**Multi-Layer Perceptron for Binary Classification**

ASHIR BORAH

Dickinson College

Comp 364: Artificial Intelligence

Professor MacCormick

Dec 16, 2016

**Introduction**

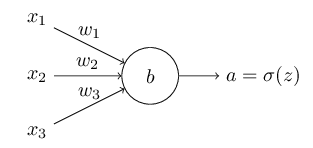
Neural networks have gained a lot of popularity in the last decade because of the increase in accuracy that has been achieved in many problems using the neural nets which have been very difficult to solve otherwise. The goal of the project is to build a feed forward neural network from scratch in Java and explore their functionality.

Neural networks have a wide range of uses and so the project beings by defining milestones for progression:

* Build a class that neural networks can inherit and have the functionality required to perform basic operations to achieve a functional neural network.
* Build a feed forward neural with no hidden nodes and try to achieve good results when the output node depends on just a single input node
* Build a feed forward neural network with a hidden layer such that the classifications are no longer a linear classification of the input variables
* Build an interface for functions such that several types of learning functions can be used to test their performance on the data set

This paper assumes a basic knowledge of linear algebra (Matrix multiplication, dot product) and multivariable calculus. It also assumes some basic knowledge of perceptrons.

**Perceptrons and How They Function**



**Figure 1:** Perceptron with 3 inputs and one output. [3]

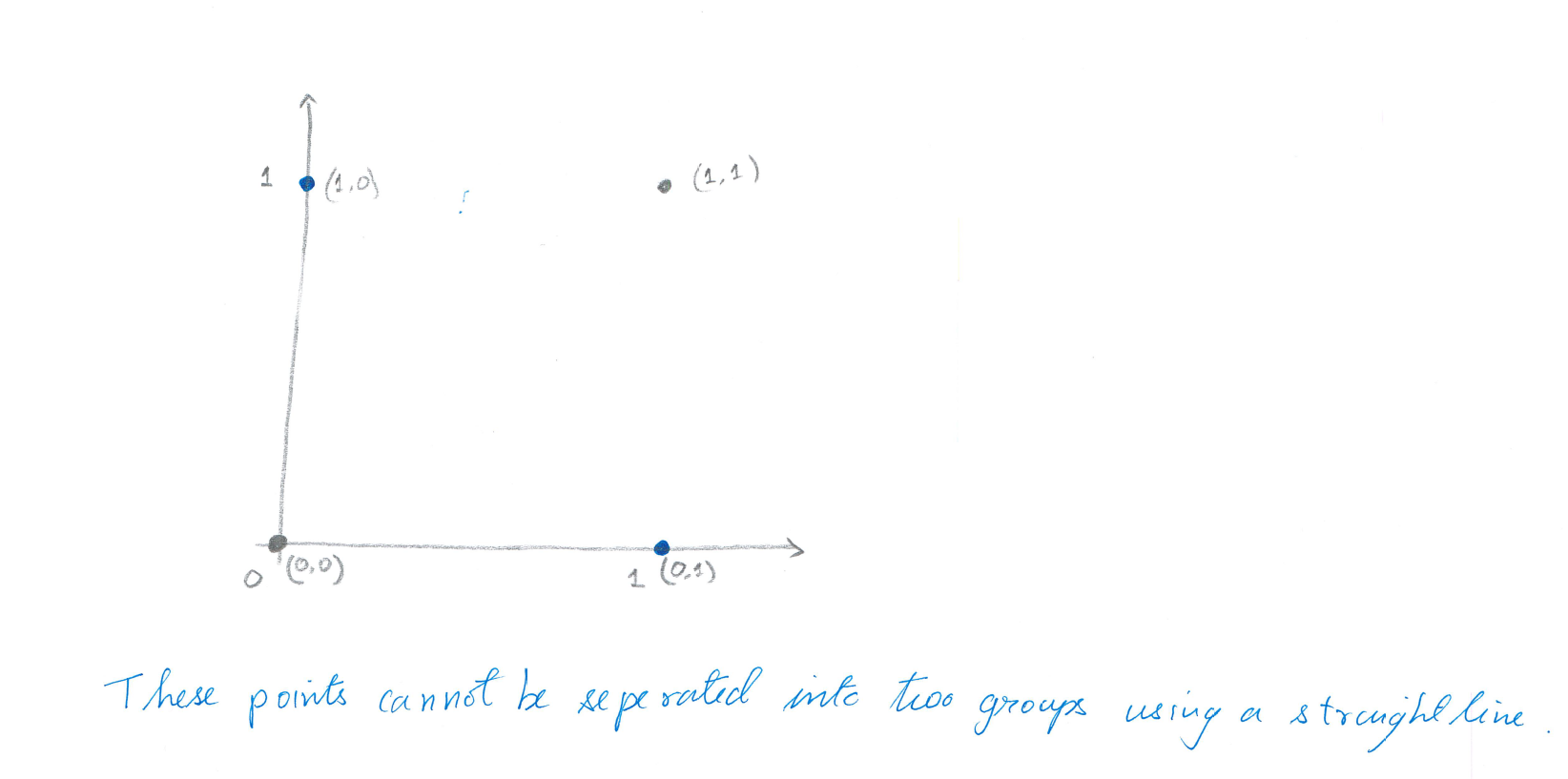
Perceptrons were inspired by the human neuron cells and are an approximation for use in computer science. [2] The perceptron has inputs and outputs which are modified by the appropriate edge weights. A perceptron can have more than one input and output. It depends on how the neural net has been designed.

