Sorbonne Université

M2 Informatique - DAC
RDFIA

Basics on deep learning for vision

Students :
Amayas Sadi
Ghiles Ouhenia

Supervisor :
Mustafa Shukor

Table des matières

1 Intro to Neural Networks

1.1 Supervised dataset

Q.1 : What are the train, val and test sets used for ?

Train Set : Used for training the neural network model by adjusting its parameters to minimize the loss function and learn from the data.

Validation Set (Val Set) : Used for optimizing hyperparameters, preventing overfitting, and assessing the model's performance on unseen data.

Test Set : Used to evaluate the final performance of the trained model on unseen data in a real-world context.

Q.2 : What is the influence of the number of examples N ?

The number of examples N in a dataset can significantly impact a model's performance and generalization. More data typically enhances performance and reduces overfitting, but it may demand greater computational resources. Data quality is critical, and imbalanced datasets may necessitate additional processing, regardless of their size.

1.2 Network architcture (forward)

Q.3 : Why is it important to add activation functions between linear transformations ?

Activation functions are important in neural networks because they introduce non-linearity, enabling the network to learn complex patterns, represent a wide range of functions, and facilitate effective gradient flow during training.

Q.4 : What are the sizes nx, nh, ny in the figure 1 In practice, how are these sizes chosen ?

In Figure 1, the sizes are defined as follows :

nx : The size of the input vector, which is chosen based on the number of attributes in the dataset.

nh : The size of the hidden layer vector, which is selected based on various factors, including the complexity of the problem.

ny : The size of the output vector, which is determined by the number of classes in our data.

Q.5 : What do the vectors $\hat{y}$ and y represent ? What is the difference between these two quantities ?

— $\hat{y}$ represents the predicted values by the model.

— y represents the true or actual values.

— The difference between $\hat{y}$ and y is used to calculate the loss, which the neural network uses in backpropagation to improve its parameters and enhance the model's performance.

Q.6 : Why use a SoftMax function as the output activation function ?

The SoftMax function is used as the output activation function in classification because it converts raw scores into probabilities, is suitable for gradient-based training, and provides information about the relative confidence in class predictions.

Q.7 : Write the mathematical equations allowing to perform the forward pass of the neural network.

— Affine Transformation for Hidden Layer :

$$\tilde{h} = W_h x + b_h$$

— Activation Function for Hidden Layer :

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

— Affine Transformation for Output Layer :

$$\tilde{y} = W_y h + b_y$$

— SoftMax Activation for Output Layer :

$$\hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_{j=1}^{ny} \exp(\tilde{y}_j)}$$

1.3 Loss function

Q.8 : During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function L ?

To minimize the global loss function during training :

— For cross-entropy loss, adjust $\hat{y}_i$ to make predicted class probabilities more similar to true class labels.

— For mean squared error (MSE) loss, adjust $\hat{y}_i$ to make predicted values more similar to true values.

Q.9 : How are these functions better suited to classification or regression tasks ?

— Cross-entropy loss is suitable for classification tasks, where the goal is to assign data points to discrete categories.

— Mean squared error (MSE) loss is suitable for regression tasks, where the goal is to predict continuous numerical values.

1.4 Optimization algorithm

Q.10 : What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions ? Which one seems the most reasonable to use in the general case ?

Classic (Batch) Gradient Descent :

— Advantages : Utilizes the entire dataset for accurate gradient estimates and can converge to the global minimum for convex functions.

— Disadvantages : Computationally expensive, especially for large datasets, and requires storing the entire dataset in memory.

Mini-Batch Stochastic Gradient Descent (SGD) :

— Advantages : Faster convergence than classic gradient descent, particularly for large datasets, and strikes a balance between efficiency and stability.

— Disadvantages : May oscillate due to noise from using subsets of data and can be less stable compared to classic gradient descent.

Online Stochastic Gradient Descent :

— Advantages : Well-suited for online learning with continuously available data and quick adaptation to changing data distributions.

— Disadvantages : Noisy updates may lead to less stable convergence, and convergence to local minima is possible due to high variance in gradient estimates.

In the general case, Mini-Batch Stochastic Gradient Descent is often the most reasonable choice, balancing efficiency and stability.

Q.11 : What is the influence of the learning rate on learning ?

The learning rate η plays a crucial role in training neural networks :

— When it's too small, the network converges slowly and may get stuck.

— With the optimal learning rate, the network converges efficiently to a good solution.

— If the learning rate is too large, there's fast initial progress, but it's at risk of divergence and instability.

Selecting the appropriate learning rate is vital for effective training. Additionally, techniques like Adam can dynamically adjust the learning rate based on training progress, which can help improve convergence.

Q.12 : Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the loss with respect to the parameters, using the naive approach and the backprop algorithm

— The complexity of the naive approach is proportional to the number of parameters ($O(n)$) in the network, where n is the number of parameters. It requires $n + 1$ evaluations of the loss function for each of the parameters, resulting in a quadratic complexity.

— The backpropagation algorithm maintains a relatively constant complexity per layer (typically $O(L)$, where L is the number of layers), as it efficiently computes gradients layer by layer, reducing redundant computations.
As a result, backpropagation is the preferred method for training deep neural networks due to its efficiency.

Q.13 : What conditions must the network architecture satisfy to enable effective optimization through such a procedure ?

To ensure successful optimization using gradient-based methods such as backpropagation, the network architecture must meet several crucial criteria :

— Activation functions within the network must be differentiable to compute gradients at each layer. Common differentiable activation functions include ReLU, Sigmoid, and Tanh.
— The network should follow a feedforward connection structure, allowing information to flow in a single direction, facilitating backpropagation.
— The chosen loss function for evaluating the network's performance must be differentiable concerning the model's predictions. Common differentiable loss functions include mean squared error and cross-entropy.
— Proper initialization of network parameters is essential to prevent issues like vanishing gradients at the beginning of training. Techniques such as Xavier initialization or He initialization are commonly employed.
— The architecture should possess an appropriate number of parameters relative to the available training data. Excessive parameters can lead to overfitting and pose challenges for optimization.

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

The cross-entropy loss (L) used in conjunction with the SoftMax function can indeed be simplified. Let's demonstrate this simplification :
The cross-entropy loss for a single training example is given by :

$$L = -\sum_i y_i \log(y^i)$$

where :
— y_i is the true target or ground truth value for the i-th class.
— y^i is the predicted probability for the i-th class after applying the SoftMax function.
Let's work on simplifying this expression :
1. Expand the summation :

$$L = -\sum_i y_i \log(y^i)$$

2. Rewrite $\log(y^i)$ using the SoftMax function :

$$\log(y^i) = \log\left(e^{\tilde{y}_i} / \sum_i e^{\tilde{y}_i}\right) = \log(e^{\tilde{y}_i}) - \log(\sum_i e^{\tilde{y}_i}) \ = \tilde{y}_i \ - \log(\sum_i e^{\tilde{y}_i})$$

3. Rewrite the loss with this simplification :

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

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

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

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

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

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

$$\begin{aligned}\frac{\partial l}{\partial \tilde{y}_i} &= -y_i + \frac{\frac{\partial}{\partial \tilde{y}_i}(\sum_{j=1} e^{\tilde{y}_j})}{\sum_{j=1} e^{\tilde{y}_j}} \\ &= -y_i + \frac{e^{\tilde{y}_i}}{\sum_{j=1} e^{\tilde{y}_j}} \\ &= -y_i + \text{SoftMax}(\tilde{y})_i\end{aligned}$$

Generalizing, we obtain :

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

Q.16 : Using the backpropagation, write the gradient of the loss with respect to the weights of the output layer.

