K)^T\mathbf{1}_L$ to compose matrix D:

$$\Phi(Q)\Phi(K)^T\mathbf{1}_L = \begin{bmatrix} \phi(q_1) \\ \phi(q_2) \\ \vdots \\ \phi(q_L) \end{bmatrix} \begin{bmatrix} \phi(k_1)^T & \phi(k_2)^T & \dots & \phi(k_L)^T \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \tag{6}$$

The matrix $D = \text{Diag}(\Phi(Q)\Phi(K)^T\mathbf{1}_L)$ is then equivalent to a diagonal matrix:

$$\begin{bmatrix} \sum_{j=1}^{L} \phi(q_1)\phi(k_j)^T & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sum_{j=1}^{L} \phi(q_L)\phi(k_j)^T \end{bmatrix} \tag{7}$$

and D^{-1} is obtained by inverting each element on its diagonal. Multiplying the inverted D matrix by $\Phi(Q)\Phi(K)^T$, results in the follow:

$$\begin{bmatrix} \frac{\phi(q_1)\phi(k_1)^T}{\sum_{j=1}^{L}\phi(q_1)\phi(k_j)^T} & \cdots & \frac{\phi(q_1)\phi(k_j)^T}{\sum_{j=1}^{L}\phi(q_1)\phi(k_j)^T} \\ \vdots & \ddots & \vdots \\ \frac{\phi(q_L)\phi(k_1)^T}{\sum_{j=1}^{L}\phi(q_L)\phi(k_j)^T} & \cdots & \frac{\phi(q_L)\phi(k_L)^T}{\sum_{j=1}^{L}\phi(q_L)\phi(k_j)^T} \end{bmatrix} \tag{8}$$

So, as the matrix 8 and 5 are equal, the aproximation is proven to be correct.

1.4 Question 1.4

The computational complexity of the self-attention mechanism, defined by the equation

$$Z = D^{-1}\Phi(Q)\Phi(K)^T V, \tag{9}$$

is calculated, where D^{-1}, the inverted diagonal matrix, is computed as:

$$D^{-1} = \text{Diag}(\Phi(Q)\Phi(K)^T \mathbf{1}_L). \tag{10}$$

In this context, $\Phi(Q)$ and $\Phi(K)$ are the feature maps of the queries and keys, respectively, with V representing the values matrix.

The associative property of matrix multiplication is employed to strategically order operations and minimize computational complexity. Initially, $\Phi(K)^T\mathbf{1}_L$ from 10 is computed with a complexity of $O(M \times L)$, involving M multiplications for each of the L elements in vector $\mathbf{1}_L$.

The multiplication of this result by $\Phi(Q)$ does not alter the complexity, maintaining it at $O(M \times L)$. The time complexity of diagonalizing and inverting matrix D is $O(L)$, dominated by the preceding multiplication complexity.

Further, the multiplication of D^{-1} by $\Phi(Q)$, resulting in $C = D^{-1}\Phi(Q)$, retains a complexity of $O(L \times M)$.

Subsequently, the computation of $E = \Phi(K)^T V$ incurs a complexity of $O(M \times L \times D)$, considering the multiplication across L rows and D columns.

The final product $Z = CE$ results in a complexity of $O(L \times M \times D)$, which is equivalent to $O(L \times D^2)$ and $O(L \times M^2)$, given that $M = 1 + kD$.

In conclusion, the approximation enables the self-attention operation to be computed with linear complexity in terms of the sequence length L, while being influenced quadratically by the feature space and input space dimensions M and D, respectively.

2 Question 2

In this section, the study focuses on how Convolutional Neural Networks (CNNs) classify OCTMNIST dataset images with four different categories. The problem involve comparing CNNs with and without max-pooling layers. The parameters like learning rates (0.1, 0.01, 0.001) over 15 epochs were tested, using Stochastic Gradient Descent for optimization, with a dropout rate set at 0.7. The findings are shown in tables and graphs bellow, detailing training loss, validation accuracy, and test accuracy for each setup.

2.1 Question 2.1 - CNN with Max Pooling

In this subsection, the performance of CNNs with max pooling layers is assessed. The focus is on how the network behaves with different learning rates: 0.1, 0.01, and 0.001 over 15 epochs.

