RDFIA - M2 IMA

Practical work report 1-ab
Introduction to neural networks

Done by :

Carlos GRUSS & Oussama RCHAKI

25/09/24

Contents

Introduction

A neural network in deep learning is a computational model inspired by the human brain, designed to recognize patterns and solve complex tasks. It consists of layers of interconnected nodes (or neurons), where each node processes input data and passes the result to the next layer. These networks learn from data by adjusting the connections between neurons based on errors made during predictions, a process guided by backpropagation. Neural networks are used in a wide range of tasks, including image recognition, natural language processing, and predictive modeling.

The objective of this lab exercise is to construct a basic feed-forward neural network and develop a strong understanding of these models, as well as their training process using gradient backpropagation. We will start by exploring the theory behind a a network with a single hidden layer and its learning mechanism. Following this, we'll implement the network using the PyTorch library, first testing it on a simple example to ensure its functionality, and then applying it to the MNIST dataset for a more practical demonstration.

1 – Theoretical foundation

In order to apply a neural network to a supervised learning problem, four key components must be tailored to the specific task:

- A labeled dataset;
- A suitable network architecture;
- A loss function to be optimized;
- An optimization algorithm to minimize the loss function.

1.1 – Supervised dataset

We are dealing with a supervised classification problem. In this context, we have access to a labeled dataset composed of N pairs of (features, targets) $(x^{(i)}, y^{(i)})$, where $i \in \{1, 2, \ldots, N\}$, and each feature vector $x^{(i)} \in \mathbb{R}^{n_x}$ is associated with a target (or ground truth) $y^{(i)} \in \{0, 1\}^{n_y}$, such that $\|y^{(i)}\| = 1$. This represents a one-hot encoding, meaning that only one element of $y^{(i)}$ is equal to 1, indicating the class to which the sample belongs. The dataset is typically divided into several subsets: training, testing, and optionally validation.

Question 1

These three sets are used for :

- **Train set:** Used to train the neural network by adjusting the model's weights through the optimization process. The model learns from this data.
- **Validation set:** Used to tune hyperparameters and monitor the model's performance during training to avoid overfitting. The model doesn't learn from this data, but it helps guide adjustments like learning rate, network architecture, etc.
- **Test set:** Used to evaluate the final performance of the trained model on unseen data, providing an estimate of how well the model generalizes to new, real-world data.

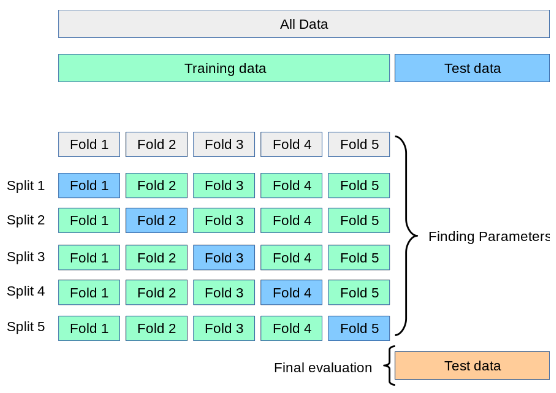


Figure 1 : Cross-validation split.

Question 2

In general, having a larger dataset allows for better learning of the underlying distribution in supervised classification tasks for several reasons:

- With more examples, the influence of random noise diminishes, reducing the risk of overfitting, especially in complex models that require more data to generalize well. This enables the model to focus on true underlying patterns rather than fitting to random fluctuations or memorizing the training data, resulting in more reliable models that perform better on unseen data.
- Conversely, if a model is too simple relative to the data's complexity, it may underfit, failing to capture essential structures in the data. Having more data can highlight complex patterns that necessitate more sophisticated models.
- A larger dataset increases the likelihood of including sufficient examples of minority classes or rare events. This is crucial for the model to learn and accurately predict these less frequent but important cases.

However, large datasets can be challenging to construct and require more computational resources to process. Additionally, arbitrarily increasing N may eventually lead to diminishing returns, as the model's performance gains plateau with additional data.

1.2 – Network architcture (forward)

In this section, we will explore the fundamental architecture of neural networks, focusing on the forward propagation process. Forward propagation refers to the method by which input data is passed through the network's layers to produce an output. Each layer of the network consists of neurons that perform weighted computations on the input, apply activation functions, and pass the result to subsequent layers. Understanding this process is crucial for grasping how neural networks learn and make predictions. We will examine the key components of the architecture, including input layers, hidden layers, and output layers, as well as the significance of activation functions in transforming the data at each stage.

Question 3

Generally, each linear transformation in a neural network is followed by a non-linear activation function, which serves a crucial role by introducing non-linearity into the model. This non-linearity enables the network to learn complex patterns and relationships within the data, allowing for the effective stacking of multiple transformations.

Question 4

In Figure 2, we have the following sizes:

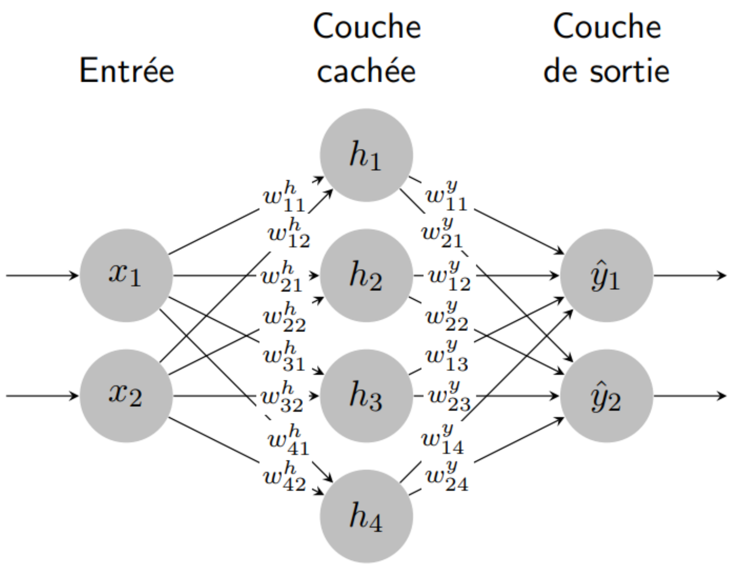


FIGURE 2 : Neural network architecture with only one hidden layer.

