RDFIA

Basics on deep learning for vision

BOUCHOUCHI Nour et FAURE Guillaume

Septembre-Octobre 2023

Contents

I. Introduction to Neural Networks (a-b) 3
1. Theoretical foundation . . . 3
2. Implementation . . . 10
   a. Forward and backward manuals . . . 10
   b. Simplification of the backward pass with torch.autograd 13
   c. Simplification of the forward pass with torch.nn layers 13
   d. Simplification of the SGD with torch.optim . . . 14
   e. MNIST application . . . 14
   f. Bonus: SVM . . . 15

II. Convolutional Neural Networks (c-d) 16
1. Introduction to convolutional networks . . . 16
2. Training from scratch of the model . . . 17
   a. Network architecture . . . 17
   b. Network learning . . . 19
3. Results improvements . . . 23
   a. Standardization of examples . . . 23
   b. Increase in the number of training examples by data increase . . . 25
   c. Variants on the optimization algorithm . . . 27
   d. Regularization of the network by dropout . . . 29
   e. Use of batch normalization . . . 31

III.Transformers (e) 32
1. Implementation of a transformer . . . 32
2. Larger transformers . . . 39

I. Introduction to Neural Networks (a-b)

1. Theoretical foundation

1. The train set is used to fit the model (the model learns the data). The validation set is used to evaluate the model, it is frequently used in the process as we use it to fine-tune the model hyper-parameters. Finally, the test set is used to do the final evaluation of a model.

2. The number of examples N has an influence on the generalisation ability of the model. Indeed, a greater number of examples allows to avoid over- fitting the data (with a small N, there is a risk to stick to the data and thus not being able to generalize to new examples). Moreover, having more examples can improve the stability as it will be less sensitive to noise in the data. However, a too large amount of data can represent a challenge in terms of resources (memory and processing power for instance).

3. Adding activation functions between linear transformation allows to introduce non-linearity. Indeed, otherwise, the gradient would not change through the network which would make the adjustments of the network parameters more challenging. Moreover, a lot of problems are not linear, thus a non-linear function is more adapted to learn and approximate more complex phenomenons.

4. The sizes :

- n_x : dimension of the input example (number of attributes). In practise, it is thus determined by the dimension of the input.
- n_h : number of neurons in the hidden layer (it corresponds to the dimension of the intermediate representation of x). This size is chosen by experimenting with different values. We generally start with a small number and gradually increase it in order to fit the complexity needed while being cautious about overfitting.
- n_y : dimension of the output (it corresponds to the number of classes). This number depends on the task, for instance, in binary classification it would be equal to 2, in multi-class problem it depends on the number of classes.

5. The vector $\hat{y}$ represents the predicted value obtained with the neural network (its output), while the vector y represents the ground truth. The

goal is to have a ŷ as close as possible to y. For instance, in a binary classification context, ŷ would could represent the probability of belonging to both classes and y could be a one-hot vector. The objective is to have a ŷ as close to y as possible. Different metrics are used in order to quantify the difference between these two quantities such as the mean-square error (MSE) or the cross-entropy loss (CE).

6. The SoftMax function allows to get a probabilistic interpretation of the output. The output is normalized which allows a better understanding of the resutl. It alsoa allows multi-class classification.

7.

1. Hidden Layer:

- Linear Transformation:

$$\tilde{h} = xW_h^T + b_h$$

- Activation (tanh):

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

2. Output Layer:

- Linear Transformation:

$$\tilde{y} = hW_y^T + b_y$$

- Activation (SoftMax):

$$\hat{y} = \text{SoftMax}(\tilde{y})$$

Where:

- x represents the input vector of size n_x.
- $\tilde{h}$ is the intermediate output of the hidden layer, a vector of size n_h.
- h is the output of the hidden layer after applying the tanh activation, also a vector of size n_h.
- $\tilde{y}$ is the intermediate output of the output layer, a vector of size n_y.
- $\hat{y}$ is the final output of the network after applying the SoftMax activation, also a vector of size n_y.
- W_h is the weight matrix for the hidden layer with dimensions $n_h \times n_x$.
- b_h is the bias vector for the hidden layer with dimensions n_h.
- W_y is the weight matrix for the output layer with dimensions $n_y \times n_h$.
- b_y is the bias vector for the output layer with dimensions n_y.

The forward pass is :

$$\hat{y}(x) = \text{SoftMax}((\tanh(xW_h^T + b_h))W_y^T + b_y)$$

8. During training, the value of $\hat{y}_i$ should approach the true label class, indicating that the predicted value becomes a more accurate representation of the true value. Specifically, for cross-entropy loss, $\hat{y}_i$ should approach 1 for the correct class and approach 0 for all other classes. In the case of squared error loss, the goal is for $\hat{y}_i$ to be as close as possible to the true value $\hat{y}_i$, ensuring that the predicted value closely approximates the original data point

9. The cross entropy loss is better suited for classification tasks because it penalizes incorrect class predictions so it encourages the model to produce class probabilities. Indeed, it is used when the task involves predicting discrete class labels or the associated probability for each of them. In contrast, the MSE loss is better suited for regression tasks as it calculates the average squared difference between predicted values and real ones. This loss function is used when the objective is to predict continuous numerical values.

10. The classic gradient descent presents the advantage to be the best estimation of the gradient as all training examples are taken into account in every iteration. However, there is a risk that the algorithm never ends because of its computational cost in terms of calculus. The online stochastic version is quicker as it only processes one training example at a time, but this version is very sensitive to noise. Finally, the mini-batch stochastic gradient descent strikes a good balance between the computational efficiency of classic gradient descent and the stochastic nature of online gradient descent. It still can be sensitive to noise in the data and requires a good selection of the mini-batch size. In the general case, the mini-batch stochastic gradient descent seems to be the most reasonable choice to make to achieve this compromise between computational efficiency and convergence stability.

11. The learning rate has a huge impact on learning. If this learning rate is too small, there is a risk to never converge to a minima as parameters updates would be very small. It also improves the risk of getting stuck in a local minima. On the contrary, if this learning rate is too big there is risk of divergence resulting in an oscillation of the parameters. Therefore it is very important to do several tests in order to adjust this learning rate.

12. Forward accumulation evaluates the function and calculates the derivative with respect to one independent variable in one pass. For each

independent variable X1...Xn a separate pass is therefore necessary to compute its corresponding derivative.

In contrast, reverse accumulation evaluates the function forward to compute intermediate values during a forward pass. It then calculates the derivatives with respect to all independent variables in a backward pass.

Forward accumulation is more efficient than reverse accumulation for functions f : Rn $\rightarrow$ Rm with n $\ll$ m (when there are more inputs than outputs). It is because only n sweeps (passes) are necessary, compared to m sweeps for reverse accumulation.

Reverse accumulation is more efficient than forward accumulation for functions f : Rn $\rightarrow$ Rm with n $\gg$ m as only m sweeps are necessary, compared to n sweeps for forward accumulation. Usually, in Neural Nets, n$\gg$ m so the reverse accumulation is more efficient.

For backpropagation, the naive implementation complexity which does not store the gradient is O(n^2) where n is the number of layers in the network. More precisely $O(\frac{n(n-1)}{2})$. The backpropagation algorithm complexity which stores the gradient is in O(n).

13. In order to back-propagate the gradient, each module(linear, activation and loss) must be differentiable. Also it should have should have a layered structure where each layer's output serves as the input to the subsequent layer.

14.

$$L(Y, \hat{y}) = -\sum_i y_i log(\hat{y}_i)$$

$$= -\sum_i [y_i log(\frac{e^{\tilde{y}_i}}{\sum_j e\tilde{y}_j})]$$

$$= -\sum_i (y_i \tilde{y}_i) + \sum_i (y_i) \times log(\sum_j e\tilde{y}_j)$$

Knowing that $\sum_i y_i = 1$, we can delete the term (and replace index j with i to return to the formula in the question, but this doesn't change anything).

$$L(y, \hat{y}) = \sum_i y_i \tilde{y}_i + \log(\sum_i e\tilde{y}_i)$$

15. The gradient of the loss (cross-entropy) relative to the intermediate output $\tilde{y}_i$ is :

$$\frac{\partial \ell}{\partial \tilde{y}_i} = -y_i + \frac{e^{\tilde{y}_i}}{\sum_{i'} e^{\tilde{y}_{i'}}} = \hat{y}_i - y_i$$

$$\nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_n} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix}$$

16. The gradient of the loss with respect to the weight of the output layer $\nabla_{W_y} l$ is:

$$\frac{\partial \ell}{\partial W_{y,i,j}} = \frac{\partial \ell}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial W_{y,i,j}} = \frac{\partial \ell}{\partial \tilde{y}_i} \frac{\partial \sum_k \left(W_{y,i,k} h_k + b_{y,i} \right)}{\partial W_{y,i,j}} = \left(\hat{y}_i - y_i \right) h_j$$

So :

$$\nabla_{W_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{W}_{y,11}} & \cdots & \frac{\partial \ell}{\partial \tilde{W}_{y,1 n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \tilde{W}_{y,n_y 1}} & \cdots & \frac{\partial \ell}{\partial \tilde{W}_{y,n_y n_h}} \end{bmatrix}$$

$$\nabla_{W_y} \ell = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix}^T \begin{bmatrix} h_1 \ddots h_{n_h} & 0 & \dots & 0 \\ 0 & h_1 \ddots h_{n_h} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & h_1 \ddots h_{n_h} \end{bmatrix}$$

The gradient of the loss with respect to the bias of the output layer $\nabla_{b_y} l$ is:

$$\frac{\partial \ell}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \tilde{y}_i} \frac{\partial \sum_k \left(W_{y,i,k} h_k + b_{y,i} \right)}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i$$

$$\nabla_{b_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{b}_{y,1}} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{b}_{y,n_y}} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix}^T$$

17.

