DEEC
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
TÉCNICO LISBOA

Master in Electrical and Computer Engineering

Deep Learning Course

2nd Homework

2nd Period 2024/2025

Authors:
João Gonçalves – ist199995
jrazevedogoncalves@tecnico.ulisboa.pt
Teresa Nogueira – ist1100029
maria.teresa.ramos.nogueira@tecnico.ulisboa.pt

Department of Electrical and Computer Engineering
Instituto Superior Técnico, Lisbon, Portugal

Team Contribution Disclaimer

The elaboration of this report was a collaborative effort, with both students contributing equally to its completion. Question 1 was jointly developed by both members, and we attribute Question 2 to Teresa Nogueira and Question 3 to João Gonçalves.

Question 1 – Modern Hopfield Networks, the Convex-Concave Procedure, and Transformers

In this section, we analyze the energy function of a modern Hopfield network [1], decompose it into convex and concave components, and demonstrate how the convex-concave procedure (CCCP) from [2] can be applied to derive iterative updates. Furthermore, we establish a connection between these updates and the attention mechanism in transformers.

Question 1.1 – Convex-Concave Decomposition of the Energy Function

Let $\mathbf{X} \in \mathbb{R}^{N \times D}$ be a matrix whose rows hold a set of examples $\mathbf{x}_i \in \mathbb{R}^D$ and $\mathbf{q} \in \mathbb{R}^D$ denote the state vector, the considered energy function of the modern Hopfield network is defined as

$$E(\mathbf{q}) = \frac{1}{2}\mathbf{q}^\top \mathbf{q} + \beta^{-1}\log(N) + \frac{1}{2}M^2 - \operatorname{lse}(\beta, \mathbf{X}\mathbf{q}),$$

where $M = \max_i \|\mathbf{x}_i\|$ and $\operatorname{lse}(\beta, z)$ is the log-sum-exp function with temperature β given by

$$\operatorname{lse}(\beta, \mathbf{z}) = \beta^{-1}\log\left[\sum_{i=1}^{N}\exp(\beta z_i)\right].$$

We can decompose the energy function as $E(\mathbf{q}) = E_1(\mathbf{q}) + E_2(\mathbf{q})$ with

$$E_1(\mathbf{q}) = \frac{1}{2}\mathbf{q}^\top \mathbf{q} + \beta^{-1}\log(N) + \frac{1}{2}M^2, \quad E_2(\mathbf{q}) = -\operatorname{lse}(\beta, \mathbf{X}\mathbf{q}).$$

We'll now prove that $E_1(\mathbf{q})$ is convex and that $E_2(\mathbf{q})$ is concave. Let us first consider $E_1(\mathbf{q})$. To determine whether it is convex, we compute its gradient and Hessian. The function is

$$E_1(\mathbf{q}) = \frac{1}{2}\mathbf{q}^\top \mathbf{q} + \beta^{-1}\log(N) + \frac{1}{2}M^2.$$

The terms $\beta^{-1}\log(N)$ and $\frac{1}{2}M^2$ are constant with respect to $\mathbf{q}$, so they do not contribute to the gradient or Hessian. Focusing on the first term, we compute

$$\mathbf{q}^\top \mathbf{q} = \sum_{i=1}^{D} q_i^2.$$

The gradient of $\frac{1}{2}\mathbf{q}^\top \mathbf{q}$ is

$$\nabla\left(\frac{1}{2}\mathbf{q}^\top \mathbf{q}\right) = \nabla\left(\frac{1}{2}\sum_{i=1}^{D} q_i^2\right) = \left[\frac{\partial}{\partial q_1}\frac{1}{2}q_1^2, \ldots, \frac{\partial}{\partial q_D}\frac{1}{2}q_D^2\right]^\top = \mathbf{q}.$$

Thus, the gradient of $E_1(\mathbf{q})$ is

$$\nabla E_1(\mathbf{q}) = \mathbf{q}.$$

The Hessian is computed as

$$\nabla^2 \left(\frac{1}{2} \mathbf{q}^\top \mathbf{q} \right) = \nabla^2 \left(\frac{1}{2} \sum_{i=1}^{D} q_i^2 \right) = \operatorname{diag}(1, 1, \ldots, 1) = I_D,$$

where $I_D \in \mathbb{R}^{D \times D}$ is an identity matrix. Hence, the Hessian of $E_1(\mathbf{q})$ is

$$\nabla^2 E_1(\mathbf{q}) = I_D.$$

The matrix I_D is symmetric and diagonally dominant (i.e., $|I_{Di,i}| \geqslant \sum_{j \neq i} |I_{Di,j}|$ for any i), therefore positive semidefinite (since all eigenvalues are 1). By a standard result in convex analysis (as seen in [3]), a twice-differentiable function is convex if and only if its Hessian is positive semidefinite. Therefore, $E_1(\mathbf{q})$ is convex.

