Deep Learning Practical Work Section 1

Report

Basics on deep learning for vision

Student :
Robin SOARES,
Ménalie SELLAHENNEDI

Supervised by :
Clément RAMBOUR,
Matthieu CORD

Repository GitHub : `https://github.com/srsrb/RDFIA`

29 octobre 2024

Table des matières

1 Introduction to Neural Networks

The goal of this practical work is to understand the concepts behind a simple neural network and get familiar with PyTorch useful tools like autograd for the backpropagation. We will start by studying a perceptron with a hidden layer and its learning procedure. We will then implement this network with the PyTorch library first on a toy problem to check that it is working correctly, then on the MNIST dataset.

1.1 Theoretical foundation

1.1.1 Supervised dataset

1. The Train set is used to learn the parameters (w and b) of the model. The val (validation) set is used to compute the performance of the model and therefore adjust the parameters of the model during the training. The test set is used to to evaluate the model's final performance with the optimised parameters after the training and on unseen data.

2. The more the numbers of examples is bigger the more precise will be our model. We will be able to have a larger train set and therefore get a more accurate performance which will reduce the risks of over-fitting.

Network architecture (forward)

3. It is important to add activation functions between linear transformations to break the linearity. It enables the neurons to not give a global linear transformation and to complexify the process.

4. n_h corresponds to the numbers of neurons in the hidden layer. n_x corresponds to the numbers of example n_y corresponds to the numbers of neurons in the output layer. In the case of classification, n_y is the number of classes.

5. y represents the true label (or ground truth) of the data while $\hat{y}$ is the predicted label from the model during the forward pass. The difference between this 2 quantities represents globally the error of the model which is used in the loss function to adjust the model's parameters, so $\hat{y}$ can get closer to y.

6. We use a $SoftMax$ function in the output layer to break the linearity so that the model can render more complex results but also to create a probability distribution that allows us to interpret the outputs as class probabilities.

7. Operations in the hidden layers :
The linear function : $\tilde{h} = W_h \cdot X + b_h$
The activation function : $h = tanh(\tilde{h})$

Operations in the output layer :
The linear function : $\tilde{y} = W_y \cdot h + b_y$
The activation function : $\hat{y} = SoftMax(\tilde{y})$

1.1.2 Loss function

8. To minimize the loss function in the cross-entropy case, $\hat{y}_i$ where $y_i = 1$ (the predicted probability for the correct class) should increase towards 1, while the predicted probabilities for incorrect classes should decrease towards 0.
Whereas in the MSE case, $\hat{y}_i$ needs to get closer to y_i so that their difference could get smaller and therefore decrease the loss function.

9. The MSE function decreases when the difference between the predicted labels and the ground truth gets smaller. We can say it's practical for regression tasks.
The cross-entropy is best suited for classification tasks because its goal is to match high predicted probabilities with the correct classes (from y_i) so that theses high probabilities becomes small numbers due to the *log* and therefore decrease the overall loss, which is the opposite case for low probabilities for right classes.

1.1.3 Optimization algorithm

10. The **stochastic (online) gradient descent (SGD)** method's advantage resides in the frequent update of the model's parameters, making the training faster and dynamically adaptative. However, because this back-propagation happens for each example one by one, the loss value during the train will have an erratic behaviour during the train and converging to a representative model will take much more time.

In contrast, the **classic gradient descent** method, often referred to as batch gradient descent, is the opposite of SGD method. It computes the parameter updates once the entire set of training examples has been processed. This results in more precise and representative updates of the model's parameters, but the algorithm can be quite slow due to the high computational complexity, especially for large datasets.

Finally, the **mini-batch method** is the most well-suited for general cases because it divides the full batch into smaller sections. This approach creates a balance between the precision of the gradient descent and the complexity or convergence time, while also allowing for efficient use of computational resources, especially with GPUs.

11. A too high value of the learning rate means a larger step in the gradient descent computation, which can lead to overshoot the optimized model's parameters and diverge. In consequence, an erratic behavior in the loss value during the train can be observed.

In the other case, a too low value of the learning rate can lead to a slow convergence. Although the loss value will be more stable through the train, results can get stuck in local minima and converge to a non optimized model.

12. The **backpropagation method** apply the *chain rule* to compute the gradient of the parameters. The computation can be seen as follows : $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$
The gradient computation for the parameters in each layer depends on the number of connections in that layer and in the following layers. So considering that the complexity for the gradient computation in one connection in the last layer is $O(1)$, we can estimate

the complexity of backpropagation as

$$O\left(n_x \cdot n_h^1 + n_h^L \cdot n_y + \sum_{l=1}^{L-1} n_h^l \cdot n_{h+1}^l\right)$$

where :

— n_h^l is the number of neurons in the hidden layer l,

— n_y is the number of neurons in the output layer.

— L is the number of hidden layers.

The **naive approach** consists in calculating multiple time the partial derivative for each weight and bias in addition to the previous layers, we can write the complexity as follows :

$$O\left(n_y \cdot n_x \cdot \prod_{l=1}^{L} n_h^l\right)$$

This approach is computationally really expensive and inefficient especially for larger models and datasets. Whereas the backpropagation approach is more efficient due to its linear complexity, making it much more affordable for deep neuron networks.

13. To optimize the procedure, the model architecture could limit the number of layers and neurons per hidden layer. It will reduce the term $\sum_{l=1}^{L} n_h^{l-1} \cdot n_h^l$ and can improve the complexity but it is essential to maintain sufficient capacity to capture the underlying patterns in the data. Also, the model shouldn't have loops or recurrent connections so that it doesn't complexify the computation. A good initialization of the weights can prevent the risk of having exploding gradients. The use of a learning rate that gets smaller as iterations proceeds can help improve convergence and so the complexity.

14. Cross-entropy : $l(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$ Softmax : $\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Let's show that $l = -\sum_{i=1}^{n_y} y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right)$

$$\begin{aligned} l(y, \hat{y}) &= -\sum_{i=1}^{n_y} y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_{k=1}^{n_y} e^{\tilde{y}_k}}\right) \\ &= -\sum_{i=1}^{n_y} y_i \left[\tilde{y}_i - \log\left(\sum_{k=1}^{n_y} e^{\tilde{y}_k}\right)\right] \\ &= -\sum_{i=1}^{n_y} y_i \tilde{y}_i + \sum_{i=1}^{n_y} y_i \times \log\left(\sum_{k=1}^{n_y} e^{\tilde{y}_k}\right) \end{aligned}$$

But $\sum_{i=1}^{n_y} y_i = 1$ parce que $y_i = \begin{cases} 0 & \text{si } i \neq C^*, \\ 1 & \text{si } i = C^* \end{cases}$ with C^* being the correct class.

So we have :

$$= -\sum_{i=1}^{n_y} y_i \tilde{y}_i + \log\left(\sum_{k=1}^{n_y} e^{\tilde{y}_k}\right)$$

15.

$$\begin{aligned}\frac{\partial l}{\partial \tilde{y}_i} &= \frac{\partial \left[-\sum_{i=1}^{n_y} y_i \tilde{y}_i + \log\left(\sum_{k=1}^{n_y} e^{\tilde{y}_k}\right)\right]}{\partial \tilde{y}_i} \\ &= -y_i + \frac{e^{\tilde{y}_i}}{\sum_{k=1}^{n_y} e^{\tilde{y}_k}} \\ &= -y_i + \text{SoftMax}(\tilde{y})_i\end{aligned}$$

$$\nabla_{\tilde{y}} l = \begin{pmatrix} \frac{\partial l}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}} \end{pmatrix} = \begin{pmatrix} \text{SoftMax}(\tilde{y})_1 - y_1 \\ \vdots \\ \text{SoftMax}(\tilde{y})_{n_y} - y_{n_y} \end{pmatrix} = \hat{y} - y$$

16. Let's start by calculating $\nabla_{W_y} l$. According to the *chain rule*, 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 $\tilde{y}_k$

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

Thus, the gradient of $\tilde{y}_k$ with respect to W_{ij}^y is :

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

Now, we can combine this equation with the result of the question 15 to have :

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

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

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

17.

