Instituto Superior Técnico

Deep Learning 2023

Homework 2 - Development and results

Report

Name	**IST Number**
Davi Giordano Valério	108497
Vicente Lorenzo	92569

Lisbon
2023

INSTITUTO SUPERIOR TÉCNICO

Homework 2 - Development and results

Report

Report of homework 2 to prof. André Martins at the Deep Learning course, at Instituto Superior Técnico, in Lisbon, Portugal.

Lisbon, Portugal
2023

Contents

List of Figures

0. Introduction

This document reports the results of homework 2 from the Deep Learning course at Instituto Superior Técnico. The members organized the tasks in the project according to Table 1. The difficulty of each task was estimated so as to attribute an equal ammount of complexity to each member of the group (26 points).

In this project, python 3.11 was used.

Item	Who was responsible	Estimated complexity
Solve Question 1	Vicente Lorenzo	5
Report Question 1	Vicente Lorenzo	3
Solve Question 2	Davi Giordano	3
Report Question 2	Davi Giordano	1
Solve Question 3	Davi Giordano	5
Report Question 3	Vicente Lorenzo	1

Table 1: Division of tasks in homework 2

1. Question 1

1.1 Item 1.1

To understand the computational complexity of computing Z in the self-attention layer of a transformer with a single attention head, we need to analyze the computation $Z = \text{Softmax}(QK^T)V$, where Q, K, and V are matrices of size $L \times D$ (sequence length L and hidden size D):

1. **Computing QK^T**:
   - The matrix multiplication of Q and K^T involves matrices of size $L \times D$ and $D \times L$, respectively.
   - The complexity of matrix multiplication is $O(n^3)$ for square matrices of size $n \times n$.
   - In this case, the operation is not exactly square matrix multiplication but is more akin to $O(L \times D \times L)$.
2. **Applying Softmax**:
   - The Softmax function is applied to each row of the QK^T matrix.
   - The complexity of Softmax is linear with respect to the number of elements in the row, so this operation has a complexity of $O(L^2)$.
3. **Multiplying by V**:
   - After Softmax, the resultant matrix (of size $L \times L$) is multiplied by V (of size $L \times D$).
   - This multiplication again has a complexity of $O(L^2 \times D)$.

Combining these, the overall computational complexity of computing Z in the self-attention layer is $O(L^2 \times D)$. The dominant factor here is L^2 as it appears in both major steps of the computation.

This quadratic dependency on the sequence length L can be problematic for long sequences because:

1. **Memory Usage**: The memory required to store the attention scores scales quadratically with the sequence length, which can become infeasible for very long sequences.

2. **Computational Resources**: The time to compute the attention scores and the subsequent multiplication increases quadratically, making it computationally expensive for longer sequences. This can be a bottleneck in terms of both processing time and energy consumption.

3. **Parallelization Limits**: While some parts of the transformer model can be parallelized, the quadratic nature of this computation can limit the extent to which parallelization can speed up the process, especially for long sequences.

In summary, the $O(L^2 \times D)$ complexity of self-attention is a major factor limiting the scalability of transformer models to very long sequences.

1.2 Item 1.2

To find a feature map $\varphi : \mathbb{R}^D \to \mathbb{R}^M$ such that $\exp(q^T k) \approx \varphi(q)^T \varphi(k)$ using the first three terms of the McLaurin series expansion for $\exp(t)$, we start by approximating $\exp(t)$ as $1 + t + \frac{t^2}{2}$. Here, t is a scalar, and in the context of the self-attention mechanism, t can be represented by the dot product $q^T k$.

1. **Approximation using McLaurin Series**:

$$\exp(q^T k) \approx 1 + q^T k + \frac{(q^T k)^2}{2}$$

2. **Finding the Feature Map φ**:

   To express $\exp(q^T k)$ in the form $\varphi(q)^T \varphi(k)$, we need to construct $\varphi(q)$ and $\varphi(k)$ such that their dot product gives the right-hand side of the approximation.

   We can construct $\varphi(q)$ and $\varphi(k)$ as follows:

$$\varphi(q) = \begin{bmatrix} 1 \\ \sqrt{1}q \\ \sqrt{\frac{1}{2}}q^2 \end{bmatrix}, \quad \varphi(k) = \begin{bmatrix} 1 \\ \sqrt{1}k \\ \sqrt{\frac{1}{2}}k^2 \end{bmatrix}$$

Here, q^2 and k^2 represent element-wise squares of the vectors q and k.

3. **Dimensionality of the Feature Space** M:

   Since each q and k are in $\mathbb{R}^D$, their squared versions will also be in $\mathbb{R}^D$. Therefore, the dimensionality M of the feature space for each $\varphi(q)$ or $\varphi(k)$ is given by:

   $$M = 1 + D + D = 2D + 1$$

   This comes from one dimension for the scalar 1, D dimensions for the vector q or k, and D dimensions for the element-wise square of q or k.

4. **Dimensionality for** $K \geq 3$ **Terms**:

   If we use K terms in the McLaurin series expansion, the feature map will include terms up to t^{K-1}. Each term t^n corresponds to the n-th power of the vector (element-wise), adding D dimensions for each power. Thus, the total dimensionality M as a function of D and K is:

   $$M = 1 + D + D + D + \cdots + D \quad \text{(K terms)}$$

   $$M = 1 + D(K - 1)$$

   This accounts for one scalar term and $K - 1$ vector terms, each contributing D dimensions.