- $n_x = 2$: the number of input features
- $n_h = 4$: the number of hidden units in the hidden layer
- $n_y = 2$: the number of output units

In practice, n_x corresponds to the dimensionality of the input features fed into the model, n_h is typically chosen based on the desired complexity and capacity of the network, and n_y depends on the nature of the task. For instance, in a classification task with k classes, we set $n_y = k$, where the output of the i-th neuron represents the probability that the input corresponds to the i-th class.

QUESTION 5

The vector $\hat{y}$ denotes the output generated by our neural network, representing the model's predictions. In contrast, the vector y signifies the ground truth, encompassing the target values or the actual data we aim to predict.

The difference between theses two vectors is referred to as the residual or prediction error, which is quantified in our network by the loss function.

QUESTION 6

The SoftMax activation function normalizes the raw output of a neural network into a probability distribution, where each value in the output vector is between 0 and 1, and the sum of all values equals 1. This is particularly useful for k-class classification problems, as the output values can be interpreted as the predicted probabilities that a given input corresponds to each of the k classes. At test time, the class with the highest probability is typically chosen as the model's prediction. Moreover, during training, having non-binary outputs provides richer information for gradient-based optimization, which helps the model adjust weights more effectively than with binary outputs.

QUESTION 7

The mathematical equations allowing to perform the forward pass of the neural network are :

$$\tilde{h} = xW_h^T + b_h \tag{1}$$

$$h = \tanh(\tilde{h}) \tag{2}$$

$$\tilde{y} = hW_y^T + b_y \tag{3}$$

$$\hat{y} = \text{Softmax}(\tilde{y}) \tag{4}$$

Where:

- $x \in \mathbb{R}^{1 \times n_x}$
- $y \in \mathbb{R}^{1 \times n_y}$
- $h \in \mathbb{R}^{1 \times n_h}$
- $W_h \in \mathbb{R}^{n_h \times n_x}$
- $W_y \in \mathbb{R}^{n_y \times n_h}$

1.3 – Loss function

The loss function is a critical component in training neural networks, as it quantifies the difference between the model's predictions and the actual target values. By measuring this error, the loss function guides the optimization process, helping the model adjust its parameters to improve accuracy. In this section, we will explore different types of loss functions, such as those used for classification and regression tasks, and understand their role in shaping the learning process.

Question 8

- For **cross-entropy**, given a one-hot encoded target vector y, the predicted probability $\hat{y}_i$ must be as close as possible to 1 for the entry corresponding to the true class (i.e., the non-zero value of the target). This effectively pushes the predicted probability of the correct class to 1 while pushing the probabilities for all other classes towards 0. Minimizing cross-entropy involves maximizing the log-probability of the correct class.
- For **mean squared error (MSE)**, the predicted values $\hat{y}_i$ must closely match the target values y_i for all entries. MSE penalizes the squared difference between the predicted values and the target values, so minimizing it requires each $\hat{y}_i$ to approach the corresponding target value y_i as closely as possible.

Question 9

- **Cross-Entropy for Classification**: Cross-entropy compares the predicted probability $\hat{y}_i$ with the true one-hot encoded labels y_i. Since only the correct class has a value of 1, the loss simplifies to $-\log \hat{y}_{\text{true class}}$, meaning the model is penalized based only on the probability it assigns to the correct class, encouraging the model to increase this probability. Additionally, when dealing with regression outputs, it is possible for $\hat{y}$ to exceed 1, which would result in a positive logarithm and thus a negative loss value, which is not meaningful in the context of a loss function.
- **MSE for Regression**: Mean squared error calculates the difference between predicted and actual values $\hat{y}_i$ and y_i using the formula $\sum_i (y_i - \hat{y}_i)^2$. Squaring the difference penalizes larger errors more, encouraging the model to make predictions that are closer to the actual continuous values.

1.4 – Optimization algorithm (Gradient descent)

Gradient descent is one of the most widely used optimization algorithms in deep learning. It works by iteratively updating the model's parameters in the direction that reduces the loss function, aiming to find the optimal set of parameters. In this section, we will examine how gradient descent operates, the different variants of the algorithm, and its importance in efficiently training neural networks.

Question 10

- Classic gradient descent has the advantage of providing more accurate and stable gradient updates, as it computes the gradient using the entire dataset at each step. This can lead to a more stable convergence. However, for large datasets, this approach becomes computationally expensive and slow since it must process all data before making a single update. Additionally, it can be prone to getting stuck in local minima or saddle points because of the deterministic nature of its updates.
- Mini-batch stochastic gradient descent strikes a balance by computing the gradient on small random subsets (mini-batches) of the data. This makes the gradient computation less intensive, speeding up the learning process while also introducing some randomness. The stochastic nature helps the algorithm escape local minima or saddle points. Mini-batches also provide a smoother approximation of the true gradient compared to pure stochastic gradient descent, which leads to more reliable updates.
- Online stochastic gradient descent (or simply stochastic gradient descent, SGD) computes the gradient for each individual example, making it the fastest in terms of gradient updates. This results in noisy updates, which can help escape local minima but also leads to higher variance in the optimization path, making convergence less stable. Furthermore, the high variance can cause the model to over-fit specific examples since the gradient at any given step is based on a single example, which may not be representative of the whole dataset.

In practice, mini-batch gradient descent is the most commonly used variant because it provides a good trade-off between computational efficiency and stable, reliable convergence.

Question 11

The learning rate η plays a crucial role in the training process of a neural network. It determines the size of the steps taken during gradient descent to update the model's parameters.