The equation used to calculate the size after each layer was the following:

$$(N–F+2P)/S+1 \tag{11}$$

where $N*N*D$ is a given image, $F*F*D$ is a given filter and the output will be size $M*M*K$, with K being the number of channels.

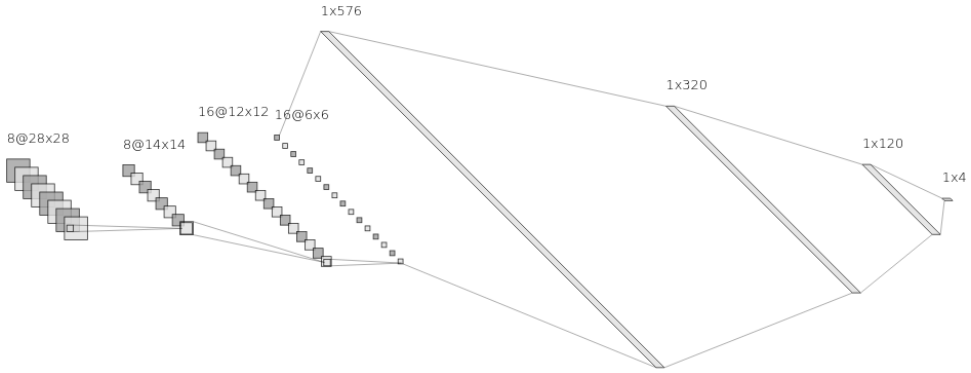
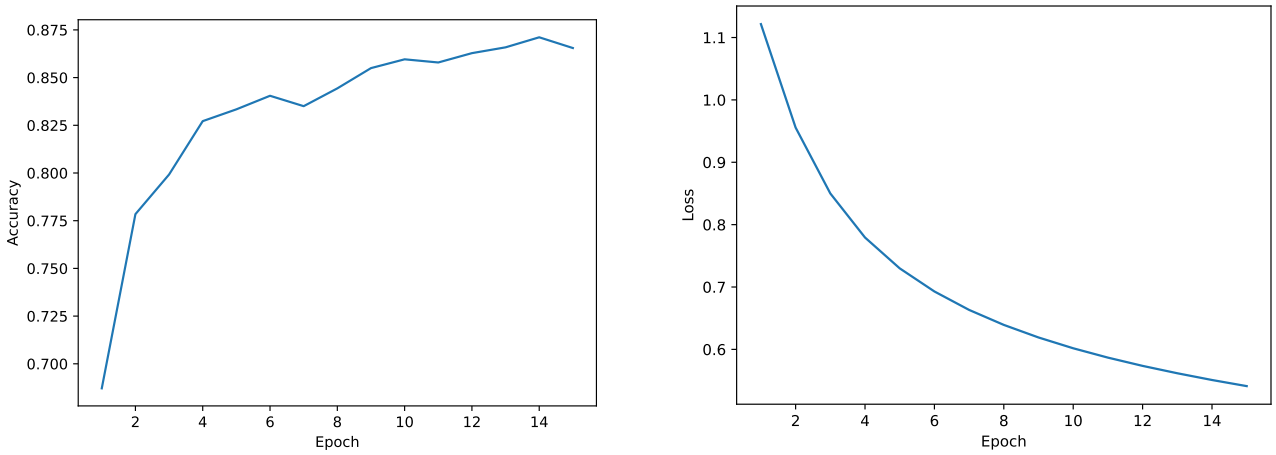


Figure 1: Architecture of the Convolutional Neural Network for 2.1

So for this question the image size evolved from the original 28x28 to 28x28 after the first convolution, then to 14x14 after the first max pooling, followed by 12x12 after the second convolution, and finally to 6x6 after the second pooling. The number of layers of the image also changed.



(a) Validation accuracy **(b)** Training loss

Figure 2: Plots of validation set scores over epochs - using "TextDecoderRecurrent"

Table 1: CNN Performance with Max Pooling

Learning Rate	Training Loss	Validation Accuracy	Test Accuracy	Trainable Parameters
0.1	0.6148	0.8192	0.7713	224892
0.01	0.5411	0.8655	0.8223	224892
0.001	0.9560	0.7720	0.7580	224892

2.2 Question 2.2 - CNN without Max Pooling

This subsection deals with CNNs lacking max pooling layers. Similar to the previous section, the learning rates tested are 0.1, 0.01, and 0.001 across 15 epochs.

