Advanced Concepts in Machine Learning
Assignment 1

Eliott Simon, Kevin Müller

November 2021

1 Introduction

To start the assignment, we first started by implementing our input "training" and the corresponding output "testing" without any hidden layers in between. We tested if the feedforwarding works and if the error term delta is backpropagated through the network as a perceptron.

After that worked out, we added a hidden layer with randomly assigned weights, using the Xavier Weight Initialization technique. Since we have 8 input nodes, the weights have the structure of a 8x3 matrix for the weights between input and the hidden layer, and the structure of a 3x8 between the hidden layer and the 8 output nodes. These weights will get updated in every iteration of learning the network.

Just like the weights, we assign a random value to the Bias from the start. When implementing the backpropagation, we had a lot of problems to handle the bias weights in the right way. As we discussed in the Q&A session the bias weights should be added to the weight matrices, so that we get a 9x3 matrix for example for the weights from input to the hidden layer. What our network does right now is that the bias gets added after we multiply the input with the weight. We could not find a solution to backpropagate without the bias so the delta in our network depends on the bias value, as it should not be.

We are also aware that we could not find the bias delta and update the bias in the training of the network. The bias value gets randomly assigned in the beginning and stays the same over all iterations. In that case we are dependant on the values that get randomly chosen in the beginning.

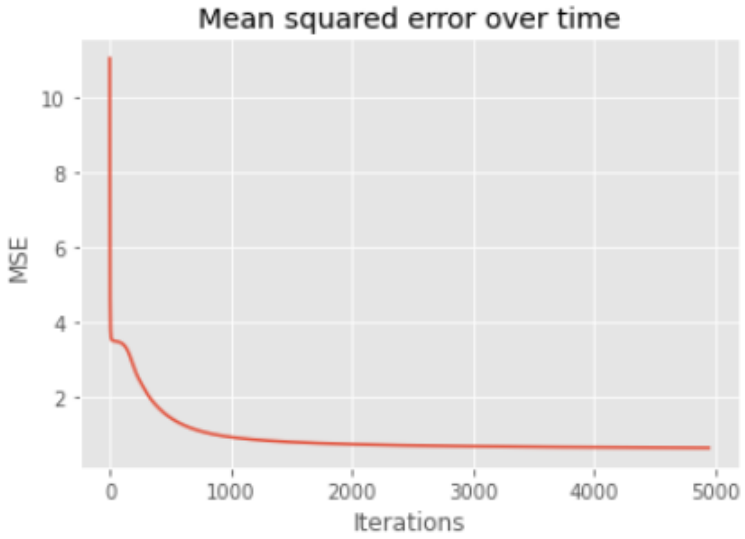
To present a running program we still went with the network, while having the flaw of backpropagating the bias and not being able to adjust the bias weights.

2 Project/Code Structure

The program was implemented using Jupyter Notebook, hence it results in a single (.piynb) file. The training/testing instances, represented as identity matrices are already included in the code.