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{h}_i} &= \frac{\partial \ell}{\partial h_i}\frac{\partial h_i}{\partial \tilde{h}_i} \\
&= \left(\sum_j \frac{\partial \ell}{\partial \tilde{y}_j}\frac{\partial \tilde{y}_j}{\partial h_i}\right)\frac{\partial h_i}{\partial \tilde{h}_i} \\
&= \left[\sum_j \frac{\partial \ell}{\partial \tilde{y}_j}\frac{\partial \sum_k \left(W_{y,j,k}h_k + h_{y,j}\right)}{\partial h_i}\right]\frac{\partial tanh(\tilde{h}_i)}{\partial \tilde{h}_i} \\
&= \left[\sum_j \left(\hat{y}_j - y_j\right) W_{y,j,i}\right]\left(1 - tanh^2(\tilde{h}_i)\right) \\
&= \left(1 - h_i^2\right)\left[\sum_j \left(\hat{y}_j - y_j\right) W_{y,j,i}\right]
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \ell}{\partial W_{h,i,j}} &= \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial \tilde{h}_i}{\partial W_{h,i,j}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial \sum_k \left(W_{h,i,k}x_k + b_{h,i}\right)}{\partial W_{h,i,j}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_i}x_j \\
&= \left(1 - h_i^2\right)\left[\sum_j \left(\hat{y}_j - y_j\right) W_{y,j,i}\right]x_j
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \ell}{\partial b_{h,i}} &= \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial \tilde{h}_i}{\partial b_{h,i}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial \sum_k \left(W_{h,i,k}x_k + b_{h,i}\right)}{\partial b_{h,i}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_i} \\
&= \left(1 - h_i^2\right)\left[\sum_j \left(\hat{y}_j - y_j\right) W_{y,j,i}\right]
\end{aligned}$$

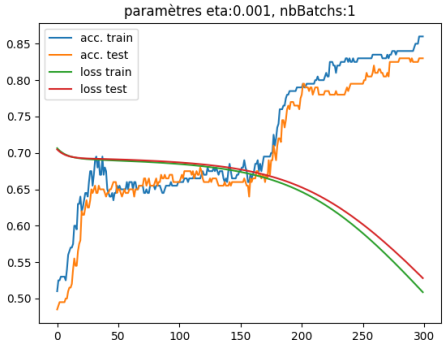
$$\nabla_{\tilde{h}}\ell = \begin{bmatrix} \left(1 - h_1^2\right)\left[\sum_j \left(\hat{y}_j - y_j\right) W_{y,j,1}\right] \\ \vdots \\ \left(1 - h_{n_h}^2\right)\left[\sum_j \left(\hat{y}_j - y_j\right) W_{y,j,n_h}\right] \end{bmatrix}$$

$$\nabla_{W_h}\ell = \begin{bmatrix} (1-h_1^2)\left[\sum_j (\hat{y}_j - y_j)\, W_{y,j,1}\right] x_1 & \dots & (1-h_1^2)\left[\sum_j (\hat{y}_j - y_j)\, W_{y,j,1}\right] x_{n_x} \\ \vdots & \ddots & \vdots \\ \left(1-h_{n_h}^2\right)\left[\sum_j (\hat{y}_j - y_j)\, W_{y,j,n_h}\right] x_1 & \dots & \left(1-h_{n_h}^2\right)\left[\sum_j (\hat{y}_j - y_j)\, W_{y,j,n_h}\right] x_{n_x} \end{bmatrix} -$$

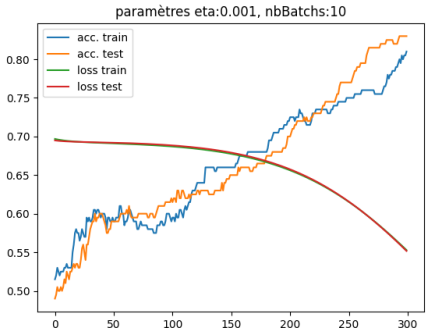
$$\nabla_{b_h}\ell = \nabla_{\tilde{h}}\ell^{.T} \begin{bmatrix} (1-h_1^2)\left[\sum_j (\hat{y}_j - y_j)\, W_{y,j,1}\right] \\ \vdots \\ \left(1-h_{n_h}^2\right)\left[\sum_j (\hat{y}_j - y_j)\, W_{y,j,n_h}\right] \end{bmatrix}^{.T}$$

2. Implementation

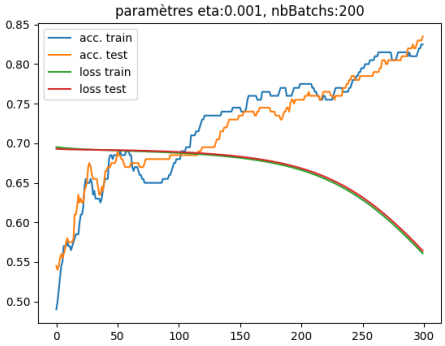
a. Forward and backward manuals



(a) stochastic gradient n = 1

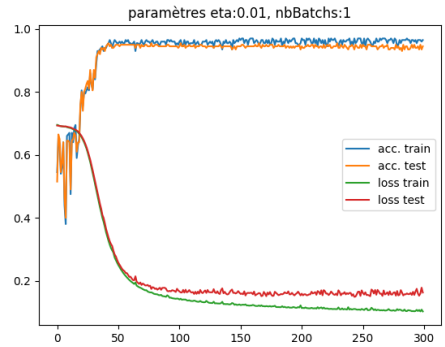


(b) mini batch n = 10

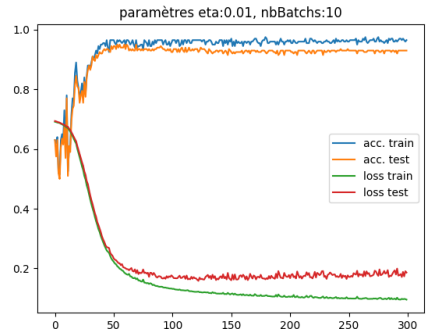


(c) Batch n = 200

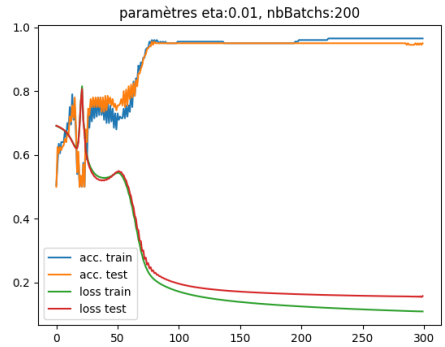
Figure 1: Learning rate : 0.001



(a) stochastic gradient n = 1

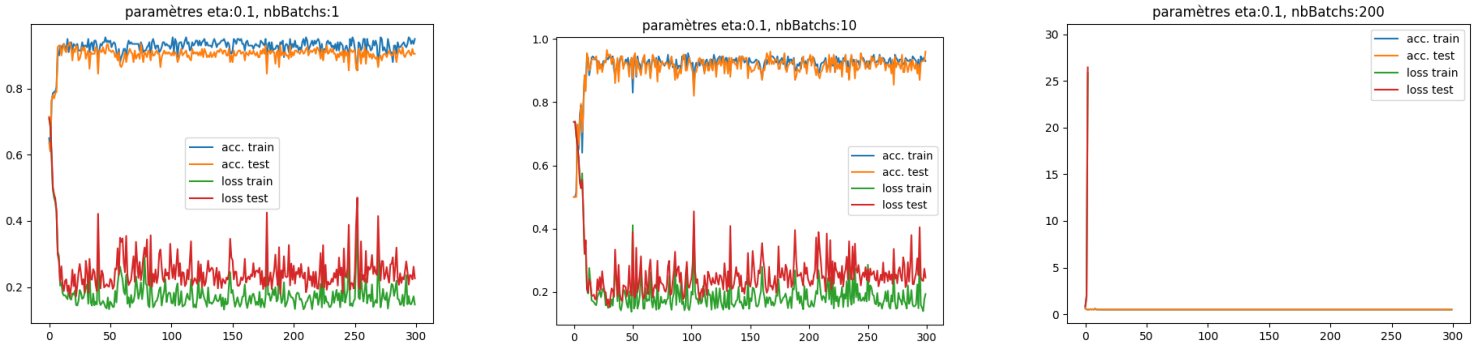


(b) mini batch n = 10



(c) Batch n = 200

Figure 2: Learning rate : 0.01



(a) stochastic gradient n = 1 (b) mini batch n = 10 (c) Batch n = 200

Figure 3: Learning rate : 0.1

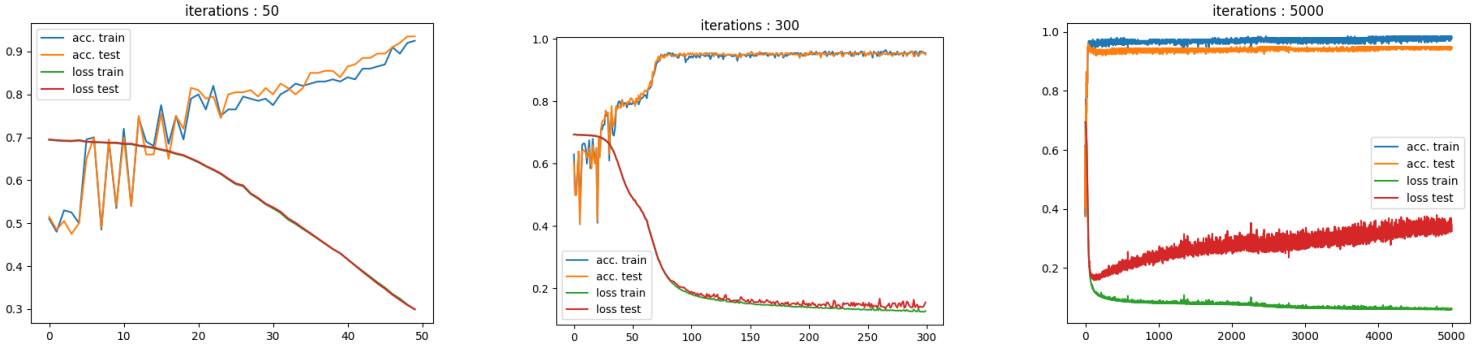
In regards to the batch, we are currently examining the impact of the learning rate. As shown in Figure 1, employing a learning rate of 0.001 results in a gradual decline of the loss, with parameters being less modified at each epoch. Unfortunately, the accuracy rate exhibited in both train and test is less impressive than that achieved using a learning rate of 0.01. This is an instance of underfitting, which may be corrected through either increasing the learning rate or adding additional epochs to reach convergence.

On Figure 11, with a 0.1 learning rate, we observe a rapid decrease in the loss, with modified parameters at each epoch. The accuracy and loss both fluctuate at each epoch, with evident spikes in the figures. This increased variability makes it more likely to diverge, as large steps are being taken at each epoch. To solve this issue, we need to decrease the learning rate as simply adding extra epochs will not mitigate the problem of divergence.

The optimal learning rate for this neural network is 0.01 figure 2 because it enables quick loss convergence without the risk of divergence, as indicated by the absence of spikes. Moreover, the accuracy in both train and test is remarkably good, without any signs of overfitting.

