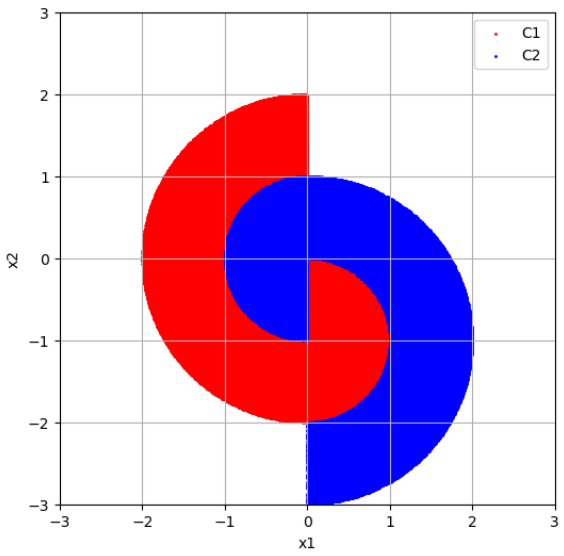
COEN 6331
Neural Networks

Problem Set 1

Two-dimensional non-linear classification problem



Benjamin Vauchel

February 14th, 2024

Table of Contents

Table of Figures

Introduction

Classification problems can be solved with unsupervised and supervised machine learning. When labels are provided, it is preferable to use supervised machine learning as they allow the model to accurately learn the relationship between inputs and outputs. There are many supervised machine learning methods, such as K-nearest neighbors, decision trees, linear and logistic regressions, support vector machines, naive Bayes classifiers, etc. However, some methods cannot solve a non-linear classification problem, which required to extend them to handle non-linear data. In this assignment, we are interested in solving a two-dimensional non-linear classification problem using deep learning by creating a neural network. Neural networks, especially deep networks, are inherently good at capturing non-linearity due to their layered architecture and activation functions.

The problem is to design and implement a neural network classifier that decides whether an input pattern point belongs to one class or the other. In addition, the objective of this work is to study the effects of change of the structure of the network and the hyperparameters above the performance of the network (speed, convergence, accuracy, etc.).

Analysis

The distribution of pattern classes C_1 and C_2 described in the problem shows non-convex decision regions.

The choice of pattern sets used for this problem is very important, since the effectiveness of a model relies heavily on the quality of the data. A large pattern set is created, from which two pattern sets are extracted: the training set and the test set. We have chosen to generate points randomly and uniformly in the area combining zones C_1 and C_2, and label them subsequently. To respect this uniform distribution, we used polar coordinates, randomly generating a radius and an angle. Random adjustments allow us to generate the point in the desired zone, and not just in a semicircle. In addition, it is necessary to take the square root of the random number between 0 and 1 to generate the radius, in order to have a uniform distribution in the circle; otherwise, more points would be generated in the center of the circle. **Figure 2** shows the repartition of pattern points in the area with and without the square root of the radius.

The label for each pattern point is determined mathematically. If a point lies exactly on a boundary between the two classes, it is randomly labeled in one class or the other. Points in class C_1 are displayed in red, and those in class C_2 in blue (**Figure 1**).

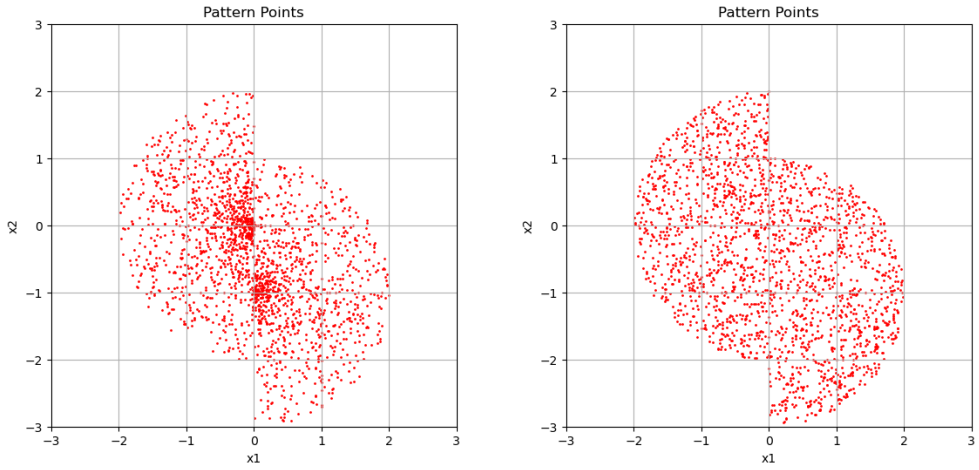


Figure 2. 2000 pattern points generated without the square root of the radius (left) and with the square root (right)

