RDFIA : Section 1

Report 1 : Basics on deep learning for vision

Simay Celik, 28713301

Aylin Soykok, 28711545

Sorbonne University - Faculty of Science and Engineering

Master DAC

Academic year: 2024/2025

Contents

1 Introduction

This homework will cover the basics of deep learning for computing vision where we focus on important models such as Neural Networks, Convolutional Neural Networks (CNNs) and Transformers. These networks are inspired by the architecture of our brain and mimic our pattern recognition ability with deep learning.

Neural Network : As explained in the HW instructions, a neural network is an architecture where we apply sequential mathematical transformations on a dataset "to pass from the space of *features* to the space of *predictions*".

Convolutional Neural Networks : These NNs are adapted for signal inputs like images and speeches. They have different layers such as the convolutional layer, pooling layer and fully-connected layer[2] and in 1-cd, we will create one and try to improve the model.

Transformer Models : These NNs are adapted to learn and track the context in sequential data[10]. They were first created for language models but are quite adaptable. In this part, we will introduce concepts like self attention, implement the model and test different hyperparameters to see their influence.

In this project, we will explore these models through a variety of experiments by adjusting parameters and architectures in order to observe their impact on model performance in visual tasks. We will also base our work with theoretical foundations and answer the questions on the subject.

2 1-ab: Intro to Neural Networks

2.1 Introduction

In the first section, we will start with an introduction to neural networks. Neural networks are sequentially connected nodes which can process the data layer by layer, inspired by the human brain. We will first respond to some questions to better understand what a neural network is. By doing this, we will explain in detail how the sequential calculations are done and how we test the model which will give us insights in the practical part. Afterwards, we will code this network ourselves and do the calculations by hand. Then we will integrate existing modules to create an optimized network. We will also cover how *batch_size* and *learning_rate* can affect the performance of the model, how it works on different data and what an SVM is.

Moving forward, our network structure will be based on the architecture provided in the instruction paper shown in Figure 1.

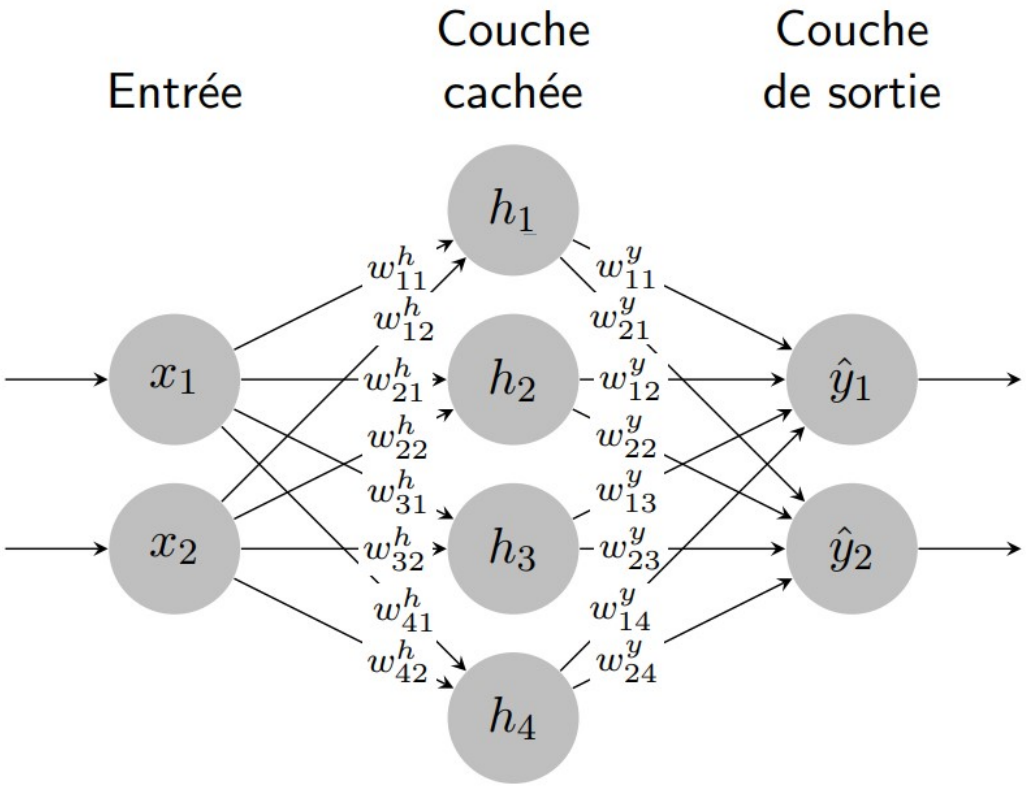


Figure 1: Neural network architecture with only one hidden layer

2.2 Section 1 - Theoretical foundation

2.2.1 Supervised dataset

⋆ **Q1** **train set:** This set is used during the learning phase where we fit the parameters of our model. **validation set:** In neural networks, during training, we set aside a validation set different from the train set and try our model on it to have an unbiased evaluation while we tune the hyperparameters. We monitor the performance continuously during training to avoid overfitting.
test set: This set contains examples different from those in the train and validation set to test the performance of the final trained classifier. [11]
Sometimes, the difference between the validation set and the test set is ambiguous. In the implementation part, we test the performance of our model at each iteration which would normally correspond to validation but we will consider them as test.

Q2 In most cases, a large number of examples lead to better performance because we have more information and more generalization. However, as the dataset gets larger, the training takes longer.

2.2.2 Network architecture (forward)

Q3 A neuron is activated by an activation function. By calculating the weighted sum of its output, we define the importance of the output. In addition, we introduce non-linearity to the network. Otherwise, using a sequence of linear transformations would result in a linear transformation and the NN would perform like a perceptron.[7] But with non-linear activation functions in between two layers, we allow

back-propagation which would otherwise not be possible because the derivative of a linear function is a constant. [1]

⋆ **Q4** n_x = number of input features n_y = number of output classes n_h = number of hidden units These sizes depend on the problem at hand, n_x is chosen based on the input data, it is the dimension of an example. n_y is equal to the number of target classes. We do not have a predefined value for n_h, it is optimized based on the data and experimentation. Having a large n_h can add complexity to the model but it it is too big, we risk overfitting.

Q5 During supervised training, we have sets of features and targets that we keep as ground truth. $y \in \{0,1\}^{n_y}$ represents a true label. $\hat{y} \in [0,1]^{n_y}$ or $\hat{y} \in [-1,1]^{n_y}$ represents the output classes predicted by the model. The difference is that $\hat{y}$ is a score based on the probability of an example belonging in a class defined by the activation function whilst y directly indicates the class.

Q6 The SoftMax function converts the outputs into a probability distribution which sums to 1.

Q7 $\widetilde{h}$ corresponds to the output the affine transformation in the hidden layer: $\widetilde{h} = X(W_h^T) + b_h^T$ where W_h a matrix of weights of size $n_h \times n_x$ and b_h a bias vector of size n_h

h defines the output vector of the activation function applied on the output: $h = tanh(\widetilde{h})$

$\widetilde{y}$ is the output of the linear transformation in the output layer: $\widetilde{y} = hW_y^T + b_y^T$ where W_y a matrix of weights of size $n_y \times n_h$ and b_y a bias vector of size n_y

$\hat{y}$ defines the output of the activation function SoftMax: $\hat{y} = SoftMax(\widetilde{y})$

2.2.3 Loss function

Q8 The cross-entropy measures the difference between the true probability distribution and the predicted distribution. For cross-entropy, $\hat{y}_i$ should get closer to the true target probability distribution y_i. The mean squared error calculates the mean squared difference between each target and the predicted label. Hence, $\hat{y}_i$ should directly get closer to y_i.

Q9 Cross-entropy is more suited to classification tasks because we measure the difference between probability distributions. It penalizes confident incorrect predictions more than the uncertain ones. However, in regression tasks, our goal is to predict a continuous value so there are no discrete labels. MSE calculates how far the model predicts from true values.

2.2.4 Optimization algorithm

Q10 Classic gradient descent: Stable but slower because it computes the gradient of the entire dataset in each epoch. This helps reduce large fluctuations in the cost function. However, it is computationally expensive and the training time is long.

Online stochastic gradient descent: Has the fastest convergence because it updates the parameters after every single example. It is also the noisiest because as it updates each example individually, the variance tends to increase.

Mini-batch stochastic gradient descent: Proposes a compromise between stability and speed. It goes through a subset of the data. But in this case, we have to define a suitable batch size as it has an impact on the performance.

⋆ **Q11** Learning rate controls the size of changes made to the weights and biases during each epoch. This influences the speed at which the model converges, ideally, it should assist the model to converge after an acceptable number of iterations.
High learning rate: A high learning rate can cause it to converge too quickly and potentially miss the optimal solution, causing the loss to increase. In this case, it can also oscillate around the optimal solution and diverging.
Low learning rate: A low learning rate can stabilize the convergence by slowing it down but this also means it requires more iterations, increasing computational time. If the number of iterations is not sufficient, then we might not reach the local minima.
Below, on Figure 2, we applied gradient descent on x^2 with different learning rates to see the possible behaviors. We observe how a higher rate can cause oscillations and how a lower rate can not be enough to learn. In the middle, we see a suitable learning rate for this function.

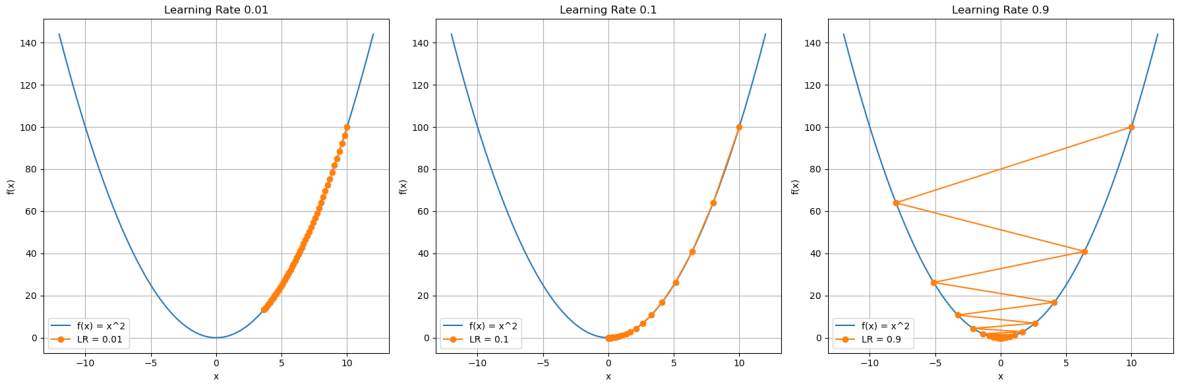


Figure 2: Effects of different learning rates

⋆ **Q12** **Naive approach** The naive approach recalculates the gradient at each layer independently. This increases the computational lost because the gradient is computed for the same parameters in all layers. Hence, as the number of layers increase, computational complexity increases exponentially. So, it is inefficient approach.

Backpropagation The backprop algorithm on the other hand, reuses the previously calculated derivatives while calculating gradients. So, for each parameter, it calculates its gradient with a single pass through the layers by reusing intermediate calculations. This is called the chain rule. It reduces the time complexity as it does not repeat calculations. As seen in Figure 3 which was provided in the instructions, starting from the loss, we go back in the network calculating the derivatives by reusing previously calculated derivatives.

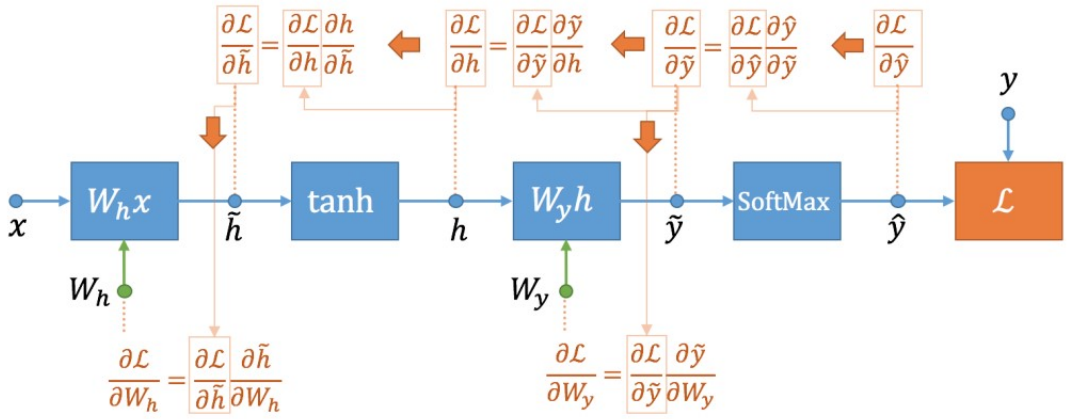


Figure 3: Schematic view of the backpropagation on a neural network

Q13 We calculate the gradient with respect to each parameter, so all the activation and loss functions in the architecture should be differentiable. The layers must be sequential and connected allowing the chain rule from the output layer until the input layer.

Q14 With CE-Loss, the loss is equal to : $\ell = -\sum_i y_i \log(\hat{y_i})$ Given that SoftMax : $\hat{y_i} = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}$, we can substitute it for $\hat{y}_i$:

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

With the logarithmic property : $\log(\frac{a}{b}) = \log(a) - \log(b)$

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

Using the logarithmic property on exponential : $log(e^a) = a$

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

Let's apply distribution:

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

$\sum_i y_i = 1$ for only one component due to the one hot encoding and is 0 for the rest so we can remove the sum in the second part of the equation.

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

j and i do not make a difference in this case as they represent the same values so we can rewrite the equation using only i.

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

Q15 In the previous question, we simplified the cross-entropy loss equation : $\ell = -\sum_i y_i\tilde{y}_i + log(\sum_i e^{\tilde{y}_i})$ Let's calculate the gradient of the loss relative to the intermediate output $\tilde{y}$. For a class k:

$$\frac{\partial \ell}{\partial \tilde{y_k}} = -y_k + \frac{\partial}{\partial \tilde{y_k}} \log(\sum_i \exp \tilde{y}_i)$$

With the property : $\frac{\partial}{\partial x}\log(x) = \frac{1}{x}$:

$$\frac{\partial \ell}{\partial \widetilde{y}_k}\log(\sum_i \exp \widetilde{y}_i) = \frac{1}{\sum_i \exp \widetilde{y}_i} \cdot \frac{\partial}{\partial \widetilde{y}_k}(\sum_i \exp \widetilde{y}_i)$$

$\frac{\partial}{\partial \widetilde{y}_k}(\sum_i \exp \widetilde{y}_i)$ is 0 everywhere except $k = i$:

$$\frac{\partial \ell}{\partial \widetilde{y}_k}\log(\sum_i \exp \widetilde{y}_i) = \frac{\exp \widetilde{y}_k}{\sum_i \exp \widetilde{y}_i}$$

Therefore we have :

$$\frac{\partial \ell}{\partial \widetilde{y}_k} = -y_k + \frac{\exp \widetilde{y}_k}{\sum_i \exp \widetilde{y}_i}$$

Where $\frac{\exp \widetilde{y}_k}{\sum_i \exp \widetilde{y}_i} = SoftMax(\widetilde{y}_k) = \hat{y}_k$ Finally we have,

$$\frac{\partial \ell}{\partial \widetilde{y}_k} = \hat{y}_k - y_k$$

Thus,

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

Q16 Using the backpropagation, we can write the gradient of the loss with respect W_y : By using the chain rule :

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \widetilde{y}_k} \frac{\partial \widetilde{y}_k}{\partial W_{y,ij}}$$

where k indexes the network outputs.
Furthermore, $\frac{\partial \ell}{\partial \widetilde{y}_i} \frac{\partial \widetilde{y}_i}{\partial W_{y,ij}}$ is the only non zero term in the sum so we can simplify the equation :

$$\frac{\partial \ell}{\partial W_{y,ij}} = \frac{\partial \ell}{\partial \widetilde{y}_i} \frac{\partial \widetilde{y}_i}{\partial W_{y,ij}}$$

We already calculated $\frac{\partial \ell}{\partial \widetilde{y}_k}$ in Q15 : $\frac{\partial \ell}{\partial \widetilde{y}_k} = \hat{y}_k - y_k$

From Q7, we know that $\widetilde{y} = hW_y^T + b_y^T$ where W_y a matrix of weights of size $n_y \times n_x$ and b_y a bias vector of size n_y

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

$$\widetilde{y} = \begin{bmatrix} h_1 & h_2 & \dots & h_{n_h} \end{bmatrix} \odot \begin{bmatrix} W_{y,11} & \dots & W_{y,1n_y} \\ \vdots & \ddots & \vdots \\ W_{y,n_h1} & \dots & W_{y,n_hn_y} \end{bmatrix} + \begin{bmatrix} b_{y,1} \\ b_{y,2} \\ \vdots \\ b_{y,n_y} \end{bmatrix}$$

$$\widetilde{y} = \begin{bmatrix} W_{y,11}h_1 & \dots & W_{y,1n_y}h_{n_h} \\ \vdots & \ddots & \vdots \\ W_{y,n_h1}h_1 & \dots & W_{y,n_hn_y}h_{n_h} \end{bmatrix} + \begin{bmatrix} b_{y,1} \\ b_{y,2} \\ \vdots \\ b_{y,n_y} \end{bmatrix}$$

$$\widetilde{y} = \begin{bmatrix} \sum_{j=1}^{n_h} W_{y,1j}h_j + b_{y,1} \\ \vdots \\ \sum_{j=1}^{n_h} W_{y,n_y}h_j + b_{y,n_y} \end{bmatrix}$$

In this case for i, $\widetilde{y}_i$ represents the i-th row in $\widetilde{y}$: $\widetilde{y}_i = \sum_{j=1}^{n_h} W_{y,ij} h_j + b_{y,i}$ Let's replace $\widetilde{y}_i$ with this expression in the derivative to find $\frac{\partial \widetilde{y}_i}{\partial W_{y,ij}}$

$$\frac{\partial \widetilde{y}_i}{\partial W_{y,ij}} = \frac{\partial}{\partial W_{y,ij}} \sum_{j=1}^{n_h} W_{y,ij} h_j + b_{y,i}$$

$b_{y,i}$ does not depend on $W_{y,ij}$ so it is a constant :

$$\frac{\partial \widetilde{y}_i}{\partial W_{y,ij}} = \frac{\partial}{\partial W_{y,ij}} \sum_{j=1}^{n_h} W_{y,ij} h_j$$

$\frac{\partial \widetilde{y}_i}{\partial W_{y,ij}} = h_j$ in only one occasion and is 0 otherwise.