Next, we will delve into the influence of batch size on performance On the plots for a learning rate of 0.1 shown in figure 11, we see that the batch method results in rapid loss divergence due to a high learning rate and abrupt gradient changes, while mini-batch and stochastic methods have less stable gradients but undergo updates at a slower pace. Upon closer examination of the plots for a learning rate of 0.01 figure 2, we can ascertain that the batch method is more stable than the mini-batch and stochastic methods. Nonetheless, slight instability is not detrimental, as it can aid in escaping a

local minima. This is why mini-batch methods are generally used in deep learning, where objective functions are often non-convex.

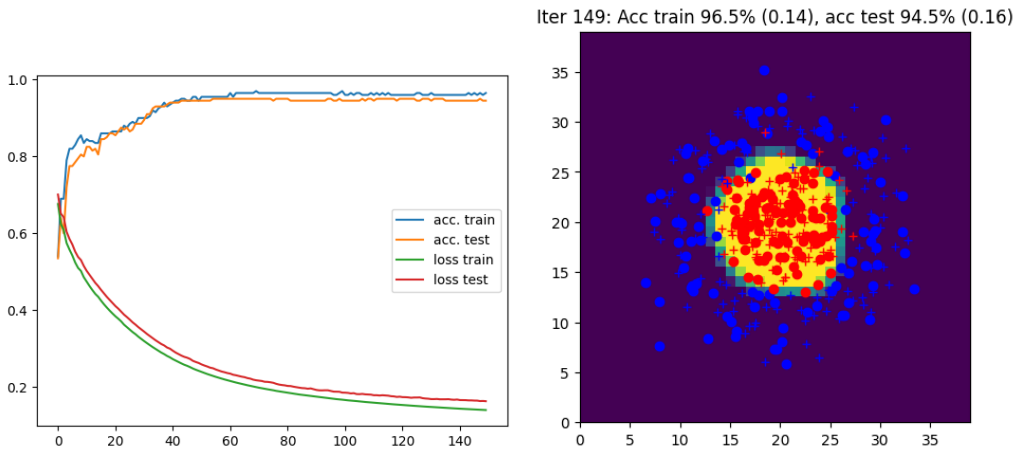


(a) Epochs : 50 (b) Epochs : 300 (c) Epochs : 5000

Figure 4: loss and accuracy in function of the number of iterations

On Figure 4, we conducted experiments using varying numbers of epochs. On the other hand, the graph for 5000 epochs suggests overfitting. The chart corresponding to 50 epochs indicates underfitting as demonstrated by the ongoing decrease in loss coupled with a continuous increase in accuracy for the train and test datasets. This is due to a period of decreasing loss present in both the train and test datasets, followed by continued reductions in train dataset loss but increased loss for the test dataset. Alongside this, the accuracy in the test set is lower than the accuracy in the training set, and this model exhibits poorer generalization. The optimal choice of epoch for this neural network is the second figure with 300 epochs, wherein the loss converges near 0 in both the test and train sets without any increase in test error. Additionally, the accuracy in both the test and train sets is identical, and there is no decrease in test performance. Thus, this model demonstrates better generalization.

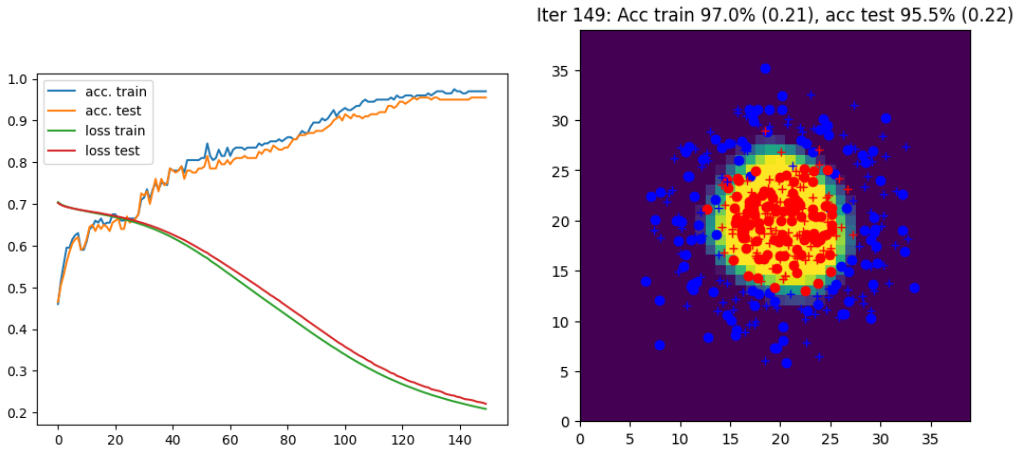
b. Simplification of the backward pass with torch.autograd



(a) Loss (b) Final plot

Figure 5: Loss and plot with the use of torch.autograd

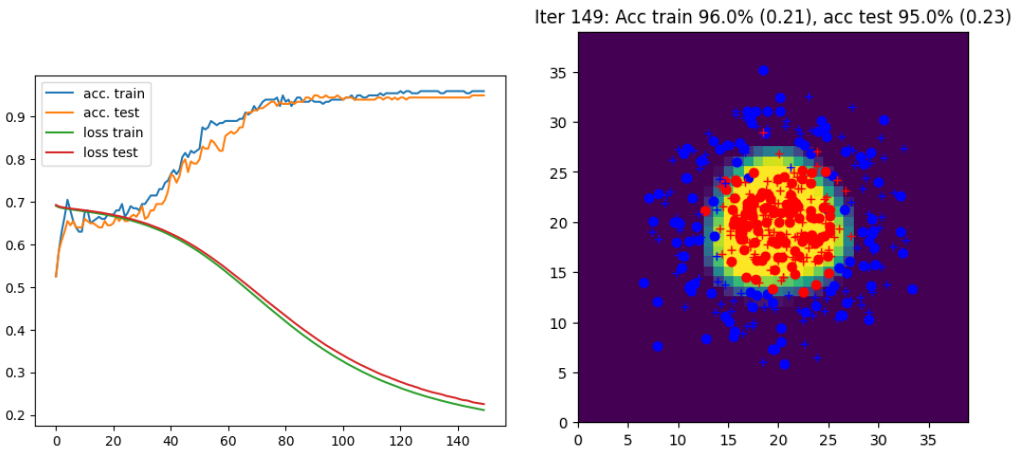
c. Simplification of the forward pass with torch.nn layers



(a) Loss (b) Final plot

Figure 6: Loss and plot with the use of torch.nn

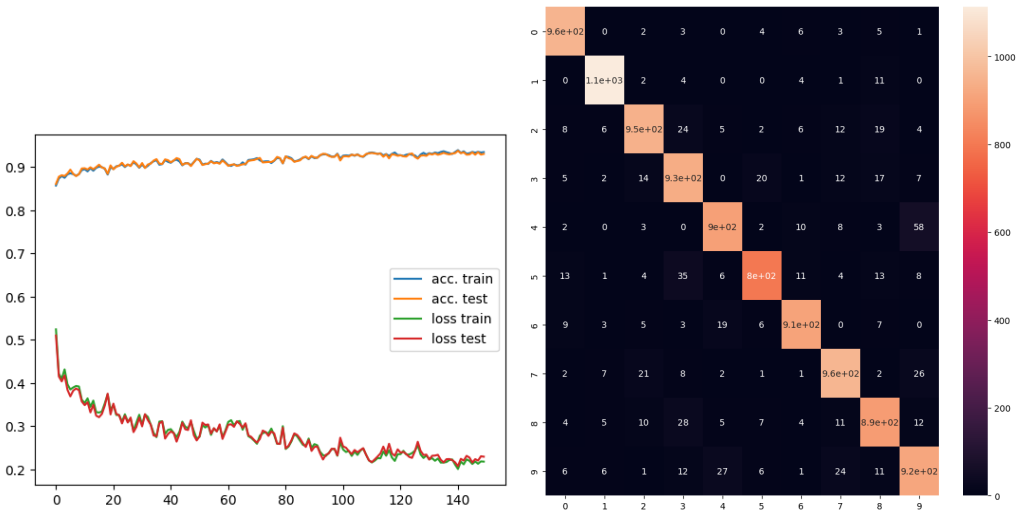
d. Simplification of the SGD with torch.optim



