Deep Learning

MEEC

Homework 2

Authors:

Rodrigo Faria (100081) rodrigo.faria@tecnico.ulisboa.pt
Ricardo Silva (100071) ricardo.querido@tecnico.ulisboa.pt

Group 35

2023/2024 – 1º Semester, P2

The workload was distributed in the following way: both elements discussed Q1; Rodrigo wrote Q1 and Q2 answers, while Ricardo wrote Q3. This was agreed by both elements in the group.

1 Question 1

1.1 Problem 1

We can estimate the computational complexity required to compute $\mathbf{Z}$ by examining each step of its computation.

- $\mathbf{QK}^T$: The matrix multiplication between $\mathbf{QK}^T$ has a complexity of $O(L^2D)$. This arises from the operation of traversing through each of the L rows in matrix $\mathbf{Q}$ and multiplying it by each of the L columns in matrix $\mathbf{K}^T$, with each row and column containing D terms, resulting in the stated complexity.

- Softmax($\mathbf{QK}^T$): The multiplication of $\mathbf{QK}^T$ yields a $\mathbb{R}^{L\times L}$ matrix. To compute Softmax($\mathbf{QK}^T$), we must traverse each row to calculate its exponentials and their summation. Since there are L rows, and each row contains L elements, the complexity is $O(L^2)$.

- $\mathbf{Z} = \text{Softmax}(\mathbf{QK}^T)\mathbf{V}$: Finally, we perform the multiplication of Softmax($\mathbf{QK}^T$) $\in \mathbb{R}^{L\times L}$ with $\mathbf{V} \in \mathbb{R}^{L\times D}$. Similar to the computation of $\mathbf{QK}^T$, this step also has a complexity of $O(L^2D)$.

The total computational complexity of computing $\mathbf{Z}$ can be summarized as follows:

$$O(L^2D) + O(L^2) + O(L^2D) \equiv O(L^2D) \tag{1}$$

It's worth noting that the self-attention layer in a transformer that employs this method exhibits quadratic complexity concerning the parameter L. This quadratic complexity can pose challenges for processing long sequences as L represents the number of tokens in each input sequence (and D represents the embedding dimensionality of each token), making the training of long sequences practically infeasible, due to the enormous computational cost.

1.2 Problem 2

Using the McLaurin series expansion for the exponential function, we can make an approximation for the exponential function by only using the first K terms of the expansion.

We can obtain an expression for the feature map $\phi : \mathbb{R}^D \to \mathbb{R}^M$, so that its dot product gives an exponential approximation for the exponential function. Since the computation of terms, $K > 2$ involves expanding the K-th power of a sum of D terms, we can apply the multinomial theorem. The formula for the expansion is:

$$(x_1 + x_2 + \ldots + x_d)^k = \sum_{n_1+n_2+\ldots+n_d=k} \frac{k!}{n_1!n_2!\ldots n_d!} x_1^{n_1} x_2^{n_2} \ldots x_d^{n_d} \tag{2}$$

Since we are considering only the first three terms of the McLaurin expansion, we can derive the following expression for the feature map:

$$\phi(\mathbf{t}) = \begin{bmatrix} 1 & t_1 & \cdots & t_D & \frac{t_1^2}{\sqrt{2}} & \cdots & \frac{t_D^2}{\sqrt{2}} & t_1t_2 & \cdots & t_1t_D & t_2t_3 & \cdots & t_{D-1}t_D \end{bmatrix}^T \tag{3}$$

The obtained expression for ϕ has a dimensionality that is given by the following expression:

$$M = 1 + D + \sum_{n=0}^{D}(D-n) = 1 + 2D + \frac{D(D-1)}{2} \tag{4}$$

We can show that for arbitrary K terms the dimensionality of the feature map expands as the following expression:

$$M = \sum_{n=0}^{K}\binom{n+D-1}{n} = \sum_{n=0}^{K}\frac{(n+D-1)!}{(D-1)!} \tag{5}$$

1.3 Problem 3

We can show that the self-attention operation can be approximated as $\mathbf{Z} \approx \mathbf{D}^{-1}\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{V}$ by showing that the Softmax$(\mathbf{QK}^T)$ operation can be approximated by $\mathbf{D}^{-1}\Phi(\mathbf{Q})\Phi(\mathbf{K})^T$, where $\mathbf{D} = \mathbf{Diag}(\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{1}_L)$.