— Let's compute $\frac{\partial l}{\partial \tilde{h}_i}$ with the chain rule :

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

First,

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

With the answer to the previous questions, we get :

$$\frac{\partial l}{\partial h_k} = \sum_{j=1} \frac{\partial l}{\partial \tilde{y_j}} \frac{\partial \tilde{y_j}}{\partial h_k} = \sum_{j=1} (\hat{y}_j - y_j) \cdot W_{y,j,k}$$

Finally,

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

So, the gradient $\nabla_{\tilde{h}} l$ can be seen as :

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

— $\nabla_{W^h} l$ is a matrix as follows

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

We know that $\frac{\partial l}{\partial h_k} = (1 - h_k^2)(\sum_j \delta_j^y W_{y,j,k}) = \delta_k^h$. So let's compute the other term $\frac{\partial \tilde{h}_k}{\partial W_{h,i,j}}$:

$$\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h} = \frac{\tilde{h}_k = \sum_{j=1}^{n_x} W_{k,j}^h x_j + b_k^h}{\partial W_{i,j}^h} = \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

So :

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

— Finally, still from the *chain rule* we can get,

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

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

1.2 Implementation

1.2.1 Forward and backward manuals

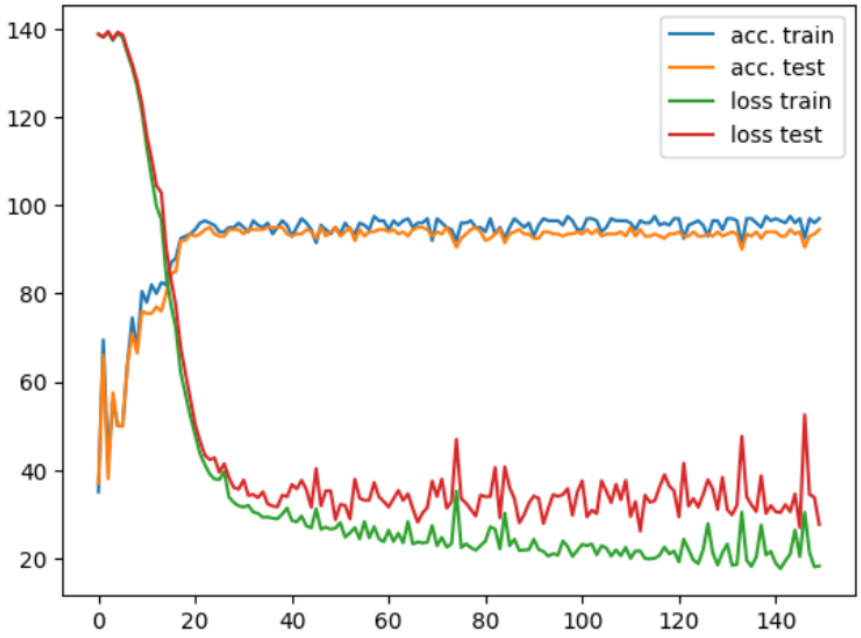

FIGURE 1 – Neural Network's Accuracy and loss for train and test sets

The figure 1 shows the accuracy and loss curves for the train set and the test set. We can see an accuracy of approximately 100% which is reached from approximately 20 rounds. However, we can observe that the loss is still converging especially in the test where the optimal value tend to diverge from the train loss. Besides its erratic behavior, we can understand that the model isn't still 100% performent. It still needs little improvements.

1.2.2 Simplification of the backward pass with torch.autograd

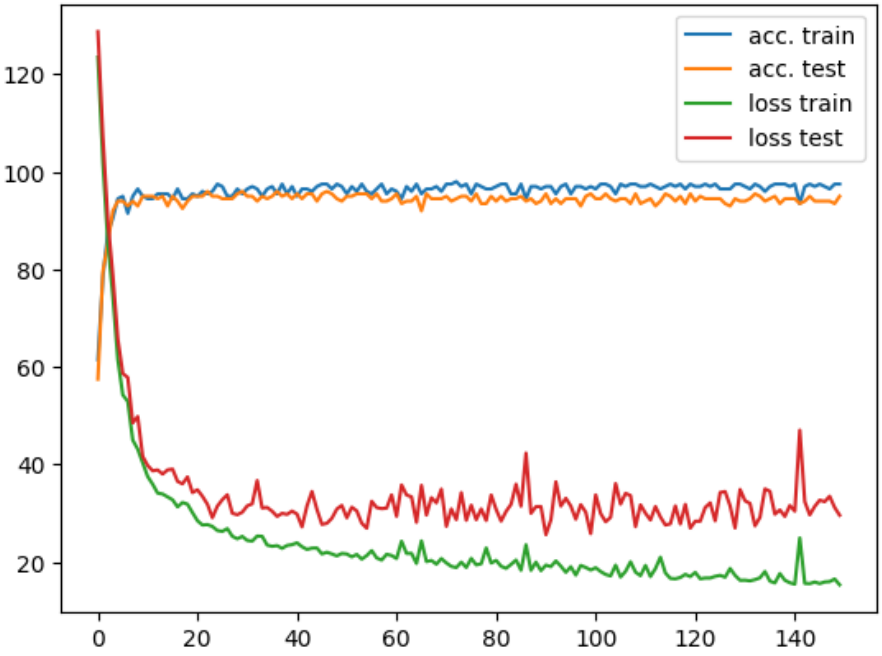

FIGURE 2 – Neural Network's Accuracy and loss for train and test sets with torch.autograd

In this part, we obtained similar results then the previous part. We notice that it converges more quickly than before.

1.2.3 Simplification of the forward pass with torch.nn

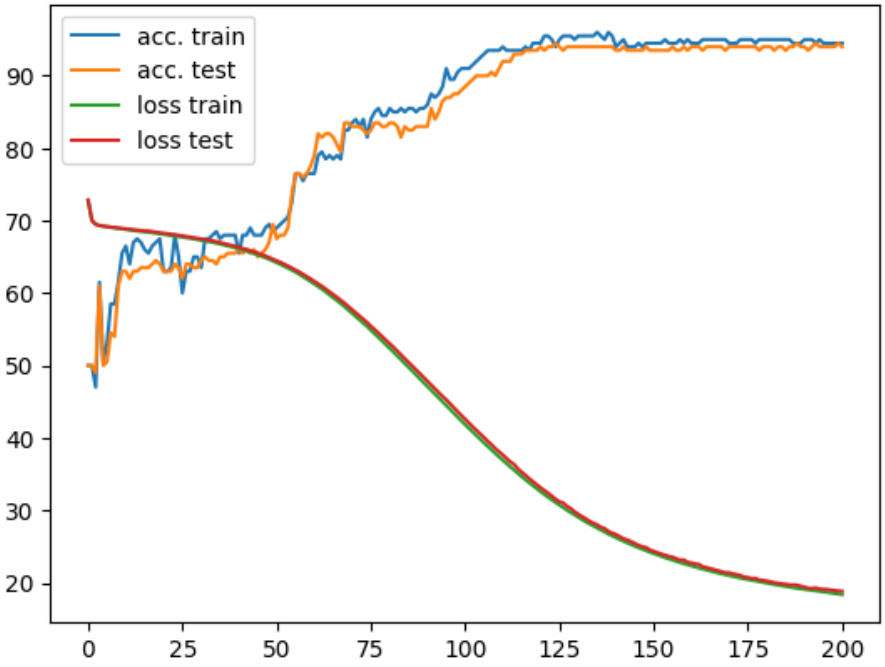

FIGURE 3 – Neural Network's Accuracy and loss for train and test sets with torch.nn

In this case, we observe that the learning takes longer compared to the previous implementations. The maximum performance is reached at 115 epochs. But the losses curves show that the process is more stable than the manual one.

1.2.4 Simplification of the SGD with torch.optim

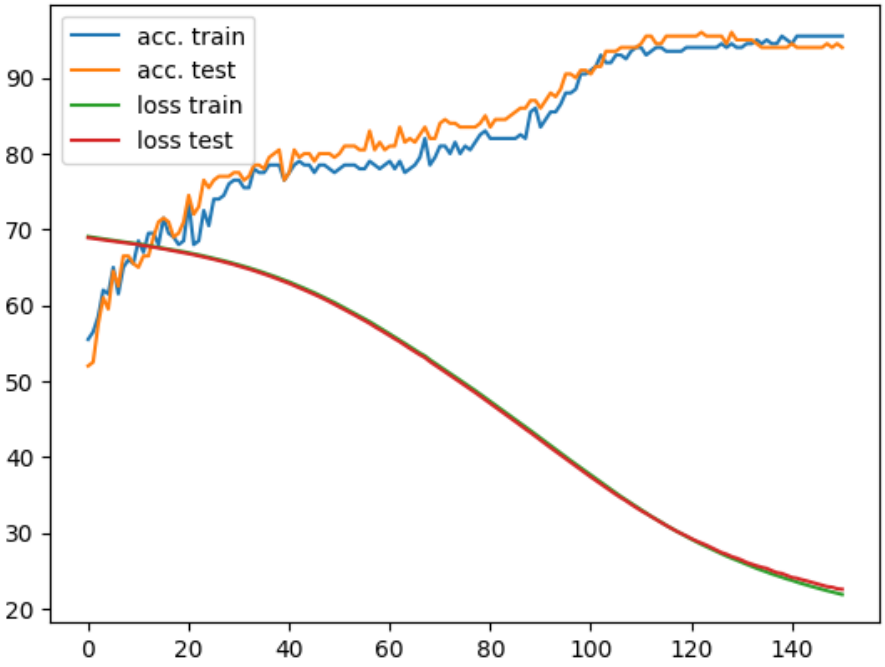

FIGURE 4 – Neural Network's Accuracy and loss for train and test sets with torch.optim

With torch.optim, we observe that the results is almost the same as when we only used torch.nn.