(a) Loss (b) Final plot

Figure 7: Loss and plot with the use of torch.optim

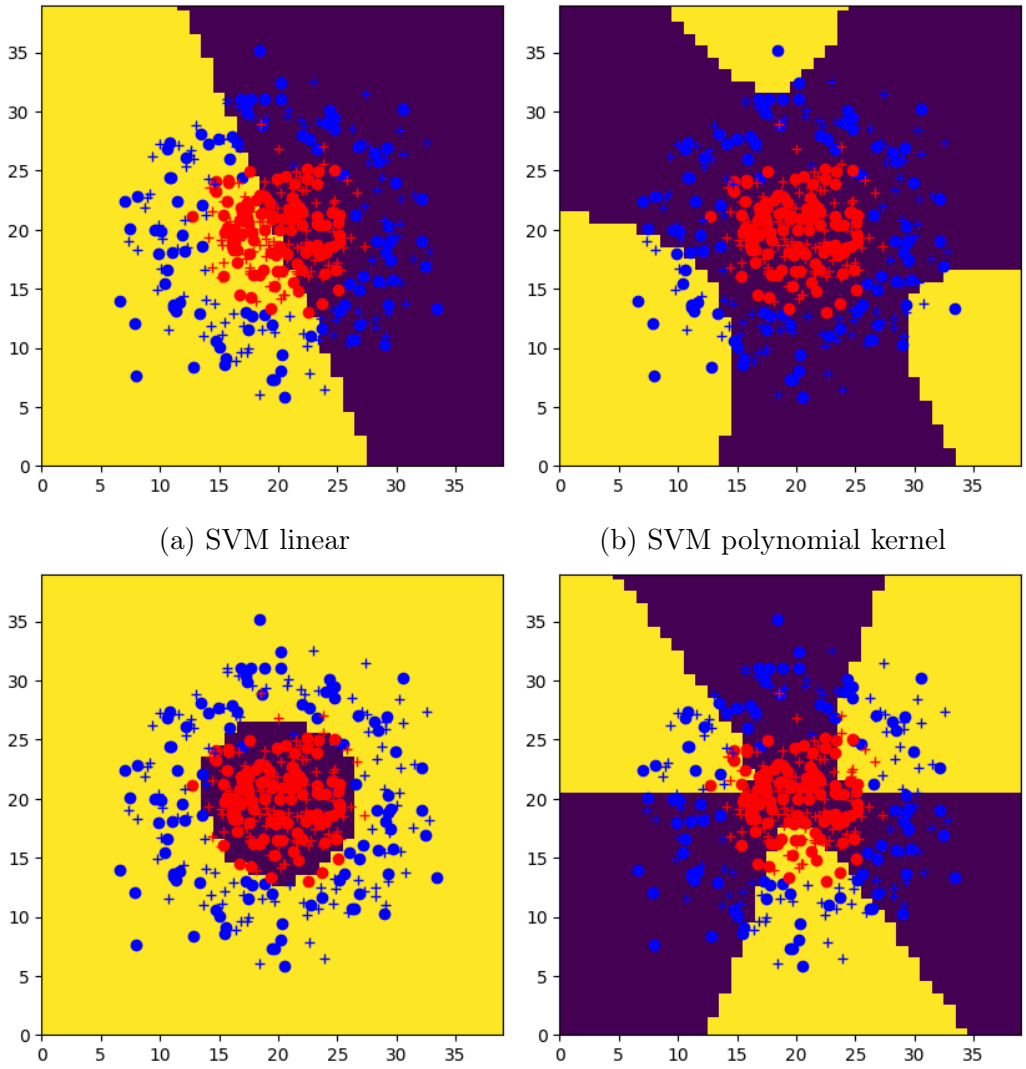
e. MNIST application



(a) Loss (b) Confusion matrix

Figure 8: Loss and confusion matrix for MNIST data

f. Bonus: SVM



(a) SVM linear

(b) SVM polynomial kernel

(c) SVM radial basis function kernel

(d) SVM sigmoid kernel

Figure 9: SVM

II. Convolutional Neural Networks (c-d)

1. Introduction to convolutional networks

1. Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size $x \times y \times z$ the output size will be :

$$\left(\lfloor\frac{x-k+2p}{s}\rfloor+1\right) \times \left(\lfloor\frac{y-k+2p}{s}\rfloor+1\right) \times z$$

There would be $k \times k \times z$ weights to learn.

If a fully-connected layer were to produce an output of the same size, we would have had to learn : $(x \times y \times z)$.

2. Convolutional layers allows to improve scalability as we can learn a less important number of parameters to learn than with fully-connected layers. Also it improves the stability of the representation, as in fully connected layers a small distortion can drastically change the representation while convolutionnal layers has equivariant properties.

3. Spatial pooling reduces the dimensionality of the representation and aggregates information, thereby providing certain invariant properties under specific conditions. Max pooling guarantees these invariances under certain conditions, while average pooling doesn't offer such guarantees (but it does improve overall invariance to some extent).

4. If we try to compute the output of a classical convolutional network for an input image larger than the initially planned size there will be a dimension issue when flattening the output of the convolutional blocks as the vector will be larger than the one which was expected. Therefore, the weight vector will not match the shape of the input vector. Nevertheless, for convolutional and pooling layers there will not be any problem.

5. Fully connected layers in a neural network are typically used to perform a linear transformation on the flattened or pooled feature maps from the preceding convolutional layers. They can be thought of as a convolution operation with a kernel (filter) that spans the entire spatial dimension of the feature maps. To analyze fully connected layers as convolutions, we replace each fully connected layer with a convolutional layer. The kernel size of this convolution should match the spatial dimensions of the feature maps and the pareameters this convolutional layer should match the number of parameters of the fully connected layer. The number of filters in this convolutional layer

is equal to the number of neurons in the fully connected layer.

6. If we replace fully-connected by their equivalent in convolutions we would be featuresmap shape[0] parametre The interest is that we can use the same network architecture on images of varying sizes without changing the model's parameters also The network can capture features at multiple scales in the larger image, which can be useful for tasks like object detection.

Spatial Information: Retaining spatial information in the output allows you to analyze and interpret features in the context of the input image.

7. The size of the receptive field of the first layer is equal to the size of the kernel, thus $k \times k$. For instance, for 3×3 kernel , the receptive field for the first layer will be 3×3.

For the second layer, the receptive field depends on the size of the kernel and the stride. For a kernel of size 3×3 and a stride of 1, the receptive field of the second layer would be 5×5. If the stride was equal to 2, it would be equal to 7×7.

We can define it by recurrence in 1D as follows (it can be generalized to other dimension):

$$RFS_0 = k$$
$$RFS_n = \mathrm{RFS}_{n-1} \cdot k + (k - s)$$

2. Training from scratch of the model

a. Network architecture

8. In order to conserve the same spatial dimension we need to put: padding = k//2 and stride = 1.

9. For max poolings, if we want to reduce the spatial dimensions by a factor of 2 we need : padding = k//2 -1 , stride 2

10.

Layer	Output size	Number of weights
conv1	$32 \times 32 \times 3 = 3072$	$5 \times 5 \times 3 \times 32 = 2400$
pool1	$16 \times 16 \times 32 = 8\ 192$	0
conv2	$16 \times 16 \times 64 = 16384$	$16 \times 16 \times 32 \times 64 = 524\ 288$
pool2	$8 \times 8 \times 64 = 4096$	0
conv3	$8 \times 8 \times 64 = 4096$	$8 \times 8 \times 64 \times 64 = 262\ 144$
pool3	$4 \times 4 \times 64 = 1024$	0
fc4	1000	$4 \times 4 \times 64 \times 1000 = 1\ 024\ 000$
fc5	10	$1000 \times 10 = 10\ 000$

11. the total number of parameters is 1.822.832 with a majority of parameters contained in the first fully connected layer.

12. Let's consider an example of a Bag of Words (BOW) and multiclass Support Vector Machine (SVM) with 10 classes. The number of parameters varies based on whether a visual word dictionary is created. In this case, we will assume the need to create a dictionary. An image is represented by a Scale-Invariant Feature Transform (SIFT) that is typically divided into 16 parts, each with 8 directions. This results in a 128-dimensional vector. To learn the dictionary, a K-means algorithm must be performed, resulting in K parameters of 128 dimensions, each corresponding to a centroid. Generally, K is between 100 and 1000. This process produces a histogram that illustrates the proximity of each sift in an image to one of the clusters. No new parameters are needed for this step. The next step is to create a support vector machine (SVM) with 10 classes, which is equivalent to training 10 binary SVMs. Each binary SVM has K weights and 1 bias, so the total number of weights and biases for the SVM is 10 * (K+1).

For k = 100:

$$(100 \times 128) + 10 \times (100 + 1) = 13,810.$$

For k = 1000:

$$(1000 \times 128) + 10 \times (1000 + 1) = 138,010.$$

As demonstrated, the number of parameters required for learning is lower than that of our neural network.

b. Network learning

13.

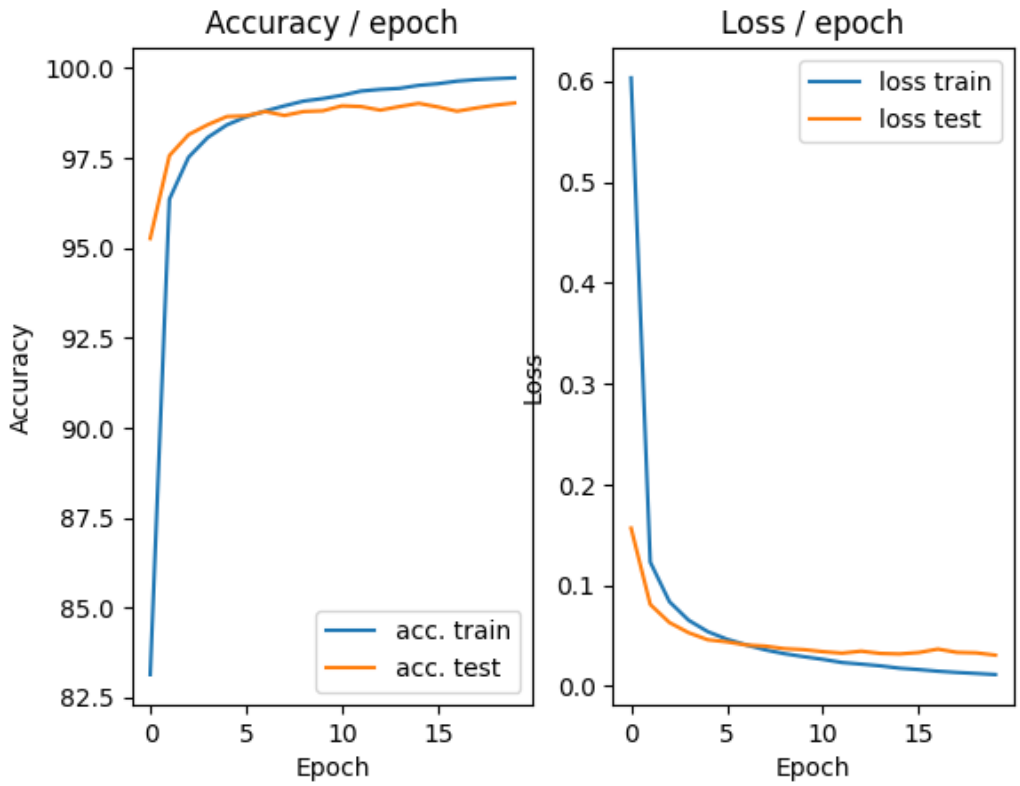
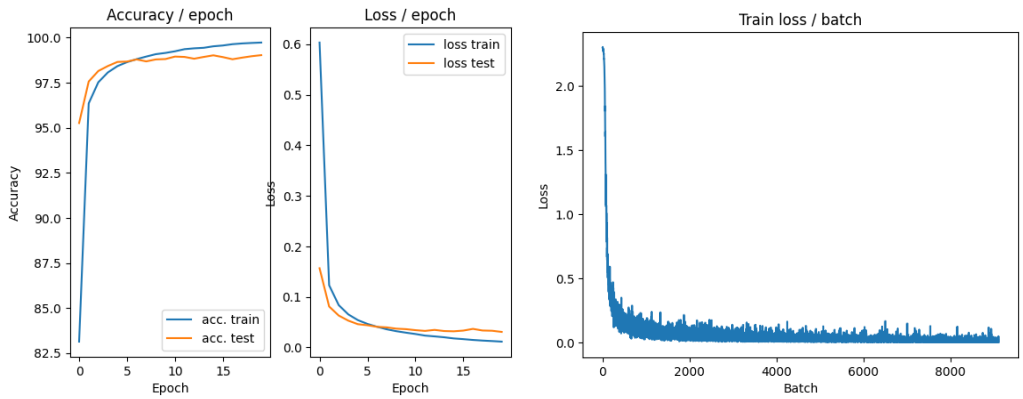


Figure 10: MNIST (learning rate 0.1, epoch 20) : Prec@1 98.92%

14.