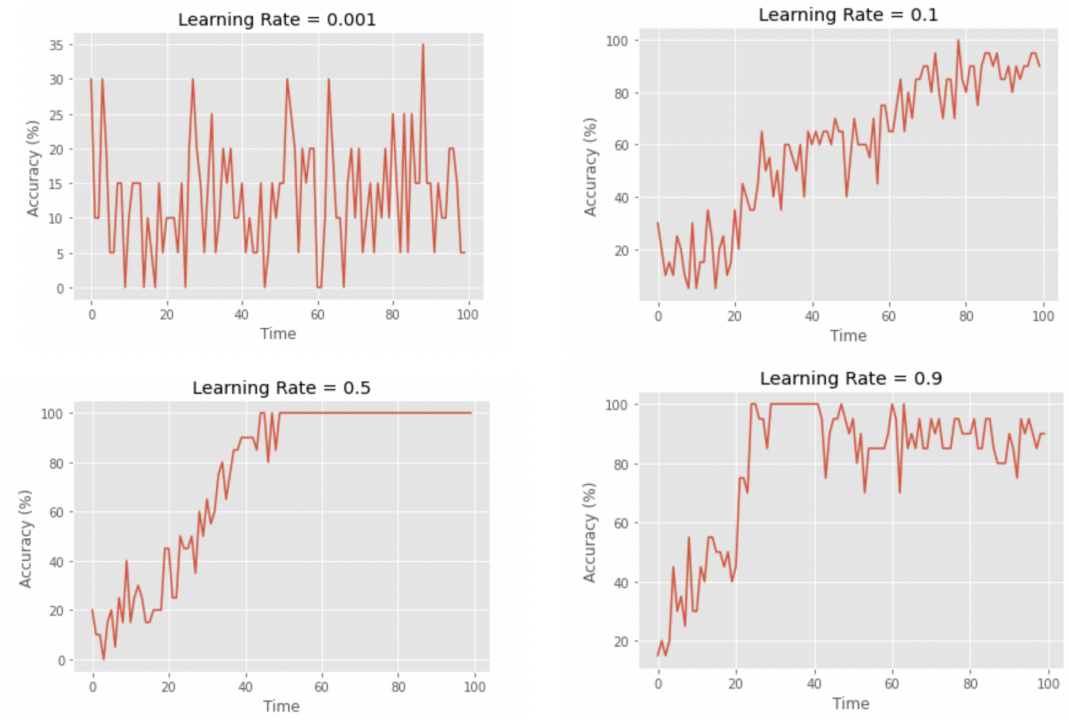
We train the network first with feedforwarding the input from previous layers, by multiplying with its corresponding weights and adding the bias. These are passed through an activation function to get the final output of the node, that gets forwarded to the next layer, until it reaches the output layer. With the output of the output layer, the mean squared error between the actual output and the predicted output is computed. The training then has the goal to minimize that error.

We compute error terms for both of the weight matrices going from input to the hidden layer and from the hidden layer to the output layer and update the weights through every iteration, by using gradient descent. This backpropagation phase results in the mean squared error to decrease significantly in the beginning from over 10 nearing towards 0.



3 Results/Learning performance

The network runs in a for-loop for the times specified with the variable "epochs", meaning it will go through the network and adjust the weights a certain amount of times. The learning rate of the network can be adjusted if wanted. We tested different learning rates, using 100 epochs for each training session. By adding a validation set, it is possible to estimate the accuracy of the model (throughout the training), as follows:

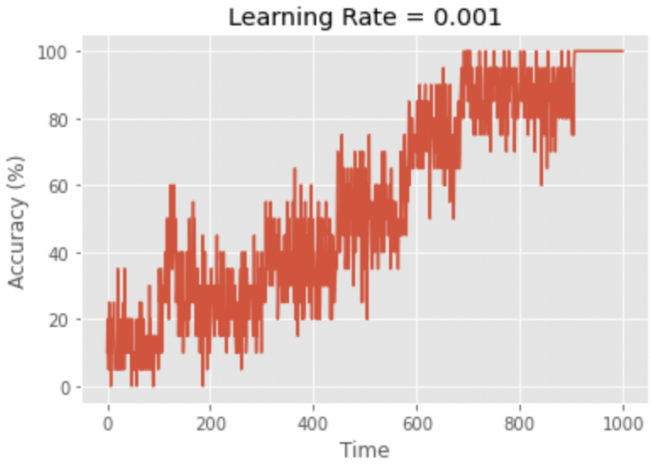


As discussed earlier, we are aware that the weights learned in training are still influenced by the bias, so the accuracy can also be influenced by that.

4 Interpretation

Learning rates

- With a small learning rate ($\alpha = 0.001$), the model does not manage to achieve a high accuracy within few times, since it will need far more passes through the network to update the weights correctly, as follows:



  However, this is not optimal since it takes way too much time to train.

- When using a learning rate of ($\alpha = 0.9$), the model learns very fast and achieves very accurate results in early iterations. However, it is **very** likely that the model over fits the data, when using such a high learning rate. We can clearly see that after 40 iterations, the model's accuracy drops from a 100 % to lower accuracies, which confirms that it **did** over fit.

- The learning rate of ($0.1 < \alpha < 0.5$) seems to be the best option, since it converges to high accuracies within a short amount of time. This model is relatively simple (only 3 nodes and one hidden layer), which is the reason why it takes a few epochs before reaching promising results.