In summary, using a McLaurin series approximation for the exponential function in the context of self-attention allows for a reduction in computational complexity by transforming the operation into a dot product of transformed feature vectors, with the dimensionality of these vectors depending on the number of terms used in the series expansion.

1.3 Item 1.3

To show how the self-attention operation can be approximated using the given approximation of the exponential function and the feature maps $\Phi(Q)$ and $\Phi(K)$, let's break down the steps:

1. **Self-Attention Operation**:

   The standard self-attention operation in a transformer is defined as $Z = \text{Softmax}(QK^T)V$. Here, Q, K, and V are matrices representing queries, keys, and values, respectively.

2. **Approximation of Exponential Function**:

   Using the approximation $\exp(q^T k) \approx \varphi(q)^T \varphi(k)$, we can approximate the softmax operation, which involves the exponential of dot products, with the dot products of feature maps.

3. **Feature Maps $\Phi(Q)$ and $\Phi(K)$**:

   $\Phi(Q)$ and $\Phi(K)$ are matrices whose rows are $\varphi(q_i)$ and $\varphi(k_i)$, respectively, for the i-th rows of the original matrices Q and K. Thus, $\Phi(Q) \in \mathbb{R}^{L \times M}$ and $\Phi(K) \in \mathbb{R}^{L \times M}$.

4. **Approximating the Self-Attention Operation**:

   The matrix product $\Phi(Q)\Phi(K)^T$ approximates QK^T, with each element $[\Phi(Q)\Phi(K)^T]_{ij}$ approximating the dot product $q_i^T k_j$, and thus, the exponential in the softmax. However, the softmax operation also involves normalization. To approximate this, we introduce the diagonal matrix D. $D = \text{Diag}(\Phi(Q)\Phi(K)^T \mathbf{1}_L)$ is a diagonal matrix where each diagonal element is the sum of the corresponding row in $\Phi(Q)\Phi(K)^T$, effectively representing the sum of exponentials in the softmax denominator. The matrix D^{-1} is used to normalize each row of $\Phi(Q)\Phi(K)^T$.

5. **Final Approximated Self-Attention Operation**:

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

   Here, $D^{-1}\Phi(Q)\Phi(K)^T$ approximates the softmax of QK^T, and the multiplication with V completes the self-attention mechanism.

In summary, this approximation transforms the computationally intensive parts of the self-attention mechanism into operations involving dot products of lower-dimensional feature maps and a normalization step. This approach can significantly reduce computational complexity, especially beneficial for long sequences where traditional self-attention becomes impractical due to its quadratic complexity with respect to sequence length.

1.4 Item 1.4

To show how the above approximation leads to a computational complexity that is linear in L (the sequence length), and to understand how this complexity depends on M (the dimensionality of the feature space) and D (the hidden size), let's analyze each part of the approximated self-attention operation: $Z \approx D^{-1}\Phi(Q)\Phi(K)^T V$.

1. **Computing $\Phi(Q)$ and $\Phi(K)$:**

   Each row of Q and K (each of dimension D) is transformed into a row of $\Phi(Q)$ and $\Phi(K)$ (each of dimension M). This transformation is linear in D since it involves operations like addition, multiplication, and potentially squaring each element (for the first three terms of the McLaurin series). The computation for each row is independent and can be done in $O(DM)$ time. Since there are L rows, the total complexity for computing $\Phi(Q)$ and $\Phi(K)$ is $O(LDM)$.

2. **Computing $\Phi(Q)\Phi(K)^T$:**

   This is a matrix multiplication of two $L \times M$ matrices. The complexity of multiplying two $L \times M$ matrices is $O(L^2 M)$.

3. **Computing D and D^{-1}:**

   D is a diagonal matrix with each diagonal element representing the sum of the elements in the corresponding row of $\Phi(Q)\Phi(K)^T$. Computing D requires summing up L elements for each of the L rows, which is $O(L^2)$. Inverting D, a diagonal matrix, is linear in the number of diagonal elements, so this is also $O(L)$.

4. **Multiplying $D^{-1}\Phi(Q)\Phi(K)^T$ with V:**

   The result of $D^{-1}\Phi(Q)\Phi(K)^T$ is an $L \times L$ matrix, and V is an $L \times D$ matrix. The complexity of this multiplication is $O(L^2 D)$.

5. **Total Computational Complexity**:

   The dominant terms in the computation are $O(LDM)$ for computing $\Phi(Q)$ and $\Phi(K)$, and $O(L^2 M) + O(L^2 D)$ for the matrix multiplications and normalization. Assuming M is chosen to be much smaller than L (which is the case for practical

purposes to reduce complexity), the $O(L^2M)$ term becomes smaller than $O(L^2D)$. Therefore, the overall complexity is dominated by $O(LDM) + O(L^2D)$.

6. **Linear Dependence on L**:

   While the complexity is quadratic in terms of L due to the $O(L^2D)$ term, it is important to note that this is a significant improvement over the original $O(L^2D)$ complexity of standard self-attention. The $O(LDM)$ term shows a linear dependence on L for the feature map computation, which is a key improvement for handling longer sequences.

