Assignment 1 - Computational Intelligence
Group 26

David Sachelarie, Sorana Stan, Iarina Tudor, Jason Miao

March 2022

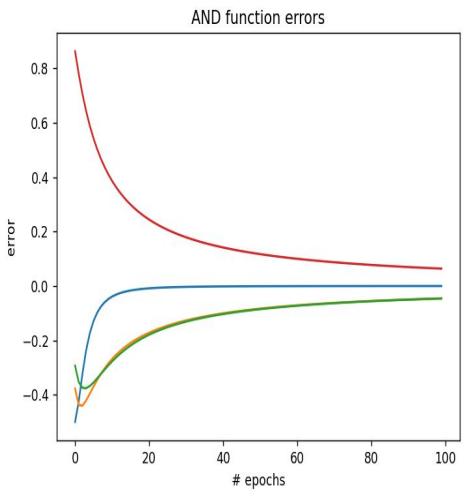
2.1. Set Up

The first part containing only one perceptron can be ran from the perceptron class (path: Group_26/Group_26_code/src/perceptron.py), each method will show a plot for the 3 functions: AND, XOR, OR. For running the neural network, the main class (path: Group_26/Group_26_code/src/main.py) can be ran which will result in running the neural network and outputting the results for the unknowns in the Group_26_classes text file (path: Group_26/ Group_26_code/ data/Group_26_classes.txt). A version of it can also be found at Group_26/Group_26_classes.txt. The packages needed are: numpy, matplotlib.pyplot, random, collections, sklearn.metrics import confusion_matrix, sklearn.model_selection import train_test_split.

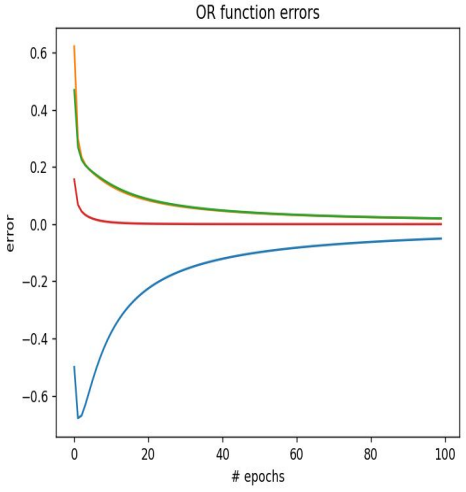
2.2. Architecture

1. Single Perceptron

After developing our single perceptron, we plotted the errors for each input X (we used four inputs containing all the possible combinations of 2 bits: 0 and 1. We first initialized the weights randomly and trained our perceptron across 100 epochs, with a learning rate of 1. As can be deduced from the graphs, when doing this, the error for the AND and OR functions decreases over time, converging close to 0.



(a) Error plot over epochs for AND function



(b) Error plot over epochs for OR function

For the XOR function, the errors converge to some values close to 1, meaning that the perceptron is not able to learn the XOR function. This is because perceptrons are limited to only solving problems that are linearly separable, and I the case of the XOR function, there is not a single line that can separate the 2 classes.

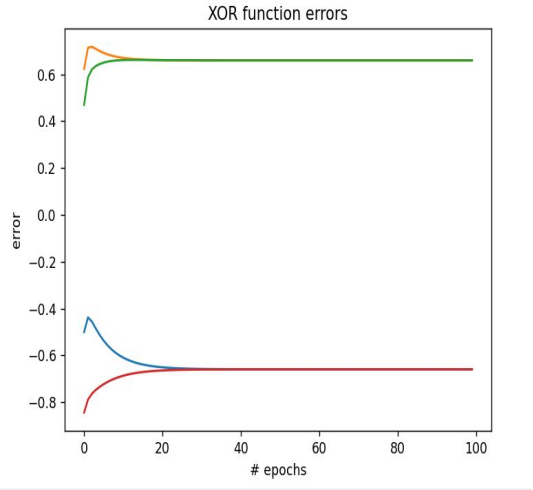


Figure 2: Error plot over epochs showing how a perceptron cannot learn XOR function

2. Input Neurons

Our data set needs to be classified into 7 classes by looking at 10 features. We learned from our single perceptron implementation that the number of features will determine the number of input neurons. Following this logic, we chose to use 10 input neurons representing the features of the inputs we will have to classify.

3. Output Neurons

As we mentioned above, we have 7 possible classes which means that 7 output neurons will be needed, each one of them representing a class.