For the given perceptron, the output a=x1w1+x2w2+x3w3-b (where b is the bias of the node)which is a linear combination of the weights, bias and the inputs. [3]

The edge weights are randomly initialized and then adjusted by the gradient descent algorithm and back propagation. The weights are adjusted to relate the input to the output. For example, if the output was directly related to input x1, the weights would be w1=1, w2=0 and w3=0. The setting of the edge weights is the learning process for the neural net.

**The XOR problem**

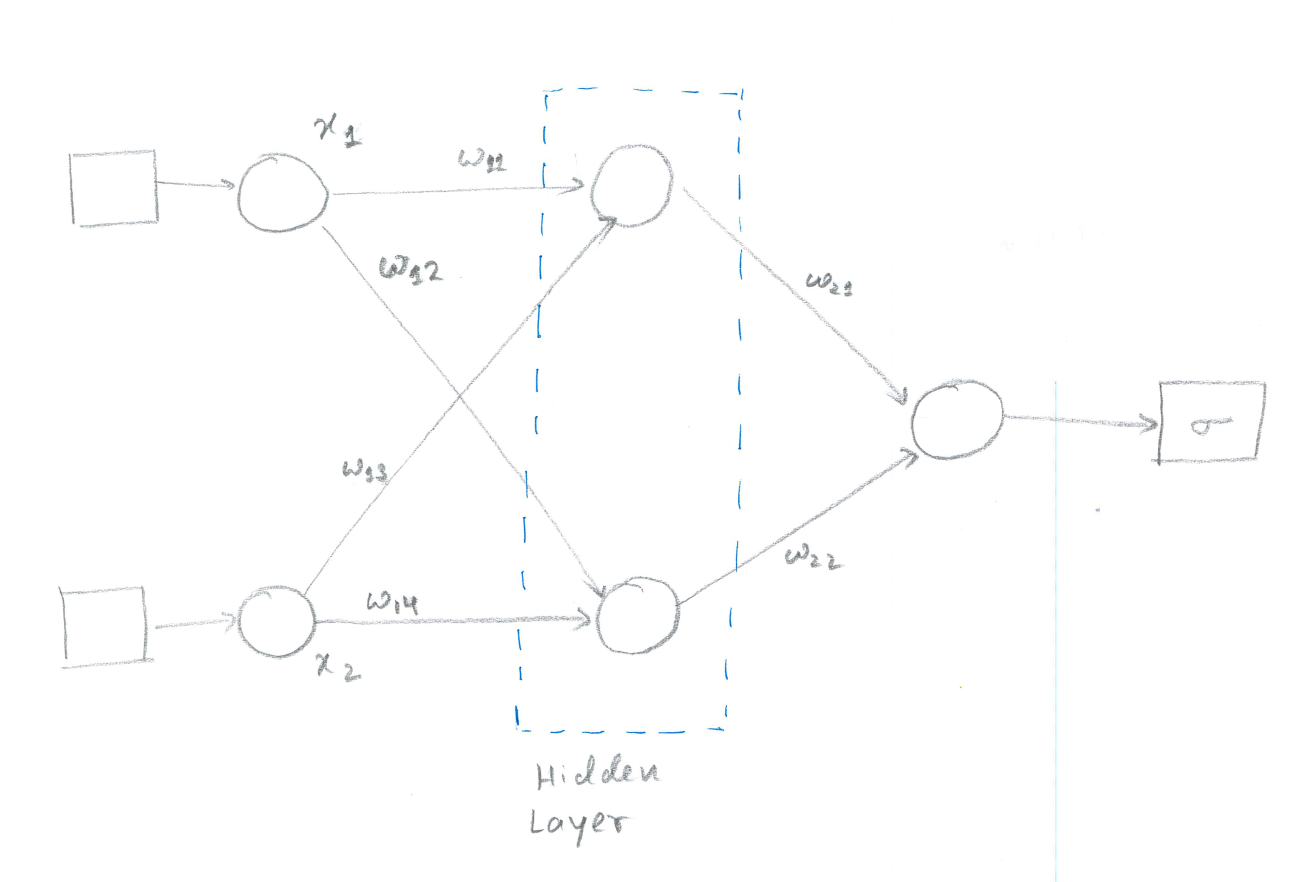
The XOR output when plotted on a graph is:



**Figure 2:** XOR output on Cartesian plane

Points (1,0) and (0,1) are on opposite sides of the unit square while (0,0) and (1,1) are on opposite sides. Hence they cannot be separated by a linear decision boundary. [2] This is because a single perceptron’s output is always a linear combination of its input and weights. Hence to solve this problem, a combination of perceptrons need to be used.

The network which can solve the given problem is:



Input

Output

**Figure 3:** Neural net with hidden layer

The addition of the hidden layer modifies the input and gives the network the ability to have nonlinear decision boundaries. With the correct learning function, the XOR problem can be solved with a single hidden layer.

**Learning Algorithms**

**Gradient Descent:** Gradient descent optimizes a function by following the gradient of the function to be optimized. In this case, it tries to minimize the cost function which is the error for each layer. [4]

**Back Propagation:** The gradient descend formula uses gradients to minimize the loss function. Backpropagation is an efficient algorithm which can be used to calculate the gradients in a computational graph efficiently. [2]

**Functions Used for Learning**

The step function:

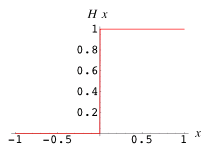


Figure 4: The step function

The step function has frequently used for binary classification. The function has a value called the threshold value(b). If an output is below the threshold value, it assigns the input to one class and if the output is above the value, it is assigned to the other class.

But a problem with the function is the fact that it has gradient 0 which is not very informative. An informative gradient is important because the gradient descent algorithm relies on the gradient to optimize the cost function. So, several other functions which can be considered smooth versions of the step function are used for learning purpose. Some of them are:

1. **The sigmoid function**

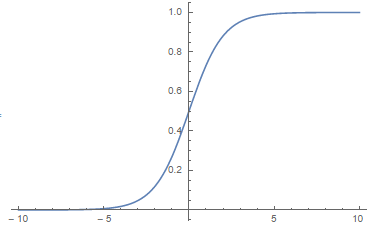


Figure 5: The sigmoid function

The sigmoid function works great as a learning function its gradient can be computed very efficiently. The slope of the sigmoid function at a point x is where is the value of the sigmoid function.