Starting with $\nabla_{W_y} l$, we have :

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

This can be expressed as a matrix :

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

Firstly, we express $\tilde{y}$ as the product of the hidden layer h, the transpose of weight matrix W_y, and the bias b^y :

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

This expression can be broken down for each element $\tilde{y}_k$ as the sum over the hidden layer neurons h_j with weights $W_{k,j}^y$ and bias b_k^y :

$$\tilde{y}_k = \sum_{j=1}^{n_h} W_{k,j}^y h_j + b_k^y, \text{ where } k \in [1, n_h]$$

The partial derivative of $\tilde{y}_k$ with respect to W_{ij}^y is then given by :

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

Next, we determine $\frac{\partial l}{\partial \tilde{y}_k}$ using the result from a previous question :

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

Combining these results, we find the expression for the gradient of the loss with respect to the weight W_{ij}^y in the neural network.

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

So, the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} l$ is given by :

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

Q.17 : Compute other gradients.

1. The gradient of the loss concerning $\tilde{h}$ is computed using the chain rule :

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

Let's calculate the two terms :

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

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

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

Finally,

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} = \left(\sum_j \delta_j^y W_{j,i}^y \right) (1 - h_i^2) = \delta_i^h$$

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

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

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

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

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

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

So,

$$\frac{\partial l}{\partial W_{i,j}^h} = \sum_k \delta_k^h \times \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} = \delta_i^h x_j$$

$$\nabla_W^h = \nabla_{\tilde{h}}^T l \cdot x$$

3. Finally :

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

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

2 Convolutional Neural Networks

2.1 Introduction to convolutional networks

Q.1 : Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size x × y × z what will be the output size ? How much weight is there to learn ? How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size ?

The output size of a convolutional layer can be calculated using the following formula :

$$\text{Output Size} = x_{out} \times y_{out} \times C$$

with

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

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

Here :

— x, y, z are the dimensions of the input volume.

— k is the kernel size.

— p is the padding.

— s is the stride.

— C is the number of filters, there is 1

The number of weights in a convolutional layer is determined by the size of the kernel. The total number of weights is given by $k \times k \times z \times C$. In the case where C is 1, the number of weights simplifies to $k^2 \times z$. This is because there is a set of weights for each channel in the input volume at every spatial location covered by the kernel.

For a fully-connected layer, the number of weights is the product of the input size and the output size. If the input size is $x \times y \times z$ and the output size is $x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}}$, then the number of weights is $x \times y \times z \times x_{\text{out}} \times y_{\text{out}} \times z_{\text{out}}$. In this case, when $z_{\text{out}} = C = 1$, the number of weights simplifies to $x \times y \times z \times x_{\text{out}} \times y_{\text{out}}$.

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

— Convolutional layers have several key advantages over fully-connected layers. First, they share parameters using filters, which reduces the number of learnable parameters and simplifies training. Second, they're designed to recognize patterns regardless of their location in the input, crucial for tasks like image recognition. They also efficiently capture local features like edges and textures with local receptive fields, acting as a form of regularization to prevent overfitting. Lastly, they are computationally efficient, making them suitable for processing grid-like data such as images and sequences.

— However, convolutional layers come with constraints when it comes to understanding global context. Their main strength lies in recognizing local patterns and they may find it challenging to grasp connections between distant elements of the input.

Q.3 : Why do we use spatial pooling ?

Spatial pooling is a crucial technique in convolutional neural networks (CNNs) for downsampling input feature maps and enhancing the network's performance. By summarizing local information through operations like max pooling or average pooling, spatial pooling reduces computational demands, provides translation invariance, and promotes generalization by focusing on essential features. This downsampling also aids in preventing overfitting by discarding irrelevant details, making the network more robust and computationally efficient. Overall, spatial pooling plays a key role in optimizing CNNs for tasks such as image classification and object detection.

Q.4 : Suppose we try to compute the output of a classical convolutional network (for example the one in Figure 2) for an input image larger than the initially planned size (224×224 in the example). Can we (without modifying the image) use all or part of the layers of the network on this image ?

In convolutional neural networks (CNNs), the convolutional layers work independently of the input size, performing operations on the entire image to create an output image. When combined with pooling layers, which also involve operations on the image, the output size is determined by the input characteristics. Essentially, the input size of the image isn't a hyperparameter for the convolutional layer itself.

However, in a typical CNN, the output from convolutional and pooling layers is flattened for a fully-connected layer. Importantly, the flattened vector size depends on the output image size. So, if you use a larger input image than expected, the network can process the data until the fully-connected layer, which requires a fixed-size input. Mismatches in input size at this point can be challenging, emphasizing the need to carefully plan initial input specifications for smooth network integration.

Q.5 : Show that we can analyze fully-connected layers as particular convolutions.

Fully-connected layers in a neural network can be analyzed as specific convolutions, particularly 1x1 convolutions. This means they perform a global operation on the entire input, much like a 1x1 convolution. The main difference lies in how weights are shared, with a 1x1 convolution sharing weights, while fully-connected layers use separate weights for each connection. In summary, fully-connected layers are a specific form of convolution, performing a global operation on the entire input, and the number of kernels corresponds to the number of neurons in the fully-connected layer.

Q.6 : Suppose that we therefore replace fully-connected by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest ?

Fully connected layers typically require inputs of specific sizes. However, if we replace the fully connected layers in a neural network with their convolutional counterparts, we can effectively process images of different sizes than what the network was originally trained on. When a larger input image than the expected size is used, the convolution operations will generate larger feature maps in the intermediate layers, and these will propagate throughout the network. The final convolutional layers, which take the place of the fully connected layers, will apply their filters across the entire spatial extent of the input feature maps.

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

In the context of convolutional neural networks (CNNs), the receptive field of neurons evolves across different layers of the network. In the first convolutional layer, each neuron focuses on a region of the image equivalent to the size of the applied filter, for example, 3x3 pixels with a 3x3 filter. Progressing to subsequent layers, such as the second convolutional layer, the receptive fields of neurons expand. This expansion results from the integration of information from multiple neurons in the previous layer. The broadening receptive fields enable neurons in deeper layers to capture increasingly complex features from the input. Thus, the entire process promotes hierarchical learning, where early layers specialize in detecting simple patterns such as edges, and deeper layers aggregate these patterns to represent complex structures and semantic features.

