Backpropagation in neural networks

Authors:

Guilherme Pereira 202200945

Karolina Mierzwa 202201435

Leonardo Andrade 201700555

1. Description of the problem

Backpropagation is an algorithm used in training neural networks (NN) to calculate the gradient of a loss function with respect to the weights of the network. This gradient founded is then used to update the weights in a way that reduces the loss and improves the performance of the neural network. Backpropagation is used in supervised learning, which is a type of machine learning where the model is trained using labelled examples.

In other words, the desired outputs are the observed label values on each row of the train set, this is what we want to predict so if the observed value on the sample and the predicted value are equal (or approximated) it means that our NN output is good. It is important to understand why backpropagation is used. There are many challenges that can arise when training artificial neural networks, some of them are discussed below:

- Overfitting: This occurs when a model performs well on the training data but poorly on new, unseen data. This can happen if the model is too complex and the learning was based on random noise in the training data, rather than the underlying patterns.
- Underfitting: This occurs when a model performs poorly on both the training and test data. This can happen if the model is not complex enough to capture the underlying patterns in the data, or if it has not been trained for long enough.
- Local minima: When training a neural network, the goal is to find the set of weights that minimizes the loss function. However, the loss function can have many local minima, where the model gets stuck and is unable to find a better set of weights. This can prevent the model from achieving good performance.
- Vanishing gradients: When training deep learning models with many layers, the gradients of the loss function with respect to the weights can become very small, making it difficult for the model to learn. This is known as the vanishing gradient problem.
- Slow training: Training large neural networks can be computationally intensive and can take a long time, especially if the data is large or the model is complex. This can make it difficult and expensive to explore different NN architectures and hyperparameters.
- Hardware limitations: Training large neural networks can require a lot of computational power, which can be limited by the hardware available. This can make it difficult to train very large models, or to train them in a reasonable amount of time.

The goal of training a neural network is to find the set of weights that minimizes the difference between the predicted outputs and the true labels, which is called the loss. While backpropagation is a way to optimize the weights of a neural network, it is not itself an optimization problem, rather, it is a way to solve the optimization problem of finding the weights that will minimize the loss function.

Providing a way to calculate the gradient of the loss function with respect to the weights of the network, backpropagation allows the use of optimization algorithms to find the best possible values for the weights, this can help to deal with some problems:

The “Vanishing gradients” preventing the gradients from becoming too small and can make it easier for the model to learn, the “Overfitting” and the “Underfitting” improving the performance on the training data and can make it more likely to generalize well to new data, also the “Slow training” and “Hardware limitations” helping the model to learn more quickly and converge on a good set of weights in less time, saving computational resources.

2. Description of the algorithm

The algorithm implemented is an Artificial Neural Network with Backpropagation. The Artificial Neural Network (ANN) is inspired by the human brain and is good for recognizing patterns. The Backpropagation function is to optimize the ANN by reducing its error by adjusting the “parameters” of the algorithm as the model is trained.

First to understand the algorithm with all its complexities, its necessary to understand its most basic fundament, the Artificial Neuron. The Artificial Neuron is a function, that given its parameters (the weights), takes any number of inputs, and gives an output, usually 1 or 0. It can be mathematically described as below:

$$y(x) = g\,(\sum_{i=0}^{n} w_i x_i)$$

Where x is a neuron (node), that takes n inputs $(x_0, \ldots, x_n)$, and consequentially gives an output $y(x)$, where the weights $(w_{0\ldots}\, w_n)$ determine the weight that each input will have.

g is the activation function, that based on the inputs and the weights, returns an output. In a real neuron the output is 1 or 0 (trigger or not). But in this project Artificial Neurons, a smother function was used, with the output between 0 and 1.

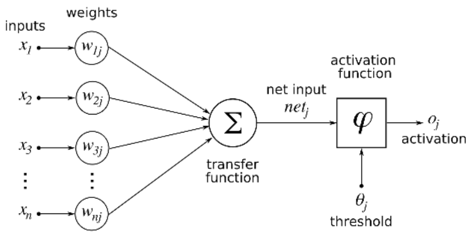


Figure 1: An Artificial Neuron

The activation function used was a sigmoid function: $g(x) = \frac{1}{1+e^{x}}, with\ x = \sum_{i=0}^{n} w_i x_i$, that returns a result between 0 and 1.

