Implementing Backpropagation

Group 14

Popa Stefan Andrei
Levinschi Eduard

Maastricht University Department of Advanced Computing Sciences

November 6, 2025

1 Abstract

This report presents the implementation of a simple three-layer feedforward neural network (8-3-8 architecture) trained from scratch using gradient descent and backpropagation. Two activation configurations are evaluated: sigmoid on both layers and ReLU on the hidden layer with sigmoid on the output. The performance is assessed in terms of convergence behaviour and mean squared error (MSE) evolution.

2 Software description and usage

This project was implemented entirely in Python using only NumPy(as a library) for numerical computations and Matplotlib for visualizations. The input layer values(X) are already given and the aimed output should be similar(e.g. if X = [1 0 0 0 0 0 0 0] => Y = [1 0 0 0 0 0 0 0]).

The next step in order to begin our forward-back propagation process is to initialize weights, biases for each layer and a convenient learning rate(the choices will be explained later). Weights describes how important is the activation of a neuron the next layer. Biases act as adjustable offsets that allows the neuron to activate when all the input values are zero. Learning rate acts as a step in order to pass from one iteration of forward-back propagation to other. The main idea of our main file executes both forward and back propagation on a 8-3-8 neural network until the limit number of iterations given by us(10000) or a specific value of mean squared error is reached($<$0.05).

At the end of the forward propagation process a cost function is computed which in our case is MSE(mean squared error). Backpropagation performs the learning step of the network. It first calculates the output error and how much each neuron contributed to it, then propagates that error backward to the hidden layer. Using these error values, the gradients for all weights and biases are computed, and each parameter is slightly adjusted by the learning rate so the network's predictions move closer to the target outputs in the next iteration.

In the very end of the code,the MSE values over the training iterations are plotted, displaying the evolution of the error throughout the learning process.

3 Explanation on weights and learning rate

The network was trained on eight one-hot encoded vectors, and in all experiments it successfully learned to reproduce the input patterns at the output. When using sigmoid activation in both the hidden and output layers, the convergence was slower but stable, with the MSE gradually decreasing to around 0.05 after several thousand iterations. Replacing the hidden-layer sigmoid with ReLU made the training faster and more consistent, as the gradients remained stronger and the network avoided saturation. The learning rate α was tuned experimentally: a value of 0.1 provided smooth and stable convergence, while higher values such as 0.5–0.8 accelerated training but sometimes caused oscillations in the error curve. The network converged reliably for multiple random initializations when the weights were small and centered around zero, but when all weights started positive (0.5–0.9), the activations saturated and the gradients vanished, stopping learning after the first iteration. Overall, the training process remained stable once the initialization and learning rate were properly chosen.

4 Results and Comparisons

4.1 Predicted Values

In the table below is presented predicted values that has been computed during my forward-back propagation process, with an learning rate of 0.5 given and weighted values around 0.

	1	2	3	4	5	6	7	8
Y1	0.94	0.03	0.03	0.03	0.06	0.04	0.03	0.02
Y2	0.04	0.96	0.04	0.04	0.02	0.02	0.03	0.04
Y3	0.05	0.03	0.96	0.05	0.04	0.05	0.04	0.03
Y4	0.03	0.05	0.02	0.97	0.06	0.04	0.05	0.02
Y5	0.04	0.11	0.04	0.06	0.95	0.02	0.02	0.04
Y6	0.04	0.07	0.04	0.03	0.07	0.95	0.03	0.05
Y7	0.05	0.03	0.03	0.02	0.03	0.01	0.96	0.05
Y8	0.04	0.09	0.05	0.04	0.02	0.02	0.04	0.93

Table 1: Final layer values after reaching MSE$<$0.05

It is quite normal not reaching exactly 1.0 and 0.0 value because it is a machine and can not compute totally correct but the main the trend goes in the way that a neural network should learn.