By employing the exponential approximation derived in the preceding problem, we can obtain the following approximation for the softmax function:

$$\text{Softmax}(\mathbf{QK}^T) = \begin{bmatrix} \frac{\exp(\mathbf{q}_1^T\mathbf{k}_1)}{\sum_{i=1}^L \exp(\mathbf{q}_1^T\mathbf{k}_i)} & \frac{\exp(\mathbf{q}_1^T\mathbf{k}_2)}{\sum_{i=1}^L \exp(\mathbf{q}_1^T\mathbf{k}_i)} & \cdots & \frac{\exp(\mathbf{q}_1^T\mathbf{k}_L)}{\sum_{i=1}^L \exp(\mathbf{q}_1^T\mathbf{k}_i)} \\ \frac{\exp(\mathbf{q}_2^T\mathbf{k}_1)}{\sum_{i=1}^L \exp(\mathbf{q}_2^T\mathbf{k}_i)} & \frac{\exp(\mathbf{q}_2^T\mathbf{k}_2)}{\sum_{i=1}^L \exp(\mathbf{q}_2^T\mathbf{k}_i)} & \cdots & \frac{\exp(\mathbf{q}_2^T\mathbf{k}_L)}{\sum_{i=1}^L \exp(\mathbf{q}_2^T\mathbf{k}_i)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\exp(\mathbf{q}_L^T\mathbf{k}_1)}{\sum_{i=1}^L \exp(\mathbf{q}_L^T\mathbf{k}_i)} & \frac{\exp(\mathbf{q}_L^T\mathbf{k}_2)}{\sum_{i=1}^L \exp(\mathbf{q}_L^T\mathbf{k}_i)} & \cdots & \frac{\exp(\mathbf{q}_L^T\mathbf{k}_L)}{\sum_{i=1}^L \exp(\mathbf{q}_L^T\mathbf{k}_i)} \end{bmatrix} \tag{6}$$

$$\approx \begin{bmatrix} \frac{\phi(\mathbf{q}_1)^T\phi(\mathbf{k}_1)}{\sum_{i=1}^L \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_i)} & \frac{\phi(\mathbf{q}_1)^T\phi(\mathbf{k}_2)}{\sum_{i=1}^L \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_i)} & \cdots & \frac{\phi(\mathbf{q}_1)^T\phi(\mathbf{k}_L)}{\sum_{i=1}^L \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_i)} \\ \frac{\phi(\mathbf{q}_2)^T\phi(\mathbf{k}_1)}{\sum_{i=1}^L \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_i)} & \frac{\phi(\mathbf{q}_2)^T\phi(\mathbf{k}_2)}{\sum_{i=1}^L \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_i)} & \cdots & \frac{\phi(\mathbf{q}_2)^T\phi(\mathbf{k}_L)}{\sum_{i=1}^L \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_i)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\phi(\mathbf{q}_L)^T\phi(\mathbf{k}_1)}{\sum_{i=1}^L \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_i)} & \frac{\phi(\mathbf{q}_L)^T\phi(\mathbf{k}_2)}{\sum_{i=1}^L \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_i)} & \cdots & \frac{\phi(\mathbf{q}_L)^T\phi(\mathbf{k}_L)}{\sum_{i=1}^L \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_i)} \end{bmatrix} \tag{7}$$

We can further break down the resulting matrix into the following matrix multiplication:

$$\begin{bmatrix} \frac{1}{\sum_{i=1}^L \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_i)} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sum_{i=1}^L \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_i)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sum_{i=1}^L \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_i)} \end{bmatrix} \begin{bmatrix} \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_1) & \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_2) & \cdots & \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_L) \\ \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_1) & \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_2) & \cdots & \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_L) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_1) & \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_2) & \cdots & \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_L) \end{bmatrix} \tag{8}$$

Since we have a diagonal matrix we can express the fractions of the elements in its diagonal with the inverse of the matrix, yielding:

$$\begin{bmatrix} \sum_{i=1}^{L}\phi(\mathbf{q}_1)^T\phi(\mathbf{k}_i) & 0 & \cdots & 0 \\ 0 & \sum_{i=1}^{L}\phi(\mathbf{q}_2)^T\phi(\mathbf{k}_i) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sum_{i=1}^{L}\phi(\mathbf{q}_L)^T\phi(\mathbf{k}_i) \end{bmatrix}^{-1} \begin{bmatrix} \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_1) & \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_2) & \cdots & \phi(\mathbf{q}_1)^T\phi(\mathbf{k}_L) \\ \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_1) & \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_2) & \cdots & \phi(\mathbf{q}_2)^T\phi(\mathbf{k}_L) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_1) & \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_2) & \cdots & \phi(\mathbf{q}_L)^T\phi(\mathbf{k}_L) \end{bmatrix} \tag{9}$$