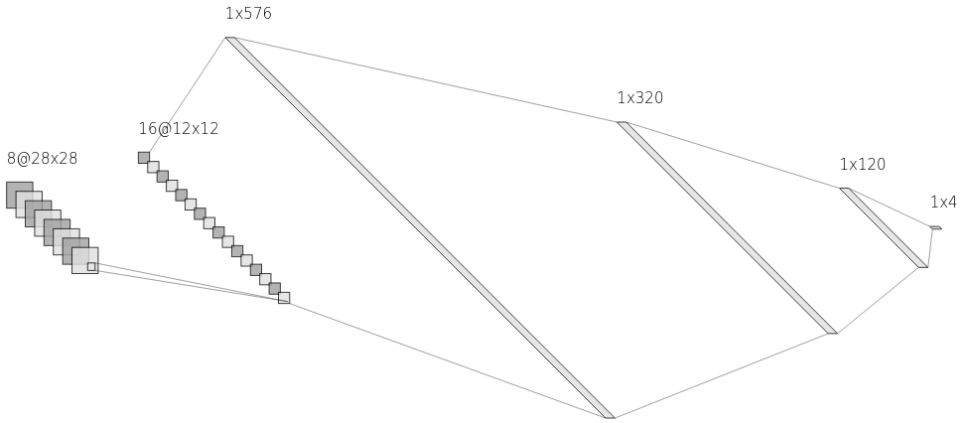


Figure 4: Architecture of the Convolutional Neural Network for Q2.2

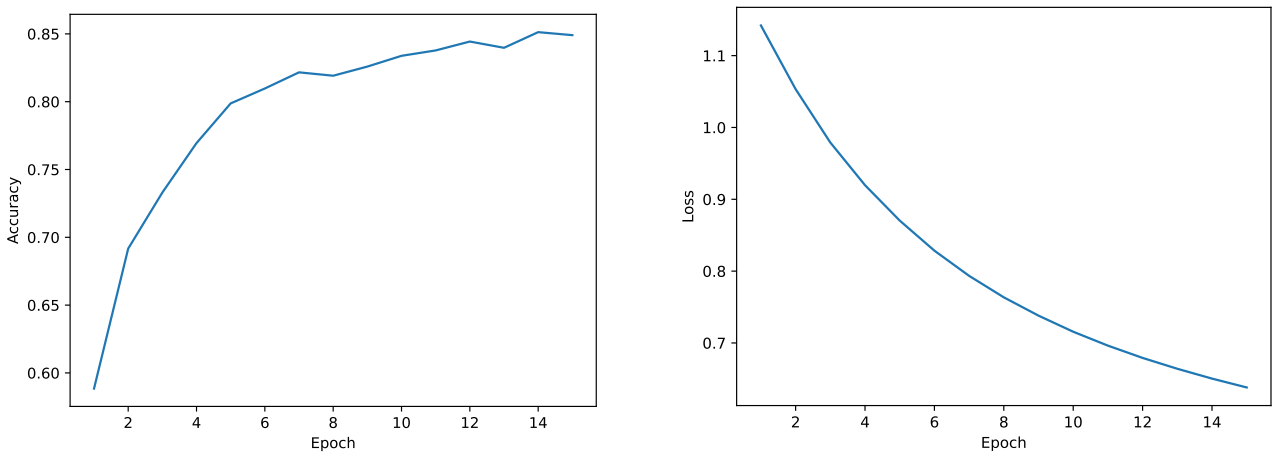
Applying the equation 11 once again, the image size evolved from the original 28x28 to 14 after the first convolution, then to 6 after the second convolution with 16 channels, that's why the fully connected layer as 576 elements (16*6*6).

Table 2: CNN Performance without Max Pooling

Learning Rate	Training Loss	Validation Accuracy	Test Accuracy	Trainable Parameters
0.1	0.6571	0.7971	0.7543	224892
0.01	0.6381	0.8491	0.7996	224892
0.001	1.0510	0.6889	0.7127	224892

2.3 Question 2.3 - Discussion and Conclusion

From the visualization of the table 1 and 2, it was clear that using max pooling made the models with a better accuracy. Max pooling simplifies what the CNN has to look at by making the image representations smaller and easier to manage. This process also helps the model ignore small changes like shifts, turns, or resizing in the images, which makes the model more reliable when it sees new, slightly different images.