4.2 Activation Function - Sigmoid

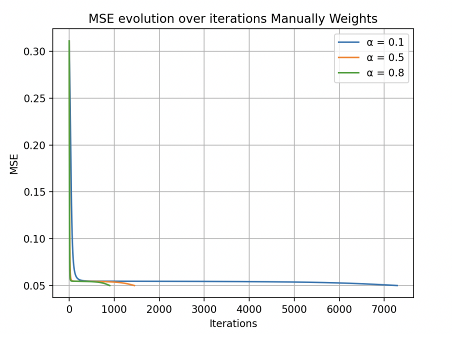

Figure 1: MSE evolution over iterations (manually initialized weights, Sigmoid activation)

The graph shows how the Mean Squared Error (MSE) changes during training for three different learning rates (α = 0.1, 0.5, and 0.8). In all cases, the error decreases very fast at the beginning, meaning the network quickly starts learning the correct output patterns. The higher learning rates (0.5 and 0.8) make the network learn faster, while the smaller one (0.1) is slower but more stable. After several hundred iterations, all three curves reach about the same error level, showing that the network was able to learn properly with all these values of α, which it was expected.

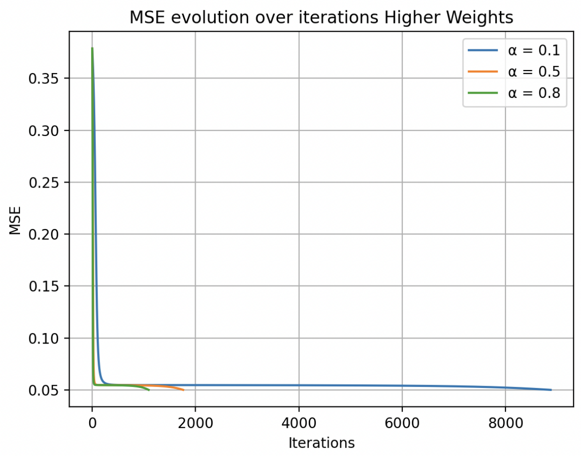

Figure 2: MSE evolution over iterations (higher initialized weights, Sigmoid activation)

The plot shows how the network's error decreases during training when the initial weights have higher values. In this case, the learning process takes more iterations to reach the same MSE level as before. Because the weights start larger, the neuron activations are pushed closer to the saturated regions of the sigmoid function, which makes the gradients smaller and slows down convergence. Even though the network eventually learns, it needs more time to adjust its parameters and reduce the error to the desired level.

4.3 Activation Function - ReLU-Sigmoid

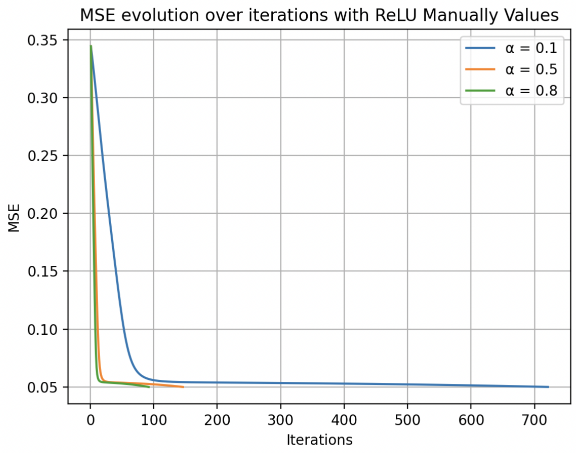

Figure 3: MSE evolution over iterations (manually initialized weights, ReLU-Sigmoid activation)

The graph shows how the error decreases during training when using ReLU in the hidden layer and Sigmoid in the output layer. The MSE drops very fast at the beginning and then slowly stabilizes around 0.05 for all learning rates. This means the network quickly adjusts its weights and finds a stable solution, showing that the chosen activations and parameters allow it to learn effectively.

4.4 Comparison

When comparing the two setups, the network using ReLU in the hidden layer and Sigmoid in the output layer learned faster and reached the desired error level in fewer iterations, which is quite normal. The ReLU activation allowed the gradients to stay stronger during training, which helped the network adjust its weights more effectively. In contrast with that, when Sigmoid was used in both layers, the learning process was slower because it is a difference between ReLU and Sigmoid level of speed, making the gradients smaller. In conclusion, in both setups the desired values was almost reached after few iterations, the only notable difference between them was n.o. iterations until reaching MSE established value(0.05), showing that they can both learn the identity mapping correctly.