1. **The hyperbolic tangent function (tanh)**

The hyperbolic tangent function is also used for learning for some problems:

The function is defined as

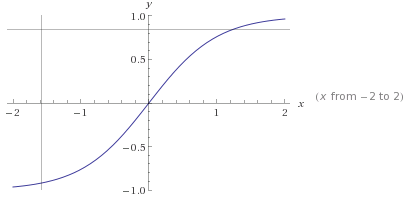


Figure 6: The hyperbolic tangent function

This function can also be considered a smooth version of the step function and is used for learning.

1. **The reLU and Softplus functions**

Another alternative to the step function is the Rectifier Linear Unit (reLU).

The reLU has the same problems as the step function and one of its smooth version is the softplus function. [6]

Rectifier(reLU)

Softplus

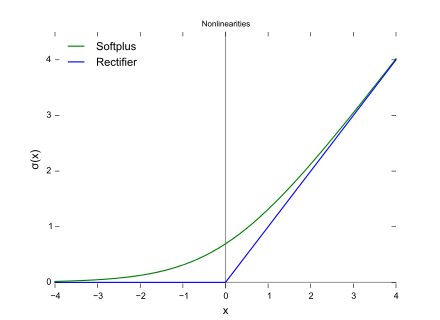


Figure 7: The reLU and softplus function

Many different functions are used as learning functions as specific problems tend to perform better on specific learning functions.

**Class Design**

The project focuses on the Rosenblatt’s perceptron. [1] Biases are not added to the perceptrons to reduce the number of layers in the network. The bias can be added by the same techniques which are used to add the hidden layers.

*The ffNeuralNetwork class*

The base class that the neural networks inherit have methods to help implement gradient descent and back propagation which are the algorithms used to implement the networks. The documentation covers each of the functions in detail. As matrices are used for knowledge representation, polymorphism is extensively used because matrix operations act differently depending on the dimension of the matrix.

The key functions of the base class are:

arrayThreshold: This function takes in an array and applies the step function to it. If the value of the specific element is below the threshold amount, the element is assigned to the first class else it is assigned to the other class.

correctNetwork: This function adjusts the weights in a layer using the gradient descent algorithm.

corrTerms: This function calculates the amount by which each node needs to be corrected by. Back propagation is used to calculate the amount. This is the key method for the gradient descent algorithm.

threshold: This method calculates the threshold value for the step function. It does this by taking the mean for the maximum and minimum of the output that the neural network produces.

The code also uses an interface that needs to be implemented by every learning function. This is done to increases modularity and enable the implementation of multiple functions which can be quickly implemented by extending the interface.

The learning functions that have been implemented in the project are:

1. Sigmoid
2. Hyperbolic tangent
3. Rectifier Linear Unit (reLU)

The neural network classes:

SingleNodeDependencyFFNN:

The single node dependency feed forward neural network has inputs such that the outputs depend on just one of the inputs. This class was created to test the various functions of the base class and the try the simplest case to facilitate debugging.

Some important sections of the code are:

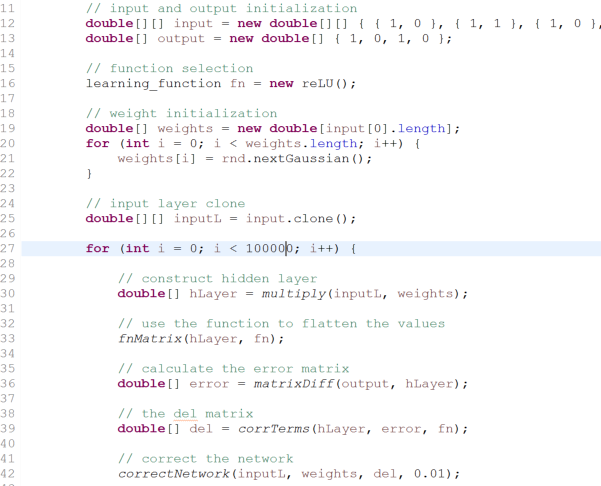


Figure 8: Code from the SingleNodeDependencyFFNN class

Line 16 states the learning function that is being used. Any function that implements the learning\_function interface can be used.