2.2 Training from scratch of the model

Q.8 : For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed ?

To maintain the same spatial dimensions at the output as at the input in convolutional layers, you can use appropriate padding and stride values. Here's what you can do :

— **Padding** : To keep the spatial dimensions the same, you need to add padding. The amount of padding added depends on the size of the convolutional kernel. Specifically, for a convolutional kernel of size K x K, you add (K-1)/2 pixels of padding to each side (top, bottom, left, and right) of the input. This ensures that the output size remains the same as the input.

— **Stride** : To keep the spatial dimensions the same, you typically use a stride of 1. A stride of 1 means that the convolutional kernel moves one pixel at a time across the input image. This allows for a direct one-to-one mapping of input and output pixels.

Q.9 : For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed ?

To reduce spatial dimensions by a factor of 2 in max pooling layers, you typically use a stride of 2 without adding any extra padding. With a stride of 2, the max pooling operation evaluates every other pixel in the input, effectively cutting the spatial dimensions in half. This approach is commonly used in convolutional neural networks to perform downsampling, which helps reduce spatial resolution while preserving essential features.

Q.10 : For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.

We process input images with dimensions of $32 \times 32 \times 3$. Following the guidelines from questions 8 and 9, convolution operations utilize a padding of $(k-1)/2$ and a stride of 1, ensuring that the spatial dimensions of the image remain unchanged, only modifying the number of feature maps based on the number of filters. Max pooling operations, on the other hand, have a padding of 0 and a stride of 2, reducing the spatial dimensions of the image by half without affecting the depth dimension determined by the feature maps of the preceding convolutional layer. Taking these parameters into account, the outcomes for each layer are as follows :

conv1 :
— Output size : $32 \times 32 \times 32$
— Number of weights to learn : $32 \times 5 \times 5 \times 3 = 2,400$ parameters
pool1 :
— Output size : $16 \times 16 \times 32$
conv2 :
— Output size : $16 \times 16 \times 64$
— Number of weights to learn : $64 \times 5 \times 5 \times 32 = 51,200$ parameters
pool2 :
— Output size : $8 \times 8 \times 64$
conv3 :
— Output size : $8 \times 8 \times 64$
— Number of weights to learn : $64 \times 5 \times 5 \times 64 = 102,400$ parameters
pool3 :
— Output size : $4 \times 4 \times 64$
fc4 :
— Output size : 1000
— Number of weights to learn : $1000 \times 1024 = 1,024,000$ parameters
fc5 :
— Output size : 10
— Number of weights to learn : $1000 \times 10 = 10,000$ parameters

These calculations reflect the output dimensions of each layer and the number of weights to be learned, determined by the filter sizes and the number of feature maps from the previous convolutional layer.

Q.11 : What is the total number of weights to learn ? Compare that to the number of examples.

This results in a total of 1,191,000 parameters to train. Taking into account a dataset consisting of 50,000 training samples and 10,000 test samples, for a total of 60,000 examples, it justifies a degree of caution regarding the potential for overfitting. To mitigate the risk of overfitting, a considerably larger dataset is needed, ideally at least ten times larger, which would amount to tens of thousands of examples.

Q.12 : Compare the number of parameters to learn with that of the BoW and SVM approach.

In BoW and SVM, the complexity and the number of parameters depend on the chosen visual vocabulary size. For example, using a dictionary of 1000 visual words to classify items into 10 classes means we need to learn 10,000 parameters. It's crucial to highlight that, in this approach, adjusting the settings of our classifier is necessary, especially the method it uses (kernel). As a result, the number of parameters in this approach is much fewer than in our deep Convolutional Neural Network (CNN) model.

Q.14 : In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the the difference in data) ?

The key distinction in the code lies in the absence of an optimizer within the **epoch()** function. Consequently, during training, the network computes gradients and adjusts the model's parameters (backward pass), while in testing, the model's parameters remain unchanged, and its performance is assessed on unseen data without a backward pass.

Q.16 : What are the effects of the learning rate and of the batch-size ?

The learning rate and batch size are critical hyperparameters that exert a significant impact on the performance, speed, and stability of the training process :

Learning Rate :
A high learning rate can induce excessively rapid convergence, risking the overshooting of optimal weights and causing oscillations or divergence, potentially resulting in the model's failure to attain an optimal solution. Conversely, an excessively low learning rate may lead to slow convergence, rendering the training process time-consuming and possibly causing the model to become ensnared in local minima or plateaus. To tackle these challenges, techniques such as learning rate schedules or the application of adaptive learning rate methods like Adam or Adagrad are often employed. These methods help mitigate issues and facilitate efficient training.

Batch Size :
A small batch size introduces more noise into the training process, as the model updates its weights on a limited subset of the data. While this can expedite convergence, it concurrently compromises the stability of the training process and the precision of weight updates. On the contrary, a large batch size provides more stable updates by considering a more extensive portion of the dataset. However, this approach may demand increased

memory and computational resources, potentially slowing down convergence and proving less effective for certain types of models. Thus, the selection of the batch size must be meticulously balanced, taking into account the specific characteristics of the model and resource constraints.

Q.17 : What is the error at the start of the first epoch, in train and test ? How can you interpret this ?

The error at the start of the first epoch, in both training and testing, is typically high and expected. This initial high error is due to the random initialization of the model's weights, the absence of early learning, and the lack of an optimization process to adjust the weights. As training progresses in subsequent epochs, the model learns from the data, gradually reducing the error.

Q.18 : Interpret the results. What's wrong ? What is this phenomenon ?

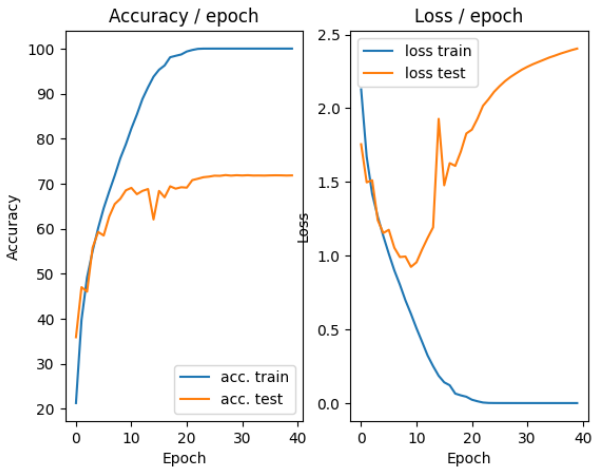

FIGURE 1 – Accuracy / Loss on CIFAR10 dataset

Observing our results, we notice a visible overfitting that becomes apparent around the tenth epoch. The loss curves reveal a divergence, with the loss on the test set increasing while the training set loss continues to decrease. Similarly, accuracy curves indicate a plateau at around 65% for the test set, whereas training accuracy keeps growing after the tenth epoch. These observations suggest that while the model effectively learns from the training data, it struggles to generalize to new test data, underscoring the need to implement methods to mitigate overfitting and enhance generalization.