1.2.5 Application on MNIST

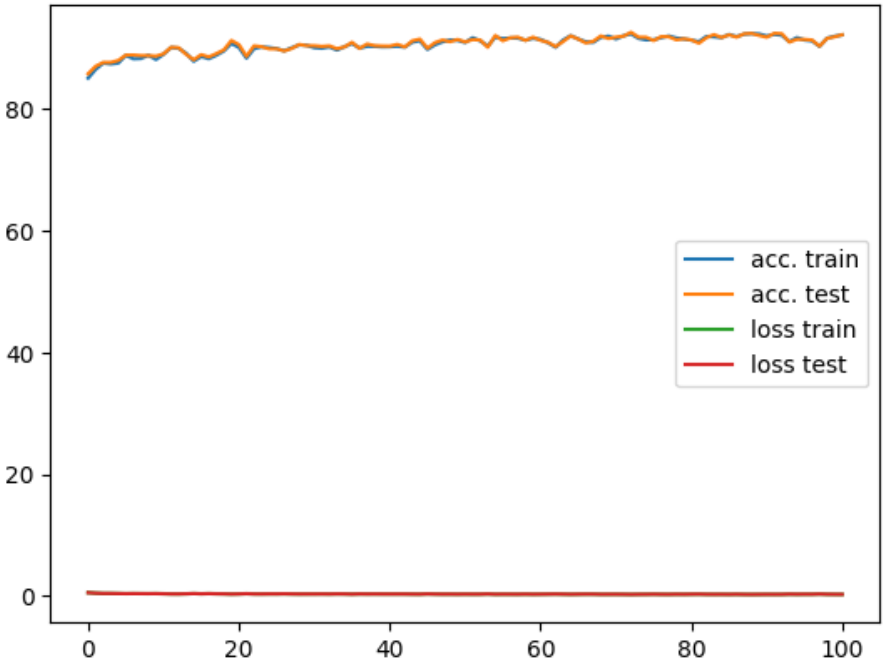

FIGURE 5 – Neural Network's Accuracy and loss for train and test sets with torch.optim

The application of our model on the MNIST dataset preforms extremely well.

1.2.6 SVM

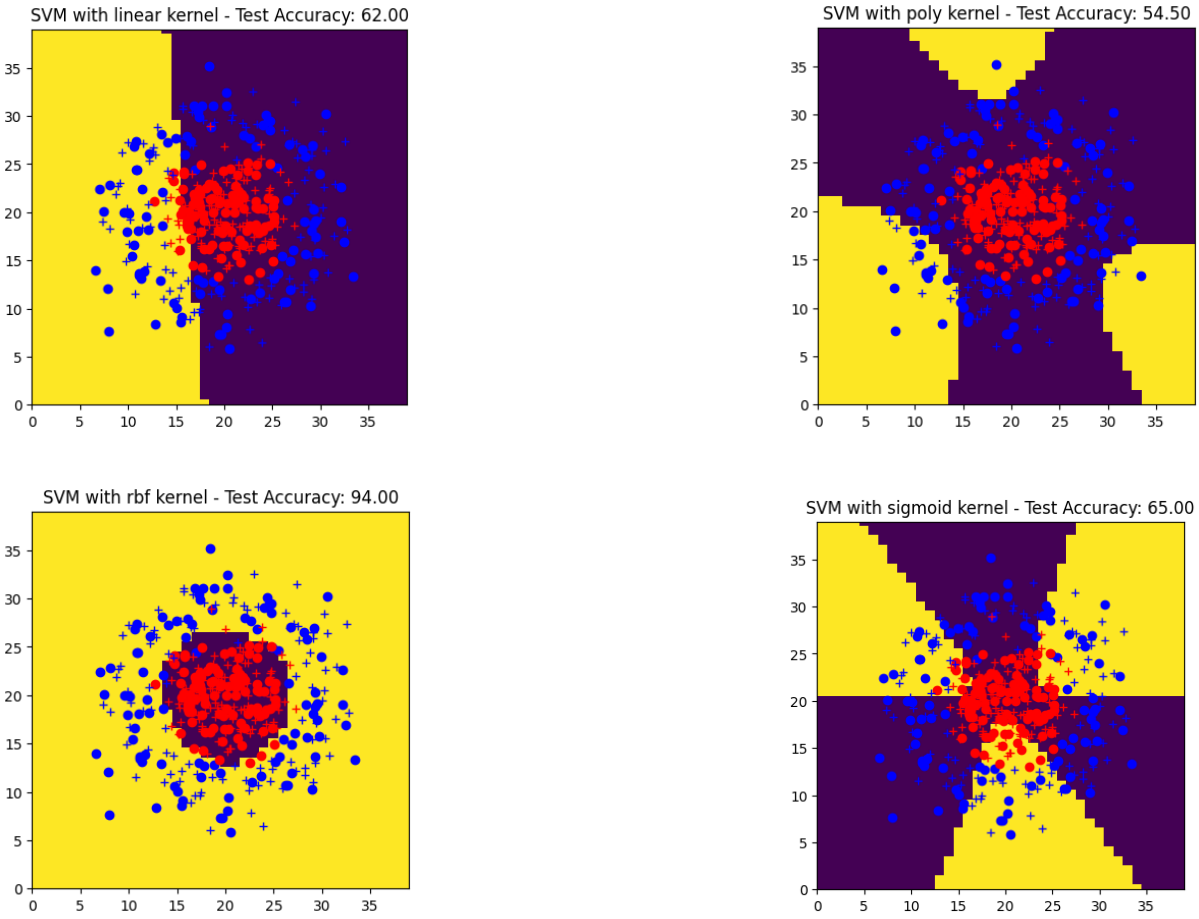

FIGURE 6 – Comparison of Different SVM Models

Obviously, the linear SVM doesn't work with circular data. A linear SVM needs that the data be linearly separable. We observe in the figure 6 that the best SVM kernel is the rbf. It's the best choice because it adapts to the non-linear structure, creating decision boundaries that fit circular or spherical data better than the other kernels.

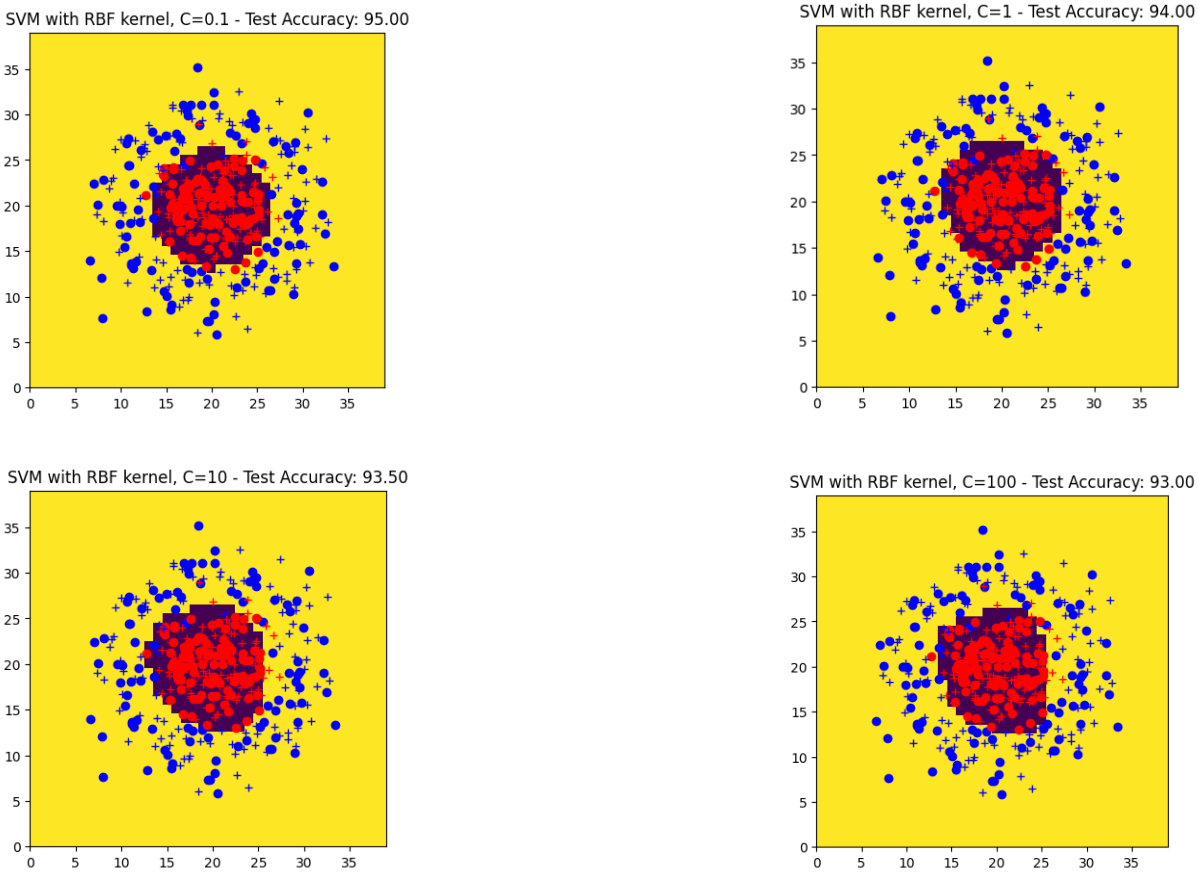

FIGURE 7 – Comparison of the results with different values of C

As we can see in the figure 7, as C increases, test accuracy gradually decreases. When C is large, the SVM try to minimize the error, which leads to a smaller margin. A smaller C allows or more misclassified points and a larger margin. This leads to a simpler model with a smoother decision boundary

1.3 Conclusion

We explored the theoretical aspects of perceptrons, activation functions, and loss functions, followed by hands-on implementation of forward and backward passes with backpropagation. Utilizing PyTorch's autograd and built-in layers simplified our model development. Training on the MNIST dataset demonstrated the effectiveness of neural networks for classification tasks. T

2 Convolutional Neural Networks

By studying about the architecture of a CNN model, the goal of the practical work is to compute and analyse the different methods used to improve the model performance and prevent overfitting. Starting from preprocessing techniques such as normalization or data augmentation, we explore the integration of adaptative tools within the architecture like the dynamic learning rate or the dropout to ultimately develop a critical perspective on these methods.