Thus,

$$\frac{\partial \ell}{\partial W_{y,ij}} = \frac{\partial \ell}{\partial \widetilde{y}_i} \frac{\partial \widetilde{y}_i}{\partial W_{y,ij}} = (\hat{y}_i - y_i) \cdot h_j^T$$

With this we can calculate the gradient :

$$\nabla_{W_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1 n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y 1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_y n_h}} \end{bmatrix} = \begin{bmatrix} (\hat{y}_1 - y_1) \cdot h_1 & \cdots & (\hat{y}_1 - y_1) \cdot h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y}) \cdot h_1 & \cdots & (\hat{y}_{n_y} - y_{n_y}) \cdot h_{n_h} \end{bmatrix} = \begin{bmatrix} h_1 & \cdots & h_{n_h} \end{bmatrix} \begin{bmatrix} (\hat{y}_1 - y_1) \\ \vdots \\ (\hat{y}_{n_y} - y_{n_y}) \end{bmatrix}$$

We do the same for $\nabla_{b_y} \ell$: Using the same logic from before, only one value of the sum is not zero because of the one hot encoding.

$$\frac{\partial \ell}{\partial b_{y,i}} = \sum_k \frac{\partial \ell}{\partial \widetilde{y}_k} \frac{\partial \widetilde{y}_k}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \widetilde{y}_i} \frac{\partial \widetilde{y}_i}{\partial b_{y,i}}$$

Once again, we only need to calculate the second part.

$$\frac{\partial \widetilde{y}_k}{\partial b_{y,i}} = \frac{\partial (h W_y^T + b_y^T)_k}{\partial b_{y,i}} = \frac{\partial b_{y,k}}{\partial b_{y,i}} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

Hence,

$$\frac{\partial \ell}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \widetilde{y}_i} \frac{\partial \widetilde{y}_i}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \widetilde{y}_i} \cdot 1 = (\hat{y}_i - y_i)$$

We have:

$$\nabla_{b_y} \ell = \begin{bmatrix} (\hat{y}_1 - y_1) \\ \vdots \\ (\hat{y}_{n_y} - y_{n_y}) \end{bmatrix}$$

⋆ **Q17** In previous questions, we calculated the backpropagation until the second node which is the activation function TanH depicted in Figure 3. As we apply the backpropagation algorithm, we will use the gradients we already derived to calculate the rest of the architecture.
Computing $\nabla_{\widetilde{h}} \ell$:
We can apply the chain rule because ℓ depends on $\widetilde{y}$ which depends on $\widetilde{h}$.

$$\frac{\partial \ell}{\partial \widetilde{h}_i} = \sum_j \frac{\partial \ell}{\partial h_j} \cdot \frac{\partial h_j}{\partial \widetilde{h}_i}$$

- First part: $\frac{\partial \ell}{\partial h_k}$:

$$\frac{\partial \ell}{\partial h_k} = \sum_k \frac{\partial \ell}{\partial \widetilde{y}_k} \cdot \frac{\partial \widetilde{y}_k}{\partial h_j}$$

– We already calculated $\frac{\partial \ell}{\partial \widetilde{y}_k}$ in Q15: $\frac{\partial \ell}{\partial \widetilde{y}_k} = (\hat{y}_k - y_k)$.

– We learned that $\widetilde{y}_k = \sum_{a=1}^{n_h} W_{y,ka} h_a + b_{y,k}$, so the second derivative can be written as:

$$\frac{\partial \widetilde{y}_k}{\partial h_j} = \frac{\partial}{\partial h_j} \left(\sum_{a=1}^{n_h} W_{y,ka} h_a + b_{y,k} \right)$$

Given that b_y is constant and the sum is non-zero only when $a = j$:

$$\frac{\partial \widetilde{y}_k}{\partial h_j} = \frac{\partial}{\partial h_j} \left(W_{y,kj} h_j \right) = W_{y,kj}.$$

Thus, we have:

$$\frac{\partial \ell}{\partial h_k} = \sum_k \frac{\partial \ell}{\partial \widetilde{y}_k} \cdot \frac{\partial \widetilde{y}_k}{\partial h_j} = (\hat{y}_k - y_k) \cdot W_{y,kj}.$$

- The second part is easier to calculate: $\frac{\partial h_j}{\partial \widetilde{h}_i}$.

  By definition, $h = \tanh(\widetilde{h})$, so we can directly calculate the derivative:

$$\frac{\partial h_j}{\partial \widetilde{h}_i} = \frac{\partial}{\partial \widetilde{h}_i} \tanh(\widetilde{h}_j).$$

The derivative of the hyperbolic tangent is defined as:

$$\frac{\partial}{\partial x} \tanh(x) = 1 - \tanh^2(x).$$

Therefore,

$$\frac{\partial h_j}{\partial \widetilde{h}_i} = \begin{cases} 1 - \tanh^2(\widetilde{h}_j) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

Given that $h = \tanh(\widetilde{h})$, we also have:

$$\frac{\partial}{\partial \widetilde{h}_i} \tanh(\widetilde{h}_j) = 1 - \tanh^2(\widetilde{h}_i) = 1 - h_i^2.$$

Now that we calculated each part, we can do the overall calculation.

$$\frac{\partial \ell}{\partial \widetilde{h}_i} = \sum_j \frac{\partial \ell}{\partial h_j} \cdot \frac{\partial h_j}{\partial \widetilde{h}_i} = \sum_j (\sum_k (\hat{y}_k - y_k) \cdot W_{y,kj})(1 - h_i^2)$$

The sum on j depends on h_i so it has only one non null value for $(1 - h_i^2)$ which is when j=i so,

$$\frac{\partial \ell}{\partial \widetilde{h}_i} = (\sum_k (\hat{y}_k - y_k) \cdot W_{y,ki})(1 - h_i^2)$$

In vector form, it can be represented as :

$$\nabla_{\widetilde{h}} \ell = (\nabla_{\widetilde{y}} \ell \cdot W_y) \odot (1 - h^2),$$

Where $\nabla_{\widetilde{y}} \ell = \hat{y} - y$.

Computing $\nabla_{W_h} \ell$: Using the chain rule,

$$\frac{\partial \ell}{\partial W_{h,ij}} = \sum_k \frac{\partial \ell}{\partial \widetilde{h}_k} \frac{\partial \widetilde{h}_k}{\partial W_{h,ij}}$$

- We already calculated $\frac{\partial \ell}{\partial \widetilde{h}_i}$, and we will denote it by $\delta_{h,i}$.
- $\widetilde{h}_k = \sum_{j=1}^{n_x} W_{h,kj} x_j + b_{h,k}$. Knowing this, we can directly calculate the derivative:

$$\frac{\partial \widetilde{h}_k}{\partial W_{h,kj}} = \frac{\partial}{\partial W_{h,kj}} \left(\sum_{j=1}^{n_x} W_{h,kj} x_j + b_{h,k} \right) = \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

because $b_{h,k}$ is a constant.

Thus,

$$\frac{\partial \ell}{\partial W_{h,ij}} = \delta_{h,i} x_j$$

and

$$\nabla_{W_h} = \nabla_{\widetilde{h}}^T x$$

We Compute $\nabla_{b_h} \ell$ with the same steps : Using the chain rule,

$$\frac{\partial \ell}{\partial b_{h,i}} = \sum_k \frac{\partial \ell}{\partial \widetilde{h}_k} \frac{\partial \widetilde{h}_k}{\partial b_{h,i}}$$

- We already calculated $\frac{\partial \ell}{\partial \widetilde{h}_i}$, and we will denote it by $\delta_{h,i}$.
- $\widetilde{h}_k = \sum_{j=1}^{n_x} W_{h,kj} x_j + b_{h,k}$. Knowing this, we can directly calculate the derivative:

$$\frac{\partial \widetilde{h}_k}{\partial b_{h,i}} = \frac{\partial}{\partial b_{h,i}} \left(\sum_{j=1}^{n_x} W_{h,kj} x_j + b_{h,k} \right) = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

because $W_{h,kj} x_j$does not depend on b_h and is constant.

Thus,

$$\frac{\partial \ell}{\partial b_{h,i}} = \delta_{h,i}$$

and

$$\nabla_{b_h} = \nabla_{\widetilde{h}}^T$$

2.3 Section 2 - Implementation

In this part, we are going to implement this network with PyTorch and experiment on the effects of different hyperparameters.

2.3.1 Part 1 : Forward and backward manuals

First, we coded the given neural network model from scratch. Let's train our model on the CirclesData represented in Figure 4. We created a non linearly separable dataset where a circle of one class is surrounded by a ring of another class. In total, we have 200 train examples and 200 test examples.

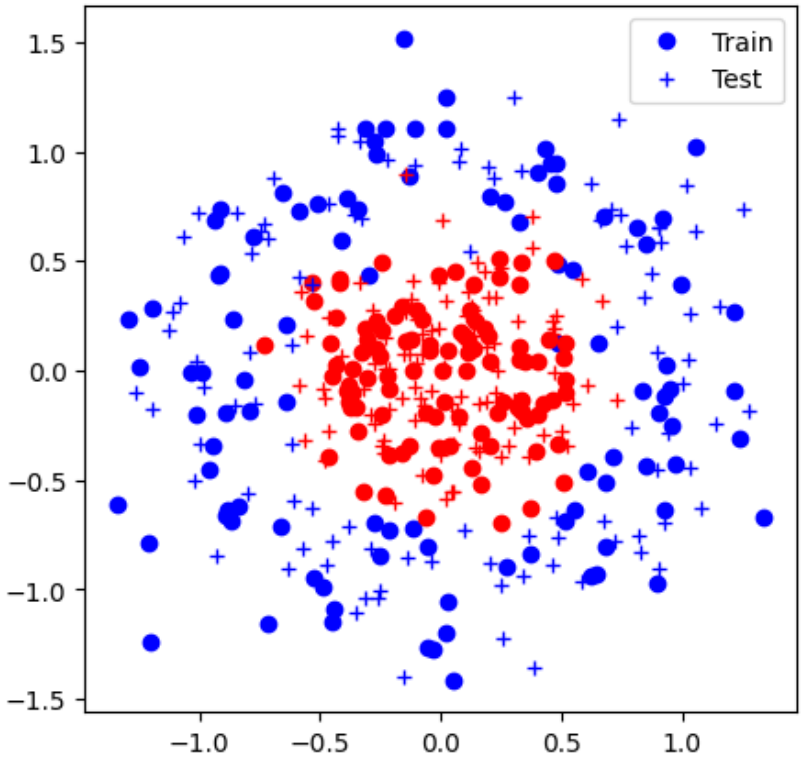


Figure 4: Circular Data Separated in Train/Test

Initial experiment To see if our model is working, we will first test it with: learning rate $\eta = 0.03$, $nb_{iter} = 150$, $batch_{size} = 10$.
In Figure 5, we see classification zones throughout the first 25 iterations of our handmade neural network. Although we ran it 150 times, after the first 20 iterations, the model performs well and after that, the accuracy slightly fluctuates between $93 - 96\%$. We can observe the behavior of our model, unlike a linear classification, by modifying the weights and using non-linearity, we encapsulate the areas corresponding to different classes. And as we use a probabilistic approach with SoftMax, we can see the ambiguity in the colors. Blue and yellow representing certainty of different classes and green representing uncertainty.

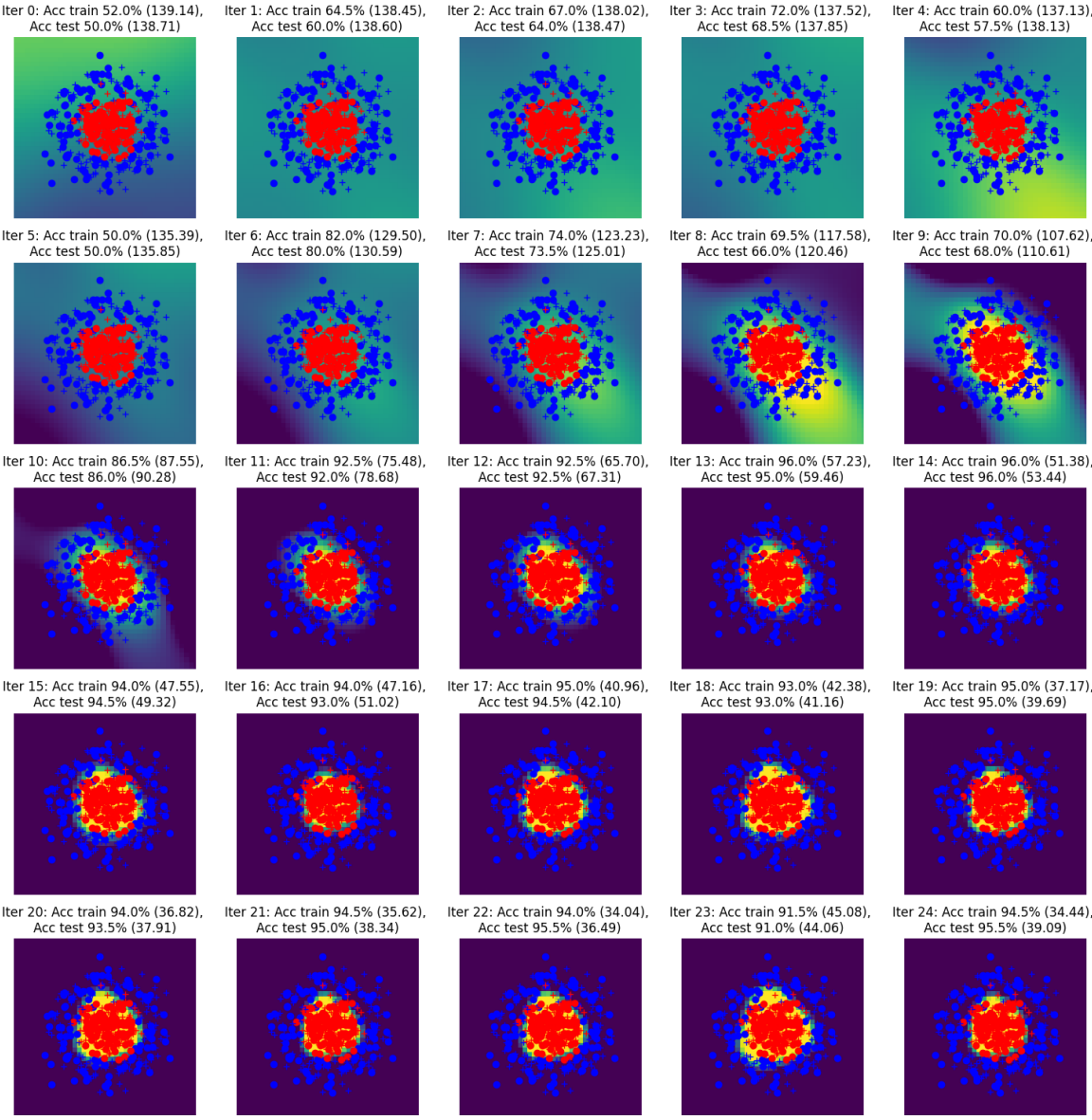


Figure 5: Classification : Forward and backward passes "by hands"

In the figure below 6, we see the evolution of the models accuracy and loss through the iterations during the training phase. As we mentioned above, the model reaches a sufficient performance in 20 iterations and the accuracy and loss for train and test fluctuate between decent values. The evaluation between the train set and the test set behave similarly, this means we do not observe any over-fitting even after many supplementary iterations because our model is not too complex for the data.

