Advanced Concepts of Machine Learning: Implementing an Artificial Neural Network by Backpropagation

Steffen Schneider (i6169435), Kira Ruschmeier (i6181835)

November 6, 2018

1 Introduction

In this assignment, we implemented a neural network by back-propagation. The network consists of 3 layers whereby the input and the output layer contain 8 nodes respectively while the hidden layer implies 3 nodes. Additionally, a bias node is appended to the input layer as well as to the hidden layer. Thus, the model is more flexible as the activation function is shifted vertically to fit and predict the data in a more reliable manner.

2 Implementation details

The assignment was implemented in Python (version 3.7.1) using the libraries *numpy* for the matrix operations and *matplotlib* for plotting the results.

We have tried to vectorize the implementation for feed-forward and back-propagation. The neural network takes a matrix of input values, runs the feed-forward and learns the weights using the back-propagation algorithm.

Initially the weights for all nodes (including the bias node weights) are initialized to small random values (between 0 an 0.1). The bias nodes are set to a constant value of 1.

In the feed-forward part, the bias nodes are appended to the input value matrix. Similarly the bias weights are appended to the weight matrix of layer 1 (hidden layer). We append the bias node and weight for the output layer in a similar fashion.

3 Selection of parameter values

In the implementation there are mainly two parameters of the neural network which have to be tuned in order to see a good learning performance:

1. learning rate (α)
2. regularization term (λ)

We first looked at different learning rates. The α values should be neither too high nor too low. The following examples show the effects of a too high or a too low α value. Afterwards we look at the regularization term λ.

3.1 High α

If α is set to a too high value the neural network will decrease the error quickly but it might "step over" the local minima in the gradient descent. Therefore, it might not actually find the optimum. Figure 1 shows such a case where the α value is too high. For this experiment we set $\alpha = 20$.

The first graph in the figure shows the mean error which drops immediately after only a few iterations.

The second graph shows the prediction value for the actual value of the point in the dataset (e.g. the first predicted value the input (1, 0, 0, 0, 0, 0, 0, 0), the second predicted value for the input (0, 1, 0, 0, 0, 0, 0, 0), etc.). The value for the expected output (where there is 1's in the input) should be at least larger than the values the other output (where there is 0's in the input). This is because we are essentially looking at a One-vs-Rest classification. Since any derivation from the input-data value leads to a higher cost, the neural network is trying to reduce these values by adapting the weights in the back propagation.

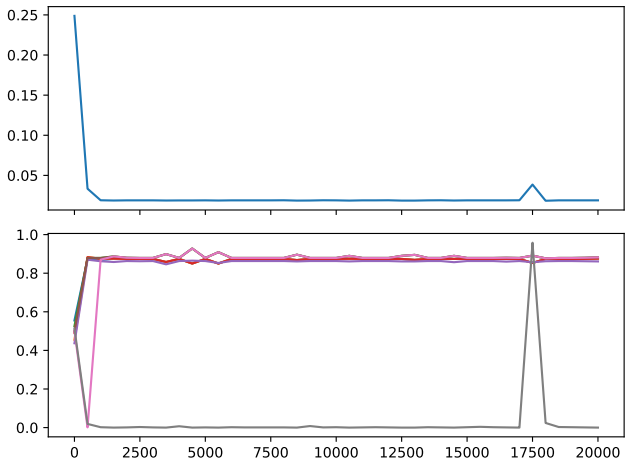

Figure 1: $\alpha = 20$ is too high

In Figure 1 we can see that for at least one of the inputs the predicted value stays close to 0 for a long time. At some point gradient descent manages to optimize the weights such that the predicted value is correct. However, due to the high learning rate the weights are adapted too much, the gradient descent "steps over" the optimal value. This overstepping can be explained by the rise of the mean error which can be seen at iteration 17500. The algorithm tries to optimize the cost and ends up changing the weight too much in the "other" direction. (The graphs in Figure 1 only plot steps of every 500 iterations, which is why the peak isn't completely steep at iteration 17500. Another graph with a higher resolution can be found in the appendix.)

3.2 Low α

If on the other hand the α value is chosen to be too small the algorithm might take a long time to converge. In Figure 2 we can see that (except for the initial drop) the error takes a long time to get smaller. Further, we can see that it takes many iterations for the values to be predicted correctly. Some of the values are still not predicted correctly even after 20000 iterations.