2.1 Introduction to convolutional networks

1. Let's name $x_{\text{out}}, y_{\text{out}}, z_{\text{out}}$ as our input size and $x_{\text{in}}, y_{\text{in}}, z_{\text{in}}$ as our output size, with z_{in} the number of convolution filters and z_{out} the number of canals. We can find the input size through these equations representing a convolution filter's output :

$$\begin{cases} x_{\text{out}} = \left\lfloor \frac{x_{in}-k+2p}{s} + 1 \right\rfloor \\ y_{\text{out}} = \left\lfloor \frac{y_{in}-k+2p}{s} + 1 \right\rfloor \end{cases}$$ with p as the padding, s as the stride and kernel size k

By solving this equation, with our single convolution filter making $z_{out} = z_{in}$, we can determine the number of weights as

$$((z_{in} \times k \times k) + biais) \times z_{\text{out}}$$

If a fully connected layer were to produce an output in these conditions, every input neurons will be connected to every output neurons in this one layer which results in $(x_{in} \times y_{in} \times z_{in}) \times (x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}})$ weights for the input side and the output side. Thus, considering the addition of a bias for each output neuron, the number of parameters to learn would be :

$$(x_{in} \times y_{in} \times z_{in}) \times (x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}}) + (x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}})$$

2. Convolution layers would create a more performent model due to its ability to detect local spatial patterns and to combine them to have more complex local patterns as desired. Whereas the use of fully connected layers with images will involve a more amount of parameters, one for each pixel of the image. Not only its architecture would be computationally really heavy and expensive but this fully connected model would also struggle with associations of patterns next to convolution layers.

However, the local pattern focus the convolution layers may influence negatively their understanding of global concepts like the relations between distant features which can be crucial in the model's interpretation and robustness.

3. Spatial pooling is used to reduce the dimensions of feature maps to have a lower computational cost as we go deeper into the layers of the model which complexifies the feature maps. By summarizing a local region with a pooled value which can be the average, the maximum or the minimum, not only the exact location of the feature becomes less critical but it also make the algorithm emphasizes the most relevant information, making the model more generalized.

4. We can use all or part of the layers of a convolutional network on an input image that is larger than the initially planned size. Indeed, convolutional layers can generally handle larger input sizes as long as the spatial dimensions allow the convolution and pooling operations to be performed. In both case, the model's primary function will be executed, which is the detection of global features through local analysis, so the size of the input doesn't really matter.

However, if the network has fully connected layers following the convolutional and pooling layers, these layers will likely require modification. Fully connected layers expect a specific input size, corresponding to the flattened output of the preceding layers. In that case, it would be needed to adjust the architecture of the network by adding input neurons in the fully connected layers or by adding adding more pooling layers by the end of the convolutional and pooling layers.

5. Fully-connected layers can be analyzed as convolutional layers by observing the way they operate on input data. In a fully-connected layer, each neuron is connected to every neuron of the previous layer, effectively applying a linear transformation to the input data. This operation can be represented as a convolution with a kernel size equal to the input size and a stride of 1. Indeed, the weights of the fully-connected layer, they can be seen as a convolutional kernel that spans the entire input.

6. If fully connected layers are replaced by equivalent convolutions, the network can process input images of varying sizes. The output will then be a spatial map of scores, rather than a single vector. This score map will have dimensions proportional to the input's size, such as $(r+1) \cdot (c+1)$ reflecting additional input rows and columns.

7. The receptive field of a neuron in a convolutional layer is the region of the input image that influences the output of that neuron. Therefore, in the first convolutional layer, each neuron "sees" only a small patch of the input image, defined by the kernel size.

However, neurons in the second layer "see" a larger region of the input, as each of them is connected to multiple neurons in the first layer, whose receptive fields are already small regions of the original image. For instance, with a kernel of 3×3 in both layers and no strides or padding, the receptive field of neurons in the second layer would be 5×5 pixels.

And as we move deeper, each successive layer combines information from increasingly larger regions of the original input. The receptive field grows significantly and can match the size of the input image. This allow the model to recognize larger structures and more abstract features like more defined shapes.

2.2 Training from scratch of the model

2.2.1 Network architecture

8. Going back to the equation of question 1 : $\begin{cases} x_{\text{out}} = \left\lfloor \frac{x_{in}-k+2p}{s} + 1 \right\rfloor \\ y_{\text{out}} = \left\lfloor \frac{y_{in}-k+2p}{s} + 1 \right\rfloor \end{cases}$

And from the dataset CIFAR10 and our CNN, we get the values of x_{in}, y_{in} and k which are respectively 32, 32 and 5. By isolating p or s we get $p = 2, s = 1$.

9. This time we want $x_{out} = x_{in}/2, y_{out} = y_{in}/2$. For a pooling condition, we will set $k = 2$ to be able to pool a value from 4 inputs. 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.

10. With the values defined for p and s for the convolutional and pooling layers and a bias of value 1, we can get this table.

Layer Name	**Output Size**	**Number of Weights**
CONV1	$(32 - 5 + 2 \times 2) + 1 = 32$ Output Size : $32 \times 32 \times 32$	$32 \times ((5 \times 5 \times 3) + 1)$ $= 32 \times 76 = 2,432$
POOL1	$(32 \div 2) = 16$ Output Size : $16 \times 16 \times 32$	None
CONV2	$(16 - 5 + 2 \times 2) + 1 = 16$ Output Size : $16 \times 16 \times 64$	$64 \times ((5 \times 5 \times 32) + 1)$ $= 64 \times 801 = 51,264$
POOL2	$(16 \div 2) = 8$ Output Size : $8 \times 8 \times 64$	None
CONV3	$(8 - 5 + 2 \times 2) + 1 = 8$ Output Size : $8 \times 8 \times 64$	$64 \times ((5 \times 5 \times 64) + 1)$ $= 64 \times 1601 = 102,464$
POOL3	$(8 \div 2) = 4$ Output Size : $4 \times 4 \times 64$	None
FC4	1000	$(4 \times 4 \times 64 + 1) \times 1000$ $= 1,025,000$
FC5	10	$(1000 + 1) \times 10$ $= 10,010$

TABLE 1 – Output size and number of weights for each layer.

11. This results in a total of 1,191,170 parameters to train. Given that the dataset consists of 50,000 training samples and 10,000 test samples, totaling 60,000 examples (with 6,000 images per class), caution is warranted regarding the potential for overfitting because the model can memorize the examples rather than learning from them.

12. In comparison to a CNN with 1,191,170 parameters, a BoW model with a SVM classifier typically requires significantly fewer parameters. The BoW model creates a fixed-length vector for each sample, where each element represents the frequency of specific features. For example, a vocabulary of 1,000 visual words would require that each sample is represented by a 1,000-dimensional vector. An SVM learns a hyperplane in this feature space to separate different classes. With a linear kernel, the number of parameters would be equal to N features + 1 (for the bias term), so it would have around 1,000 parameters (plus bias) in our case. Even with a non-linear kernel, the number of parameters is still typically far lower than that of a CNN.

2.2.2 Network learning

14. The main difference in the computation of loss and accuracy in train and data is that train use an optimizer and test doesn't, it uses evaluation and doesn't have a backward process to updata the loss.

16.

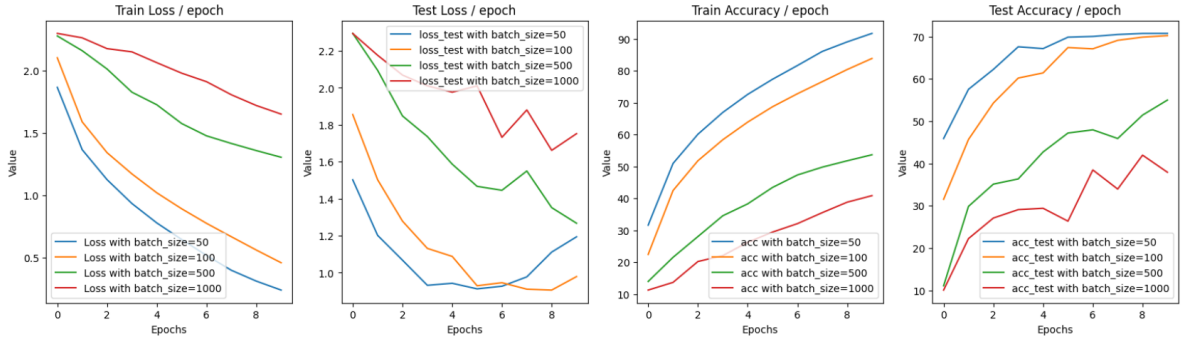

FIGURE 8 – Evolution of the model's performance depending on the variation of the batch size

A smaller batch size often leads to noisier updates due to the limited sample size used to estimate the gradient. This can slow down the convergence but also make the model more robust to variation and prevent overfitting.
With a larger batch size, the gradient updates are more stable and accurate and can accelerate convergence. However, it can computationally more expensive.