The matrix on the right is the result of the multiplication $\Phi(\mathbf{Q})\Phi(\mathbf{K})^T$. The matrix on the left can be obtained through the diagonalization of the vector whose elements contain the summations. This vector can be easily derived as follows:

$$\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{1}_L = \begin{bmatrix} \sum_{i=1}^{L}\phi(\mathbf{q}_1)^T\phi(\mathbf{k}_i) \\ \sum_{i=1}^{L}\phi(\mathbf{q}_2)^T\phi(\mathbf{k}_i) \\ \vdots \\ \sum_{i=1}^{L}\phi(\mathbf{q}_L)^T\phi(\mathbf{k}_i) \end{bmatrix} \tag{10}$$

By combining all the terms, we obtain the following approximation for the softmax function:

$$\text{Softmax}(\mathbf{QK}^T) \approx \mathbf{Diag}(\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{1}_L)^{-1}\Phi(\mathbf{Q})\Phi(\mathbf{K})^T \tag{11}$$

Substituting this approximation into the expression for $\mathbf{Z}$ yields the desired result.

1.4 Problem 4

We can estimate the complexity of the previous approximation of $\mathbf{Z}$, keeping into mind that $\Phi(\mathbf{Q}), \Phi(\mathbf{K}) \in \mathbb{R}^{L\times M}$, $\mathbf{V} \in \mathbb{R}^{L\times D}$ and $\mathbf{1}_L \in \mathbb{R}^L$.

- $\Phi(\mathbf{K})^T\mathbf{V}$: The complexity of this multiplication is $O(LMD)$, which yields from the fact that we need to transverse the M rows of the matrix $\Phi(\mathbf{K})^T$ and apply a dot product between the row and each of the D columns of the matrix $\mathbf{V}$. Since each dot product involves L multiplications and $L-1$ additions, we get the stated complexity.

- $\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{V}$: The complexity of this multiplication is $O(LMD)$. Similarly to the previous, we have the multiplication between $\Phi(\mathbf{Q})$ and the matrix that wields from $\Phi(\mathbf{K})^T\mathbf{V} \in \mathbb{R}^{M\times D}$.

- $\Phi(\mathbf{K})^T\mathbf{1}_L$: The complexity of this multiplication is $O(LM)$. Since we have a vector we only need to compute the dot product between each row of the matrix $\mathbf{K})^T$ and the one column of the vector.

- $\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{1}_L$: The complexity of this multiplication is $O(LM)$.

- $\mathbf{D} = \mathbf{Diag}(\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{1}_L)$: The complexity of the diagonalization of the vector is $O(L)$. Since the matrix is diagonal all its entries except the diagonals are null, so we only need to transverse the vector and assign each entry to the corresponding diagonal.

- $\mathbf{D}^{-1}$: The complexity of computing the inverse of the diagonal matrix is $O(L)$. Since the matrix is diagonal, we only need to compute the inverse of each of the non-zero diagonal entries.
- $\mathbf{Z} \approx \mathbf{D}^{-1}\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{V}$: The complexity of this multiplication is $O(LD)$. In this multiplication, since the diagonal matrix only has non-zero elements along its diagonal, the multiplication between the diagonal matrix and the other matrix is element-wise, and it involves multiplying each element in the $\Phi(\mathbf{Q})\Phi(\mathbf{K})^T\mathbf{V} \in \mathbb{R}^{L\times D}$ matrix by the corresponding diagonal element.

The total computational complexity of computing this approximation of $\mathbf{Z}$ can be summarized as follows:

$$O(LMD) + O(LMD) + O(LM) + O(LM) + O(L) + O(LD) \equiv O(LMD) \tag{12}$$

This approximation allows us to transition from a quadratic dependency on L to a linear one, significantly enhancing the transformer's efficiency when processing long-sequence inputs. As a result, the computational load scales linearly with the input length, which is particularly advantageous for handling extended sequences.

In scenarios where the input size is smaller, with $L < D$, it may still be preferable to employ the standard computation method.

2 Question 2

2.1 Problem 1

The following graphs and table show the training loss as well as the accuracies of the specified convolutional neural network (CNN) specified in the problem.

η	Train loss	Val Acc	Test Acc
0.1	0.6361	0.8095	0.7845
0.01	0.5288	0.8804	0.8393
0.001	0.9584	0.7786	0.7977

Table 1: Learning rate influence in performance in CNN with max-pooling

The optimal model is identified as the one utilizing a learning rate of $\eta = 0.01$. This model strikes a balance between convergence speed and stability, achieving a lower minimum within the specified number of epochs without overshooting.

In contrast, the model with $\eta = 0.1$ exhibits an overshooting behavior when updating weights. This leads to divergent tendencies, as illustrated in Figure 1, where a gradual decrease in accuracy over time is observable.

