

# Basics on deep learning for vision

## **Deep Learning Practical Work**

Aymeric DELEFOSSE & Charles VIN

2023 – 2024



# Contents

<b>1</b>	<b>Introduction to neural networks</b>	<b>2</b>
1.1	Theoretical Foundation	2
1.1.1	Supervised dataset	2
1.1.2	Network architecture	2
1.1.3	Loss function	3
1.1.4	Optimization algorithm	4
1.2	Implementation	8
1.2.1	Forward and backward manuals	8
1.2.2	Simplification of the backward pass with <code>torch.autograd</code>	9
1.2.3	Simplification of the forward pass with <code>torch.nn</code> layers	9
1.2.4	Simplification of the SGD with <code>torch.optim</code>	10
1.2.5	MNIST application	11
1.2.6	SVM	12
<b>2</b>	<b>Introduction to convolutional networks</b>	<b>14</b>
2.1	Questions	14
2.2	Training <i>from scratch</i> of the model	16
2.2.1	Network architecture	16
2.2.2	Network learning	17
2.3	Results improvements	19
2.3.1	Standardization of examples	19
2.3.2	Increase in the number of training examples by <i>data increase</i>	21
2.3.3	Variants of the optimization algorithm	22
2.3.4	Regularization of the network by <i>dropout</i>	25
2.3.5	Use of <i>batch normalization</i>	27
2.3.6	Combination of all methods	28
<b>3</b>	<b>Introduction to transformers</b>	<b>29</b>
3.1	Self-Attention	29
3.2	Multi-head self-attention	30
3.3	Transformer block	30
3.4	Full ViT model	31
3.5	Experiment on MNIST!	31
3.5.1	Hyperparameters influences	31
3.6	Larger transformers	34
3.6.1	Questions	34
3.6.2	Learning from scratch	34
3.6.3	Fine-tuning	35
3.7	Experiments on Fashion MNIST!	35
<b>4</b>	<b>Appendix</b>	<b>37</b>

# Chapter 1

## Introduction to neural networks

### 1.1 Theoretical Foundation

#### 1.1.1 Supervised dataset

1. ★ **What are the train, val and test sets used for?** The training dataset is utilized to train the model, while the test dataset is employed to assess the model's performance on previously unseen data. Lastly, the validation set constitutes a distinct subset of the dataset employed for the purpose of refining and optimizing the model's hyperparameters.

2. **What is the influence of the number of examples  $N$ ?** A larger number of examples can enhance the model's capacity to generalize and improve its robustness against noise or outliers. Conversely, a smaller number of examples can make the model susceptible to overfitting. It is important to note that increasing the dataset size can also lead to an escalation in the computational complexity of the model training process.

#### 1.1.2 Network architecture

3. **Why is it important to add activation functions between linear transformations?** Otherwise, we would simply be aggregating linear functions, resulting in a linear output. Activation functions introduce non-linearity to the network, enabling the model to capture and learn more intricate patterns than those achievable through linear transformations alone.

4. ★ **What are the sizes  $n_x$ ,  $n_h$ ,  $n_y$  in the figure 1? In practice, how are these sizes chosen?**

- $n_x = 2$  represents the input size (data dimension).
- $n_h = 4$  represents the hidden layer size, selected based on the desired complexity of features to be learned in the hidden layer. An excessively large size can result in overfitting.
- $n_y = 2$  represents the output size, determined according to the number of classes in  $y$ .

5. **What do the vectors  $\hat{y}$  and  $y$  represent? What is the difference between these two quantities?**  $y \in \{0, 1\}$  represents the ground truth, where the values are binary (0 or 1).  $\hat{y} \in [0, 1]$  represents a probability-like score assigned to each class by the model.  $\hat{y}$  reflects the model's level of confidence in its predictions for each class.

6. **Why use a SoftMax function as the output activation function?** The reason for employing the SoftMax function is to transform  $\tilde{y} \in \mathbb{R}$  into a probability distribution. While there are several methods to achieve this transformation, SoftMax is a commonly utilized choice, especially in multi-class classification problems.

7. **Write the mathematical equations allowing to perform the *forward* pass of the neural network, i.e. allowing to successively produce  $\hat{h}$ ,  $h$ ,  $\tilde{y}$ ,  $\hat{y}$ , starting at  $x$ .** Let  $W_i$  and  $b_i$  denote the parameters for layer  $i$ ,  $f_i(x) = xW_i^T + b_i$  represent the linear transformation, and  $g_i(x)$  be the activation function for layer  $i$ . Calculate the weighted sum and activation for the first hidden layer:

$$\tilde{h} = f_0(x)$$

$$h = g_0(\tilde{h})$$

Proceed to the output layer by computing the weighted sum and activation for the output layer:

$$\begin{aligned}\tilde{y} &= f_1(h) \\ \hat{y} &= g_1(\tilde{y})\end{aligned}$$

These equations describe the sequential steps involved in the forward pass of the neural network, ultimately producing the output  $\hat{y}$  based on the input  $x$ .

### 1.1.3 Loss function

**8. During training, we try to minimize the loss function. For cross entropy and squared error, how must the  $\hat{y}_i$  vary to decrease the global loss function  $\mathcal{L}$ ?** Our aim is to minimize the loss function  $\mathcal{L}$  to train a model effectively. To decrease cross-entropy loss, make  $\hat{y}_i$  closer to 1 when  $y_i$  is 1 and closer to 0 when  $y_i$  is 0.

For the cross-entropy loss,  $\hat{y}_i$  should vary in a way that makes it closer to the true target value  $y_i$  for each data point. Specifically, when  $y_i = 1$ ,  $\hat{y}_i$  should be pushed towards 1. The closer  $\hat{y}_i$  is to 1, the lower the loss. Conversely, when  $y_i = 0$ ,  $\hat{y}_i$  should be pushed towards 0. The closer  $\hat{y}_i$  is to 0, the lower the loss.

For squared error loss,  $\hat{y}_i$  should vary in a way that makes it closer to the true target value  $y_i$  for each data point. The goal is to minimize the squared difference between  $\hat{y}_i$  and  $y_i$ . This means that if  $\hat{y}_i$  is greater than  $y_i$ , it should decrease, and if  $\hat{y}_i$  is smaller than  $y_i$ , it should increase.

**9. How are these functions better suited to classification or regression tasks?** Cross-entropy loss is better suited for classification tasks for several reasons:

- Cross-entropy loss is based on the negative logarithm of the predicted probability of the true class. This logarithmic nature amplifies errors when the predicted probability deviates from 1 (for the correct class) and from 0 (for incorrect classes), depicted in Figure 1.1. Consequently, it effectively penalizes misclassifications, making it particularly suitable for classification tasks. However, it is imperative that  $\hat{y}$  remains within the range  $[0, 1]$  for this loss to be effective.
- Cross-entropy loss is often used in conjunction with the SoftMax activation function in the output layer of neural networks for multi-class classification. The SoftMax function ensures that the predicted probabilities sum to 1, aligning perfectly with the requirements of the cross-entropy loss.

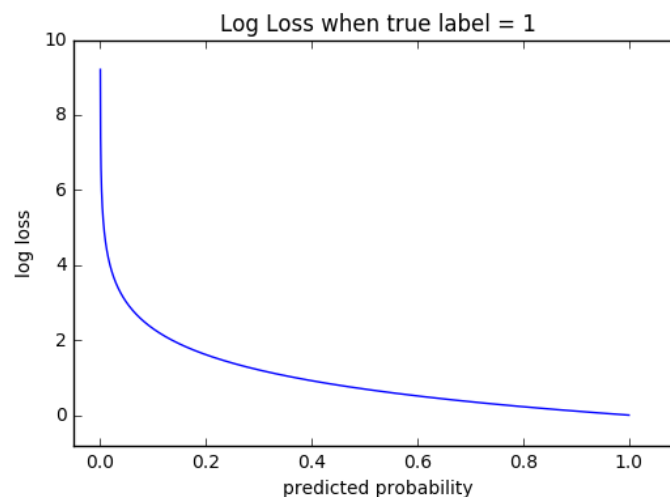


Figure 1.1

On the other hand, Mean Squared Error serves a different role and is better suited for regression tasks:

- MSE is ideal when dealing with  $(y, \hat{y}) \in \mathbb{R}^2$  (as opposed to the bounded interval  $[0, 1]$ ).
- MSE is advantageous due to its convexity, which simplifies optimization. However, it's important to note that it may not always be the best choice for every regression task, especially when dealing with outliers, as it can be sensitive to extreme values.

### 1.1.4 Optimization algorithm

**10. What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions? Which one seems the most reasonable to use in the general case?** Gradient computation becomes computationally expensive as it scales with the number of examples included. However, including more examples enhances gradient exploration with precision and stability, analogous to the exploration-exploitation trade-off in reinforcement learning.

Classic gradient descent offers stability, particularly with convex loss functions, featuring deterministic and reproducible updates. Nonetheless, its stability may lead to getting trapped in local minima when dealing with non-convex loss functions. Additionally, it is computationally intensive, especially for large datasets, as it demands evaluating the gradient with the entire dataset in each iteration.

Mini-Batch Stochastic Gradient Descent (SGD) converges faster compared to classic gradient descent by updating model parameters with small, random data subsets (mini-batches). This stochasticity aids in escaping local minima and exploring the loss landscape more efficiently. However, it may introduce noise and oscillations.

On the other hand, Online Stochastic Gradient Descent offers extremely rapid updates, processing one training example at a time, making it suitable for streaming or large-scale online learning scenarios. However, its highly noisy updates can result in erratic convergence or divergence, necessitating careful learning rate tuning.

For training deep neural networks in the general case, mini-batch stochastic gradient descent is the most reasonable choice. It strikes a balance between the computational efficiency of classic gradient descent and the noise resilience and convergence speed of online SGD.

**11. ★ What is the influence of the *learning rate*  $\eta$  on learning?** The learning rate plays a crucial role in influencing various aspects of the learning process, including convergence speed, stability, and the quality of the final model.

A higher learning rate typically results in faster convergence because it leads to larger updates in model parameters during each iteration. However, an excessively high learning rate can cause issues such as over-shooting, where the optimization process diverges or oscillates around the minimum, preventing successful convergence.

Conversely, a smaller learning rate allows the optimization algorithm to take smaller steps, which can be advantageous for exploring local minima more thoroughly and escaping shallow local minima. Nonetheless, if the learning rate is too small, it may lead to slow convergence or getting stuck in local minima.

To address these challenges, various techniques like learning rate schedules (gradually reducing the learning rate over time) or adaptive learning rate methods (e.g., Adam) have been developed to automate learning rate adjustments during training. The choice of learning rate may also be influenced by the batch size used in mini-batch stochastic gradient descent, as smaller batch sizes may require smaller learning rates to maintain stability.

In practice, selecting an appropriate learning rate often involves experimentation and can be problem-specific. Techniques such as grid search or random search, in combination with cross-validation, can aid in determining an optimal learning rate for a particular task.

**12. ★ Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the *loss* with respect to the parameters, using the naive approach and the *back-prop* algorithm.** The naive method involves calculating the gradient of the loss function with respect to each parameter independently, without leveraging the computational advantages of knowing the derivatives' interdependencies across layers. For a network with a large number of parameters, this is computationally expensive and inefficient, essentially repeating a substantial amount of calculation needlessly. The complexity becomes substantial as the number of layers (and, as a result, the number of parameters) increases. It becomes a combinatorial problem.

Backpropagation is a far more efficient strategy due to its intrinsic use of the chain rule from calculus to "remember" past calculations. This technique essentially breaks down the full problem of calculating a derivative on a complex, multi-stage function into smaller, more manageable pieces. Then, it smartly recombines these pieces, significantly reducing computational redundancy. This methodology dramatically reduces the number of computations, particularly for networks with a large number of interconnected layers and parameters.

**13. What criteria must the network architecture meet to allow such an optimization procedure?** For the backpropagation algorithm to be applicable, several criteria must be met. First of all, each layer function, including the activation functions and the loss function, must be differentiable with respect to their parameters.

This is crucial for calculating gradients during the training process. Furthermore, as backpropagation relies on the sequential flow of information from the input layer to the output layer, the network architecture should possess a feedforward, sequential structure without loops or recurrent connections. Loops or recurrent connections can introduce complications in gradient calculations.

**14. The function SoftMax and the *loss of cross-entropy* are often used together and their gradient is very simple. Show that the *loss* can be simplified by:**

$$l = - \sum_i y_i \tilde{y}_i + \log \left( \sum_i e^{\tilde{y}_i} \right)$$

Let us define the cross-entropy loss as  $l(y, \hat{y}) = - \sum_{i=1}^N y_i \log \hat{y}_i$ , where  $N$  represents the total number of examples. The SoftMax function for a vector  $\tilde{y} \in \mathbb{R}^D$  is defined as  $\text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^D e^{\tilde{y}_j}}$ . Notably, the output of the SoftMax function serves as the input for the cross-entropy loss, denoted as  $\hat{y}_i = \text{SoftMax}(\tilde{y}_i)$ . Let us substitute this value into the expression.

$$\begin{aligned} l(y, \hat{y}) &= - \sum_{i=1}^N y_i \log \hat{y}_i \\ &= - \sum_{i=1}^N y_i \log \text{SoftMax}(\tilde{y}_i) \\ &= - \sum_{i=1}^N y_i \log \frac{e^{\tilde{y}_i}}{\sum_{j=1}^D e^{\tilde{y}_j}} \\ &= - \sum_{i=1}^N y_i \left[ \log e^{\tilde{y}_i} - \log \sum_{j=1}^D e^{\tilde{y}_j} \right] \\ &= - \sum_{i=1}^N y_i \tilde{y}_i - y_i \log \sum_{j=1}^D e^{\tilde{y}_j} \\ &= - \sum_{i=1}^N y_i \tilde{y}_i + \sum_{i=1}^N y_i \log \left( \sum_j e^{\tilde{y}_j} \right) \\ &= - \sum_{i=1}^N y_i \tilde{y}_i + \log \left( \sum_j e^{\tilde{y}_j} \right) \sum_{i=1}^N y_i \end{aligned}$$

Since  $y_i$  is one-hot encoded, we know that  $\sum_i y_i = 1$ . Thus, we have the final expression for the cross-entropy loss:

$$l(y, \hat{y}) = - \sum_i y_i \tilde{y}_i + \log \left( \sum_i e^{\tilde{y}_i} \right)$$

**15. Write the gradient of the *loss (cross-entropy)* relative to the intermediate output  $\tilde{y}$**

$$\begin{aligned} \frac{\partial l}{\partial \tilde{y}_i} &= -y_i + \frac{\frac{\partial}{\partial \tilde{y}_i} (\sum_{j=1}^N e^{\tilde{y}_j})}{\sum_{j=1}^N e^{\tilde{y}_j}} \\ &= -y_i + \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \\ &= -y_i + \text{SoftMax}(\tilde{y})_i \\ \nabla_{\tilde{y}} l &= \begin{pmatrix} \frac{\partial l}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}} \end{pmatrix} = \begin{pmatrix} \text{SoftMax}(\tilde{y})_1 - y_1 \\ \vdots \\ \text{SoftMax}(\tilde{y})_{n_y} - y_{n_y} \end{pmatrix} = \hat{y} - y \end{aligned}$$

**16. Using the *backpropagation*, write the gradient of the *loss* with respect to the weights of the output layer  $\nabla_{W_y} l$ . Note that writing this gradient uses  $\nabla_{\tilde{y}} l$ . Do the same for  $\nabla_{b_y} l$ . Starting with  $\nabla_{W_y} l$ , we have:**

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$$

This can be expressed as a matrix:

$$\nabla_{W_y} l = \begin{pmatrix} \frac{\partial l}{\partial W_{y,1,1}} & \cdots & \frac{\partial l}{\partial W_{y,1,n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial W_{y,n_y,1}} & \cdots & \frac{\partial l}{\partial W_{y,n_y,n_h}} \end{pmatrix}$$

First, let's compute  $\tilde{y}_k$

$$\begin{aligned} \tilde{y} &= hW_y^T + b^y \\ &= (h_1 \quad \cdots \quad h_{n_h}) * \begin{pmatrix} W_{1,1} & \cdots & W_{1,n_y} \\ \vdots & \ddots & \vdots \\ W_{n_h,1} & \cdots & W_{n_h,n_y} \end{pmatrix} + (b_1^y \quad \cdots \quad b_{n_y}^y) \\ &= (\sum_{j=1}^{n_h} W_{1,j}^y h_j + b_1^y \quad \sum_{j=1}^{n_h} W_{2,j}^y h_j + b_2^y \quad \cdots \quad \sum_{j=1}^{n_h} W_{n_y,j}^y h_j + b_{n_y}^y) \in \mathbb{R}^{1 \times n_h} \\ \tilde{y}_k &= \sum_{j=1}^{n_h} W_{k,j}^y h_j + b_k^y, k \in [1, n_h] \end{aligned}$$

With this expression, we can now proceed with the calculation of the partial derivative of  $\tilde{y}_k$  with respect to  $W_{ij}^y$ :

$$\frac{\partial \tilde{y}_k}{\partial W_{ij}^y} = \begin{cases} h_j & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

Now, we need to find  $\frac{\partial l}{\partial \tilde{y}_k}$ . From the previous question, we have:

$$\frac{\partial l}{\partial \tilde{y}_k} = -y_k + \text{SoftMax}(\tilde{y})_k = \hat{y}_k - y_k$$

Now, combining these results:

$$\frac{\partial l}{\partial W_{i,j}^y} = \sum_{k=1}^{n_h} \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{i,j}^y} = \frac{\partial l}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial W_{i,j}^y} = (\hat{y}_i - y_i) h_j = (\nabla_{W_y} l)_{i,j}$$

So, the gradient of the loss with respect to the weights of the output layer  $\nabla_{W_y} l$  is given by  $\nabla_{\tilde{y}}^T h$ .

$$\begin{aligned} \nabla_{W_y} l &= \begin{pmatrix} \frac{\partial l}{\partial W_{1,1}^y} & \cdots & \frac{\partial l}{\partial W_{1,n_h}^y} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial W_{n_y,1}^y} & \cdots & \frac{\partial l}{\partial W_{n_y,n_h}^y} \end{pmatrix} \\ &= \begin{pmatrix} (\hat{y}_1 - y_1) h_1 & \cdots & (\hat{y}_1 - y_1) h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y}) h_1 & \cdots & (\hat{y}_{n_y} - y_{n_y}) h_{n_h} \end{pmatrix} \\ &= \begin{pmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{pmatrix} (h_1 \quad h_2 \quad \cdots \quad h_{n_h}) \\ &= \nabla_{\tilde{y}}^T h \end{aligned}$$

### 17. ★ Compute other gradients : $\nabla_{\tilde{h}} l, \nabla_{W^h} l, \nabla_{b^h} l$

1. The gradient of the loss with respect to  $\tilde{h}$  can be computed using the chain rule:

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i}$$

Let's compute those two terms:

$$\frac{\partial h_k}{\partial \tilde{h}_i} = \frac{\partial \tanh(\tilde{h}_k)}{\partial \tilde{h}_i} = \begin{cases} 1 - \tanh^2(\tilde{h}_i) = 1 - h_i^2 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

Having  $\tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y$  in mind and recovering  $\frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i = \delta_i^y$  from past question:

$$\frac{\partial l}{\partial h_k} = \sum_{j=1} \frac{\partial l}{\partial \tilde{y}_j} \frac{\partial \tilde{y}_j}{\partial h_k} = \sum_{j=1} \delta_j^y W_{j,k}^y$$

Finally,

$$\begin{aligned} \frac{\partial l}{\partial \tilde{h}_i} &= \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} \\ &= \sum_k \sum_j \delta_j^y W_{j,k}^y \times \begin{cases} 1 - h_i^2 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} \\ &= (1 - h_i^2) \left( \sum_j \delta_j^y W_{j,i}^y \right) \\ &= \delta_i^h \end{aligned}$$

So, the gradient  $\nabla_{\tilde{h}} l$  is a vector with elements  $\delta_i^h$ :

$$\nabla_{\tilde{h}} l = (1 - h^2) \odot (\nabla_{\tilde{y}} l * W^y)$$

2.  $\nabla_{W^h} l$  is a matrix composed of elements

$$\frac{\partial l}{\partial W_{i,j}^h} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{i,j}^h}$$

From the previous question, we already have  $\frac{\partial l}{\partial h_k} = (1 - h_k^2) (\sum_j \delta_j^y W_{j,k}^y) = \delta_k^h$ . So let's compute the other term  $\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h}$  from  $\tilde{h}_k = \sum_{j=1}^{n_x} W_{k,j}^h x_j + b_k^h$

$$\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h} = \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

So:

$$\begin{aligned} \frac{\partial l}{\partial W_{i,j}^h} &= \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{i,j}^h} \\ &= \sum_k \delta_k^h \times \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} \\ &= \delta_i^h x_j \\ \nabla_W^h &= \nabla_{\tilde{h}}^T l * x \end{aligned}$$

3. Last but not least:

$$\frac{\partial l}{\partial b_i^h} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_i^h} = \sum_k \delta_k^h * \begin{cases} 1 & \text{if } k = i \\ 0 & \text{else} \end{cases} = \delta_i^h$$

$$\nabla_{b^h} = \nabla_{\tilde{h}} l$$



## 1.2 Implementation

In this section, the primary focus was on exploring how a simple neural network model can perform at classifying tasks which can be difficult with normal machine learning techniques. The context varied from circular data to image classification, and the overall approach was to confirm the theory of precedent section and experiment with different hyperparameters to understand their effects on model performance.