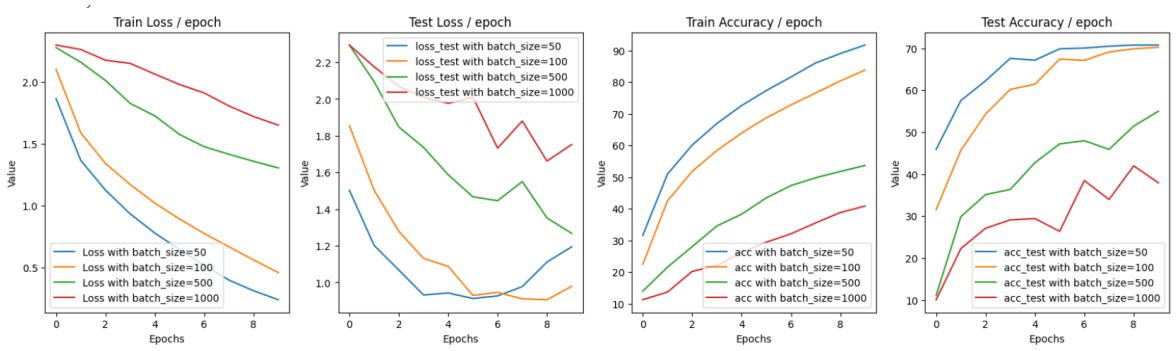

FIGURE 9 – Evolution of the model's performance depending on the variation of the learning rate

A too high value of the learning rate means a larger step in the gradient descent computation, which can lead to overshoot the optimized model's parameters and diverge. In consequence, an erratic behavior in the loss value during the train can be observed.
In the other case, a too low value of the learning rate can lead to a slow convergence. Although the loss value will be more stable through the train, results can get stuck in local minima and converge to a non optimized model.

17. The error at the start of the first epoch, in train and test are due to a random initialization of the different layer's weights. This effect can be seen more drastically in the beginning of the loss in Figures 11 and 12.

18.

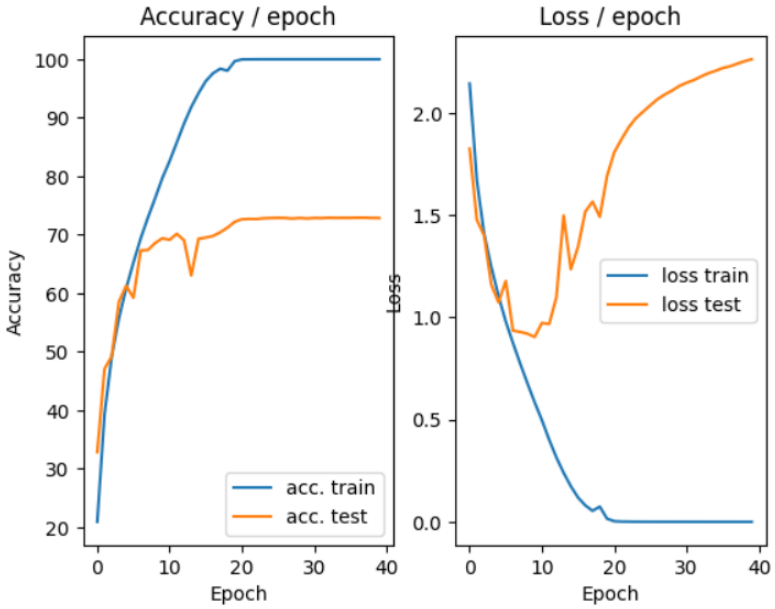

FIGURE 10 – Loss and accuracy per epoch

We can see that the test's outputs aren't accurate as the train's results : the test accuracy is about 70% while the test loss doesn't converge at all, it even diverge. These are signs of overfitting, which means that the model memorized the train set and wasn't able to generalized to unknown data.

2.3 Results improvements

2.3.1 Standardization of examples

19.

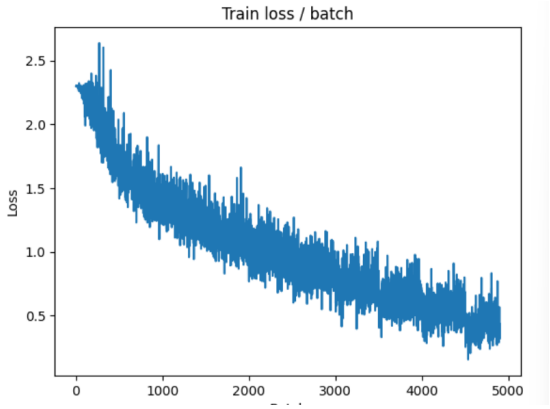

FIGURE 11 – Train loss/batch sans normalisation

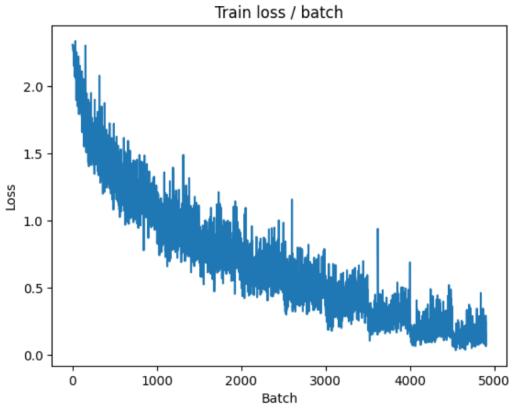

FIGURE 12 – Train loss/batch après normalisation

The data normalization before processing results in a more regular and stable decrease in the loss, as shown in Figure 12. While the loss in Figure 11, without normalization, exhibits a more erratic behavior.

From the scale, we can also observe that the loss in Figure 12 has converged further, reaching a lower final value compared to Figure 11. This is because normalization reduces the amplitude of fluctuations in the loss computation, allowing the model to be more precise and to optimize faster.

20. The normalization's values (mean and std) are calculating using only the training examples in order to avoid introducing any information from the validation set into the training process. This phenomenon is called data leakage. Indeed, it is important that the train set and the validation set are completely independent because it ensures that the model's performance on the validation set remains unbiased, thus preserving its ability to generalize to unseen data.

21. The ZCA normalization can be quite efficient due to its ability to remove correlation between data while maintaining their spatial distribution. It can be more efficient than a classic normalization of the data due to the fact that it will reduce redundancy and make the model focus on unique features, making the model converge faster and be more precise. It will also tend to avoid local minima existence in the loss function making the gradient descent more fluid.

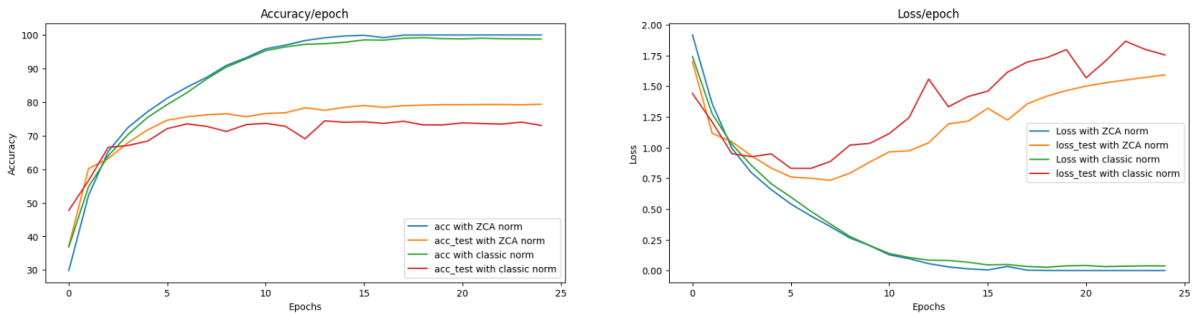

FIGURE 13 – Train and Test Losses and Accuracies after ZCA normalization

Indeed, we can see in the ZCA results in Figure 13 a more stable and faster convergence for both train and test, compared to the classic normalization. This is due to the ability of the ZCA to help the model focus on unique features and therefore accelerate convergence. This help the model get more performant and can occasionally help prevent overfitting.

Indeed, results for the testing set with the ZCA are more precise and less unstable with an accuracy nearly 5% higher than the test accuracy without ZCA and a lower loss than the test loss without ZCA. However, because the loss value in the test is still going higher, we can see that this method is still unable to suppress overfitting.

Another interesting method is the PCA normalization, it functions quite similarly as the ZCA normalization but won't maintain the original data representation. Despite their advantages, they can be computationally really expensive and time consuming in comparison to a classic normalization.

2.3.2 Increase in the number of training examples by data increase

22.