Line 27 states the number of iterations that the training will happen for. Each iterations training over all the training samples.

Line 30 constructs the hidden layer which is also our output because it does not have any other hidden layers. This is done by multiplying the inputs by the weights.

Line 33 applies the function that is being used to learn to smoothen the values so that mathematical operations can be used on them.

Line 36 calculates the error of the predicted output from the actual output.

Line 39 calculates the amount by which the weights need to be corrected and this correction is applied on Line 42.

This process is continued repeatedly.

The gradient descent algorithm takes place in functions corrTerms and correctNetwork while back propagating can be seen in the corrTerms function. Back propagation is more distinct as the number of layers increase.

TwoNodeDependencyFFNN

The class tries to solve the XOR problem by having two hidden layers. The layers are organized as shown in figure 3. The mathematical functions are identical to the singleNodeDependencyFFNN class.

**Experiments**

Experiment on the singleNodeDependencyFFNN class

The input is A and B and the expected output of the pair is listed in the output column.

|  |  |  |
| --- | --- | --- |
| A | B | Output |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 1: Input A and B with output ~B

Here the output directly depends on node B.

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Iteration | Sigmoid | Tanh | reLU |
| 0 | 0.462006 | 0.418315 | 0.215318 |
| 10000 | 0.058052 | 7.061807 | 0.00853 |
| 20000 | 0.02948 | 7.721013 | 0.004253 |
| 30000 | 0.019675 | 8.096071 | 0.00283 |
| 40000 | 0.014745 | 8.358312 | 0.00212 |
| 50000 | 0.011785 | 8.559768 | 0.001694 |
| 60000 | 0.009811 | 8.72321 | 0.001411 |
| 70000 | 0.008402 | 8.860639 | 0.001209 |
| 80000 | 0.007347 | 8.979153 | 0.001057 |
| 90000 | 0.006526 | 9.0833 | 9.39E-04 |

Table 2: The average absolute error for each learning function

The reLU function performs better than the other functions. The sigmoid function also works on the dataset while the tanh function increases the error. All the functions predict the correct output after the threshold function is applied to them.

Experiments on the TwoNodeDependencyFFNN class

**The NAND gate**

Input A and B for the function NAND with their corresponding outputs.

|  |  |  |
| --- | --- | --- |
| A | B | Output |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 3: NAND function input A and B with their corresponding output

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Iteration | Sigmoid | Tanh | reLU |
| 0 | 0.491472 | 3628.54 | 0.215318 |
| 10000 | 0.264137 | 6701.379 | 0.00853 |
| 20000 | 0.344408 | 6981.25 | 0.004253 |
| 30000 | 0.352908 | 7180.25 | 0.00283 |
| 40000 | 0.325425 | 7342.372 | 0.00212 |
| 50000 | 0.349298 | 7482.21 | 0.001694 |
| 60000 | 0.363293 | 7606.652 | 0.001411 |
| 70000 | 0.367899 | 7719.591 | 0.001209 |
| 80000 | 0.37003 | 7823.48 | 0.001057 |
| 90000 | 0.371226 | 7919.988 | 9.39E-04 |

Table 4: The average absolute error for each learning function for NAND

The sigmoid and tanh functions perform very poorly. The functions cannot predict the correct class that the inputs belong to. The reLU function performs well and classifies the inputs correctly.