- **If η is too large** the model may overshoot the optimal point, causing the loss function to fluctuate or even diverge, preventing the model from converging and making the learning process more unstable.
- **If η is too small** the model will learn very slowly, requiring many iterations to converge. This can lead to long training times and may also cause the model to get stuck in local minima.

An appropriate learning rate strikes a balance between fast convergence and stable learning, enabling the model to quickly reduce the loss without overshooting.

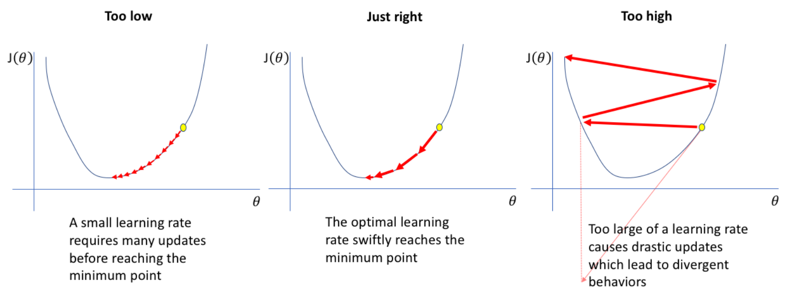

Too low
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
Just right
J(θ)
θ
The optimal learning rate swiftly reaches the minimum point
Too high
J(θ)
θ
Too large of a learning rate causes drastic updates which lead to divergent behaviors

FIGURE 3 : Gradient descent with high, low and optimal value of η.

Question 12

The naive approach has significantly higher computational complexity than the backpropagation algorithm. To illustrate, let's revisit the example of a neural network with one hidden layer. In this case, the backpropagation algorithm requires calculating $n_x n_h + n_h n_y = 16$ gradients (that is, one gradient for each parameter), whereas the naive method would compute $2 n_x n_h n_y = 32$ gradients.

Now, consider a more general neural network with d hidden layers, each having n neurons. Without taking into account the output layer weights (which wouldn't affect the comparison), the number of gradients that the backpropagation algorithm must compute in this setup is $(d-1)\cdot n^2$, whereas the naive approach would require computing $(d-1)\cdot n^d$ gradients.

Question 13

For the network architecture to allow for an optimization procedure like backpropagation, it must meet the following criteria:

- **Differentiability:** The activation functions, loss functions and the layers in the network must be differentiable, as backpropagation relies on calculating gradients of the loss function with respect to the model parameters.
- **Layered structure:** The network must have a layered structure where each layer's output depends on the previous layer. This allows the gradients to be propagated backward through the network.
- **Weight parameters:** The network must have trainable weight parameters, which are updated during the optimization process based on the computed gradients.

Question 14

We have that the cross-entropy loss is given by:

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

Using SoftMax as the activation function for the output layer we have:

$$\hat{y}_i = \text{SoftMax}(\tilde{y})_i = \frac{\exp \tilde{y}_i}{\sum_j \exp \tilde{y}_j}$$

Substituting in the previous equation:

$$\begin{aligned} l(y,\hat{y}) &= -\sum_i y_i \log\left(\frac{\exp \tilde{y}_i}{\sum_j \exp \tilde{y}_j}\right) \\ &= -\sum_i y_i \left(\tilde{y}_i - \log\left(\sum_j \exp \tilde{y}_j\right)\right) \\ &= -\sum_i y_i\tilde{y}_i + \sum_i y_i \log\left(\sum_j \exp \tilde{y}_j\right) \end{aligned}$$

Finally:

$$\boxed{l(y,\hat{y}) = -\sum_i y_i\tilde{y}_i + \log\left(\sum_j \exp \tilde{y}_j\right)} \tag{5}$$

Where the last equivalence arises from observing that y_i is a one-hot encoded vector.

Question 15

Computing the gradient of the cross-entropy loss with respect to the intermediate output $\tilde{y}$:

$$\frac{\partial l}{\partial \tilde{y}_i} = -y_i + \frac{\partial \log\left(\sum_j e^{\tilde{y}_j}\right)}{\partial \tilde{y}_i} = -y_i + \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} = -y_i + \text{Softmax}(\tilde{y}_i) = \hat{y}_i - y_i$$

Finally, we have:

$$\boxed{\nabla_{\tilde{y}} l = \hat{y} - y} \tag{6}$$

Question 16

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

From the previous question we get:

$$\begin{aligned}
\frac{\partial l}{\partial W_{y,ij}} &= \sum_k (\hat{y}_k - y_k) \frac{\partial}{\partial W_{y,ij}} \left(b^y_k + \sum_{l=1}^{n_h} h_j w^y_{kl} \right) \\
&= \sum_k (\hat{y}_k - y_k) h_j \delta_{k=i} \\
&= (\hat{y}_i - y_i) h_j
\end{aligned}$$

We can thus write:

$$\nabla_{W_y} l = (\hat{y} - y) \otimes h = (\hat{y} - y)^T \cdot h = \boxed{\nabla_{\tilde{y}} l^T \cdot h} \tag{7}$$

Where "$\cdot$" represents matrix multiplication, $(\hat{y} - y)$ and h are $1 \times n_y$ and $1 \times n_h$ row vectors respectively.

Likewise:

$$\begin{aligned}
\frac{\partial l}{\partial b_{y,i}} &= \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial b_{y,i}} \\
&= \sum_k (\hat{y}_k - y_k) \frac{\partial}{\partial b_{y,i}} \left(b^y_k + \sum_{l=1}^{n_h} h_j w^y_{kl} \right) \\
&= \sum_k (\hat{y}_k - y_k) \delta_{k=i} \\
&= (\hat{y}_i - y_i)
\end{aligned}$$

So we get:

$$\boxed{\nabla_{b_y} l = \nabla_{\tilde{y}} l^T} \tag{8}$$

Where we're considering the bias vector as a column vector of size $n_y \times 1$.