2.3 Results improvements

Q.19 : Describe your experimental results.

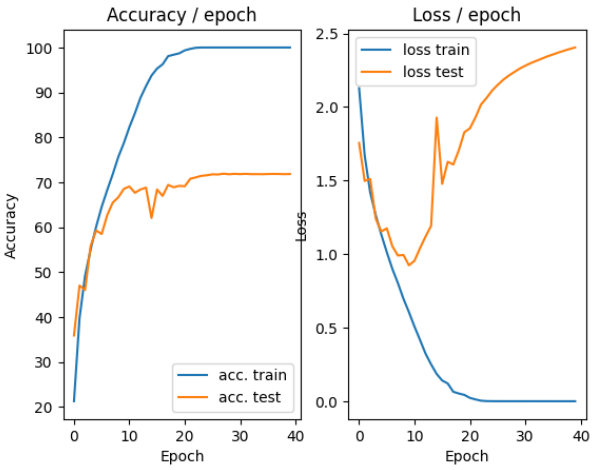

FIGURE 2 – Accuracy / Loss before normalization

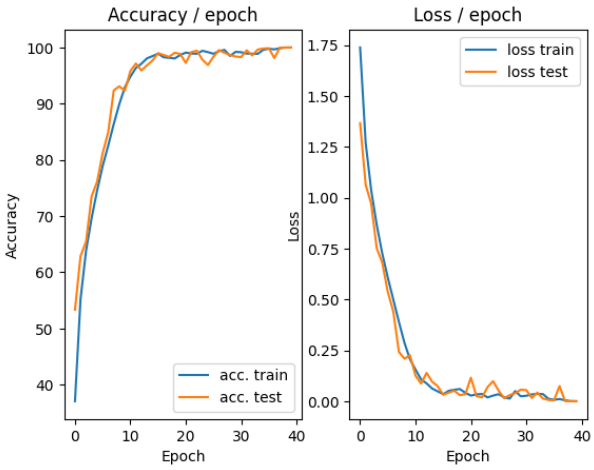

FIGURE 3 – Accuracy / Loss after normalization

The introduction of image normalization in our CNN has yielded significant improvements. Prior to this adjustment, precision and loss curves exhibited pronounced oscillations, especially in the test data, indicating instability in the model's performance. However, following the application of normalization techniques, a remarkable absence of oscillations was observed, with precision and loss curves for both training and test sets displaying striking similarities. Importantly, this normalization also resulted in an overall enhancement of the model's performance, characterized by a notable reduction in overfitting. Precision and loss curves, initially divergent between training and test sets, showed increased convergence, indicating a more robust generalization of the model. These findings underscore the effectiveness of normalization techniques in stabilizing and globally improving the performance of the CNN.

Q.20 : Why only calculate the average image on the training examples and normalize the validation examples with the same image ?

The practice of calculating the average image on the training examples and normalizing the validation examples with the same image is essential to maintain consistency, avoid biases, enable a realistic evaluation, and prevent information leakage when assessing the model's performance. This ensures that the model is tested impartially on data processed in a similar manner, without being influenced by specific characteristics of the validation set.

Q.22 : Describe your experimental results and compare them to previous results.

By deliberately increasing our dataset, we anticipate that our model will be able to generalize more effectively, as it becomes less responsive to the idiosyncrasies of the training images due to the introduction of diversity. Furthermore, the application of random

cropping and flipping introduces a certain level of perturbation in the data, thus contributing to strengthening the model's resilience to variations it might encounter in real-world scenarios.

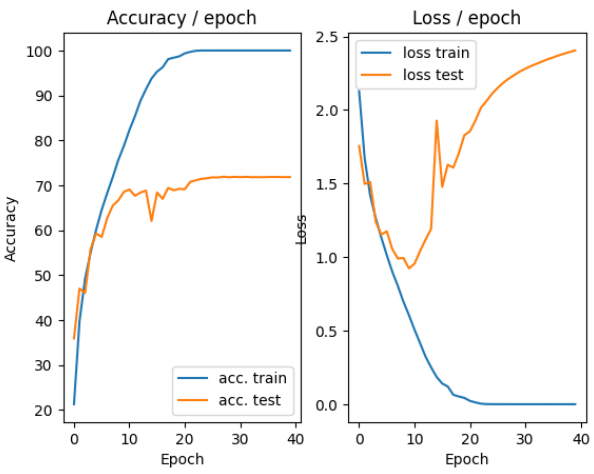

FIGURE 4 – Accuracy / Loss before increasing

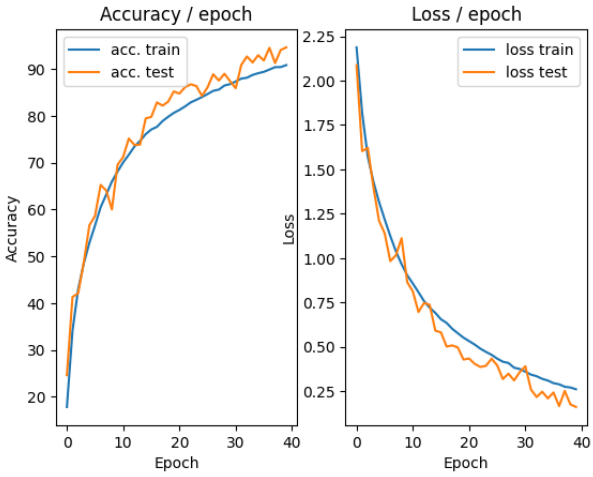

FIGURE 5 – Accuracy / Loss after increasing

Our experimental results corroborate the enhancement of our model's generalization capacity, and we observe a reduction in overfitting, which is a positive development.

Q.23 : Does this horizontal symmetry approach seems usable on all types of images ? In what cases can it be or not be ?

The usability of the horizontal symmetry approach depends on the nature of the images. It is particularly effective for images with clear horizontal symmetry, such as those featuring human faces or symmetrical objects. In such cases, it can enhance model performance. However, it may not be suitable for images without inherent horizontal symmetry, such as landscapes or text-heavy images, where its application could distort the content and lead to incorrect results. The decision to use this approach should be context-specific, carefully considering the nature of the images and the task requirements to determine whether it is beneficial or potentially detrimental.

Q.24 : What limits do you see in this type of data increase by transformation of the dataset ?

Increasing data through transformations has its limitations. One key limitation is that transformations cannot create entirely new information or patterns that do not exist in the original data. They can only exploit existing patterns and variations. Additionally, overuse of data augmentation, especially aggressive transformations, can lead to overfitting, where the model becomes too tailored to the augmented data and may not generalize well to real-world, unseen examples. Moreover, not all transformations are suitable for all types of data ; some transformations may be irrelevant or even harmful in specific contexts. Lastly, the computational cost of applying transformations, especially in real-time or on large datasets, can be a limiting factor. Balancing the benefits of data augmentation with

its potential drawbacks and understanding the specific needs of the task is essential to maximize its effectiveness.

