TOML - Projects 3 and 4

Oriol Martínez Acón

June 2022

1 Project 4: Neural Networks

1.1 Introduction

The objective of this project is to test, to understand, how a 3 layer Feedforward Neural Network (FNN) performing binary classification works. The way to test with the NN will be by changing the following parameters:

- **T**: Batch size of the training set and test set. Is generated by normal distribution.
- **H**: Number of hidden dimensions in the Neural Network.
- **Learning rate** (η): the value use in the gradient descent, it sets the amount of speed to move in a direction (gradient, ∇), to find the minimum of the function.
- **Momentum** (α): The amount of inertia speed conserved in the previous gradient descent step.

The current neural network is doing two different steps to create a model by the training data:

1. **Forward propagation**: In this step the neural network is calculating the results by the inputs (x), initial randomized weights for each neuron (w) and their respective active functions (a), i.e. moving from the input layer (left) to the Output layer (right). Once is done, the result from the output layer is taken to compare with the real value to see which is the error.
2. **Back propagation**: In this step is doing the computation of the gradients starting from the last layer to the first one. Is important to highlight, that this step is very high in terms of computational cost because you have to perform the gradient for each of the parameters in the neural network, further, each gradient calculated at a current layer is necessary for the following layer, what it translates into a sequential computation model.

The active functions play a very important role in the neural networks. They are responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. These active functions can be either linear or nonlinear. Linear active functions are not applying any transformation. A network that only applies linear activation functions is very easy to train but can not learn complex mapping function what it can translates into underfitting issues. On the other hand, nonlinear activation functions are preferred because give more complex structures in the data.

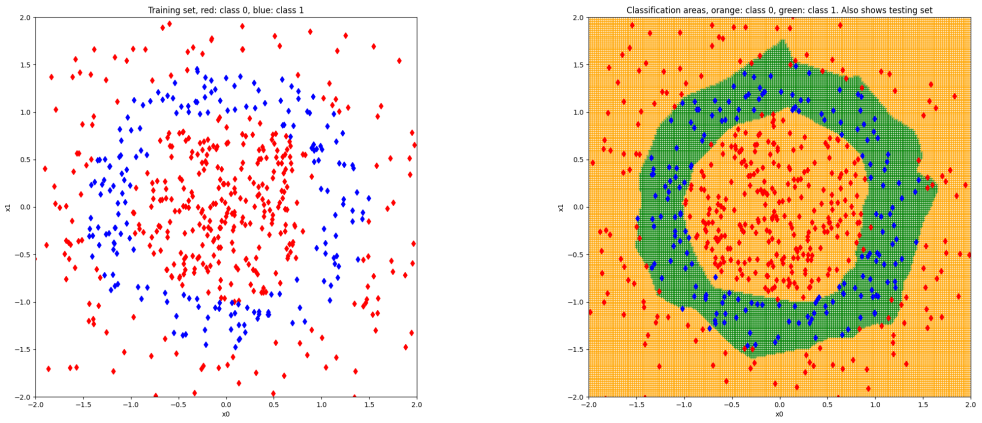
In this project the activation functions used are the nonlinear **sigmoid** active function for the y prediction ($\hat{y}$) and **Rectified Linear activation Unit** as the active functions in each of the hidden layers. Is important to remark that even it looks and acts like a linear function, is in fact, a nonlinear function that allows complex relationships in the data to be learned.
All the code used in this project can be found in the following GitHub repository: `https://github.com/oriolmartinezac/TOML-Labs/tree/main/project-4`

1.2 Initial scenario

In this section we will see the results of executing the neural network without changing any of the parameters given to make the respective comparison with respect the other scenarios.

1.2.1 Results obtained

In the following figures we can see the model obtained by the training data with the values of: $T = 640$ (batch size), $H = 20$ (hidden dimension), $\eta = 1e - 4$ (learning rate) and $\alpha = 0.7$ (momentum).



(a) Training data points.

(b) Model in iteration 2000 over the test data.

Figure 1: Initial scenario.

In the figure 1 we can see that the model generated by the training set classifies very good the training data (low training loss) and it classifies good in comparison with the test data (low test loss). The model seems to be good-fitting (low bias and low variance) with the data given.
To verify that the model is good-fitting let us take a look to the training loss function and the test loss function plot.

Figure 2: Plot of the training loss and test loss with the parameters specified.

As we can see in the figure 2 the model is good-fitting due to the low distance between both loss functions and the low value of each of them at the iteration 2000, training loss function = 0.08855 and the test loss function = 0.12581. It is important to highlight that if we keep iterating over the model there is a chance to reach an overfitting scenario.