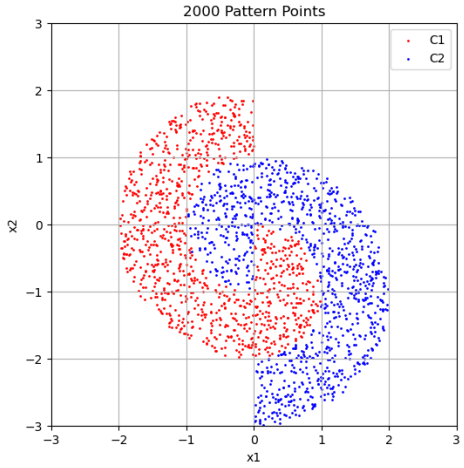


Figure 1. 2000 labeled pattern points

These pattern points are normalized before being used, i.e. the mean of each coordinate is subtracted and divided by the standard deviation. This step is important as it helps in speeding up the learning process by ensuring that the gradient descent moves smoothly and more uniformly across all inputs, preventing the issue of exploding or vanishing gradients.

For this problem, 70% of the pattern points are used for the training phase, and 30% for the test phase. Of the 70%, we actually consider 80% for the training dataset and 20% for the validation dataset.

To solve this classification problem, we implement a multilayer perceptron from scratch. The latter is modular: its structure and other values are parameters that can be modified for comparison.

The network has two inputs, which are the x_1 and x_2 coordinates of the point pattern, and one output corresponding to the probability that the point pattern belongs to class C_1. It is worth noting that we could have opted for two outputs, corresponding to the probability of being in each of the classes, but since this is a binary classification problem, we avoid this redundancy which would add complexity to the model. The number of hidden layers and the number of neurons for each layer can be modulated to analyze the consequences.

Weights are initialized with a normal distribution, including a mean (set to 0) and a standard deviation (typically 0.05, but you can also choose $\sqrt{1 \,/\, \text{nb\_input\_units}}$).

The standard activation functions are used: ReLu, sigmoid and softmax, but it's worth trying out alternatives such as leaky ReLu.

The loss function used is Binary Cross Entropy, because it is perfect for binary classification problems, but we could also have used Hinge. The groundtruth vector $\hat{y}$ does not contain the labels 'C1' and 'C2', but the probability that each pattern point belongs to class C_1, in this case 0 or 1.

The model is implemented in Python, using the basic modules: random, math, numpy and matplotlib. The basic model consists mainly of 4 functions: forward pass, backward pass (backpropagation of the gradient), optimize, and fit (also called train in the code).

Simulations

Most simulations are performed with 150,000 pattern points, i.e. 105,000 for the training phase (with a *validation_split* parameter of 0.2) and 45,000 for the test phase.

Single perceptron

The first model simulated is a simple perceptron with the ReLu function then sigmoid function as output. **Figure 3** shows the learning curves of the simulation. We observe that the model fails to learn perfectly and is also not very good at generalization.

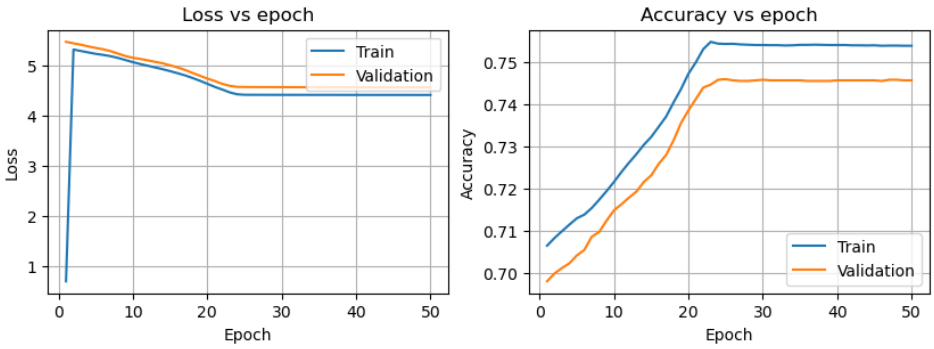


Figure 3. Cross-Entropy loss and accuracy vs epoch (simple perceptron, batch gradient descent, learning rate 0.05)

Figure 4 explains why it is unable to learn: it has only one neuron and can only attempt to solve the problem linearly. Yet, the accuracy on this test dataset is around 75%.