### 1.2.1 Forward and backward manuals

In neural network training using SGD, it is common practice to average the gradients calculated during the backward pass over the batch size. The idea behind this scaling is to make the learning rate independent of the batch size, so that the updates to the model parameters are roughly consistent regardless of the batch size. We compared these two approaches in Figure 1.2.

One significant observation is that when using batch normalization, the network requires much more time to achieve its performance, needing 100 epochs instead of 10. However, the trade-off is that the training process becomes much more stable.

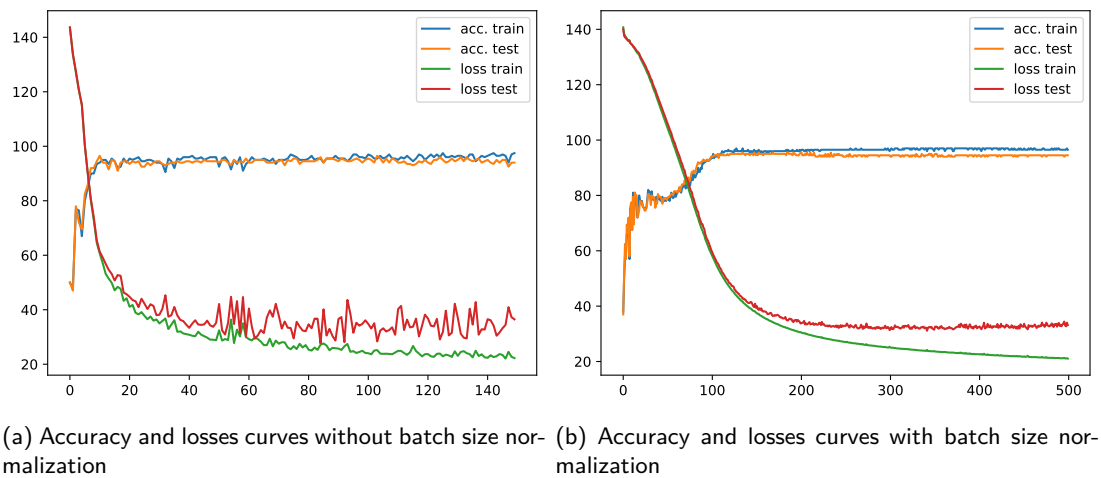


Figure 1.2: To batch normalize, or not to batch normalize.

In the following experiments, we have chosen to continue with batch normalization. We then proceeded to investigate the influence of learning rate and batch size. The results are presented in Figure 1.3. Our findings align with the expectations discussed in Question 11.

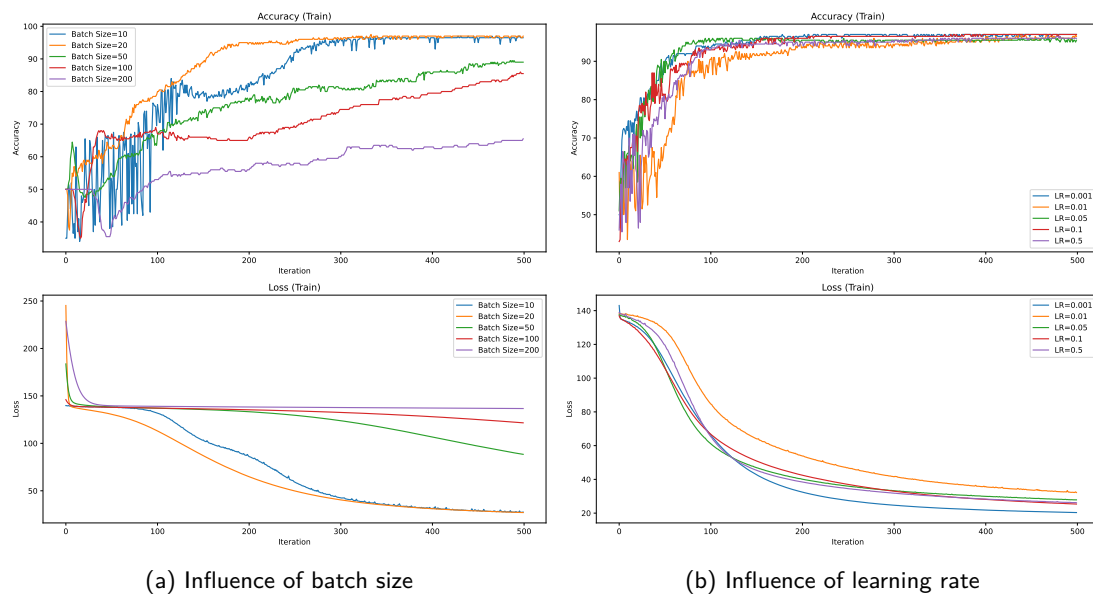
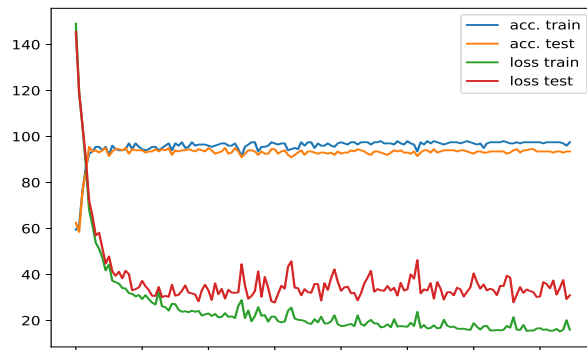


Figure 1.3: Influence of learning rate and batch size

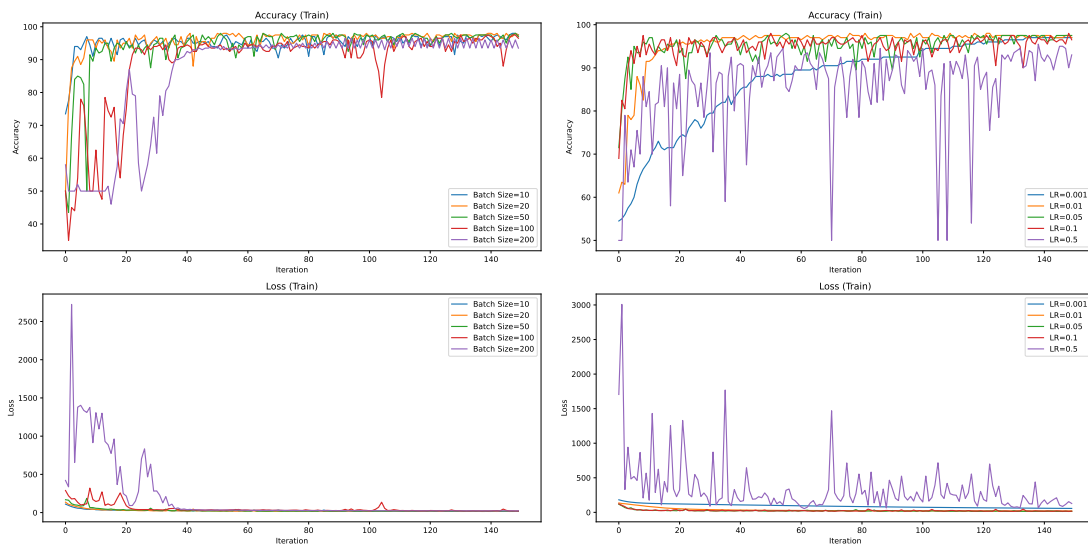
In the provided notebook, you will also find animated figures that showcase the evolution of the learning process and the visualization of the decision boundaries. Unexpectedly, by the end of the training, we can observe circular decision boundaries for the circular data as the model is able to classify the data correctly.

### 1.2.2 Simplification of the backward pass with `torch.autograd`

The results of this section are presented in Figure 1.4. It's worth noting that we obtained similar results to when we did not use batch normalization, as indicated by the learning instability. However, our overall conclusions remain consistent.



(a) Accuracy and losses curves with `torch.autograd`



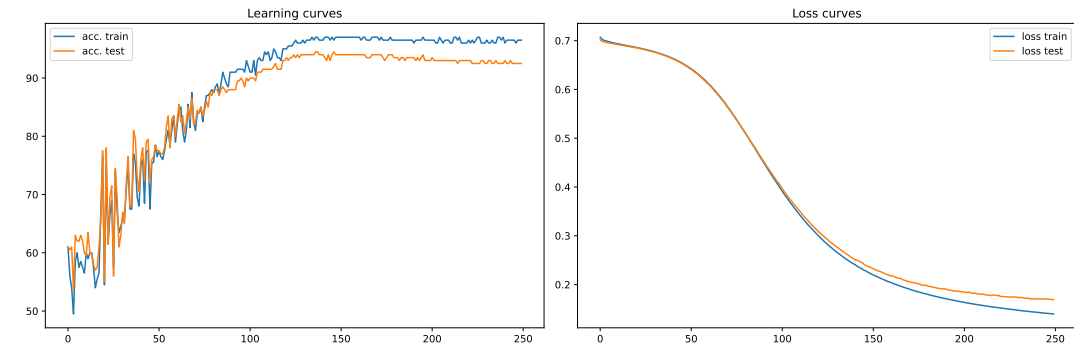
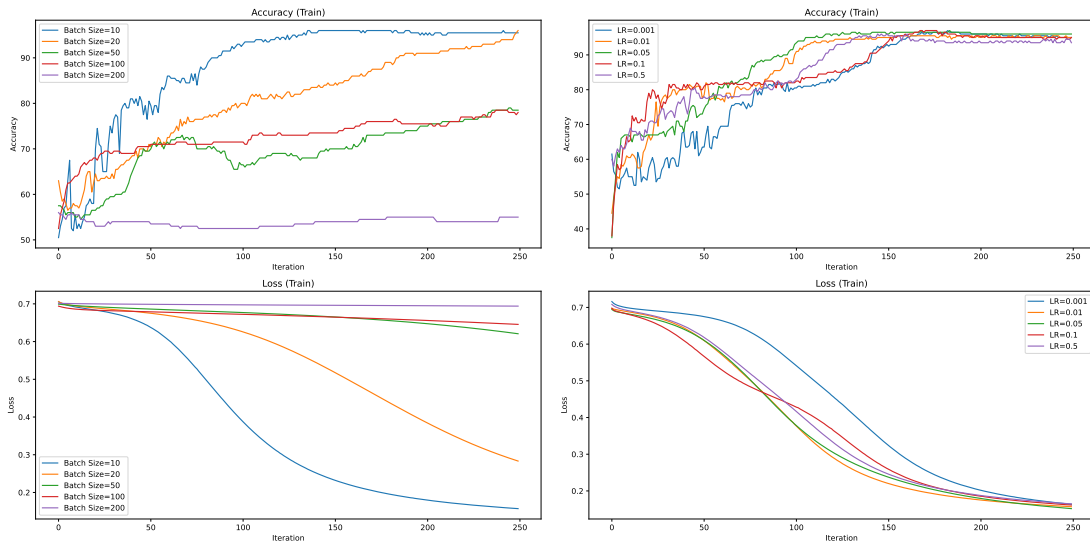
(b) Influence of batch size

(c) Influence of learning rate

Figure 1.4: `torch.autograd`

### 1.2.3 Simplification of the forward pass with `torch.nn` layers

The results of this section can be found in Figure 1.5. In this case, we observe that learning takes longer compared to using manual layers. It requires 150 epochs to reach its maximum performance. This could be attributed to how Torch initializes its layers. However, the training process appears to be more stable, as evidenced by the loss curve.

(a) Accuracy and losses curves using `torch.nn` layers

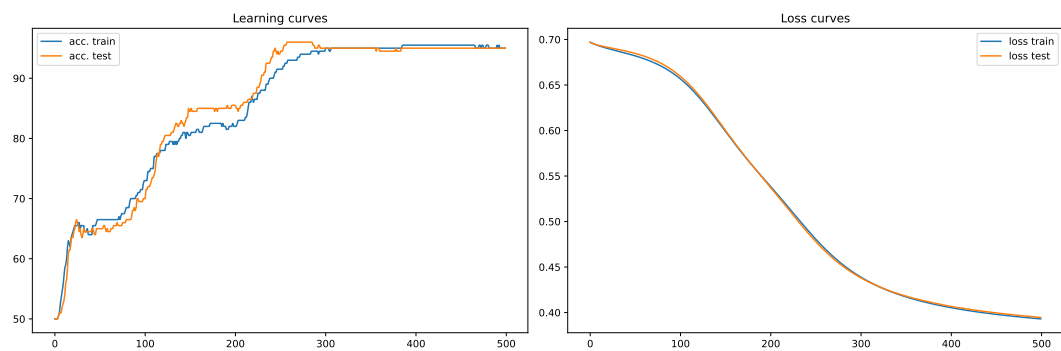
(b) Influence of batch size

(c) Influence of learning rate

Figure 1.5: `torch.nn`

### 1.2.4 Simplification of the SGD with `torch.optim`

The results of this section are presented in Figure 1.6. The training process is significantly more stable, but it requires a much longer time to converge, taking 300 epochs to reach its maximum performance.

(a) Accuracy and losses curves using `torch.optim`

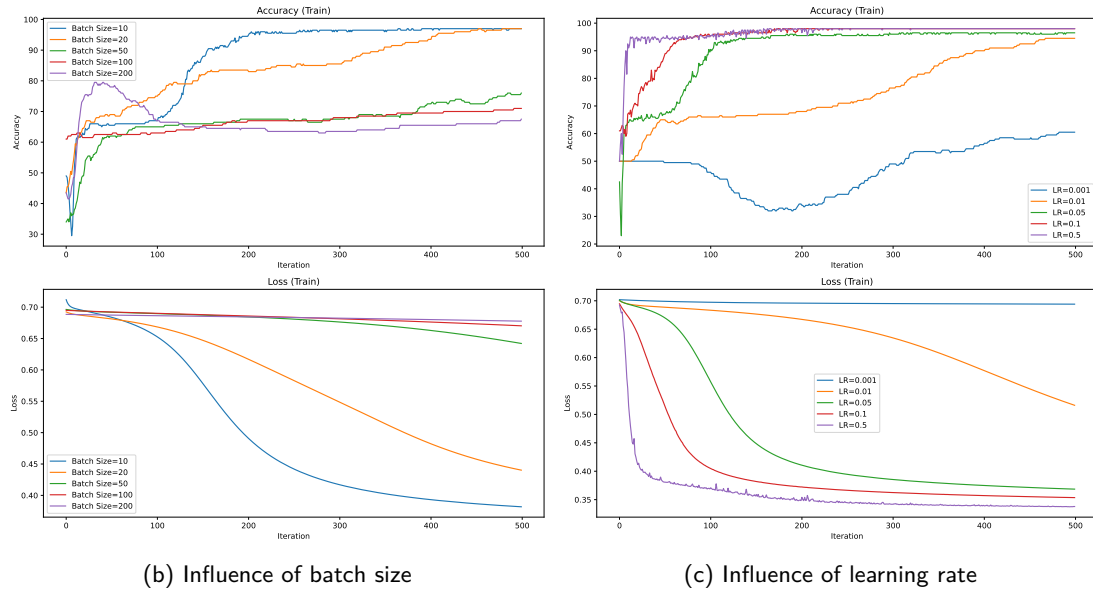


Figure 1.6: torch.optim

## 1.2.5 MNIST application

The following section presents the results of our model on the MNIST dataset, as shown in Figure 1.7. Despite its simplicity, the model performs remarkably well. However, there is some instability, which can likely be addressed with a learning rate scheduler or by trying a different optimizer.

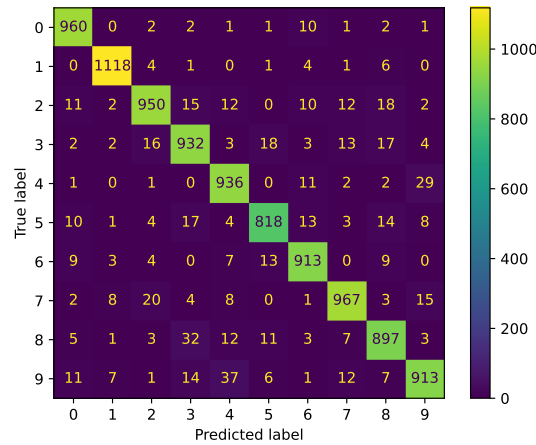
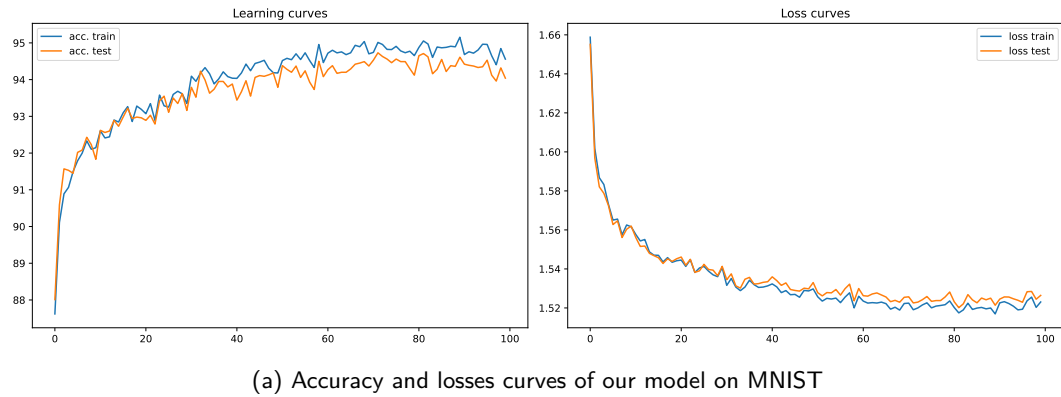


Figure 1.7: Performance on MNIST

### 1.2.6 SVM

We also explored the SVM approach on our circular data, and the decision boundaries, along with their corresponding accuracies, are presented in Figure 1.8.