Next, we turn our attention to $E_2(\mathbf{q})$. To establish that it is concave, we'll also compute its gradient and Hessian. The function is given by

$$E_2(\mathbf{q}) = -\operatorname{lse}(\beta, \mathbf{X}\mathbf{q}),$$

where

$$\operatorname{lse}(\beta, \mathbf{z}) = \beta^{-1} \log \left(\sum_{i=1}^{N} \exp(\beta z_i) \right), \quad \mathbf{z} = \mathbf{X}\mathbf{q}.$$

We begin by computing the gradient of $-E_2(\mathbf{q}) = \operatorname{lse}(\beta, \mathbf{X}\mathbf{q})$. Using the chain rule

$$\nabla \operatorname{lse}(\beta, \mathbf{X}\mathbf{q}) = \mathbf{X}^\top \nabla_{\mathbf{z}} \operatorname{lse}(\beta, \mathbf{z}),$$

where $\mathbf{z} = \mathbf{X}\mathbf{q}$. For $\operatorname{lse}(\beta, \mathbf{z}) = \beta^{-1} \log \left(\sum_{i=1}^{N} \exp(\beta z_i) \right)$, we compute

$$\nabla_{\mathbf{z}} \operatorname{lse}(\beta, \mathbf{z}) = \frac{\beta \left[\exp(\beta z_1), \ldots, \exp(\beta z_N) \right]^\top}{\sum_{i=1}^{N} \exp(\beta z_i)} = \operatorname{softmax}(\beta \mathbf{z}).$$

Thus

$$\nabla \left(-E_2(\mathbf{q}) \right) = \nabla \operatorname{lse}(\beta, \mathbf{X}\mathbf{q}) = \mathbf{X}^\top \operatorname{softmax}(\beta \mathbf{X}\mathbf{q}).$$

Next, we compute its Hessian. Using the chain rule, the Hessian is

$$\nabla^2 \operatorname{lse}(\beta, \mathbf{X}\mathbf{q}) = \mathbf{X}^\top \nabla_{\mathbf{z}}^2 \operatorname{lse}(\beta, \mathbf{z}) \mathbf{X}, \quad \mathbf{z} = \mathbf{X}\mathbf{q}.$$

The Jacobian of $\operatorname{softmax}(\beta \mathbf{z})$ gives the Hessian

$$\frac{\partial p_i}{\partial z_j} = \begin{cases} \beta p_i (1 - p_i), & \text{if } i = j, \\ -\beta p_i p_j, & \text{if } i \neq j. \end{cases}$$

In matrix form

$$\nabla_{\mathbf{z}}^2 \operatorname{lse}(\beta, \mathbf{z}) = \beta \Big[\operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top \Big],$$

where $\operatorname{diag}(\mathbf{p})$ is the diagonal matrix with entries $p_1, p_2, \ldots, p_N$, and $\mathbf{p}\mathbf{p}^\top$ is the outer product of $\mathbf{p}$ with itself.

Thus, the Hessian of $-E_2(\mathbf{q})$ is

$$\nabla^2 \big(-E_2(\mathbf{q}) \big) = \mathbf{X}^\top \Big[\beta \big(\operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top \big) \Big] \mathbf{X}.$$

To confirm positive semidefiniteness, let $\mathbf{v} \in \mathbb{R}^N$. For the term $\operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$, we compute

$$\mathbf{v}^\top \Big(\operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top \Big) \mathbf{v} = \sum_{i=1}^{N} p_i v_i^2 - \Big(\sum_{i=1}^{N} p_i v_i \Big)^2 .$$

By the Cauchy-Schwarz inequality, this difference is non-negative

$$\sum_{i=1}^{N} p_i v_i^2 \geqslant \Big(\sum_{i=1}^{N} p_i v_i \Big)^2 .$$

This shows that $\operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top$ is positive semidefinite. Since a congruence transformation (multiplication by $\mathbf{X}^\top$ and $\mathbf{X}$) preserves positive semidefiniteness, $\nabla^2(-E_2(\mathbf{q}))$ is positive semidefinite. Thus, $-E_2(\mathbf{q})$ is convex, and $E_2(\mathbf{q})$ is concave. □

Question 1.2 – Iterative Optimization via CCCP

Let $E(\mathbf{q}) = E_1(\mathbf{q}) + E_2(\mathbf{q})$ be the energy function, where $E_1(\mathbf{q}) = \frac{1}{2}\mathbf{q}^\top\mathbf{q} + \beta^{-1}\log(N) + \frac{1}{2}M^2$ is convex and $E_2(\mathbf{q}) = -\operatorname{lse}(\beta, \mathbf{X}\mathbf{q})$ is concave (as seen in Q1.1). The CCCP is an iterative algorithm for finding local minima of such functions, which proceeds as follows at each iteration:

1. Linearize the concave component $E_2(\mathbf{q})$ around the current iteration's $\mathbf{q}_t$ using a first-order Taylor approximation
$$E_2(\mathbf{q}) \approx \tilde{E}_2(\mathbf{q}) := E_2(\mathbf{q}_t) + \nabla E_2(\mathbf{q}_t)^\top(\mathbf{q} - \mathbf{q}_t);$$
2. Solve the resulting convex optimization problem
$$\mathbf{q}_{t+1} = \operatorname*{argmin}_{\mathbf{q}} \left[E_1(\mathbf{q}) + \tilde{E}_2(\mathbf{q})\right].$$

The logic of the CCCP algorithm is based on iterative convexification: by linearizing $E_2(\mathbf{q})$, the resulting optimization problem becomes convex and solvable at each step. This guarantees that the energy decreases at every iteration under mild assumptions [2], leading to convergence to a local minimum.