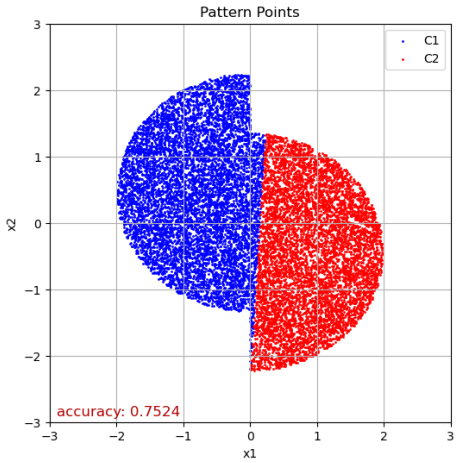


Figure 4. Classification performed by the first model on the test dataset

Single hidden layer

The next simulation therefore involves adding a hidden layer. **Figure 5** shows the learning curves and the classification on the test dataset performed by a 1-neuron multilayer perceptron. The hidden layer performs the ReLu function, and the second layer performs the sigmoid for the output.

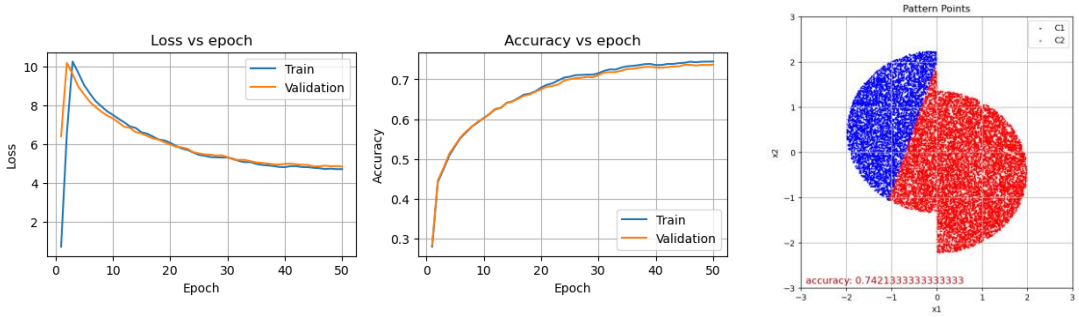


Figure 5. Learning curves and classification performed on the test dataset by the second model (2 1-neuron layers, batch gradient descent, learning rate 0.05)

The learning phase is still bad, and the position of the decision line differs a lot from one simulation to another; it often occurs that the model attributes every point to class C_1.

Now, let's try to add neurons in the hidden layer. Adding neurons implies increasing the complexity of the model, meaning that the model must learn more parameters (weights and biases), leading to a more flexible model that can capture more intricate patterns in the data. However, this increased complexity also makes the optimization landscape more complicated. This is why we adjusted the learning rate to get a better accuracy. The learning curves and classification result of the test dataset for a MLP with 50 neurons in the hidden layer are shown in **Figure 6**.

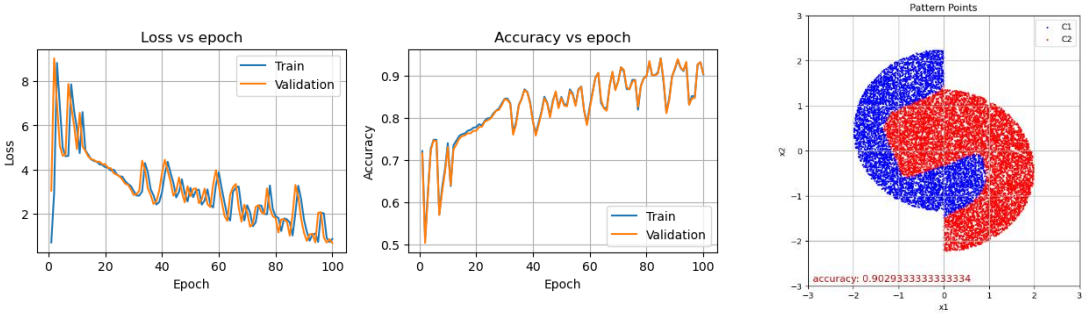


Figure 6. Learning curves and test dataset classification result (50-neuron hidden layer, batch gradient descent, learning rate 0.005)

A deeper analysis was conducted to evaluate the effect of the number of neurons in the single hidden layer over the performance of the model, especially its accuracy (**Figure 7**). Each simulation was performed 20 times to get an accurate accuracy average, with 50 epochs.