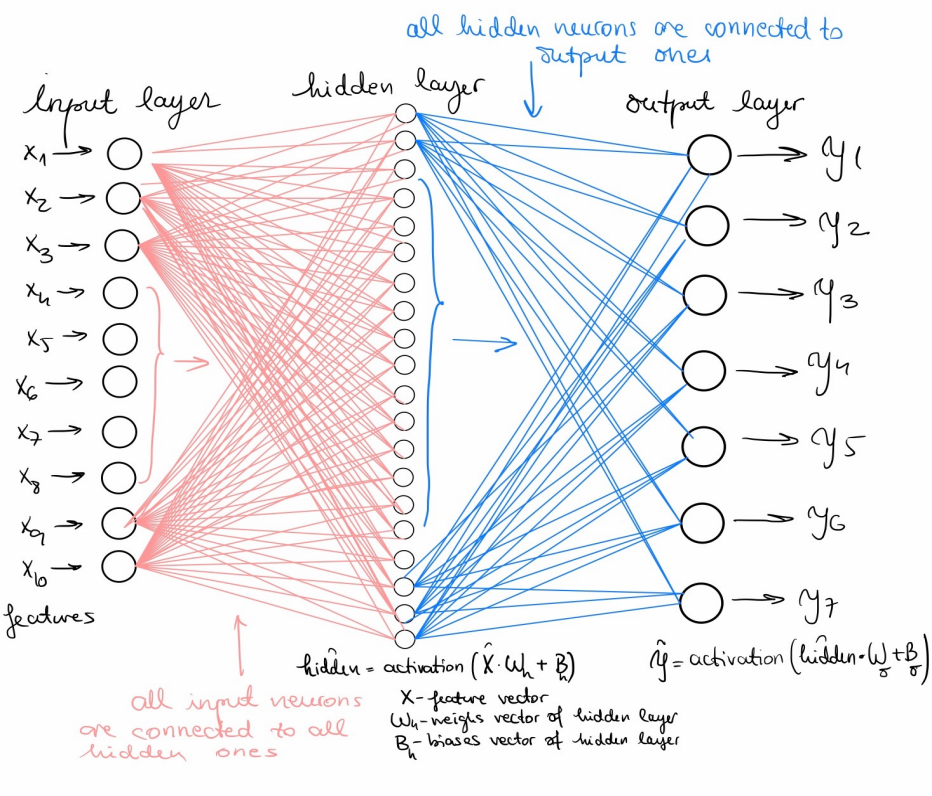
4. Hidden Layers

Most of the Neural Network architectures used in practice, contain only one hidden layer because the computation time can raise exponentially with each hidden layer added. Taking this into consideration, we chose to also use only one hidden layer, since our problem is not really complex and does not need to be broken up into too many subproblems. As a guess for the number of hidden neurons we chose 20.

5. Activation Function

The activation function that we are going to use is the sigmoid function. We chose to use this function because this is a common activation function used when dealing with classification problems.

6. Diagram



2.3. Training

1. Training, validation, test sets

As with most machine learning algorithms, a perceptron network needs a training set, used to calculate the weights and thus "learn" patterns in the input data, and a test set, which applies learned behaviour on a different collection of inputs, later comparing the returned to the expected results and thus calculating the accuracy of the algorithm. In addition, a perceptron network makes use of a lot of hyperparameters which are hard to optimize by hand, so a special type of test set, called a validation set, has to be used for tuning these parameters. As a result, the program mainly has three steps: training the network on the training set, tuning hyperparameters on the validation set, and testing the accuracy of the algorithm on the test set.

More pragmatically, we split the data as follows: 90% goes to the training set and 10% to the test set. In addition to that, we apply cross-validation and thus split the training set into batches, using each batch successively as a validation test.

2. Performance evaluation

As previously stated, the performance of the network is evaluated using the test set, which compares the network's prediction to the actual labels that should have been returned.

3. End of training

The provided data is split into training and test sets of fixed sizes, so training will end when all of the samples in the training set are evaluated. The answer is not so straightforward when using cross-validation, which will run the training set multiple times. A certain amount of batches have to be created from the training set, and each batch must act in turn as a

validation set. The other batches will constitute a new training set. The program will thus train and validate num_batches times, where num_batches is the number of batches chosen.