In conclusion, the approximation leads to a computational complexity that is linear in L for the part involving the computation of the feature maps, with a remaining quadratic term due to matrix multiplication. The total complexity also depends on M and D, where choosing a smaller M can help reduce the overall complexity. The key improvement here is in handling longer sequences more efficiently compared to the standard self-attention mechanism.

2. Question 2

In this question, a convolutional neural network (CNN) was implemented using Pytorch to perform classification using the OCTMNIST dataset. The data is available at https://medmnist.com/ with the CC BY 4.0 licence.

The dataset contains 109,309 valid optical coherence tomography (OCT) images for retinal diseases and presents a multi-class classification problem.

2.1 Item 2.1

For this item, the proposed architecture for the convolutional network is:

- A convolution layer with 8 output channels, a kernel of size $3x3$, stride of 1, and padding of 1.
- A rectified linear unit activation function.
- A max pooling with kernel size 2×2 and stride of 2.
- A convolution layer with 16 output channels, a kernel of size 3×3, stride of 1 , and padding of zero.
- A rectified linear unit activation function.
- A max pooling with kernel size 2×2 and stride of 2.
- An affine transformation with 320 output features.
- A rectified linear unit activation function.
- A dropout layer with a dropout probability of 0.7.
- An affine transformation with 120 output features.
- A rectified linear unit activation function.
- An affine transformation with the number of classes followed by an output LogSoftmax layer.

And its implementation can be seen below. Specifically, the desired network is obtained setting `no_maxpool` to False.

```
class CNN(nn.Module):

    def __init__(self, dropout_prob, no_maxpool=False):
        super(CNN, self).__init__()
        self.no_maxpool = no_maxpool

        if not no_maxpool:
            # Q2.1 Configuration
            self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1, padding=1)
            self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=1, padding=0)
            fc1_input_size = 16 * 6 * 6
        else:
            # Q2.2 Configuration
            self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=2, padding=1)
            self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2, padding=0)
            fc1_input_size = 16 * 6 * 6

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(fc1_input_size, 320)
        self.drop = nn.Dropout(p=dropout_prob)
        self.fc2 = nn.Linear(320, 120)
        self.fc3 = nn.Linear(120, 10)

    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        x = F.relu(self.conv1(x))
        # Max-pooling 1
        if not self.no_maxpool:
            x = self.pool(x)

        x = F.relu(self.conv2(x))
        # Max-pooling 2
        if not self.no_maxpool:
```