Conversely, the model employing $\eta = 0.001$ demonstrates a steady increase in accuracy across epochs. However, due to its smaller step size, it fails to attain an optimal minimum within the set epochs, highlighting the need for a balance between step size and convergence efficiency.

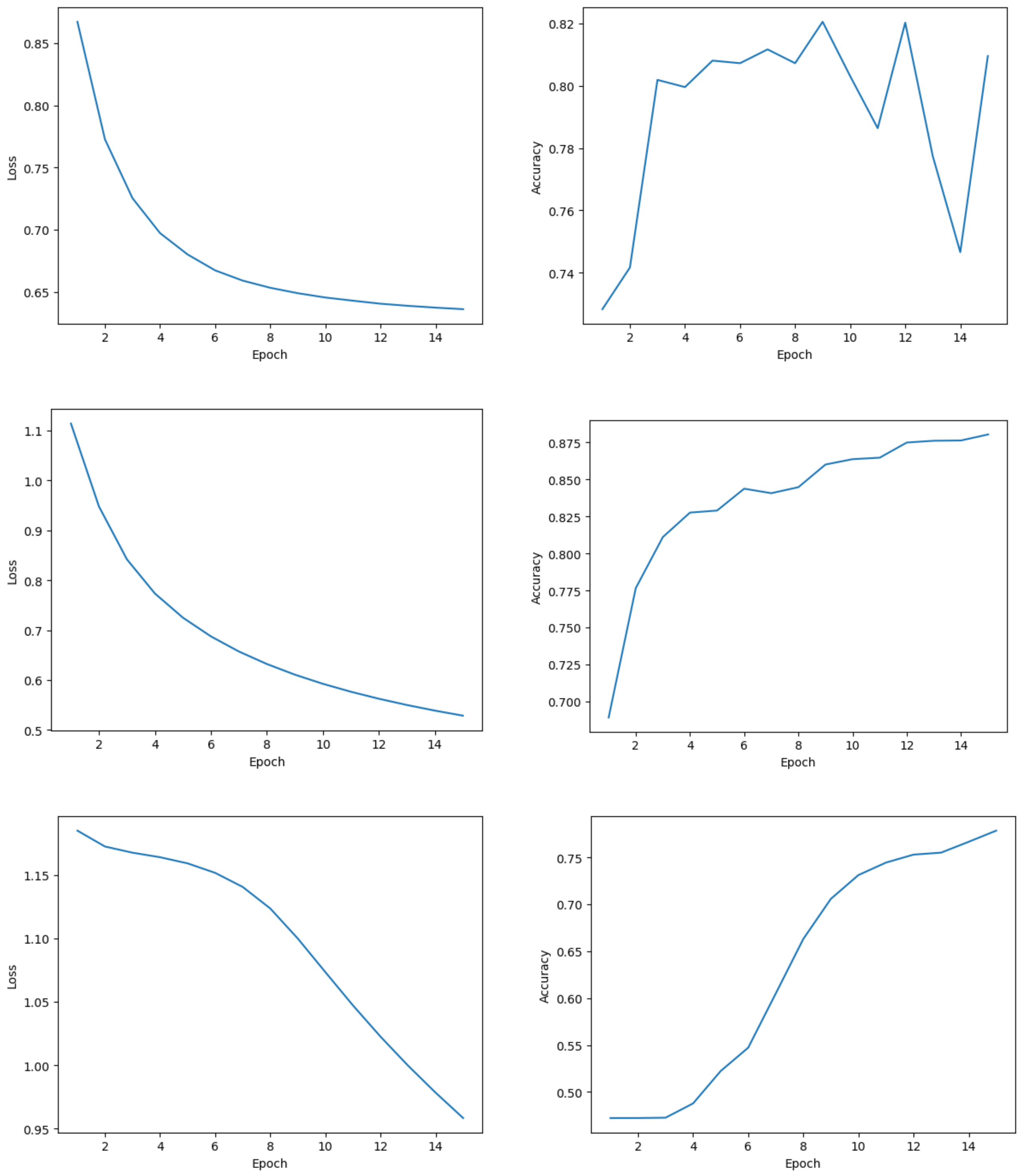

Figure 1: Training Loss and Validation Accuracy with $\eta = \{0.1, 0.01, 0.001\}$, respectively, for CNN with max-pooling

2.2 Problem 2

The following graphs 2 and table show the performance of the specified CNN using the same hyper-parameters as the one specified in the previous problem.

η	Train loss	Val Acc	Test Acc
0.1	0.6652	0.7881	0.7807
0.01	0.6501	0.8500	0.8204
0.001	1.0541	0.6889	0.7108

Table 2: Learning rate influence in performance in CNN without max-pooling

Similarly to the preceding case, the learning rate of $\eta = 0.01$ again emerges as the most effective, mirroring the issues observed with the other learning rates in the prior model. This rate optimally balances the convergence speed and stability, avoiding the pitfalls seen with the other alternative rates.