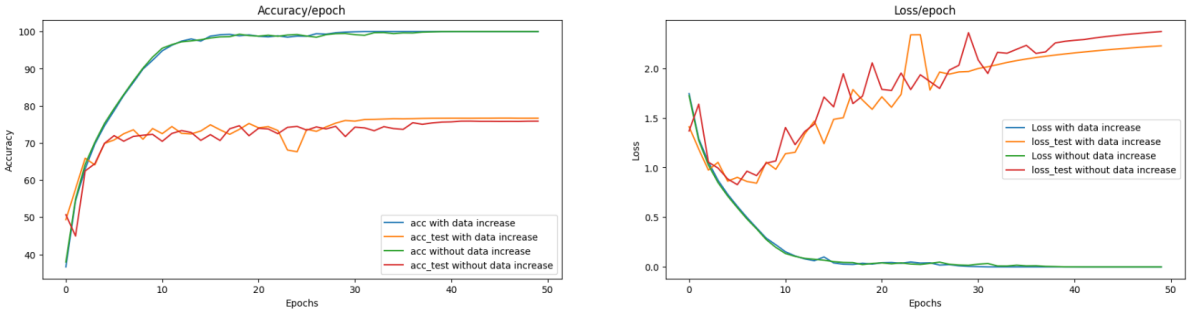

FIGURE 14 – Loss and accuracy for train and test sets with and without data increase

Augmenting the training and testing data artificially by transforming the initial examples helps create a more robust model as it becomes less senstive to specific patterns. This effect can be seen in Figure 14, where the test accuracy with data augmentation converges faster than the test accuracy without it, which only stabilizes around 10 epochs later. This improvement is due to the model's enhanced generalization when trained on more diverse data. However, while data augmentation reduces overfitting, it does not eliminate it entirely, as seen in the test loss values, which still indicate some degree of overfitting less pronounced than without augmentation.

23. The horizontal symmetry approach is suitable for topics like image classification (with an object and a background) where the orientation of the object doesn't really affect the model's output due to its local computation (through CNNs convolutions) to recognize patterns. However, the horizontal symmetry can be inefficient for topics that require an order recognition with a couple of objets next to each other like in visual words recognition where a word flipped won't form the same word.

24. Data Increase has limits. First of all, the addition of diverse transformations to increase the number of examples can be computationally expensive as seen with this ex-

periment. Moreover, data increase doesn't necessarily mean better performance of the model. Even though the goal is to have a more generalized and robust model by making it optimized on complex features, some transformations might introduce features that are unnatural, potentially leading the model to learn false patterns.

25. Other common methods to increase datasets are rotating images by a few degrees to make the model less sensitive to object orientation. Also slightly zooming in or out to simulate changes in distance can be a robust approach.Translating the image can help build spatial invariance. Adjusting brightness, contrast, saturation, or color variety regularly or irregularly on the image can highly influence the models robustness by varying its inputs. Let's not forget about a fundamental pre-processing method, the Gaussian noise that simulate real-world acquisition conditions. All theses methods can help in preventing overfitting due to its goal to generalize the model.

We tested the gaussian noise approach :

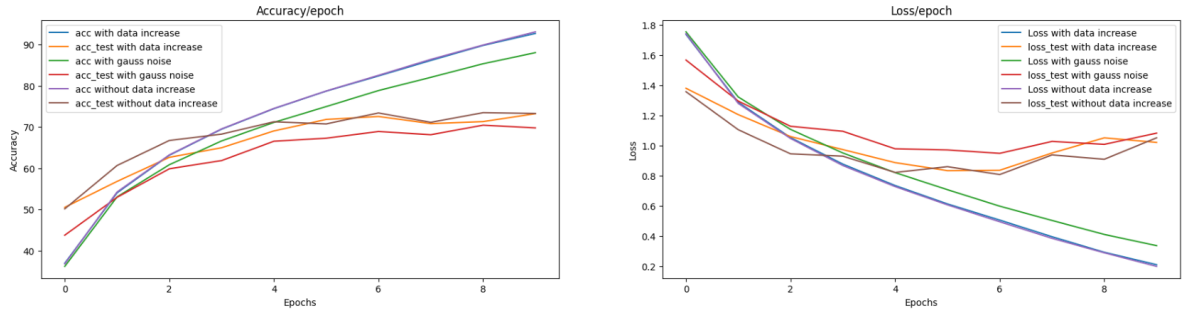

FIGURE 15 – Outputs of the model with the Gaussian noise, data increase and no augmentation

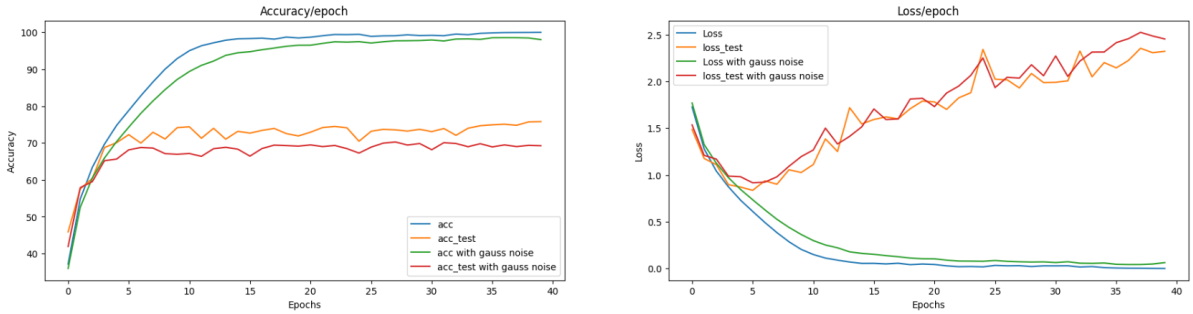

FIGURE 16 – Outputs of the model with Gaussian noise with more epochs

By adding Gaussian noise with a mean of 0 and a standard deviation of 0.1 to our dataset, we see a less performent model with less optimal values at the end of the iterations even though the loss and accuracy curves show a same pattern during the iterations due to the use of the same architecture parameters (lr and batch size).

This lower robustness with Gaussian noise makes sense considering that the data has more noise and less shaped patterns. So this method, especially with a mean of 0 and

a std of 0.1 for the Gaussian noise, isn't the best way to enhance the robustness of the model. The overfitting confirms it. It should be interesting to explore other data increase algorithms.

2.3.3 Variants on the optimization algorithm

26. The dynamic exponential decrease of the learning rate is efficient to help the model converge more accurately by avoiding for example local miminas. In Figure 17, we see that the train outputs with the learning rate decrease converge faster towards the 13th epoch, while the train output with a fixed learning rate reach the optimal values towards the end of the iterations. This happens due to the schedule capacity to make the model converge faster with initial big steps.

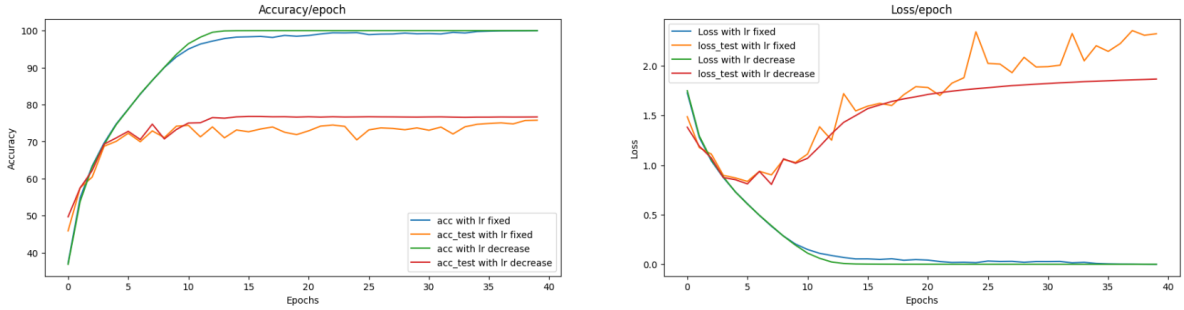

FIGURE 17 – Outputs of the model with the addition of a dynamically decrease of the learning rate

Moreover, the difference between the outputs with and without the schedule is only seen from approximately the 10th epoch because the decrease is done exponentially so it is more efficient when it comes closer to the optimal values.

Regarding the test loss and accuracy with the schedule, we see a slightly better performance with a higher accuracy and a lower loss compared to the outputs without the schedule. However, it still isn't a win because, from the shape of the test outputs curves with or without the schedule, we can still deduce the presence of overfitting.

27. An exponential learning rate schedule gradually decreases the learning rate over time, which can improve learning by allowing the model to take larger steps initially (facilitating faster convergence) and smaller, more precise steps later (reducing the risk of overshooting the optimal solution). Starting with a high learning rate not only helps the model get closer to the global minima faster but also enables it to avoid local minima by taking large steps initially. Additionally, this schedule helps to stabilize learning toward the end, allowing the model to fine-tune its weights more precisely for optimal convergence.

2.3.4 Regularization of the network by dropout

29. The dropout method consists in dropping out some neurons between iterations and in our case, on an additionnal layer between FC4 and FC5, to make sure that neurons

don't memorize features. The dropout happens for a percentage of neurons that is defined by the p_value.