1.2.2 Conclusions

The main goal of ML is to create a model that does good-fitting by a given data. There is no recipe-book that tells you exactly how to do it, the best way is by testing and changing the parameters needed to create our model.
In this first scenario we have seen a good-fitting but we are going to discuss about the underfitting scenario and overfitting scenario in neural networks in the following sections.

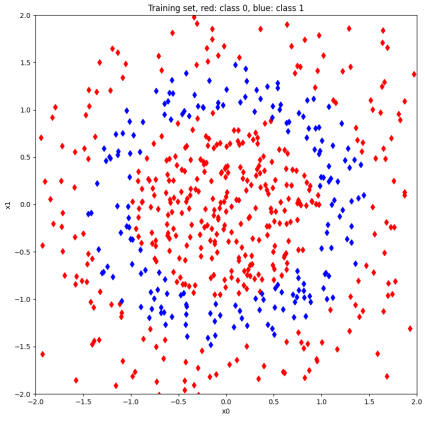
1.3 Underfitting scenario

When we say that a model is underfitting in Machine Learning/Deep Learning. We mean that the model created by the training data is getting really far from the true-relationship. In other words the model is not complex enough to depicts the behaviour of the data given and furthermore, it can not give a good predictions with any data given, i.e. the model has a lot of bias but low variance.

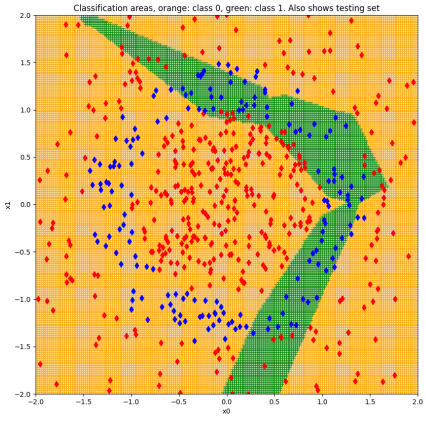
Having in mind this definition, in this section the neural network parameters like the number of hidden dimensions will be decreased to see how that affects to the given results without changing other of the parameters given.

1.3.1 Results obtained

In the following figures we can see the model resulted by the training data applying these values $T = 640$ (batch size), $H = 8$ (hidden dimension), $\eta = 1e - 4$ (learning rate) and $\alpha = 0.7$, and how good it is at predicting with the test set.



(a) Training data points.



(b) Model in iteration 2000 over the test data.

Figure 3: Underfitting scenario with $H = 8$.

In the figure 3 we can see how much is the model underfitting the data, neither fitting training set nor test set. The classified points as orange class includes a lot of red and blue points; in addition, the green class is classifying just a few of the blue points in one class and taking some other red points.

To verify that we are underfitting let us take a look to the plot of the loss functions.

Figure 4: Plot of the training loss and test loss with the parameters specified.

The results got from the training loss function in the last iteration of the model 0.34247, on the other hand, the result got from the test loss function is 0.38447. Taking this and the figure 4 we can see that the values obtained are big to say that our model is fitting correctly.

1.3.2 Conclusions

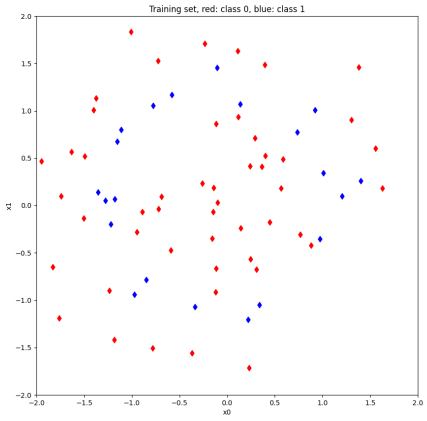
If we modify the decrease the number of hidden dimensions, with a batch size fixed of 640, we are decreasing the complexity of our neural network model. In other words, the model will be underfitting and will not be able to find the true-relationship to classify the data given. The way to solve that problem is by increasing the complexity, i.e. increase the size of the hidden layer (H size).