2.3 Problem 3

The number of trainable parameters is the same for each model, that is 224892. This can be shown analytically as follows.

For the CNN that makes use of max-pooling, we have the following number of trainable parameters for each layer:

- First convolution layer with input dimensions of 28×28×1 and output dimensions of 28×28×8: n$^{\text{o}}$ train. params. $= 8 \times (3 \times 3 \times 1 + 1) = 80$
- First max-pooling layer with input dimensions of 28×28×8 and output dimensions of 14×14×8: n$^{\text{o}}$ train. params. $= 0$
- Second convolution layer with input dimensions of 14×14×8 and output dimensions of 12×12×16: n$^{\text{o}}$ train. params. $= 16 \times (3 \times 3 \times 8 + 1) = 1168$
- Second max-pooling layer with input dimensions of 12×12×16 and output dimensions of 6×6×16: n$^{\text{o}}$ train. params. $= 0$
- First fully connected layer with input size of $6 \times 6 \times 16 = 576$ and output size of 320: n$^{\text{o}}$ train. params. $= 320 \times (576 + 1) = 184640$
- Second fully connected layer with input size of 320 and output size of 120: n$^{\text{o}}$ train. params. $= 120 \times (320 + 1) = 38520$
- Third fully connected layer with input size of 120 and output size of 4: n$^{\text{o}}$ train. params. $= 4 \times (120 + 1) = 484$

Summing all the trainable parameters we obtain the result previously stated. For the CNN that uses a stride of 2 instead of max-pooling, we have the following number of parameters (taking into account that the fully connected layers will have the same architecture in both models):

- First convolution layer with input dimensions of $28\times28\times1$ and output dimensions of $14\times14\times8$: n$^{\text{o}}$ train. params. $= 8 \times (3 \times 3 \times 1 + 1) = 80$
- Second convolution layer with input dimensions of $14\times14\times8$ and output dimensions of $6\times6\times16$: n$^{\text{o}}$ train. params. $= 16 \times (3 \times 3 \times 8 + 1) = 1168$

The difference in performance between both networks in this case may be justifiable by the differences in computational efficiency between them. Specifically, the CNN employing a stride of 2, instead of max-pooling, demonstrates a notable speed increase, averaging 1.5 times faster run times compared to its counterpart. This efficiency gain is due to the CNN's ability to simultaneously execute feature extraction and input downsampling within a single convolution layer. In contrast, the alternative network requires an additional max-pooling layer for downsampling. Furthermore, the CNN with a stride of 2 exhibits improved memory usage. This is because the other model necessitates storing a larger feature map during the intermediate step between convolution and max-pooling, leading to higher memory demands. However, by using this type of convolution layer we sacrifice the max-pooling robustness to noise and translation of the input, which enables us to extract more abstract features.