**The XOR Gate**

Input A and B for the function XOR with their corresponding outputs.

|  |  |  |
| --- | --- | --- |
| A | B | Output |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Table 5: XOR function input A and B with their corresponding output

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Iteration | Sigmoid | Tanh | reLU |
| 0 | 0.500664 | Infinity | 0.504853 |
| 10000 | 0.497832 | NaN | 0.032848 |
| 20000 | 0.491691 | NaN | 0.010224 |
| 30000 | 0.484029 | NaN | 0.005817 |
| 40000 | 0.472542 | NaN | 0.004012 |
| 50000 | 0.448208 | NaN | 0.003043 |
| 60000 | 0.395008 | NaN | 0.002442 |
| 70000 | 0.313591 | NaN | 0.002034 |
| 80000 | 0.237461 | NaN | 0.00174 |
| 90000 | 0.183618 | NaN | 1.52E-03 |

Table 6: The average absolute error for each learning function for XOR

The sigmoid and tanh functions perform poorly again. The functions cannot predict the correct class that the inputs belong to. The tanh function gives the NaN output because it cannot find any solution to the problem. The reLU function performs well and classifies the inputs correctly.

**The NOR Gate**

Input A and B for the function NOR with their corresponding outputs.

|  |  |  |
| --- | --- | --- |
| A | B | Output |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 0 |

Table 7: NOR function input A and B with their corresponding output

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Iteration | Sigmoid | Tanh | reLU |
| 0 | 0.625139 | Infinity | 0.832828 |
| 10000 | 0.090161 | NaN | 0.001853 |
| 20000 | 0.038706 | NaN | 7.37E-04 |
| 30000 | 0.023525 | NaN | 4.45E-04 |
| 40000 | 0.01669 | NaN | 3.14E-04 |
| 50000 | 0.012884 | NaN | 2.41E-04 |
| 60000 | 0.01048 | NaN | 1.94E-04 |
| 70000 | 0.00883 | NaN | 1.62E-04 |
| 80000 | 0.007629 | NaN | 1.39E-04 |
| 90000 | 0.006717 | NaN | 1.22E-04 |

Table 8: The average absolute error for each learning function for NOR

The tanh functions perform very poorly again. The function cannot predict the correct class that the inputs belong to. The tanh function gives the NaN output because it cannot find any solution to the problem. The reLU and sigmoid functions perform well and classify the inputs correctly.

**Conclusion**

As the NOR and the NAND gates were constructed using the TwoNodeDependencyFFNN, it can be concluded that any logic gates can be made from a combination of either of the gates as they are both universal gates.

The sigmoid and tanh functions performed poorly as bias was not added to the nodes. This forces the functions to start at the origin which interfered with their prediction. The reLU function does well on the problems that we being explored.

The logic gates could have been made with decision trees but the problem with decision trees is the fact that we need all the training data when training begins. Neural networks can train and classify even with incomplete training and continue as more data arrives without discarding the previous training.

**Future Work**

Introduce bias in the network.

Include more functions and make the code more modular so that layers can be easily added. Possibly implement a wrapper class of the neural network.

Explore the performance of the neural network for classifying real world data. In principle, they should perform well after bias is added to the layers.

**Bibliography**

1. Denny Britz. 2016. Implementing a Neural Network from Scratch in Python – An Introduction. (October 2016). Retrieved December 16, 2016 from
2. Haykin, S. Neural Networks and Learning Machines. PHI Learning, New Delhi, 2011.
3. Russell, S and Norvig, P. Artificial Intelligence: A Modern Approach Third Edition. Pearson, Delhi, 2015.
4. Michael Nielsen. Neural networks and deep learning. Retrieved December 16, 2016 from <http://neuralnetworksanddeeplearning.com/>
5. <http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/>
6. Wikipedia. Rectifier (neural networks). Retrieved December 16, 2016 from <https://en.wikipedia.org/wiki/Rectifier_(neural_networks)>

Images and Graphs:

Figure 1: <http://neuralnetworksanddeeplearning.com/>

Figure 4: <https://en.wikipedia.org/wiki/Heaviside_step_function>

Figure 5: Generated on wolframalpha.com

Figure 6: Generated on wolframalpha.com

Figure 7: <https://en.wikipedia.org/wiki/Rectifier_(neural_networks)>