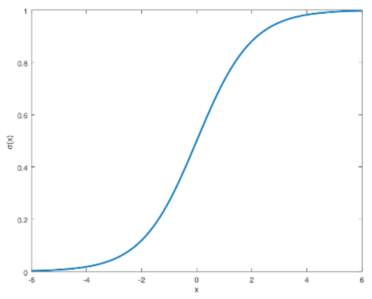


Figure 2 Sigmoid Function

After understanding the Artificial Neuron, is possible to understand the Artificial Neural Network. The algorithm used a multi-layered ANN, with the neurons ordered in layers, with connections between neurons from different layers going forward. To be more specific there were three layers: the input layer, the hidden layer, and the output layer, in that order. In this function, the input is presented and is propagated through the layers until it gives an output.

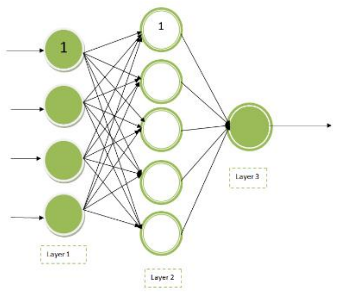


Figure 3: 3 Layers ANN

During the training phase of the model the weights are adjusted in order to reduce the error, i.e., the misclassification of the predict value in relation the desired correct value. In this process an optimization problem is executed to minimize the mean square error of the training set. That is when the backpropagation comes into play. The weights are usually initiated with small random numbers, and the input is propagated through the network until it comes to the output/prediction. After this, the error ε_i of a neuron i is calculated with the difference of the predicted value and the correct value. Then ε_i is used to calculate $\delta_k = \varepsilon_k y_k(1 - y_k)$, which is therefore used to calculate the δ_j of a previous layer with $\delta_j = \sigma\, y_j\left(1 - y_j\right) \sum_{k=0}^{K} \delta_k\, w_{jk}$, where K is the number of neurons in this layer and σ being the learning rate, which determines how much a weight will be adjusted. Using the δ values the weights are adjusted by summing $\Delta w_{jk}\ to\ w_{jk}$, with $\Delta w_{ik} = \delta_j y_k$, while the backpropagation algorithm moves to the next input and adjust the weights according to outputs. This goes until the square error reaches its minimum limit.

3. Implemented functions

To implement the algorithm described in the 2nd chapter, there were 2 main and 3 auxiliary functions proposed. Their parameters are summarized in the table below.

Main functions

BPcreate	• **V**: number of neurons in the hidden layer • **TrainSet**: a Pandas DataFrame with the training data, where the last column is the label and the rest of the columns are the features • **learning_rate**: the learning rate for the optimization algorithm (default value is 0.1) • **epochs**: number of epochs to train the model (default value is 1 000)
BPpredict	• **NN**: a list with the weights of the connections between the input layer and the hidden layer, and the weights of the connections between the hidden layer and the output layer, as returned by the BPcreate function • **TestSet**: a Pandas DataFrame with the test data (features only)

Auxiliary functions

sigmoid	• **X**: numpy array
mean_squared_error	• **Predictions:** array of predicted classification by neural network

	• **Labels:** array of true classification values
accuracy	• **Predictions:** array of predicted classification by neural network • **Labels:** array of true classification values

Table 1 Implemented functions with parameters' description.

BPcreate function is the one that with the use of backpropagation creates neural network with single hidden layer with given number of neurons (V) based on the given training set. The function returns a list with two elements: the weights of the connections between the input layer and the hidden layer, and the weights of the connections between the hidden layer and the output layer. The function also plots the mean squared error and accuracy of the model on the training set as a function of the epoch.

BPpredict is a function that takes a trained neural network obtained from BPcreate and a test set, and returns the predictions made by the neural network on the test set. The function returns a matrix with the predictions of the neural network on the test set. The rows of the matrix correspond to the samples in the test set, and the columns correspond to the different output classes. The values in the matrix are the probabilities predicted by the neural network for each class for each sample.

Sigmoid function applies the sigmoid transformation elementwise to a numpy array. The sigmoid function is defined as:

$$f(x) = \frac{1}{1 + e^x}$$

Sigmoid is often used as the activation function in the output layer of a neural network, because it maps the output of the network to a probability between 0 and 1.