4. Trained 10 times

Our network initializes weights randomly in the constructor, but changes in values have little effect upon the final result. As seen in the graph below, all of the accuracies are between 93.2% and 94.2%, so there are indeed some slight variations, the reason why we should choose the mean of all 10 trainings as a final result.

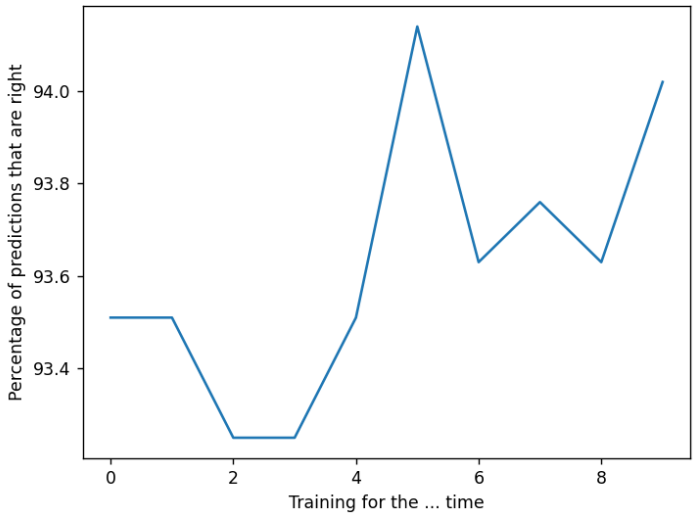


Figure 3: Accuracy on the test set if the network is run 10 times

2.4. Optimization

1. Performance vs number of neurons in hidden layer

We chose five different numbers of neurons in the hidden layer at random from the provided interval and applied cross validation successively on those values. As seen in the plot, the optimal amount lies somewhere around 20, and our program identified 22 as the one that gives the best result. An explanation to this is the way our network relates to the given data: too many neurons in the hidden layer and it overfits, too few and its decision boundary becomes inexact.

2. Performance of network with best result over epochs

The following plot is kind of predictable, since the general behaviour of the training remains the same. We chose the number of neurons in the hidden layer which give the best result, which is 22.

2.5 Evaluation

1. Success rate on the test set

After applying cross-validation and reaching our optimal number of neurons in the hidden layer (22), we are ready to test the network. The success rate on the test set is 93.67%, compared to 92.43% on the validation set corresponding to using 22 neurons.