Question 17

Computation of $\nabla_{\tilde{h}} l$

We start writing:

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

Also:

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

Using the equations 3 and 6 we obtain the formula:

$$\frac{\partial l}{\partial h_k} = \sum_j \frac{\partial l}{\partial \tilde{y}_j} \frac{\partial \tilde{y}_j}{\partial h_k} = \sum_j (\hat{y}_j - y_j) W_{y,jk}$$

So:

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} = (1 - h_i^2) \left(\sum_j (\hat{y}_j - y_j) W_{y,ji} \right)$$

Finally, we write:

$$\boxed{\nabla_{\tilde{h}} l = (\nabla_{\tilde{y}} l \cdot W_y) \odot (1 - h^2)} \tag{9}$$

where we consider that the operation $\odot$ represents the element-wise multiplication, also known as the Hadamard product, referring to the operation where two vectors (or matrices) of the same size are multiplied together such that each element in the first vector is multiplied by the corresponding element in the second vector. Here W_y is of dimension $n_y \times n_h$.

Computation of $\nabla_{W_h} l$

Starting by the formula:

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

Using the equations 1 and 9 we obtain the formula:

$$\frac{\partial l}{\partial W_{h,ij}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} = \frac{\partial l}{\partial \tilde{h}_i} \frac{\partial \tilde{h}_i}{\partial W_{h,ij}} = (1 - h_i^2) \left(\sum_j (\hat{y}_j - y_j) W_{y,ji} \right) x_j$$

Finally, we have:

$$\boxed{\nabla_{W_h} l = \nabla_{\tilde{h}} l^T \cdot x} \tag{10}$$

Where we consider that x is a row vector of size $1 \times n_x$, with $\nabla_{\tilde{h}} l$ being of dimension $1 \times n_h$.

Computation of $\nabla_{b_h} l$

We have:

$$\frac{\partial l}{\partial b_{h,i}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \frac{\partial l}{\partial \tilde{h}_i} \frac{\partial \tilde{h}_i}{\partial b_{h,i}} = \frac{\partial l}{\partial \tilde{h}_i}$$

Finally:

$$\boxed{\nabla_{b_h} l = \nabla_{\tilde{h}} l^T} \tag{11}$$

2 – Implementation

We now have all the necessary equations to make predictions (forward pass), evaluate the performance (loss function), and train our model using backpropagation and gradient descent. Next, we will implement this network using PyTorch.

We will begin by experimenting with the "Circle" dataset (refer to figure 4), which features data points organized in two concentric circles, a common benchmark for tackling nonlinear classification challenges. Afterward, we will apply the same code to the MNIST dataset. MNIST consists of images of handwritten digits (see Figure 5) and includes 10 classes ($ny = 10$). Each image is 28×28 pixels, represented as a vector of 784 values.

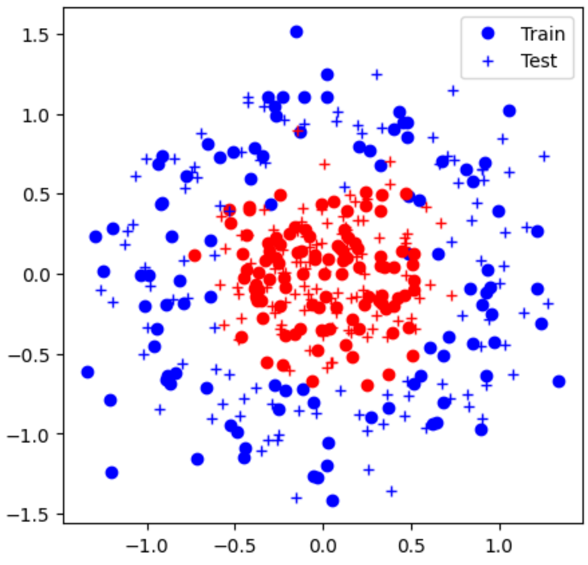


Figure 4 : Circle dataset.

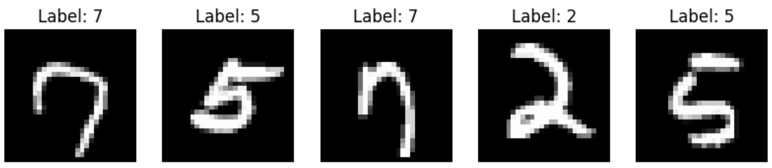


Figure 5 : MNIST dataset.

2.1 – Manual implementation

In this section, we carefully designed both the forward and backward propagation functions, utilizing the Stochastic Gradient Descent (SGD) optimizer. We employed a learning rate of 0.03 and divided the data into 10 batches. After 150 epochs, the model successfully converged, reaching an accuracy of 97.5% on the training set and 93.5% on the test set, as illustrated in the Figure 6.

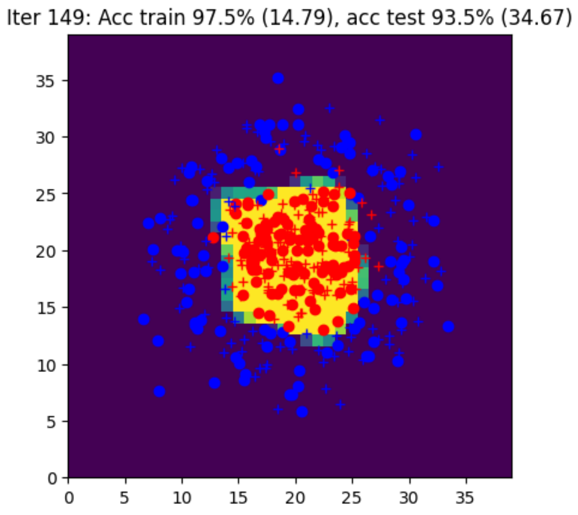
By the end of the training, it is evident that the model has learned circular decision boundaries, enabling accurate classification of the circular dataset.

Next, we experiment what was done with with various learning rates and batch sizes to observe their respective influences on the model's performance.