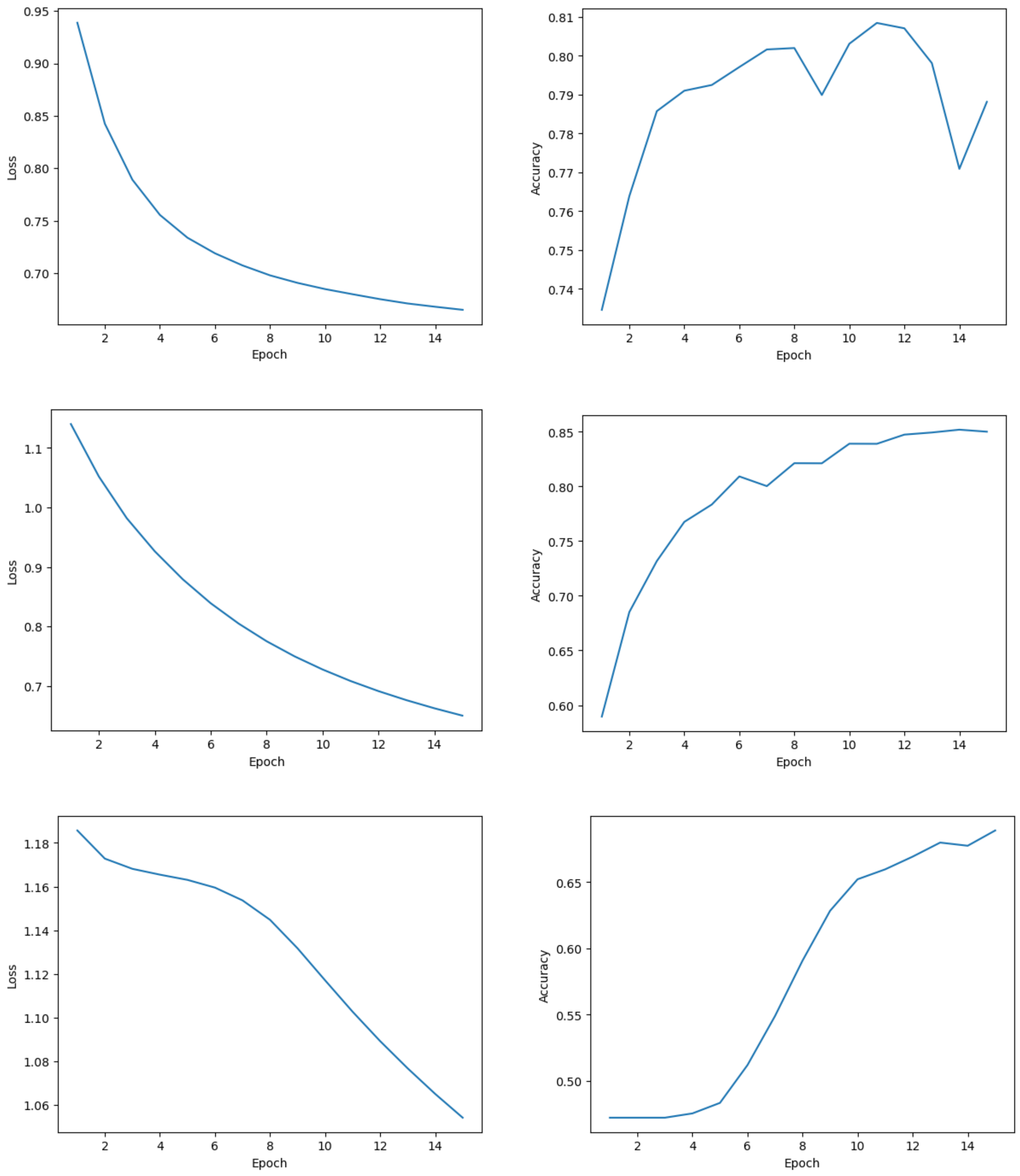

Figure 2: Training Loss and Validation Accuracy with $\eta = \{0.1, 0.01, 0.001\}$, respectively, for CNN without max-pooling