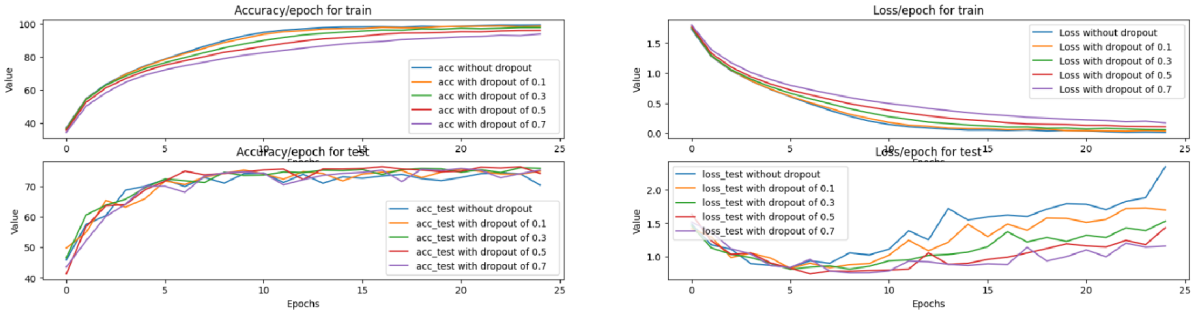

FIGURE 18 – Loss and Accuracy with different dropout probability values

We tested the dropout method within our CNN model with different probability of dropout (0.1, 0.3, 0.5 and 0.7). Figure 18 show the results in comparison to the outputs without dropout. As we can see, as we increase the dropout probability, the loss for the test set has a lower value showing less overfitting. It's because the more we variate the number of neurons dropout, the more the model will be forced to generalize and adapt during training preventing memorization and overfitting.

We can also observe that as the dropout's probability value rise, the train accuracy and loss show slightly less optimal values. Indeed, higher probability of dropout will make the algorithm too different between iterations and that will require more iterations of train to generalize accurately and therefore to converge.

30. Regularization represents techniques that constrain a model's learning capacity in order to prevent overfitting, an issue where the model memorizes training data too precisely and struggles to generalize to new data. These methods function by introducing penalties to the loss function or by modifying the training data or model structure.

31. The random deactivation of neurons make neurons less dependent on certain features or on other neurons outputs. This ensure that each neuron contributes individually and not together to the results.

The introduction of dropout creates different versions of the model, it can be seen as a way of adding noise into the network helping the model to not fit too precisely to defined features or outputs. This can also be seen as a interior version of the data augmentation.

32. As we raise the parameter of dropout which is the probability of neuron's dropout in that layer, we raise the probability of having different versions of the model through the iterations making it more constraint to generalize and get more robust.

2.3.5 Use of batch normalization

Batch normalization is a technique used to make neural networks more stable and to accelerate training by normalizing the inputs of each layer with the average value and

variance of the batch. Indeed, this normalization tends to reduce the need for careful weight initialization allowing higher learning rates. Moreover, this method can act as a regularizer by adding noise to the data through batch variations, which often leads to less overfitting due to the model's need to generalize. Furthermore, by normalizing the inputs to each layer, it prevents extreme values that can lead to issues like gradient explosion or vanishing gradients.

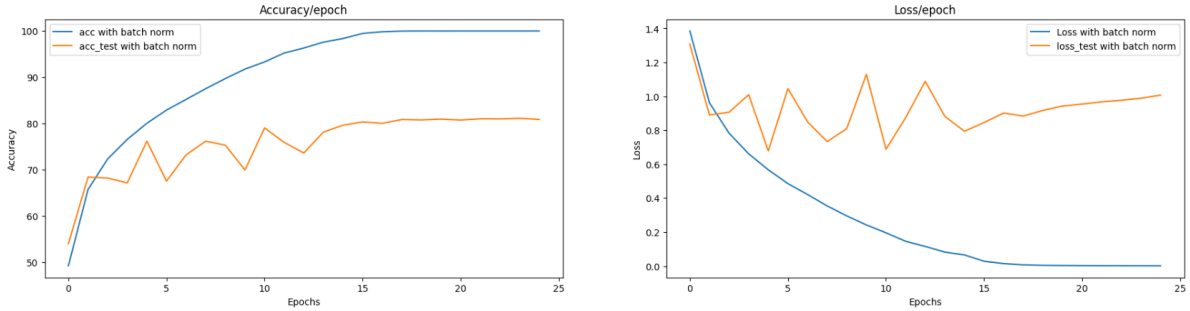

FIGURE 19 – Train et Test Loss and Accuracy with Batch Normalization

The Figure 19 shows quite similar performance results as the dropout method with a probability of 70%. We see a slightly higher accuracy with a value approaching 80% while the dropout converge at nearly 75%. This happens because this method keeps stable all the neurons activations while the dropout method deactivate randomly neurons. We also see a less erratic behavior in the test loss and accuracy curves, as mentioned above with a more stable result with batch normalization.

2.4 Conclusion

In this study, we examined various techniques, implementation methods, advantages, and limitations of algorithms used to enhance CNN performance. For our context, Batch Normalization demonstrating the best results in addressing overfitting, which was our main concern. However, no single method is a universal solution, each algorithm must be tailored to the specific model and dataset to achieve optimal performance. This adaptability underscores the importance of evaluating each approach in the context of the network's goals and constraints.

3 Transformers

In this work, we explore concepts of transformer models, including Self-Attention, Multi-Head Self-Attention, and the Transformer Block structure, which incorporates elements like layer normalization and residual connections. We then go deeper into the Vision Transformer (ViT) model, examining its unique components and leading us to experiment it on the MNIST dataset to evaluate the impact of different parameters on model performance.

3.1 Self-Attention

The main feature of self-attention is its ability to capture long-range dependencies between elements in a sequence. Unlike convolutions, which apply a local filter with a fixed receptive field, self-attention allows each element in the sequence to interact directly with all other elements. This makes it particularly effective for modeling global relationships.

The primary challenge of self-attention lies in its high computational and memory cost, which grows quadratically with the sequence length N. Specifically, calculating attention weights between each pair of elements in the sequence requires forming an attention matrix of size $N \times N$, which can become infeasible for very large sequences or high-resolution images.

For self-attention with an embedding dimension C, we proceed as follows :

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

where W_Q, W_K, and W_V are projection matrices of size $C \times C$.

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{C}}\right)$$

$$\text{Output} = \text{Attention} \cdot V \cdot W_O$$

where W_O is a learned weight matrix for the final projection back to the input dimension.

3.2 Multi-head self-attention

Multi-Head Self-Attention divides the embedding dimension C among multiple heads :

$$\text{head\_dim} = \frac{C}{\text{num\_heads}}$$

For each head i, we compute *queries* Q_i, *keys* K_i, and *values* V_i :

$$Q_i = XW_{Q_i}, \quad K_i = XW_{K_i}, \quad V_i = XW_{V_i}$$

The *attention score* for head i :

$$\text{Attention}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{\text{head\_dim}}}\right) V_i$$

Finally, we **concatenate** the outputs and apply a projection W_O :

$$\text{Output} = \text{Concat}(\text{Attention}_1, \ldots, \text{Attention}_{\text{num\_heads}})W_O$$

3.3 Transformer Block

A Transformer Block consists of two main components : Multi-Head Self-Attention and a Multi-Layer Perceptron (MLP).

3.3.1 Multi-Head Self-Attention

The input X is transformed through Multi-Head Self-Attention, which can be expressed as :

$$\text{MHSA}(X) = \text{Concat}(\text{Attention}_1, \ldots, \text{Attention}_{\text{num\_heads}})W_O$$

3.3.2 Layer Normalization and Residual Connection

After applying the attention mechanism, a residual connection is added :

$$Y_1 = X + \text{MHSA}(X)$$

Then, layer normalization is applied :

$$Z_1 = \text{LayerNorm}(Y_1)$$

3.3.3 Multi-Layer Perceptron (MLP)

The output from the first layer normalization is passed through the MLP :

$$\text{MLP}(Z_1) = \text{GELU}(Z_1 W_1 + b_1)W_2 + b_2$$

where W_1 and W_2 are weight matrices, and b_1 and b_2 are biases.

3.3.4 Second Residual Connection and Layer Normalization

Similar to the attention mechanism, the output of the MLP is added to the input Z_1 :

$$Y_2 = Z_1 + \text{MLP}(Z_1)$$

$$Z_2 = \text{LayerNorm}(Y_2)$$

3.3.5 Summary of the Transformer Block

The final output of the Transformer Block can be expressed as :

$$\text{Output}_{\text{Transformer}} = \text{LayerNorm}\left(X + \text{MHSA}(X) + \text{MLP}(Z_1)\right)$$

3.4 Full ViT model

3.4.1 Class token

A Class token is a unique token that is appended to the input sequence in order to compile data from the complete sequence for classification. By acting as a representative feature vector for the picture, it enables the transformer model to generate predictions using the data that the self-attention layers have processed. The class token, which is often appended to the input patch embeddings, is the output used for the final classification head.

3.4.2 Positional Embedding

The PE are used to allow the model to incorporate information about the relative or absolute position of the patches in the input sequence. This is important for tasks where the order of inputs affects the interpretation of the data, like images. Sinusoidal encoding or learned positional embeddings can be utilized, as they enable the model to recognize spatial relationships among patches.