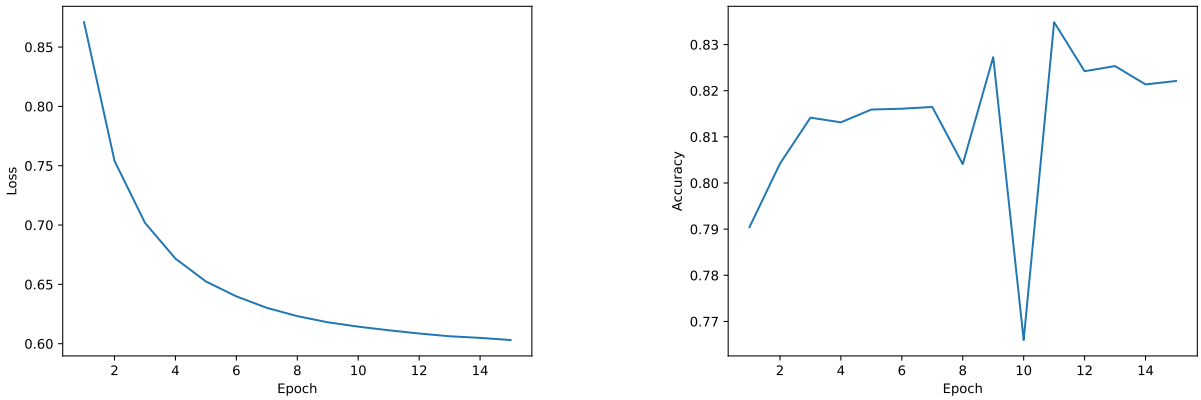
```
            x = self.pool(x)

        x = x.view(-1, 16 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

With this configuration, the network was trained using 15 epochs and SGD tuning, with learning rate equal to 0.1, 0.01 and 0.001. The final validation accuracy and the test results can be seen in Table 2.1. The loss function and the validation accuracy for each scenario throughout the epochs can be seen in Figures 2.1, 2.2 and 2.3, respectively.

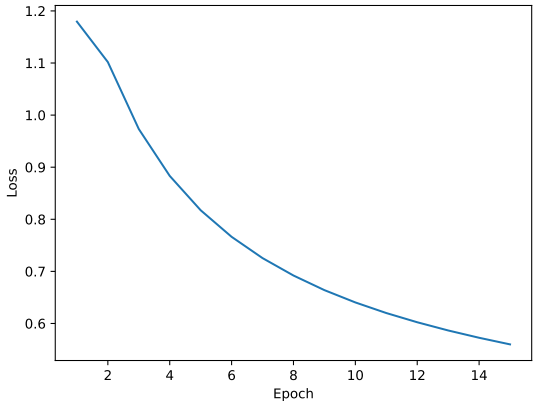
Learning rate	Final validation accuracy	Test accuracy
0.1	0.8221	0.7921
0.01	0.8724	0.8526
0.001	0.6811	0.7259

Table 2.1: Results for different learning rate choices

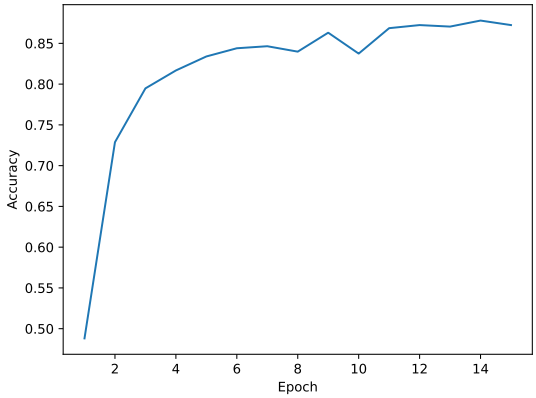


(a) Loss function (b) Validation accuracy

Figura 2.1: Results for the CNN with maxpool and learning rate 0.1

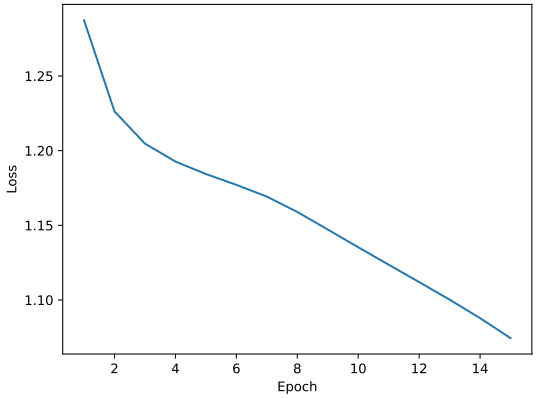


(a) Loss function

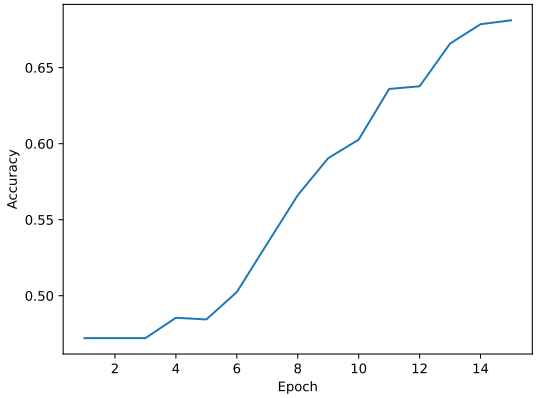


(b) Validation accuracy

Figura 2.2: Results for the CNN with maxpool and learning rate 0.01



(a) Loss function



(b) Validation accuracy

Figura 2.3: Results for the CNN with maxpool and learning rate 0.001

With these results, one can conclude that the best configuration had learning rate equal to 0.01, since it shows the best test accuracy. In addition, it can be seen that the training of the network using learning rate equal to 0.1 shows oscilation of the validation accuracy. Finaly, it appears that 150 epochs was not enough for the value 0.001, so it could be interesting to run it for longer and assess if this configuration achieves better results.

2.2 Item 2.2

For this item, the proposed architecture for the convolutional network is:

- A convolution layer with 8 output channels, a kernel of size $3x3$, padding of 0 and a stride of 2

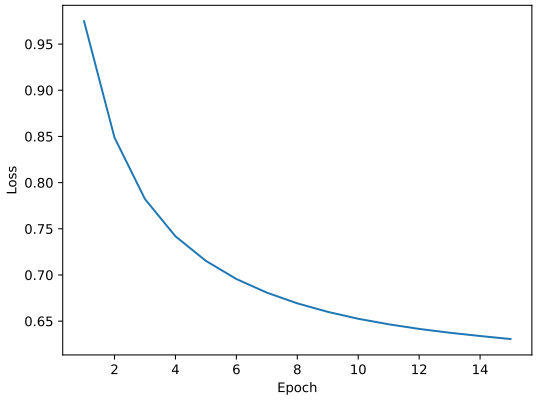
- A rectified linear unit activation function.
- A convolution layer with 16 output channels, a kernel of size 3×3, stride of 2 and padding of zero.
- A rectified linear unit activation function.
- An affine transformation with 320 output features.
- A rectified linear unit activation function.