Q.26 : Describe your experimental results and compare them to previous results, including learning stability.

Incorporating a learning rate scheduler controls how quickly or slowly a model adapts to the specific problem at hand. Therefore, we anticipate a more controlled model convergence, as the model's training process benefits from a gradual approach to reaching the global minimum of the loss function by progressively reducing the learning rate. This approach should also help avoid plateaus, which are regions where the model's accuracy does not significantly improve due to slow learning. Ultimately, we expect an enhancement in training stability.

In the following figure, we will use normalized data to prevent overfitting and to better observe the impact of the learning rate scheduler on the convergence speed of our model.

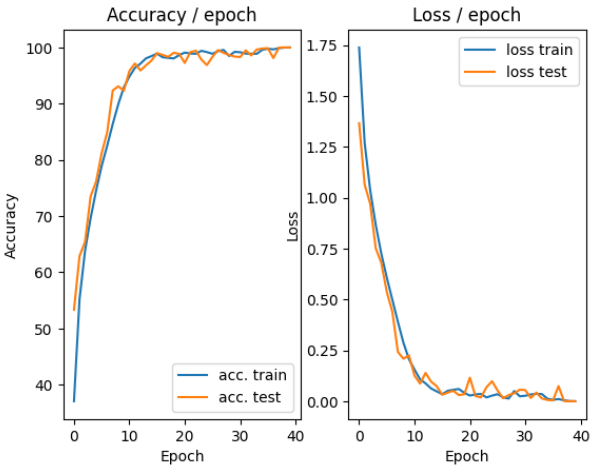

FIGURE 6 – Accuracy / Loss before learning rate scheduler

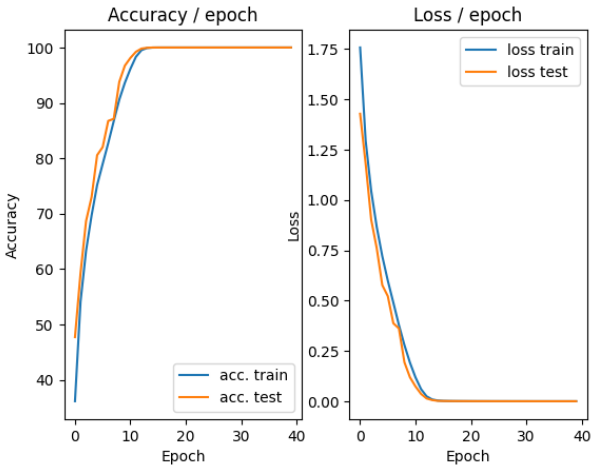

FIGURE 7 – Accuracy / Loss after learning rate scheduler

Comparing it to the baseline results, we observe a higher level of stability in the loss curve.

Q.27 : Why does this method improve learning ?

Exponential decay learning rate scheduling improves learning by dynamically adapting the learning rate to the training process's needs. In the early stages of training, a higher learning rate allows for faster progress, while later on, smaller updates refine the model. Additionally, this method accounts for the varied shape of the loss function, naturally adjusting to steep or complex regions. However, it's crucial to carefully set the initial parameters to avoid too rapid a decay, which limits learning, or too slow a decay, leading to oscillation issues.

Q.29 : Describe your experimental results and compare them to previous results.

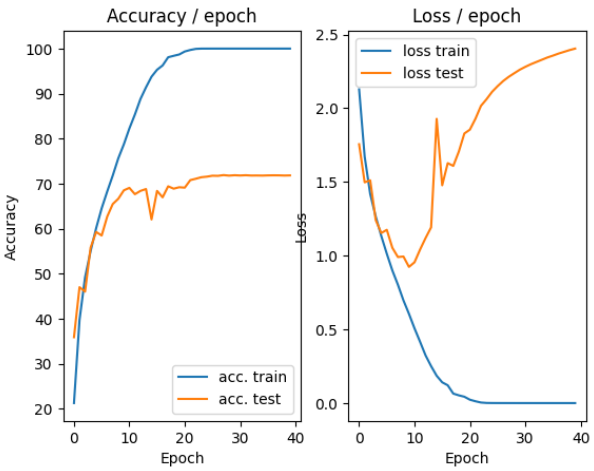

FIGURE 8 – Accuracy / Loss before dropout regularization

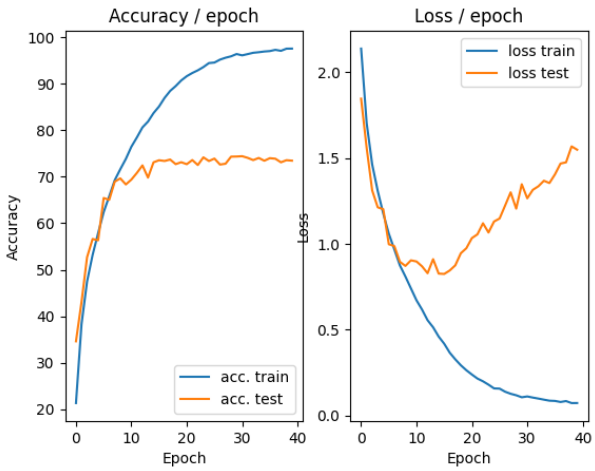

FIGURE 9 – Accuracy / Loss after dropout regularization

The use of "dropout" regularization has improved the regularization of our data, reducing the divergence in test results compared to its absence. However, it is essential to emphasize that "dropout" alone is not sufficient to prevent overfitting in our scenario, as our data has not been normalized.

Q.30 : What is regularization in general ?

In general, regularization is a technique used in machine learning and statistics to prevent overfitting, which occurs when a model becomes too complex and fits the training data too closely, leading to poor performance on unseen data. Regularization adds a penalty term to the model's cost function, discouraging overly complex or extreme parameter values. This helps to balance the trade-off between fitting the training data well and maintaining the model's ability to generalize to new, unseen data. Common types of regularization techniques include L1 regularization, L2 regularization, and dropout in neural networks, each with its own way of penalizing extreme parameter values to improve the model's robustness.

Q.31 : Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it ?

Dropout is a regularization technique in neural networks, and its effect can be interpreted in several ways. One key interpretation is that it acts as an ensemble learning method within a single model by randomly deactivating neurons during training and averaging their predictions during inference. This ensemble approach enhances generalization and robustness. Additionally, dropout reduces co-adaptation among neurons, discourages overfitting, and prevents neuron co-dependency, fostering a more stable and adaptable network. By introducing noise and simplifying the model's capacity, it mitigates the impact of overfitting and enhances the model's ability to generalize by exploring different

pathways. In essence, dropout's effect can be seen as creating a more diverse, robust, and better generalizing neural network.

Q.32 : What is the influence of the hyperparameter of this layer ?