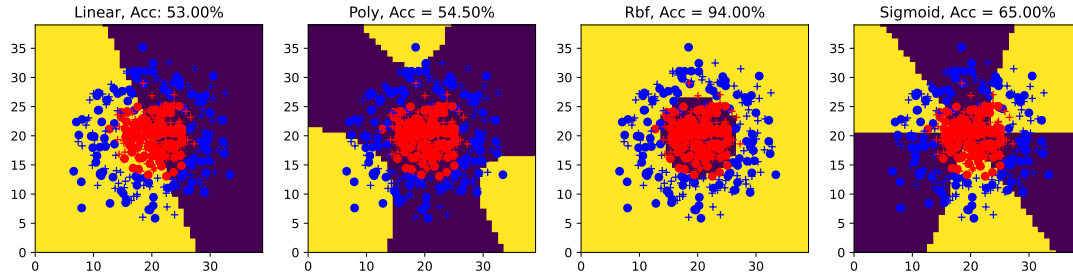
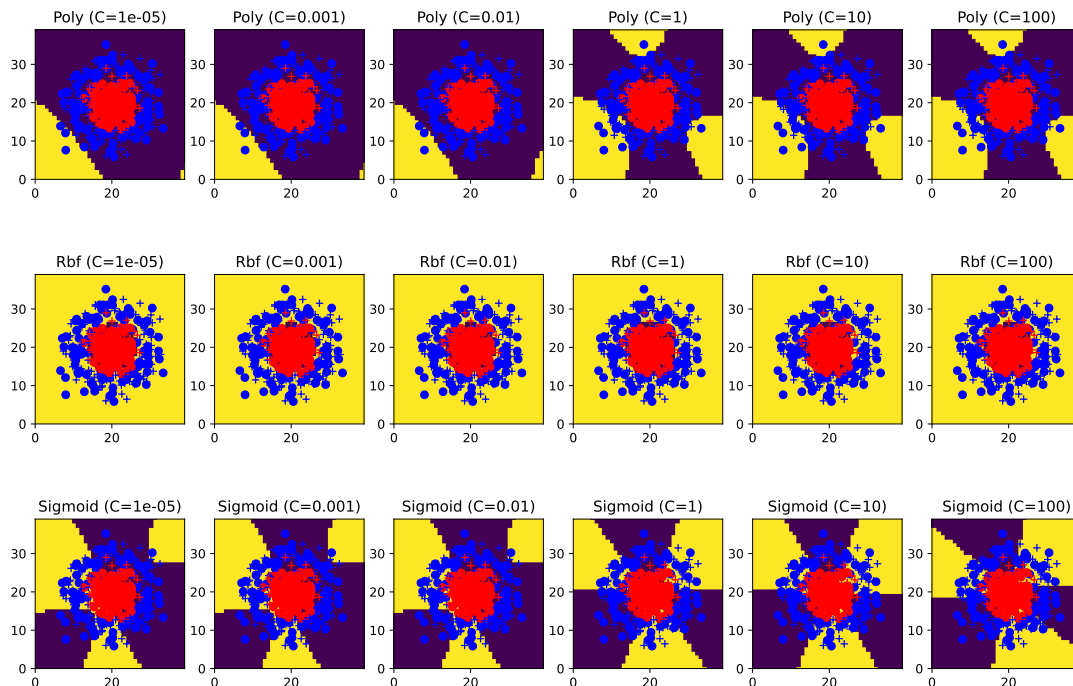


Figure 1.8: Comparison of SVM

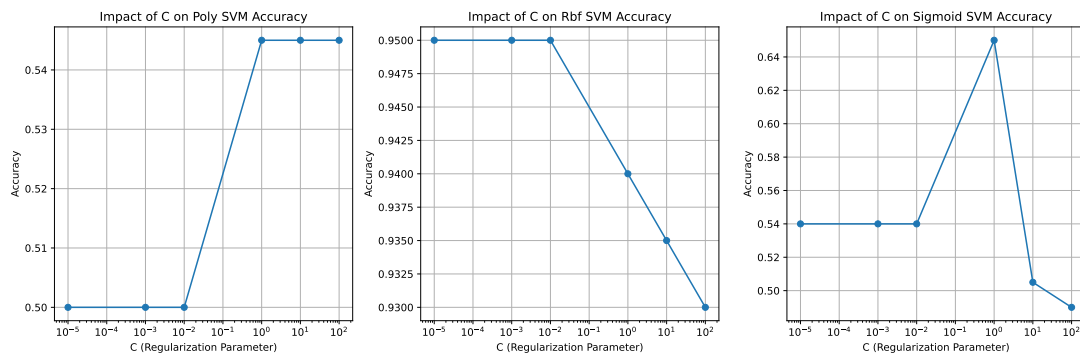
It's evident that a linear SVM doesn't work well with circular data. Linear SVMs are designed to find a linear decision boundary that separates data into two classes. Circular data cannot be accurately separated by a single straight line, which is our case here. The Radial Basis Function (RBF) kernel is often the best choice compared to polynomial and sigmoid kernels for circular data because it can capture complex, nonlinear patterns more effectively, and it is better suited to model the circular decision boundaries typically found in such data. RBF kernels can flexibly adapt to various shapes and are more versatile in representing circular patterns, as shown here. It's worth noting that the linear and polynomial kernels did not perform significantly better than a random function.

Furthermore, we conducted experimentations into the influence of the regularization parameter  $C$  in the SVM. The results are visualized in Figure 1.9. The parameter  $C$  in SVMs controls the trade-off between maximizing the margin and minimizing the classification error on the training data.

- When  $C$  is large, the SVM aims to minimize the classification error on the training data, which results in a smaller margin, meaning that the decision boundary can be more flexible and may even fit noisy data points. This can lead to overfitting. This effect can especially be seen on the polynomial SVM.
- When  $C$  is small, the SVM prioritizes maximizing the margin, even if it means allowing some training points to be misclassified. It results in a larger margin, which often leads to a simpler decision boundary. It encourages the SVM to find a more robust decision boundary that generalizes better to unseen data.



(a) Decision boundaries



(b) Impact of  $C$  on accuracy

Figure 1.9: Influence of regularization parameter

## Chapter 2

# Introduction to convolutional networks

### 2.1 Questions

**1. Considering a single convolution filter of padding  $p$ , stride  $s$  and kernel size  $k$ , for an input of size  $x \times y \times z$  what will be the output size? How much weight is there to learn? How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size?** Let's note  $x_{\text{out}}, y_{\text{out}}, z_{\text{out}}$  our output size. Then, given a convolution filter of padding  $p$ , stride  $s$  and kernel size  $k$ :

$$x_{\text{out}} = \left\lfloor \frac{x - k + 2p}{s} + 1 \right\rfloor \quad (2.1)$$

$$y_{\text{out}} = \left\lfloor \frac{y - k + 2p}{s} + 1 \right\rfloor \quad (2.2)$$

$$z_{\text{out}} = \text{number of filters} = 1 \quad (2.3)$$

Given our single convolution filter and our kernel of size  $k \times k$ , we thus have  $((z \times k \times k) + 1) \times z_{\text{out}} = ((z \times k \times k) + 1)$  weights to learn (where "+1" represents the bias).

If a fully connected layer were to produce an output with dimensions equivalent to the output of the convolutional layer, it would need to connect every input neuron to every output neuron. For the input layer, every input pixel is a neuron, so there are  $x \times y \times z$  neurons. The total number of neurons in the output layer is  $x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}}$ . Thus, considering a bias, there is  $(x \times y \times z + 1) \times (x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}}) = (x \times y \times z + 1) \times (x_{\text{out}} \times y_{\text{out}})$  parameters to learn.

**2. ★ What are the advantages of convolution over fully-connected layers? What is its main limit?**

Using fully-connected layers with images involves a substantial number of parameters, one for each pixel of the image. Fully-connected layers are also sensitive to variations in images, including simple translations.

On the other hand, convolutional layers address these issues. By utilizing convolution with a shared set of weights (for filters/kernels) across the entire input, they enable the learning and detection of local spatial patterns and features at various locations. Convolutional layers also possess the ability to hierarchically combine features. Lower layers capture low-level features such as edges and textures, while higher layers capture more intricate features and representations of objects.

However, convolutional layers have limitations in terms of global context. They primarily focus on local patterns and may struggle to comprehend relationships between distant parts of the input, which can be crucial in certain applications (medical imaging, autonomous vehicles...).

**3. ★ Why do we use spatial pooling?** Spatial pooling allows for increased invariance, particularly in the context of translation. By summarizing a local region with a pooled value, the precise feature location becomes less critical. This constitutes a form of regularization that prioritizes the most pertinent information.

Pooling also facilitates dimensionality reduction by decreasing the spatial dimensions of the resulting feature map. This reduction in dimensionality significantly lowers the computational cost of subsequent layers.

Max and average pooling layers are the most commonly employed pooling techniques. Max pooling aids in achieving translation invariance and reducing spatial dimensions. It effectively preserves the most essential features in the input data. On the other hand, average pooling also reduces spatial dimensions and contributes to achieving translation invariance. It can exhibit greater robustness to outliers in the input data when compared to max pooling.

**4. ★ Suppose we try to compute the output of a classical convolutional network for an input image larger than the initially planned size. Can we (without modifying the image) use all or part of the layers of the network on this image?** A convolution layer in a convolutional network is size-agnostic regarding its input: it simply performs convolutions across the entire span of the input image and outputs a correspondingly altered image. Sequentially, when convolution and pooling layers are stacked together, the resulting output size is purely a function of the input size. To rephrase, the input size of the image is not a critical hyperparameter for the convolution layer.

Within the conventional architecture of a convolutional network, the output derived from successive convolutional and pooling layers is typically "flattened" to form the input for a subsequent fully-connected layer. The size of this flattened vector is contingent on the size of the image output from the preceding convolution layers.

The challenge arises when we involve fully connected layers, as their initialization requires a priori knowledge of their input size to properly set up parameters such as weights. Consequently, when an input image is larger than initially planned, the network can operate normally up to the point of the fully connected layers, which anticipate input of a predetermined, fixed size.

**5. Show that we can analyze fully-connected layers as particular convolutions.** In a fully-connected layer, each pixel of the flattened image is associated with one weight. Both the weights and the flattened image are vectors. However, if we rearrange the weights into a matrix that matches the shape of the image, we can perform element-wise multiplication between this matrix and the image, which is akin to a one slide convolution!

This  $1 \times 1$  convolution employs a stride equal to the width of the input image, ensuring that the filter remains fixed and does not slide to different positions. Furthermore, the filter size is exactly the same as the image. The number of filters used in this convolution is equivalent to the number of neurons in the fully-connected layer.

**6. Suppose that we therefore replace fully-connected by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest?** Fully connected layers usually expect inputs of a specific size. However, if we replace the fully connected layers in a neural network with their convolutional equivalents, we can indeed process images of sizes different from what the network was initially trained on. If an input image larger than the expected size is used, the convolution operations will produce larger feature maps in the intermediate layers, and these will propagate through the network. The final convolutional layers, which replace the fully connected layers, will apply their filters to the full spatial extent of the incoming feature maps. The output shape, in this case, won't be the typical single row of class scores. Instead, it will be a spatial map of scores, with dimensions depending on the input size, the characteristics of the convolutional layers, and the operations applied during the forward pass.

**7. We call the receptive field of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers? Can you imagine what happens to the deeper layers? How to interpret it?** The receptive field  $l_k$  of layer  $k$  is:

$$l_k = l_{k-1} + \left( (f_k - 1) \times \prod_{i=1}^{k-1} s_i \right)$$

where  $l_{k-1}$  is the receptive field of layer  $k - 1$ ,  $f_k$  is the filter size (height or width, but assuming they are the same here), and  $s_i$  is the stride of layer  $i$ .

The formula above calculates receptive field from bottom up (from layer 1). Intuitively, a receptive field in layer  $k$  covers  $(f_k - 1) \times s_{k-1}$  more pixels relative with layer  $k - 1$ . However, the increment needs to be translated to the first layer, so the increments is a factorial: a stride in layer  $k - 1$  is exponentially more strides in the lower layers. In our case, the receptive field of the first convolutional layer is of size 5, which matches the size of the kernel. Consequently, the size of the receptive field of the second layer increases to  $5 + (5 - 1) = 9$ . This means that each pixel in the output of the second layer takes into account information from 9 pixels in the previous layer. As the neural network progresses into deeper layers, the receptive field continues to grow. When done correctly, the receptive field size of the final layer will match the size of the input image, allowing each pixel in the output to gather information from every pixel in the input image. One can think of the receptive fields as the "eyes" of the neural network.



## 2.2 Training *from scratch* of the model

### 2.2.1 Network architecture

**8. For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed?** We want  $x = x_{out}$ ,  $y = y_{out}$ : let's use our equations 2.1, 2.2 to find the values of  $p$  and  $s$ .

$$p = \frac{(s-1)x + k - s}{2}$$

Knowing  $k = 5$  and that our images are of size  $32 \times 32$ , we can pick any  $p = \frac{31s-27}{2}$ . Since we typically aim to utilize all available pixels in an image while minimizing the need for padding, we choose a stride value of  $s = 1$  and a padding value of  $p = 2$ , which minimize both parameters.

**9. For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed?** A pooling layer acts like a convolution: let's proceed as in question 8. This time we want  $x_{out} = x/2$ ,  $y_{out} = y/2$ .

$$p = \frac{(\frac{1}{2}s - 1)x + k - s}{2}$$

Knowing  $k = 2$  and that our images are of size  $32 \times 32$ , we can pick any  $p = \frac{15s-30}{2}$ . Following the same logic as in the previous question, we opt for a stride value of  $s = 2$  and a padding value of  $p = 0$ , which minimize both parameters.

**10. ★ For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.** To simplify the presentation, we have provided a plot of the CNN architecture in Figure 2.1. In this architecture, we begin with a 3-channel image measuring 32 by 32 pixels. As mentioned in questions 8 and 9, convolutional layers maintain the same spatial dimensions in the outputs as in the inputs, while pooling layers reduce the spatial dimensions by a factor of two. The following is a breakdown of the number of parameters:

- conv1: 32 filters of 5 by 5 for 3 channels  $\rightarrow 32 \times 5 \times 5 \times 3 = 2,400$  parameters
- conv2: 64 filters of 5 by 5 for 32 channels  $\rightarrow 64 \times 5 \times 5 \times 32 = 51,200$  parameters
- conv3: 64 filters of 5 by 5 for 64 channels  $\rightarrow 64 \times 5 \times 5 \times 64 = 102,400$  parameters
- 1000 neurons with 1024 values as input  $\rightarrow (1000 + 1) \times 1024 = 1,025,024$  parameters. The "+1" is for biases.
- 10 neurons with 1000 values as input  $\rightarrow 11 \times 1000 = 11,000$  parameters

The fully connected layer fc4 alone accounts for up to 86 % of the total number of parameters, underscoring the significant computational demands associated with these layers during the learning process.

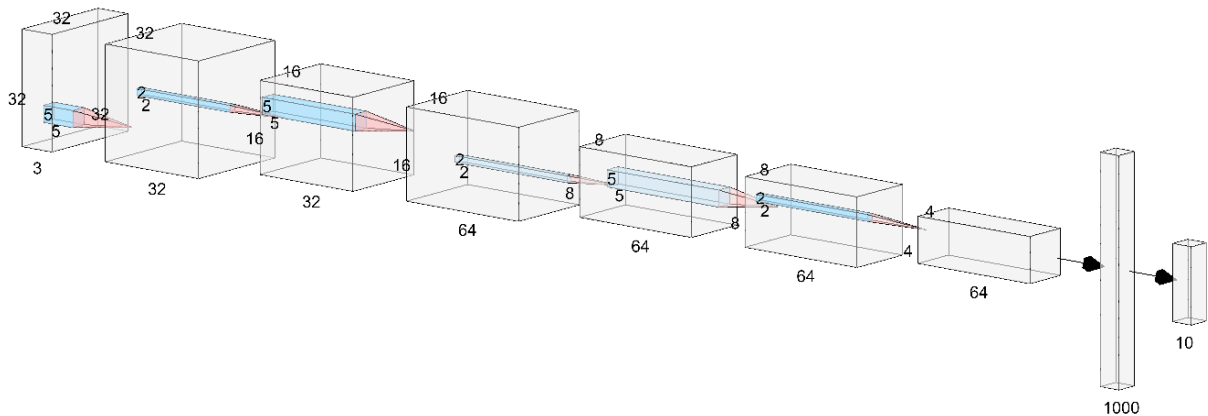


Figure 2.1: Network architecture

**11. What is the total number of weights to learn? Compare that to the number of examples.** This leads to a total of 1,192,024 parameters for training. Considering a dataset comprising 50,000 training samples and 10,000 test samples, totaling 60,000 examples (with 6,000 images per class), it does warrant caution regarding the potential for overfitting.

**12. Compare the number of parameters to learn with that of the BoW and SVM approach.** The complexity and, hence, the number of parameters in a BoW and SVM approach depend on the chosen visual vocabulary size. For instance, employing a dictionary of 1000 visual words for the task of classifying 10 classes would result in  $(1000+1) \times 10 = 10,010$  parameters to be learned. It's important to note that in this approach, we also need to fine-tune the hyperparameters of our classifier, with the kernel type and the regularization parameter  $C$  being the most significant. As a result, the parameter space in this approach is considerably smaller when compared to our deep CNN model.

## 2.2.2 Network learning

**14. ★ In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the the difference in data)?** The difference in the code lies in the fact that we do not provide an optimizer to the `epoch()` function. Consequently, we do not train the network using test examples, and there is no backward pass to calculate and update the loss.

**16. ★ What are the effects of the learning rate and of the batch-size?** The learning rate and batch size are critical hyperparameters that significantly impact training performance, speed, and stability:

- A high learning rate can cause the model to overshoot the minimum, causing divergent behavior and an inability for the model to learn effectively. On the other hand, having a low learning rate can slow down the training process significantly, possibly leading to a premature convergence to suboptimal solutions. In some cases, a lower learning rate results in a more precise convergence by allowing the model to explore the parameter space thoroughly. Figure 2.2 illustrates the effects of different learning rates with a fixed batch size of 32.
- Larger batch size provides a more accurate estimate of the gradient, potentially leading to more stable and consistent training steps. However, this comes with increased memory usage. Conversely, smaller batch sizes can introduce noise, making the path to convergence less direct and possibly longer in terms of epochs. Smaller batches reduce memory requirements, making them suitable for resource-constrained systems. Figure 2.3 illustrates the impact of various batch sizes with a fixed learning rate of 0.01.

It's important to note that learning rate and batch size are often interrelated. Changes in batch size may require adjusting the learning rate to maintain training quality. This relationship arises from the need to balance the aggressiveness of updates when averaging gradients across more samples. Typically, a linear scaling rule is applied: multiplying the batch size by a factor  $k$  also multiplies the learning rate by  $k$ , while keeping other hyperparameters constant. This relationship is shown in Figure 2.4.