(a) Validation accuracy **(b)** Training Loss

Figure 5: Plots for the best configuration, learning rate - 0.01

Moreover, it was noted that the number of trainable parameters consistently remained at 224892 for both variants of the CNN. This uniformity suggests that the observed differences in network performance were more likely a consequence of architectural decisions rather than the complexity of the models. The consistent parameter count across various configurations provided a uniform scenario to perceive the impact of max pooling on model performance. It was therefore concluded that CNNs demonstrate superior performance when their architecture alternate between convolutional and pooling layers.

Finally, the learning rate of 0.01 was found to be the most effective. It allowed the network to learn quickly and in a controlled way, without the risk of getting stuck in less optimal solutions or skipping over the best ones. This learning rate was just right, making sure the models didn't learn too quickly or too slowly. With a learning rate of 0.01, the models improved steadily and could perform well.

3 Question 3

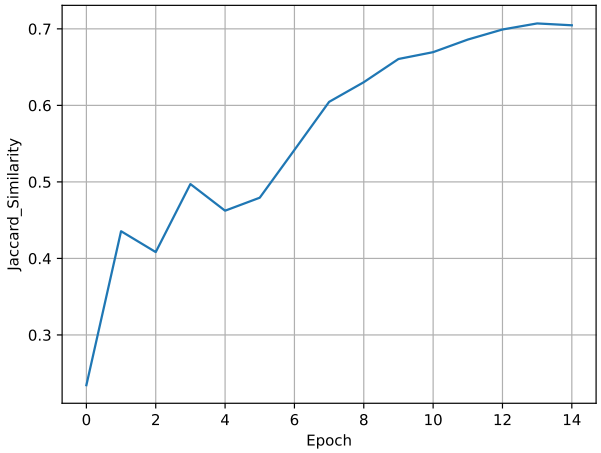
3.1 Question 3.1

For this question, the provided "TextDecoderRecurrent" decoder was used (provided in the skeleton code). This decoder's foward method was completed, using the LSTM arquitecture.

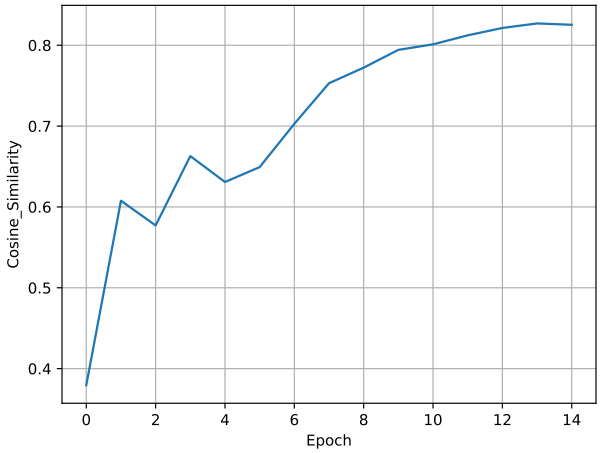
	Jaccard similarity	**Cosine similarity**	**Damerau-levenshtein similarity**	**Loss**
Test set	0.715	0.832	0.509	1.183
Validation set (last epoch)	0.705	0.825	0.513	1.174

Table 3: Scores for Loss and String Similarities - "TextDecoderRecurrent"

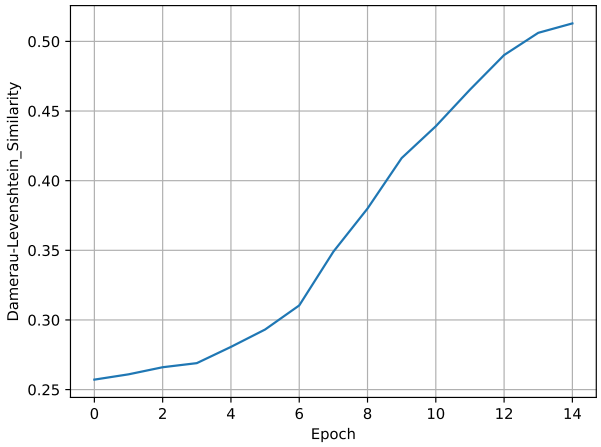
The results of training the model with this recurrent decoder are presented in this section.(table 3 and figure 6)