Substituting $E_2(\mathbf{q}) = -\operatorname{lse}(\beta, \mathbf{X}\mathbf{q})$, we compute the gradient
$$\nabla E_2(\mathbf{q}) = -\nabla \operatorname{lse}(\beta, \mathbf{X}\mathbf{q}) = -\mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}).$$

Thus, the linearized approximation becomes
$$\tilde{E}_2(\mathbf{q}) = E_2(\mathbf{q}_t) - \left(\mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t)\right)^\top(\mathbf{q} - \mathbf{q}_t).$$

Next, we solve the optimization problem
$$\mathbf{q}_{t+1} = \operatorname*{argmin}_{\mathbf{q}} \left[E_1(\mathbf{q}) + \tilde{E}_2(\mathbf{q})\right].$$

Substituting $E_1(\mathbf{q}) = \frac{1}{2}\mathbf{q}^\top\mathbf{q} + \beta^{-1}\log(N) + \frac{1}{2}M^2$ and $\tilde{E}_2(\mathbf{q})$, the optimization problem becomes
$$\mathbf{q}_{t+1} = \operatorname*{argmin}_{\mathbf{q}} \left[\frac{1}{2}\mathbf{q}^\top\mathbf{q} + \beta^{-1}\log(N) + \frac{1}{2}M^2 - \left(\mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t)\right)^\top(\mathbf{q} - \mathbf{q}_t)\right].$$

Taking the gradient of the objective function with respect to $\mathbf{q}$ (constants vanish), we can find
$$\nabla\left(\frac{1}{2}\mathbf{q}^\top\mathbf{q} - \left(\mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t)\right)^\top\mathbf{q}\right) = \mathbf{q} - \mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t).$$

Setting the gradient to zero yields
$$\mathbf{q} - \mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t) = 0 \implies \mathbf{q}_{t+1} = \mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t).$$

In this optimization, finding the zero of the gradient corresponds to finding a stationary point of the objective function, which is a minimum in this case due to the convexity of $E_1(\mathbf{q})$ and the linearization of concave $E_2(\mathbf{q})$. Thus, applying the CCCP algorithm to the target energy function produces the iterative update
$$\mathbf{q}_{t+1} = \mathbf{X}^\top \operatorname{softmax}(\beta\mathbf{X}\mathbf{q}_t).$$

□

Question 1.3 – Linking Hopfield Updates to Transformer Attention

The Hopfield network update derived in Q1.2 is given by

$$\mathbf{q}_{t+1} = \mathbf{X}^\top \operatorname{softmax}\left(\beta \mathbf{X}\mathbf{q}_t\right),$$

where we take $\beta = 1/\sqrt{D}$ as the temperature, $\mathbf{X} \in \mathbb{R}^{N\times D}$ as the matrix of stored examples, and $\mathbf{q}_t \in \mathbb{R}^D$ as the current state vector. We aim to compare this update to the cross-attention mechanism in a transformer.

The transformer cross-attention mechanism for a query matrix $\mathbf{Q}_t \in \mathbb{R}^{N\times D}$, keys $\mathbf{K} \in \mathbb{R}^{N\times D}$, and values $\mathbf{V} \in \mathbb{R}^{N\times D}$ is defined as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \operatorname{softmax}\left(\frac{\mathbf{Q}_t\mathbf{K}^\top}{\sqrt{D}}\right)\mathbf{V}.$$

Here, the problem specifies that the projection matrices are identity matrices: $\mathbf{W}_K = \mathbf{W}_V = \mathbf{I}$. This means that no learned transformations are applied to the input $\mathbf{X}$ to generate the keys and values. Consequently, we have

$$\mathbf{K} = \mathbf{X}, \text{ and } \mathbf{V} = \mathbf{X}.$$

The query matrix is $\mathbf{Q}_t = [\mathbf{q}_t^{(1)}, \ldots, \mathbf{q}_t^{(N)}]^\top \in \mathbb{R}^{N\times D}$, where $\mathbf{q}_t^{(i)}$ corresponds to each individual query vector.

Substituting $\mathbf{K} = \mathbf{X}$ and $\mathbf{V} = \mathbf{X}$ into the attention formula above, we have

$$\mathbf{Q}_{\text{out}} = \operatorname{softmax}\left(\frac{\mathbf{Q}_t\mathbf{X}^\top}{\sqrt{D}}\right)\mathbf{X}.$$

For a single query $\mathbf{q}_t \in \mathbb{R}^D$ (the i-th row of $\mathbf{Q}_t$), the output of the attention mechanism is

$$\mathbf{q}_{\text{out}} = \operatorname{softmax}\left(\frac{\mathbf{q}_t^\top\mathbf{X}^\top}{\sqrt{D}}\right)\mathbf{X}.$$

To compare this with the Hopfield update, recall that it is expressed as

$$\mathbf{q}_{t+1} = \mathbf{X}^\top \operatorname{softmax}\left(\frac{\mathbf{X}\mathbf{q}_t}{\sqrt{D}}\right).$$