The mean_squared_error function calculates the mean squared error between two numpy arrays. The mean squared error is defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(h_i - y_i)^2$$

where:

n – number of predictions

h_i – predictions

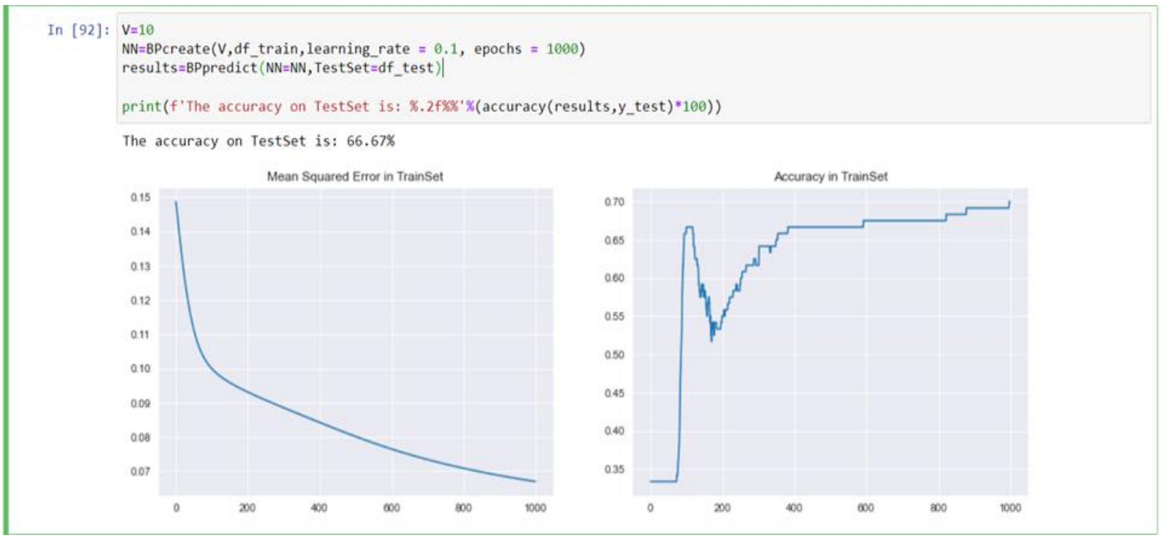
y_i – true labels.

The accuracy function calculates the accuracy of a set of predictions, given the true labels. It does this by comparing the index of the maximum value in each row of the predictions array with the index of the maximum value in each row of the labels array and taking the mean of the resulting boolean array. The resulting value is the fraction of correct predictions.

4. Small examples

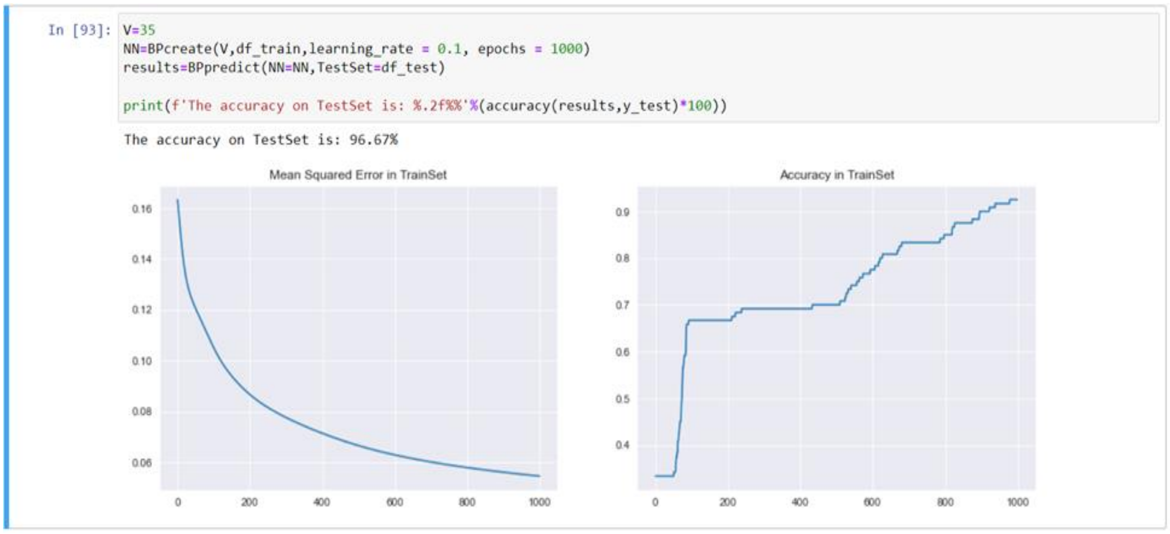
We decided to test our functions with some datasets that we have available on Data Mining I course to see how it runs on different datasets, first we will start with simple datasets then continue to a complex dataset and see if the Neural Network generated is able to learn how to predict the class with more complexity.