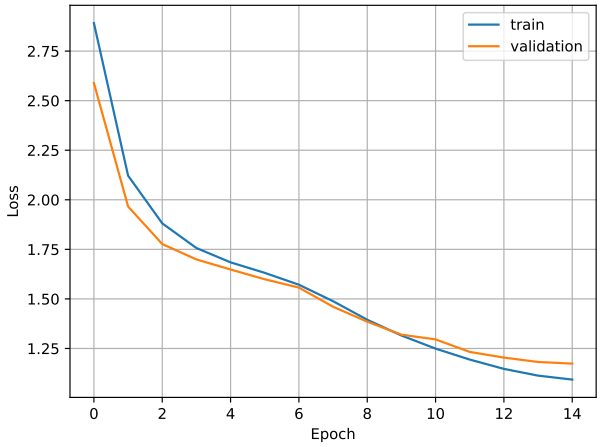
(a) Jaccard string similarity



(b) Cosine string similarity



(c) Damerau Levenshtein string similarity



(d) Training and validation sets Loss values

Figure 6: Plots of validation set scores over epochs - using "TextDecoderRecurrent"

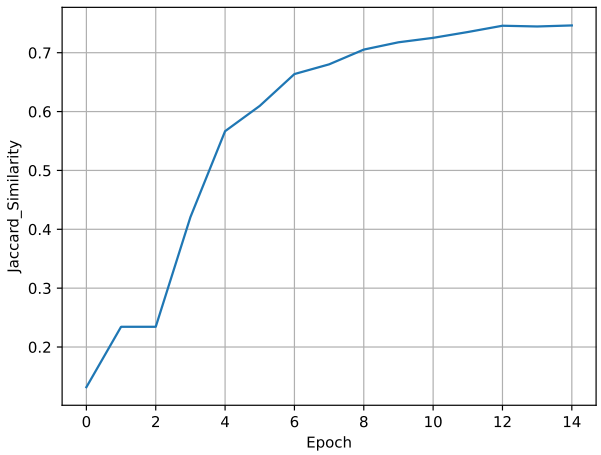
3.2 Question 3.2

For this question, the provided "TextDecoderTramsformer" decoder was used (provided in the skeleton code). This decoder's foward method was completed, using and attention based mecanism.

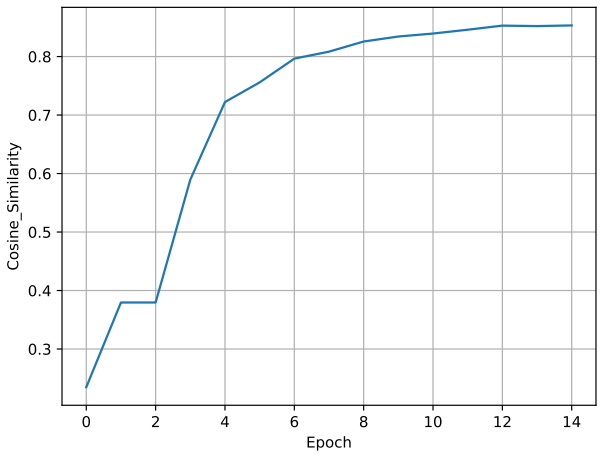
	Jaccard similarity	Cosine similarity	Damerau-levenshtein similarity	Loss
Test set	0.766	0.866	0.633	1.160
Validation set (last epoch)	0.756	0.860	0.624	1.165

Table 4: Scores for Loss and String Similarities - "TextDecoderTramsformer"

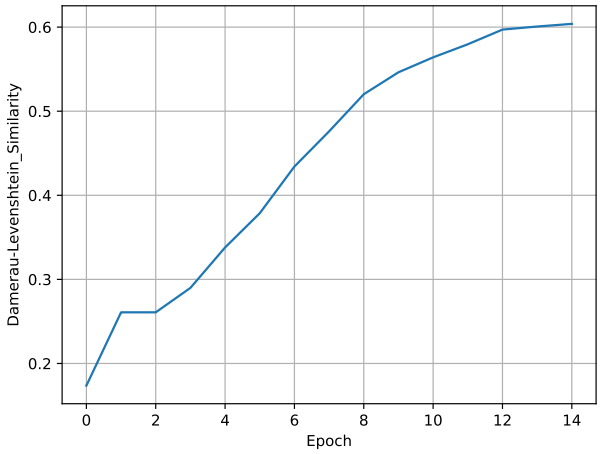
The results of training the model with this recurrent decoder are presented in this section. (table 4 and figure 7)