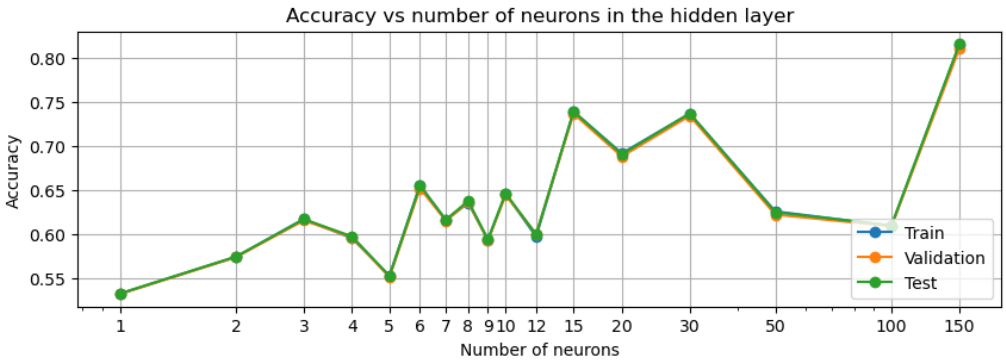
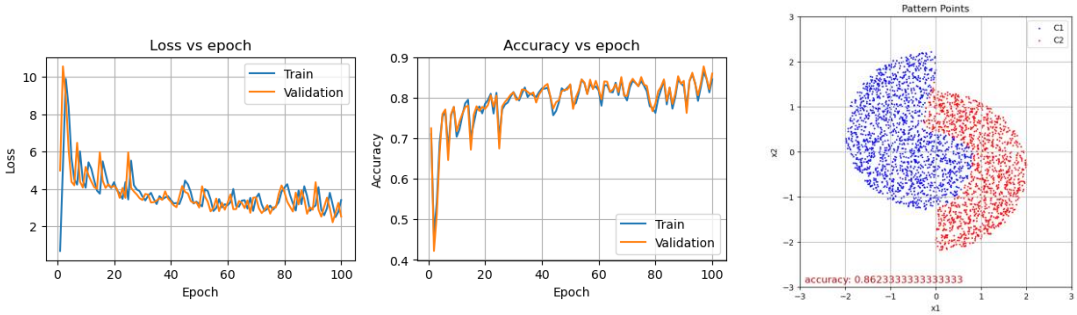
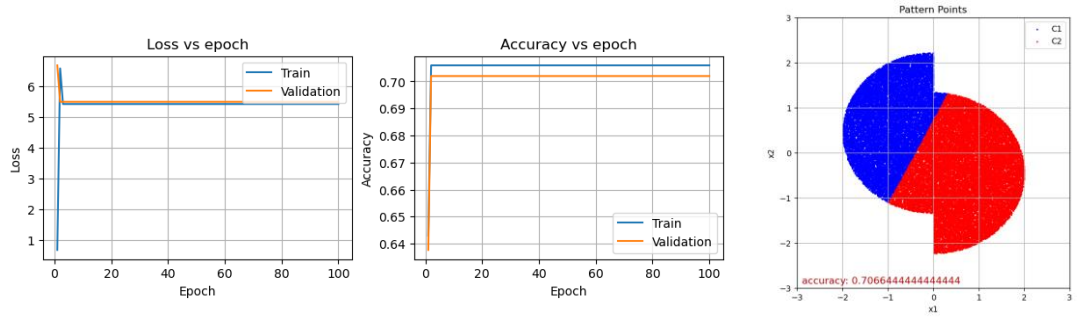


Figure 7. Accuracy vs number of neurons in the hidden layer (batch gradient descent, learning rate 0.005, 50 epochs)

Some numbers of neurons allow the model to perform better in terms of accuracy, especially when the number is large. However, this MLP is not performs badly, as the accuracy does not exceed 85% even with 150 neurons. **Figure 8** shows the learning curves and the classification of the model on the test dataset with 150 neurons in the single hidden layer, with respectively 5600 and 84,000 train inputs. The classification is not satisfactory in both cases.