Both computations involve a softmax operation over a scaled dot product between the query and the stored examples. In the attention mechanism, the argument to the softmax is $\mathbf{q}_t\mathbf{X}^\top/\sqrt{D}$, while in the Hopfield update, the argument is $\mathbf{X}\mathbf{q}_t/\sqrt{D}$. These arguments are transposes of one another, this is,

$$\left(\mathbf{q}_t^\top\mathbf{X}^\top\right)^\top = \mathbf{X}\mathbf{q}_t.$$

Since the softmax operation is invariant to transposition when applied to a single axis, the resulting probabilities are identical but the dimensions of the resulting vectors will be transposed

$$\operatorname{softmax}\left(\frac{\mathbf{q}_t^\top\mathbf{X}^\top}{\sqrt{D}}\right) = \left[\operatorname{softmax}\left(\frac{\mathbf{X}\mathbf{q}_t}{\sqrt{D}}\right)\right]^\top.$$

Another difference lies in the aggregation step. In the Hopfield update, the weighted sum of the stored examples is computed as

$$\mathbf{q}_{t+1} = \mathbf{X}^\top \operatorname{softmax}\left(\frac{\mathbf{X}\mathbf{q}_t}{\sqrt{D}}\right),$$

whereas in the transformer attention mechanism, the aggregation is performed as

$$\mathbf{q}_{\text{out}} = \operatorname{softmax}\left(\frac{\mathbf{q}_t^\top \mathbf{X}^\top}{\sqrt{D}}\right)\mathbf{X}.$$

Despite the reversed order of matrix multiplications, these operations are naturally equivalent, as the same values are multiplied with each other (the same happens inside softmax). This can be intuitively understood refering to the generic matrix multiplication depiction in Figure 1.1.

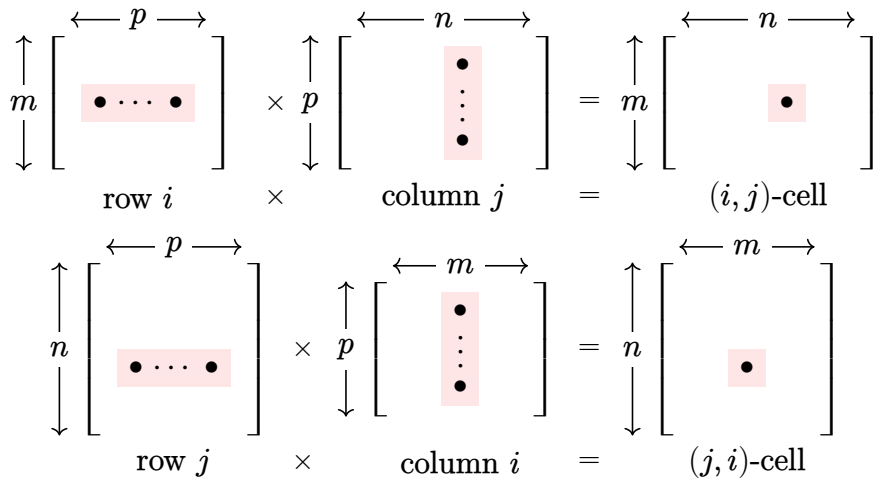


Fig. 1.1. Matrix multiplication. Above we can visualize $AB = C$, and below we can see $B^\top A^\top = C^\top$.

After this discussion, we can go back and consider the full query matrix $\mathbf{Q}_t$. The Hopfield network, generalized to update all N queries simultaneously, will compute

$$\mathbf{Q}_{t+1} = \mathbf{X}^\top \operatorname{softmax}\left(\frac{\mathbf{X}\mathbf{Q}_t^\top}{\sqrt{D}}\right),$$

while the transformer computes

$$\mathbf{Q}_{\text{out}} = \operatorname{softmax}\left(\frac{\mathbf{Q}_t\mathbf{X}^\top}{\sqrt{D}}\right)\mathbf{X}.$$

The key observation is that $\mathbf{Q}_{\text{out}}$ and $\mathbf{Q}_{t+1}$ are related by a transpose, i.e.,