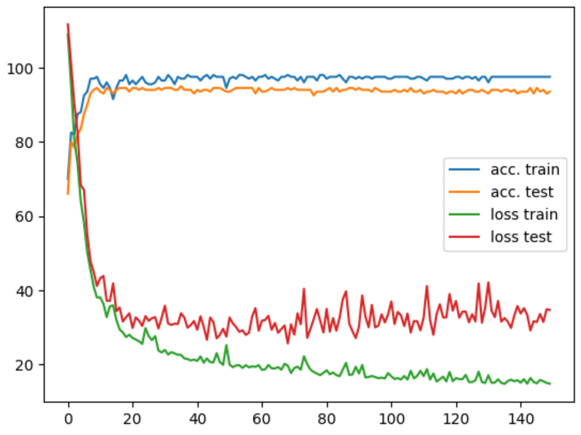
We observe that with a large learning rate (LR=1), our model oscillates significantly and does not converge. Conversely, with a small learning rate (LR=0.001), the model takes many iterations to converge (Fig 7a), which is consistent with the theory we discussed earlier.

In Figure 7b, we observe the evolution of the loss function as a function of the number of epochs for each batch size N_{batch}. We see that for $N_{\text{batch}} = 200$, the model shows instability at the end of each iteration. This is not contradictory to what we found in Question 10, where we concluded that the model is more stable with larger batch sizes. In the case where $N_{\text{batch}} = 200 = N$, the model updates the gradient only once per iteration, which is naturally less precise compared to updating the gradient multiple times when using batch sizes smaller than 200.

Finally, in Figure 8 we see how the decision boundary evolves to correctly represent the likely underlying distribution of the Circles dataset.

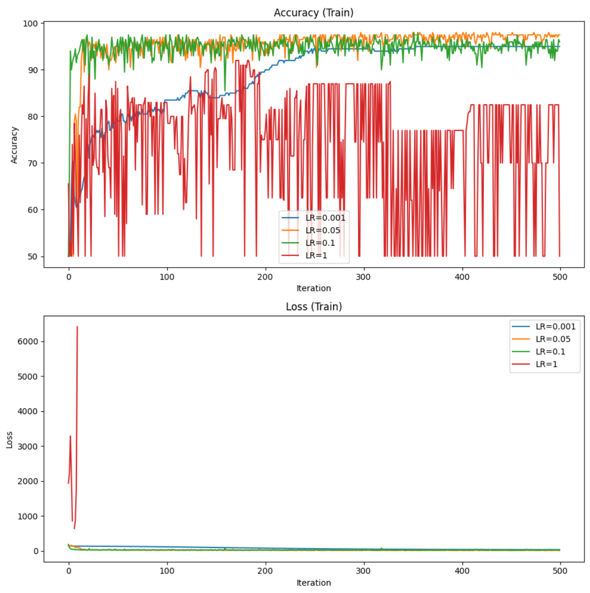


(A) Classification of our algorithm on the Circle dataset.

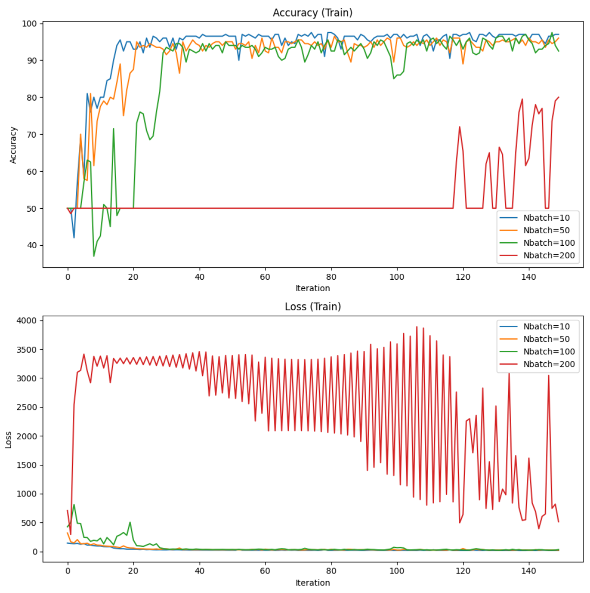


(B) Loss and accuracy functions with respect to the training and test datasets.

FIGURE 6 : Results of our algorithm applied to the Circle dataset with manual implementations of the forward and backward functions.



(A) Influence of learning rate.



(B) Influence of batch size.

FIGURE 7 : Influence of learning rate and batch size.

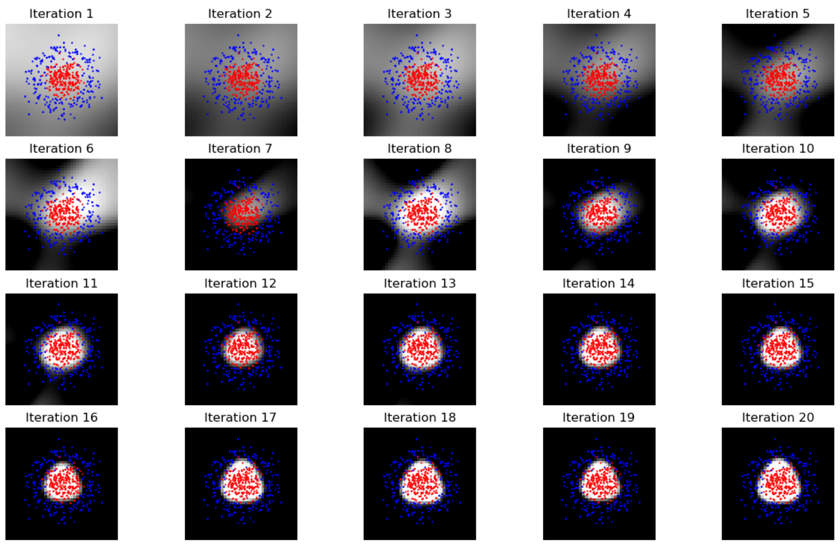


FIGURE 8 : Evolution of the decision boundary.