3 Question 3

3.1 Problem 1

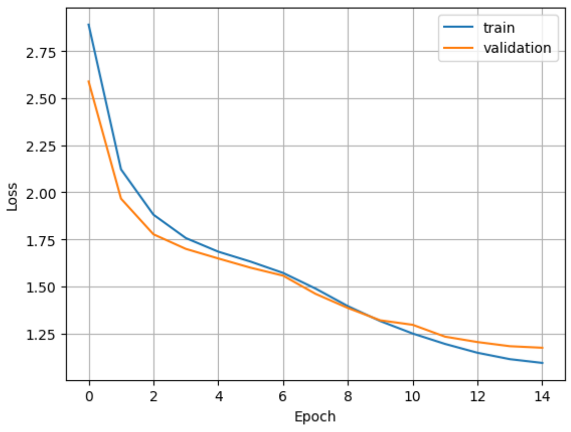

Figure 3: Training and Validation Loss Plots

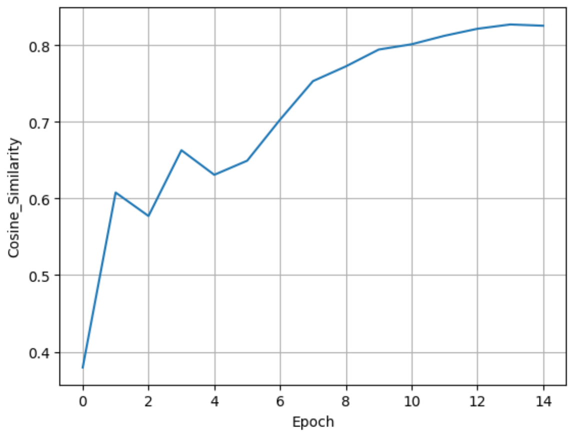

Figure 4: Cosine Similarity Plot

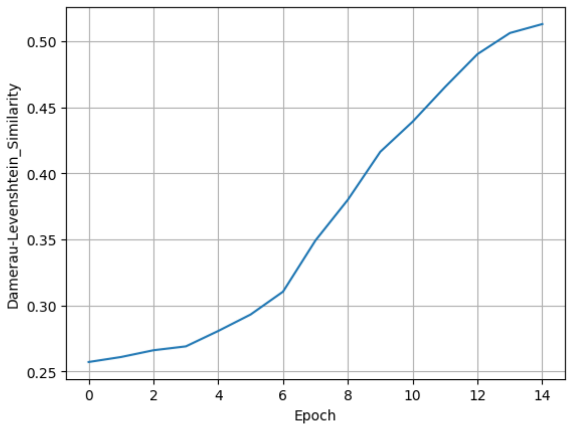

Figure 5: Damerau-Levenshtein Similarity Plot

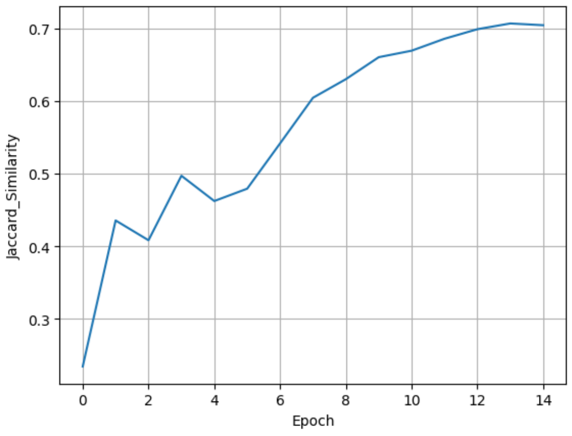

Figure 6: Jaccard Similarity Plot

Loss	Cosine Similarity	Damerau-Levenshtein Similarity	Jaccard Similarity
1.1828	0.8324	0.5087	0.7149

Table 3: Test Metrics

3.2 Problem 2

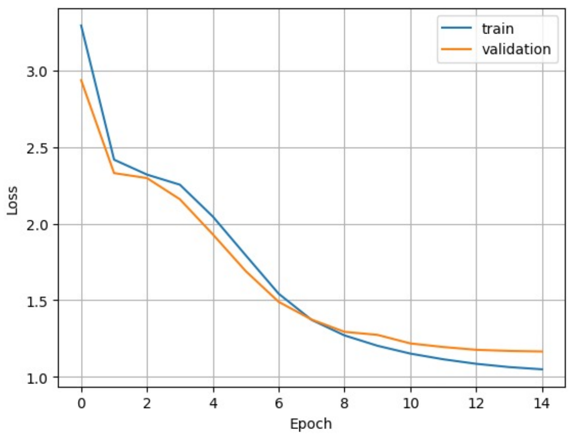

Figure 7: Training and Validation Loss Plots

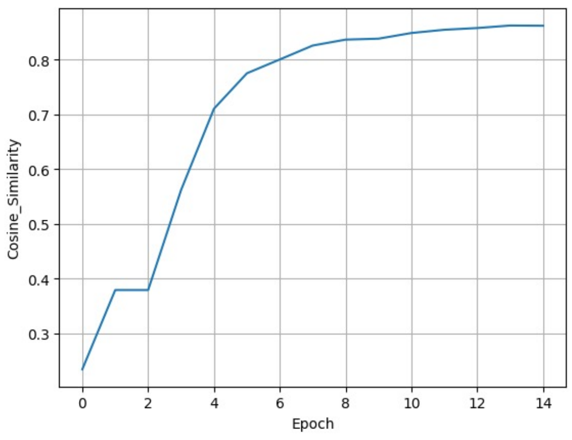

Figure 8: Cosine Similarity Plot

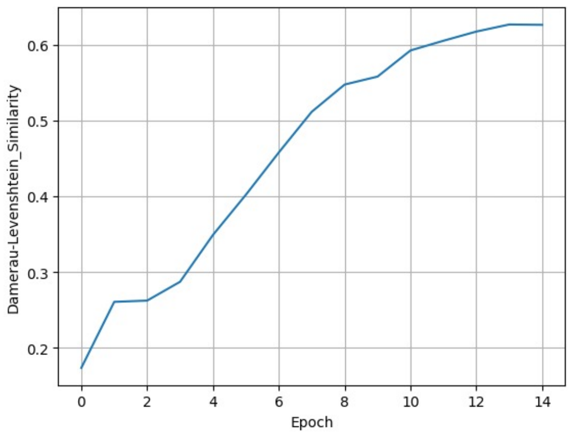

Figure 9: Damerau-Levenshtein Similarity Plot

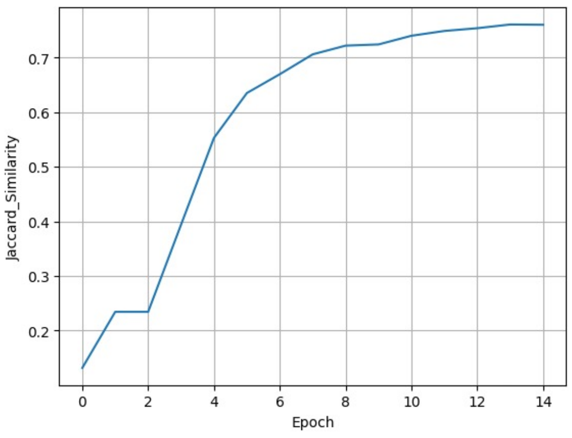

Figure 10: Jaccard Similarity Plot

Loss	Cosine Similarity	Damerau-Levenshtein Similarity	Jaccard Similarity
1.1606	0.8653	0.6330	0.7646