(a) Results with 10,000 data inputs



(b) Results with 150,000 data inputs

Figure 8. Learning curves and test dataset classification result ([2, 150, 1], batch gradient descent, learning rate 0.05)

Deep neural networks

We then added a second hidden layer with different numbers of neurons to see if the model is able to learn with more complexity. In the report, the structure of the network is formatted as follows:

[2, 50, 20, 1] represents a network with 2 inputs, 2 hidden layers (respectively 50 and 20 neurons) and 1 output. The results of the simulation with this structure are shown in **Figure 9**.

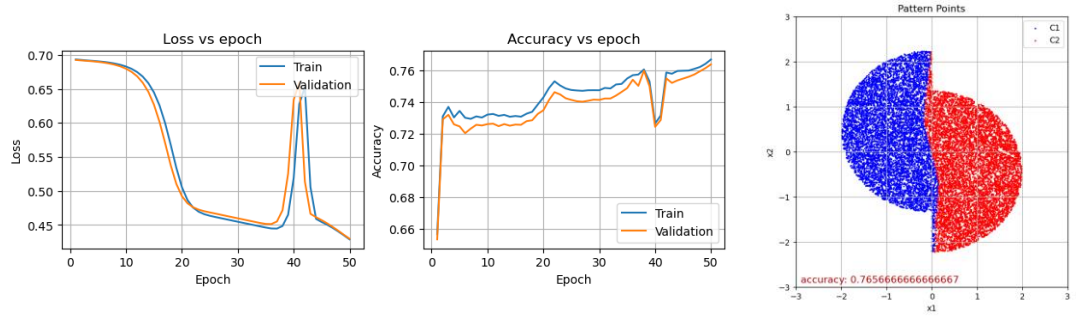


Figure 9. Learning curves and test dataset classification result ([2, 50, 20, 1], batch gradient descent, learning rate 0.00005)

As the accuracy of the neural network does not increase, we have improved it by implementing various common optimizations.

Mini-Batch Stochastic Gradient Descent (SGD)

The idea of this optimization is to perform the SGD on a small part of the training data instead of the whole training dataset. The principle is to divide the entire training dataset into smaller subsets (mini-batches) of a fixed size. For each mini-batch, the algorithm computes the gradient of the loss function.

Figure 10 shows a simulation of the model with mini-batch SGD, with batch size of 500 and a learning rate of 0.005.

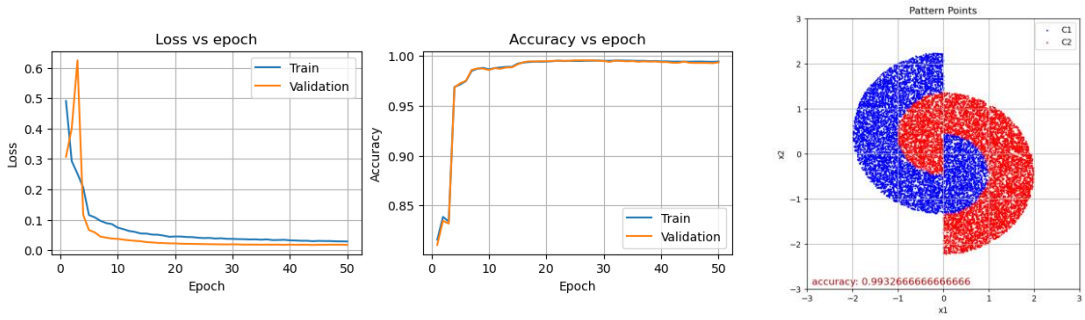
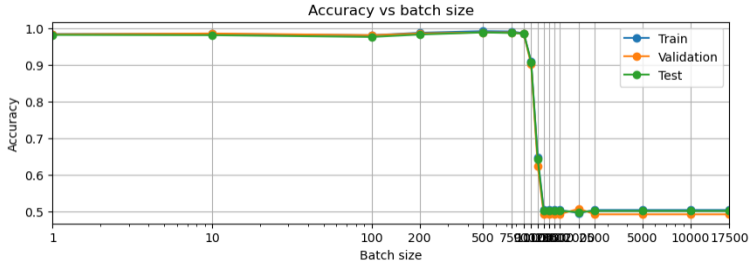