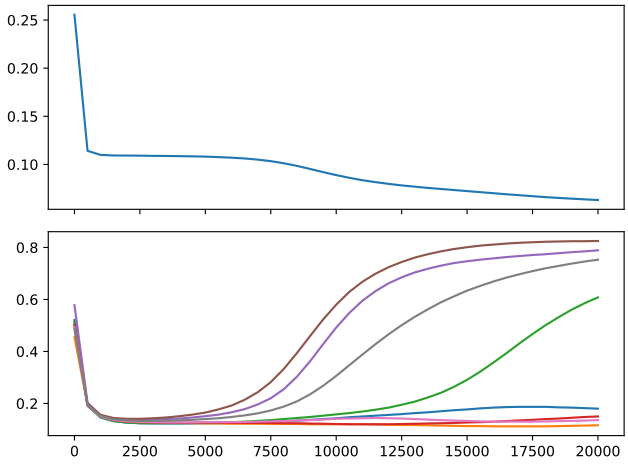

Figure 2: $\alpha = 0.01$ is too low

3.3 High λ

Figure 3 shows that the predicted values stay low. In this example a learning rate $\alpha = 0.5$ was used. Looking at the prediction we could see that all predictions were identical. No sensible weights for the neural network have been learned in this instance. This is because the regularization term puts too much importance on the sum of weights, opposed to the cost due to the deviation in the prediction

from the actual value. Hence the back-propagation algorithm will keep all the weights as small as possible.

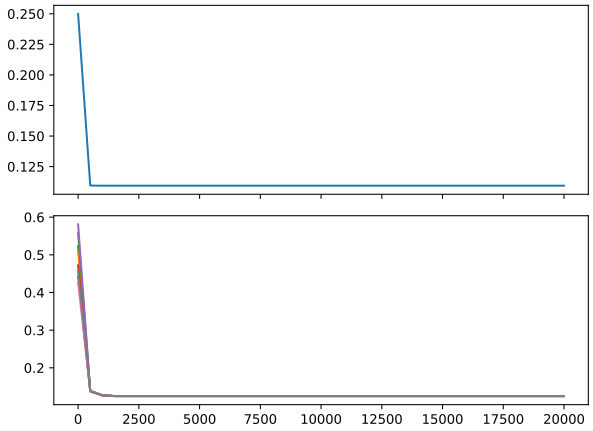

Figure 3: $\lambda = 0.2$ is too high

3.4 Low λ

Figure 4 shows exactly the opposite effect. In this example again a learning rate $\alpha = 0.5$ was used. Here we can see that most of the values converge to 1 rather quickly (after 1000 to 2000 iterations). Although this doesn't seem to be a problem in our task, in a real world example this might cause our model to over-fit the data.

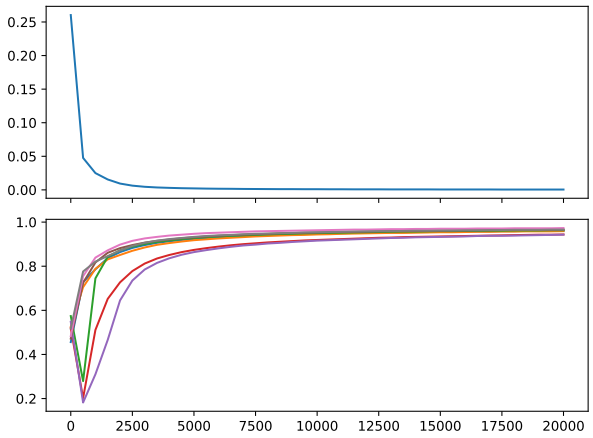

Figure 4: $\lambda = 0.00001$ is low

4 Learning Performance

Figure shows an experiment with a learning rate of $\alpha = 0.5$. Another observation that can be made by figure 6 is that all predictions converges nearly at the same time. As they get different initial weights, one input value is learned faster than the other which can be observed in the interval from 0 to 2.500. In this configuration it takes about 5 seconds to finish all 20000 iterations. The mean difference between the expected and the predicted output is shown in figure 5. There you can see that this error decreases during an increasing number of iterations.