(a) MNIST (train/test loss and accuracy) (b) MNIST (train loss batch 200/569)

Figure 11: Loss and accuracy curves in train and in test on the MNIST dataset for a learning rate of 0.1 and 20 epoch (Prec@1 98.92%).

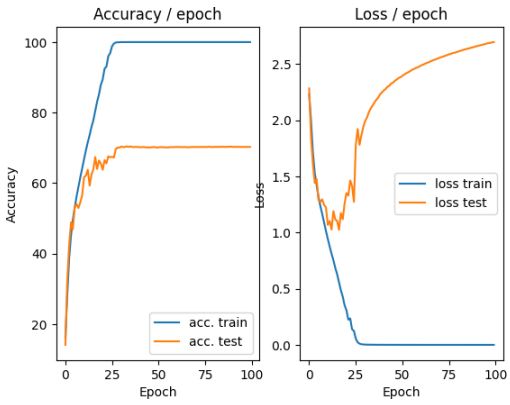
The test loss is calculated by averaging the losses of all test batches at each epoch, while the train loss is calculated not only by averaging the losses of all train batches but also at each individual batch. In fact, during the test evaluation phase in the code, the loss gradient is not updated after every batch, as opposed to the training phase. Therefore, it is not beneficial to have the model's weights adjusted based on the test data, as this could lead to overfitting or other undesirable effects. The main goal of the test phase is to assess how well the trained model generalizes to unseen data, rather than optimizing its parameters.

15. We have decided to to do 100 epochs for a learning rate of 0.1 and for a learning rate of 0.05 to be sure that the model has finished converging.

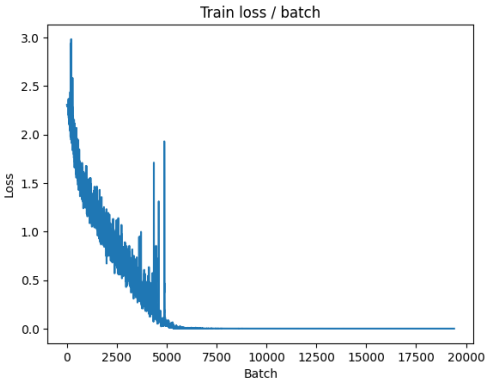
16. If the learning rate is too high, the model might not converge. It can oscillate around the optimal weights or even diverge, leading to poor performance.The training process may become unstable, with loss values increasing or showing extreme fluctuations. If the learning rate is too low, the model might converge very slowly. Training can take an impractically long time. The model can also easily get stuck in local minima, making it harder to reach the global optimal solution.

Using a large batch size can speed up training by processing more data points simultaneously. However, large batch sizes may reduce the variability within each batch. Smaller batch sizes result in more frequent updates of

model weights and introduce a higher variability which can help the model avoid getting stuck in local minima. Training with small batch sizes requires less memory, which is advantageous when working with limited resources.

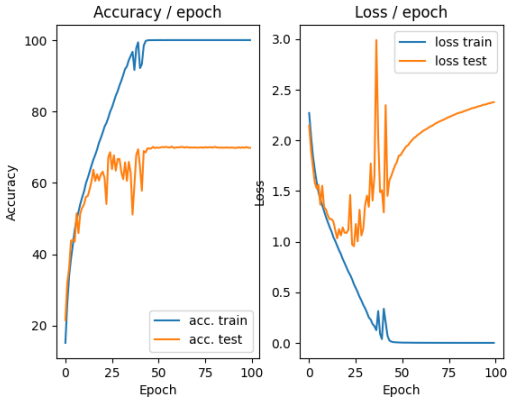


(a) CIFAR10 (train/test loss and accuracy)

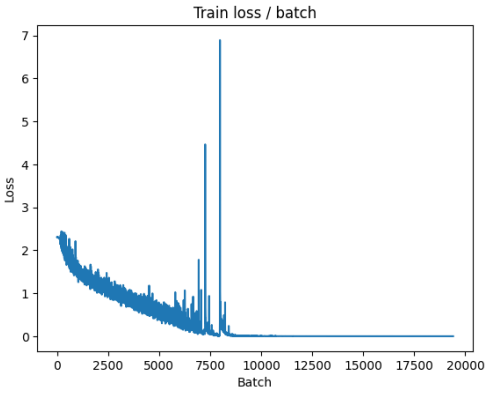


(b) CIFAR10 (train loss batch 512)

Figure 12: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch (Prec@1 70.26%).

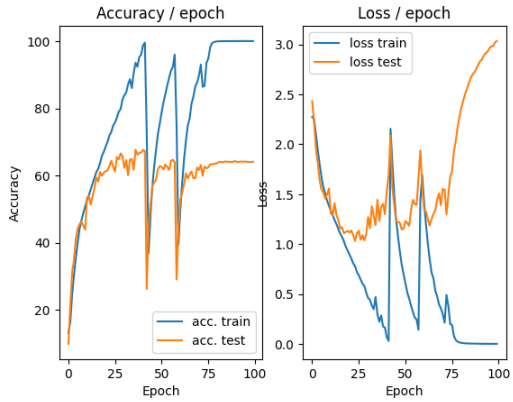


(a) CIFAR10 (train/test loss and accuracy)

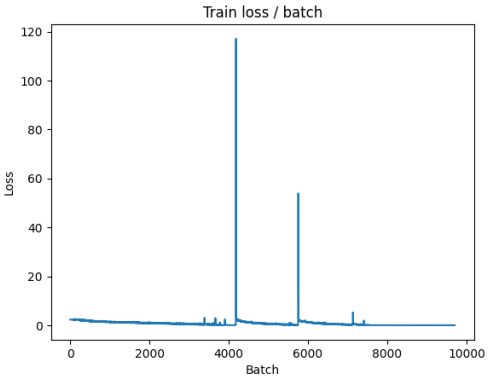


(b) CIFAR10 (train loss batch 512)

Figure 13: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.05** and 100 epoch.

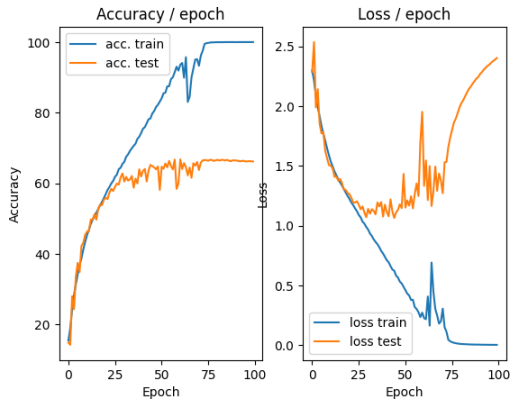


(a) CIFAR10 (train/test loss and accuracy)

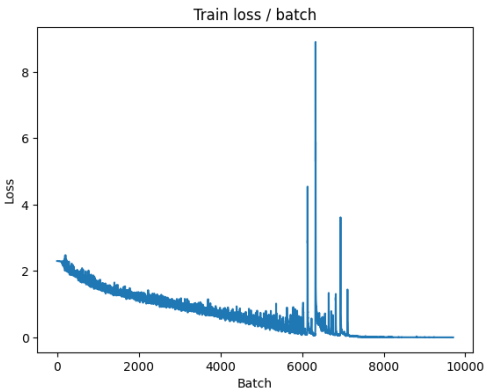


(b) CIFAR10 (train loss batch 128)

Figure 14: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **512**, learning rate of **0.1** and 100 epoch (Prec@1 64.15%).



(a) CIFAR10 (train/test loss and accuracy)



(b) CIFAR10 (train loss batch 128)

Figure 15: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **512**, learning rate of **0.05** and 100 epoch (Prec@1 66.17%).

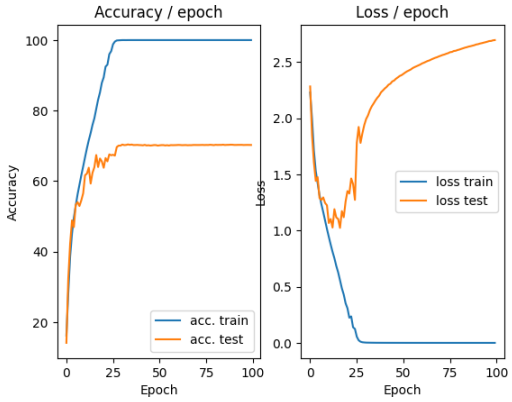
17. At the start of the first epoch in train and test the network's weights are initialized randomly. The precision at one (prec@1) is approximately 10%, which is consistent with the CIFAR10 database containing 10 distinct classes. The performance resembles that of random guessing.

18. Upon achieving convergence at the conclusion of training, depicted in Figure 15, the loss in the train remains stable at nearly 0, yet the loss in the test rises again from 20 epochs. Correspondingly, train accuracy exhibits favorably, but test accuracy is inadequate and similarly declines from 15 epochs. This phenomenon is known as overfitting.

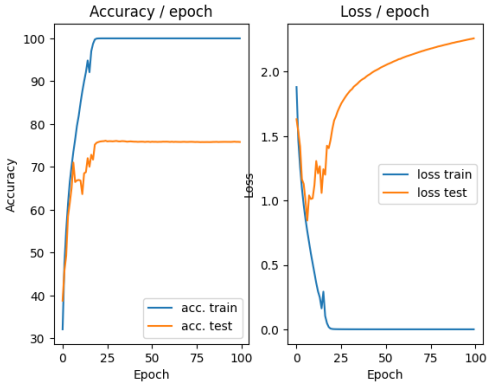
3. Results improvements

a. Standardization of examples

19.



(a) Without standardization : Prec@1 **70.26**%



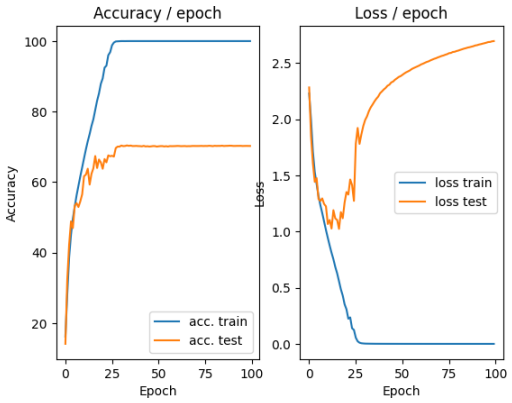
(b) With **standaridzation** of examples: Prec@1 **75.82**%

Figure 16: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch.