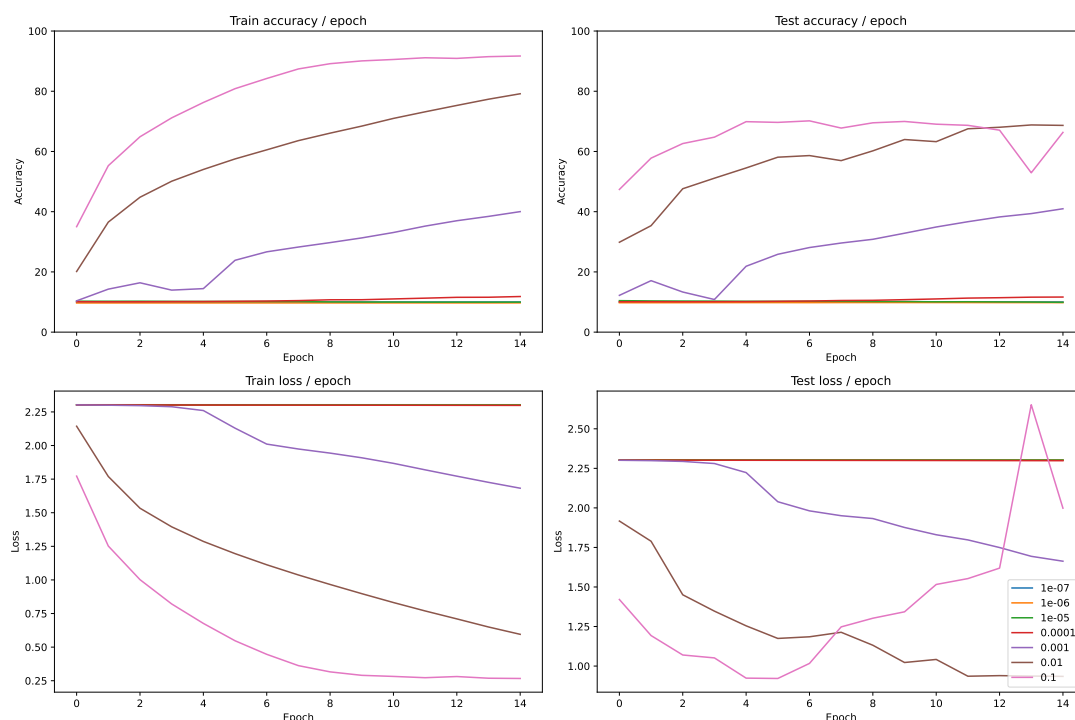


Figure 2.2: Influence of learning rate with a batch size fixed at 32

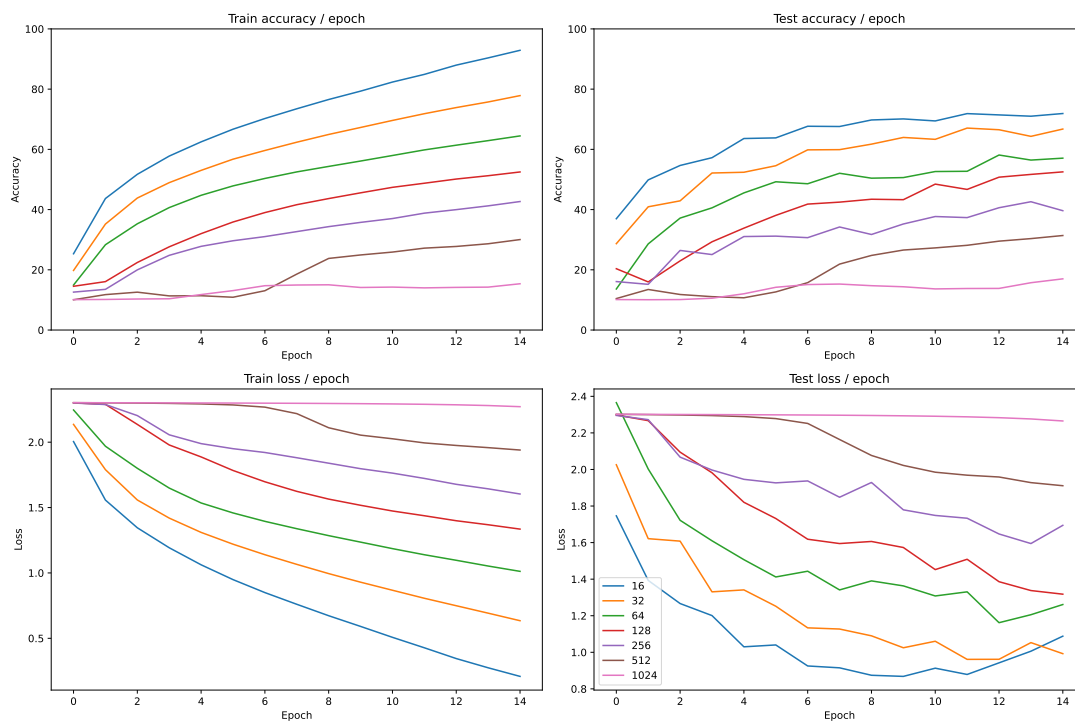


Figure 2.3: Influence of batch size rate with a learnig rate fixed at 0.01

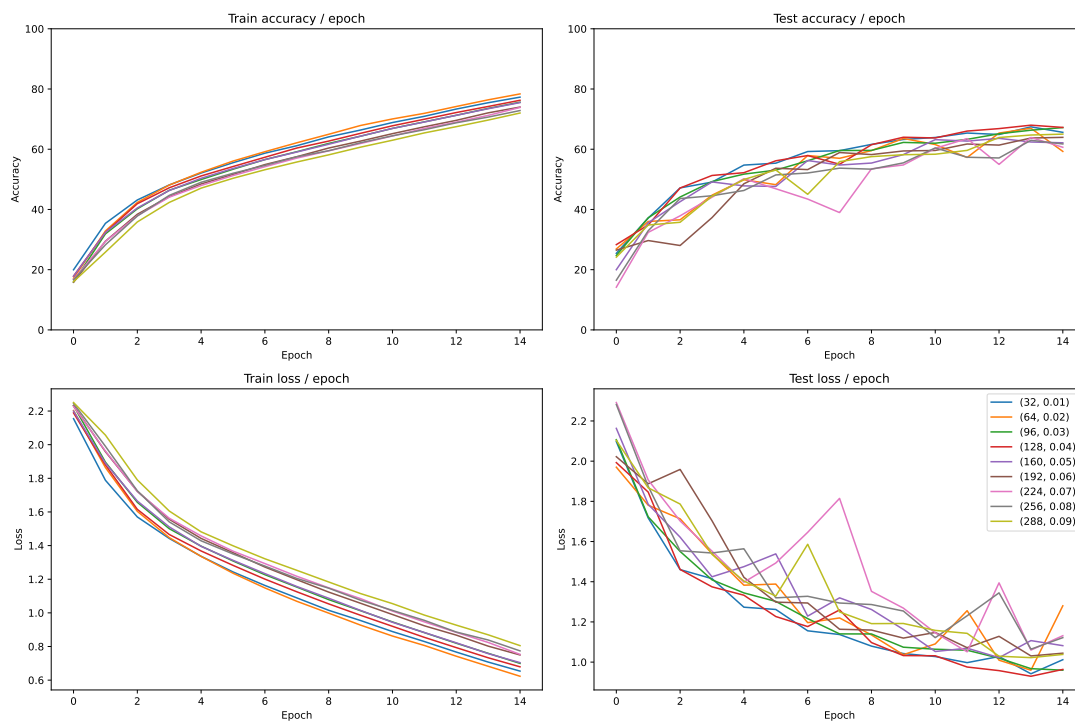


Figure 2.4: Relationship between batch size and learning rate

**17. What is the error at the start of the first epoch, in train and test? How can you interpret this?**  
 For the following experiments, we will compare our results with those presented in Figure 2.5, where a batch size of 128 and a learning rate of 0.01 were employed.

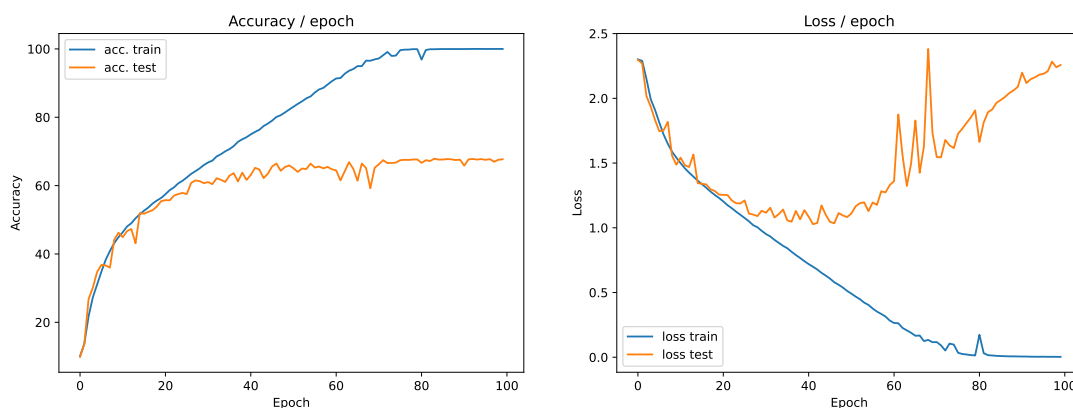


Figure 2.5: Accuracy and losses in train and test

The situation where test accuracy surpasses training accuracy at the beginning can be attributed to the fact that training loss and accuracy are calculated at each batch, implying that parameters and, consequently, results (loss and accuracy) fluctuate considerably before the completion of the first epoch. This means that during the first batch of the first epoch, the loss is computed based on only a subset of the data equal to the batch size. In contrast, test loss and accuracy are determined at the end of the first epoch, after the model has been exposed to the entire training dataset. These values are more reliable than those obtained from the training phase at the outset of the first epoch.

**18. ★ Interpret the results. What's wrong? What is this phenomenon?** We observe overfitting, as our caution led us to anticipate. This phenomenon becomes evident around the thirtieth epoch when our test accuracy plateaus at approximately 60 %, while the training accuracy continues to improve. This signifies that while the model is improving its performance on the training dataset, it is struggling to generalize effectively to new, unseen test images.

## 2.3 Results improvements

### 2.3.1 Standardization of examples

**19. Describe your experimental results.** Standardization helps mitigate the issues caused by the varying scales and ranges in raw data. By doing this, we can expect improved model training, consistent data distribution and reduced numerical instability. In our experimental results, we do find that standardizing accelerates model training, as we need 20 less epochs to reach convergence in train. Furthermore, we observe increased stability, both in loss and accuracy.

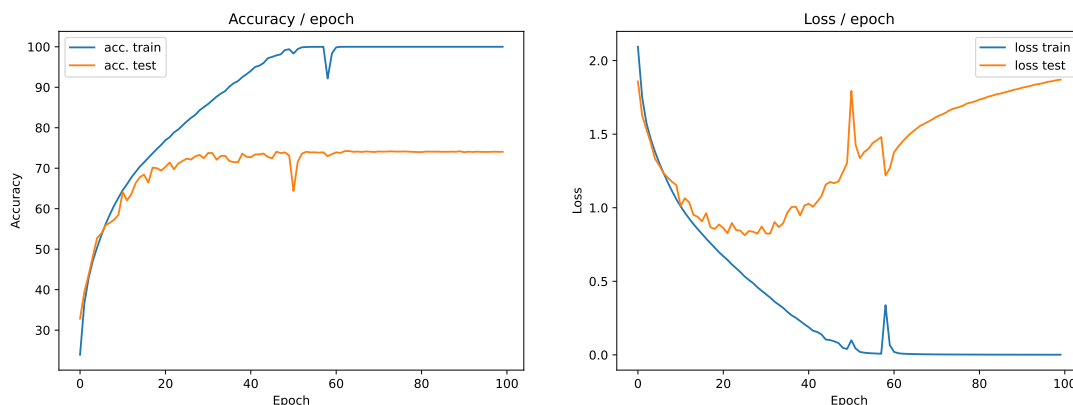


Figure 2.6: Accuracy and losses in train and test using standardization

**20. Why only calculate the average image on the training examples and normalize the validation examples with the same image?** This practice guarantees that the model evaluates its performance on data processed similarly to the training data, without any prior knowledge of the validation set's specific characteristics. Calculating these statistics using the validation set would result in "data leakage", where the

model gains insights into the validation data. This approach is essential for an impartial evaluation of the model's performance and its ability to generalize to new, unseen data.

**21. Bonus: There are other normalization schemes that can be more efficient like ZCA normalization. Try other methods, explain the differences and compare them to the one requested.** A few common normalization techniques are :

- **Global Contrast Normalization:** this technique involves normalization of images by subtracting the mean of the entire image (not per channel) and dividing by the standard deviation of the entire image. GCN is aimed at reducing the effect of lighting variations by flattening the image illumination and enhancing contrast.
- **PCA/ZCA Whitening:** these are linear algebra techniques that aims to decorrelate features. This is beneficial because decorrelated features can improve the learning efficiency of some models. ZCA is a variant of PCA that maintains the original data representation whereas PCA completely destroys it. However, these techniques can be computationally intensive.
- **L1/L2 Normalization:** this technique normalizes the pixels of the image by dividing each pixel value by the L1 (sum of absolute pixel values) or L2 norm (square root of the sum of the squared pixel values) of the image. This normalization is useful when we want to measure the relative importance of pixel values rather than their absolute magnitudes.
- **Min-Max Scaling:** this technique involves rescaling the range of pixel intensity values. Each pixel value is scaled based on the minimum and maximum pixel values found in the dataset. This is a simple scaling technique but does not handle outliers well.

We performed a comparison of these techniques, and the results are displayed in Figure 2.7. It is worth noting that PCA proved to be computationally demanding, doubling the required training time. L1 standardization exhibited poor performance, which we suspect may be attributed to the normalization process involving the division of each pixel value by the sum of absolute pixel values, resulting in excessively low values. L2 normalization performed better than its counterpart during training but did not generalize effectively during testing. In contrast, per-channel standardization consistently ranks among the top-performing methods, demonstrating superior performance. It's worth noting that there is little to no discernible difference in performance between per-channel standardization and its global counterpart, GCN.

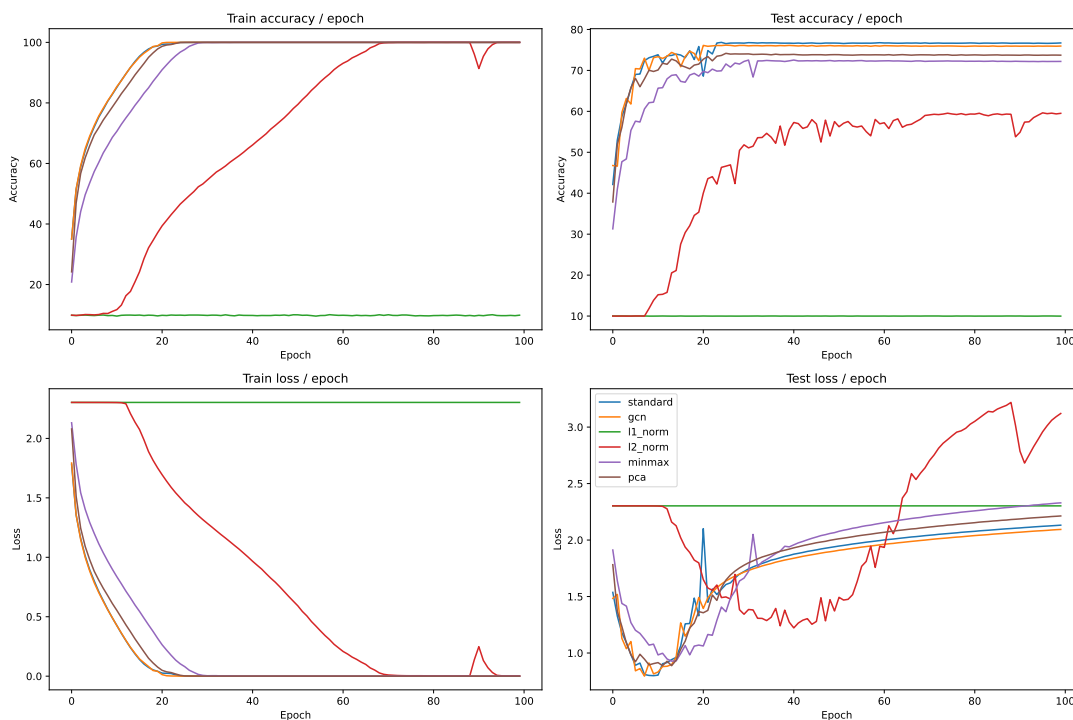


Figure 2.7: Influence of standardization techniques

### 2.3.2 Increase in the number of training examples by *data increase*

**22. Describe your experimental results and compare them to previous results.** By artificially increasing our dataset, we expect to have a model that generalize better, as it becomes less sensitive to the specificities of the training images, due to the introduction of variability. Moreover, random crops and flips introduce a level of perturbation to the data, helping the model to become more robust to variations it might encounter in real-world scenarios.

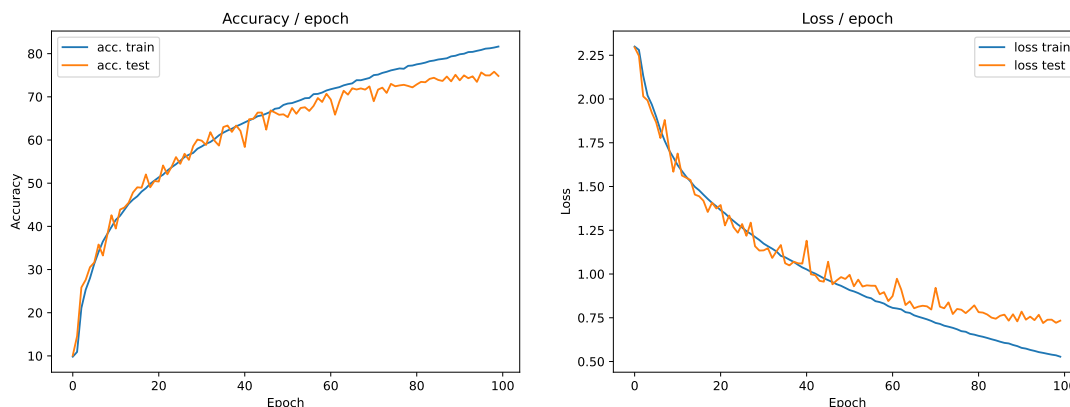


Figure 2.8: Accuracy and losses in train and test using data augmentation

Our experimental results demonstrate that our model exhibits improved generalization and reduced overfitting, which is a positive outcome. However, it's important to note that we still observe some degree of instability in testing.

**23. Does this horizontal symmetry approach seems usable on all types of images? In what cases can it be or not be?** For many general tasks, especially those involving natural scenes or objects, horizontal flipping does not change the essence of the subject. For instance, a car is still recognizable as a car, whether flipped or not. This technique is effective in such contexts, contributing to model robustness without misleading the training process. However, in tasks where orientation is crucial to the object's identity or the scene's context, flipping may not be suitable. For example, in handwritten text recognition, letter and number orientation is fundamental, and flipping changes the characters' semantics. Similarly, in scenarios like medical imaging, where a left-right flip might alter the diagnosis (consider organs' positions), this method is not applicable.

**24. What limits do you see in this type of data increase by transformation of the dataset?** Augmentation can sometimes exacerbate class imbalances, particularly if transformations are applied uniformly across classes without considering their initial distribution. Augmenting already overrepresented classes can intensify the imbalance. Some transformations might introduce features that are unnatural, potentially leading the model to learn false patterns. For instance, excessive rotation or scaling might create unnatural shapes or perspectives that the model might wrongly interpret as relevant features. Data augmentation, especially sophisticated methods, increases the computational burden. It requires additional processing for each image and can slow down the training process, demanding more efficient hardware solutions. While augmentation expands the dataset, it doesn't contribute new information beyond the initial data's constraints. The transformations create variability but within the context of existing data, possibly limiting the model's ability to generalize in radically new scenarios.

**25. Bonus: Other data augmentation methods are possible. Find out which ones and test some.** A wide array of augmentation methods is available, including but not limited to rotation, flips, zooming, perspective alteration, affine transformations, color adjustments (brightness, contrast, saturation, hue), sharpness modification, Gaussian blur, polarization, solarization, pixel erasure, channel permutation, grayscale conversion, and more.

We tried random erasing and random rotation ( $0^\circ$  to  $180^\circ$ ) on the train dataset and random rotation ( $0^\circ$  to  $90^\circ$ ) on the test dataset. Results are displayed on Figure 2.9.