- A dropout layer with a dropout probability of 0.7 .
- An affine transformation with 120 output features.
- A rectified linear unit activation function.
- An affine transformation with the number of classes followed by an output LogSoftmax layer.

The main differences between this configuration and the previous is the stride of both convolution layers (was equal to 1 and now is equal to 2) and the removal of pooling. Due to the similarity between the two architectures, the same code was used, as shown in item 1.1. To achieve the desired implementation, the parameter `no_maxpool` must be set to `True`.

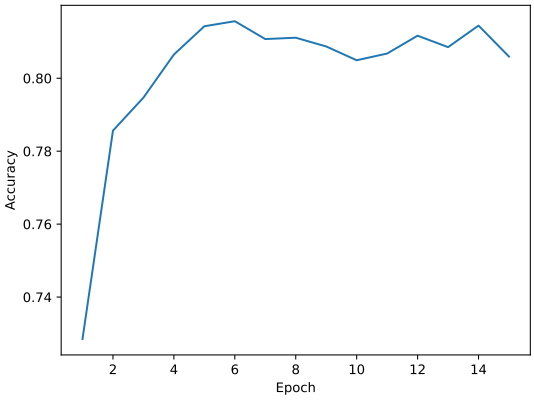
With this, the network was trained using 15 epochs and SGD tuning, with learning rate equal to 0.1, 0.01 and 0.001. The final validation accuracy and the test results can be seen in Table 2.2. The loss function and the validation accuracy for each scenario throughout the epochs can be seen in Figures 2.4, 2.5 and 2.6, respectively.

Learning rate	Final validation accuracy	Test accuracy
0.1	0.8059	0.7977
0.01	0.8514	0.8261
0.001	0.6666	0.6843

Table 2.2: Results for different learning rate choices

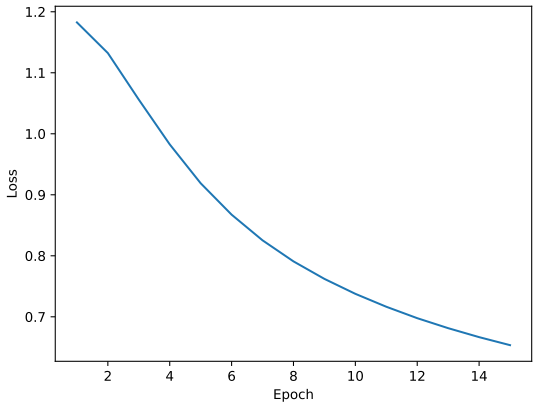


(a) Loss function

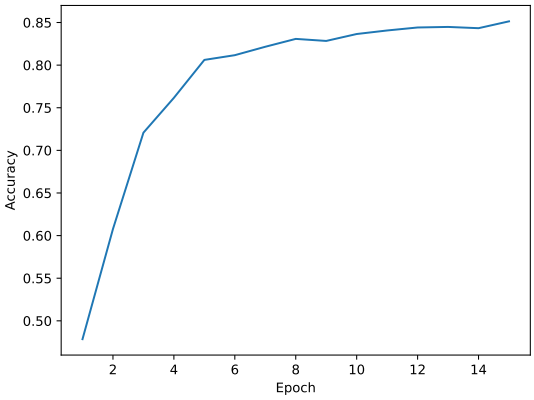


(b) Validation accuracy

Figura 2.4: Results for the CNN with learning rate 0.1 and without pooling

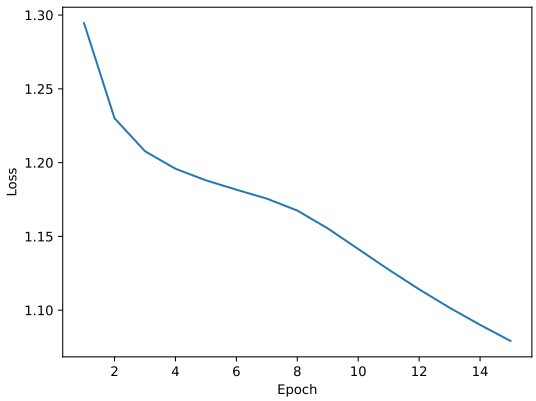


(a) Loss function

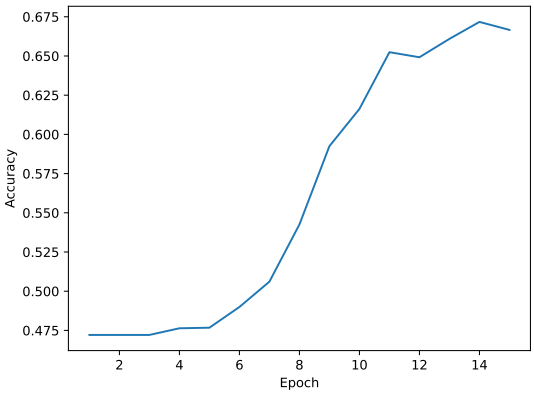


(b) Validation accuracy

Figura 2.5: Results for the CNN with learning rate 0.01 and without pooling



(a) Loss function



(b) Validation accuracy

Figura 2.6: Results for the CNN with learning rate 0.001 and without pooling

As before, it can be seen that the best configuration has learning rate equal to 0.01, since it shows the best test accuracy. The training with the value 0.1 showed less oscilation

than in the previous architecture, and it also likely that 150 epochs was not enough for learning rate equal to 0.001, since the loss function did not stabilize.

2.3 Item 2.3

To evaluate the number of trainable parameters of each architecture, the following function was implemented:

```
1 def get_number_trainable_params(model):
2     model_parameters_cnn = filter(lambda p: p.requires_grad, model.parameters())
3     params_cnn = sum([np.prod(p.size()) for p in model_parameters_cnn])
4     return params_cnn
```

With the application of the function, it was noted that both architectures have 225618 trainable parameters. Despite this, the first architecture has better results, since it has a best test accuracy of 0.85, while for the second configuration the best result has accuracy of 0.83.

The first architecture uses max-pooling, which reduces the dimensionality of the data, makes feature extraction easier and thus helps prevent overfitting. The lack of pooling in the second architecture may lead to less effective feature generalization. In addition, the second architecture uses a higher stride in convolutional layers. This can result in missing details that may be important for the classification task.

As a result, it can be seen that the combination of convolution and max-pooling of the first architecture leads to a better performance on the test dataset. This is likely because it learns more generalizable features that capture the most important parts of the data.

3. Question 3

This item aims to develop two encoder-decoder architectures for an Automatic Speech Recognition (ASR) task. Specifically, the *LJ Speech Dataset* will be used to train an model that uses a recurrent network as decoder and another that uses a transformer instead.

3.1 Item 3.1

The recurrent decoder architecture can be described with the following sequence of steps:

- An embedding layer converts the input text tokens into dense vectors (token embeddings).
- The token embeddings pass through a normalization layer followed by an LSTM.
- A residual cross-attention block will merge and attend the LSTM's output with the encoder's output. The query input will receive the LSTM's output, and the key and value inputs receive the encoder's output.
- The residual block output is normalized.
- The output of the attention mechanism passes through a single linear layer (classifier) to scale the hidden representation to the desired output dimension.

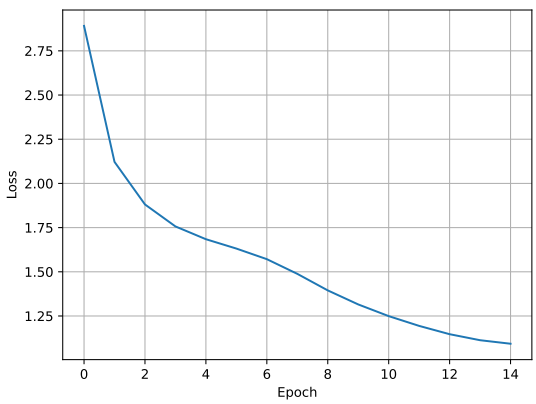
Bellow, the implemented forward pass for this architecture can be seen.