1.4 Overfitting scenario

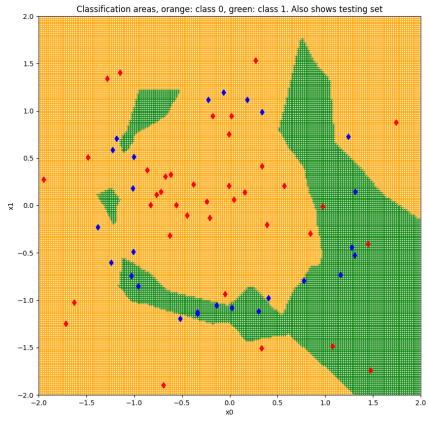
When we say that a model is overftting in ML/DL. We mean that the model created by the training data is predicting perfectly all the points given without any error. Because the model is too much complex in comparison to the amount of data given. Even it seems that is something good, if we take new data to make the predictions we will see that the model overfitting does not perform any good with the new data, the model only learns how to predict one specific data set.
Having in mind this definition, in this section the neural network parameters like the batch size will be decreased to see how that affects to the given results.

1.4.1 Results obtained

In the following figures we can see the model resulted by the training data applying these values $T = 75$ (batch size), $H = 20$ (hidden dimension), $\eta = 1e - 4$ (learning rate) and $\alpha = 0.7$, and how good it is at predicting with the test set.



(a) Training data points.



(b) Model in iteration 2000 over the test data.

Figure 5: Overfitting scenario with $T = 75$.

As we can see in the figure 5 the model created is predicting correctly all the training data points (green shape on the left plot) but if we take different data from the training set, it is misclassifying a lot of the red and blue points. I.e. the model has a low bias but big variance. Our model will be only good predicting a specific data set, and we do not want that.

In order to verify this we are going to take a look to the training loss function and test loss function.

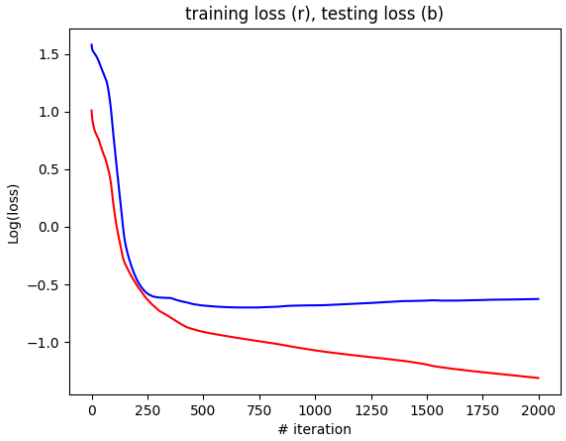


Figure 6: Plot of the training loss and test loss with the parameters specified.

The figure 6 depicts how the model is overffiting. The training loss function is really small because the model generated suits really good with the training data given but, if we give new points to classify to the model (test data) we can see that its loss function (test loss function) is giving worse results due to the differences from the training data which the model fits perfectly. The training loss seems to be decreasing (keeps learning) if we do more iterations but the test loss, on the other hand, increase when we reach approximately the iteration 500, this confirm what it has been said.

1.4.2 Conclusions

If we decrease the number of T without changing the complexity, H, of our neural network we are leading to an overfitting scenario.

To solve this we can try to:

- Increase the amount of data.
- Decrease the complexity of the model.
- Early stop. Stop the learning step of the neural network before reaching the overfitting case.
- Regularization. Using dropouts to drop randomly some of the neurons and then decrease the complexity of the NN.

1.5 Finding the right values of H

To find the right value of H we need to take into account the other parameters. As we have fixed the value of the $T = 640$ we have to create a model with more size of H than the underfitting scenario but without exceeding too much, because we can lead into an overfitting scenario.

1.5.1 Results obtained

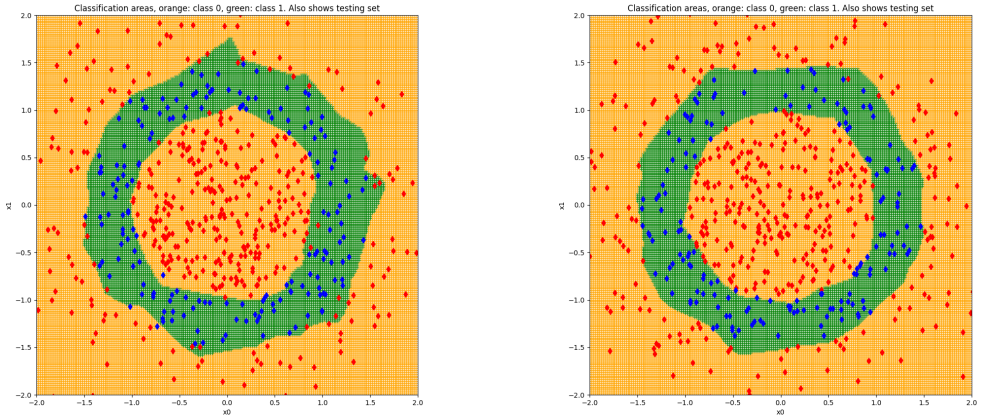
If we decrease the **learning rate**, each time the weights are just only slightly changing their values, what it translates into less correction on both loss functions and then more computational time (more iterations) to create the training model. Having a smaller learning rate in the traditional gradient decent also can lead us into a local minima without finding the global minimum of the objective function, if we do not have enough momentum, but it converges more precisely to the local/global minimum value, and then reach to an **underfitting scenario**.
On the other hand, if we increase the learning rate we are changing the weights of the neural network a lot, what it means that maybe we can not reach to the optimal point of the minimization function. We can reach to an **overfitting scenario**.