To do these tests, we used the "*train_test_split*" from "*sklearn.model_selection*" to split the dataset in "TrainSet" and "TestSet" also the "y_test" in order to measure the accuracy (capacity to learn) and further necessity of grow our network or do more pre-processing tasks as scaling the features and/or deal with missing values if we are not satisfied with the learnability of NN. First the **Iris Classification,** which is a classic dataset on data science community:



```
In [92]: V=10
         NN=BPcreate(V,df_train,learning_rate = 0.1, epochs = 1000)
         results=BPpredict(NN=NN,TestSet=df_test)

         print(f'The accuracy on TestSet is: %.2f%%'%(accuracy(results,y_test)*100))

         The accuracy on TestSet is: 66.67%
```

Figure 4: Test with Iris dataset and V=10.

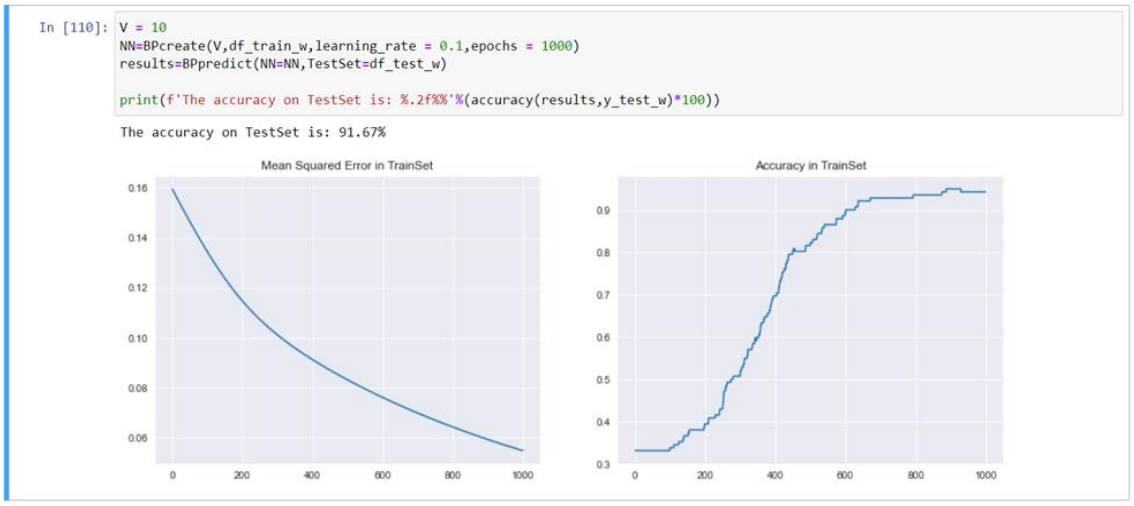


```
In [93]: V=35
         NN=BPcreate(V,df_train,learning_rate = 0.1, epochs = 1000)
         results=BPpredict(NN=NN,TestSet=df_test)

         print(f'The accuracy on TestSet is: %.2f%%'%(accuracy(results,y_test)*100))

         The accuracy on TestSet is: 96.67%
```

Figure 5: Test with Iris dataset and V=35.