```
def _forward(self, x: torch.Tensor, latent: torch.Tensor, state: Any) ->
    Tuple[torch.Tensor, Any]:
    embedded_tokens = self._embed(x)
    normalized_tokens = self._rnn_norm(embedded_tokens)
    lstm_out, new_state = self._rnn(normalized_tokens, state)
    attention_out = self._cross_att(lstm_out, latent)
    normalized_attention_out = self._out_norm(attention_out)
    output = self._classifier(normalized_attention_out)
    output = output.permute(0,2,1)
    return output, new_state
```

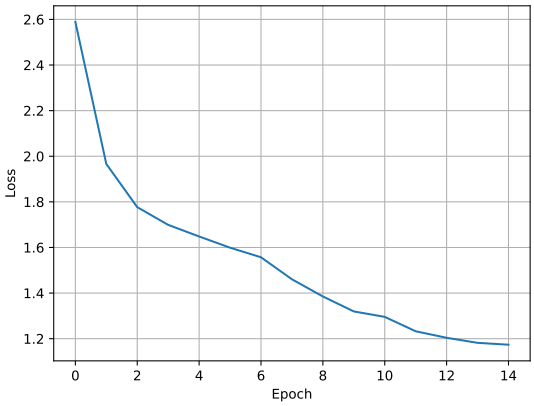
With this, the network was trained for 15 epochs with three different similarities scores: the cosine similarity, the damerau-levenshtein similarity and the jaccard similarity. The model had a final test loss of 1.183, and the test string similarities scores can be seen in Table 3.1. The train and test loss function throughout the epochs can be seen in Figure 3.1, and the similarity scores throughout the epochs can be seen in Figures 3.2a, 3.2b and 3.2c.

Similarity score	Test accuracy
Cosine	0.832
Damerau-Lev.	0.509
Jaccard	0.715

Table 3.1: Results for different similarities

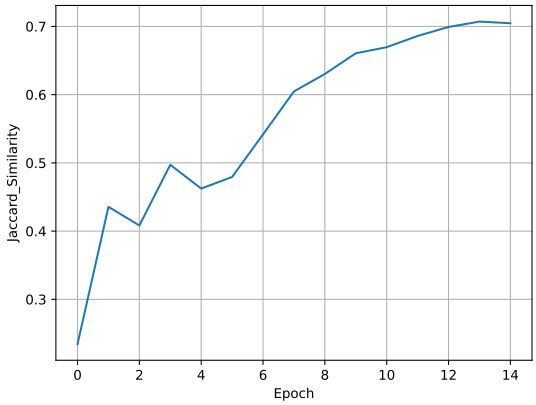


(a) Loss function in training

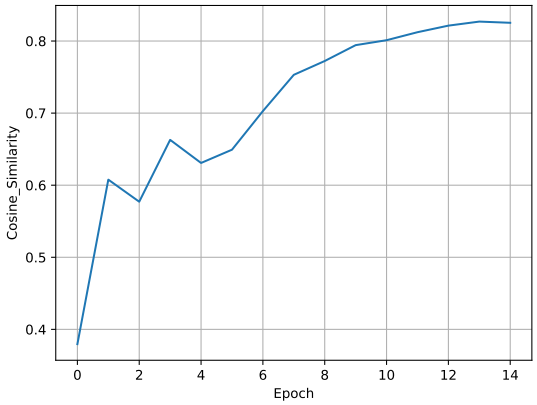


(b) Loss function in validation

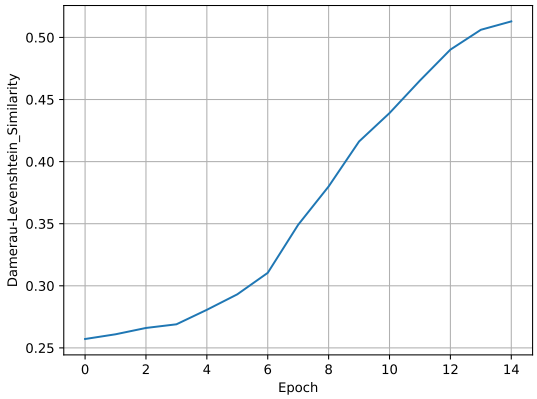
Figura 3.1: Loss functions for the recurrent architecture



(a) Jaccard Similarity



(b) Cosine Similarity



(c) Damerau-Levenshtein similarity

Figura 3.2: Similarity results on validation dataset with the recurrent architecture

3.2 Item 3.2

The transformer decoder architecture can be described with the following sequence of steps:

- A Embedding layer is used to convert the input text tokens into dense vectors.
- A position embeddings matrix is applied to the token embeddings.
- The token embeddings pass through a masked residual attention layer as input for the queries, keys, and values.
- A residual cross-attention block merges and attends to the previous attention's output with the encoder's output. The query input will receive the attention layer output, and the key and value inputs receive the encoder's output.
- The residual block output is normalized.

- The output of the normalization layer passes through a single linear layer to scale the hidden representation to the desired output dimension

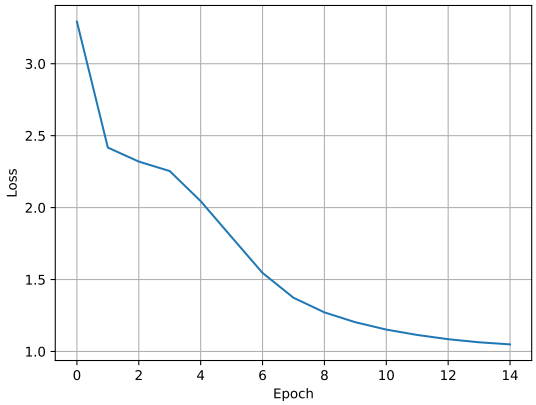
Bellow, the implemented forward pass for this architecture can be seen.