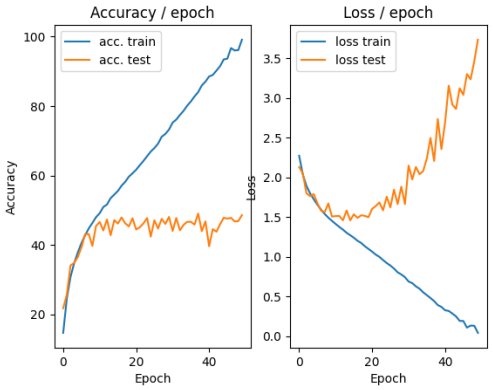
Compared to the figure 12 where the maximum accuracy was 70.26%, we can see that standardization helped to reach an accuracy of 75.82%. Moreover, we can observe that there is less overfitting when we examine the testing loss curve with standardization compared to the one without standardization.

20. We calculate the average image on the training examples and normalize the validation examples with the same image in order to ensure that the validation data is transformed in the same way as the training data. This is important because the model learned to work with the normalized training data, so the same normalization scheme should be applied to the validation data for consistency.

21. ZCA normalization (Zero-phase Component Analysis Whitening) not only centers the data but also decorrelates the features. It does this by applying a linear transformation to the data, ZCA whitening scales the features to have unit variance, making them "white" in the sense that they are uncorrelated and have the same variance. It can be beneficial when dealing with highly correlated data mostly where there might be spatial correlations. This method is more complex than the previous and has a computational price.
We have trained on only 50 epochs because of the computational price and that we ran out of google colab computing units.



(a) Without standardization : Prec@1 **70.26**%



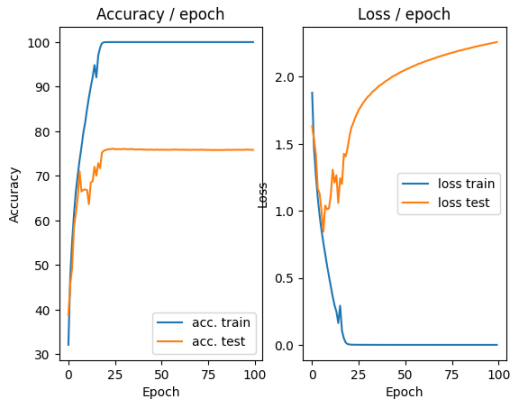
(b) With **ZCA whitening** of examples: Prec@1 **48.57**%

Figure 17: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 50 epoch.

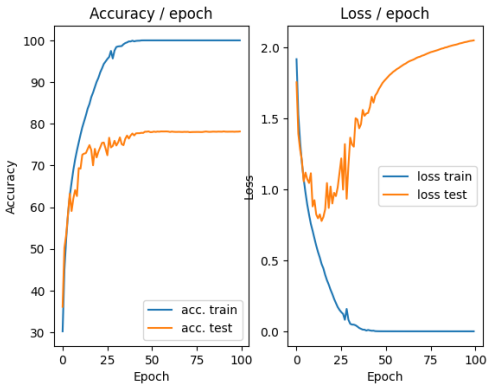
There are also :
- Min-Max Scaling : Xnormalized = (X - Xmin) / (Xmax - Xmin) (to have values within a bounded interval)
- Max Absolute Scaling : Xnormalized = X / max(abs(X)) (to preserve the sign of the values but scale them between -1 and 1)
- Robust Scaling (IQR Scaling) : Xnormalized = (X - Q1) / (Q3 - Q1) (helps when outliers)
- Log Transformation : Xnormalized = log(X) for positive-valued data that spans several orders of magnitude

b. Increase in the number of training examples by data increase

22.



(a) Without data augmentation : Prec@1 **75.82**%



(b) With **data augmentation** of examples: Prec@1 **78.15**%

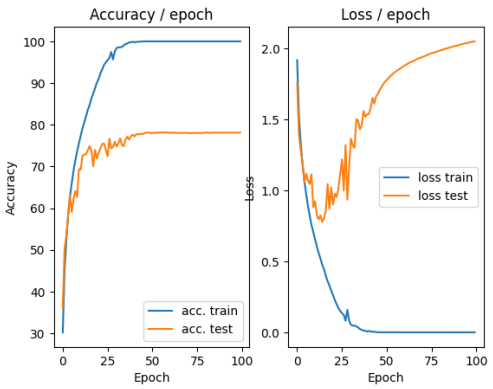
Figure 18: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch.

Here we can see that the addition of data augmentation has improved accuracy (78.15% vs. 75.82% previously). We also note that the number of epochs required for model convergence is slightly higher, which can be explained by the increase in the dataset size.

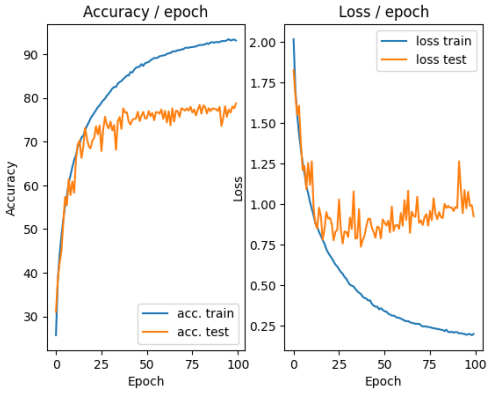
23. Horizontal transformations may not be suitable for certain image recognition tasks, such as number recognition. For example, the digits 6 and 9 are distinct numbers, but applying horizontal transformations can lead to ambiguity due to the symmetry between these two classes. This symmetry can make it challenging for the network to accurately distinguish between them.

24. This kind of modification of the dataset increases the computational cost of the training as the dataset becomes larger. Also it can lead to loss of information especially if the crop size is significantly smaller than the original image. Finally, overfitting could occur when the model learns to recognize specific augmented patterns rather than true features of the data.

25.



(a) With data augmentation **RandomHorizontalFlip** and **CenterCrop**: Prec@1 **78.15**%



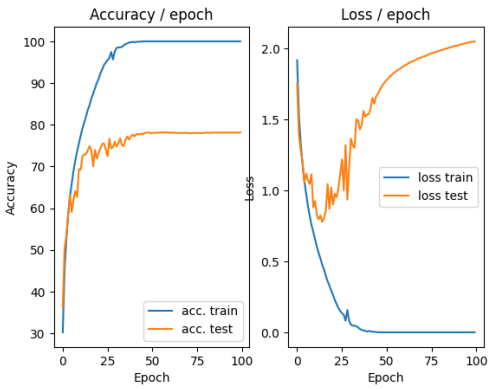
(b) With data augmentation **RandomPerspective** and **RandomCrop**: Prec@1 **78.76**%

Figure 19: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch.

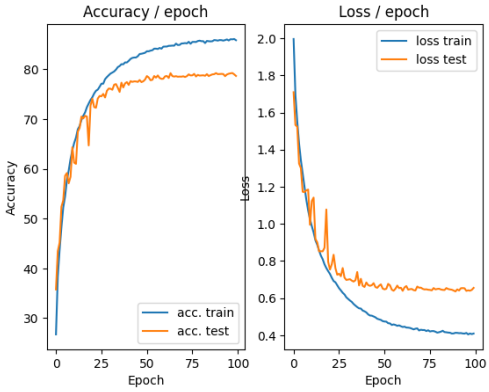
We have found many data aumentation methods possible : Rotation, Scaling, Resizing, Translation, Shearing , Grid, and Elastic Distortion, Brightness and Contrast Adjustment, Blur and Noise, Color Jitter, Cutout,Random Erasing, Flip, Channel Shuffling, Histogram Equalization or Stretching, Style Transfer, Superimposition, Mosaic Augmentation, Patch-wise Augmentation, Random Perspective, Occlusion. We have decided to test two other data augmentations (instead of the ones previously used):
- **RandomPerspective(p=0.5)** : This augmentation randomly applies a perspective transformation to the images with a probability of 0.5. It can introduce distortions that simulate the effect of viewing an object from a different perspective, which can be helpful in training a more robust model.
- **RandomCrop((28,28))** : This augmentation crops the input images to a size of 28x28 pixels at a random location. It can help the model generalize better by providing different views of the input data during training.

c. Variants on the optimization algorithm

26.



(a) Loss and accuracy curves without ExponentalLR: Prec@1 **78.15**%



(b) Loss and accuracy curves with **ExponentialLR**: Prec@1 **78.64**%

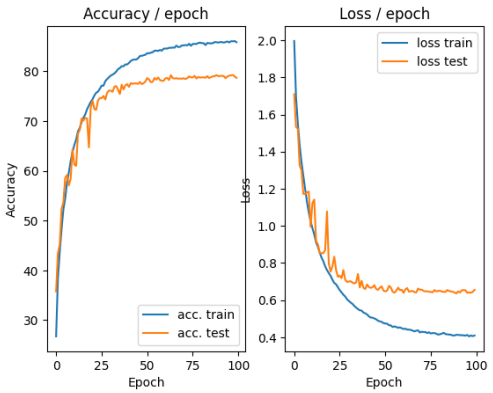
Figure 20: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch.

When adding the ExponentialLR optimization variant, a reduction in overfitting is observed when examining the test loss curve. Furthermore, there is an observable deceleration in convergence, attributed to the gradual decrease in the learning rate across epochs. This appears to contribute to a small stabilization of the learning process over the epochs.

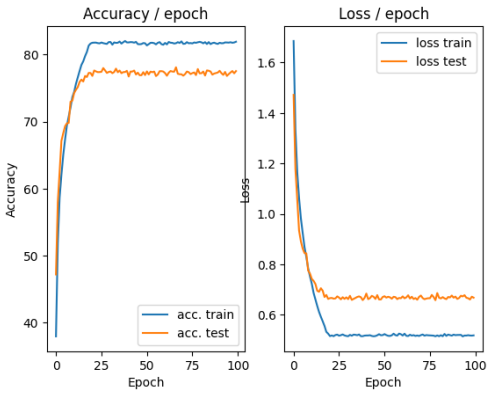
27. The introduction of the ExponentialLR optimization variant improves learning by gradually decreasing the learning rate during training. This allows the model to explore the parameter space more widely at the beginning, helping to have a faster convergence and to espcape local minima. At the end of training, the reduced learning rate helps avoid divergence and fine-tunes the model's parameters with greater precision. Moreover, the gradual reduction in the learning rate enhances the model's generalization, as it becomes less sensitive to noise in the training data as the learning rate decreases.