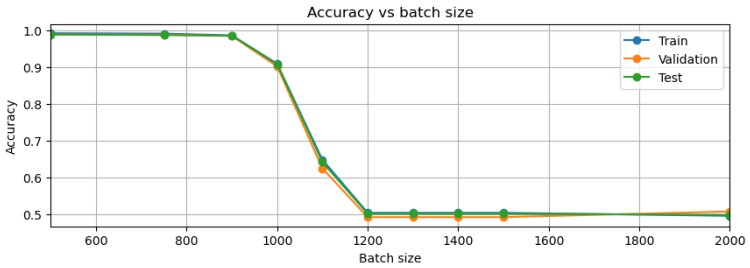
Figure 10. Learning curves and test dataset classification result ([2, 50, 25, 1], mini-batch gradient descent – size 500, learning rate 0.005)

In this example, the accuracy on the test dataset reaches 99.33%, which is excellent. However, the model can still be optimized. Indeed, we can notice some loss/accuracy spikes in the beginning of the learning phase, which can be corrected by changing the optimizer.

It is interesting to display the accuracies of the model when different batch sizes are used (**Figure 11**). The simulations were performed with 50 epochs.



(a) Batch sizes from 1 to the number of data – Logarithmic scale



(b) Batch sizes from 500 to 2000 – Linear scale

Figure 11. Train, Validation and Test Accuracy vs batch size

The accuracy of the model drops when batch size exceeds 1000. When the batch size is set to 1, the gradient is computed and the weights are updated after each training example: for 1 epoch, the weights are updated 17500 times (because the size of the training data is set to 17500 for this comparison). This practice is not efficient at all and is not robust because the optimization can take a hazardous path. Therefore, according to the simulation results, a good choice for batch size is between 500 and 800.

Adam optimizer

Up to now, the new parameters have been calculated using a fixed learning rate. To speed up convergence, we have implemented one of many optimizers: Adam, which stands for Adaptive Moment Estimation. Adam optimizer includes adaptive learning rates, incorporates a momentum and bias correction features. These improvements help to converge faster and more efficiently to the optimal solution, especially for problems with noisy or sparse gradients (which is not the case of the problem here). It introduces two additional hyperparameters β_1 and β_2, but it is more robust and requires less fine-tuning than SGD with a fixed learning rate. The momentum is particularly interesting for our problem because it helps to accelerate convergence in the relevant direction and dampens oscillations.

Figure 12 shows the results of the simulation using the exact same parameters as the previous simulation with the use of Adam optimizer. The learning curves are smoother, and a tuning of the learning rate, β_1 and β_2 would lead to an even faster convergence.

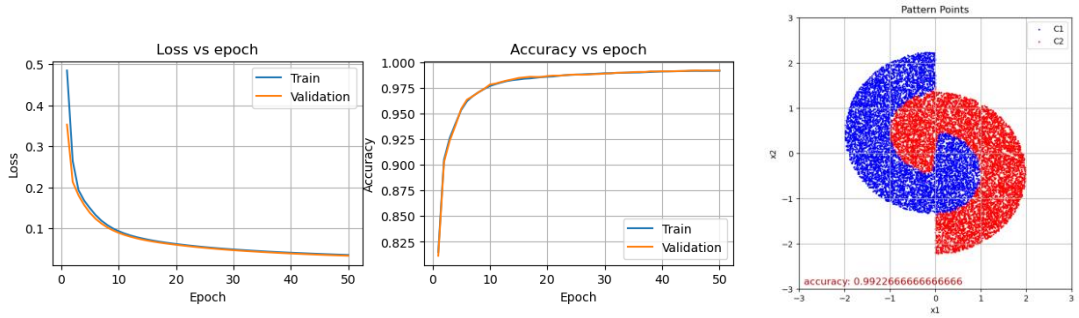


Figure 12. Learning curves and test dataset classification result ([2, 50, 25, 1], mini-batch gradient descent – size 500, Adam, learning rate 0.005)

The output probability of this network is shown in **Figure 13**. It allows us to see that the model is completely uncertain on the decision region between the two classes, especially near the angles.