2.2 – Simplification of the Backward Pass with `torch.autograd`

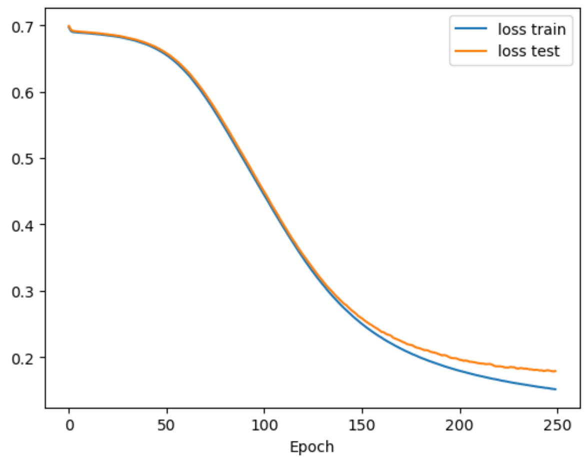
Instead of manually implementing the backward pass—requiring closed-form expressions for the gradient of the loss with respect to each parameter in the network—PyTorch provides automatic differentiation through its `torch.autograd` engine.

`torch.autograd` constructs a computational graph that dynamically tracks the operations performed on the parameters we aim to optimize. With this graph, we can compute the gradient of the relevant parameters with respect to a specified quantity using the `backward` method.

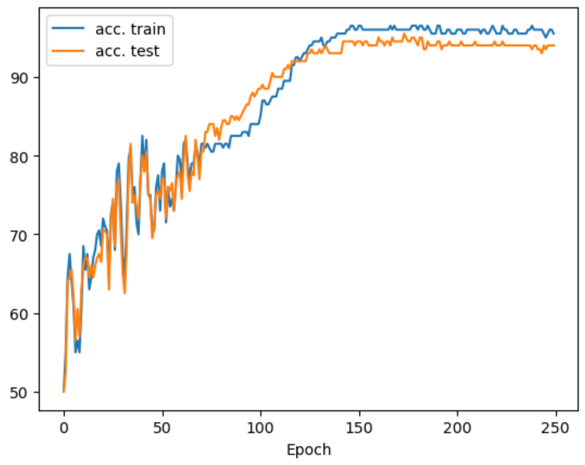
This approach is not only more efficient and accurate, but it also enables gradient computation for much more complex deep learning architectures, where deriving closed-form expressions for gradients may be impractical or impossible.

For this section, we used a learning rate of 0.03 and a batch size of 10, since we observed these hyper-parameters offered a good performance in the previous experiments. The maximum performance achieved on the test set was 96% accuracy and the evolution of the train/test accuracy and loss can be observed in Figure 9.

One thing we can observe is that the evolution of the loss is much more stable and smooth with respect to what we observed in Figure 6. We hypothesize this must be mainly due to the greater numerical precision achieved on the computed gradients when using `torch.autograd` instead of manual computation.



(A) Evolution of the loss.



(B) Evolution of the accuracy.

FIGURE 9 : Loss and accuracy using `torch.autograd`.

2.3 – Simplification of the forward pass with `torch.nn` layers

When training our model using the forward pass with `torch.nn` layers, we observe that the training process is more stable, as evidenced by the loss function's behavior (Fig 11a). This stability can be attributed to the efficient implementations and optimizations provided by the `torch.nn` module, which leverage techniques such as batch normalization and optimized activation functions. These features help to mitigate issues such as exploding or vanishing gradients, which are common challenges in deep learning. As a result, the model converges more reliably, leading to a smoother decline in the loss function over time (iterations). However, this increased stability often comes at the cost of longer training times. The complexity of the underlying computations and the added layers of abstraction in the `torch.nn` framework can require more time for each training iteration.

Figure 10 demonstrates that we achieved a test accuracy of 94% with this implementation.

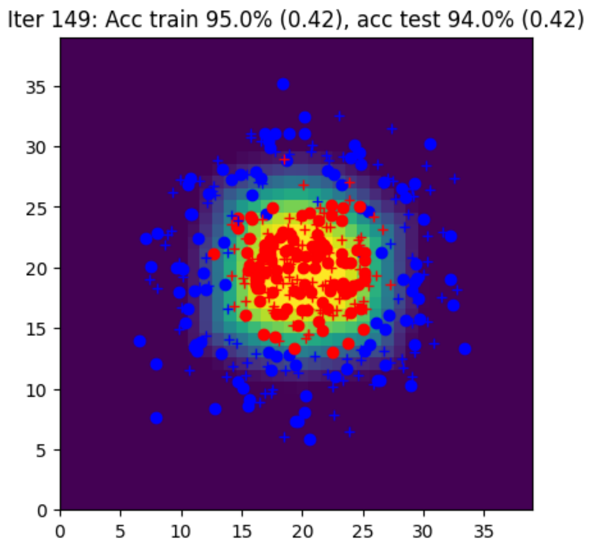
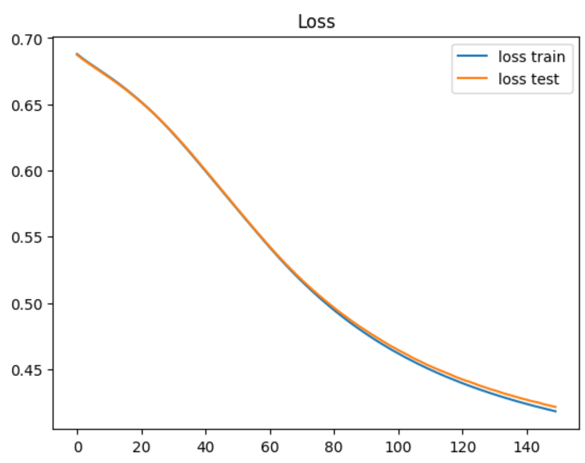
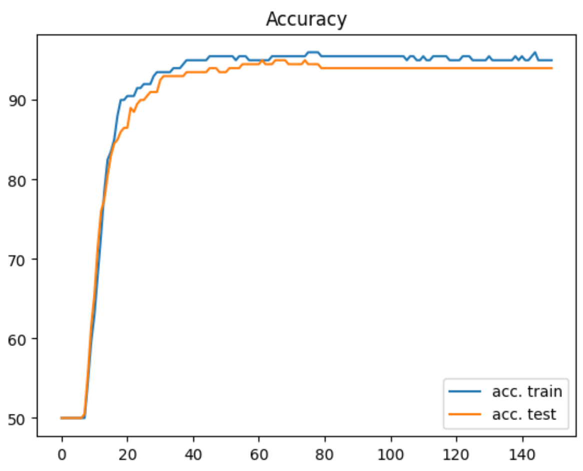