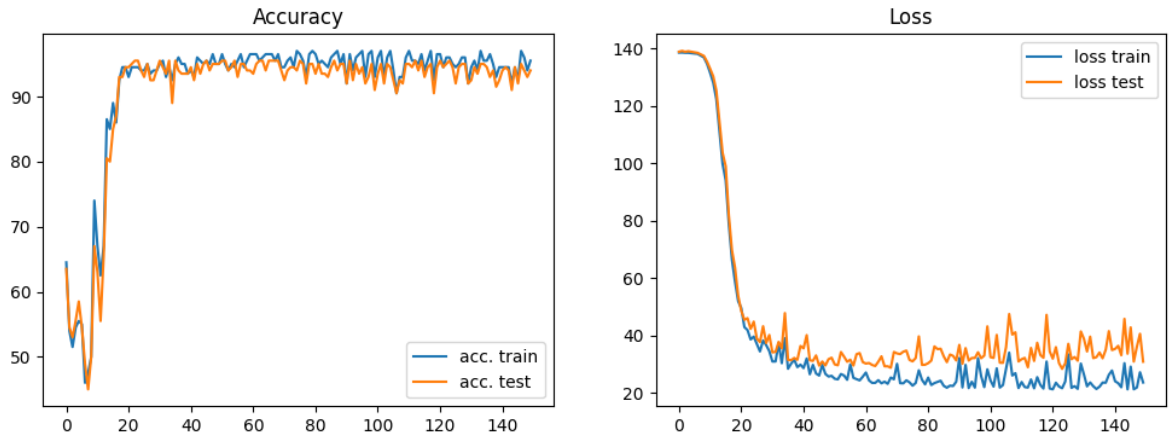
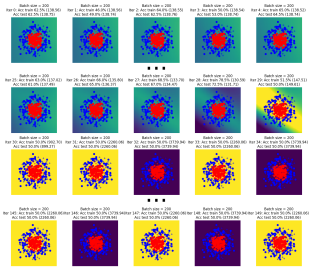
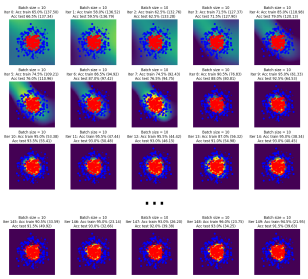


Figure 6: Accuracy and Loss / Iterations

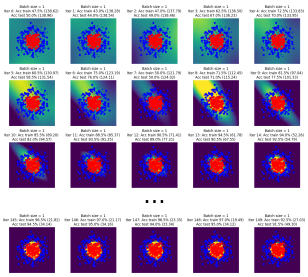
Experiment: Influence of batch size In order to see the influence of the batch_size parameter, we will run the model on different values. We fixed other hyperparameters $\eta = 0.03$, $nb_{iter} = 150$ and ran the model with $batch_{size} = 200 = N$ which is batch gradient descent because we update the model after evaluating all training examples, $batch_{size} = 100$, $batch_{size} = 50$, $batch_{size} = 10$ and $batch_{size} = 1$ which is stochastic gradient descent where we update the model by one example at a time.



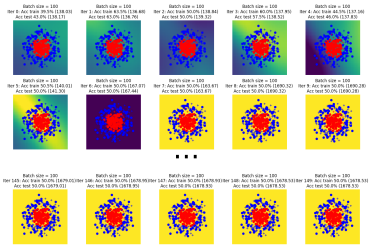
(a) batch size = 200 (Batch Gradient Descent)



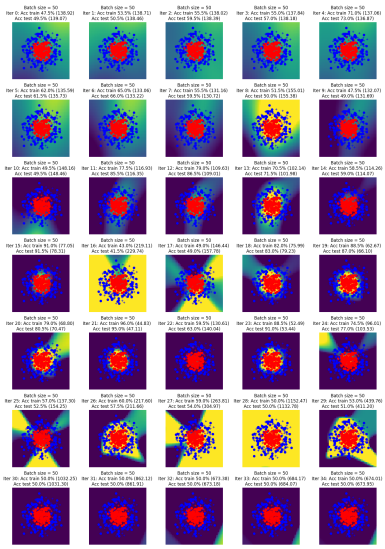
(b) batch size = 10 (Mini Batch Gradient Descent)



(c) batch size = 1 (Stochastic Gradient Descent)



(d) batch size = 100 (Mini Batch Gradient Descent)



(e) batch size = 50 (Mini Batch Gradient Descent)

Figure 7: Evolution of Decision Boundary by Batch Size

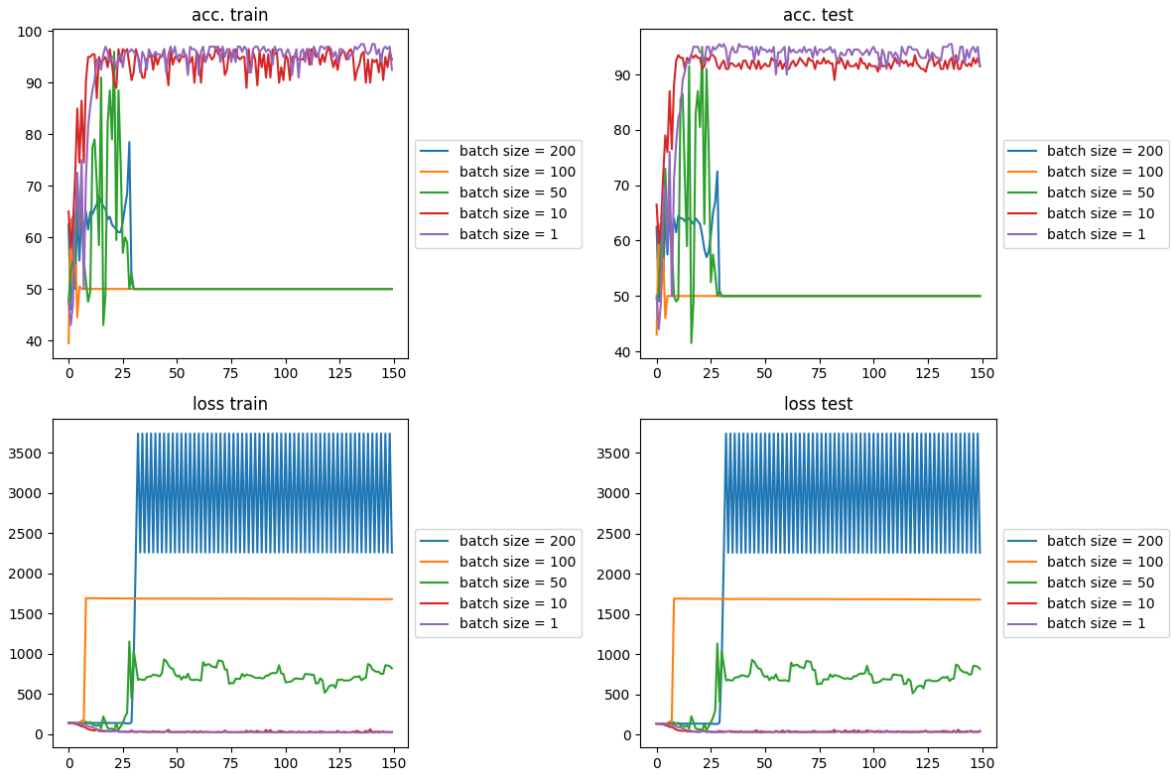


Figure 8: Accuracy and Loss / Iterations by Batch Size

In Figure 7 above, we see parts of how the decision boundary progressed based on the batch size and in Figure 8, we see the evolution of accuracy and lost for the train set and the test set for each batch size.

In Figure 7a, for $batch\_size = 200 = N$, we observe an oscillation pattern meaning that the model diverged. As we mentioned before, this means that the learning rate is too high. However, in Figure 7b, where $batch\_size = 10$, the model gradually converges and is stable hence a suitable learning rate. This is because when we apply Batch Gradient Descent, the batch size is large and there are fewer updates at each iteration, as a result, the steps are too large and unstable. When we apply the same learning rate on a smaller batch however, the weights are updated more frequently and the model takes smaller steps. We can observe the oscillation on Figure 8, with the loss of $batch\_size = 200 = N$.

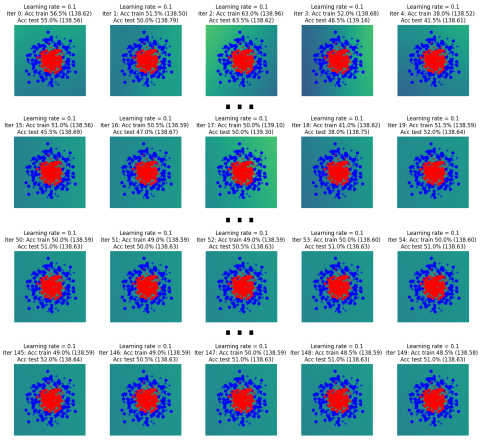
When $batch\_size = 100$ as seen on Figure 7d, we take smaller steps than $batch\_size = 200 = N$ but the learning rate is still too big. This time however, the convergence is on one direction towards one of the classes causing a divergence.

When $batch\_size = 50$, the learning rate is still relatively too big and the model overshoots the optimal solution. We can observe this on Figure 7e, at some point, the decision boundary is accurate and we can see the peak of accuracy in Figure 8, but then it drops significantly. This means that the model is not stable.

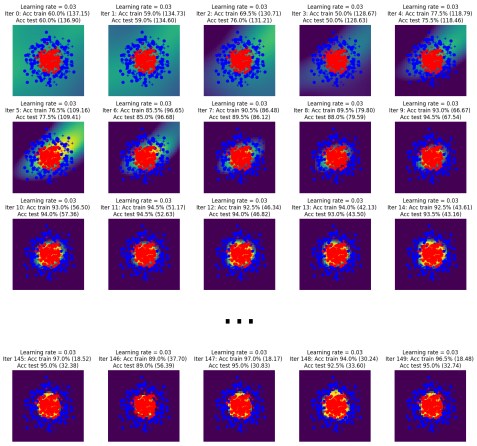
We also applied a Stochastic Gradient Descent with $batch\_size = 1$, just like $batch\_size = 10$, it works well with this learning rate. Because we update the weights at each example, the model takes really frequent but really small steps. As there were enough number of iterations, the model converged nicely.

Experiment: Influence of learning rate This time, we will observe the influence of learning rate parameter η, we will run the model on different values. We fixed other hyperparameters $batch\_size =$

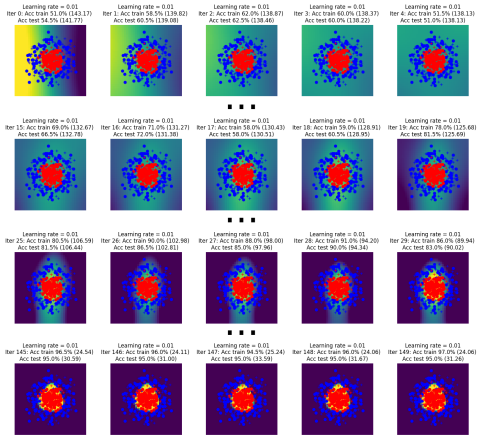
10, $nb_{iter} = 150$ and ran the model with $\eta \in [0.1, 0.03, 0.01, 0.001]$. From last experiment, we know that $batch\_size = 10$ works well with $\eta = 0.03$.



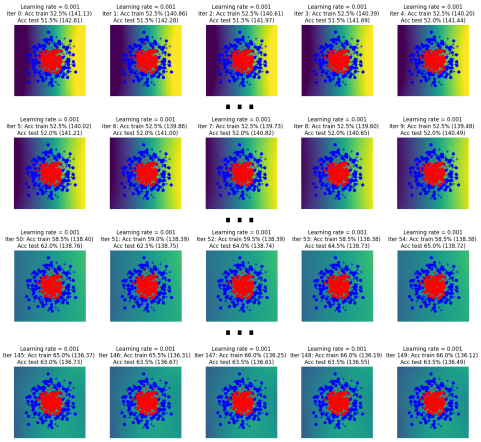
(a) Learning Rate = 0.1



(b) Learning Rate = 0.03



(c) Learning Rate = 0.01



(d) Learning Rate = 0.001

Figure 9: Evolution of Decision Boundary by Learning Rate

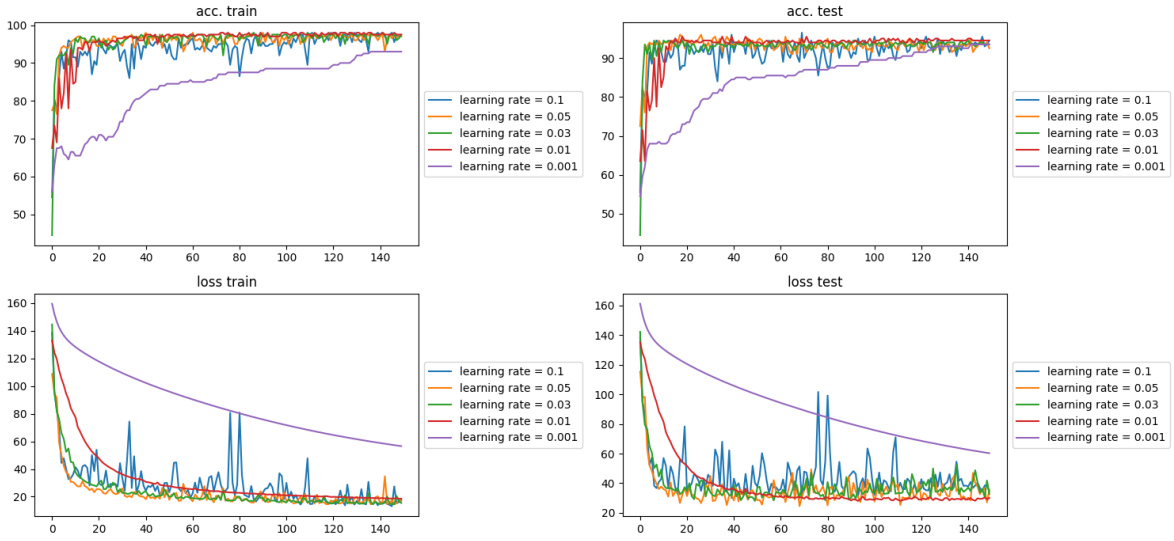


Figure 10: Accuracy and Loss / Iterations by Learning Rate

We can see in Figure 9 b and c, the evolution of the decision boundary based on the learning rate. $\eta = 0.03$ and $\eta = 0.01$ worked well with our batch size but between them, we can observe how having a smaller learning rate increases the calculation time. It is because decreasing the learning rate makes the model take smaller steps and it converges slowly. In Figure 10, we can see that the model is stable and is converging. There are just little fluctuations because the number of iteration is more than necessary. However, if the learning rate is not fitting for the batch size, it can also cause divergence. For instance, in Figure 9 a, we see how the model behaved with $\eta = 0.1$: the learning rate is too big and the model can not find the minima, diverging in ambiguous classification. It would be more efficient to use a random classifier in this case. Its loss and accuracy do not improve. When the learning rate is too little, $\eta = 0.001$, the steps taken by the model can end up being too small and the model can't converge in time before the limit on the number of iterations is passed. This happened here as seen in Figure 9.

From these experiments, we can conclude that the relation between the batch size and the learning rate can highly influence the model and it is important to tune them accordingly. The batch size influences the effect of learning rate so we have to adapt one to another.

2.3.2 Part 2 : Simplification of the backward pass with *torch.autograd*

In the first part, we had a function to apply backward propagation on the parameters and we calculated the gradient by hand. But with *requires_grad=True*, we can use the automatic differentiation mechanism of our parameters directly on our loss. We changed the model accordingly and experimented on batch_size and learning rate once again.

Experiment on autograd: Influence of batch size In Figure 11 below, we see the results from different batch sizes. Just like before, we fixed other hyperparameters $\eta = 0.03$, $nb_{iter} = 150$ and tested $batch\_size \in [N, 100, 50, 10, 1]$.

Once again, $batch\_size \in 1, 10$ worked without any problems and converged in the first 20 iterations. However, this time, we see that $batch\_size \in 50, 100, 200$ converged as well, though slowly. This might be because autograd has better tactics to scale the gradients. Nevertheless, we can see a tiny bit of oscillation pattern with Batch Gradient Search once again showing us that the learning rate is too big when it comes to larger batches.

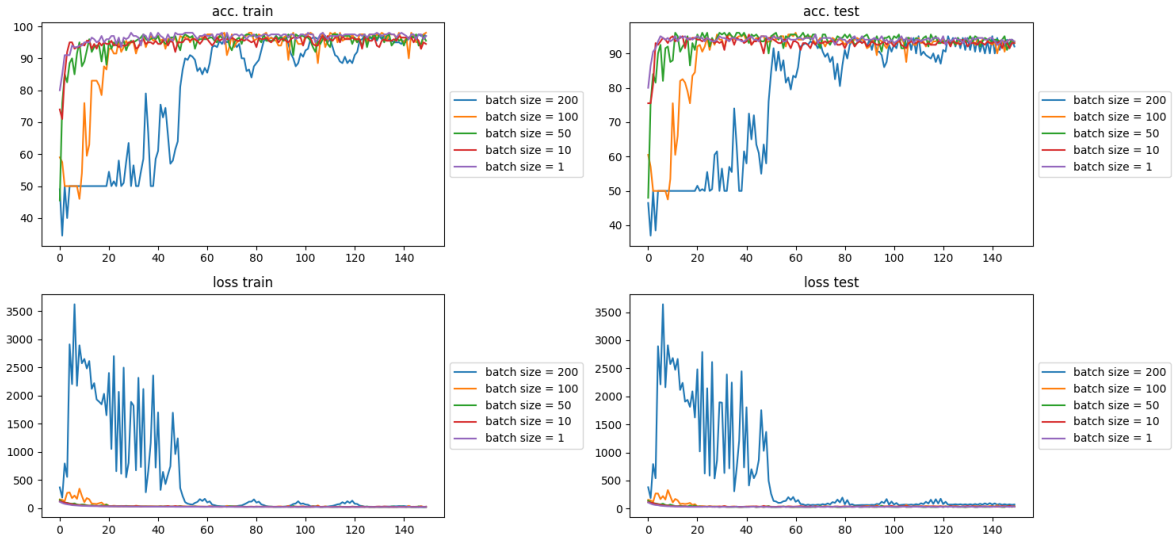


Figure 11: Autograd : Accuracy and Loss / Iterations by Batch Size

Experiment on autograd: Influence of learning rate In the figure below, we see the results from our experiment on different learning rates. As a base, we fixed other hyperparameters to $batch\_size = 10$ and $nb_{iter} = 150$. Once again, the autograd outperformed our version from scratch. Even if the learning rates were unsuitable to the batch size, they converged. But still, it was time consuming and took longer than more suitable ηs like $\eta = 0.03$.