28. We have decided to test the Adam opimizer and the PolynomialLR learning rate scheduler.
The Adam optimizer is an adaptive learning rate optimization algorithm. It combines the benefits of both AdaGrad (that maintains a per-parameter learning rate that improves performance on problems with sparse gradients)

and RMSProp (that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight). This optimizer is known for its efficiency and fast convergence.
The PolynomialLR learning rate scheduler decays the learning rate of each parameter group using a polynomial function in the given total_iters.



(a) Loss and accuracy curves with **SGD** optimizer and **ExponentialLR** and a learning rate of **0.1**: Prec@1 **78.64**%



(b) Loss and accuracy curves with **Adam** optimizer and **PolynomialLR** and a learning rate of **0.001**: Prec@1 **77.54**%

Figure 21: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256** and 100 epoch.

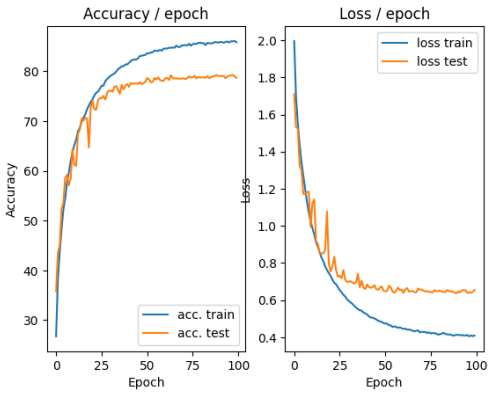
There are many other variants of stochastic gradient descent (SGD) and of learning rate planning strategies such as :
- Momentum SGD : it adds a fraction of the previous gradient to the current gradient helping smooth out oscillations and accelerating convergence in the presence of sparse gradients.
- Nesterov Accelerated Gradient (NAG) : it is a modification of momentum SGD that calculates the gradient using the momentum-adjusted position.
- Adagrad adapts the learning rate for each parameter based on its past gradients. It gives smaller updates to frequently updated parameters and larger updates to infrequently updated parameters useful for sparse data.
- RMSprop is a variant of Adagrad but uses a moving average of squared gradients to adapt the learning rate. This can help alleviate the rapid decay of learning rates in Adagrad.
- Adadelta is a variant of Adagrad that dynamically adjusts the learning rate and uses a moving average of parameter updates. we don't have to manually set a learning rate.
- Adaptive Moment Estimation (Adam) combines momentum and RMSprop

maintaining moving averages of both gradients and their squares.
- Nadam is an extension of Adam that incorporates Nesterov momentum.
- AMSGrad is a modification of Adam that ensures the denominator term of squared gradients never goes to zero.

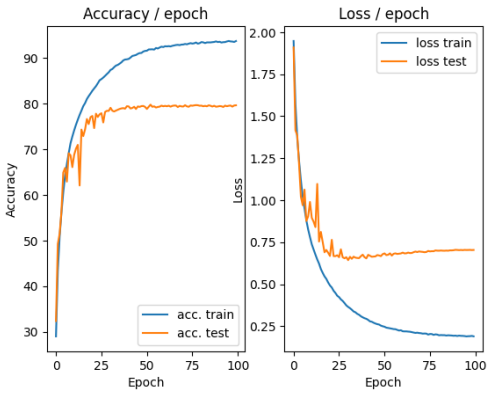
- Cyclical Learning Rates (CLR) cycle the learning rate between predefined bounds during training.
- One Cycle Learning Rate that starts with a low learning rate,then increases to a maximum value, and then decreases back to the initial value during training.
- Learning Rate Range Test (LR Range Test) This strategy involves training a model with exponentially increasing learning rates and monitoring the loss to find an optimal learning rate.

d. Regularization of the network by dropout

29.



(a) Loss and accuracy curves without dropout: Prec@1 **78.64**%



(b) Loss and accuracy curves with **dropout**: Prec@1 **79.64**%

Figure 22: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch.

Thanks to the network regularization provided by dropout, we observed a modest enhancement in accuracy, with a performance of 79.64% compared to 76.46% without dropout. However, the most significant and crucial benefit is the prevention of overfitting, as evidenced by the test loss curve.

30. Regularization refers to a set of techniques used to prevent overfitting and improve the generalization of a model. Regularization methods add constraints or penalties to the model's learning process. These constraints help the model strike a balance between fitting the training data well and being able to make accurate predictions on new data.

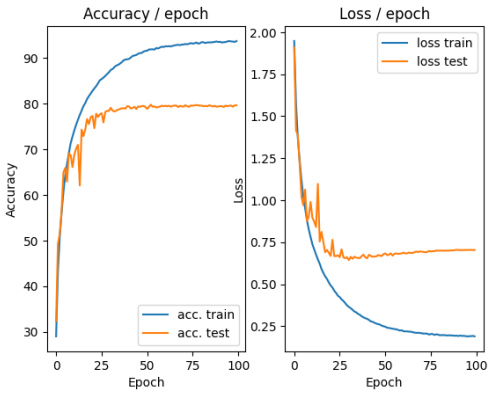
31. One interpretation of dropout is that it creates an ensemble of neural networks within a single network. During training, it achieves this by randomly deactivating individual neurons in each forward and backward pass, effectively sampling a different set of neurons each time. This variation trains multiple subnetworks. During inference, the entire ensemble can be used by keeping all neurons active but scaling their outputs by the dropout probability. This ensemble approach improves the network's ability to generalize and make more robust predictions. Additionally, dropout prevents co-adaptation of neurons, a phenomenon in which neurons in the same layer become highly specialized and overly dependent on a subset of the input features. By encouraging neurons to be robust to the absence of certain inputs during training, dropout promotes a more generalized and stable model.

32. The dropout rate controls the strength of regularization applied to the neural network. With a higher dropout rate more neurons are deactivated during the batch training so it has usually a better generalization. Also a higher dropout rate decreases the training speed.

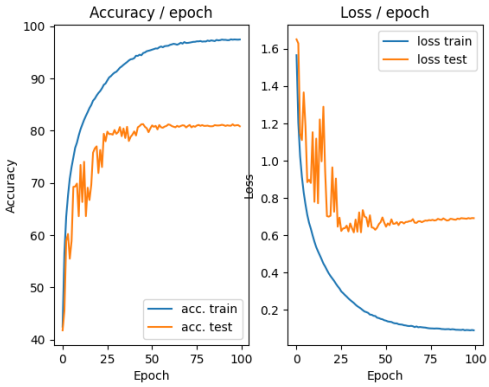
33. During training dropout leads to neuron deactivation and stochasticity. On the contrary, during testing all neurons are activated so it has deterministic prediction. To account for the increased number of active neurons during testing compared to training, the output of the dropout layer is scaled by a factor.

e. Use of batch normalization

34.



(a) Loss and accuracy curves without batch normalization: Prec@1 **79.64**%



(b) Loss and accuracy curves with **batch normalization**: Prec@1 **80.84**%

Figure 23: Loss and accuracy curves in train and in test on the CIFAR10 dataset for a batch size of **256**, learning rate of **0.1** and 100 epoch.

III. Transformers (e)

1. Implementation of a transformer

1. We have created a PatchEmbed module for computer vision. It uses a 2D convolution operation with a 7×7 kernel and a stride of 7 to divide input images of 28×28 into 16 patches. We then reshape and transpose the tensor to obtain a shape (batch, nb_tokens, embedding_dim).

2. We have then implemented the MLP part of the transformer bloch using GELU activation. When applied to a random tensor with shape (32, 16, 128), it successfully returns a tensor of the same shape, demonstrating its functionality for transformer-based models.

3. The main feature in self attention is full interaction between all patches unlike in convolution where the receptive field is possibly not on all image. The regular self-attention method becomes much slower and needs a lot of memory as the input sequence gets larger. It has to store attention information for each pair of tokens, which requires a lot of memory. When working with long sequences, this can be a limitation as it is difficult to use these models on devices with limited computing power and memory.

Equations:

- Input Embeddings:
Let X be the input embeddings, with n the sequence length (embedding dimension) and :

$$X = \{x_1, x_2, \ldots, x_n\}$$

- Query, Key, and Value:
We project the input embeddings into query (Q), key (K), and value (V) vectors using learned weight matrices:

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

with, W_Q, W_K, and W_V weight matrices for the query, key, and value projections respectively.

- Attention:
We calculate the attention scores A between each pair of tokens:

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

with d_k dimension of the key vectors.

Then the attention scores:

$$\text{Attention}(X) = AV$$

4. The multi-head self attention allows the model to capture richer interpretations of the input sequence by considering multiple perspectives and dependencies.

Equations:

- Input Embeddings:
Let X be the input embeddings, with n the sequence length (embedding dimension) and :

$$X = \{x_1, x_2, \ldots, x_n\}$$

We consider h attention heads.

- Query, Key, and Value Projection:
We project the input embeddings into query (Q), key (K), and value (V) vectors for each attention head using learned weight matrices:

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

with, W_Q, W_K, and W_V weight matrices for the query, key, and value projections respectively.

- Reshape and permute:
To achieve independent interactions between each head in a multi-head attention mechanism, we reshape and permute dimensions in a way that separates the heads from one another. This results in h sets of matrices: Q = Q1, Q2, ..., Qh, K = K1, K2, ..., Kh, and V = V1, V2, ..., Vh.

Here is the exemple for the Q matrix:

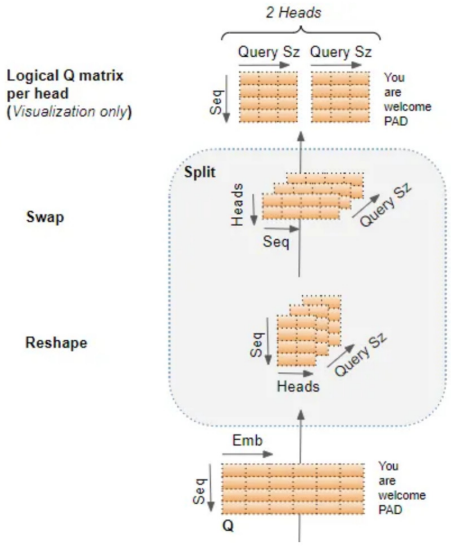


Figure 24: Q matrix split across the Attention Heads. Image from Ketan Doshi in Toward Data Science.

- Attention:

$$A_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$$

with d_k dimension of the key vectors. A_i correspond to the attention for one head

Then the attention scores:

$$\text{Attention}(X) = A_i V_i$$

- Projection:
Finally, we transpose and reshape the Attention Score A, and apply a linear projection.

5. We now build a transformer block :

- Embedded patches:
We first transform our data to have embedded patches. These patches are the result of segmenting the input data into smaller, non-overlapping sections or "patches."