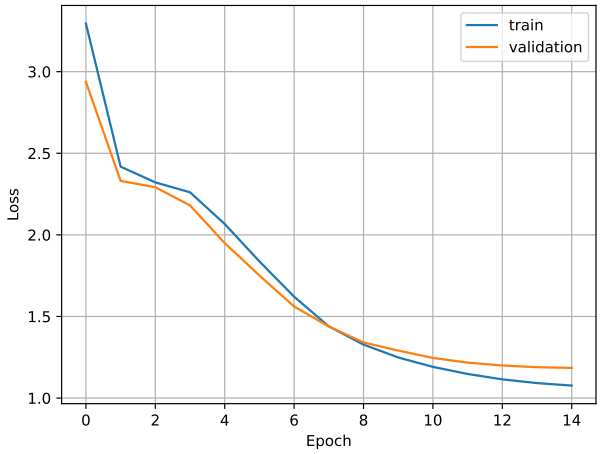
(a) Jaccard string similarity



(b) Cosine string similarity



(c) Damerau Levenshtein string similarity



(d) Training and validation sets Loss values

Figure 7: Plots of validation set scores over epochs - using "TextDecoderTramsformer"

3.3 Question 3.3

Both of the decoders are used for sequential data, which means that for each word generated, they take as input not only information that comes out of the encoder but also the output that has already been created by that same decoder. [2]

Long Short-Term Memory (LSTM) based decoders are a type of recurrent neural networks that solve the vanishing gradient problem of normal RNNs. This is a problem that affects long-range dependencies in RNN, as the gradient becomes very small when multiplying it back in time. In the case of Language models, this problem mainly appears when dealing with longer amounts of text. LSTMs use memory cells, and substitute gradient multiplication back in time, with addition. This enables them to consider long-range dependencies without the vanishing gradient problem.

A LSTM takes as input a sequence of tokens. This is is fed into the LSTM sequentially, token by token. As each token if fed, the LSTM also considers it's state from the previous step. In each of these steps, the LSTM produces an output than should be interpreted by a fully

connected layer with softmax activation, in order to conclude on the most likely word.

In the case of this LSTM based decoder architecture, a residual attention block (with Multi-head attention) is also used. This makes the decoder only take into consideration the relevant parts of the input in each forward pass.

The main problem with LSTMs is that their long-range dependencies are still tricky (even with memory cells) and training them prohibits parallelization, due to their sequential mechanism. [4]

The long-range dependencies can be solved with the included residual attention block that is added to the model of Question 3.1.

Transformer based decoders, like the one in question 2 also make use of attention mechanisms to be able to focus on any part of the input, which makes them very good at dealing with long-range dependencies. Besides this, their training can be parallelized, making better use of matrix multiplication.

This parallelization should make transformers train in less time [2]. But in practise, the model using LSTMs was a few seconds faster to train (using Kaggle with a GPU - 914.6s vs 937.1s).

The fact that both models are good with long-range dependencies, should make them behave similarly when longer strings are presented. This should also make them consider and be as good with both the beginning middle and the end part of the sentence, in relation to a 1 directional LSTMs without attention mechanism.

Analysing the test predictions after epoch 15, it is possible to see that there is no visible difference between the two models in predicting different parts of the sentence.

In the case of Transformers, the encoders output tokens is not fed to the decoder sequentially. This means that the decoder has not sense of order/position of words in a sentence. For that reason, transformers consider a encoded position matrix alongside the tokens, which differs from what the LSTM model does (feds tokens in order).

The attention mechanism on Transformer decoders are composed not only on (residual cross) attention of the encoder's state, but also of self-attention. Self-attention enables the decoder to use an attention mechanism on the output that it has already generated.

This self-attention mechanism should mean that transformers should be able to make longer sentences that are well structured a lot more regularly than the RNN-based decoder.