The best value obtained by executing the program with different learning rates has been of $\eta = 1e{-}4$.

The momentum in the gradient descent is a way to overcome the oscillations of noisy gradient and coast across flat spots of the search space. By changing the value of α we are giving more or less importance to the inertia movement speed of the gradient descent. A value close to 1 seems to give good results.

The best value obtained by executing the program with different momentums has been of $\alpha = 0.9$.

It seems that the best value obtained by changing the H is 20. In the following figure we will compare the initial scenario with our best model.

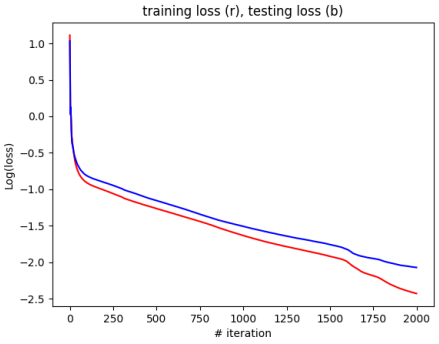


(a) Initial model in iteration 2000 over the test data points.

(b) Best model in iteration 2000 over the test data.

Figure 7: Initial scenario vs best scenario.

In the figure 7 we slightly see the differences of which one performs better (taking into account the randomness). So let us take a look to the loss function.



(a) Loss functions of initial scenario.

(b) Loss functions of best scenario.

Figure 8: Initial scenario vs best scenario.

In the figure 8 we can see that the results obtained are very similar but we have less training loss and test loss error in the last iteration of the model, instead of training loss error equal 0.0855 and test loss error equal 0.12581 we have 0.6111 and 0.9189 instead respectively.

1.6 Problems with the gradient

If we increase the number of H, the complexity of the model increases. Even what it tells us the intuition the model does not overfits. In the image below we can see how are the values of the training loss functions and test lost functions with $T = 640$, $\alpha = 0.7$, $H = 100$ and $\eta = 1e - 4$.

Figure 9: Plot of the training loss and test loss with the parameters specified.

The figure 9 shows how in the first iteration the training loss function is improving in some way, but after it remains forever with the same value. This means that the model can not learn anything more from the data given, more technically, the gradients calculated in each back-propagation step are equal to 0 what it translates to a non updated weights. This could happen because there are some

dead neurons due to the ReLu function or because the function to minimize is in a high dimensional space, due to the high complexity, and the point found as a minimum is a **saddle point**. In the saddle point, the majority of gradients are pointing to one point that is a local minimum that is also already a maximum, so for the model is really hard to escape from that specific point and it thinks that it has achieved the best solution. Is important to highlight that to fall into a saddle point also depends on the randomness of initial weights.

In the figure 10 we can see a representation of the saddle point problem.

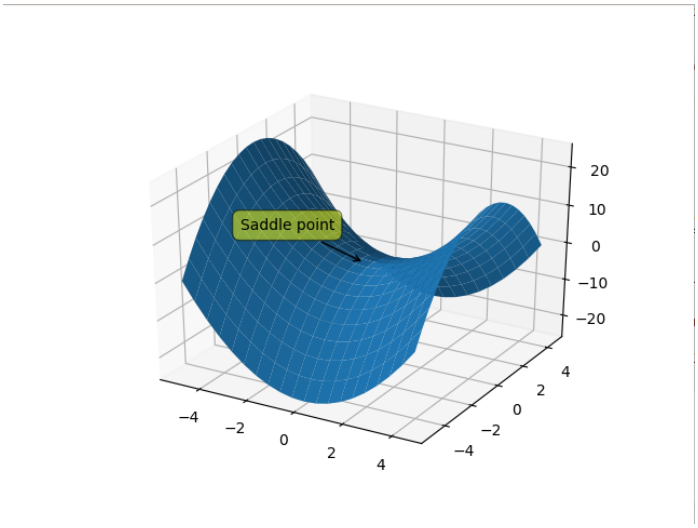


Figure 10: Graphic representation of the saddle point problem in a three dimensional space.