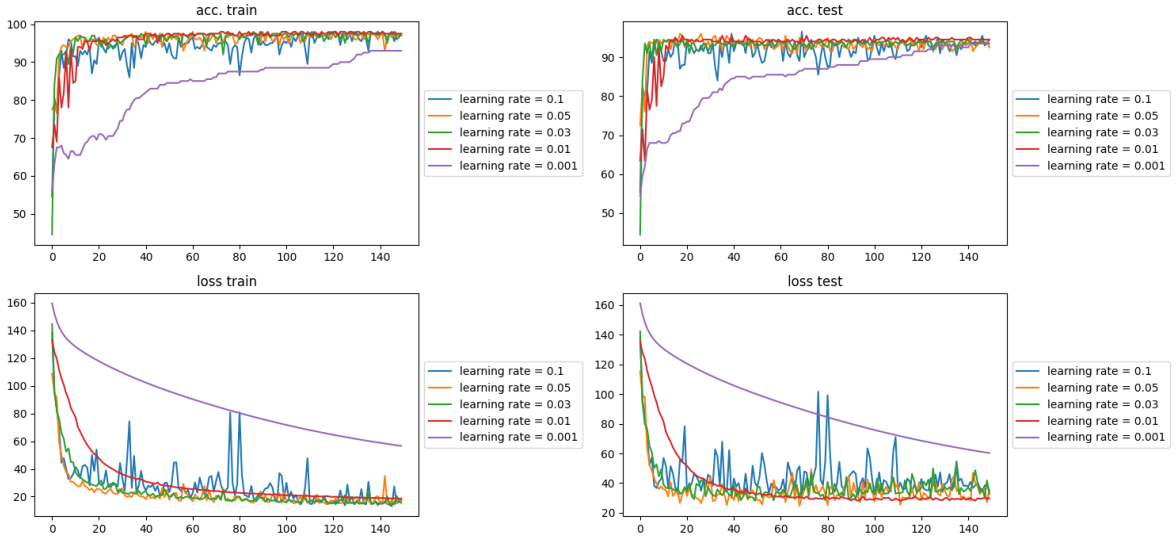


Figure 12: Autograd : Accuracy and Loss / Iterations by Learning Rate

2.3.3 Part 3 : Simplification of the forward pass with *torch.nn*

Instead of calculating the transformations by hand, we can use the modules from *torch.nn*.

Redo : Initial experiment Let's run a single test first to see if our model works with ; learning rate $\eta = 0.03$, $nb_{iter} = 150$, $batch_{size} = 10$.

Below, we have the accuracy and the loss graphs for train and test. If we compare it to our initial model in Figure 6, we can see that the model converges slower but steadier. In our initial version from scratch, the model would find an acceptable minima in the first 20 iterations but it would keep fluctuating. Now the loss is decreasing smoothly indicating a more stable execution.

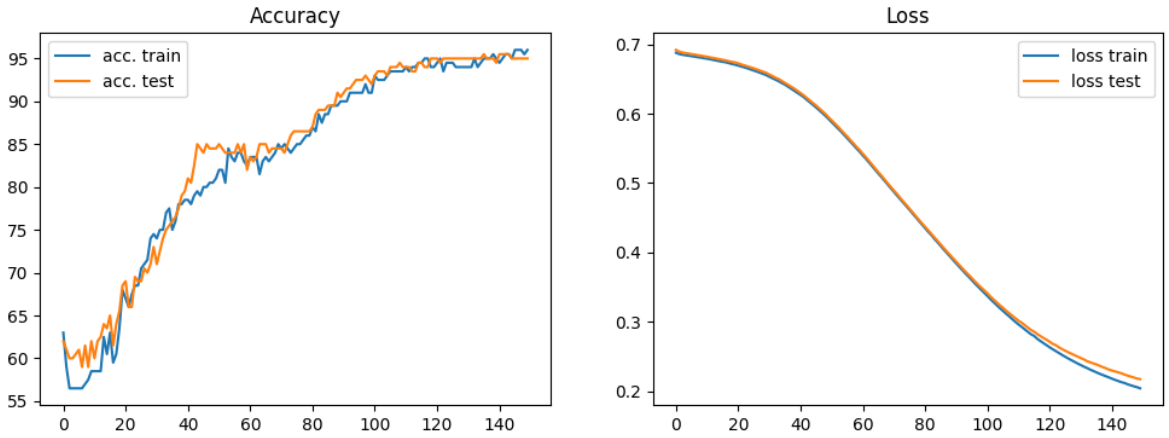


Figure 13: Autograd and torch.nn :Accuracy and Loss / Iterations

Experiment on autograd x torch.nn : Influence of batch size Once again, we fixed the other two hyperparameters to $\eta = 0.03$, $nb_{iter} = 150$.

We can see the results in Figure 14. This time, $batch\_size \in \{200, 100, 50\}$ diverged just like our initial model. $batch\_size = 10$ converged slowly but steadily giving us the most stable result and $batch\_size = 1$ converged rapidly but kept fluctuating until the iterations ended.

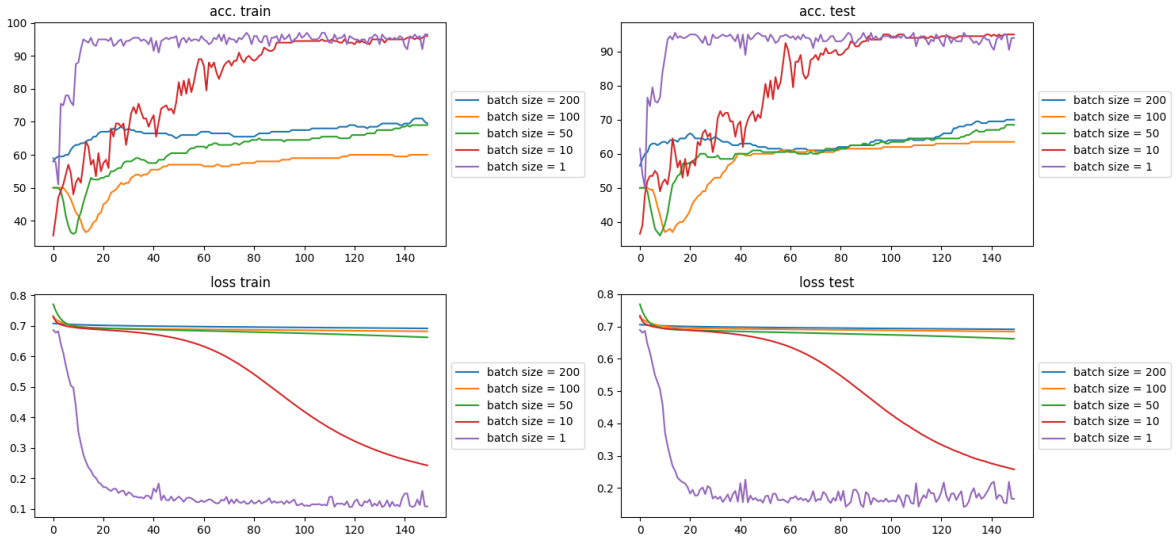


Figure 14: Autograd x torch.nn : Accuracy and Loss / Iterations by Batch Size

Experiment on autograd x torch.nn : Influence of learning rate We fixed the other two hyperparameters to $batch\_size = 10$, $nb_{iter} = 150$.

In Figure 15, we see the difference in performance for $\eta \in [0.1, 0.05, 0.03, 0.01, 0.001]$. $\eta = 0.001$ diverged once again because the number of iterations was not sufficient. Interestingly, $\eta = 0.01$, though close to 0.03, could not converge this time but $\eta = 0.1$ and $\eta = 0.05$ converged without any problem.

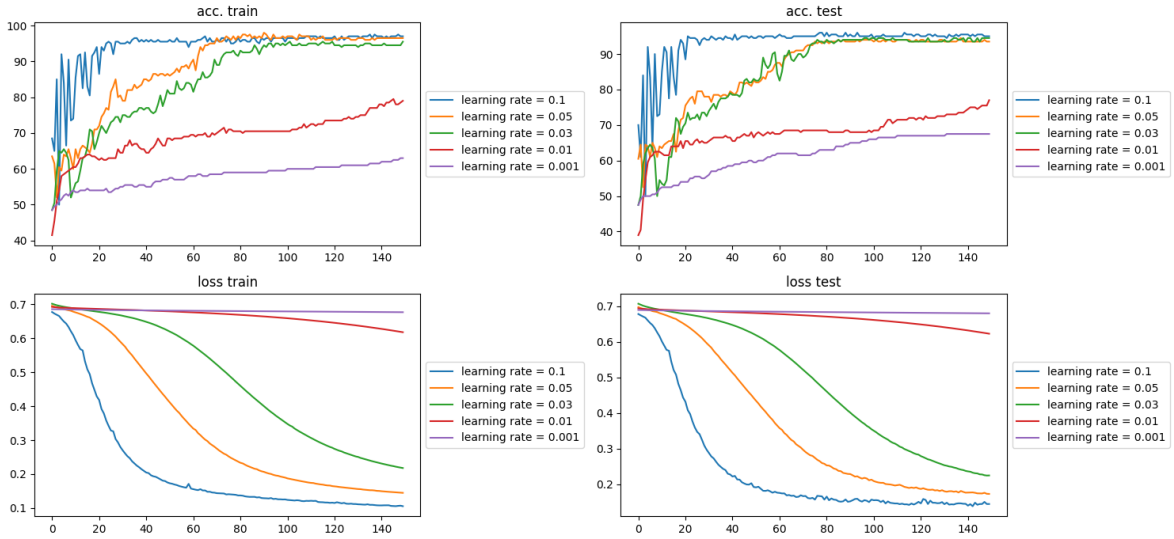


Figure 15: Autograd x torch.nn : Accuracy and Loss / Iterations by Learning Rate

These experiments showed us the difference between doing the calculations simply by hand and using already optimized modules.

2.3.4 Part 4 : Simplification of the SGD with *torch.optim*

By adding an optimizer, we created our global algorithm. When we redid the initial experimentation with fixed hyperparameters, we obtained a similar result to before. Let's directly experiment on batch size and learning rate to see how optimization influences the result.

Experiment on autograd x torch.nn x optim: Influence of batch size Finally, after adding the optimizer, let's redo our experiment on the batch size where $\eta = 0.03$, $nb_{iter} = 150$.

This time, as shown in Figure 16 all the runs converged with similar performances. Having large batches did not boost the learning rate because torch.optim.SGD handles applies the learning rate uniformly and the gradients scale correctly. So, even if batch sizes vary, the learning rate is applied consistently reducing the risks of overshooting and oscillation.

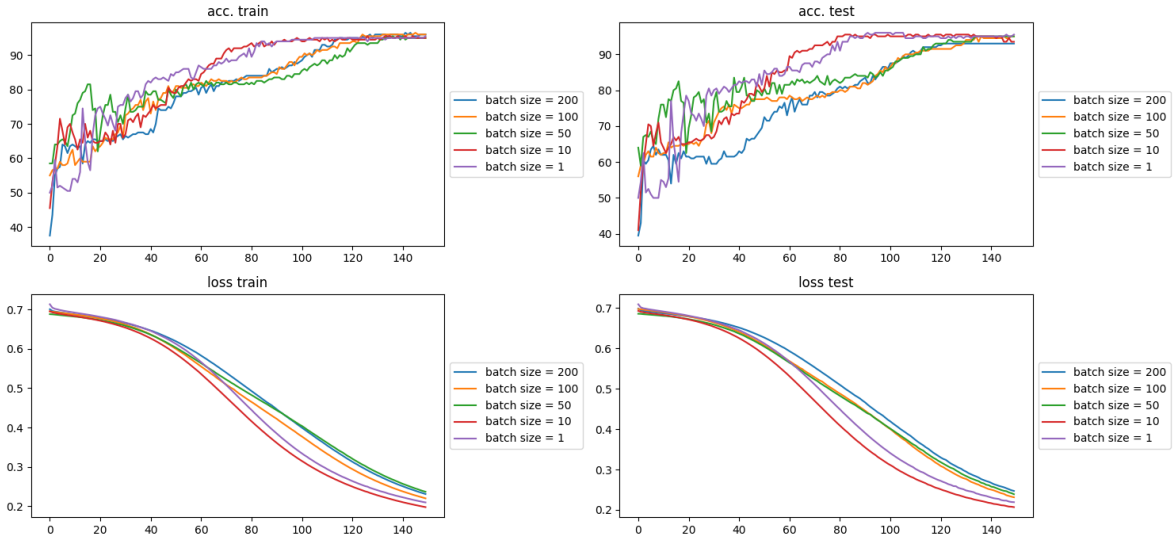


Figure 16: Autograd x torch.nn x optim : Accuracy and Loss / Iterations by Batch Size

Experiment on autograd x torch.nn x optim: Influence of learning rate Now we know that torch.optim.SGD stabilizes the learning rate based on batch size, let's test different learning rates and see if they are stable by fixing $batch\_size = 10$ and $nb_{iter} = 150$.

In Figure 17, we see that even if the optimizer applies the learning rate steadily, it is still important to tune it correcly. $\eta \in [1, 0.05, 0.03]$ eventually converged however, $\eta \in [0.01, 0.001]$ were too small to converge in 150 iterations.

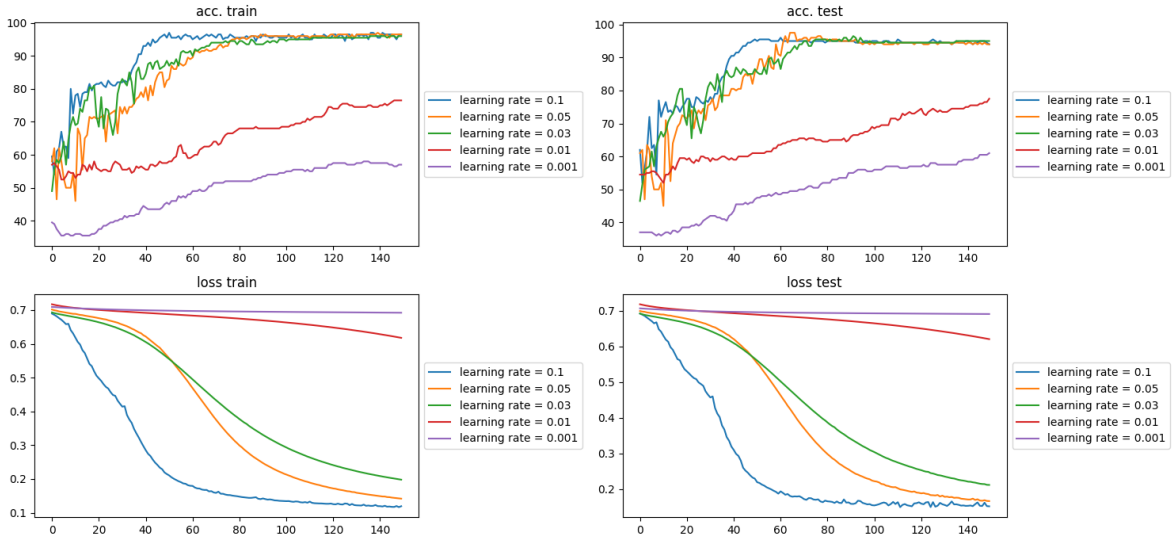


Figure 17: Autograd x torch.nn : Accuracy and Loss / Iterations by Learning Rate

2.3.5 Part 5 : MNIST

In the first 4 parts, we used a simple dataset with 2 features defining each example. This time, we will apply our final model on the MNIST data where images of handwritten numbers are classified.

Setting the training hyperparameters : $\eta = 0.03$, $batch\_size = 100$, $nb_{iter} = 150$.

This time, we have more classes and more data, so the loss graph in Figure 18 is not looking as stable but there is still a gradual decrease. After training, we chose 5 random examples to test their prediction as seen in Figure **??**

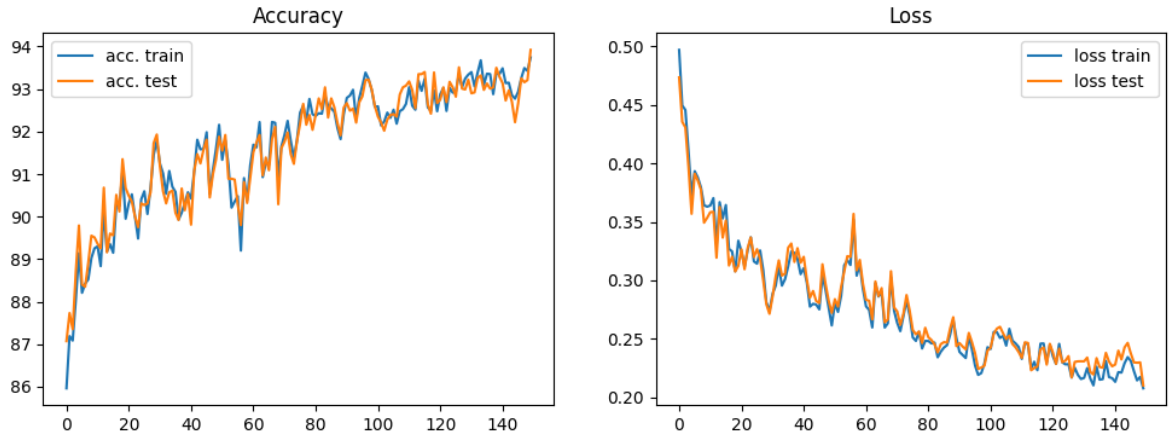


Figure 18: MNIST: Accuracy and Loss / Iterations

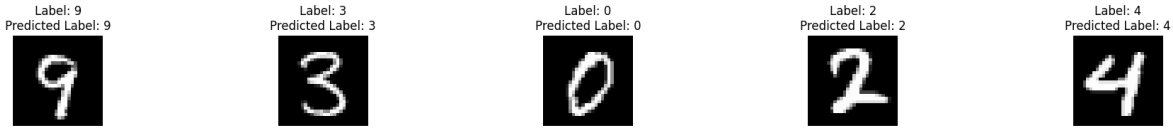


Figure 19: MNIST: True Label / Prediction / Image

2.3.6 Part 6 : SVM

Trying a linear SVM on CirclesData In the figure below, we see how a linear SVM classifies a non linear data. It simply can not. Because Linear SVM tries to divide the data into its classes using a line and optimizing the margin. However, in this case, there is no line separating the classes as one of them is surrounding the other one. So, it works as well as choosing a random class.