$$\mathbf{Q}_{t+1}^\top = \mathbf{Q}_{\text{out}}.$$

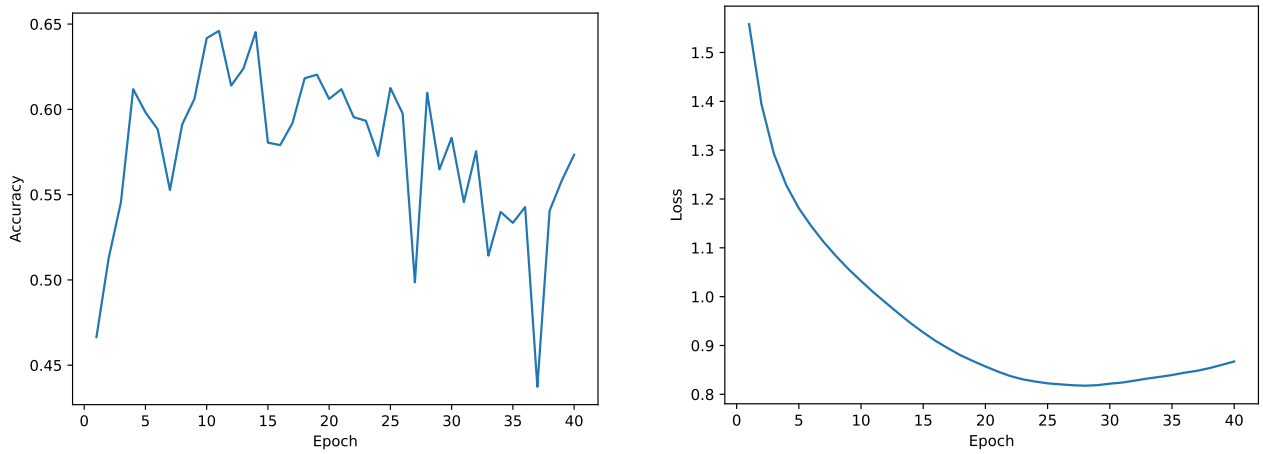
This equivalence arises because the softmax operation and matrix multiplications distribute weights identically, regardless of whether the aggregation is performed row-wise (transformer) or column-wise (Hopfield). Thus, our Hopfield update (with $\beta = 1/\sqrt{D}$) is mathematically equivalent to the cross-attention computation in a transformer with identity projection matrices for keys and values. □

Question 2 – Image Classification with CNN

In this section, we implement and evaluate convolutional neural networks (CNNs) for the task of image classification using the Intel Image Classification dataset. The goal is to explore different architectural and functional enhancements to improve classification performance. We begin by constructing a simple CNN model with convolutional blocks consisting of convolution, activation, pooling, and dropout layers. Next, we extend this model by introducing batch normalization (BN) and global average pooling (GAP), followed by a comparison of the models in terms of their number of trainable parameters and performance. Finally, we analyze the impact of kernel size and pooling layers on the network's efficiency and accuracy.

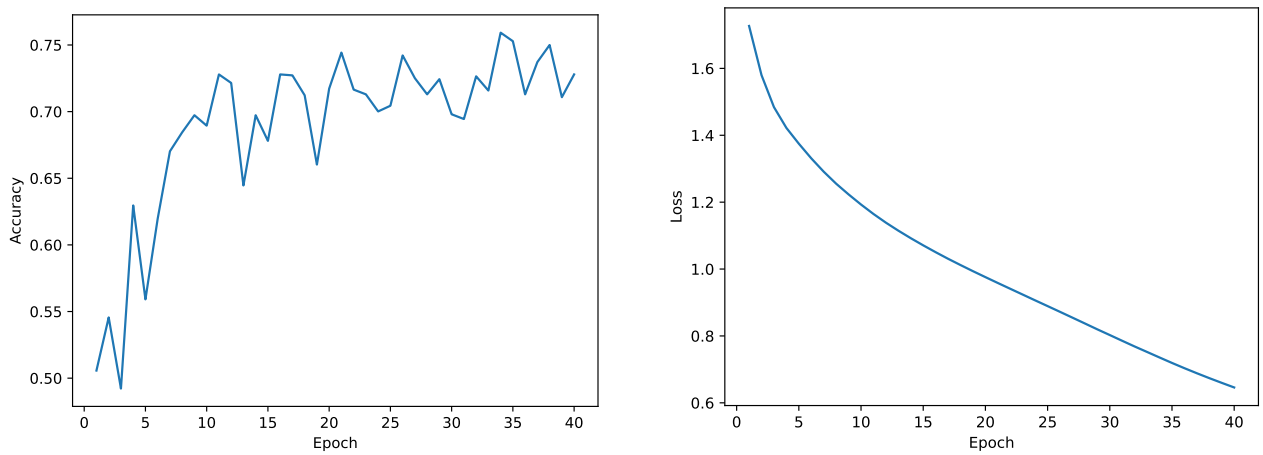
Question 2.1 – Vanilla CNN

To analyze the performance of the simpler CNN, we present the validation accuracy and training loss across 40 epochs for three different learning rates: $\mu = 0.1$, $\mu = 0.01$ and $\mu = 0.001$. The plots of accuracy and loss are shown below in Figures 2.1, 2.2 and 2.3. Lastly, Table 2.1 summarizes the results after 40 epochs.