1.6.1 Conclusions

If we increase too much the complexity of the neural network the model does not reach a good minimum and it underfits, that happens because the gradients at each back-propagation calculation are equal to 0. In order to verify if that problem was because of the dead neurons the ReLu active functions were changed to **Leaky ReLu** and **Parametrized ReLu** functions without any improvement. By discard the last option is the saddle point.

If it is a simple saddle point **Stochastic Gradient Descent** (SGD) can help to break out it, the fluctuations are along other directions and with a step size large enough to go over the flatness. In other case we can use Taylor's expansion:

$$f(y) \approx f(x) + <\nabla f(x), y - x> + \frac{1}{2}(y - x)^T \nabla^2 f(x)(y - x)$$

1.7 Implementation of Stochastic Gradient Descent

In this section we are going to implement **Stochastic Gradient Descent** (SGD) in our Neural Network. Instead of taking all the data set into account, like in the traditional Gradient Descent, is taking to account only a subset selected randomly of the data (estimate thereof).

This new approach can be very useful in high-dimensional optimization problems because it reduce the high computational burden, in other words, it tries to search in a faster fashion (lazy fashion) the minimum by doing the gradients. This translates into less time to reach for a minimum and then we can increase the number of iterations in order to get better results.

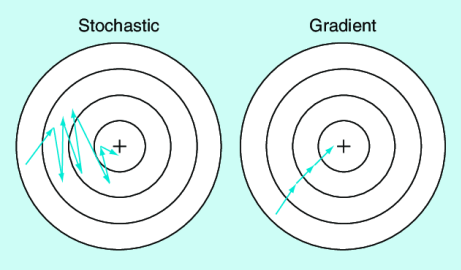


Figure 11: Graphic representation (2D) of stochastic gradient descent (left) against gradient descent (right).

To implement this part it has been used the `PyTorch` python package, and the file executed `network.py`. Is important to highlight that the number of iterations has been increased because in other case the neural network would finish before getting to a good value.

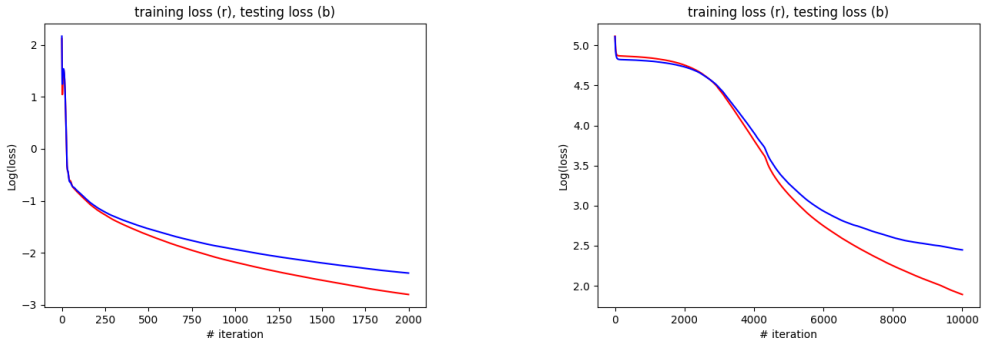
In the following figure we can see the plot of the loss functions using a $H = 150$ and using $ITER = 40000$ without changing the other parameters of the 1.6 scenario.

Figure 12: Loss functions with $H = 150$

As we can see in the 12, even having high complexity if we use SGD seems that our problem has been solved, we are not falling into a saddle point. Is important to remind that in order to verify this behaviour some models were created.

Finally, for the creation of the best new model, the parameters used have been: $\eta = 1e-4$, $H = 17$, $T = 640$, $\alpha = 0.4$ and $ITER = 10000$.

In the following figure we can see a comparison of the loss functions between the previous best model with the current new model.



(a) Loss functions of previous best scenario.

(b) Loss functions of the SGD model.

Figure 13: Loss functions from previous best scenario vs SGD model.

In the figure 13 we can see that our new model is not more accurate than the model of the previous scenario, last training loss = 5.43039 and last test loss = 15.6741.