Target sentence	he accused his wife of preferring others to himself and told her to return to the soviet union wi
LSTM decoder's prediction	he che huse to i f prof profering others to been to hintial he had teritunt the so he sout h at h
Tranformers decoder's prediction	he accessive his wall from the friends to himself, and to have tell her to the southeasmove in yo

Target sentence	in the marines, he evidenced a strong conviction as to the correctness of marxist doctrine,
LSTM decoder's prediction	in the morings, have add insto strong ching as to the crectnis of marcys doptron.
Tranformers decoder's prediction	in the morings, he added and in violent conviction as to the president more common sixteen prison

Table 5: Validation string predictions of both models after 15 epochs

This last statement can be confirmed in the 2 examples of table 5, where even though neither of the decoders get the prediction right, the Transformer based decoder's prediction string is more coherent and makes more sense as a sentence.

Finally, it is important to mention that Transformers make use of Multi-Head self-attention. This uses several attention blocks in parallel that can focus on different parts of the output at the same time. This should enable this type of decoder to output richer and more complex representations of the input data.

Considering the added self-attention mechanism of the decoder from Question 3.2, it is possible to understand why it's test results are better than the decoder from Question 3.1. This can also be checked considering the test set loss scores of each model. (second decoder has the best loss value).

3.4 Question 3.4

As is the case with loss values, the string similarity scores have overall better results for the Transformer based decoder than the LSTM based one.

For the same model, the 3 different string similarity scores have very different values between each other. This is because they evaluate in different ways how similar a prediction string is to it's target string. For all of them: string similarity of 1 is if strings are the same; string similarity of 0 is if strings have nothing in common.

- **Damerau-Levenshtein string Similarity** evaluates how many character operations are needed to transform one string into another. There are four possible operations: insertions, deletions, substitutions and transposition. [3]
- **Jaccard string Similarity** This is a token based algorithm, which means that it does not evaluate characters, but entire words. This similarity algorithm evaluates how many words the strings have in common and divides it by the total number of words. This means that the algorithm does not take into consideration the order of the words. [3]
- **Cosine string Similarity** This similarity algorithm quantifies the cosine of an angle between two vectors in a multi-dimensional space. In the case of natural language algorithms, this similarity score considers both strings as vectors and it is also a token-based algorithm. This algorithm does not take into account the length of the string, but it is very good at comparing the meaning and information of the strings. [3] [1]

The following paragraphs compare the test set scores for these similarity algorithms.

Their **Damerau-Levenshtein string Similarity scores** are the lowest for both models. These encoder-decoder models are being trained using tokens/words (besides a softmax activation decoder layer that used words in a vocabulary list). In contrast, this similarity algorithm only considers similarity between characters. This can be the reason that explains the lower scores given by this algorithm. In addition, it is important to mention that the Transformer based model has a higher Damerau-Levenshtein string Similarity, which means that the overall string is closer in shape and position of the characters than the LSTM based model.

Their **Jaccard string Similarity scores** are fairly good. This means that many of the words in the string are the same between the prediction and target strings. This does not guarantee that the words are positioned well. Comparing this score to the Damerau-Levenshtein string Similarity scores, it is possible to create an hypothesis that many of the words are correct but they are not necessarily positioned correctly in the sentence. This hypothesis aligned with the fact that the Damerau-Levenshtein similarity score of the LSTM model is worse than the transformer based model, indicates that: the position matrix added to the embedding in the forward pass of the Transformer model is better at translating positional information than the method used in the LSTM based model.

Their **Cosine string Similarity scores** is much higher than the others and is, for both models, a high value. This should mean that the predicted strings have overall a close meaning to the target strings.

References

[1] Arjun Prakash - Understanding Cosine Similarity: A key concept in data science, howpublished = https://medium.com/@arjunprakash027/understanding-cosine-similarity-a-key-concept-in-data-science-72a0fcc57599, note = Accessed on January, 03, 2024.

[2] Emily Rosemary Collins - Transformer vs LSTM: A Helpful Illustrated Guide, howpublished = https://blog.finxter.com/transformer-vs-lstm/, note = Accessed on January, 02, 2024.

[3] Yassine EL KHAL - The complete guide to string similarity algorithms, howpublished = https://yassineelkhal.medium.com/the-complete-guide-to-string-similarity-algorithms-1290ad07c6b7, note = Accessed on January, 03, 2024.

[4] Mário Figueiredo André Martins, Francisco Melo. *Deep Learning Course (Lecture Slides and Pratical Lessons)*. 2023-2024.