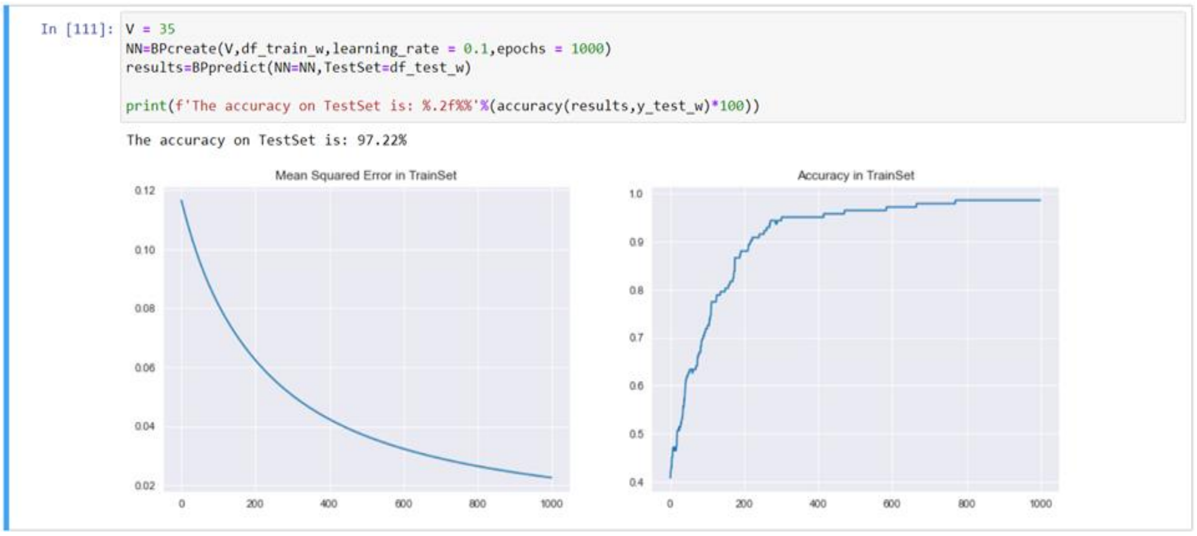
Clearly the functions works and the NN can learn how to classify each type of Iris, we also can see the necessity to increase the V (number of nodes on each layer) to the network could classify more accurate each flower on "TestSet". Trying other values on the parameters "*learning_rate*" and "*epochs*", the accuracy on "TestSet" reaches 100% which means a perfect prediction.

Secondly, the **Wine Classification** dataset, which is also a classic on data science community, in this case our NN requires less nodes to be more highly accurate on predictions:



```
In [110]: V = 10
          NN=BPcreate(V,df_train_w,learning_rate = 0.1,epochs = 1000)
          results=BPpredict(NN=NN,TestSet=df_test_w)

          print(f'The accuracy on TestSet is: %.2f%%'%(accuracy(results,y_test_w)*100))

          The accuracy on TestSet is: 91.67%
```

Figure 6: Test with Wine dataset and V=10.



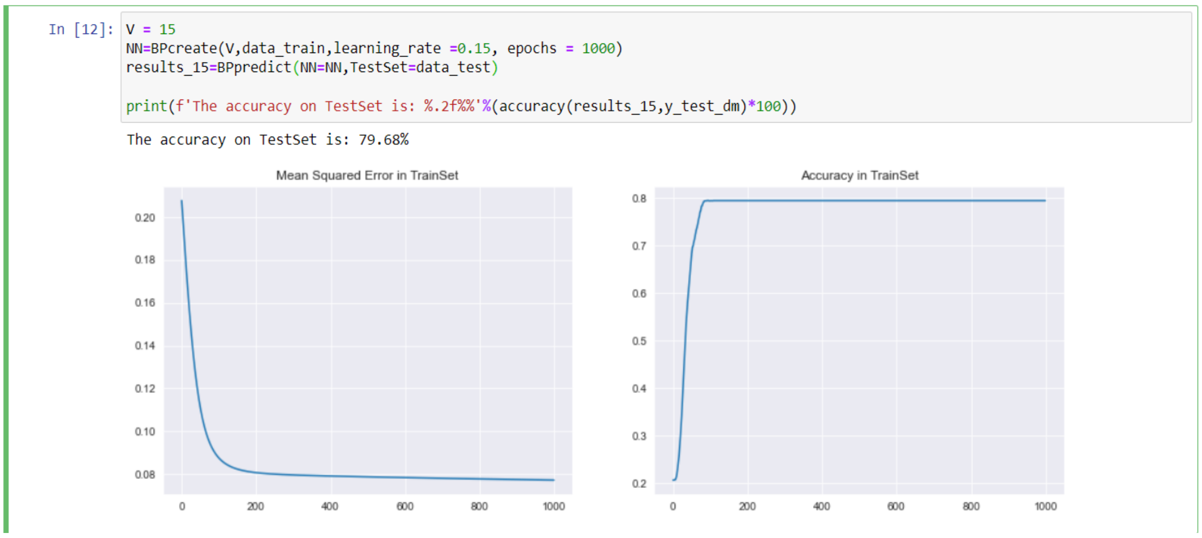
```
In [111]: V = 35
          NN=BPcreate(V,df_train_w,learning_rate = 0.1,epochs = 1000)
          results=BPpredict(NN=NN,TestSet=df_test_w)

          print(f'The accuracy on TestSet is: %.2f%%'%(accuracy(results,y_test_w)*100))

          The accuracy on TestSet is: 97.22%
```