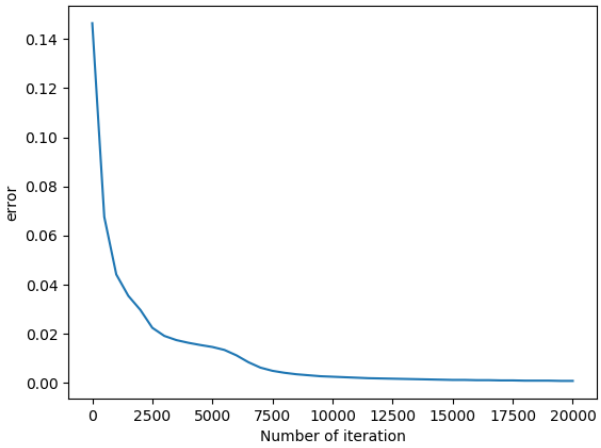

Figure 5: Decreasing error during increasing number of iteration

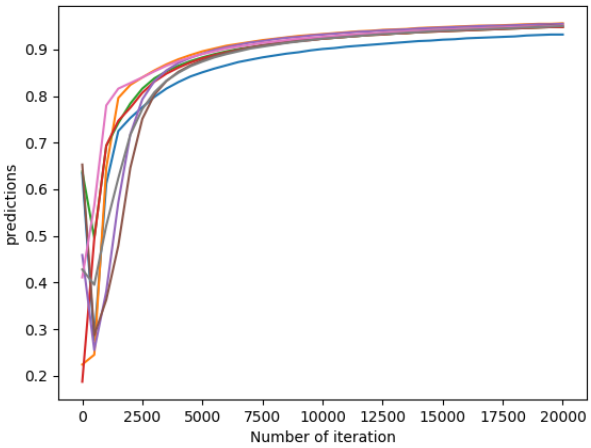

Figure 6: Prediction of each input in a different color

5 Interpretation of learned weights

Looking at the weights of the hidden layer and the activation of the hidden layer we can see that the input digits are represented in a binary system. This makes sense because there are 8 different input combinations that result in 8 corresponding output combinations. These 8 combinations can be stored in 3 bits ($2^3 = 8$). Each node in the hidden layer represents one bit, so the system can store and derive the 8 combinations from the hidden layer.

5.1 Weights hidden layer

The following example shows the weights of the hidden layer and the activation of the hidden layer after the neural network has learned the weights:

-2.6056331	-2.62016111	2.56933437
-2.58809566	-2.5570014	-2.63550411
-2.6129397	2.60306877	-2.567597
2.59352807	2.57426469	-2.62946797
2.58794089	-2.61995866	2.62216455
2.62995886	2.62756189	2.62108171
2.58818209	-2.62832781	-2.5685798
-2.61880866	2.57132855	2.62492222

Table 1: Weights of layer 1

If we put a 1 wherever we have a positive value, and a 0 wherever there is a

negative value in the above table, the resulting table will look like this:

0	0	1
0	0	0
0	1	0
1	1	0
1	0	1
1	1	1
1	0	0
0	1	1

Table 2: Binary representation of the weights of layer 1

5.2 Activation hidden layer

We can make similar observations looking at the the activation of the hidden layer:

0.06860715	0.06827653	0.92781257
0.06973695	0.07240766	0.06591061
0.06814162	0.93148763	0.07021858
0.93027203	0.9296256	0.06628367
0.92990854	0.06828961	0.93127363
0.93259897	0.93303456	0.93120389
0.92992426	0.0677587	0.07015441
0.06776983	0.92943314	0.93145

Table 3: Activation of layer 1

If we round those values we will get the resulting table:

0	0	1
0	0	0
0	1	0
1	1	0
1	0	1
1	1	1
1	0	0
0	1	1

Table 4: Activation of layer 1 (rounded)

The binary system in the hidden layer might vary each time we trained the system and are determined by the learned weights. This is due to the fact that

the order of the internal nodes is arbitrary and mostly depends on the weight initialization. These depend on the configured parameter for example number of iterations, the regularization term (or weight decay) and the learning rate.

The weights for the output layer show exactly the opposite transformation, namely a mapping from the binary representation back to the original representation.

6 Final remarks

We decided to try to vectorize as much as possible in our implementation. We are quite confident that the vectorized version of the feed-forward is correct. However, in the vectorized version of the back-propagation we still have some doubts. We are treating our input as a batch of input values. We believe that since we are taking the dot-product of the calculated error and the weights to calculate the deltas, we don't have to divide the deviation by the size of the input-batch (we've had some discussion with classmates about this). However, we are not entirely sure this is the true. We introduced a Boolean-flag in our code (scale_by_input_size) which, if set to "True", will scale the deltas by the input batch size. In our experiments we found that it would mainly affect the convergence speed.

Time spent: We roughly spent 2 days per person on this assignment. Getting the basic implementation running (without bias nodes and without regularization term) was rather straight forward. But the it took us quite a lot of time to get the details sorted out.

7 Appendix

Another example of a too high α value. This time we chose a higher resolution of the graph, which now contains datapoints for every 100th iteration. The spikes in the graph clearly show how the algorithm is "overshooting" the optimum.

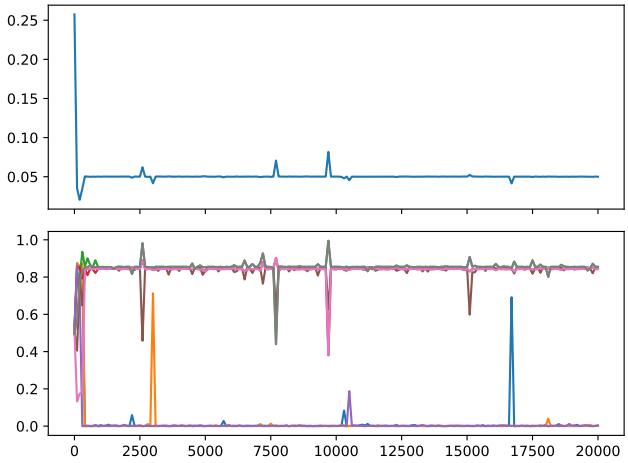

Figure 7: $\alpha = 20$ is too high