```
def forward(self, x: torch.Tensor, latent: torch.Tensor) -> torch.Tensor:
    token_embeddings = self._embed(x)
    T_d = x.size(1)
    pos_embeddings = self._pos_embed[:T_d, :]
    token_embeddings = token_embeddings + pos_embeddings
    masked_attention_output = self._att.forward(token_embeddings,
mask=self._mask[:T_d, :T_d])
    cross_attention_output = self._cross_att(masked_attention_output, latent)
    normalized_output = self._out_norm(cross_attention_output)
    output = self._classifier(normalized_output)
    output = output.permute(0,2,1)
    return output
```

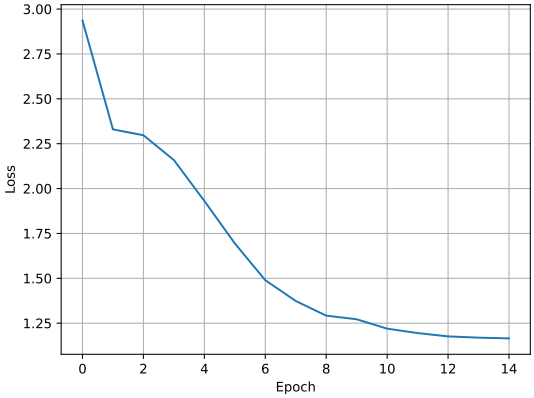
With this, the network was trained using 15 with three different similarities scores: the cosine similarity, the damerau-levenshtein similarity and the jaccard similarity. The model had a final test loss of 1.161, and the test string similarities scores can be seen in Table 3.2. The train and test loss function throughout the epochs can be seen in Figure 3.3, and the similarity scores throughout the epochs can be seen in Figures 3.4a, 3.4b and 3.4c.

Similarity score	Test accuracy
Cosine	0.865
Damerau-Lev.	0.629
Jaccard	0.764

Table 3.2: Results for different similarities

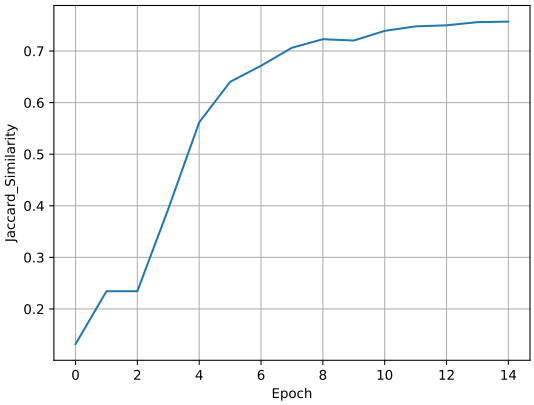


(a) Loss function in training

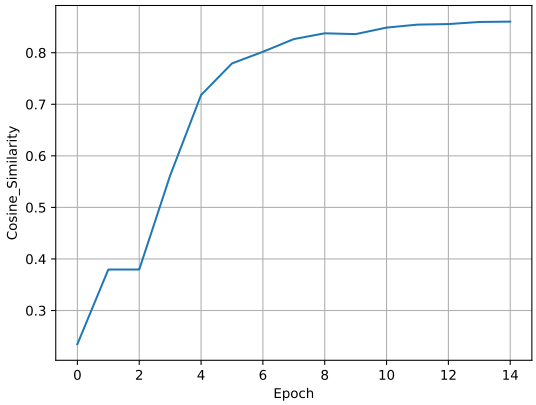


(b) Loss function in validation

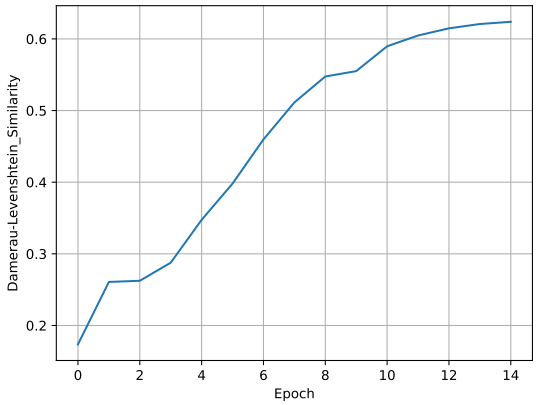
Figura 3.3: Loss functions for the transformer architecture



(a) Jaccard Similarity



(b) Cosine Similarity



(c) Damerau-Levenshtein similarity

Figura 3.4: Similarity results on validation dataset with the transformer architecture

3.3 Item 3.3

Based on the provided data and the structural differences between the LSTM and attention mechanisms, it is possible to comment on the different approaches for processing the text input and on the difference of performance in the ASR task.

First of all, the LSTM processes input text token-by-token sequentially, which allows it to capture contextual information effectively. In addition, LSTMs have an internal state that they update as they process each token. This state acts as a memory that carries information across the sequence, which can be relevant for understanding longer dependencies.

On the other hand, the attention mechanism processes all tokens simultaneously. This allows the model to consider the entire sequence at once, which may lead to capturing more complex patterns the between tokens. Further, through self-attention, the model evaluates the importance of each token relative to the others, which enables it to focus on the most relevant parts of the input sequence when making predictions.

The LSTM-based decoder showed a steady improvement in validation loss and similarity scores over the epochs. It started with a lower initial loss, suggesting it was able to capture the patterns of the task with less data.

The attention-based decoder, however, while starting with a higher validation loss, showed significant improvement, and has a similar performance to the LSTM-based model in the end of training. This suggests that although it took longer to start reducing the loss, its parallel processing capability enabled it to learn effectively from the entire dataset.

In the end of training, both models achieved similar cosine and Jaccard scores. But the Damerau-Levenshtein similarity is better for the second model, which indicates that its overall performance in predicting the sequence is better than the first one.

To conclude, while both models ended up with loss function values, the LSTM-based model was better in the first epochs due to its sequential processing. The attention-based model required more epochs to learn but was able to achieve a similar loss, which shows it captures global patterns in the data. Finally, by looking at the Damerau-Levenshtein similarity score, it can be said that the second model has a better performance than the first.

3.4 Item 3.4

The three similarities measures have different strategies for comparing strings, and this leads to different results. In this context, the cosine similarity evaluates the similarity between two sets of tokens based on the cosine of the angle between their vector representations. It is most useful when the magnitude of the vector is less important than the orientation. The values obtained by cosine similarity in both models were similar.

The Jaccard similarity considers how many tokens are shared between the predicted and reference strings, ignoring the order and repetitions. This metric is appropriate when the presence of the same words is important, but their order and frequency are not.

Finally, the Damerau-Levenshtein distance considers the minimum number of changes that are needed to go from one string into another. Then, the measure is calculated by considering how close this distance is to 0, with lower distances indicating higher similarity. Due to its reasoning, this metric is very sensitive to the exact sequence of tokens.

As a result, the Cosine measure is best used when it is important to evaluate if the model is capturing the correct tokens overall, the Jaccard for to analyze if the right words are being predicted, regardless of the order, and the Damerau-Levenshtein for assessing if the model is producing sequences that are nearly correct, by penalizing the edits required to go from the prediction to the target.

For the first model, the highest similarity is the Cosine (0.832). The second highest is the Jaccard (0.715) and the third is the Damerau-Levenshtein (0.509). For the second model, the same order is true, with values 0.865, 0.764 and 0.629, respectively. It is possible to note that the value for the Damerau-Levenshtein is significantly worse in the first model than in the second model, which indicates that the second model predicts sequences that are closer to the target.