3.5 Experiment on MNIST

3.5.1 Embed dimension

We can observe, in the figure 20, that the higher the embedding dimension is, the lower is the loss. Thats because a higher *embed_ dim* allows the model to capture richer feature representations for each token. We can see with the table 2, that a small *embed_ dim* results in a faster computation, but limit the accuracy of the model. Increasing *embed_ dim* can improves performance, but increases the model's size and took more time to train.

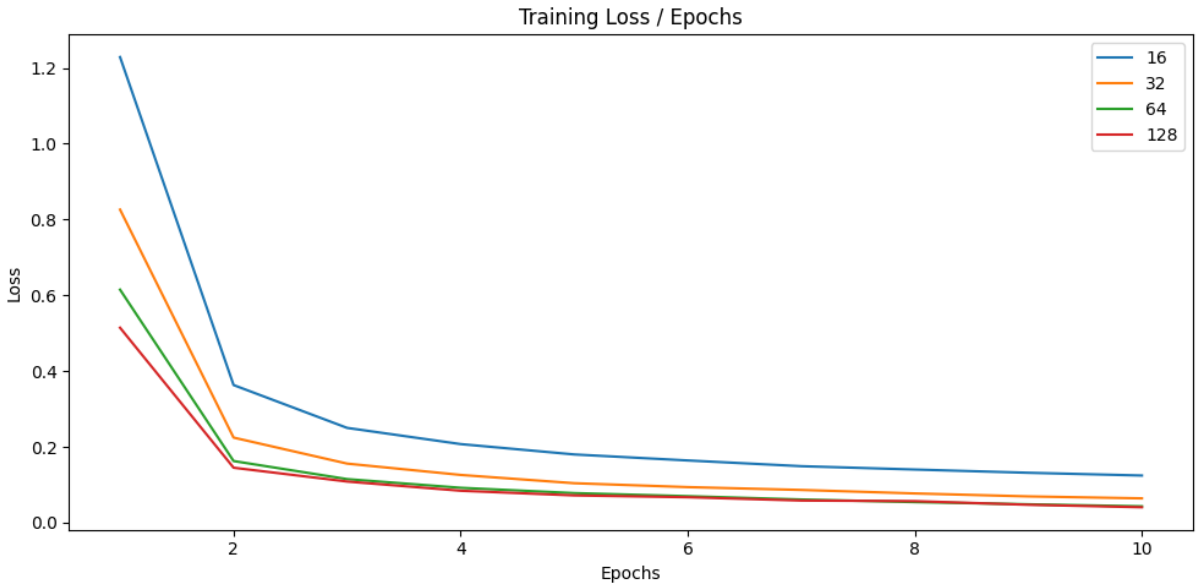

FIGURE 20

Embed dimension	Test Accuracy (%)
16	95.67
32	97.2
64	98.06
128	97.48

TABLE 2 – Test Accuracy for different Embedding dimensions

3.5.2 Patch size

In the figure 21 the *patch_size* 16 leads in a bad performance. We know that smaller is the *patch_size*, and higher is the accuray on complex image. With larger values, it cost less memory, but can lose critical details and can lead to lower performance.

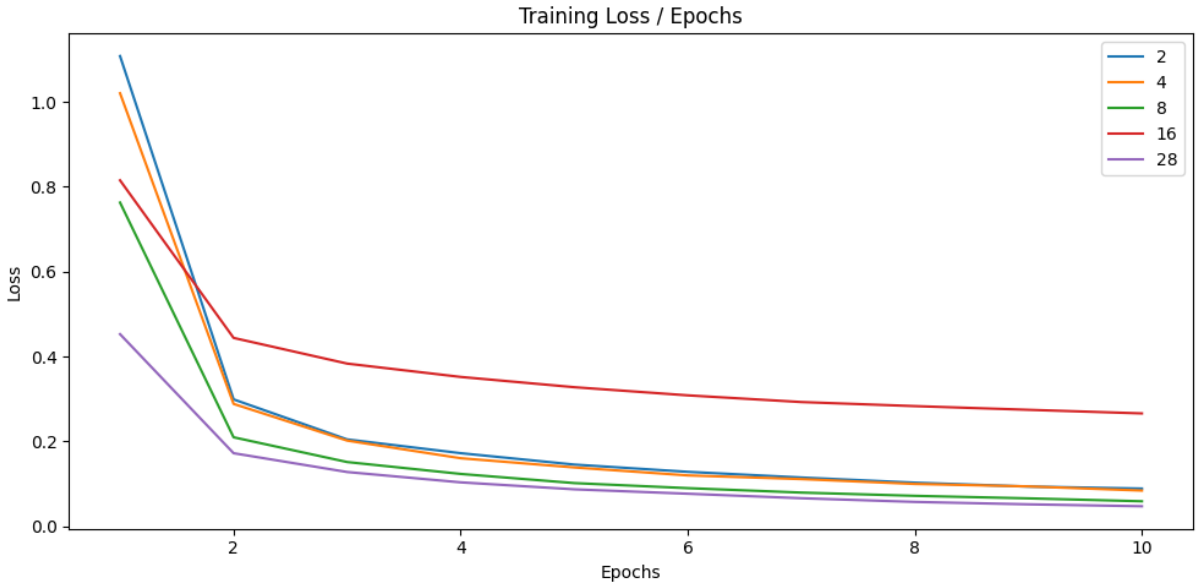

FIGURE 21

Patch size	Test Accuracy (%)
2	97.16
4	96.75
8	97.65
16	90.2
28	97.23

TABLE 3 – Test Accuracy for different Patch sizes

3.5.3 Number of transformers blocks

The number of transformer blocks impacts the complexity of the model. However, additional blocks increase computation time an memory usage. Fewer blocs might result

in a short training but can lead to a bad performance. Testing values can help to determine the model depth needed for optimal performance.

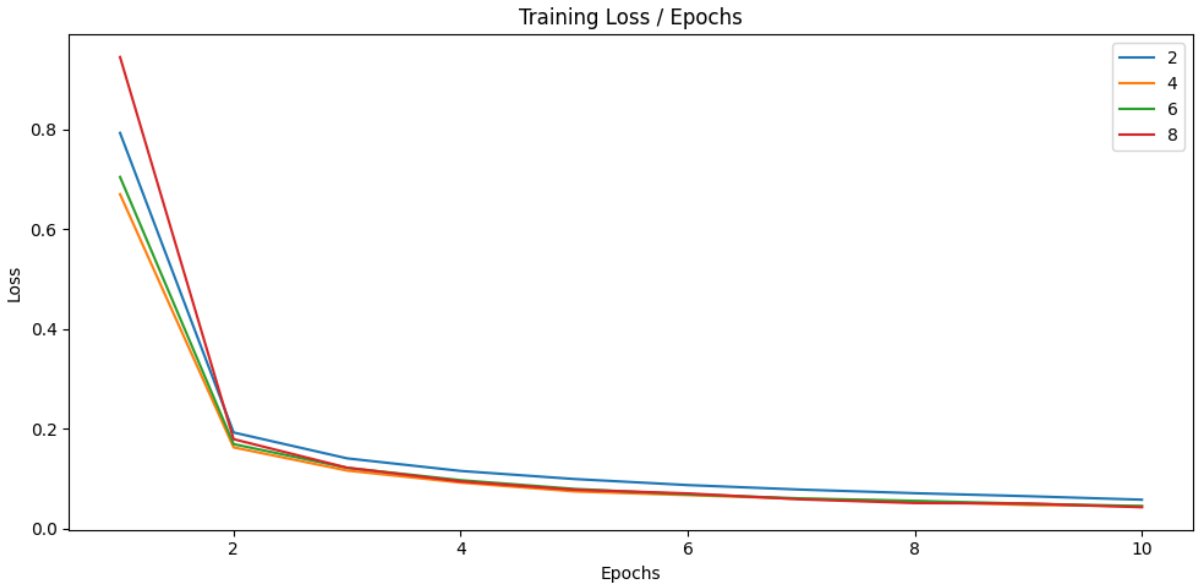

FIGURE 22

Nb blocks	Test Accuracy (%)
2	97.39
4	97.53
6	97.34
8	97.54

TABLE 4 – Test Accuracy for different Number of transformers blocks

The transformer's self-attention mechanism has a complexity of :

$$O(N^2 \cdot D)$$

où :

— N is the number of tokens (determined by patch size).

— D is the embedding dimension.

By increasing the *patch_size*, we reduce N, making the model more efficient.

3.6 Larger transformers

To use a ViT model from the *timm* library on MNIST images with a resolution 28 x 28, you will need to resize it and with 3 color channels, while MNIST images are grayscale. The *vit_base_patch16_224* model expects us to give 224 x 224 images. Using 28 x 28 images on this model would raise an error.

CNNs are more flexible regarding input image size. As long as the image has a valid channel size, a CNN can handle smaller resolutions like 28×28 without structural issues.

3.7 Conclusion

This study highlights the adaptability of transformer architectures in vision tasks. The ViT model demonstrated how effectively it can handle image data, and the experiments on MNIST revealed how each parameter influences model behavior. While transformers offer strong performance and flexibility, they require careful tuning of structural elements to best suit the specific data and task requirements. Plus, it's important to consider the limits of these transformers, such as their tendency to do overfitting on small datasets and the requirement of high computational resources, which can impact their feasibility in practical applications.