Table 4: Test Metrics

3.3 Problem 3

When contrasting the two architectures, differences in configurations emerge, ultimately resulting in variations in model performance. In the RNN-based model, input is exclusively embedded concerning tokens, and the subsequent sub-block employs a LSTM layer. This layer processes the embedded input sequentially, capturing long-term dependencies in the process, and outputs the hidden states **h**.

$$\begin{cases} h_t = o_t \odot g(c_t) \\ c_t = f_t \odot c_{t-1} + i_t \odot g(Vx_t + Uh_{t-1} + b) \\ f_t = \sigma(\boldsymbol{V_f x_t} + \boldsymbol{U_f h_{t-1}} + \boldsymbol{b_f}) \\ i_t = \sigma(\boldsymbol{V_i x_t} + \boldsymbol{U_i h_{t-1}} + \boldsymbol{b_i}) \\ o_t = \sigma(\boldsymbol{V_o x_t} + \boldsymbol{U_o h_{t-1}} + \boldsymbol{b_o}) \end{cases} \tag{13}$$

In contrast, the Attention-based model incorporates both positional embedding and token embedding, leveraging a Multi-Head Attention Layer within the sub-block. This layer processes the embedded input sequence in parallel, which incorporates sequential order information, transforms it into matrices **Q**, **K**, and **V**, and generates output **Z**. The attention mechanism allows the decoder to focus on distinct segments of the input, and the incorporation of a mask feature ensures that each word can only interact with its predecessors.

$$\begin{cases} \boldsymbol{Z} = softmax(\frac{\boldsymbol{QK}^T}{\sqrt{d_k}})\boldsymbol{V} \\ \boldsymbol{Q} = \boldsymbol{XW}^Q \\ \boldsymbol{K} = \boldsymbol{XW}^K \\ \boldsymbol{V} = \boldsymbol{XW}^V \end{cases} \tag{14}$$

Usually, attention-based models outperform RNN-based models since attention mechanisms allow the model to attend to different parts of the input sequence simultaneously, making them more effective in capturing global dependencies and can also help mitigate the vanishing problem.

Even though the LSTM is an RNN-based model that avoids some of the effects of the vanishing problem by relying on additive rather than multiplicative/highly non-linear recurrent dynamics, it still gets outperformed in models such as the transformer in most situations. In our case, when we compare test results we confirm that the Transformer performed better than the LSTM since it outputs smaller loss and higher similarities across the board.

3.4 Problem 4

Firstly let's try to understand each of the similarity algorithms. The Damerau-Levenshtein Similarity compares two strings and counts the minimum number of operations (insertions, deletions, substitutions and transpositions) required to transform one string into the other. It can be defined by the formula 3.4 and its range is [0,1]. The Jaccard Similarity takes two sets and compares the intersection and union of their elements. It can be calculated as described in equation 3.4 and its range is [0,1]. The Cosine Similarity computes the cosine of the angle between two vectors. It can be calculated by equation 3.4 and its range is [-1,1], where 1 indicates identical vectors, 0 denotes orthogonality and -1 signifies opposed vectors. In all three algorithms, the closer the value is to 1, the higher the similarity.

$$damlev\_similarity(s1, s2) = 1 - \frac{damlev\_distance(s1, s2)}{max(length(s1), length(s2))} \tag{15}$$

$$jaccard\_similarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{16}$$

$$cosine\_similarity(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||} \tag{17}$$

By analyzing Table 3.4 we can see that the Attention-based model outperformed the RNN-based model across all metrics. The most notable discrepancy lies in the Damerau-Levenshtein similarity. This implies that sentences generated by the attention-based model generally require significantly fewer operations for transformation into the target. The cosine metric underscores that the attention-based model's output vector exhibits a greater similarity in direction compared to the RNN-based model. Furthermore, the Jaccard metric indicates a more substantial overlap between the sets of elements in the attention-based output and the target, surpassing the performance of the other model.

	Cosine Similarity	Damerau-Levenshtein Similarity	Jaccard Similarity
RNN-based model	0.8324	0.5087	0.7149
Attention-based model	0.8653	0.6330	0.7646

Table 5: Similarity Metrics