FIGURE 10 : Classification of our algorithm (using the forward pass with torch.nn layers) on the Circle dataset.



(A) Loss function.



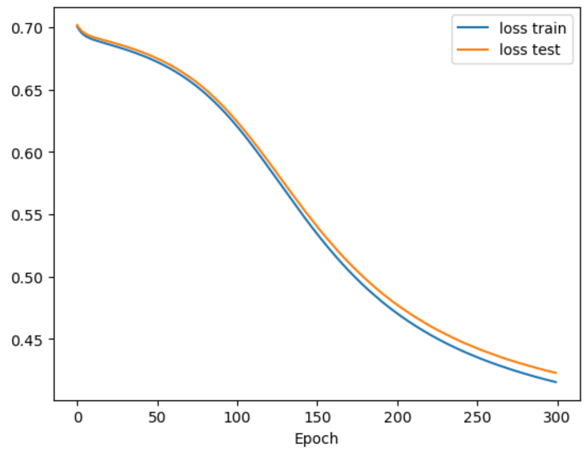
(B) Accuracy function.

FIGURE 11 : Loss and accuracy functions of our trained model using the forward pass with torch.nn layers.

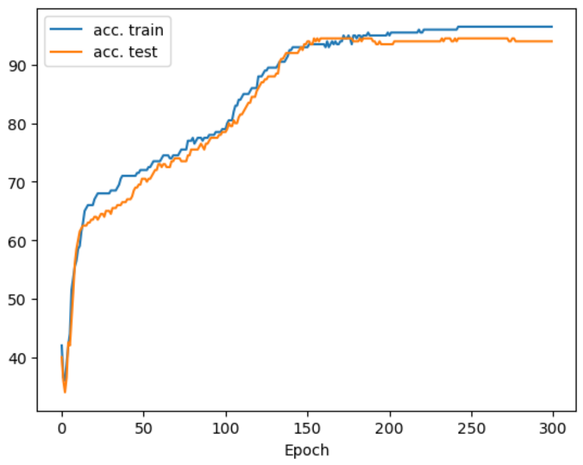
2.4 – Simplification of the SGD with `torch.optim`

We can further improve and simplify the implementation of the neural network by using the `torch.optim` package, which contains various popular optimization algorithms. In this particular case, we used PyTorch's implementation of Stochastic Gradient Descent (`torch.optim.SGD`).

The same hyper-parameters where used as in Section 2.2, and the results can be observed in Figure 12.



(A) Evolution of the loss.



(B) Evolution of the accuracy.

FIGURE 12 : Loss and accuracy using `torch.optim`.

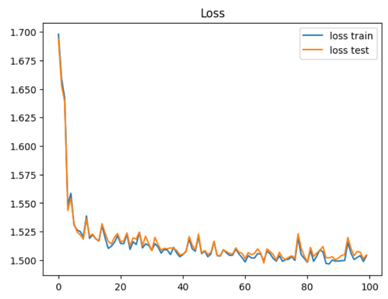
2.5 – MNIST application

In this section, we present the results of our model on the MNIST dataset. The model achieved impressive results with a test accuracy of 95.3% and a training accuracy of 95.7%.

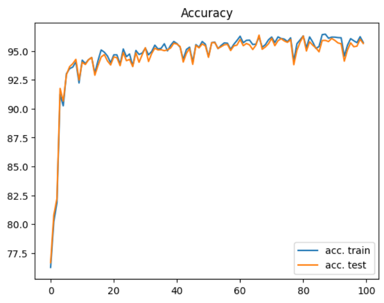
We also observed that hyperparameters significantly influence the performance of the model in this use case. For example, setting nh (number of hidden units) to 10, or using a batch size of 10, caused the model to converge to a test accuracy of only 71%. The results we report were obtained with $nh = 100$ and $Nbatch = 100$.

The confusion matrix, which is a table used to describe the performance of a classification model by comparing predicted and true labels, shows strong predictions for the digit 1, likely because it is a simple number to draw. However, there is some confusion between the digits 2 and 8, and 3 and 8, as they can appear visually similar.

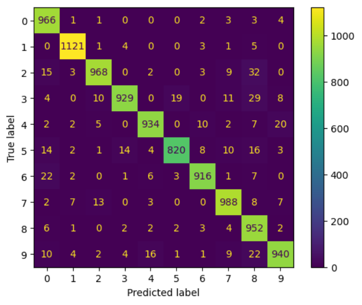
In Figure 14 we can see examples of the digits that get misclassified. As we can see, they're very difficult examples, that even a human would have trouble correctly classifying.



(A) Loss function.



(B) Accuracy function.



(C) Confusion Matrix.

Figure 13 : Loss and accuracy functions of our model applied to the MNIST dataset.