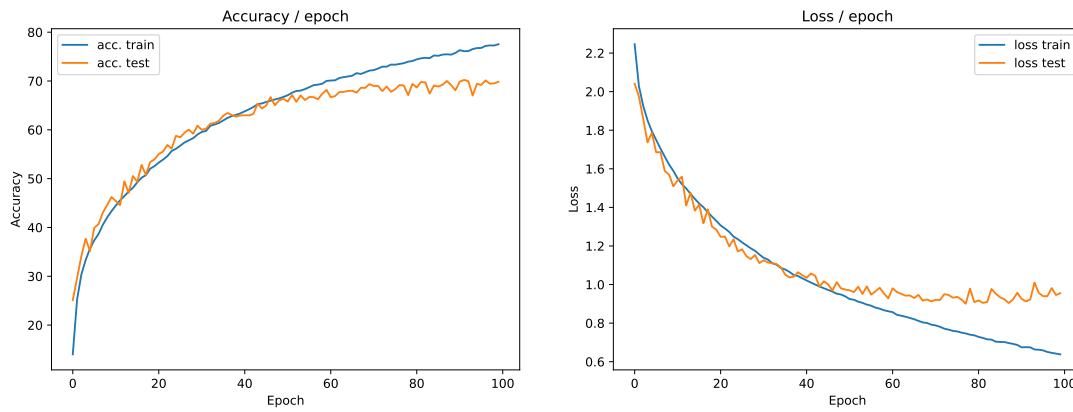


Figure 2.9: Accuracy and losses in train and test using data augmentation

One may also wonder what happens when we train a model using certain transformations and test it using transformations that the model hasn't encountered during training. Figure 2.10 illustrates the consequences of this approach, where we applied random erasing and random rotation ( $0^\circ$  to  $180^\circ$ ) to the training dataset and random rotation ( $180^\circ$  to  $360^\circ$ ) to the test dataset. This experiment highlights that even though both involve rotations, the model may struggle to generalize when faced with previously unseen transformations.

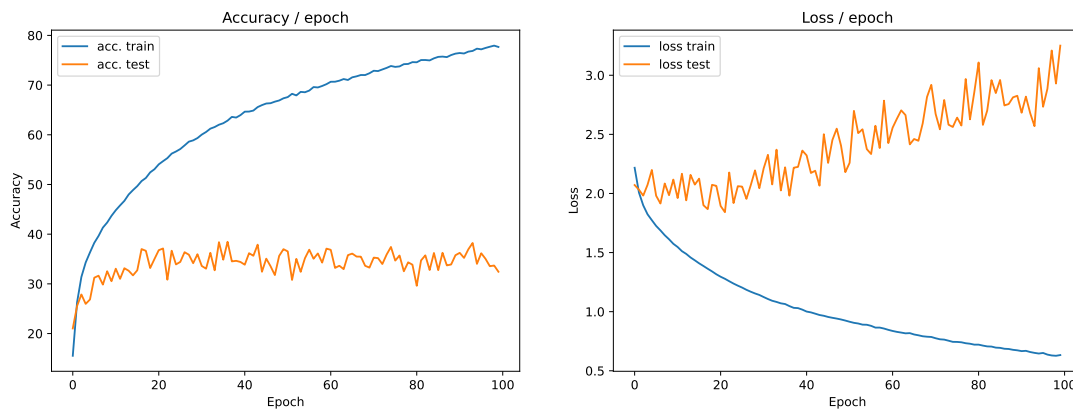


Figure 2.10: Accuracy and losses in train and test using data augmentation

As a final note, we want to mention that as we introduced more transformations, we observed a significant increase in training time, with the training process taking at least twice as long. This highlights the computational complexity associated with this technique, as discussed in question 24.

### 2.3.3 Variants of the optimization algorithm

**26. Describe your experimental results and compare them to previous results, including learning stability.** Incorporating a learning rate scheduler controls how quickly or slowly a model adapts to the problem at hand. Thus, we expect a controlled model convergence, as the model's training process benefits from a more cautious approach to reaching the global minimum of the loss function by gradually decreasing the learning rate. It should also permits to avoid plateaus, i.e. regions where the model's accuracy does not improve significantly due to slow learning. Last, we expect an enhanced training stability.

Figure 2.11 depicts our results. We can see that the loss curve exhibits greater stability when compared to the baseline results (refer to Figure 2.5).

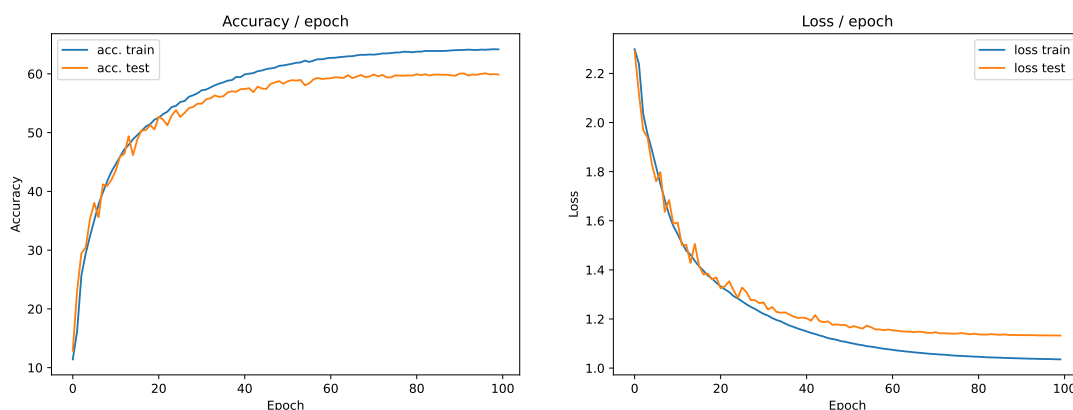


Figure 2.11: Accuracy and losses using an exponential decay scheduler with SGD

**27. Why does this method improve learning?** The fundamental reason exponential decay learning rate scheduling improves learning lies in its dynamic adaptation to the training process's needs. In the early phases of training, where the initial parameters are likely far from optimal, a larger learning rate helps cover more ground. However, as training progresses, it becomes more about refinement, requiring smaller updates to fine-tune the model, something achieved by reducing the learning rate.

Moreover, this method acknowledges that the landscape of the loss function is not uniformly shaped. Some areas might be steep (necessitating larger steps for efficiency), while others might be flat or have a complex curvature (where smaller steps are necessary for finding the precise minimum). An exponential decay schedule for the learning rate adapts to these needs inherently, allowing for a more nuanced and, consequently, more effective approach to model training.

However, it's essential to approach the setting of the initial learning rate and decay factor with care, as these hyperparameters can significantly affect the training's effectiveness and efficiency. Too rapid a decay might mean the learning rate reduces too quickly, curtailing the model's capacity to learn, while too slow a decay might keep the learning rate too high, leading to the aforementioned issues of overshooting or oscillating around the minimum.

**28. Bonus: Many other variants of SGD exist and many learning rate planning strategies exist. Which ones? Test some of them.** In this study, we employed various optimizer configurations and learning rate scheduling techniques. Initially, we visualize the learning rate dynamics throughout the training epochs using Figure 2.12. Subsequently, we analyze the impact of these scheduling methods on both training and testing accuracies, as well as losses, as demonstrated in Figure 2.13.

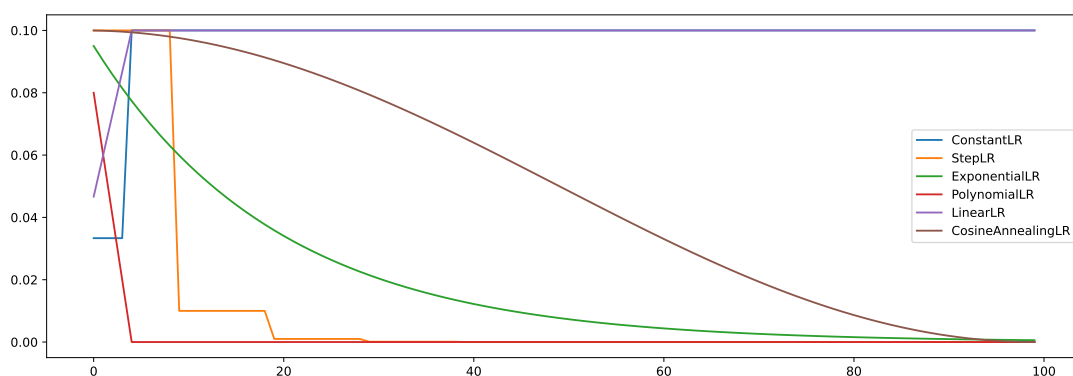


Figure 2.12: Learning rates curves



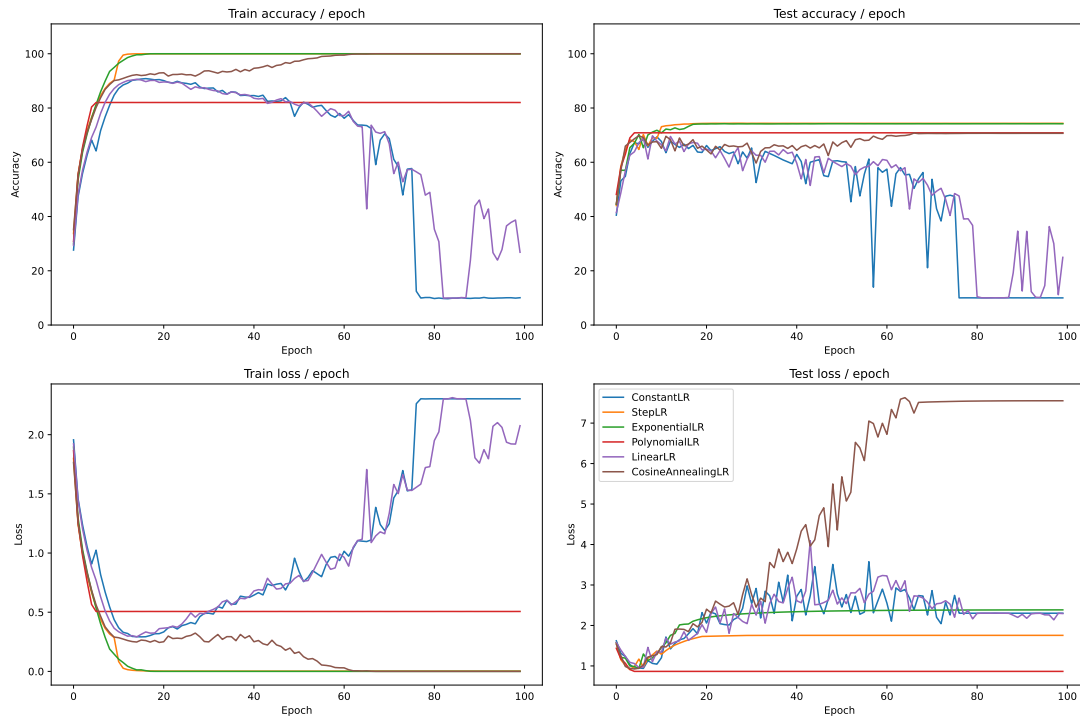


Figure 2.13: Influence of different schedulers on SGD

Firstly, it's noteworthy that both the Constant and Linear learning rate schedulers exhibit a tendency to increase the learning rate, resulting in detrimental performance after a considerable number of epochs, even though it might have been beneficial at the beginning (otherwise they would not have showed this behavior). Conversely, all the other schedulers converge to a similar learning rate value, with the Polynomial scheduler being the fastest to reach this point, albeit with a relatively modest performance of approximately 80 % in train and 70 % in test. The top three performers among the schedulers are Exponential, Step (which exhibit similar behavior), and Cosine. Cosine, owing to the nature of its function, takes the longest to reach its learning rate plateau.

Additionally, it's essential to recognize that there exist alternative optimizers more effective than SGD. We conducted a comparative analysis, the results of which are depicted in Figure 2.14. Among the optimizers, Adagrad demonstrated the highest performance, closely followed by its variant, Adadelata. Conversely, both AdamW and RMSProp exhibited relatively poorer performance and displayed numerical instability during testing, with Adam occupying an intermediate position. It's noteworthy that none of the optimization methods led to overfitting.

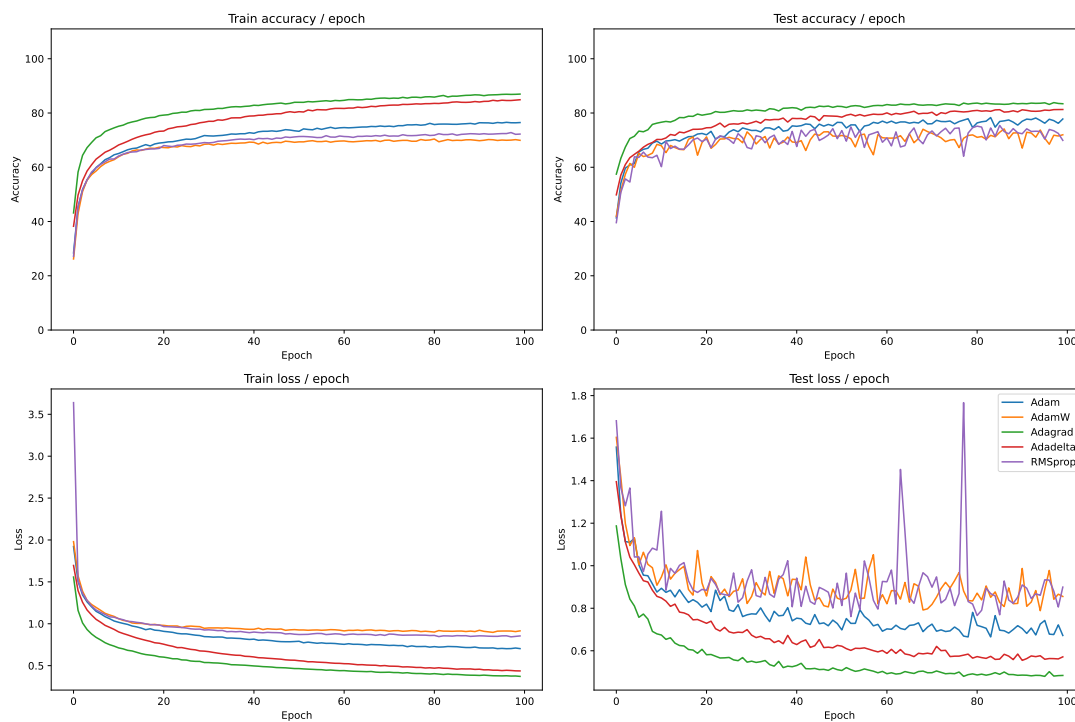


Figure 2.14: Influence of different optimizers

### 2.3.4 Regularization of the network by *dropout*

**29. Describe your experimental results and compare them to previous results.** Integrating a dropout layer in a neural network is a strategic move to combat the risk of overfitting, especially before a fully connected layer that has a substantial number of weights. Thus, we expect reduced overfitting and enhanced network performance. Figure 2.15 and Figure 2.16 depicts two experiments with dropout.

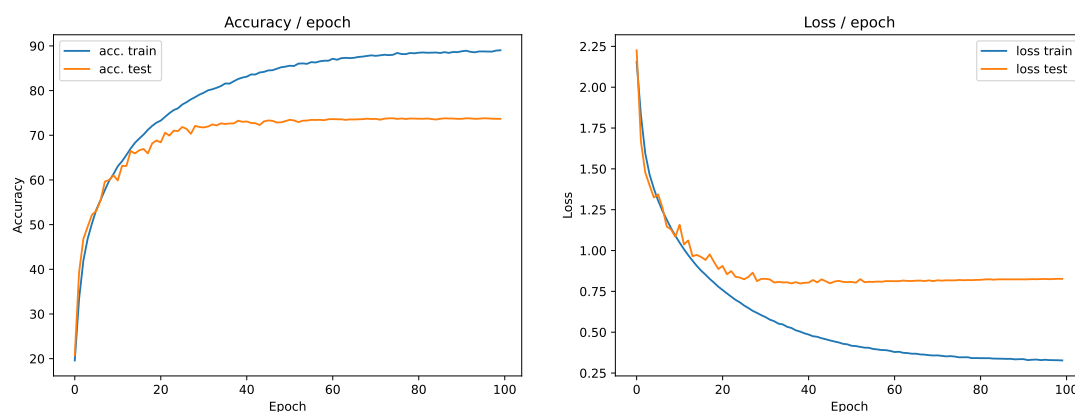


Figure 2.15: Accuracy and losses in train and test, using a dropout on fc4

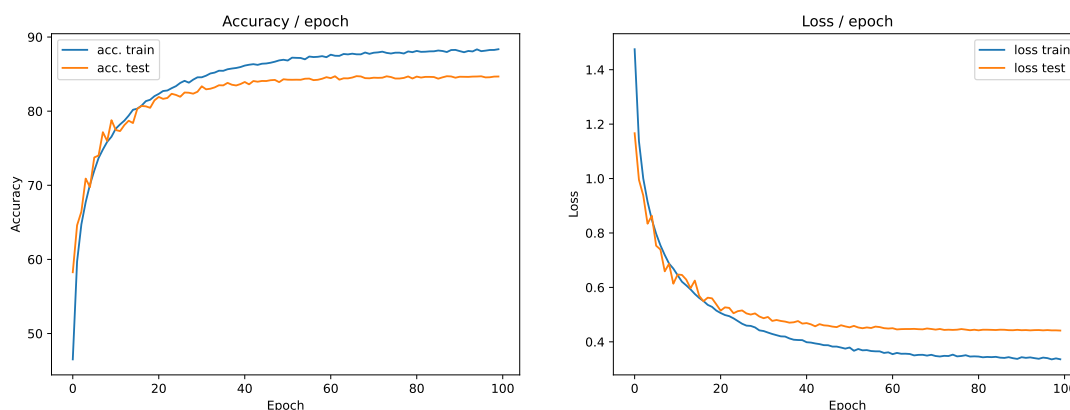


Figure 2.16: Accuracy and losses in train and test, using a dropout on fc4 and a scheduler

**30. What is regularization in general?** Regularization refers to techniques that constrain a model's learning capacity indirectly or directly to prevent overfitting, which occurs when a model learns the training data too closely and fails to generalize well to unseen data. Regularization methods work by adding a penalty to the loss function or by manipulating the training data or model architecture. The goal is to discourage the learning of overly complex models, as these are more prone to overfitting, by prioritizing simpler, more generalizable models that perform better on unseen data. Common regularization techniques include L1 and L2 regularization, dropout, data augmentation, early stopping, and even architectural choices like choosing simpler models over more complex ones.