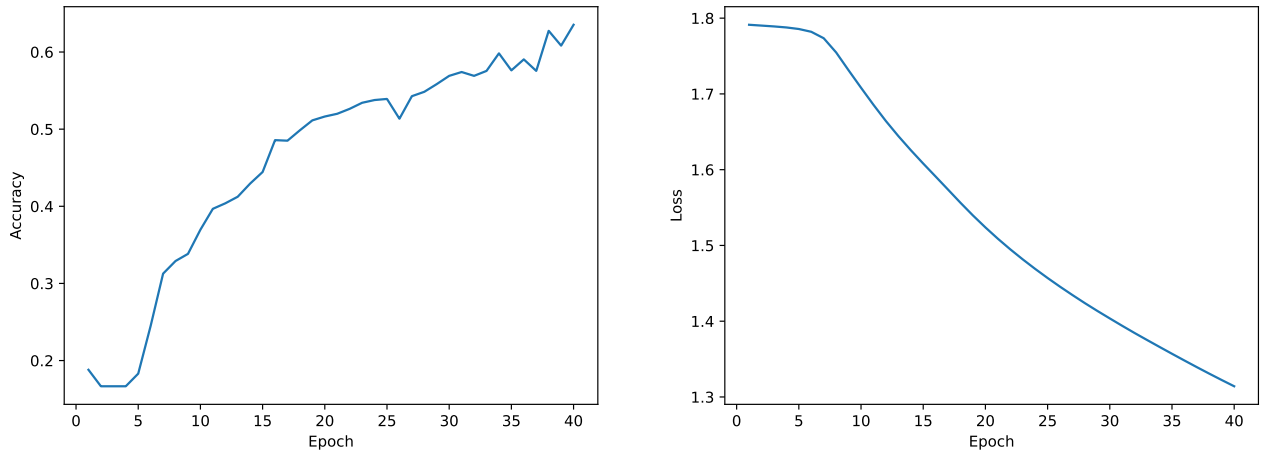
(a) Accuracy. **(b)** Loss.

Fig. 2.1. Validation accuracy (left) and training loss (right) across epochs for a learning rate of 0.1.



(a) Accuracy. **(b)** Loss.

Fig. 2.2. Validation accuracy (left) and training loss (right) across epochs for a learning rate of 0.01.



(a) Accuracy. **(b)** Loss.

Fig. 2.3. Validation accuracy (left) and training loss (right) across epochs for a learning rate of 0.001.

Learning Rate	Train Loss	Validation Loss	Validation Acc	Test Acc
0.1	0.8672	1.2963	0.5734	0.5533
0.01	0.6459	1.3102	0.7123	0.6907
0.001	1.3141	0.9911	0.6353	0.6203

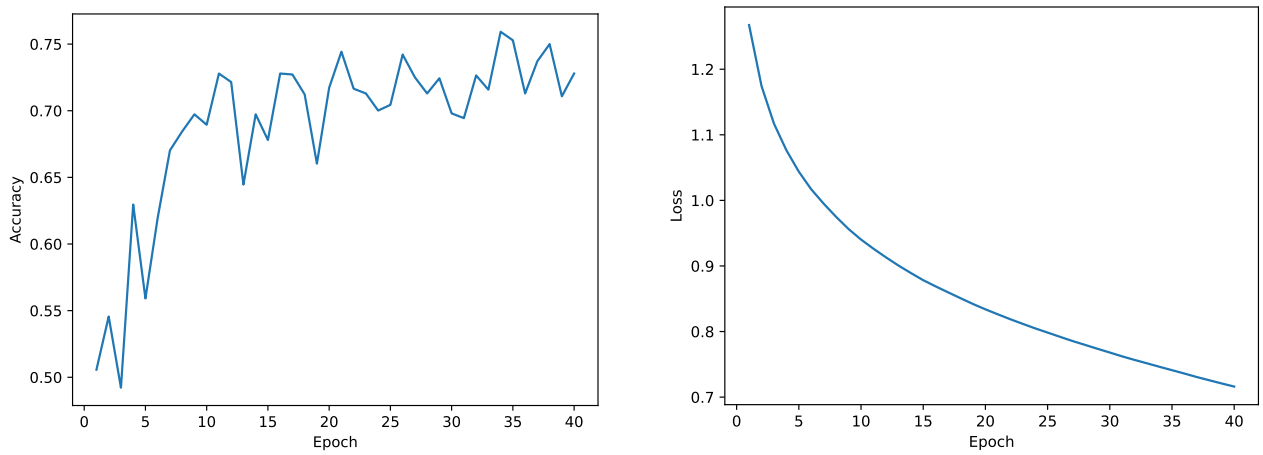
Tab. 2.1. CNN accuracy and loss after 40 epochs for different values of learning rate.

The learning rate that achieved the highest validation accuracy was 0.01.

For a learning rate of $\mu = 0.001$, the loss curve is smoother but convergence is significantly slower, resulting in a higher loss after 40 epochs compared to $\mu = 0.01$. Conversely, $\mu = 0.1$ leads to an unstableloss curve due to overly aggressive updates. Therefore, the best performance is achieved with $\mu = 0.01$, offering balance between convergence speed and training stability.

Question 2.2 – CNN with Batch Normalization and Global Average Pooling

The plots of accuracy and loss for the CNN with optimal configuration ($\mu = 0.01$) are shown below in Figures 2.1, 2.2 and 2.3. Table 2.2 summarizes the results after 40 epochs.



(a) Accuracy. **(b)** Loss.

Fig. 2.4. Validation accuracy (left) and training loss (right) across epochs for a learning rate of 0.01.

Learning Rate	Train Loss	Validation Loss	Validation Acc	Test Acc
0.01	0.7161	0.7506	0.7279	0.7440

Tab. 2.2. CNN accuracy and loss after 40 epochs for a learning rate of $\mu = 0.01$.

The CNN with BN and GAP performs better at a learning rate of 0.01 than the vanilla CNN because BN stabilizes and accelerates training by requiring less iterations to converge to a given loss value, and GAP reduces overfitting by minimizing the number of parameters, as we will discuss in Q2.3 and Q2.4.