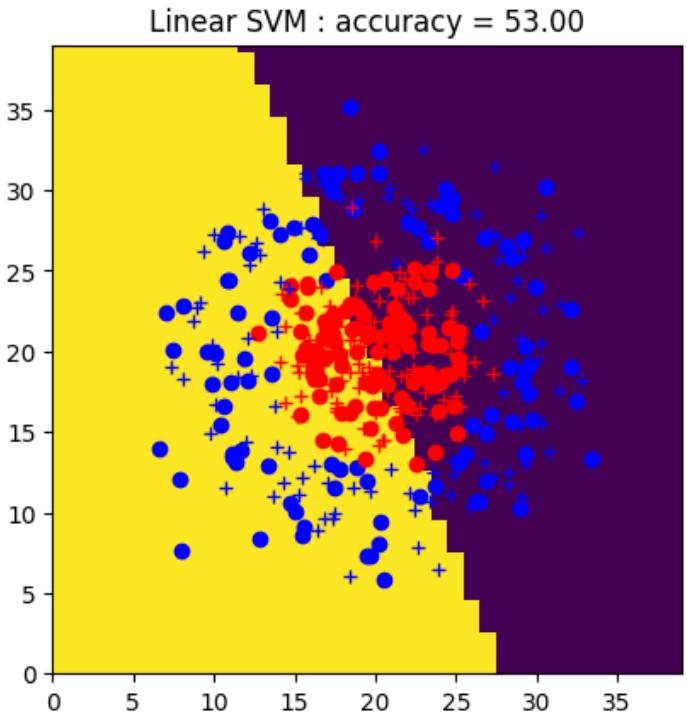


Figure 20: Linear SVM on Circular Data

Trying a more complex kernels on CirclesData Polynomial kernel does not work with the data because a circular boundary can't be represented with a polynome. Even though sigmoid kernel is also non linear, it is not suitable for circular data because it tends to create threshold-based boundaries.

RBF on the other hand can handle non-linearly separable data because it maps the data to higher dimensions and creates boundaries based on distances between examples.

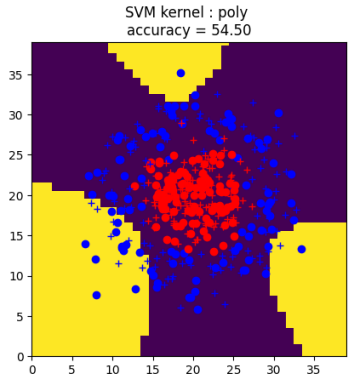


Figure 21: Polynomial SVM on Circular Data

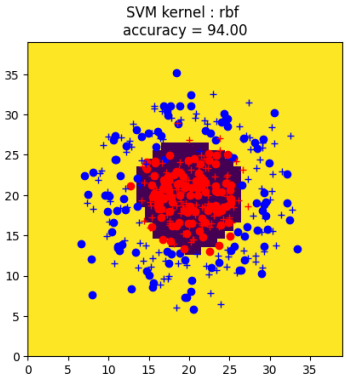


Figure 22: Radial Basis Function SVM on Circular Data

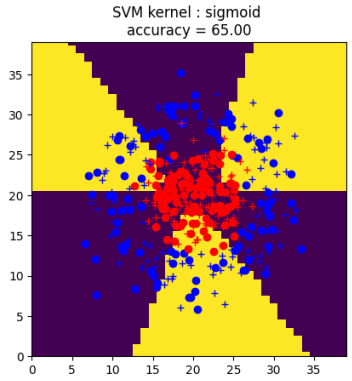


Figure 23: Sigmoid SVM on Circular Data

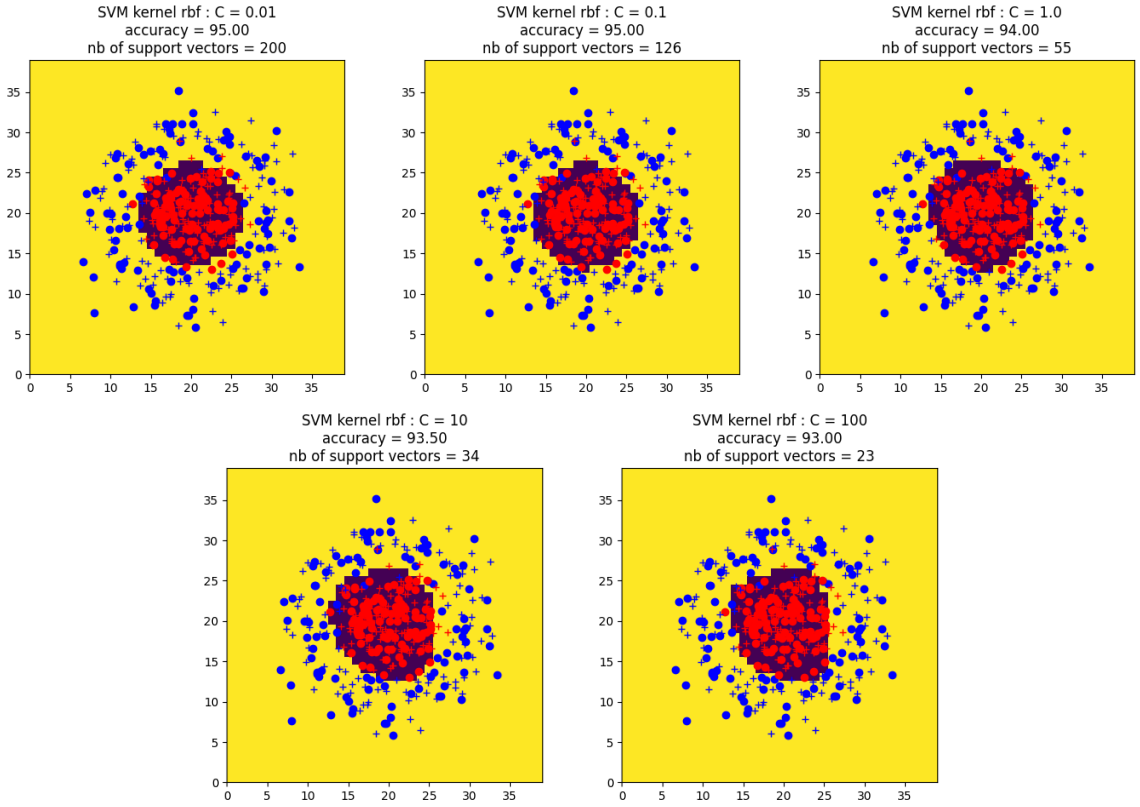


Figure 24: RBF SVM : Variance of C

Impact of parameter C In Figure 24, it is hard to differentiate the decision boundaries because the data is not complex enough but we can see how accuracy slightly changes and how the number of support vectors decreases as C increases. Because when C is smaller, SVM allows more misclassification during the training and the decision boundary is softer with more vectors. When C is too big, the margin is smaller and the model fits the training data tightly with fewer support vectors.

2.4 Conclusion

In the theoretical foundation, we calculated all the gradients by hand and explained the basic architecture of our model. This facilitated the practical part as we already knew what to do and what to expect.

By coding the neural network, we better grasped how this architecture and these calculations work and by analyzing the influence of key hyperparameters like batch size and learning rate, we learned the importance of tuning a model during training.

Testing the model on different datasets showed us how versatile and adaptable Neural Networks can be but also to always keep in mind the dimensions and the influence of hyperparameters.

We observed how *batch size* and *learning rate* parameters influence each other and without an optimizer in hand, they can make the model diverge. We always kept *number of iterations* fixed and at a high value to see if the model converges, how fast.

3 1-cd: Convolutional Neural Networks

3.1 Introduction

We look into the fundamental principles behind CNNs, including their architecture, training methodologies, and optimization techniques, while addressing the challenges encountered during the learning process.

3.2 Section 1 - Introduction to convolutional networks

In this part, we will respond to the questions.

Q 1.4.1 Considering a single convolution filter with padding p, stride s, and kernel size k, for an input of size $x \times y \times z$:

What will be the output size? The formula for the output size of a convolutional layer is:

$$\text{output\_size} = \frac{\text{input\_size} - k + 2p}{s} + 1$$

Applying this formula to the input size $x \times y \times z$:

$$\text{output\_size} = \left(\frac{x - k + 2p}{s} + 1\right) \times \left(\frac{y - k + 2p}{s} + 1\right) \times z_{\text{output}}$$

where z_{output} depends on the number of filters applied.

We can calculate each dimension of the output size separately:

$$x_{\text{output}} = \frac{x - k + 2p}{s} + 1$$

$$y_{\text{output}} = \frac{y - k + 2p}{s} + 1$$

How much weight is there to learn? Given our convolutional filter of size $k \times k \times z$ as the kernel needs to have the same depth as the input, we need a weight for each element. Therefore :

$$nb_{\text{weight\_per\_filter}} = k \times k \times z$$

If we count the bias as a weight, we will have :

$$nb_{\text{weight\_per\_filter}} = (k \times k \times z) + 1$$

If we have c filters, the total number of weights to learn would be:

$$nb_{\text{weight}} = \text{¸} \times (k \times k \times z) + 1)$$

⋆ **Q 1.4.2** Convolutional layers are better adapted for images than fully-connected layers. First of all, they are more efficient in terms of number of parameters since images are usually high dimensional. Convolutional layers learn less parameters, because they learn small convolutional filters as parameters whereas a fully-connected layer would learn parameters that are the entire size of the image. So in this case, fully-connected can be prone to overfitting due to the high number of weights.

Also, convolutional layers have local connectivity, not all neurons are connected to all the other neurons, but only to a local region. This allows them to capture local spatial patterns.

⋆ **Q 1.4.3** Pooling allows dimensionality reduction, hence also overcoming overfitting. Typically, max pooling or average pooling operations are used, with a stride of 2, allowing to downsample.

They also allow for translation invariance, making the feature maps more robust to changes in the position of the image. [6]

⋆ **Q 1.4.4** We can use parts of the network on this image. The convolutional layers can take images of any size since they're applying filters to small regions of the images. Same way, the pooling layers can be applied with no problem.

The fully-connected layers however, may not work if the dimensions of the input they receive don't correspond to the input dimension they expect. In this case, we can add more pooling layers or increase the stride of the pooling layers in order to downsample the images.

Q 1.4.5 A fully-connected layer can be seen as 1x1 convolutions. [12] Each neuron in the fully-connected layer can be interpreted as 1x1 convolutional filters.

Q 1.4.6 By replacing the fully-connected layers with 1x1 convolutions, the network can process images of larger sizes without modification.

Q 1.4.7 For the first layer the receptive field corresponds to the filter size. So if the filter size is k, then the receptive field of the first layer is also k.
In general, for each convolutional layer l, the receptive field size R_l can be calculated as:

$R_l = R_{l-1} + (k_l - 1) \times \prod_{j=1}^{l-1} s_j$ where k_l is the kernel size of l and s_j is the stride of layer j. [13] If we have a network with:
Layer 1: $k_1 = 3$ and $s_1 = 1$
Layer 2: $k_2 = 3$ and $s_2 = 2$
Then $R_1 = 3$ and $R_2 = 3 + (3 - 1) \times 1 = 5$

For the deep layers, the receptive field grows as we go deeper, each neuron "sees" a larger area of the original input image.
As we progress deeper into the network, increasingly complex features are extracted, and the final layers specialize in detecting higher-level features. [4]

3.3 Section 2 - Training from scratch of the model

3.3.1 Q 2.1 Network architecture

Q8 To keep the dimension size, the amount of padding to use can be calculated as $p = \frac{k-1}{2}$ where k is the filter size, and use a stride of 1.

Q9 No padding will be used in this case. We can use a stride value of 2.

⋆ **Q10** considering the cifar 10 images, the input images are of size 32x32x3.

conv1 : 32 convolutions 5×5 stride=1 padding=2
output height $= \frac{32-5+4}{1} + 1 = 32$ output width $= \frac{32-5+4}{1} + 1 = 32$
output size $= 32 \times 32 \times 32$
number of weights $= ((5 \times 5 \times 3) + 1) \times 32 = 2432$

pool1 : max-pooling 2×2 stride=2 padding=0
no parameters to learn
output height = output width = $\frac{32-2+0}{2} + 1 = 16$
output size = $16 \times 16 \times 32$

conv2 : 64 convolutions 5×5, stride=1, padding=2
output height = output width = $\frac{16-5+4}{1} + 1 = 16$
output size = $16 \times 16 \times 64$
number of weights = $((5 \times 5 \times 32) + 1) \times 64 = 51264$

pool2 : max-pooling 2×2, stride=2, padding=0
no parameters to learn
output height = output width = $\frac{16-2+0}{2} + 1 = 8$
output size = $8 \times 8 \times 64$

conv3 : 64 convolutions 5×5, stride=1, padding=2
output height = output width = $\frac{8-5+4}{1} + 1 = 8$
output size = $8 \times 8 \times 64$
number of weights = $((5 \times 5 \times 64) + 1) \times 64 = 102464$

pool3 : max-pooling 2×2, stride=2, padding=0
no parameters to learn
output height = output width = $\frac{8-2+0}{2} + 1 = 4$
output size = $4 \times 4 \times 64$

fc4 :
output size = 1000 and number of weights = $(64 \times 4 \times 4 + 1) \times 1000 = 1025000$

fc5 :
output size = 10 and number of weights = $(1000 + 1) \times 10 = 10010$

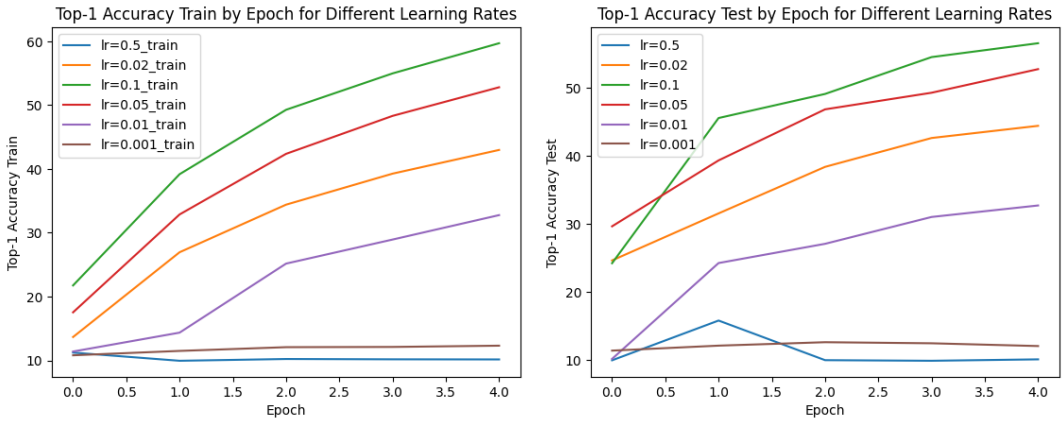
Q11 In total there are 1 191 170 weigths to learn. The cifar 10 dataset has 60000 examples so there's a lot more weights than examples. There's a risk for overfitting.

Q12 The number of parameters with the BoW and SVM approach depends on the size of the vocabulary chosen. If V is the vocabulary size, then we will have (V+1)x10 parameters (+1 is for bias). The number of parameters in the CNN is significantly higher than that of the BoW and SVM approach. A CNN will typically have over a million parameters whereas the BoW and SVM approach can have around thousands.

3.3.2 2.2 Network learning

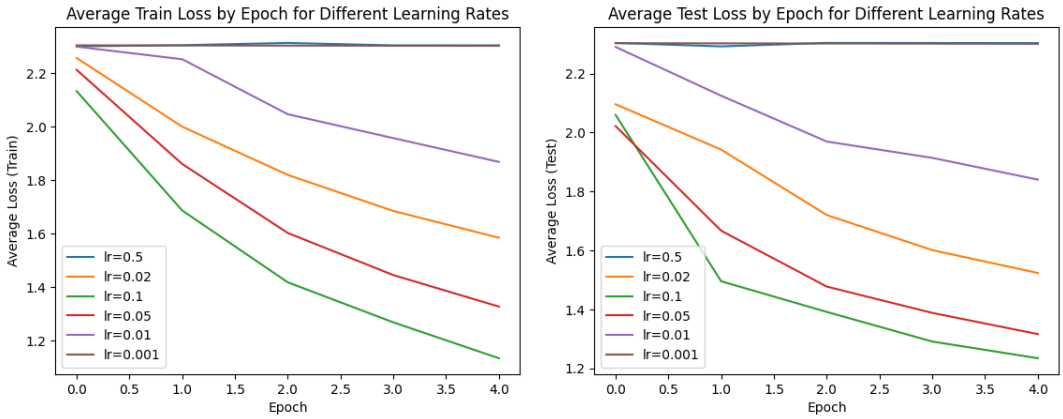
⋆ **Q14** During training we provide optimizer, we do the backpropagation of the loss and update parameters according to that. During test we don't do these, the loss isn't used to update the parameters.

⋆ **Q16** To see the influence of the learning rate, we test different learning rates with a batch size of 128 on 5 epochs.