The hyperparameter specific to this layer is known as the "dropout rate," which represents the probability of temporarily deactivating an individual neuron during training. A higher dropout rate leads to more neurons being deactivated during training, potentially enhancing regularization and preventing overfitting. However, an excessively high rate can result in the loss of crucial information, impeding effective learning and potentially causing underfitting. On the other hand, a lower dropout rate retains a larger proportion of active neurons but might not offer adequate regularization, rendering the model susceptible to overfitting, especially in the case of complex, deep networks.

Q.33 : What is the difference in behavior of the dropout layer between training and test ?

The dropout layer in a neural network exhibits distinct behavior between training and testing phases. During training, dropout is active, randomly deactivating some neurons in each iteration to introduce noise and encourage generalization. In contrast, during testing or inference, dropout is disabled, allowing all neurons to be used for producing consistent and deterministic predictions. This behavioral difference is crucial to maintain the reliability of the model's predictions when deployed in real-world applications.

Q.34 : Describe your experimental results and compare them to previous results.

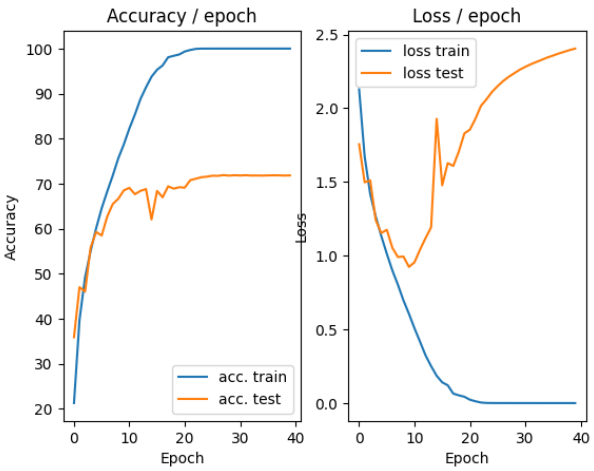

FIGURE 10 – Accuracy / Loss before batch normalization

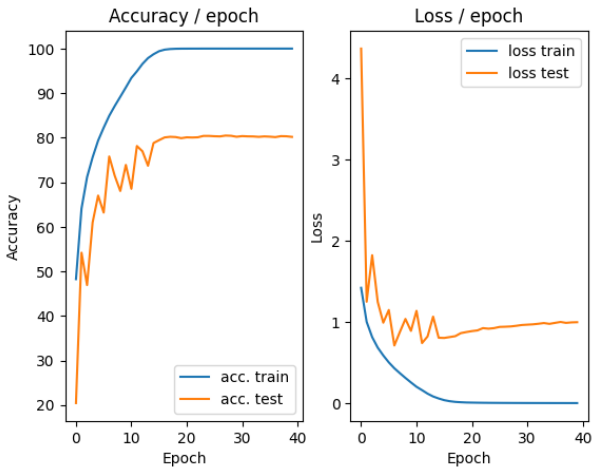

FIGURE 11 – Accuracy / Loss after batch normalization

Analyzing the obtained curves makes the significant impact of adding Batch Normalization to the image classification network's performance evident. In the absence of Batch Normalization, the model shows a tendency to overfit, a trend noticeably mitigated by this technique. The loss curves for both the training and test sets converge more closely,

revealing a substantial improvement in the model's ability to generalize to unknown data. This enhancement is reflected in a clear increase in accuracy on the test set, now reaching 80%. Thus, Batch Normalization plays a crucial role in stabilizing the network's learning, reducing overfitting, and strengthening its ability to provide accurate predictions for new data.

3 Transformers

3.1 Self Attention

What is the main feature of self-attention, especially compared to its convolutional counterpart ? What is its main challenge in terms of computation/memory ?

Self-attention is a key component of transformer architectures and is known for its ability to capture dependencies between different elements in a sequence without regard to their position or distance from each other. Unlike convolutional layers, which have a fixed receptive field, self-attention can model long-range dependencies in the input data.

The main challenge of self-attention in terms of computation and memory is its quadratic complexity with respect to the sequence length. For a sequence of length N, self-attention requires $O(N^2)$ operations and memory. This can become a computational bottleneck when dealing with long sequences, which is why techniques like scaled dot-product attention are often used to mitigate this challenge.

At first, we are going to only consider the simple case of one head. Write the equations

Equations for Self-Attention with a Single Head :

1. First and foremost, it is imperative to compute three distinct linear projections of the input X : the Query Q, the Key K, and the Value V.

$$Q = X \cdot W_q$$
$$K = X \cdot W_k$$
$$V = X \cdot W_v$$

2. The crux of attention lies in the dot product between the Query and the Key, both of which are learned vectors. The objective is to learn Key representations that provide responses to the Query to guide the attention mechanism. The dot product is highest where the model needs to focus its attention.
3. To ensure stable training, it is crucial to consider the scale of the dot products. When the values of the Key and Query vectors are large, dot products can become very large, leading to vanishing gradients during backpropagation when passed through the softmax function. To address this, the dot products are scaled down by a factor of $\sqrt{d_k}$, where d_k is the dimensionality of the Query and Key vectors. This scaling keeps gradients manageable, stabilizing training, and maintains a constant variance of the dot product.

4. The final attention score is computed using the softmax function :

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

5. The Value matrix represents the actual content of the input tokens. Once the attention scores are computed, they are used to weight the Value vectors V.
   Optionally, it is possible to perform a final linear projection at the end. In this case, the attention is calculated as :

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

3.2 Multi-head self-attention

Write the equations of a Multi-Heads Self-Attention

Equations for Multi-Heads Self-Attention :

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h) \cdot W_O$$

Where :

$$head_i = \text{Attention}(Q \cdot W_{Q_i}, K \cdot W_{K_i}, V \cdot W_{V_i})$$

The Attention function within each head follows the same equations as described in the single-head attention, and W_O represents the weights of a final linear layer.

3.3 Transfomer block

Write the equations of a Transformer Block

Operations within a Transformers Block :

1. Start by normalizing the input using Layer Normalization :

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

2. Compute the Query (Q), Key (K), and Value (V) matrices by projecting the normalized input, which are used in the Multi-Head Attention mechanism :

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

3. Pass these matrices into the Multi-Head Attention mechanism to obtain the Attention Output :

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

4. Add the normalized input x to the output of the Multi-Head Attention and apply Layer Normalization again to obtain the Middle Layer Output Norm :

$$\text{MidLayerOutput} = (x + \text{AttOutput})$$

$$\text{MidLayerOutputNorm} = \text{LayerNorm}(\text{MidLayerOutput})$$

5. The MLP head is a two-layer linear network with an activation function GeLU :

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

6. The output of the MLP is added to the Middle Layer Output Norm to obtain the final output of the Transformers block :

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

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

3.4 Full ViT model

Explain what is a Class token and why we use it ?