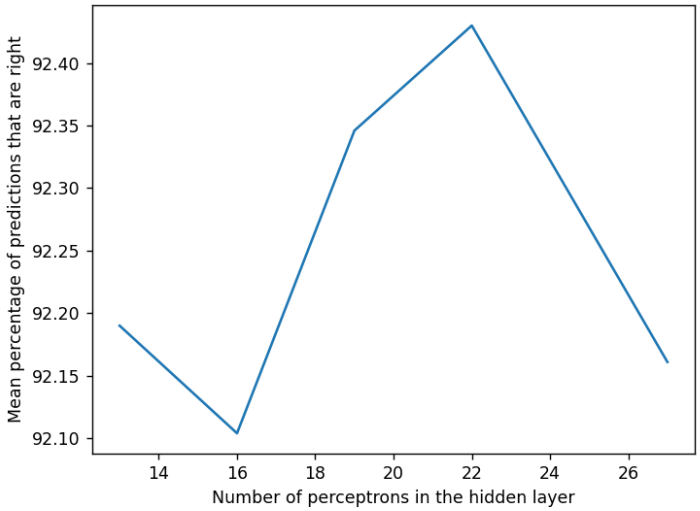


Figure 4: Accuracy depending on number of neurons

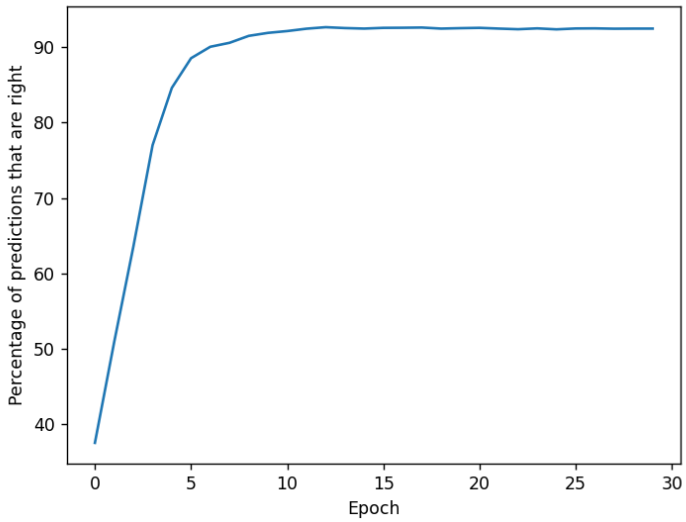


Figure 5: Performance of network with 22 neurons in the hidden layer

2. Confusion matrix

We chose to discuss the confusion matrix below. Let us name one line as i and one column as j. Then in each cell where $i = j$, all correct predictions are counted. If expected column j is wrongly predicted as i, then the value in cell *(i,j)* is incremented.

Leaving the numbers greater than 100 aside, since they count situations where the predicted class is the same as the expected class, we can make sums for each column and notice that the last one has a disproportionate amount of misclassifications. It thus comes to our attention that the network is not as good at recognizing samples from the last class as ones from the other classes. A disproportionate amount of mistakes are made on the last class.

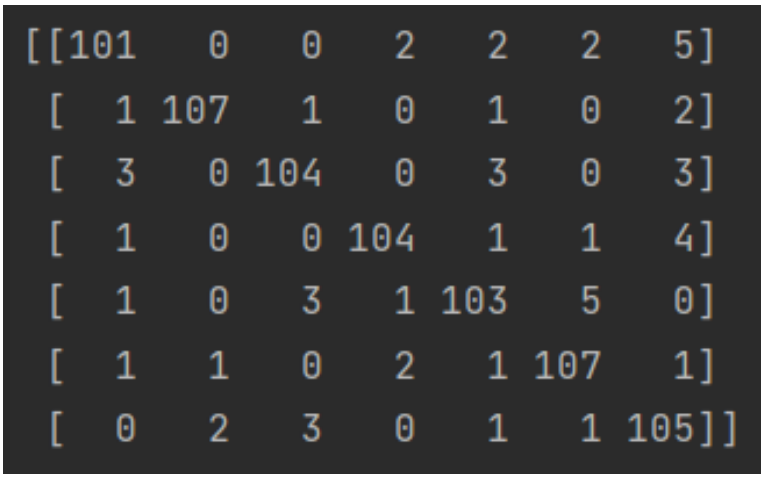


Figure 6: One of our confusion matrices

2.6 Scikit-learn

1. Optimal parameters returned by Scikit-learn

In our network, we have a few hyperparameters that we have to provide: number of nodes in the hidden layer, learning rate and batch size (this hyperparameter would give best results if it were 1, but the algorithm would become very computationally expensive; we thus left it at 10 in our program and on "auto" in the Scikit-learn notebook). We also used a logistic activation function. After running the grid search, the optimal hyperparameters returned were 0.01 for the learning rate (we used 0.1) and 23 neurons in the hidden layer (we used 22). We didn't notice any differences in the training behaviour and performance of the Scikit network compared to ours, its accuracies being close (around 92-93%).

2. Returned optimal hyperparameters plugged into our network

The parameters returned by Scikit-learn were not optimal for our program, which after using them had an accuracy on the test set of 71.83%. This is due to the too-small learning rate, which is not optimal for our implementation.

2.7 Reflection

1. Misclassification

An example of misclassification we thought of is when we want an algorithm to classify between potential criminals and citizens (to detect an upcoming terrorist attack), but all pictures used in the training phase with criminals were taken in front of shops and the ones with citizens in parks. This way, the algorithm actually learned the difference between parks and shops. It can be really harmful since innocent individuals that are just walking down the street can be considered suspicious. This can become a real problem when citizens end up in court without a valid reason, which can impact their lives forever. Furthermore, the government will not have any relevant information regarding a possible terrorist attack, therefore the software has no use in the end, it will only make harm.

2. Mitigation

A solution for unjust classification is to make sure that the training data is actually meaningful for our prediction. This means that it has to be prepared beforehand. The preparation can be made by making sure features which can create biases are not taken into consideration in the algorithm (example: gender, race features). We also need to make sure that the training data is collected from general and diverse sources, so our classification will not end up being between other classes than intended.

References

[1] Michael Nielsen
http://neuralnetworksanddeeplearning.com/

[2] 3Blue1Brown
https://youtu.be/Ilg3gGewQ5U

[3] Perceptrons: The First Neural Networks
https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/

[4] How to build your own Neural Network from scratch in Python
https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6