(a) Learning rates training accuracies (b) Learning rates test accuracies

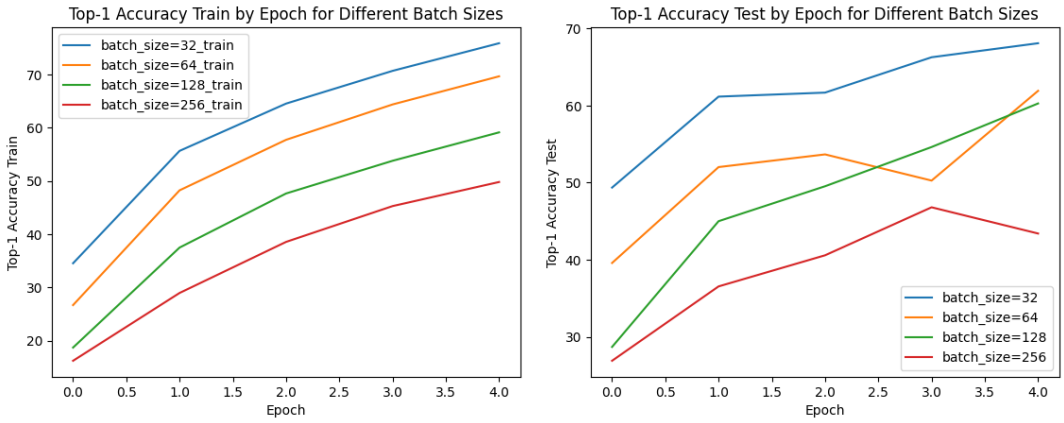
Figure 25: Effects of learning rates on accuracies



(a) Learning rates training losses (b) Learning rates test losses

Figure 26: Effects of learning rates on the loss

In figure 25, the highest learning and the lowest rates we tried (0.5 and 0.001) yielded the worst accuracies, which illustrates the importance of selecting an optimal learning rate. A learning rate that is too high can cause learning sub-optimal weights too fast and a learning rate that's too low can cause slow convergence and the model can get stuck in local minima.
The learning rate 0.1 got the highest accuracies, suggesting that it provides a balance between stability and effective gradient updates.
Figure 26 illustrates that the loss behaves consistently with these findings.



(a) Batch size training accuracies (b) Batch size test accuracies

Figure 27: Effects of batch size on accuracies

To see the influence of the batch size, we test different batch sizes with the best learning rate found earlier (0.1) on 5 epochs.
Figure 27 illustrates that a batch size of 32 performs the best with the learning rate 0.1, and actually we observe that as we increase batch size, the performance decreases. This may be because our epoch number is insufficient for convergence, particularly for larger batch sizes.

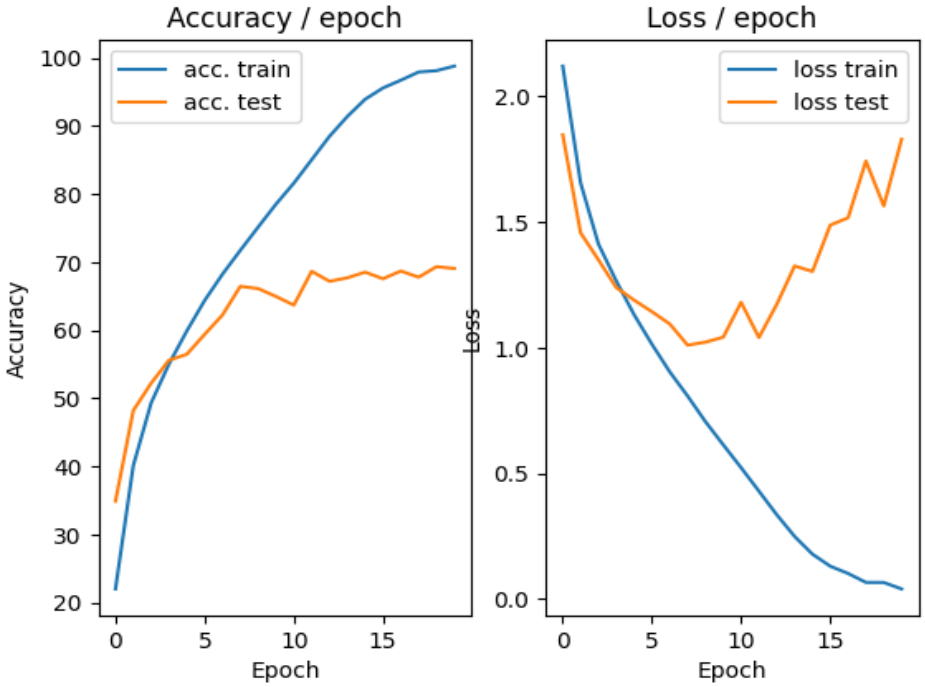


Figure 28: Observation of the train and test performances with lr=0.1 and batch size=128

Q17 In figure 28, when we compare the train and test performance of our model, we observe that test accuracy is higher at the first epoch than the training accuracy and that the test loss is lower than training loss. This suggests that the model may be capturing patterns in the test data more effectively than in the training data at this early stage.

⋆ **Q18** When we observe the curves in figure 28, we see that there's a big gap between training and test curves in both accuracy and loss. The training accuracy is a lot higher and the training loss is much smaller. This shows that we're overfitting the training data, failing to generalize to unseen data.

3.4 Section 3 – Results improvements

3.4.1 3.1 Standardization of examples

Q20 The mean image should only be computed on the training images because this would cause information leakage, we shouldn't have any statistical information about the test data. Computing the mean image on the entire dataset and then splitting into train and test sets would also give us a false optimistic impression of how the model is performing since the model would have seen information about the test data.

Q21 We tested different normalization techniques such as:

- normalization using the mean and the standard deviation as described in the TME instructions
- L1 normalization
- Min max normalization
- PCA whitening
- ZCA whitening (this was implemented but couldn't be tested since the computational requirement was too much)

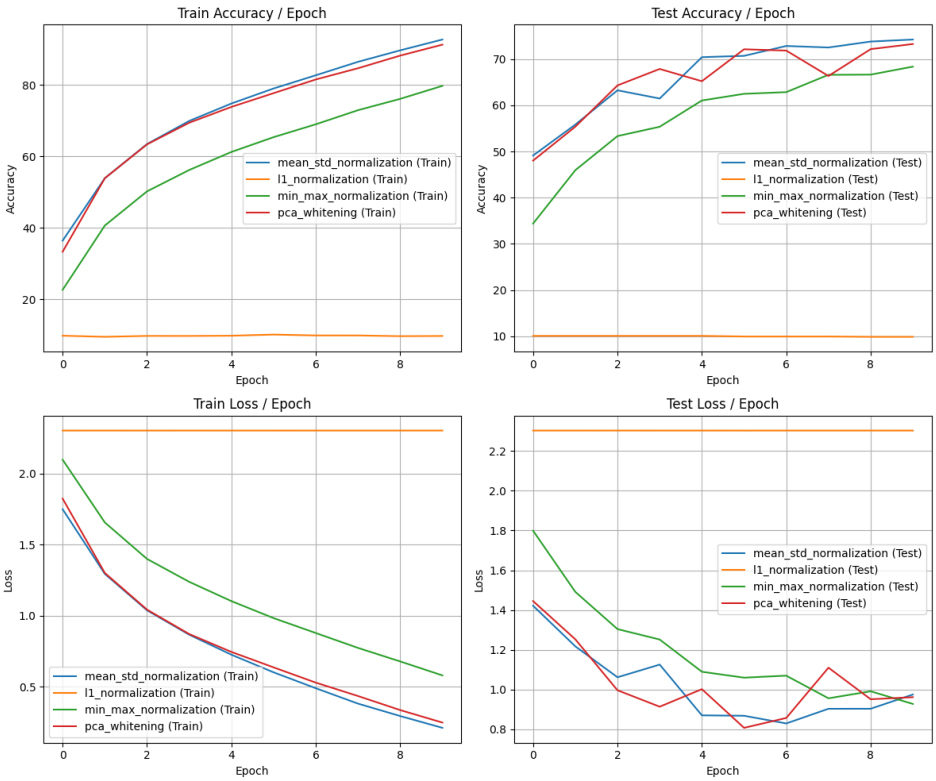


Figure 29: Performance of different normalization techniques

In figure 29, it is evident that L1 normalization had the worst performance overall.
The suggested mean-std normalization is showing consistent improvement throughout training, with improving the training accuracy and reducing the training loss at each epoch, achieving a training loss under 0.5. During testing, the accuracy continued to improve, although not as consistently as in training. Nonetheless, it reached the highest test accuracy, followed closely by PCA whitening.
PCA whitening normalization produced results that were quite similar to those of mean-std normalization during training. During test, while its accuracy occasionally exceeded that of mean-std normalization, it ultimately finished slightly below it. It's test loss decreases during test but oscillates and has a jump towards the end.
The mean-std normalization has the best performance overall, followed closely by PCA whitening. Min-max normalization ranked in the middle, while L1 normalization demonstrated the least performance.

3.4.2 3.2 Increase in the number of training examples by data increase

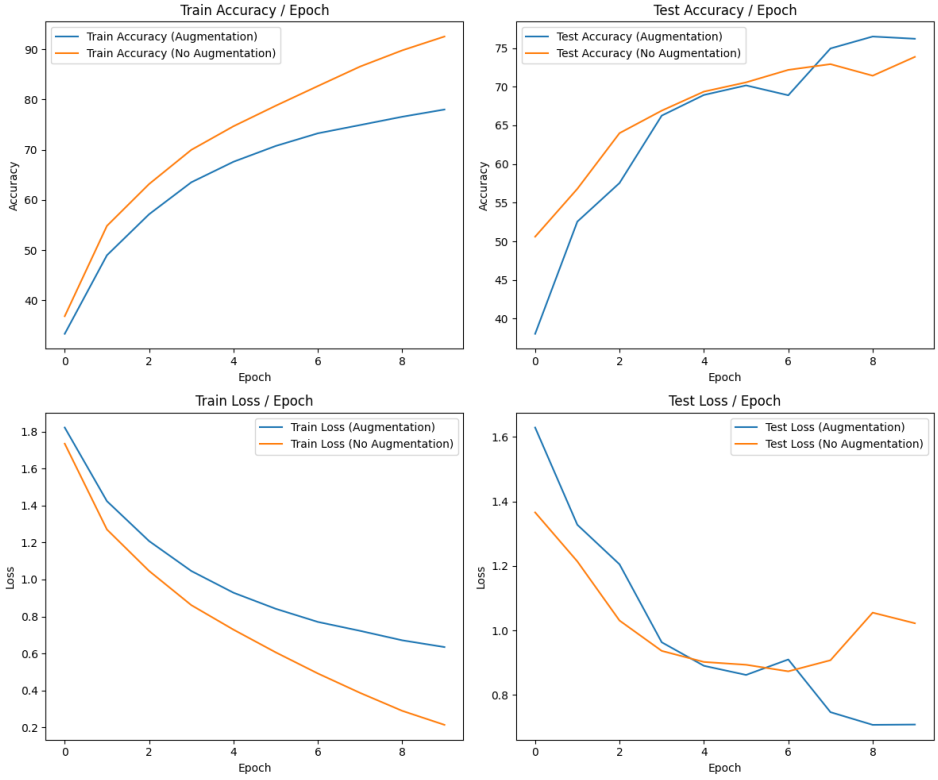


Figure 30: Performance of data augmentation vs. no augmentation

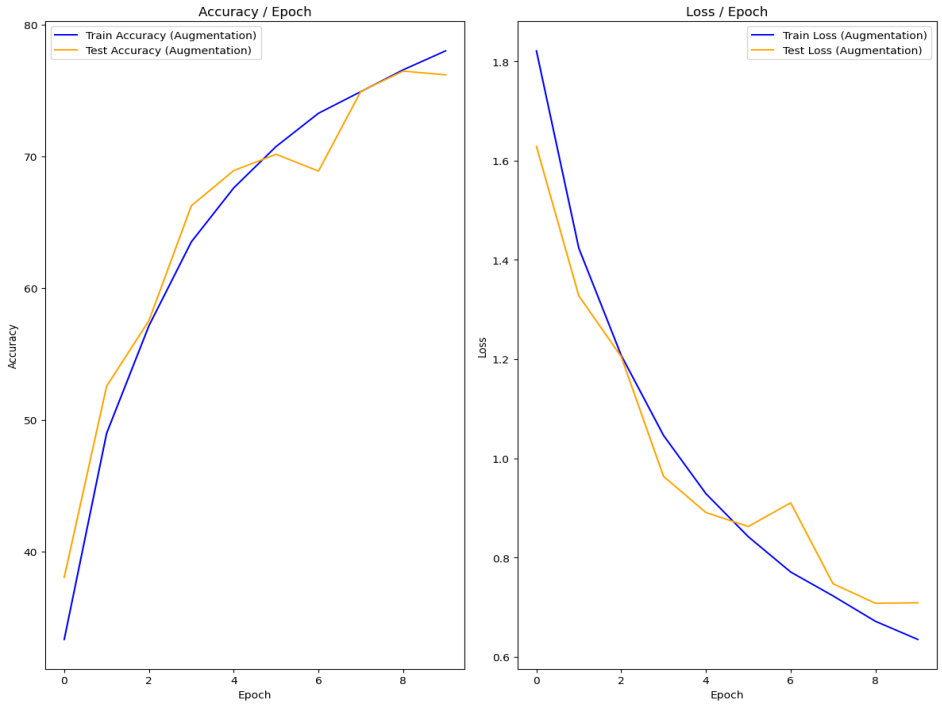


Figure 31: Train/test accuracy and loss of data augmentation

Q22 When comparing data augmentation to no augmentation, as shown in Figure 30, we observe that during training, the model without augmentation performs better, exhibiting lower training loss and higher training accuracy throughout the training process. However, during testing, data augmentation demonstrates its advantages: in the later epochs, it achieves a significantly lower error and higher test accuracy.
In figure 31, we observe the performance of the data augmentation. The test accuracy initially exceeds the training accuracy until around epoch 5, after which it falls below the training accuracy. By the end of the training, there is a slight gap between the two.
Similarly, the test loss is lower than the training loss during the initial epochs, remaining so until epoch 5, after which it slightly surpasses the training loss. This suggests that overfitting may be occurring around epoch 5.

Q23 Horizontal symmetry seems suitable for types of images such as cars, animals where this modification doesn't change what the object is, therefore we're also improving the robustness of the model to different orientations while recognizing an object. But there are cases where this method may not be suitable. For example, in digit or character recognition, flipping these may lead to incorrect results, as they may become unrecognizable or incorrect.

Q24 Data augmentation can increase the computational cost of training. In cases where data is already very diverse, it may not be very effective. [9]

Q25 We try different data augmentation methods:

- Random crop + horizontal flip (as tried above)
- Random rotation of 45 degrees
- Gray scale conversion

- Color jitter
- Gaussian blur

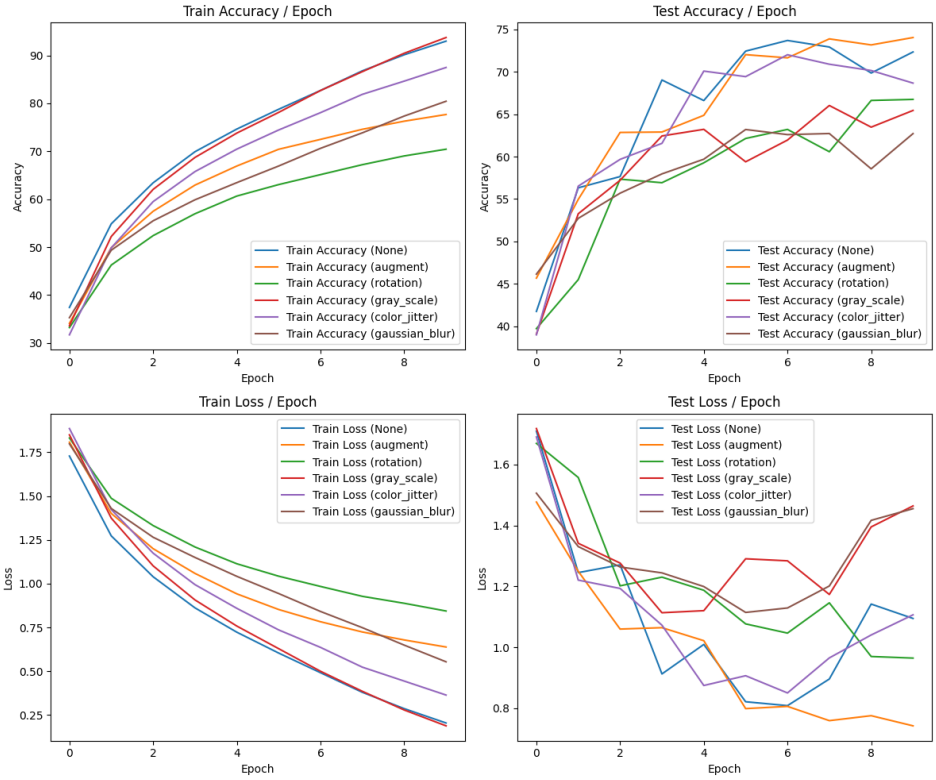
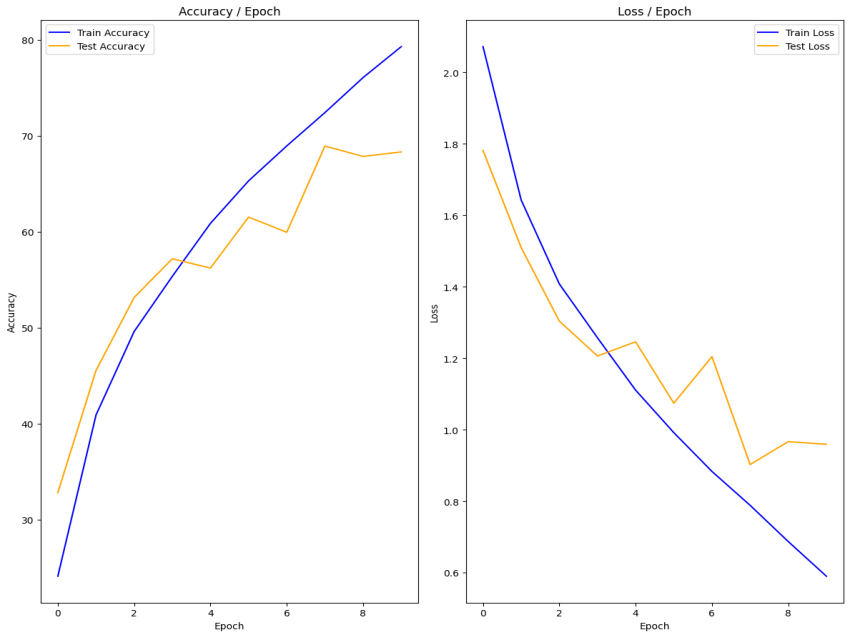