In Vision Transformer (ViT) models, the class token, positioned as the initial token, is a learnable vector updated throughout the transformer layers. It accumulates information from different parts of the image, serving as a distilled representation of the global context. This enables the model to make informed decisions for image classification by bridging localized details from individual patches with overarching image-wide information.

Explain what is the the positional embedding (PE) and why it is important ?

Positional embeddings (PE) are crucial in transformer-based models like the Vision Transformer (ViT) because these models lack inherent understanding of the order of elements in a sequence. PEs are added to input embeddings to provide the model with spatial information about token positions, enabling an understanding of the sequential order of data. In ViT, PEs are vital for capturing spatial relationships between patches in an image, which is essential for tasks like image recognition. Without PEs, the model would struggle to interpret the spatial arrangement of visual elements in the image.

3.5 Experiment on MNIST

Test different hyperparameters and explain how they affect the performance. In particular embed_dim, patch_size, and nb_blocks. Comment and discuss the final performance that you get. How to improve it ?

- **Variation of embedding dimension :**

The experiment varying the embedding dimension (embed_dim) from 8 to 64 in the Vision Transformer (ViT) reveals a correlation between this hyperparameter and model performance. A higher embedding dimension allows the model to capture more complex patterns, thereby improving accuracy and reducing loss. However, this improvement comes with an increase in the number of parameters, which can lead to longer computation times and more intensive resource usage. The observation that performance curves for dimensions 32 and 64 are similar suggests that, in this scenario, an embedding dimension of 32 strikes an optimal balance between model capacity and computational efficiency. Opting for a dimension beyond 64 does not seem to yield significant advantages while demanding more resources, emphasizing the importance of practical considerations in choosing the embedding dimension.

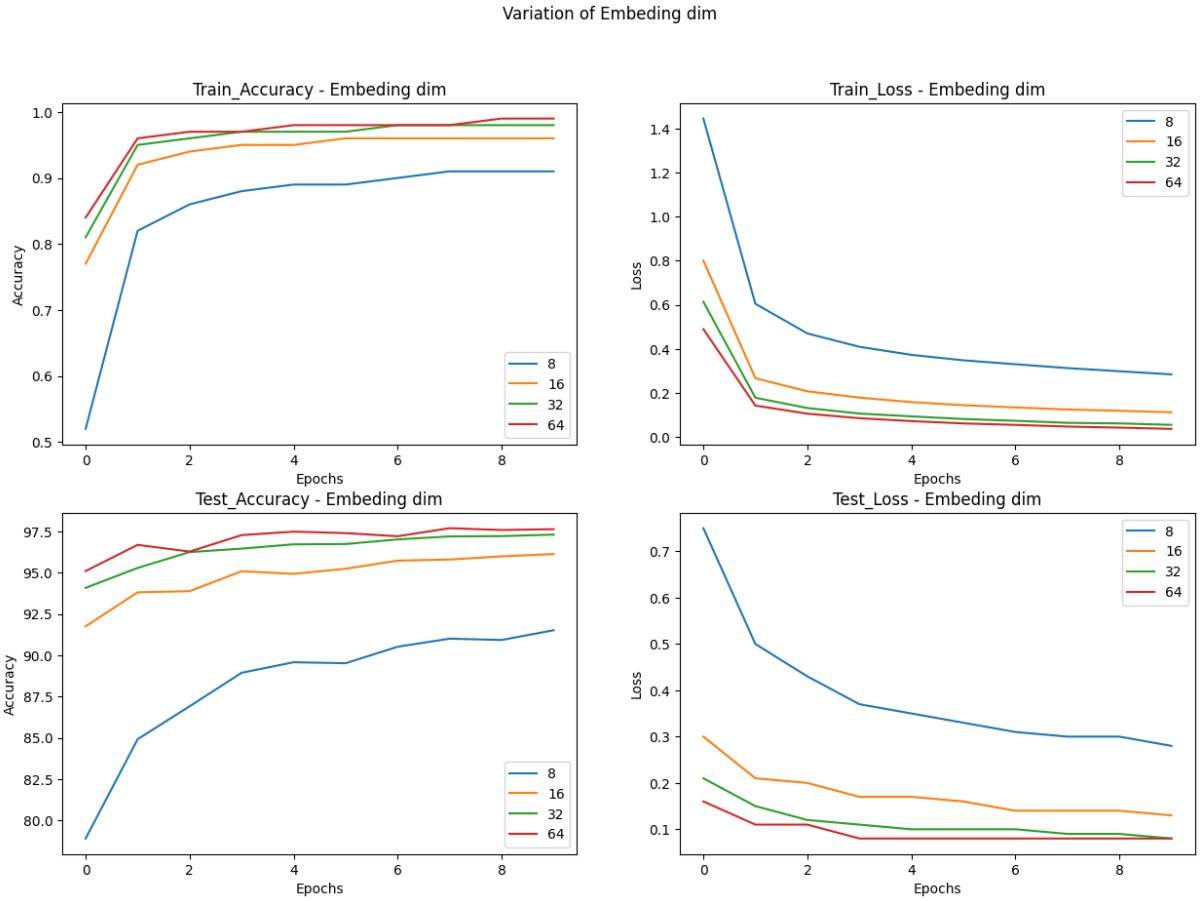

FIGURE 12 – Varation of the embeding dimension

- **Variation of the number of blocks :**

The experiment involved varying the number of blocks, with values of 2, 4, 6, and 8. The resulting curves didn't show significant differences in performance ; all remained closely grouped. However, a notable observation emerged : as the number of blocks increased, a trend of instability in the values of loss and test accuracy became apparent. This suggests that adding additional blocks may lead to increased complexity, resulting in learning difficulties and potential instability in performance. While adding blocks can potentially enhance the model's capacity to learn complex representations, there is a point where it may lead to undesirable side effects, such as the observed instability in this scenario. Thus, the choice of the number of blocks should be guided by a balance between model complexity and the ability to generalize effectively on the dataset.