Question 2.3 – Parameter Analysis and Comparison

In both CNN's, after the convolutional layers, the feature map size is $6 \times 6 \times 128$. This happens because the input image, 48×48, is halved in spatial dimensions by max pooling (with 2×2 kernel, and stride equal to 2) after each of the 3 ConvBlocks, i.e.,

$$48 \times 48 \rightarrow 24 \times 24 \rightarrow 12 \times 12 \rightarrow 6 \times 6.$$

The final spatial size is 6×6, and the last convolutional block outputs 128 channels, resulting in $6 \times 6 \times 128$, as specified before. For the vanilla CNN, after flattening, the size of the input features will be

$$\text{number of output channels} \times \text{output width} \times \text{output height} = 46084.$$

Which means the number of parameters in the next dense layer will be

$$4608 \times 1024 + 1024 = 4719616.$$

On the other hand, the application of `AdaptiveAvgPool2d(output_size)`, for the CNN with BN and GAP, works as the following:

- **Input Shape:** The input to the layer is a 4D tensor with the shape

  $$[\text{batch size, channels, height, width}];$$

- **Output Shape:** The target output spatial size becomes `(target_height, target_width)`.

To implement GAP, we set `output_size = (1, 1)`. This means the layer will compute the average of all values in the height and width dimensions for each channel, resulting in a spatial size of 1×1. The output shape becomes

$$[\text{batch size, channels, 1, 1}].$$

For the CNN with BN and GAP, after flattening, the size of the input features will be

$$\text{number of output channels} \times \text{output width} \times \text{output height} = 128.$$

Which means the number of parameters in the next dense layer will be

$$128 \times 1024 + 1024 = 131200.$$

Which is reasonably smaller when compared to the case without GAP. The total number of trainable parameters for each CNN can be checked in table 2.3.

CNN	Number of trainable parameters
Vannila	5340742
+ BN and GAP	755718

Tab. 2.3. Number of trainable parameters for each CNN.

The difference in the number of parameters and performance is justified by the use of GAP in the BN CNN, which reduces the spatial size to 1×1, drastically lowering the input size to the fully connected layers (from 46084 to 128), and thus significantly reducing parameter count.

Question 2.4 – Analysis of Kernels and Pooling Layers

A) Kernel Size: The core idea of CNNs is representation learning: gradually extracting meaningful features from raw image data. By using smaller kernel sizes there is a larger space to capture new representations, compared to a kernel that matches image size. Additionally:

- Smaller kernels will have smaller receptive field, which means they will look at very few pixels at once whereas a large kernel will "look" at a larger field view. This in turn means that features extracted by small kernels are highly local, whereas the ones extracted larger kernels are more generic and spread across the image.
- Small kernels extract small complex features where larger kernels extract simpler features due to their larger view field.
- Smaller kernels also lead to slower reduction of image dimensions, making the network deeper. Stacking multiple layers of small kernels allows the network to gradually build hierarchical features, from local to global patterns [4].
- Lastly, small kernels benefit from better weight sharing, as the number of parameters is lower compared to larger kernels[1].

B) Effect of Pooling Layers: Besides reducing the dimensions of the feature maps, as discussed in Q2.3, pooling also plays an important role in robust feature detection:

1. Pooling reduces the spatial size of the feature maps. Pooling the output layer reduces the input dimension for the next layer thus saving computation. Which makes it computationally feasible to have deeper CNNs. The formula to calculate the output dimensions after pooling is given by
$$\left(\frac{n_h - f + 1}{s}, \frac{n_w - f + 1}{s}\right) \times n_c.$$
Where n_h is the height of feature map, f is the size of the pooling filter, s is the stride length, n_w is the weight of feature map and n_c is the number of channels in the feature map. If we apply this formula to our parameters, after the 3 convolutional blocks we get the following spatial dimensions
$$48 \times 48 \rightarrow 24 \times 24 \rightarrow 12 \times 12 \rightarrow 6 \times 6,$$
as previously discussed.

[1] For example, the number of weights in two 3×3 kernels is $3 \times 3 + 3 \times 3 = 18$, whereas the number of weights in a single 5×5 kernel would be 25.

2. Pooling contributes to robust feature detection by summarizing feature activations in local regions, replacing precise positional information with a more generalized representation. This allows subsequent layers to operate on these summarized features, making the model more robust to variations in the position of features in the input image, such as shifts or distortions. For example, a minor change in an image will reflect a total change in feature maps. To solve this problem we downsample the feature maps which is done by pooling layers. Polling Layers give flexibility by filtering the important features from the unimportant ones, increasing the receptive field and reducing the risk of overfitting, all while having zero trainable parameters and being fast to compute.

Question 3 – Phoneme-to-Grapheme Conversion

In this section, we implement a character-level sequence-to-sequence model to solve the phoneme-to-grapheme (P2G) task, which involves predicting the spelling of words from their phonemic transcriptions. The task is evaluated using character error rate (CER) and word error rate (WER) metrics.