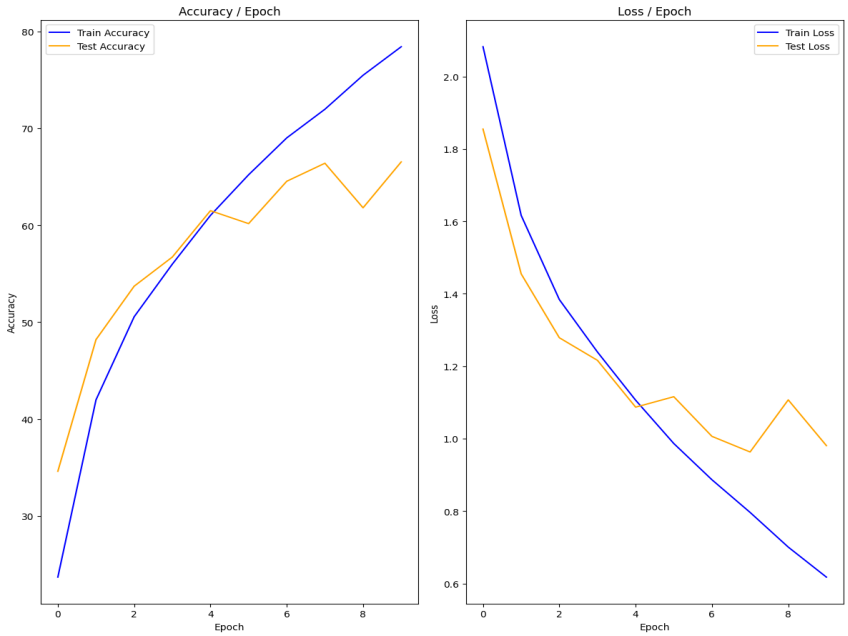
Figure 32: Performance of different data augmentation methods and no augmentation

In figure 32, during training no augmentation and gray scale conversion had the best performances, followed by color jitter, and random rotation had the worst performance. The curves during training are steady whereas during test, the curves oscillate a lot.
During testing, the previously suggested method (labeled as augment in the graphs), achieved the highest accuracy and also achieved a significanly lower test loss than the other methods. Gray scale and gaussian blur had the worst test performances even though gray scale had good results during training. Overall, the suggested augmentation method seems to be the most performant.

3.4.3 3.3 Variants on the optimization algorithm



(a) SGD without learning rate scheduler



(b) SGD with learning rate scheduler

Figure 33: Comparing Effects of the lr scheduler for SGD

Q26 In figure 33, we see that adding the learning rate scheduler results in less fluctuating test curves, we have more stability using the scheduler.

Q27 Decaying learning rates increases the likelihood of converging to a better minimum as we take smaller steps when we're approaching an optimum.

Q28 We evaluate several SGD variants as well as a range of learning rate scheduling techniques, each with distinct mechanisms for adjusting the learning rate over time.
While still using the exponential learning rate scheduler, we test optimization methods including standard SGD, ASGD (Averaged Stochastic Gradient Descent), Adadelta, Adagrad, Adam, AdamW, Adamax, RMSprop, and Rprop.

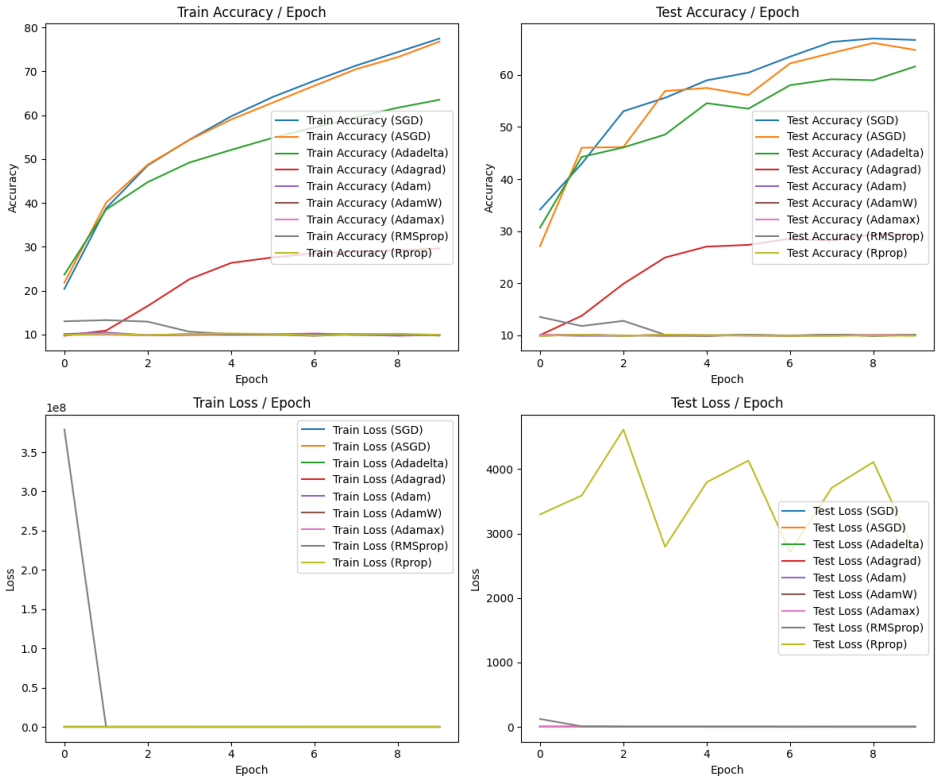


Figure 34: Performance of different optimization methods

In figure 34, we observe that during training SGD reached the highest accuracy, followed closely by ASGD. And RProp, RMSprop and Adamax had the worst train accuracies. During test, SGD and ASGD again had the highest accuracies, followed by Adadelta. Rprop also had a significantly higher test loss compared to the other methods.

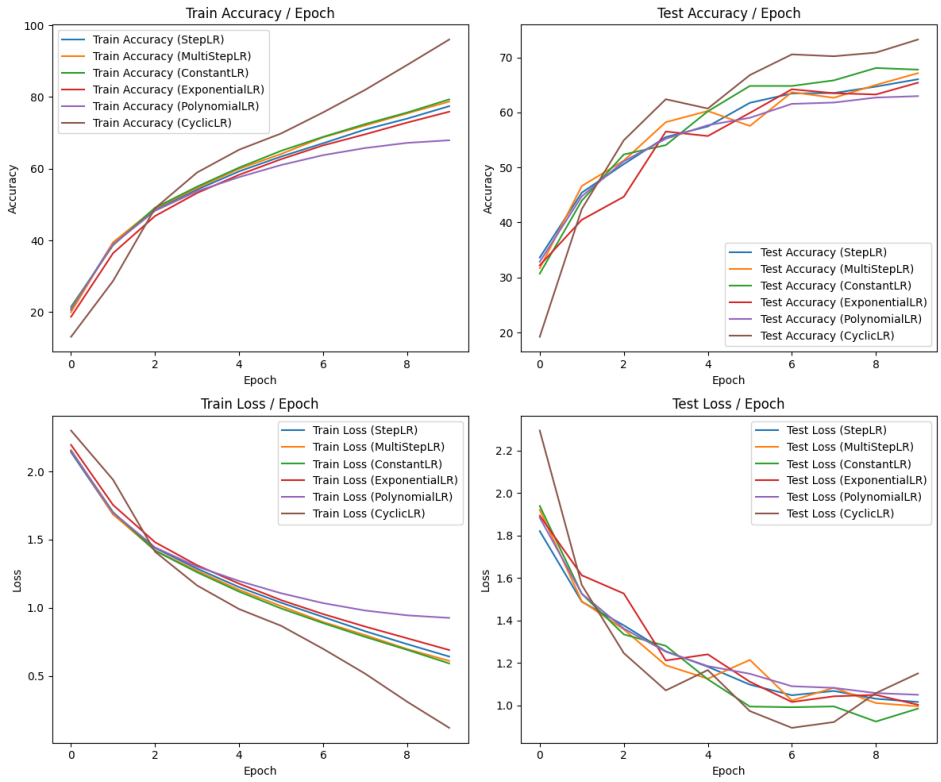


Figure 35: Performance of different learning rate schedulers

In figure 35, we use SGD and try different learning rate scheduling strategies such as StepLR, MultiStepLR, ConstantLR, ExponentialLR, PolynomialLR, and CyclicLR to assess how different scheduling methods affect the model's learning.
CyclicLR demonstrated significantly better performance during training compared to the other methods. It also achieved the highest accuracy on the test set; however, it experienced a spike in test loss toward the final epochs, ultimately resulting in the highest test loss at the end. This suggests that it may have led to overfitting.

PolynomialLR generally exhibited the worst performance, except in terms of test loss, where it ranked second-worst after CyclicLR.

ConstantLR had the lowest test error and the second-highest test accuracy while performing below CyclicLR during training. This method may be less prone to overfitting.

3.4.4 3.4 Regularization of the network by dropout

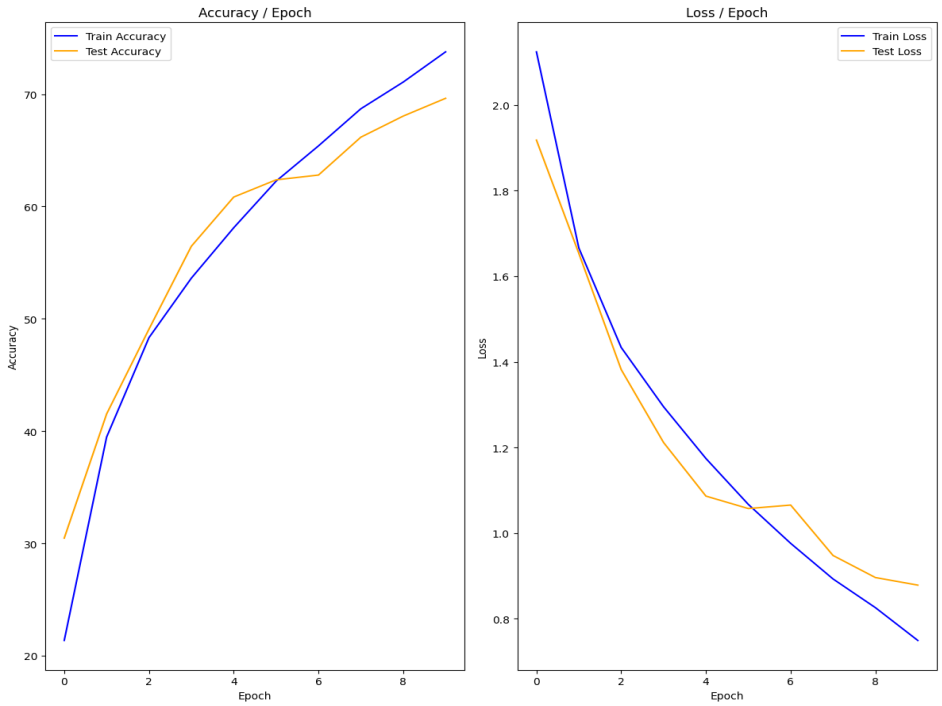


Figure 36: Performance of dropout layer with p=0.5

Q29 We compare figure 36 to figure 33a where no dropout was applied. The gaps between the train and test curves are smaller now, so we see the regularization effect of dropout. Additionally, there are fewer fluctuations in the test curves when dropout is used, suggesting improved stability in the model's performance.

Q30 Regularization is a method used to prevent overfitting. Overfitting occurs when a model works too well on training data but can't generalize to unseen test data. Regularization adds penalties to the model's loss function. One of the most common regularization techniques, L2 regularization, penalizes large weights, therefore reducing the model's complexity and making it less prone to overfit.

Q31 In dropout, during training neurons are temporarily removed from the network with a probability p of keeping the neuron. When neurons are randomly dropped, this can result in other neurons having to make predictions for the missing neurons. This can be interpreted as multiple independent representations. This way neurons would be more robust to weights and generalize better. [5]

Q32 The hyperparameter is the probability p of keeping a neuron active in the network. If p is higher, we have a higher chance of keeping more neurons. We can adjust p for different layers based on our needs. If we want to keep all the neurons of the last few layers, we can set it to 1. Similarly, for hidden layers with high dimensionality, so when there's higher chance of overfitting, we can set p to 0.5.
(here we referred to p as the proba of keeping a neuron *active*, but when we're commenting on the figure 37, we'll refer to it as the proba of *dropping* a neuron, since p means that in pytorch .

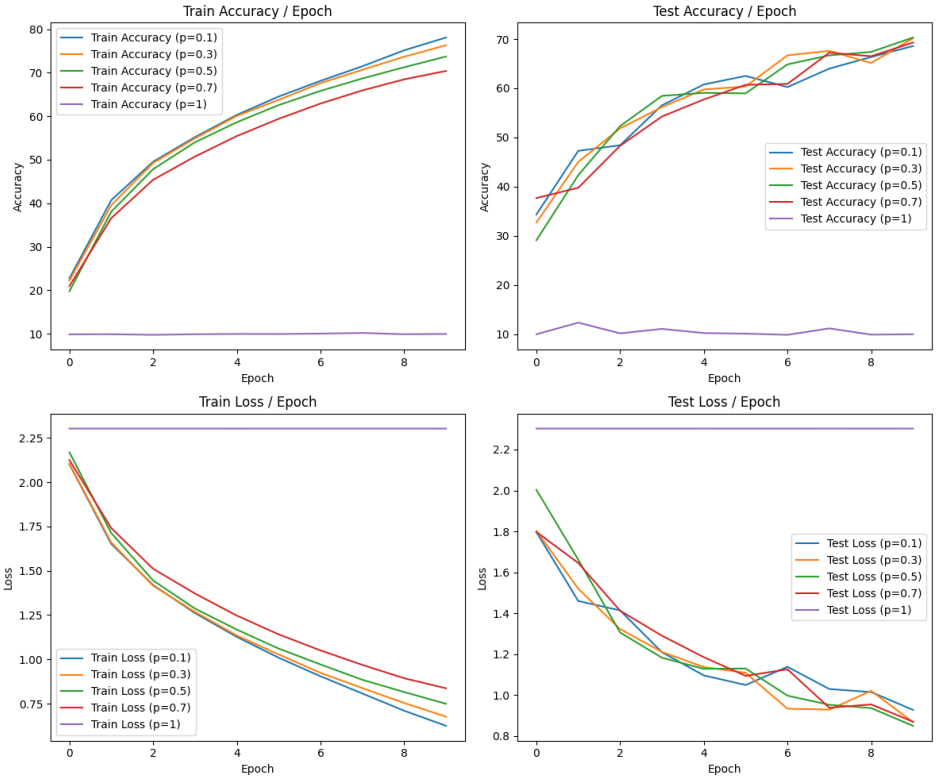


Figure 37: Different probabilities of dropping neurons

In figure 37, we observe the effects of different dropout rates. When we set p=1, so we drop all the neurons in the fully connected layer, we get the worst performance. During testing, setting p=0.5 achieves the best performances. During training, lower dropout rates such as 0.1 and 0.3 achieve better performances but 0.1 actually gets the highest test loss so it overfits.

Q33 During training, dropout is applied as described whereas during test, dropout is disabled.

3.4.5 3.5 Use of batch normalization

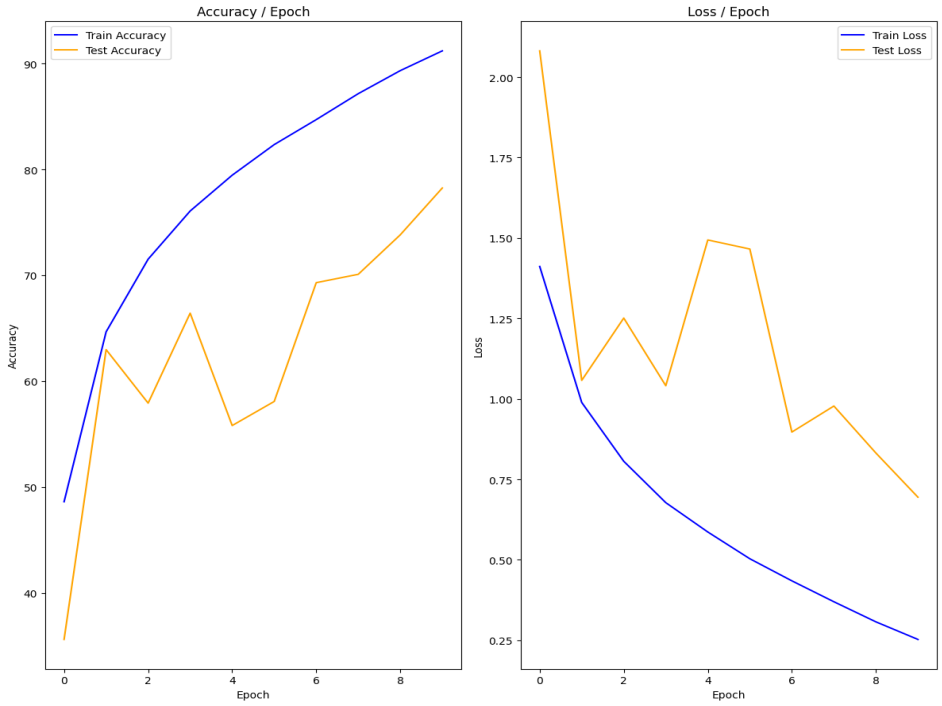


Figure 38: Batch normalization

Q34 When comparing figure 38 to figure 33a, both training and test performances have improved using batch normalization. However, the test curves display more oscillations, with noticeable jumps in test loss.