- Layer Normalization (LayerNorm1) and add with multihead attention:
We apply Layer Normalization independently to each embedded patches. Then, we apply multiheads attention on the output of this normalization. The output of multi-head self-attention is added to the input embedded patches (residual connection).

$$z'_l = \text{MSA}(\text{LN}(z_{l-1}) + z_{l-1}$$

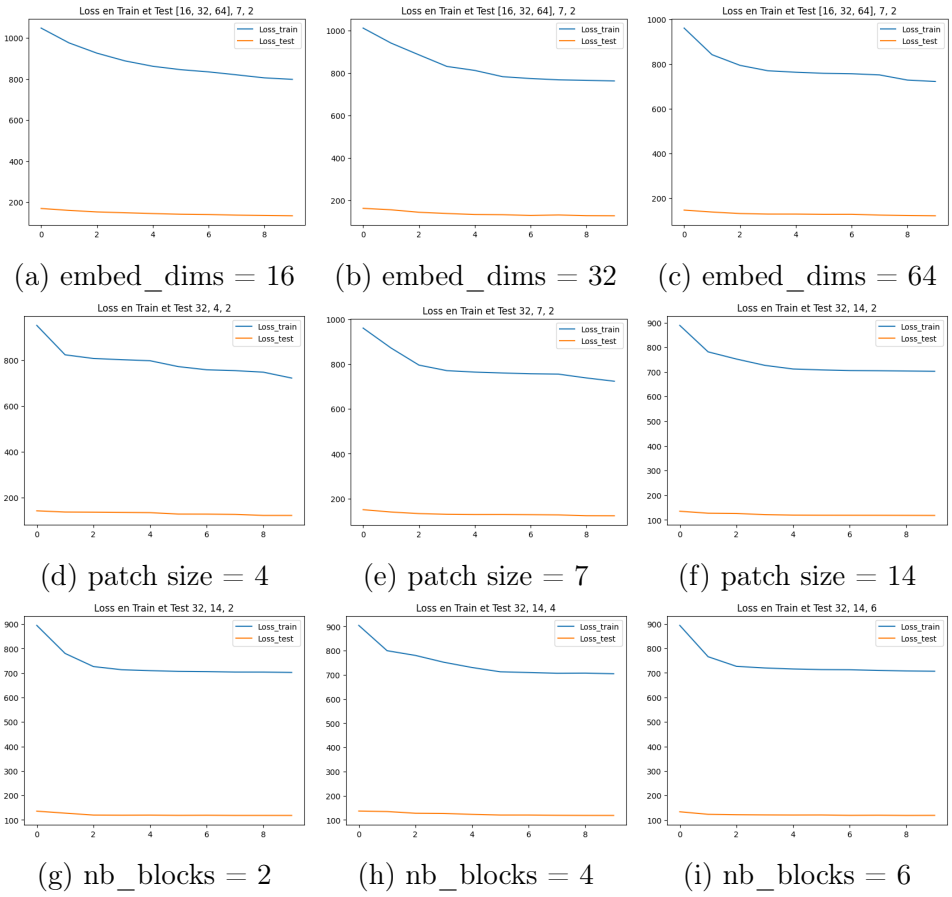
- Layer Normalization (LayerNorm2) and add with MLP: :
We apply Layer Normalization to the output obtained from the previous step. Then, we apply a MLP to this output. Following that, we will add the output to the one of the previous step.

$$z_l = \text{MLP}(\text{LN}(z'_l) + z'_l$$

6. In the context of visual transformers, a "class token" refers to a special token that is initialized randomly and represent the global information about the entire image. Therefore it is the only vector used for the final class prediction

Positional encoding is needed in transformers because otherwise we would loose the spatial information. It allows us to know where a patch comes from in an image.

7.



(a) embed_dims = 16 (b) embed_dims = 32 (c) embed_dims = 64

(d) patch size = 4 (e) patch size = 7 (f) patch size = 14

(g) nb_blocks = 2 (h) nb_blocks = 4 (i) nb_blocks = 6

Figure 25: Loss in train and test for different embed_dim, patch_size, and nb_blocks values

	embed_dim	patch_size	nb_blocks	acccuracy (%)
(a)	16	7	2	79.28
(b)	32	7	2	85.13
(c)	64	7	2	93.27
(d)	32	4	2	92.94
(e)	32	7	2	90.91
(f)	32	14	2	96.7
(g)	32	7	2	96.22
(h)	32	7	4	96.11
(i)	32	7	6	95.3

Table 1: Accuracy for different embed_dim, patch_size, and nb_blocks

Analysis:

- embed_dim:
The embed_dim parameter controls the dimensionality of the token embeddings in the transformer. As we can see on the table 1, increasing the embed_dim generally improves accuracy from an accuracy of 79.28% to 93.27%. This is because higher-dimensional embeddings allow the model to capture more complex features in the input data.

- patch_size:
The patch_size determines the size of the patches given into the transformer. Smaller patch sizes may capture finer details, while larger patch sizes may capture more global information. However, it's important to consider the trade-off between patch size and computational cost. Indeed, smaller patch sizes may require more memory and computation. In the table 1, we can see that the best accuracy occurs for with a patch size of 14 (96.7%).

- nb_blocks:
The nb_blocks parameter controls the number of transformer blocks in the architecture of our transformer. More blocks can potentially capture more complex patterns but may also be computationally expensive. In the table 1, we can see that this parameters doesn't have a great impact on the accuracy (less than a 1 percent difference) but the best accuracy is achieved with 2 blocks.

Combination of the best individual values of parameters:

We have then decided to test a model with the best value for each of

these parameters so embed_dim=64, patch_size=14 and nb_blocks=2 (figure 26). We can see that accuracy is 96.56%. This is one of the best accuracy on the model but we previously had a better accuracy (96.7%) with embed_dim=32, patch=14 and nb_blocks=2. This can be explained by the fact that the combination of best individual parameters does not necessarily leaf to the best overall model performance due to complex interactions in the model, the randomness in training... To improve the results we could try to use a different optimizer and fine-tune its hyperparameters (for instance the learning rate or the weight decay).

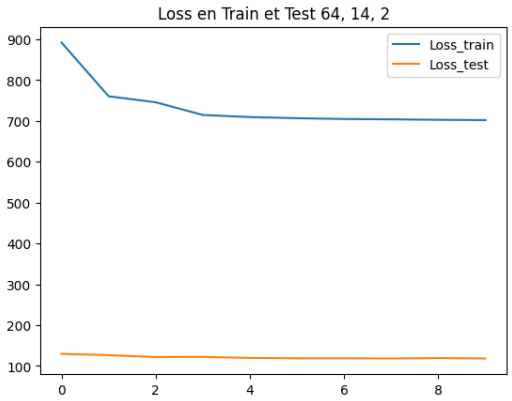


Figure 26: Loss in train and test with embed_dim=64, patch_size=14 and nb_blocks=2 (accuracy = 96.56%)

Complexity of the transformer:

Self-Attention Complexity in terms of token is O(n^2), indeed with N tokens of D dimensionality and Q, K and V of size (N,D), computing Q@K.T is $O(N \times D^2)$ and result in a matrix pre_A of size (N,N). Calculating the softmax of the attention scores on this matrix is: $O(N^2)$ and gives the matrix A of size (N,N). Finally computing the weighted sum of the input tokens A@V: $O(n^2 \times D)$
To improve the complixity of transformers it is possible to distilled Attention by focusing on a subset of important tokens instead of all tokens. It is also possible to use sparse attention matrices to limit the effective operation between tokens. *On The Computational Complexity of Self-Attention Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, Chinmay Hegde 2022* They proposed methods with Dot-Product and Softmax Dot-Product approximations but they showed that it goes with a decrease of accuracy.

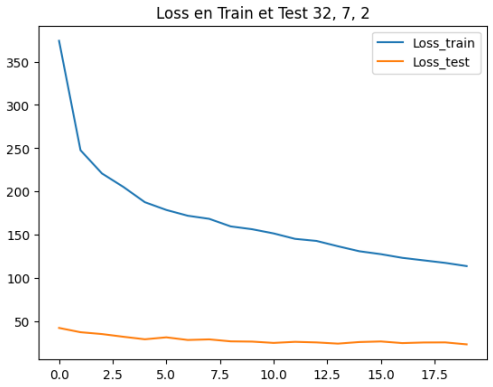
2. Larger transformers

8.a. Trying to apply directly the model using the timm library without pretrained weights causes an error. It is due to the inadequacy between the sizes of the MNIST images (H=28, W=28 , C=1), even with 3 channels, and the shape required by the model (H=224, W=224 and C=3) at the beginning of the backbone. Therefore, the ViT model cannot directly process MNIST images without resizing them to 224x224.

We don't have exactly the same problem with CNNs, indeed the shape problem occurs only in the head of the net when there is a fully connected layer.

8.b. To use this model on MNIST images, we can apply a trick that involves using the img_size and in_chans argument in the create_model function so that the images are reshaped using interpolation.

8.c. We have trained the timm model on the MNIST dataset with and without the ViT-S pretrained on ImageNet. On the figure 27. We can note that with pretraining, the model starts with a lower training loss and converges faster, suggesting that the features learned from ImageNet are beneficial in initializing the model for the specific task. It can helps the model to learn more generic features that can be useful for a lot of tasks including digit classification on the MNIST dataset.



(a) Loss curves **without** the ViT-S **pre-trained on ImageNet**: accuracy in test of **90.13**%

(b) Loss curves with the ViT-S **pre-trained on ImageNet**: accuracy in test of **91.49**%

Figure 27: Loss curves in train and in test on the MNIST dataset for 20 epoch.

8.d. We can employ transfer learning with pretrained models, as demonstrated in this practical, and fine-tune them on our small dataset. As we have observed in previous exercises, we can utilize data augmentation to increase the number of samples. We can also opt for smaller models with fewer parameters, often referred to as 'small' or 'tiny' models. It is crucial to pay attention to parameter initialization, the learning rate, and the choice of optimizer. Moreover, we can implement a technique called early stopping to prevent the model from overfitting

We have seen one technique in this paper: *How to Train Vision Transformer on Small-scale Datasets? Hanan Gani, Muzammal Naseer, Mohammad Yaqub 2022* that uses self-supervised inductive biases learned directly from small-scale datasets and serve as an effective weight initialization scheme. In this situation, a model learns from unlabeled data by creating its own supervision signals. This is often done by generating pre-tasks, where the model tries to solve tasks that are not directly related to the ultimate task of interest. Inductive biases are prior knowledge or assumptions that help the model generalize and learn more efficiently.