Question 3.1 – Character-Level Sequence-to-Sequence Modeling

A) Vanilla Encoder-Decoder Model: As a starting point, we implemented an encoder-decoder model using bidirectional long short-term memory (LSTM) networks for both the encoder and decoder components. The model was trained for 20 epochs with a hidden size of 128 and a dropout rate of 0.3 (these values will also be considered in the following task). Figure 3.1 displays the trend of the CER during training, indicating the model's learning progression.

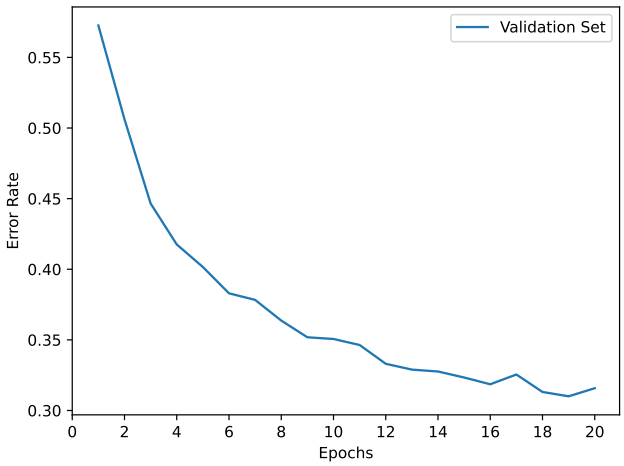


Fig. 3.1. Training CER over epochs for the vanilla encoder-decoder model.

Upon evaluation, the model achieved a CER of 0.2992 and a WER of 0.7970 on the test set, as summarized in Table 3.1.

CER	WER
0.2992	0.7970

Tab. 3.1. Test set evaluation metrics for the vanilla encoder-decoder model.

B) Bahdanau Attention Mechanism: To enhance the vanilla encoder-decoder model, we integrated the Bahdanau attention mechanism [5]. This addition enables the decoder to selectively focus on relevant parts of the input phoneme sequence during each decoding step, thereby improving the alignment between input and output sequences. Figure 3.2 illustrates the training CER over epochs with the attention mechanism in place.

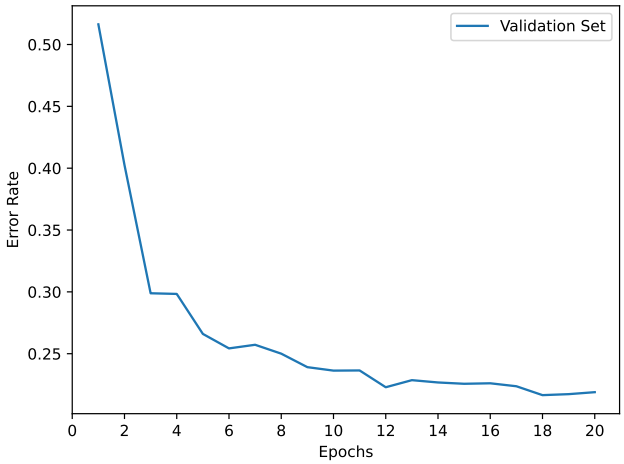


Fig. 3.2. Training CER over epochs for the encoder-decoder model with Bahdanau attention.

The improved model achieved a CER of 0.2033 and a WER of 0.7160 on the test set, as detailed in Table 3.2.

CER	WER
0.2033	0.7160

Tab. 3.2. Test set evaluation metrics for the encoder-decoder model with Bahdanau attention.

C) Nucleus Sampling for Decoding: To explore alternative decoding strategies, we implemented nucleus (top-p) sampling [6] on top of the previous attention-enhanced model. Unlike greedy decoding, which selects the highest probability token at each step, nucleus sampling considers the smallest set of top tokens whose cumulative probability exceeds a threshold p (set to 0.8 in our experiments). This approach aims to balance diversity and coherence in the generated graphemes.

The model exhibited a higher CER of 0.2242 and a WER of 0.7660 on the test set, as presented in Table 3.3. Additionally, the WER@3 (i.e., percentage of examples for which none of the model's 3 predictions are correct) was recorded at 0.6340.

CER	WER	WER@3
0.2242	0.7660	0.6340

Tab. 3.3. Test set evaluation metrics for the model with nucleus sampling.

These results show an improvement over the vanilla encoder-decoder model but a slight degradation compared to the attention-enhanced model with greedy sampling. The increase in CER and WER suggests that while nucleus sampling introduces beneficial diversity, it may also lead to increased variability in predictions.

References

[1] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, T. A. M. Widrich, L. Gruber, M. Holzleitner, M. Pavlović, G. Sandve *et al.*, "Hopfield Networks is All You Need," 2020, arXiv preprint arXiv:2008.02217.

[2] A. Yuille and A. Rangarajan, "The Concave-Convex Procedure," *Neural Computation*, vol. 15, no. 4, pp. 915–936, 2003.

[3] L. V. S. Boyd, *Convex Optimization*. Cambridge, 2009.

[4] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the Effective Receptive Field in Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, vol. 29, 2016, pp. 4898–4906.

[5] Y. B. Dzmitry Bahdanau, Kyunghyun Cho, "Neural Machine Translation by Jointly Learning to Align and Translate," 2016, arXiv preprint arXiv:1409.0473.

[6] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The Curious Case of Neural Text Degeneration," 2019, arXiv preprint arXiv:1904.09751.