3.5 Conclusion

In conclusion, the exploration of CNNs has demonstrated their capabilities in handling image data. Careful selection of hyperparameters, implementation of effective data augmentation techniques, and the use of advanced optimization algorithms and regularization methods contribute to enhancing model performance.

4 1-e: Transformers

4.1 Introduction

The last section of our report is on transformers which were created to improve NLP's by using attention mechanisms where the model can weight the importance of different words. This concept is also adapted to be used in Computer Vision with Visual Transformers. We implemented a naive version of the first Visual Transformer, ViT.

In this part, we will once again use the MNIST dataset. In Figure 39, we see some examples of handwritten numbers in the dataset.

Figure 39: Examples from MNIST dataset

4.2 Linear Projection of flattened patches

Q3: Self-attention What is the main feature of self-attention, especially compared to its convolutional counterpart?

The main feature of self-attention is its ability to model long-range dependencies. Unlike convolutional layers that have fixed receptive fields, self-attentions's receptive field covers the entire image. [8] This allows self-attention to capture context better.

What is its main challenge in terms of computation/memory?

The main challenge of self-attention is its computational complexity. Calculating the dot product between the query and key matrices requires a complexity of $O(N^2)$, where N is the sequence length, which can be especially challenging when we have long sequences.

Write the equations:

We calculate attention using Query (Q), Key (K) and Value (V) matrices:
$Q = XW^Q$
$V = XW^V$
$K = XW^K$
where X is the input

$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
where d_k is the scaling factor. (d_k is the dimension of the Q and K matrices)

Q4 : Multi-head self-attention Write the equations:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{1}$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is the learned output projection matrix.

Q5: Transfomer block Write the equations

Considering input x:
n1 = LayerNorm(x)
att = Multi-HeadAttention(n1)
n2 = LayerNorm(x + att)
ffn = FFN(n2)
output = ffn + x + att

Q6: Full ViT model Explain what is a Class token and why we use it?

A class token is added as the first token to the input sequence. It provides an aggregate representation of the input. After processing through the Transformer layers, this representation is used to determine the classification.

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

Positional embedding represents the original positions of the patches in the input sequence. This is needed in order to learn positional-dependent information.

4.3 Experimental analysis

4.3.1 Impact of embed_dim on the performance

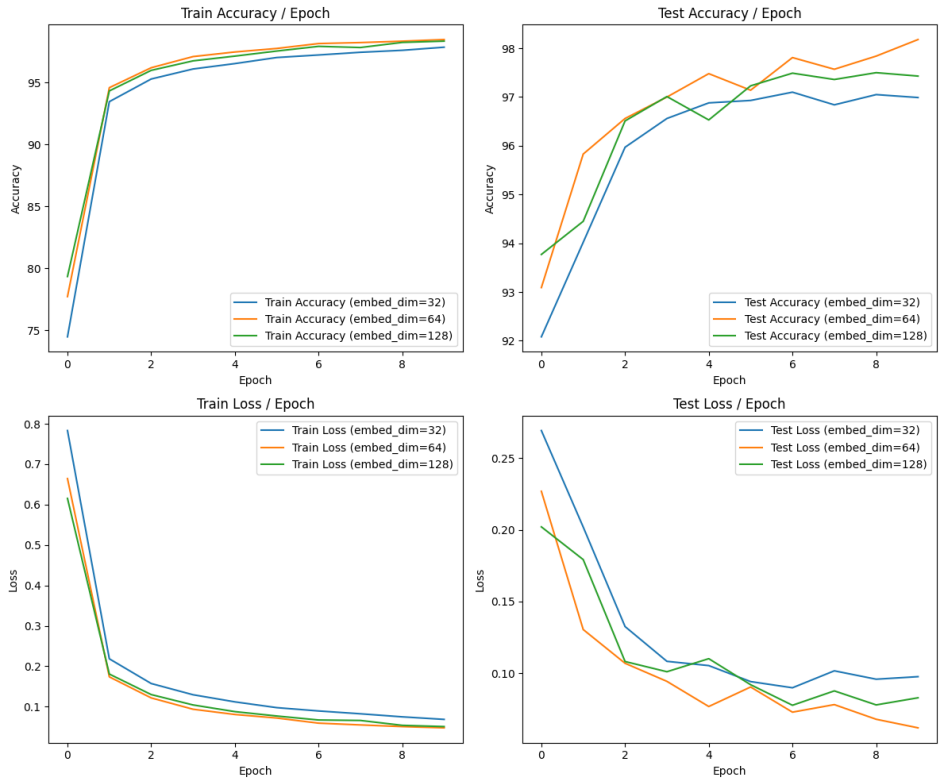


Figure 40: Effects of embed_dim

During this experiment, we fixed the patch_size to 7 and nb_blocks to 2. In figure 40, we observe that an embed_dim of 64 has the best performance. This suggests that a balance is achieved with this size, it provides a good representation while maintaining computational efficiency. The next highest performance comes from the embedding dimension of 128. While this size can potentially capture more intricate relationships in the data, it may also lead to increased complexity.
An embedding dimension of 32 performed the worst. This can indicate that the model lacks the capacity to effectively learn, leading to underfitting.

4.3.2 Impact of patch_size on the performance

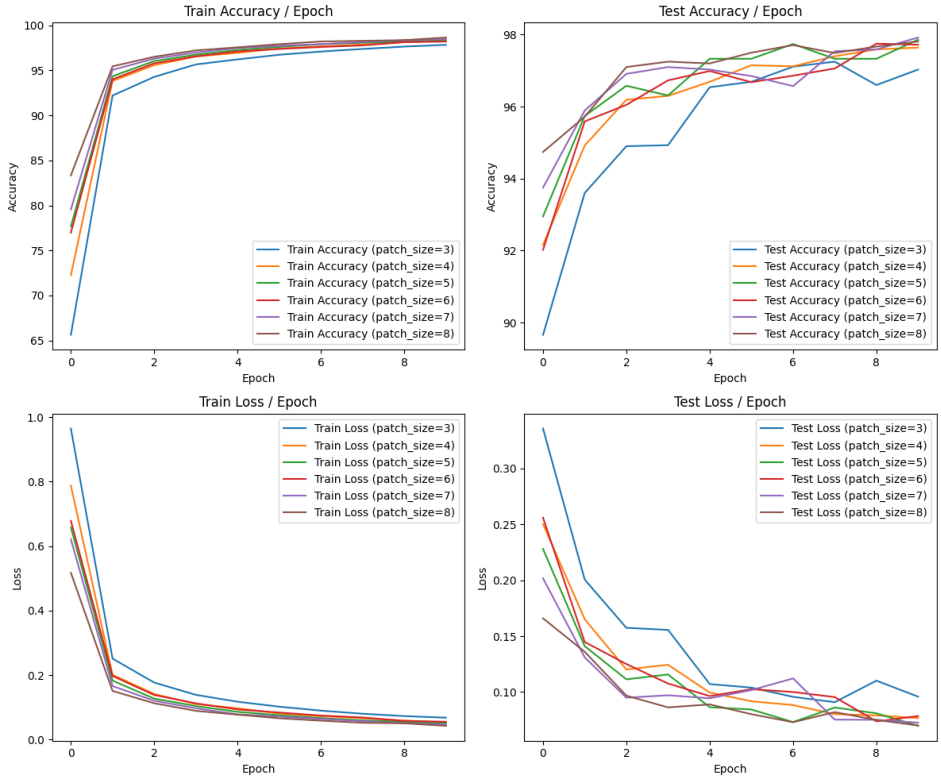


Figure 41: Effects of patch_size

During this experiment, we fixed the embed_dim to 64 which gave the best result in the previous experiment and nb_blocks to 2. In figure 41, we study the impact of the patch size. The smallest patch size of 3 exhibited the poorest performance compared to the other sizes. This suggests that such small patches may lead to an insufficient representation of the image features. Patch sizes of 7 and 8 achieved the best results. By using larger patches, the model can better aggregate information.

4.3.3 Impact of nb_blocks on the performance

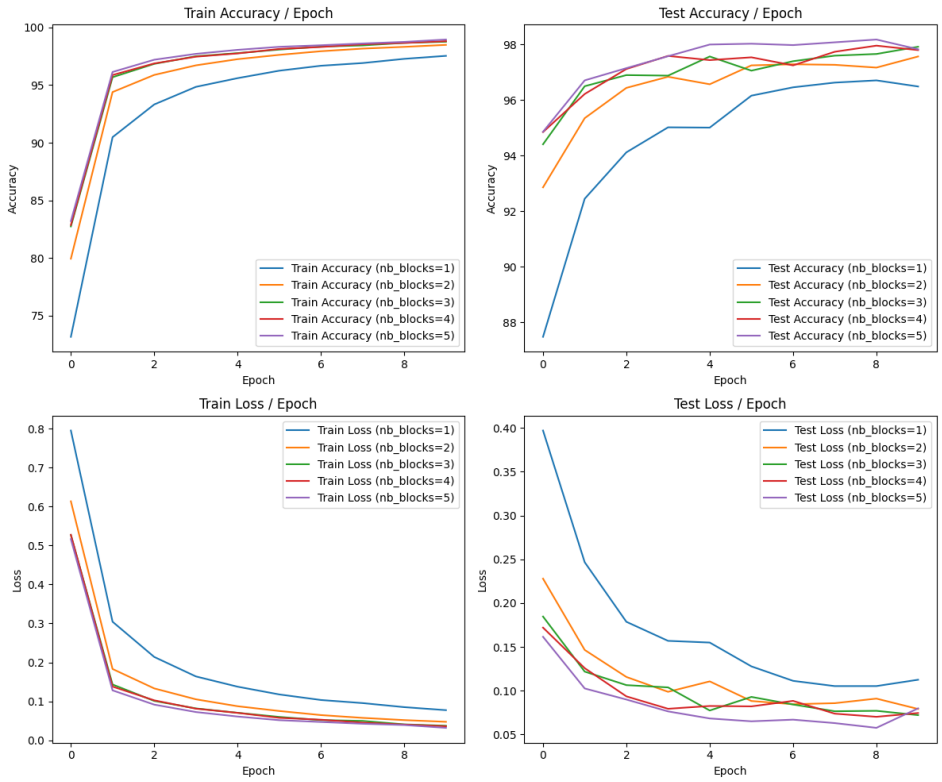


Figure 42: Effects of nb_blocks

During this experiment, we fixed the embed_dim to 64 and patch_size to 7. From figure 42 we can conclude that the model with only 1 block exhibits significant challenges, resulting in insufficient performance. This outcome suggests that a single transformer block lacks the capacity to capture complex patterns. Test accuracy generally increases as we increase nb blocks. While the 5-block configuration initially shows promise with the lowest test loss until around the 8th epoch, it ultimately experiences an increase in loss, indicating potential overfitting. The 3-block model surpasses the performance of the 5-block model by achieving the lowest test loss by the end of training, suggesting that it may achieve a more effective balance between model capacity and generalization.

Comment and discuss the final performance that you get. How to improve it? From our 3 experiments, we can see that (embed_dim=64, patch_size=7, nb_blocks=3) gave the best overall performance. For all 3 hyperparameters, we can say that smaller values underperform whilst large values risk overfitting.

To improve the performance, we can use the Dropout regularization technique which prevents overfitting. It ignores some nodes or parameters like attention weights during training.

We can also use a pretrained version.

What is the complexity of the transformer in terms of number of tokens? How you can improve it?
The complexity of the transformer is in total $O(n^2d + nd^2)$ [3] where n is the number of tokens and d is the dimension of the token embeddings.
To improve it, there are methods such as distillation, sliding window attention.

4.3.4 Q8: Larger Transformers

Load the model using the timm library without pretrained weights. Try to apply it directly on a tensor with the same MNIST images resolution. What it is the problem and why we have it? Explain if we have also such problem with CNNs.

When we load the model without pretrained weights and apply it on a tensor with MNIST image resolution (28x28), we encounter an AssertionError indicating "Input height (28) doesn't match model (224)". This error occurs because the input dimensions of the MNIST images do not match the expected dimensions of the ViT model. The model vit_small_patch16_224 is designed for input images of size 224x224 pixels. Therefore, the input height (28) does not match the expected model height (224).
CNNs can process images of various sizes due to their convolutional layers and pooling operations, as we also mentioned in the 1-cd part.

Comment the final results and provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work. In the final part, we trained a non trained and a pretrained ViT on MNIST dataset. To resolve the problem mentioned before, we defined the number of classes and the image size while creating the model.

In Figure 43 below, we see the final results on these ViTs. The pretrained one clearly outperforms the non pretrained one, starting with a better performance and finishing without overfitting. On average and throughout epochs, the accuracy of the pretrained one is 10 percent higher than the other one.

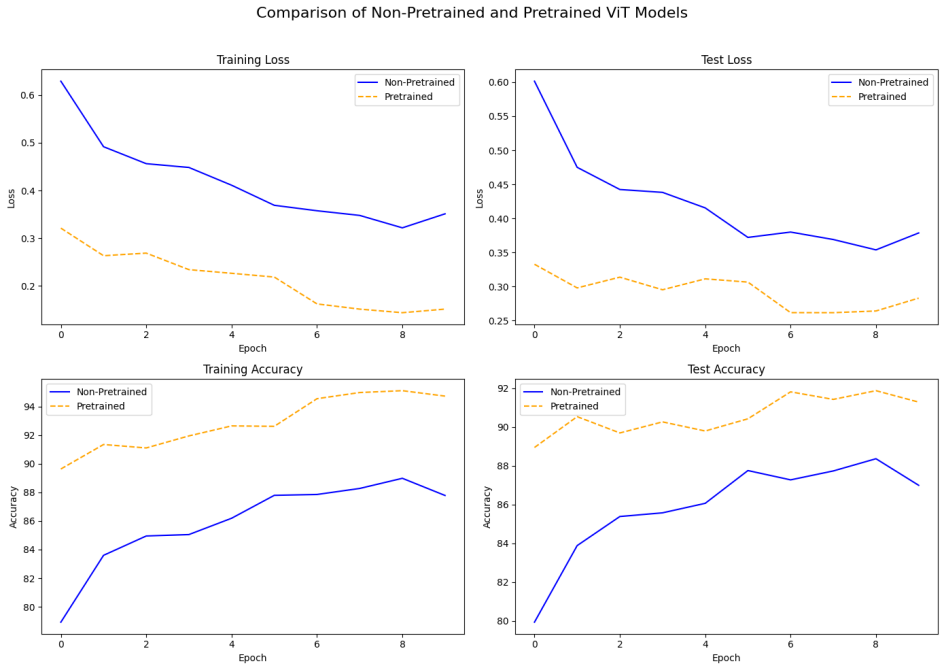


Figure 43: Final results on ViT

4.4 Conclusion

In this section, we emphasized the impact of self-attention mechanisms. We concentrated on ViTs and built one from scratch which performed well on MNIST data. We experimented on the impacts of different hyperparameters to see how we can fine tune the model. Finally, we used the ViT-S from timm library which can be pretrained before use and tested the difference between training a model

without any pretraining and trainin one which has already pretrained weights. The pretrained one outperformed even if the pretraining data was different because of its adaptability.

5 Conclusion

In this report, we explored three deep learning methods, neural networks, convolutional neural networks and transformers which are important methods for computer vision. We experimented on different hyperparameters and their influences, different datasets and their results to learn how to adapt these models.

Our experiments indicate that these models are quite versatile and adaptable depending on the situation. We also focused on data regularization techniques to improve the model's robustness and to avoid overfitting and generalizing the model.

In conclusion, with careful tuning, we can implement these models on various datasets.

References

[1] Activation functions in neural networks [12 types & use cases].

[2] What are convolutional neural networks? | IBM.

[3] Computational complexity of self-attention in the transformer model, 2024.

[4] AnalytixLabs. Convolutional neural networks — definition, architecture, types, applications, and more. https://medium.com/@byanalytixlabs/convolutional-neural-networks-definition-architecture-types-applications-and-more-106ca69d9ae6, 2024. Accessed: 2024-10-27.

[5] Jason Brownlee. Dropout regularization in deep learning models with keras. https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/, 2024. Accessed: 2024-10-27.

[6] Jason Brownlee. A gentle introduction to pooling layers for convolutional neural networks. https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/, 2024. Accessed: 2024-10-27.

[7] Mirza Cilimkovic. Neural networks and back propagation algorithm.

[8] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. On the relationship between self-attention and convolutional layers. *CoRR*, abs/1911.03584, 2019.

[9] Pouya Hallaj. Data augmentation: Benefits and disadvantages. https://medium.com/@pouyahallaj/data-augmentation-benefits-and-disadvantages-38d8201aead, 2024. Accessed: 2024-10-27.

[10] Rick Merritt. What is a transformer model?

[11] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[12] Stack Exchange User. How are 1x1 convolutions the same as a fully connected layer? https://datascience.stackexchange.com/questions/12830/how-are-1x1-convolutions-the-same-as-a-fully-connected-layer, 2024. Accessed: 2024-10-27.

[13] Nikola M. Zivkovic. Receptive field calculations for convolutional neural networks. https://rubikscode.net/2021/11/15/receptive-field-arithmetic-for-convolutional-neural-networks/, 2024. Accessed: 2024-10-27.