Figure 7: Test with Wine dataset and V=35.

5. Larger example

Considering that is clear that the NN generated and trained by our functions can learn how to predict classifications problems on simpler datasets, we decided to try that on a complex dataset that was provided on Data Mining I course, which is data from *Banco de Portugal* containing information on a range of products in different establishments. Occasionally these products are sold on sale.

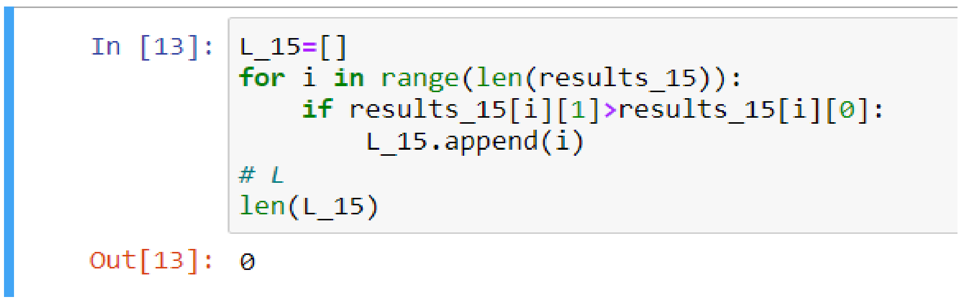
Based on a set of characteristics, it is intended to classify products with this status. The variable to predict is y (y=1 means with discount), all the other variables can be used to predict y. The dataset used on this project are not exactly the one given from *Banco de Portugal* since we did some feature selection techniques on it before importing to the Jupyter notebook sent.



```
In [12]: V = 15
         NN=BPcreate(V,data_train,learning_rate =0.15, epochs = 1000)
         results_15=BPpredict(NN=NN,TestSet=data_test)

         print(f'The accuracy on TestSet is: %.2f%%'%(accuracy(results_15,y_test_dm)*100))

         The accuracy on TestSet is: 79.68%
```

Figure 7: Test with BdP dataset and V=15.

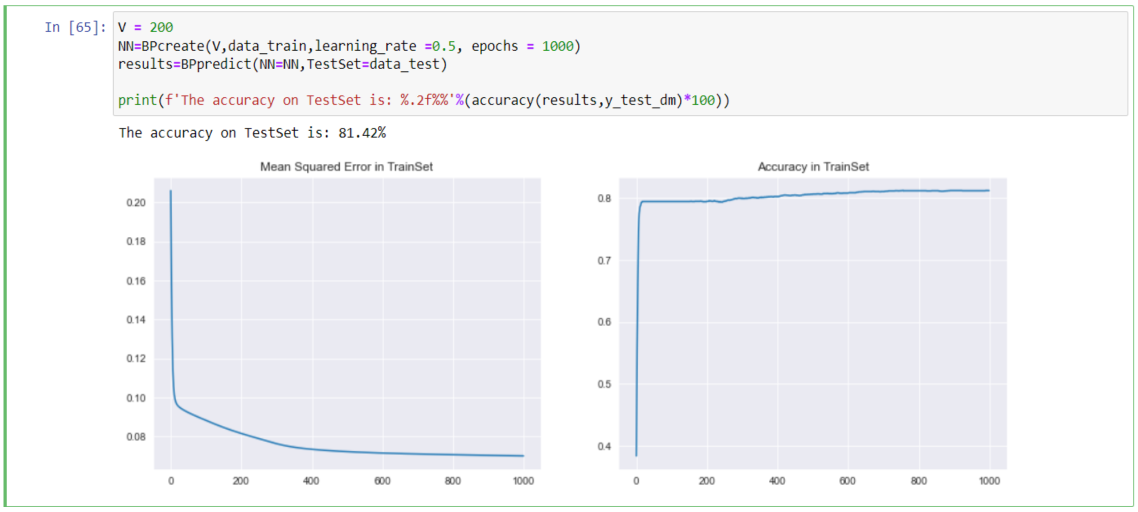
With only 15 nodes the NN cannot learn how to predict the class from the dataset, as we can see below all predictions are 0, so we need to grow the NN.



```
In [13]: L_15=[]
         for i in range(len(results_15)):
             if results_15[i][1]>results_15[i][0]:
                 L_15.append(i)
         # L
         len(L_15)

Out[13]: 0
```