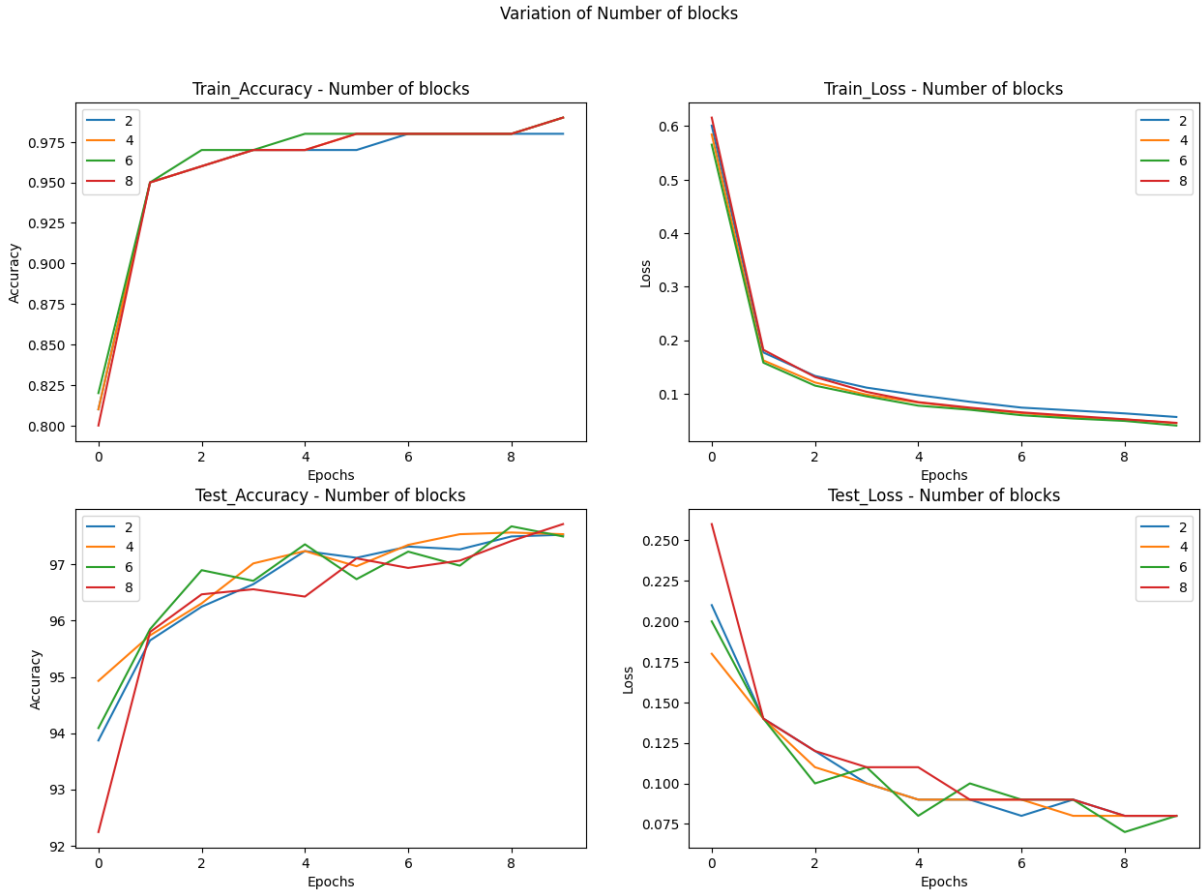

FIGURE 13 – Varation of the number of blocks

- **Variation of the patch size :**

The experiment involved varying the patch size in a Vision Transformer, testing values of 4, 7, 11, and 14. The resulting curves exhibit a notable trend : as the patch size increases, both training and testing performance improve. This observation can be explained in the context of MNIST classification, where capturing global context is crucial for identifying extended patterns in the images. By increasing the patch size, the model can better grasp the spatial relationships between pixels, resulting in enhanced classification capability. The findings underscore the significance of choosing the right patch size in a ViT for specific tasks, emphasizing the need to strike a balance between local resolution and capturing global context based on the characteristics of the classification task.

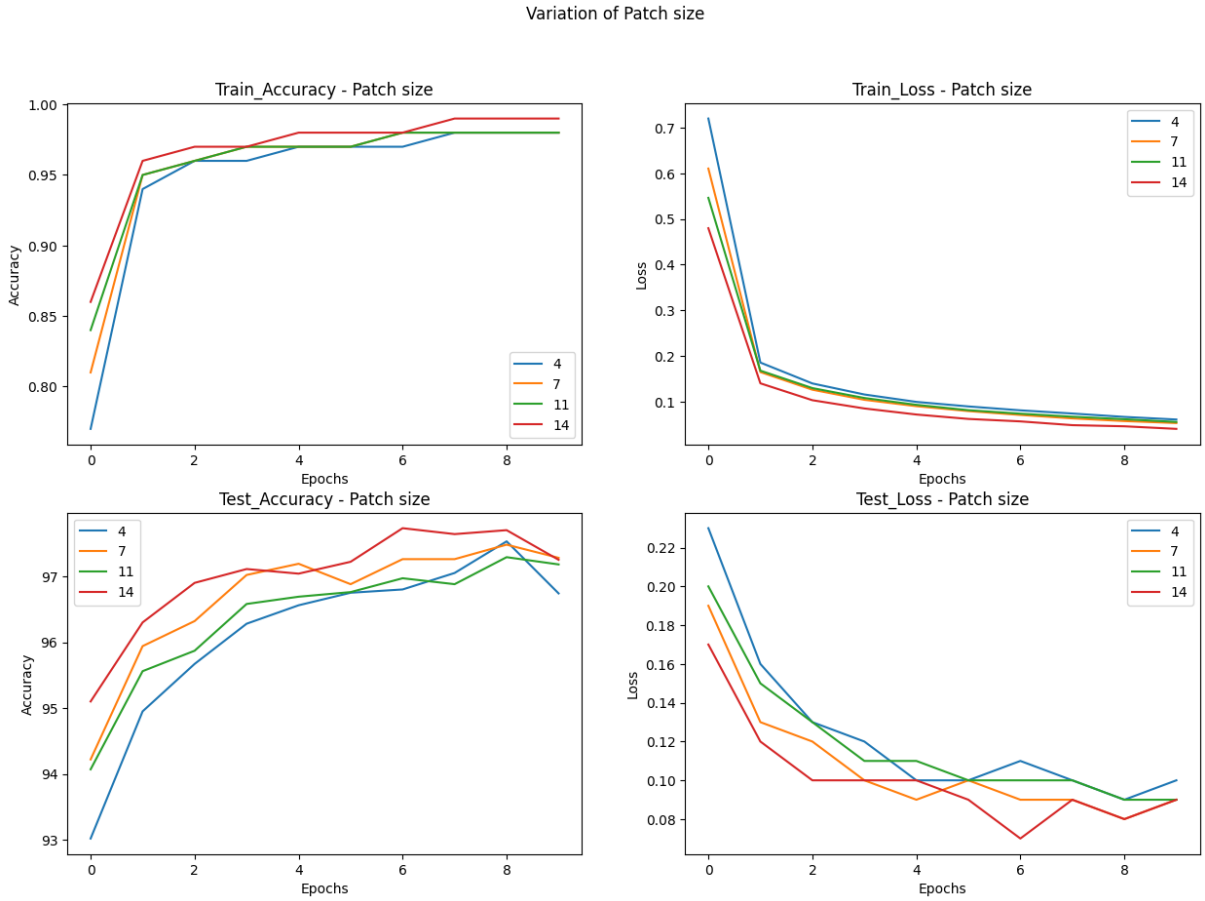

FIGURE 14 – Varation of the patch size

3.6 Larger transformers

What it is the problem and why we have it ?

The "vit_base_patch16_224" model is trained on 224x224 images from the ImageNet dataset. To utilize this model effectively, it's important to maintain the same image size as the one it was designed for. This means resizing the input images to 224x224 RGB pixels.