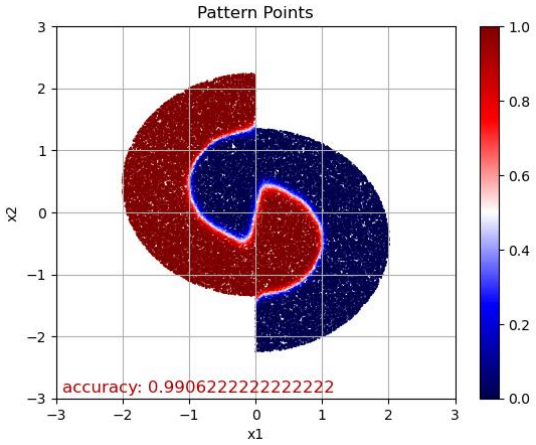


Figure 13. Output probability of belonging to class C_1 ([2, 50, 25, 1], mini-batch gradient descent – size 500, Adam, learning rate 0.001)

Learning rate tuning

We varied the learning rate with the standard optimizer, as Adam is more robust to changes in learning rate. **Figure 14** shows how essential the choice of this hyperparameter is to the model's performance. The simulation was conducted by computing the average of 5 simulations per learning rate value.

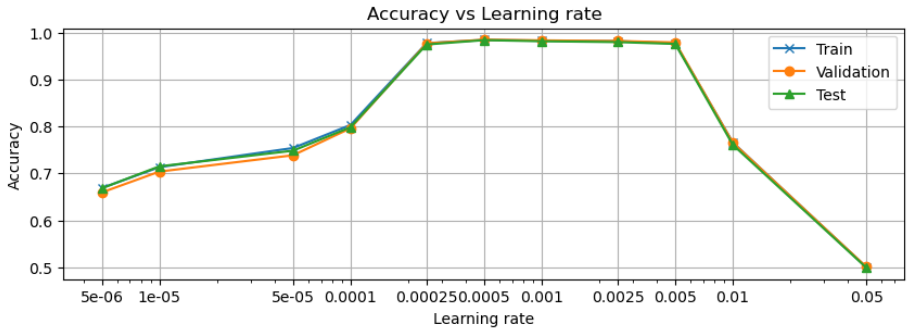


Figure 14. Accuracy versus learning rate (([2, 50, 25, 1], mini-batch gradient descent – size 500, 50 epochs)

The learning rate must be large enough to allow sufficient gradient descent and convergence, and to avoid falling on a local minimum, but not too large to ensure convergence. With the example of the figure above, the best learning rate is between 0.00025 and 0.005.

To assess the convergence speed, we simulated the model with a stop condition when the accuracy exceeded 97%, with a maximum number of epochs of 50. **Figure 15** shows the convergence speed versus the learning rate, to get the learning rate that leads with the fastest convergence. The simulations are averaged over 5 runs, and don't include Adam optimizer that increases the convergence.

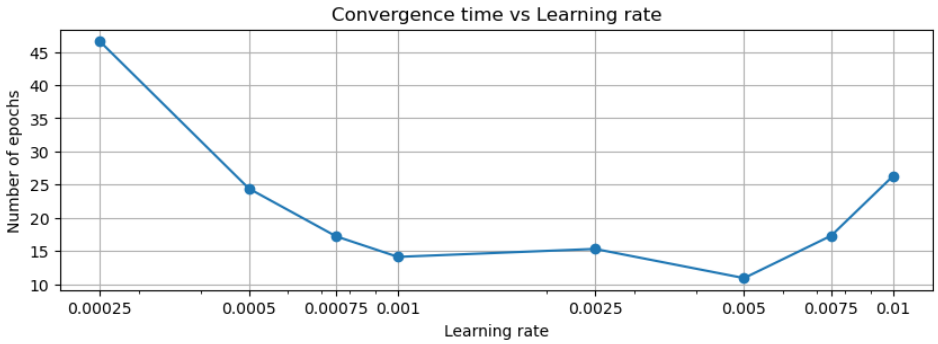


Figure 15. Convergence of the model versus learning rate ([2, 50, 25, 1], mini-batch SGD – size 500, max 50 epochs, accuracy stop 97%)

We can conclude that a smaller learning rate does not lead to a faster convergence, especially because it requires more epochs to descend the gradient. On the contrary, a larger learning rate leads to no convergence. The perfect learning rate here is 0.001, that leads to a good accuracy in 15 epochs.