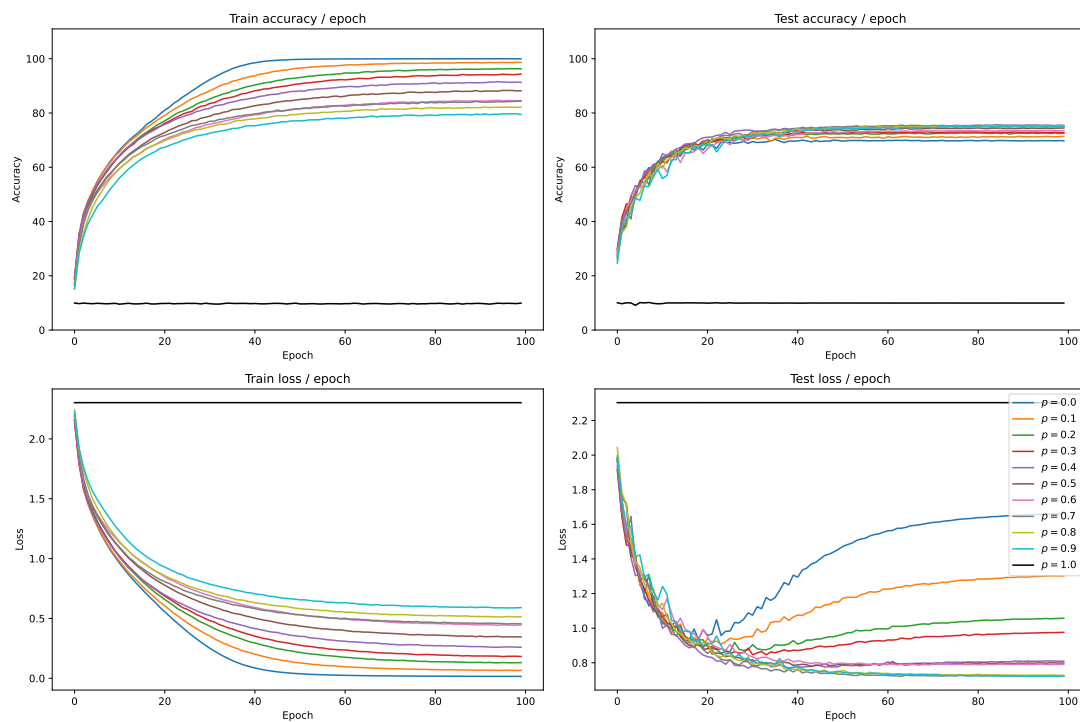
**31. Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it?** Dropout is a regularization technique where random neurons (along with their connections) are "dropped out" (or temporarily removed) from the network with a certain probability, preventing their activation, during training.

Dropout forces neurons to learn more robust and independent representations by making the presence of other neurons unreliable. Without dropout, neurons can become co-adapted, meaning they rely on the specific output of other neurons. By randomly removing different neurons, dropout disrupts these potentially intricate co-adaptations, ensuring that each neuron contributes individually to the solution.

Dropout can be interpreted as a form of model averaging or ensemble learning. Each training epoch uses a different "thinned" network, and predictions on unseen data are effectively the aggregated output of these numerous networks. This ensemble approach is known to generally improve model robustness and generalization.

Dropout introduces variability into the network, somewhat akin to injecting noise into the hidden units' outputs. This noise can help prevent the network from fitting the training data too precisely, effectively acting as a form of stochastic data augmentation.

**32. What is the influence of the hyperparameter of this layer?** The hyperparameter of this layer is called the "dropout rate" and represents the probability that any given neuron will be temporarily dropped. A very high dropout rate means more neurons are dropped during training. While this might increase regularization (thus potentially reducing overfitting), if the rate is too high, the network might lose too much information, making learning difficult and possibly leading to underfitting. A lower dropout rate maintains more neurons active but might offer insufficient regularization, making the model still prone to overfitting, especially if it's a complex, deep network. Results of this hyperparameter on our architecture are displayed on Figure 2.17. In our specific case, a minimal dropout rate enhances training performance but leads to overfitting. Conversely, employing a high dropout rate significantly mitigates the overfitting issue. Unexpectedly, setting the dropout rate to  $p = 1$  (meaning *all* layers are dropped) renders our model incapable of learning.

Figure 2.17: Influence of dropout rate  $p$ 

**33. What is the difference in behavior of the dropout layer between training and test?** During training, neurons are randomly deactivated with each batch of data, with the dropout rate determining the fraction of neurons to drop. This random deactivation happens every time a new batch is fed to the model. During test, dropout is disabled and acts as identity function, meaning all neurons are active. This approach is necessary to prevent the random uncertainty introduced by dropout during training from affecting the model's predictions on new data.

### 2.3.5 Use of *batch normalization*

**34. Describe your experimental results and compare them to previous results.** oui

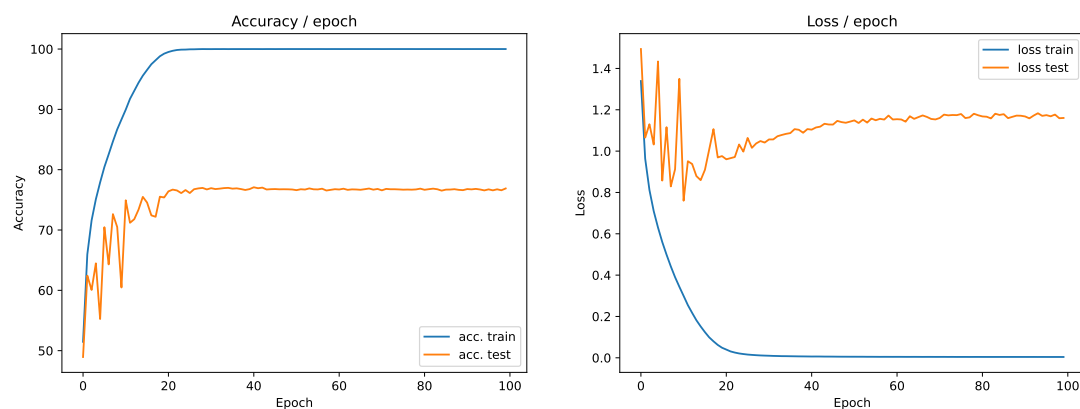


Figure 2.18: Accuracy and losses in train and test

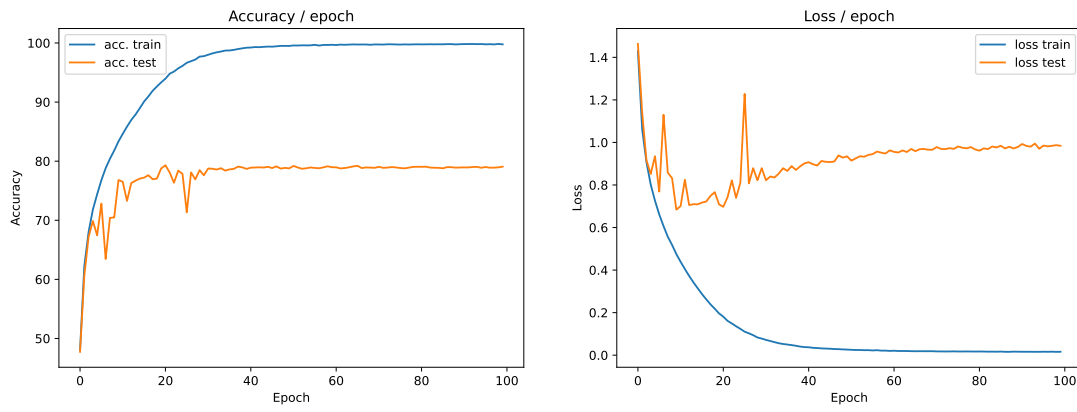


Figure 2.19: Accuracy and losses in train and test, with dropout layer

### 2.3.6 Combination of all methods

Now that we have individually explored various methods to enhance our results within a relatively straightforward convolutional network architecture, one might wonder what happens when we combine all these techniques (standardization, data increase, optimizer, scheduler, dropout, batch normalization), much like the Power Rangers assembling with their mechs. Our expectation is that this comprehensive approach will yield significantly improved results characterized by minimal overfitting, strong generalization, and enhanced robustness. Results confirm our expectation and are displayed in Figure 2.20. It's worth noting that this approach may appear excessive, as it produces results similar to the one obtained using a dropout and a learning rate scheduler.

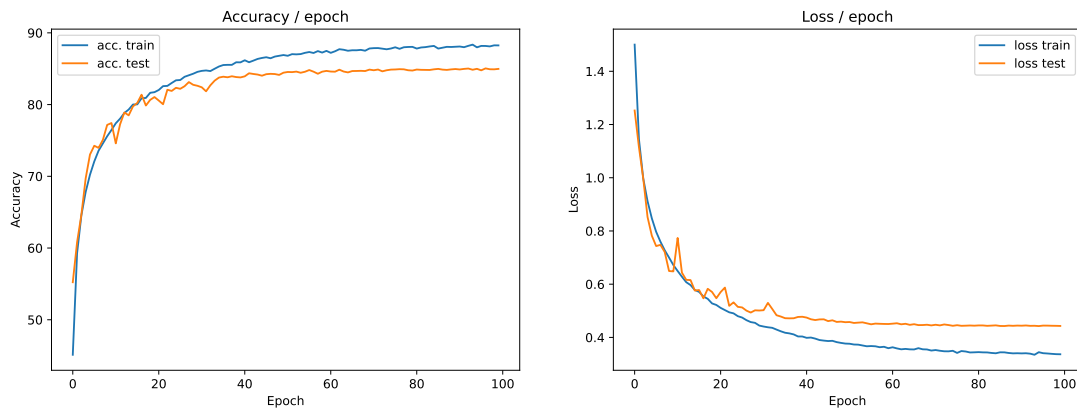
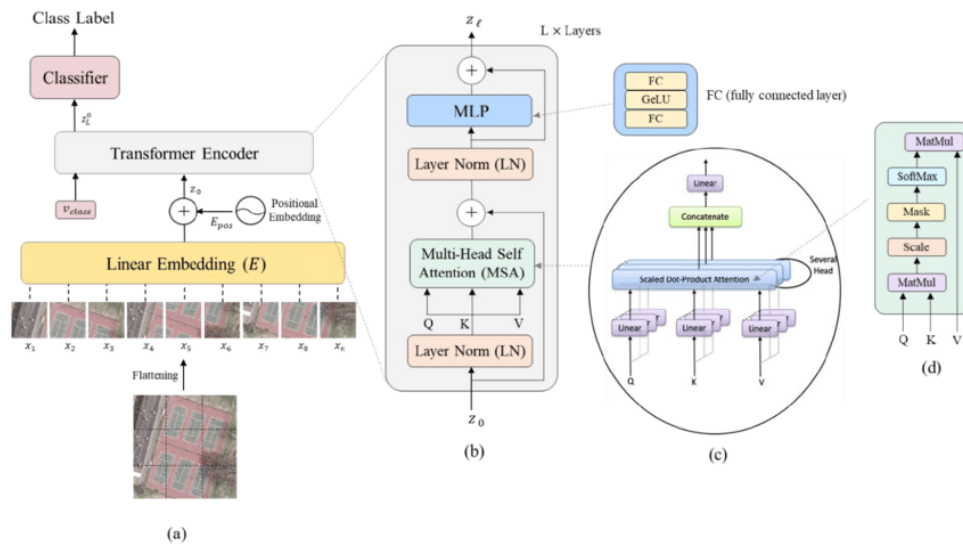


Figure 2.20: Accuracy and losses in train and test using all methods

# Chapter 3

## Introduction to transformers



### 3.1 Self-Attention

**What is the main feature of self-attention, especially compared to its convolutional counterpart? What is its main challenge in terms of computation/memory?** Unlike convolutional layers that consider a local neighborhood around each input position, self-attention can weigh all positions across the entire sequence (here the image), providing a form of global receptive field. This is beneficial for tasks where the relevant context can be far from the current position. Convolutional layers have a fixed receptive field, which limits their ability to capture long-range dependencies unless many layers are stacked or dilated convolutions are used.

The primary challenge of self-attention is its computational and memory complexity, particularly the quadratic complexity with respect to the sequence length. In self-attention, every element in the input sequence interacts with every other element, leading to a time and space complexity of something like  $O(n^2)$ , where  $n$  is the length of the input sequence. This makes self-attention computationally expensive and memory-intensive for long sequences.

**At first, we are going to only consider the simple case of one head. Write the equations and complete the following code. And don't forget a final linear projection at the end!** First we need to compute three different linear projection of the input  $X$  : the Query  $Q$ , the Key  $K$ , and the Value  $V$ .

$$\begin{aligned} Q &= XW_q, \\ K &= XW_k, \\ V &= XW_v \end{aligned}$$

The main part of attention is the dot product between the Query and the Key. Those are a set of vector that we learn. The goal is to learn representation of Key that answer the Query to orient the attention. The dot product is high where the model need attention.

But we must be aware that when values of the key and query vectors are large, the dot products can become very large. This can result in large values in the softmax function, leading to vanishing gradients during backpropagation. Scaling down the dot products by  $\sqrt{d_k}$  (where  $d_k$  is the dimensionality of the query and key vectors) helps in keeping the gradients in a manageable range, which in turn stabilizes the training. As the variance of the dot product grows with the dimensionality, it also keep it constant.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right) V.$$

The Value matrix represents the actual content of the input tokens. Once the attention scores are computed, they are used to weight the value vectors  $V$ .

Optionnaly, it's possible to do a last final linear projection at the then, in this case we have

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right) VW.$$

## 3.2 Multi-head self-attention

**Write the equations of Multi-Heads Self-Attention.**

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O.$$

where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ . The Attention function is the same as described in the single-head attention, and  $W^O$  is a final linear layer's weights.

## 3.3 Transformer block

**Write the equations of the transformer block.** Let's define the operations within a Transformers block:

1. Let  $x$  be the input to the Transformers block, which in our case is a patch from the image plus a positional embedding. We begin by normalizing the input using Layer Normalization (LayerNorm):

$$z = \text{LayerNorm}(x)$$

2. We then compute the Query (Q), Key (K), and Value (V) matrices, which are used in the Multi-Head Attention mechanism as explained in the previous question:

$$Q = z \cdot W_q, \quad K = z \cdot W_k, \quad V = z \cdot W_v$$

3. Next, we pass these matrices  $Q$ ,  $K$ , and  $V$  into the Multi-Head Attention mechanism:

$$\text{AttOutput} = \text{MultiHead}(Q, K, V)$$

4. We add the normalized input  $z$  to the output of the Multi-Head Attention and apply Layer Normalization again:

$$\text{MidLayerOutput} = \text{LayerNorm}(z + \text{AttOutput})$$

5. The MLP (Multi-Layer Perceptron) head is a two-layer linear network with a GeLU (Gaussian Error Linear Unit) or ReLU (Rectified Linear Unit) activation function, defined as follows:

$$\text{MLP}(x) = \text{ReLU}(x \cdot W_1 + b_1) \cdot W_2 + b_2$$

Finally, we compute the output of the MLP and add it to the MidLayerOutput to obtain the final output of the Transformers block:

$$\text{MLPOutput} = \text{MLP}(\text{MidLayerOutput})$$

$$\text{FinalOutput} = \text{MLPOutput} + \text{MidLayerOutput}$$

After this last addition, we obtain the final output of a Transformers block.

## 3.4 Full ViT model

**Explain what is a Class token and why we use it?** The class token is positioned as the first token. It is represented as a learnable vector which gets updated during the forward pass. It accumulates information from different parts of the image as it gets updated through the Transformer layers. As it sums up all the image, we use it to classify the image.

**Explain what is the positional embedding (PE) and why it is important?** The PE are used to provide the spatial information to the model. Here we use sinusoidal encoding, it provides a fixed, unique and easy-to-generate encoding for the position. Figure 3.1 is employed to visualize the appearance of sinusoidal encoding. In the context of an image transformer, the x-axis represents the number of image patches created, while the y-axis represents the size of an image patch embedding. Another approach is to learn this positional embedding. This method allows the model to learn the most useful positional representations. This positional embedding is then summed to the image embedding.

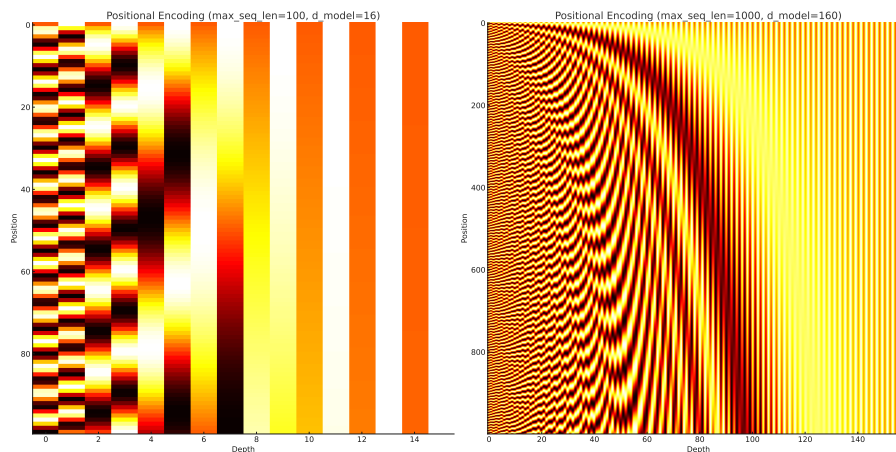


Figure 3.1: Visualization of Sinusoidal Positional Encodings in Transformer Models

## 3.5 Experiment on MNIST!

The overall performance of our small transformer is really good on MNIST. The convergence is quick and the accuracy is really high for every experiment.

### 3.5.1 Hyperparameters influences

As we did in previous experiments, we'll analyse the influence of hyperparameter by plotting loss and accuracy on the train and test dataset.

**Embed dimension** First, let's examine the performance of different embedding dimensions in Figure 3.2. Several notable observations can be made:

In this figure, it's evident that an embedding dimension of 16 takes slightly more epochs to converge compared to other sizes. Conversely, the size of 128 converges at the same speed but achieves the lowest final accuracy and loss. Importantly, this larger size does not exhibit signs of overfitting. It's worth mentioning that training with an embedding size of 128 also requires more time (in terms of minutes) due to the increased number of parameters.

On the other hand, the embedding size of 16 achieves similar performance to the other sizes but with fewer parameters, resulting in reduced energy consumption and training time. This aligns with the principle of Occam's razor, emphasizing the value of simplicity.



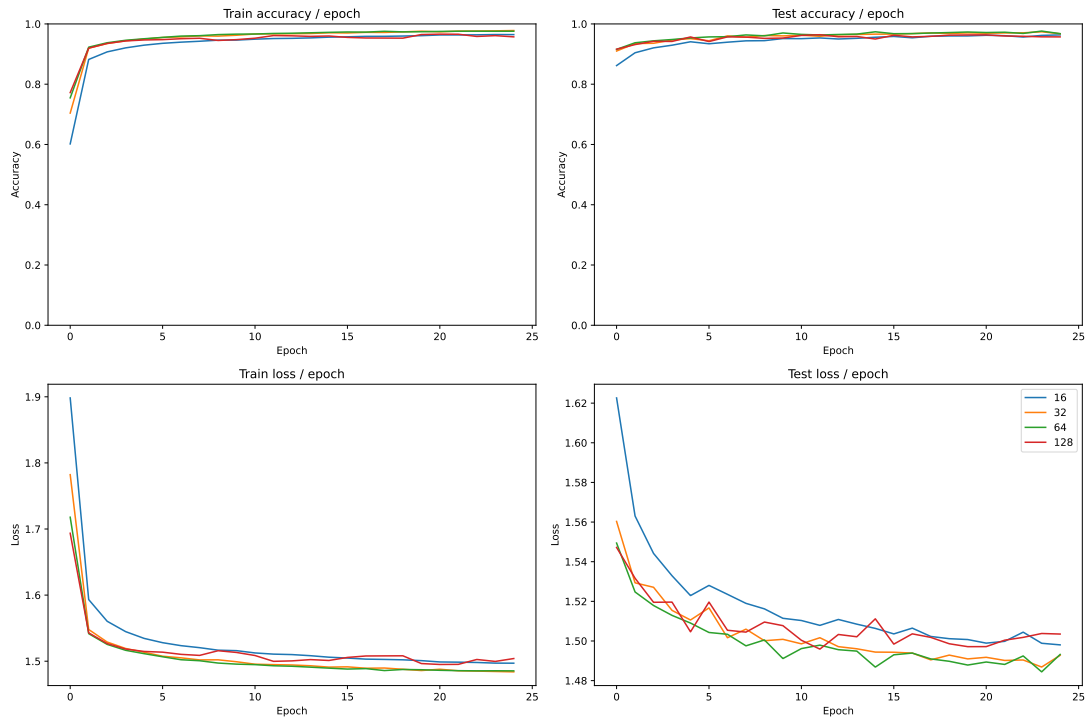


Figure 3.2: Influence of the size of the embed dimension

**Patch size** Figure 3.3 illustrates the impact of patch size on the learning process. Similar to the previous paragraph, it's apparent that all conditions ultimately converge to nearly the same level of accuracy. However, the test loss provides more informative insights.