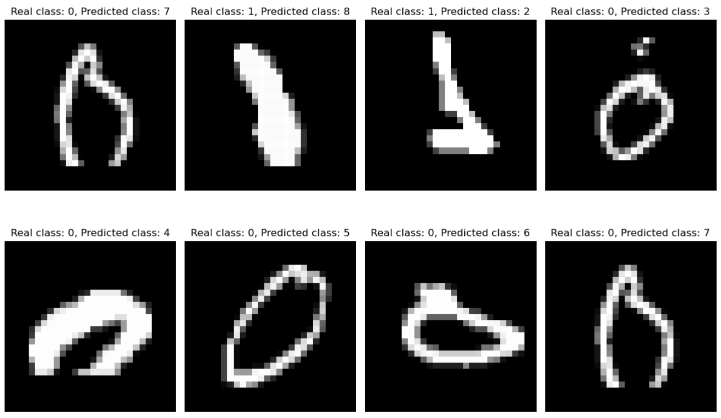

Real class: 0, Predicted class: 7
Real class: 1, Predicted class: 8
Real class: 1, Predicted class: 2
Real class: 0, Predicted class: 3
Real class: 0, Predicted class: 4
Real class: 0, Predicted class: 5
Real class: 0, Predicted class: 6
Real class: 0, Predicted class: 7

Figure 14 : Examples of misclassification on MNIST.

2.6 – Bonus: SVM

For the final part, we trained an SVM classifier on the same Circle dataset using the `scikit-learn` Python library. We can observe the results for various different kernels and regularization parameter values. The results are shown in Figure 15.

In these preliminary experiments, we observed that the best performance is obtained with the radial basis function (RBF) kernel, achieving an accuracy comparable to that of the implemented neural network.

However, we further experimented with the performance of the sigmoid and polynomial kernels by tuning the relevant hyperparameters—the degree in the case of the polynomial kernel and the scale (gamma) in the case of the sigmoid kernel. The results can be seen in Figures 16 and 17.

We observed very good performance with the polynomial kernel of degree 2, which also has the advantage of being a very simple kernel. We further explored the effect of the parameter C on this particular kernel (Figure 18) and achieved a maximum test performance of 95.5%. We could argue then that, in this simple dataset, SVMs can perform as well as or even better than neural networks, making them a convenient and more lightweight solution.

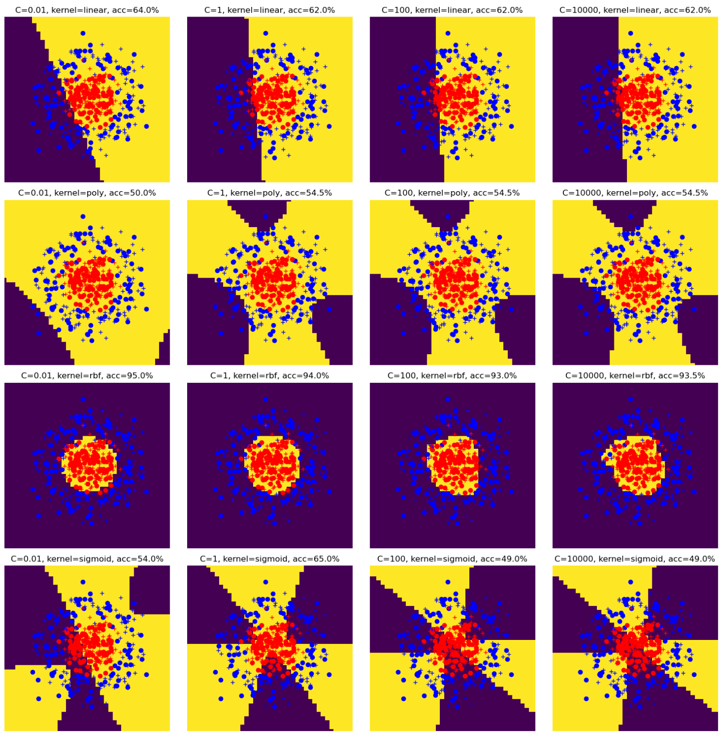


FIGURE 15 : SVMs trained with varying hyperparameter values.

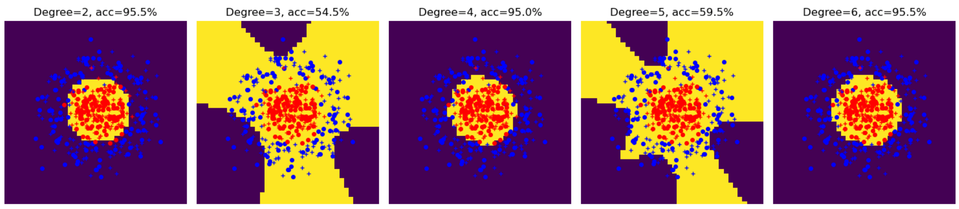


FIGURE 16 : SVM with polynomial kernel of varying degree.

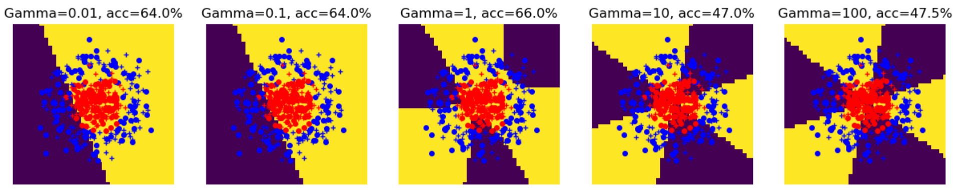


FIGURE 17 : SVM with sigmoid kernel of varying scale (gamma).

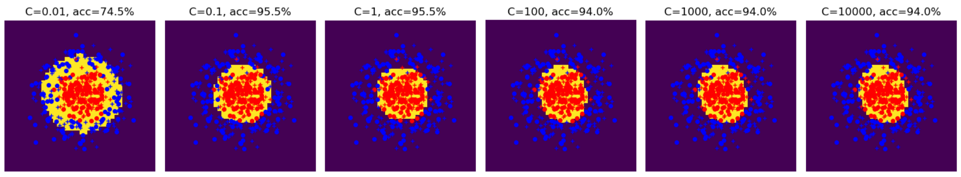


FIGURE 18 : SVM with polynomial kernel and varying regularization parameter C.