Our best MLP

We found satisfactory values for hyperparameters to get a good performance. The model uses Adam optimizer, mini-batch SGD with batch size of 500, a learning rate of 0.005. The learning curves and the test dataset classification are shown in **Figure 16**. The accuracy of the test dataset is around 99.5%.

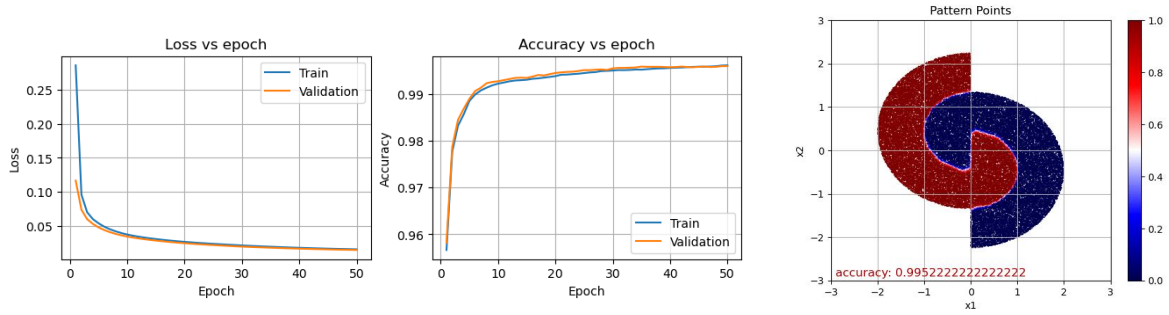


Figure 16. Learning curves and test dataset classification result ([2, 50, 25, 1], mini-batch gradient descent – size 500, Adam, learning rate 0.005)

We can eventually evaluate the model with a confusion matrix that shows the true positives, true negatives, false positives, and false negatives resulting from the prediction (**Figure 17**). The confusion matrix helps in identifying not just the overall accuracy but also specific areas of strength and weakness, such as sensitivity and specificity. However, our classification problem contains only two classes, so it is unlikely that the model provides incorrect predictions across a specific class.

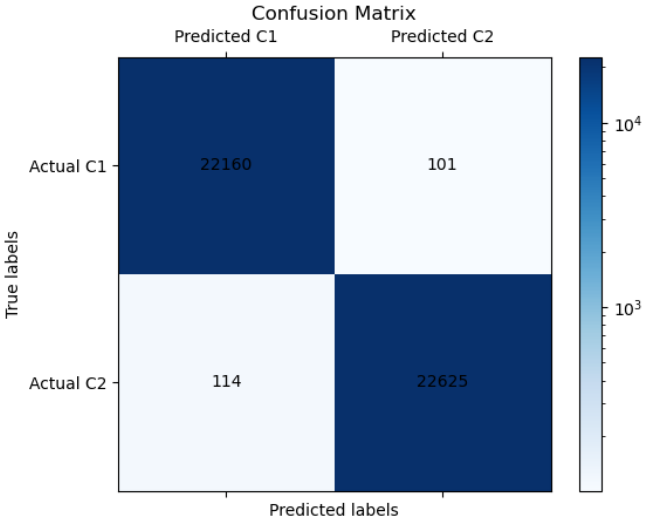


Figure 17. Confusion matrix of the last model

Discussions

In addition to numerous remarks on the various hyperparameters, a discussion on what has not been observed or achieved is necessary. Indeed, we did not observe any model overfitting, i.e. the model did not learn the data “by heart”. This is due to the quality of the input data, which is uniformly distributed, normalized and perfectly labeled. There is no noise in the data, no label errors, and the train dataset is very similar to the test dataset. Besides, we can generate an endless number of datasets. Thus, it is easy for the model to "generalize", and the inference stage runs smoothly. This is often not the case with complex problems.

Conclusion

This problem enabled us to familiarize ourselves with the basic concepts of deep learning, and to study the various aspects of neural networks. It also gave us the opportunity to code a neural network from scratch, in order to understand exactly how it works and the mathematical relationships that drive it.

In particular, we have seen that a single-layer perceptron is unable to solve a nonlinear classification problem, and that a deep neural network needs to be developed. In my opinion, three aspects are important when designing a model to solve a problem: the quality of the data, the choice of layers and number of neurons, and the choice of certain hyperparameters such as the learning rate.