As anticipated, a patch size as small as 2 by 2 pixels encountered greater challenges in learning due to its diminutive dimensions. On the other hand, a patch size of 28 by 28 pixels is equivalent to the size of an MNIST image, and thus, no patches are formed. This particular configuration, along with a patch size of 14 pixels, exhibits the lowest initial loss and maintains this characteristic throughout most of the training epochs.

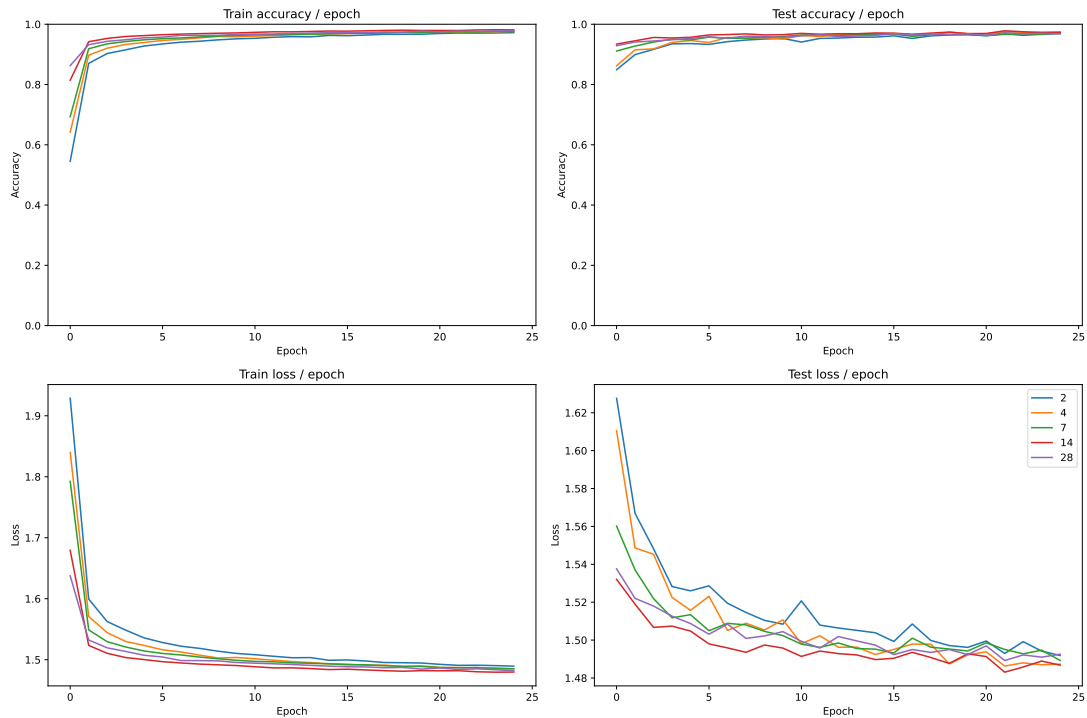


Figure 3.3: Influence of the patch size

**Number of transformers blocks** The influence of the number of transformer blocks is evident in Figure 3.4. This experiment exhibits similarities to the one conducted with embedding sizes. In all configurations, convergence to a high level of accuracy is observed. However, it's important to note that a high number of blocks can potentially result instability in learning, as indicated by the test loss curve when using 8 blocks. This condition also converge to a lower accuracy.

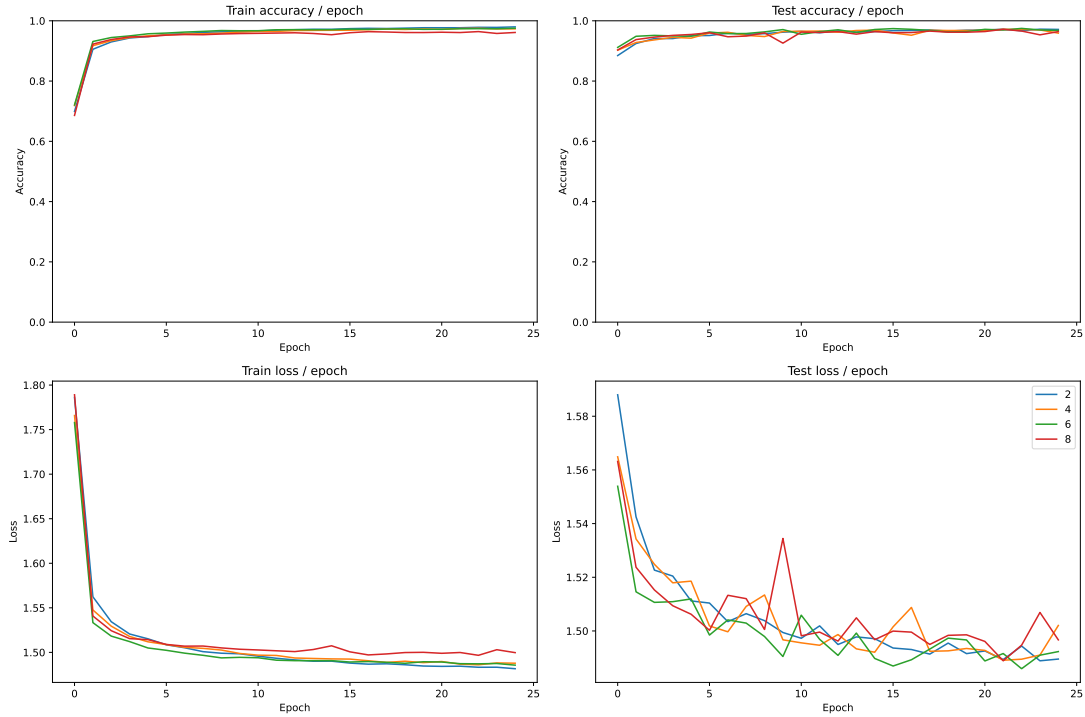


Figure 3.4: Influence of the number of transformers blocks

**Number of heads in a block** Plot in Figure 3.5 suggests that an increase in the number of self-attention heads tends to yield better results as we can see by looking at accuracy in train and test.

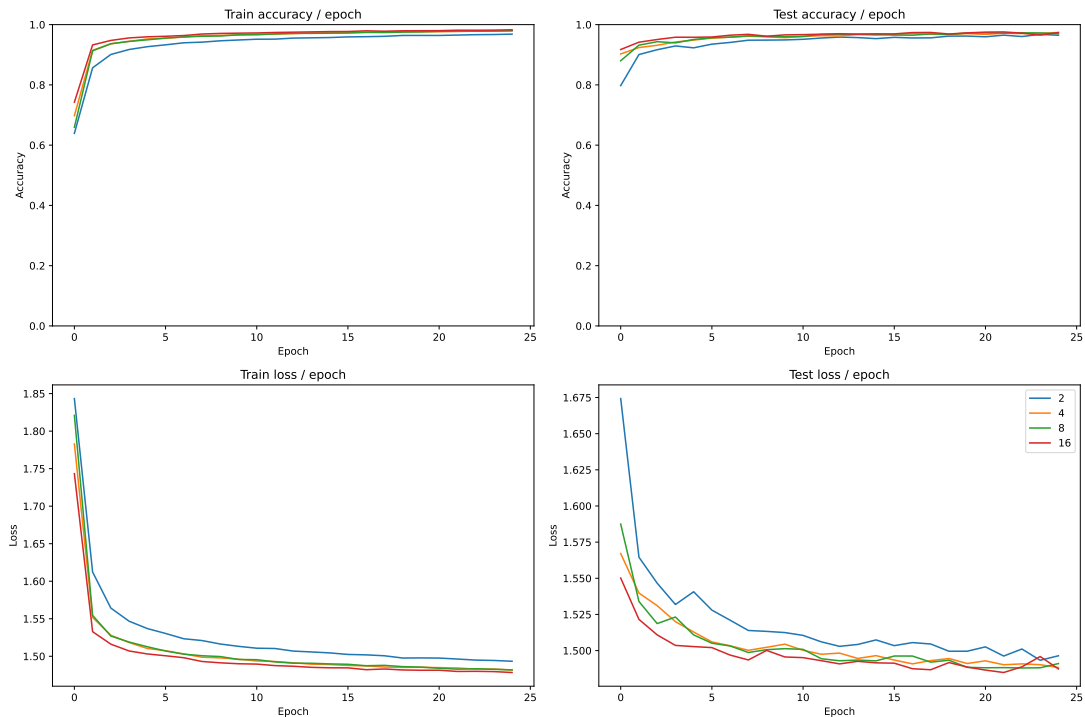


Figure 3.5: Influence of the number of self attention head in one transformer block

**MLP hidden layer size** Finally, we conducted tests to assess the influence of the size of the MLP hidden layer, and the results are displayed in Figure 3.6. Analyzing the test loss, show that increasing the hidden size does not appears to improve the results and don't induce overfitting too. However, it's important to note that a strong ceiling effect on accuracy limits our ability to draw definitive conclusions.

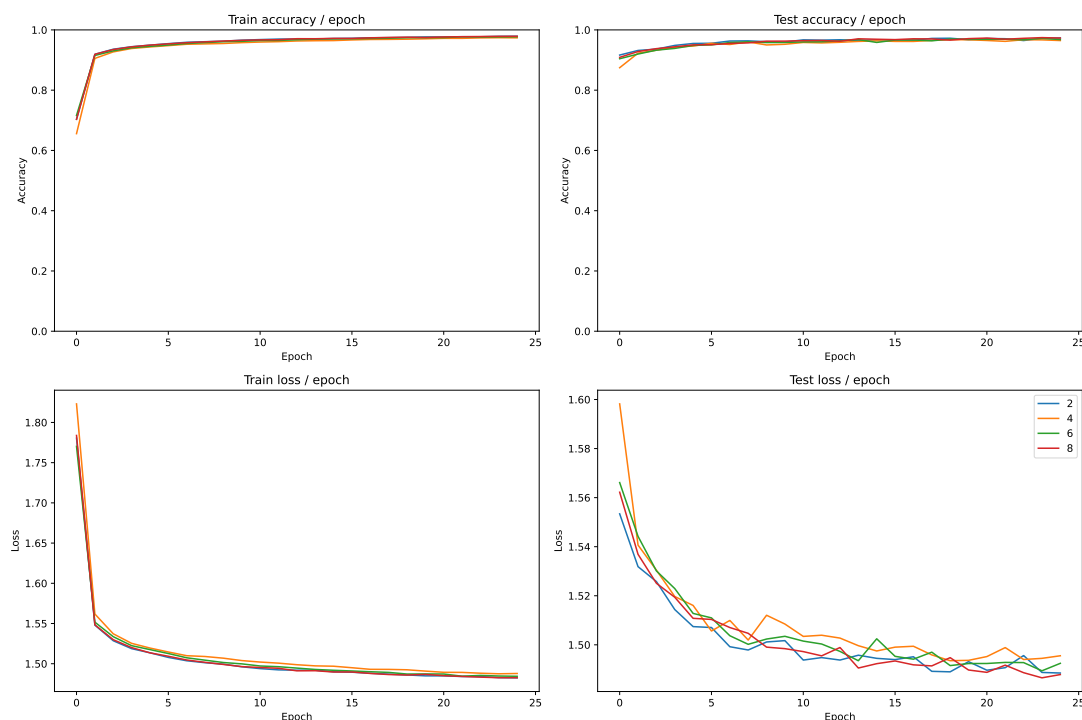


Figure 3.6: Influence of the hidden layer size in the MLP part of a transformer block

## 3.6 Larger transformers

### 3.6.1 Questions

**What is the problem and why we have it? Explain if we have also such problem with CNNs.** The model `vit_base_patch16_224` is trained on 224x224 images from the ImageNet dataset. In the first layer of the ViT, the input image is divided into fixed-size patches to create embedded vectors by passing through a linear projection. It's crucial to maintain the same image size as the one the model was designed for.

In this case, it necessitates resizing the images to 224x224 RGB pixels. However, this may not be ideal as resizing could distort the images and potentially affect the model's performance.

**Provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work.**

### 3.6.2 Learning from scratch

As denoted by Figure 3.7, the ViT had difficulty to learn from scratch on MNIST. Due to the model size (see first question of Section 3.1), four epochs took approximately 45 minutes and brought accuracy to somewhat 20 percents on test.

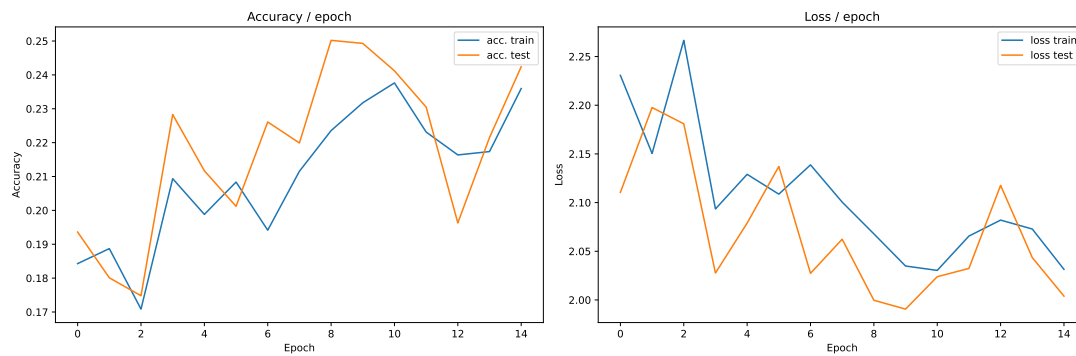


Figure 3.7

### 3.6.3 Fine-tuning

Result are much better when fine-tuning as we can see in Figure 3.8

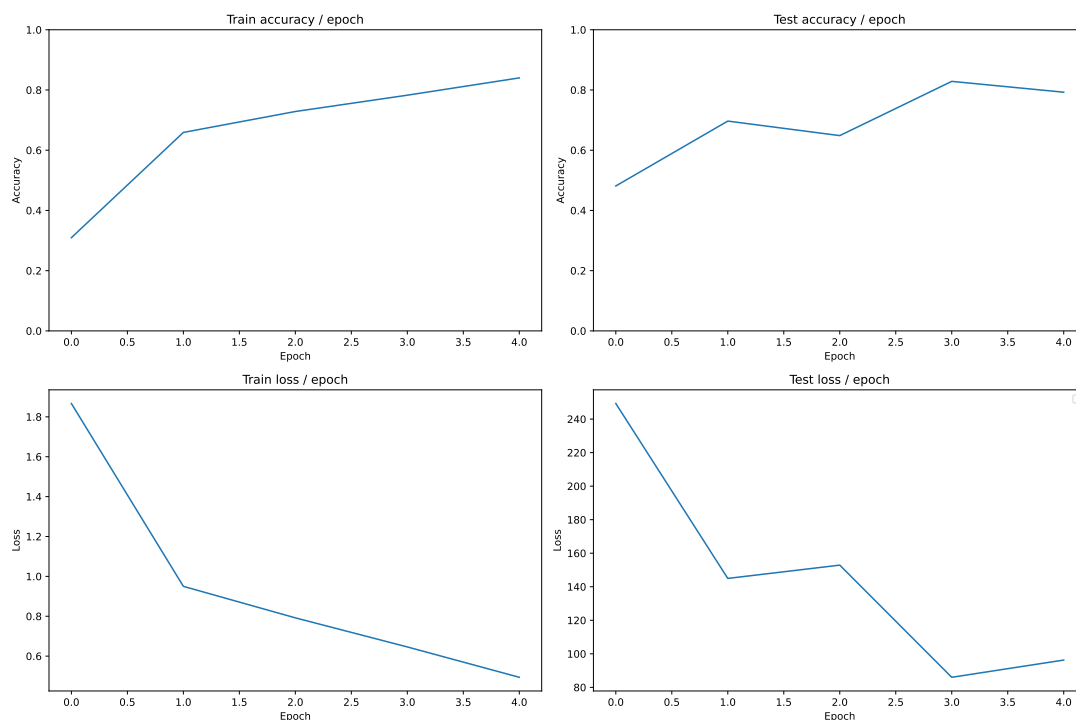


Figure 3.8

## 3.7 Experiments on Fashion MNIST!

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. We intend Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

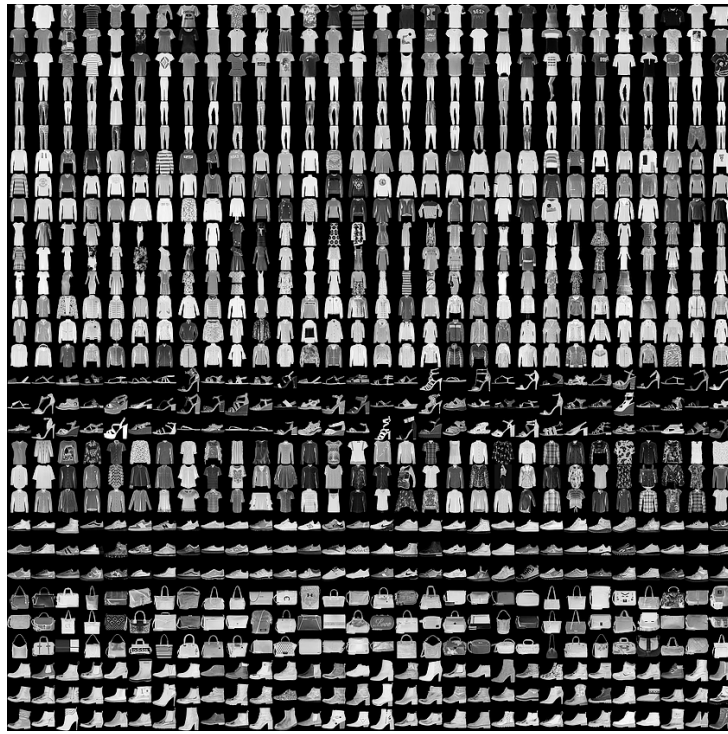


Figure 3.9: Sample of the Fashion-MNIST dataset (each class takes three-rows)

As demonstrated in the previous section, Vision Transformers have pushed the boundaries of the classic MNIST dataset, approaching near-perfect accuracy in testing. To investigate whether a ceiling effect was limiting our ability to draw meaningful conclusions, we replicated all the experiments on the Fashion MNIST dataset. Surprisingly, we did not observe a consistent ceiling effect across all experiments. In some cases, the results were quite similar between the classic MNIST and Fashion MNIST datasets, with the accuracy curves shifting from 100% to around 80%. However, in other instances, we gained valuable insights into the impact of various hyperparameters.

We will now briefly review each experiment to see if we can gain new insights. Each figure mentioned will be linked and displayed in the appendix.