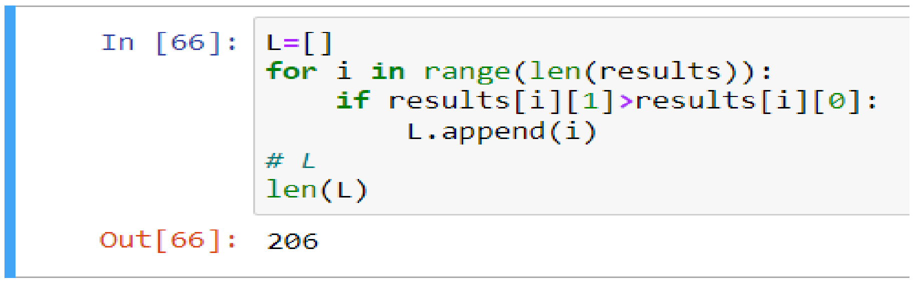
Figure 8: Code to count the predictions with value 1.

With "*V*" = 200 and the parameter "*learning_rate*" = 0.5, the NN made 206 predictions of value 1 and this improved the accuracy on "TestSet" nearly 2%. To measure the accuracy of predictions on this data set we used a prediction of another machine-learning algorithm which has almost 97% of accuracy according to the Kaggle competition.



```
In [65]: V = 200
         NN=BPcreate(V,data_train,learning_rate =0.5, epochs = 1000)
         results=BPpredict(NN=NN,TestSet=data_test)

         print(f'The accuracy on TestSet is: %.2f%%'%(accuracy(results,y_test_dm)*100))

         The accuracy on TestSet is: 81.42%
```

Figure 9: Test with BdP dataset and V=200.



```
In [66]: L=[]
         for i in range(len(results)):
             if results[i][1]>results[i][0]:
                 L.append(i)
         # L
         len(L)

Out[66]: 206
```

Figure 10: Code to count the predictions with value 1.

We think that 81.4% are not a very good accuracy, but it may for other reasons such as the NN architecture that was not the objective of this project since the goal was “implement the backpropagation algorithm for training multilayer neural networks”, and considering that the area of *deep learning* is complex and not well known for any of us (yet), we conclude that our functions definably creates and training Neural Networks, and these are able to do some correct predictions even on complex datasets.

Comparing the graphs from different values for *V* and different data sets we saw the learning of the Neural Networks by the backpropagation algorithm implemented by the function *BPcreate(), w*e also measured the accuracy of the predictions made by the function BPpredict(). This project was challenging and awakened our curiosity about deep-learning and neural networks itself, also improved our programming skills on python mainly on defining functions.

6. References

Banco de Portugal, 2022, “On sale-ECD1” dataset, https://www.kaggle.com/competitions/on-sale-ecd1/overview

Jason Brownlee PhD, 2019, “Why Stochastic Gradient Descent Is Used to Train Neural Networks”, https://machinelearningmastery.com/why-training-a-neural-network-is-hard/

Sebastian Mantey, 2019, “Basics of Deep Learning Part 13: Implementing the Backpropagation Algorithm with NumPy”, https://www.youtube.com/watch?v=ZVi647MYeXU&t=451s

Sebastian Mantey ,2019, “Basics of Deep Learning Part 14: How to train a Neural Network”, https://www.youtube.com/watch?v=Of2yOO52_t0

Steffen Nissen (DIKU), 2003, “Implementation of a Fast Artificial Neural Network Library (FANN)”