1. **Embedding Dimension:** When comparing the results between the classic MNIST dataset (Figure 3.2) and the Fashion MNIST dataset (Figure 4.1), our earlier conclusions drawn from the classic MNIST dataset hold true but are more pronounced. Larger embedding sizes converge at lower accuracy and loss values, while smaller embedding sizes take more time to converge but show promise. We also see more instability during learning with an embedding size of 128.
2. **Patch Size:** In both the classic MNIST dataset (Figure 3.3) and the Fashion MNIST dataset (Figure 4.2), patch sizes of 28 or 14 pixels continue to exhibit similar performance.
3. **Number of Transformers Blocks:** Our earlier conclusion from the classic MNIST dataset (Figure 3.4), is still valid on the Fashion MNIST dataset (Figure 4.3). Using eight transformers blocks does result in some learning difficulties, as indicated by the red curve having lower accuracy and higher loss.
4. **Number of Heads in a Block:** The observation that more self-attention heads tend to yield better results, as seen in the classic MNIST dataset (Figure 3.5), appears to hold true for the Fashion MNIST dataset as well (Figure 4.4).
5. **MLP Hidden Layer Size:** The comparison between the classic MNIST dataset (Figure 3.6) and the Fashion MNIST dataset (Figure 4.5) reaffirms our previous conclusion. While increasing the MLP hidden layer size still not induce overfitting, we still can't see any effect on the performance.
6. **Larger transformer: training from scratch:**
7. **Larger transformer: fine-tuning:**

## Chapter 4

# Appendix

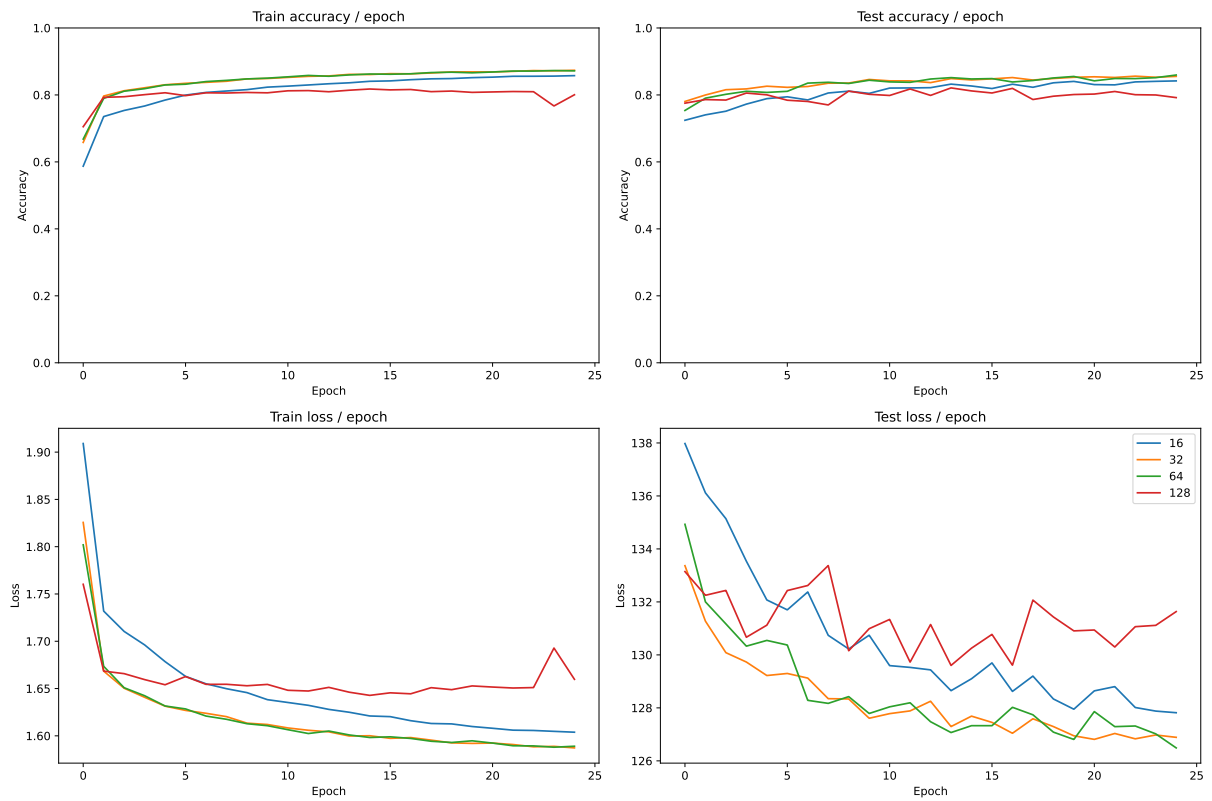


Figure 4.1: Fashion MNIST: Influence of the size of the embed dimension

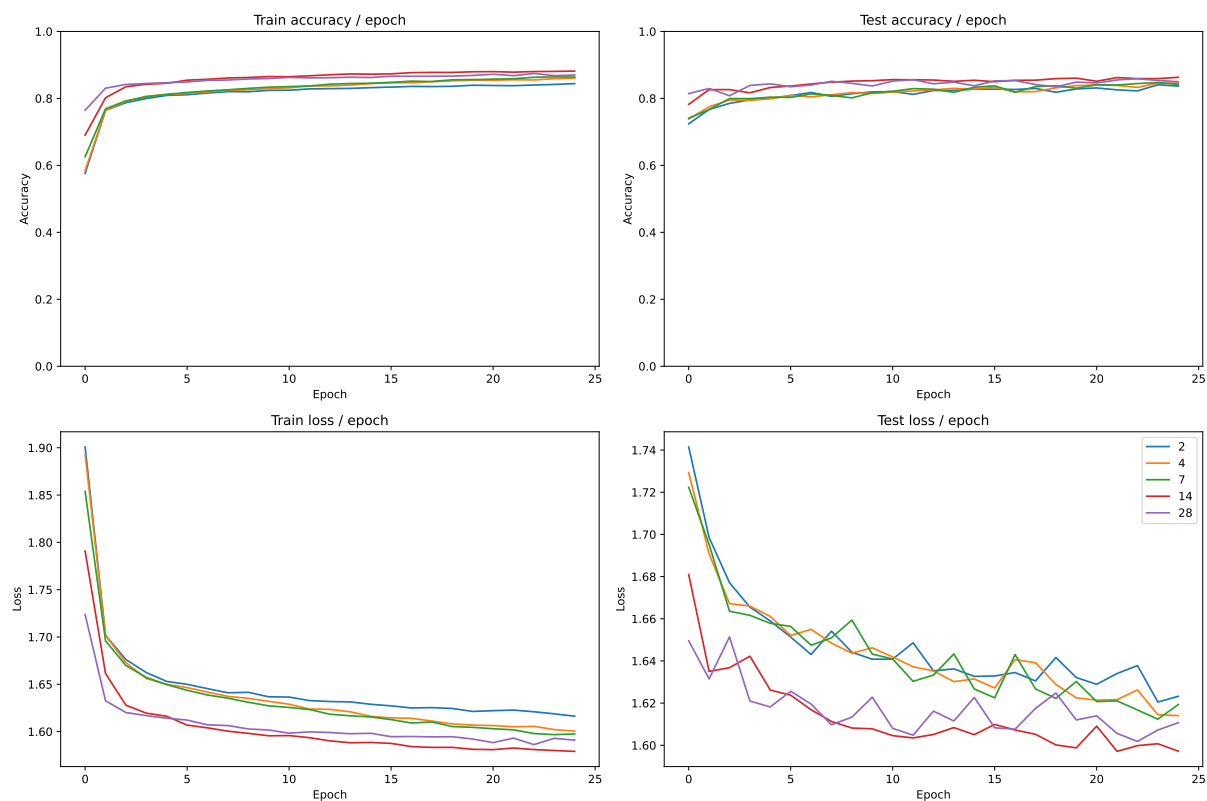


Figure 4.2: Fashion MNIST: Influence of the patch size

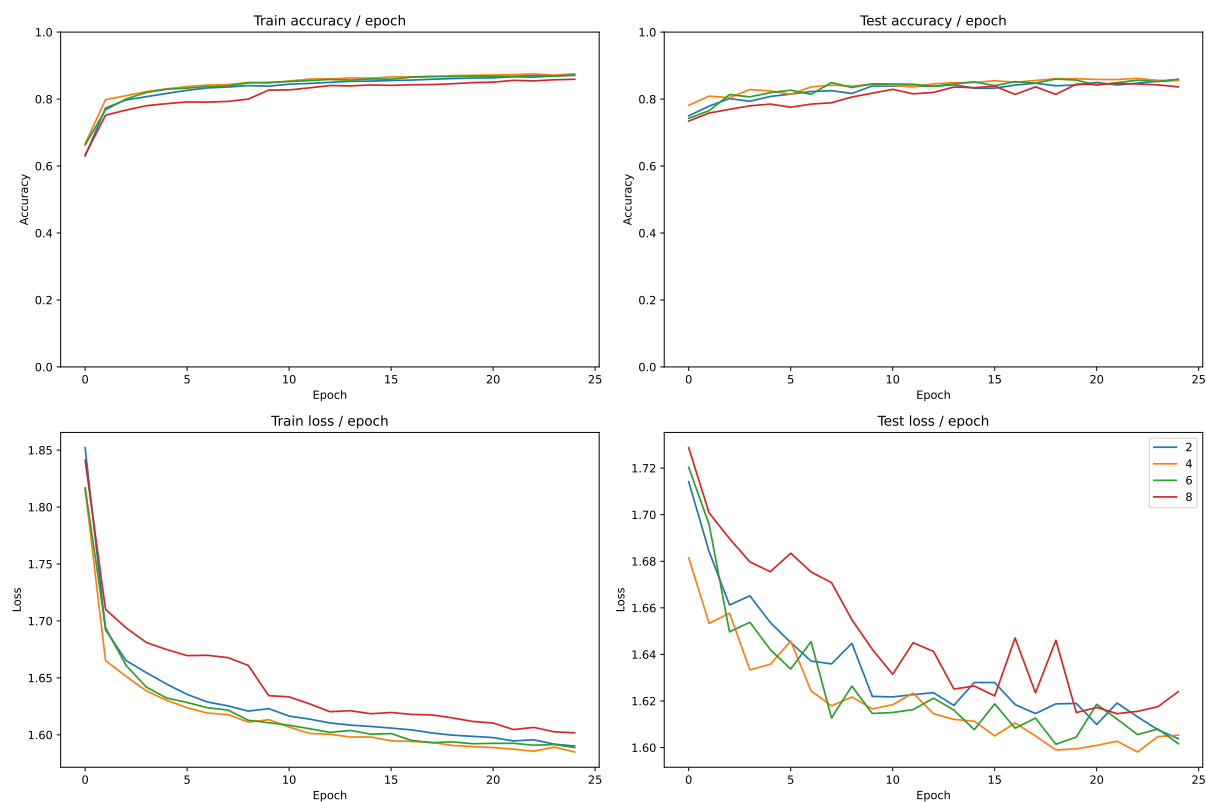


Figure 4.3: Fashion MNIST: Influence of the number of transformers blocks

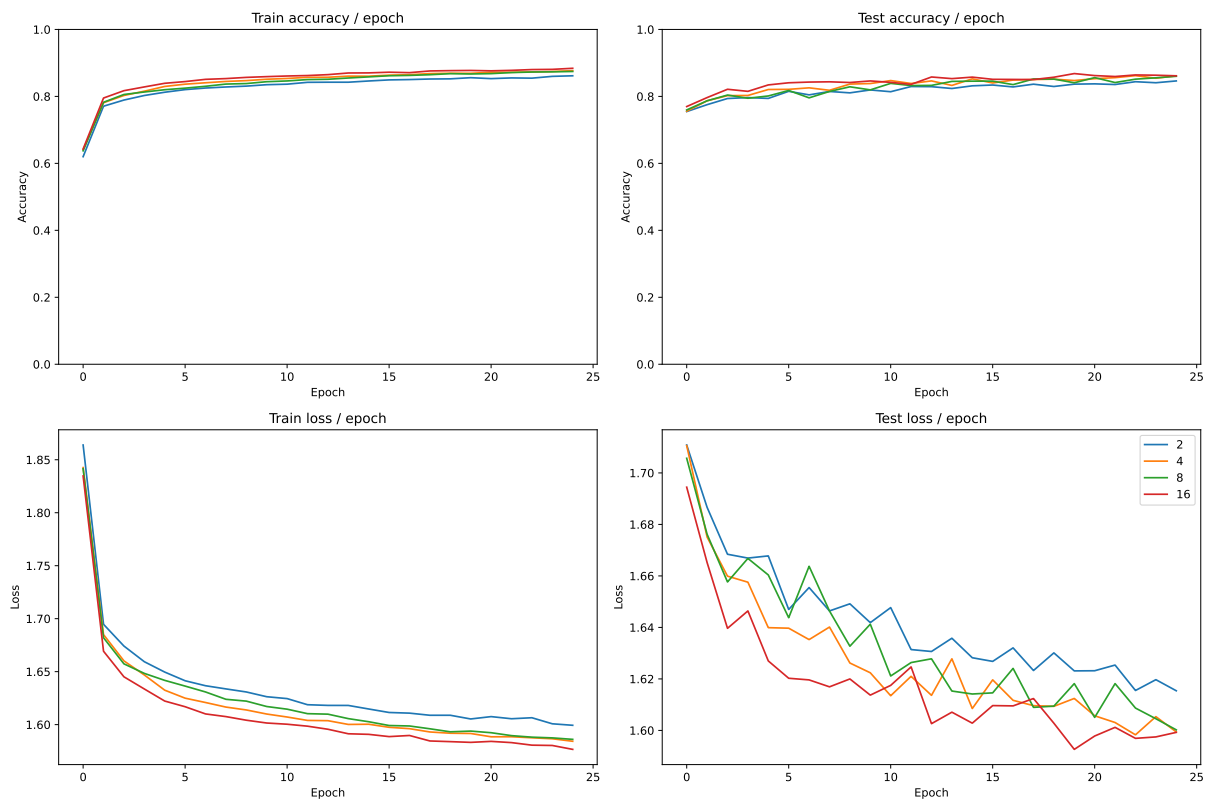


Figure 4.4: Fashion MNIST: Influence of the number of self attention head in one transformer block

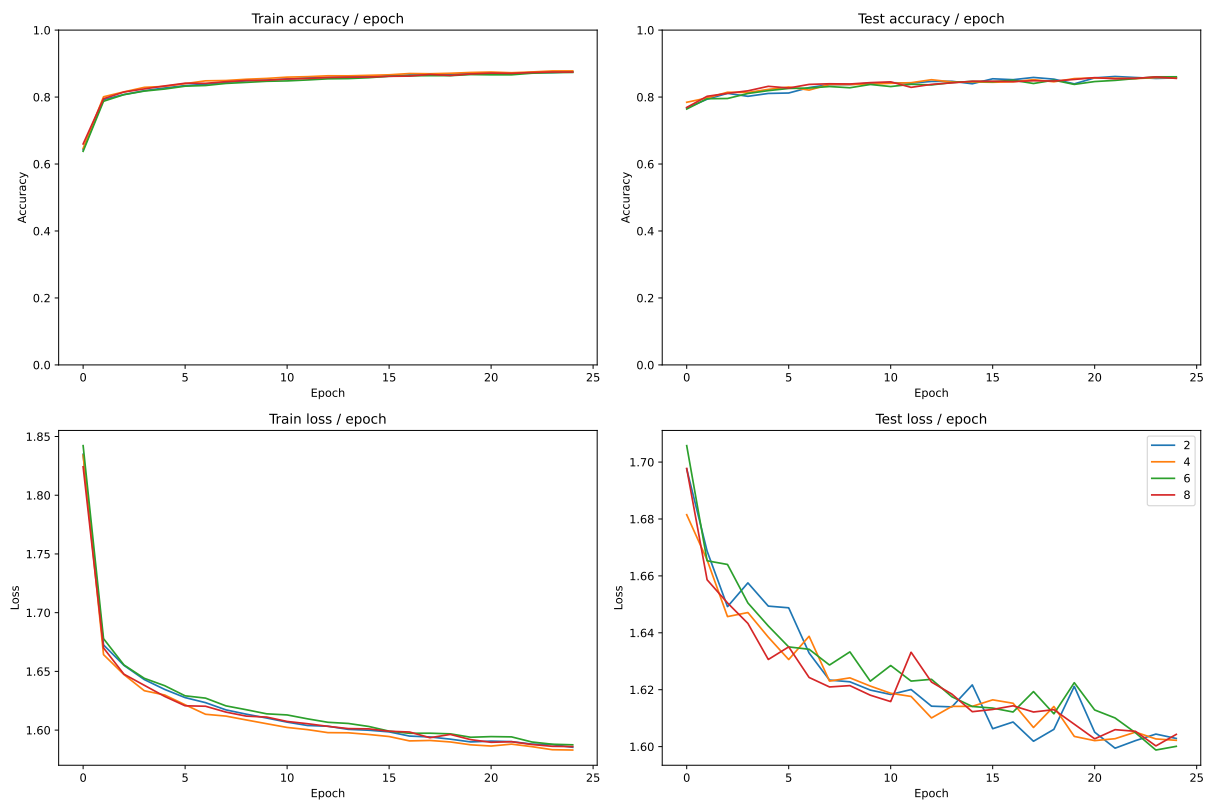


Figure 4.5: Fashion MNIST: Influence of the hidden layer size in the MLP part of a transformer block



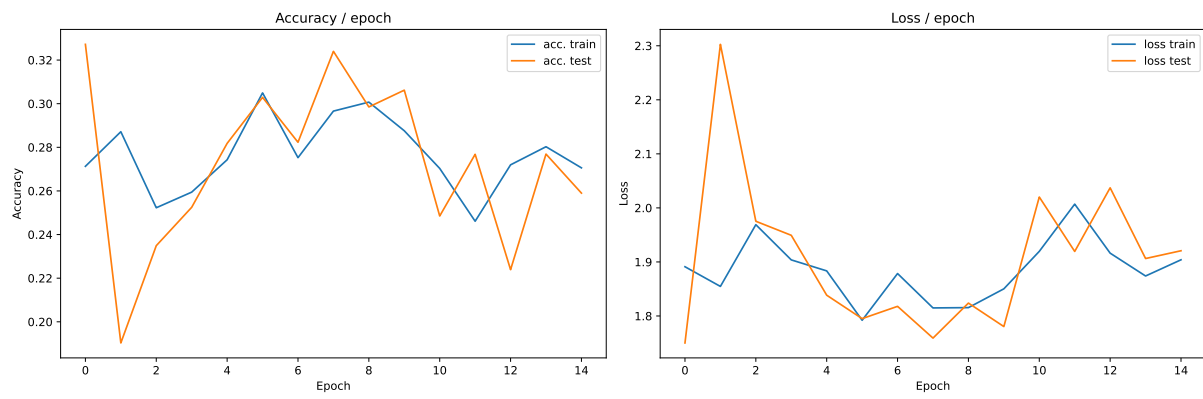


Figure 4.6: Fashion MNIST:

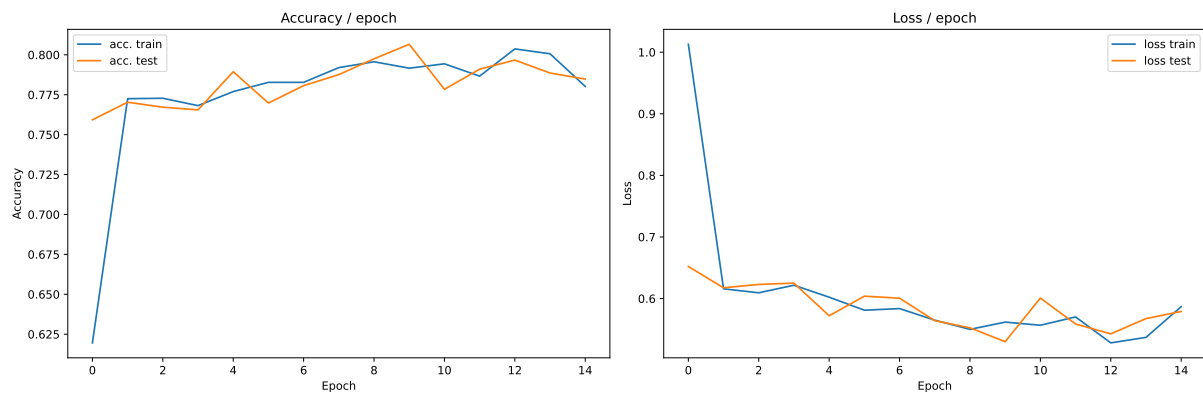


Figure 4.7: Fashion MNIST: