# Advanced Concepts in Machine Learning

## Assignment 1 Report

Maastricht University

Department of Data Science and Knowledge Engineering

Tomasz Stańczyk - i6209867

Ismail Alaoui Abdellaoui - i6204837

November 2019

## Introduction

The purpose of this assignment was to design a neural network without using high level libraries like keras or Tensorflow. The learning examples each have 7 zeros and 1 one in them. Therefore, there is only 8 different learning examples, and the weights learned by the network should contribute to the output which is exactly the same as the input. The input layer has 8 nodes and an extra bias node, the hidden layer has 3 nodes and an extra bias node, whereas last (output) layer consists of 8 nodes.

This report summarizes the work done over the assignment introduced. The structure of the report includes the following sections. Firstly, the developed software description as well as instructions how to run it are provided is provided. Next, as a part of the network performance, the experiments conducted and their results are presented. Further, the average execution time is pointed. Finally, the learned weights are discussed and interpreted.

## Software Description

The software consists of two parts, the first part consists of the main function definitions and the parameters of the network. The main functions used are:

- forward_propagation_vectorized: performs the forward propagation and returns the values of the nodes after activation
- backward_propagation_vectorized: performs the backward propagation and returns the delta values needed to update the weights
- backpropagation_algorithm: does the training of the network by iteratively calling the 2 functions above and returns the final weights and the Root Mean Square Error (RMSE)
- create_weights: generates the weights randomly from a normal distribution with a mean of 0 and a standard deviation of 0.01

The implementation of the 2 first functions are based on the lecture slides and formulas provided there. It should be also noted that the implementation is vectorized, meaning that we are providing the all the 8 inputs at once.
The other hyperparameters used are:

- Learning rate: 1
- Lambda: 0.0001
- Number of iterations: 10,000

The second part of the software consists of the different experiments in order to know the impact of the hyperparameters on the overall performance. There are two types of experiments: the first one is the parameter tuning during the training and the second one is the tuning of the parameters after the training. The parameters that were tuned are: learning rate, lambda, and number of iterations.
The functions used for plotting the experiments are:

- test_learning_rates: plots the RMSE using the learning rates 0.001,0.01,0.1,1,10,100. Every data point is separated by 50 iterations.
- test_lambdas: plots the RMSE using the lambdas 0,0.0001,0.001,0.01,0.1,1,10,100. Every data point is separated by 50 iterations.
- test_final_error_learning_rates: plots the RMSE of the whole training phase using the learning rates 0.001,0.01,0.1,1,10,100

- test_final_error_lambdas: plots the RMSE of the whole training phase using the lambdas 0,0.0001,0.001,0.01,0.1,1,10,100
- test_final_error_iterations: plots the RMSE of the whole training phase using the iterations 10,100,1000,10000,100000

### Instructions to run the program

The dependencies needed are “numpy” and “matplotlib” libraries.
In order to run the program, please execute the following steps:

- Open the command line in the same directory as the python file
- Enter the command “python a1.py”

It should be noted that the calls for the functions generating the plots are commented out, therefore we can generate the plots by uncommenting the function calls. It is advised to uncomment only 1 function at once to output clear plots.

## Brief description of the learning performance of the network

### Experiments

Several experiments have been conducted, with varying values of learning rate, regularization parameter (referenced as lambda) and number of (training) iteration. As a chosen metrics, the root-mean-square error (RMSE) between expected output and predicted output was being measured.

At first, different values of the learning rate have been tested, with the values of lambda and number of iterations set to 0.0001 and 10000 respectively. The values considered for the learning rate were 0.001, 0.01, 0.1, 1, 10 and 100. The values of RMSE were measured every 50 iterations. Figures 1-6 present RMSE per iteration for the referenced values of the relevant hyper-parameters.

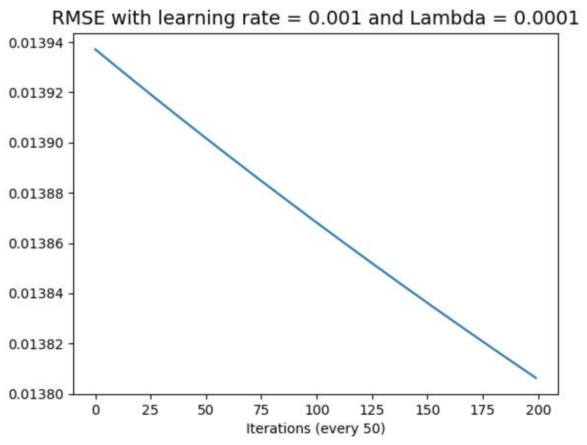


*Figure 1 - RMSE measured during training for learning rate=0.001 and lambda=0.0001*

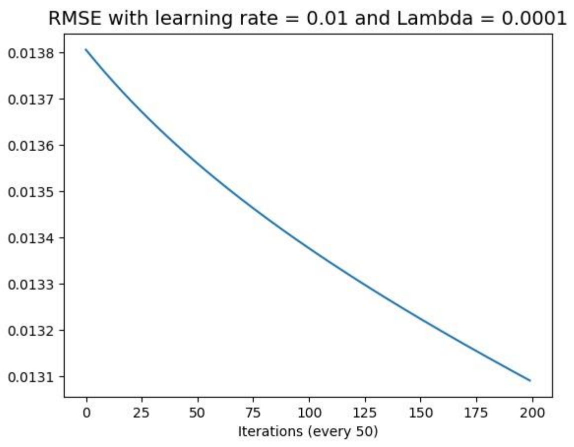


*Figure 2 - RMSE measured during training for learning rate=0.01 and lambda=0.0001*

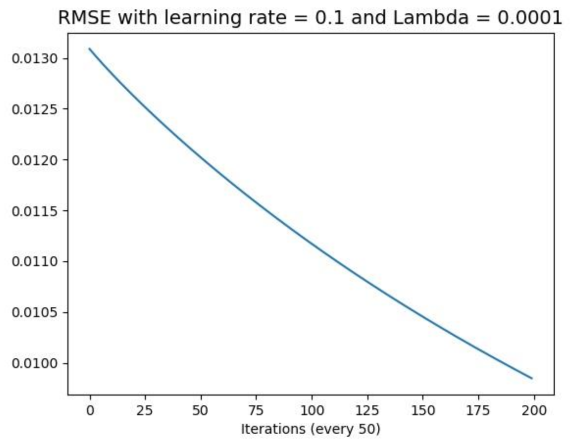


*Figure 3 - RMSE measured during training for learning rate=0.1 and lambda=0.0001*

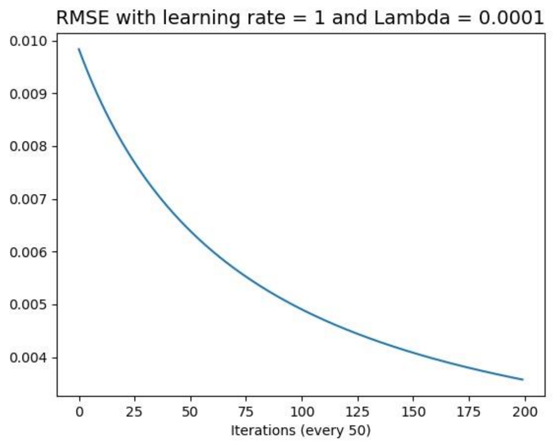


*Figure 4 - RMSE measured during training for learning rate=1 and lambda=0.0001*

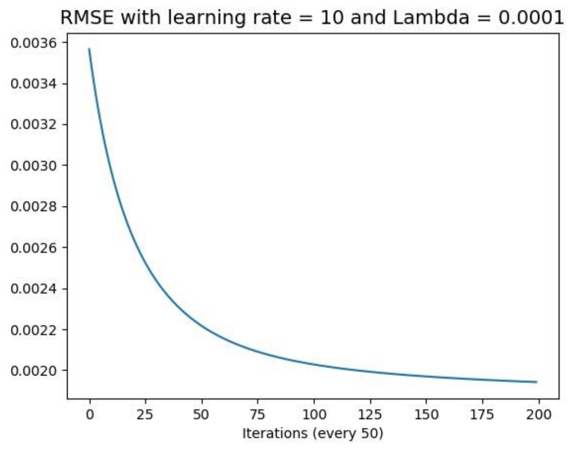


*Figure 5 - RMSE measured during training for learning rate=10 and lambda=0.0001*

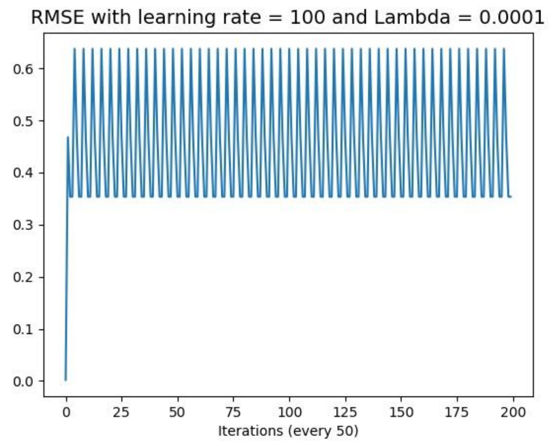


*Figure 6 - RMSE measured during training for learning rate=100 and lambda=0.0001*

As it can be observed from the Figures 1-5, higher learning rate seems to be beneficial for the learning process in terms of the error convergence. However, it is not the case, when learning rate is too high, as presented in Figure 6. In this case, gradient descent, which was the learning algorithm applied, "overshot" the (cost) minimum and could not head back to it.

Next set of experiments included varying values of lambda, with values of learning rate and number of iterations set to 1 and 10000 respectively. The values considered for lambda were 0, 0.0001, 0.001, 0.01, 0.1, 1, 10 and 100. Again, the values of RMSE were measured every 50 iterations. Figures 7-14 present RMSE per iteration for the referenced values of the relevant hyper-parameters.

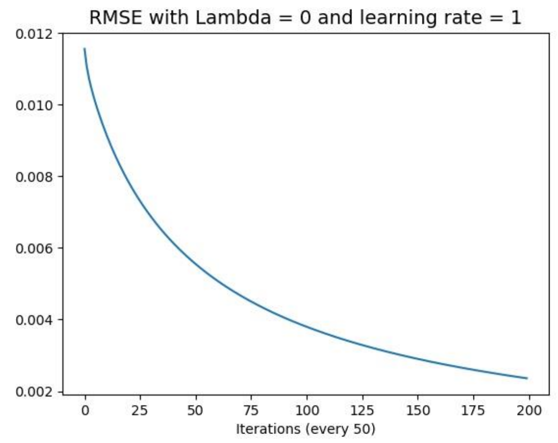


*Figure 7 - RMSE measured during training for lambda=0 and learning rate=1*

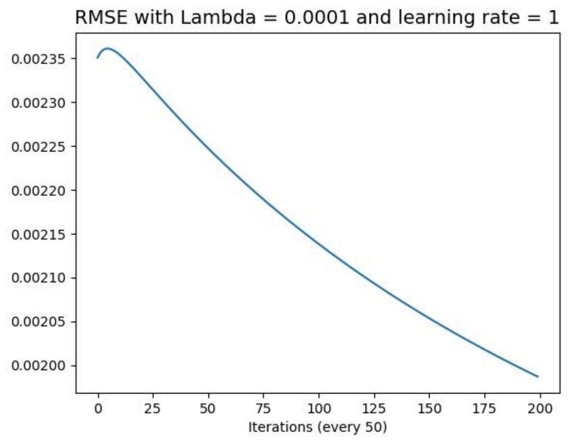


*Figure 8 - RMSE measured during training for lambda=0.0001 and learning rate=1*

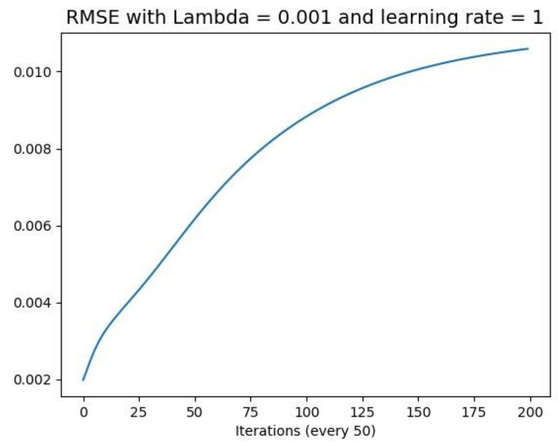


*Figure 9 - RMSE measured during training for lambda=0.001 and learning rate=1*

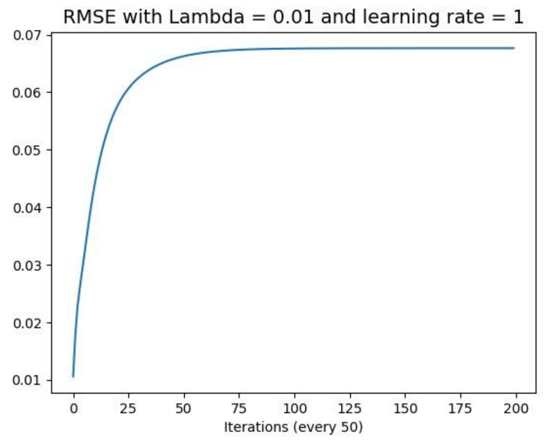


*Figure 10 - RMSE measured during training for lambda=0.01 and learning rate=1*

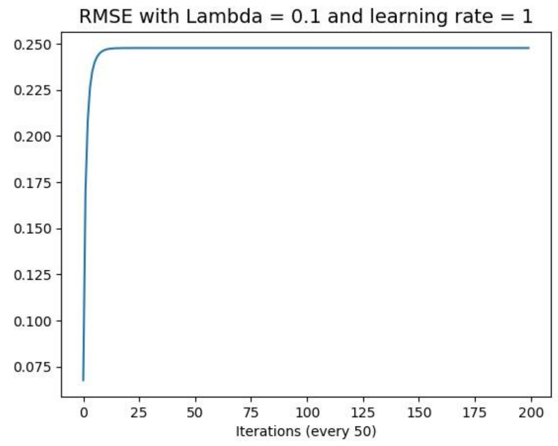


*Figure 11 - RMSE measured during training for lambda=0.1 and learning rate=1*

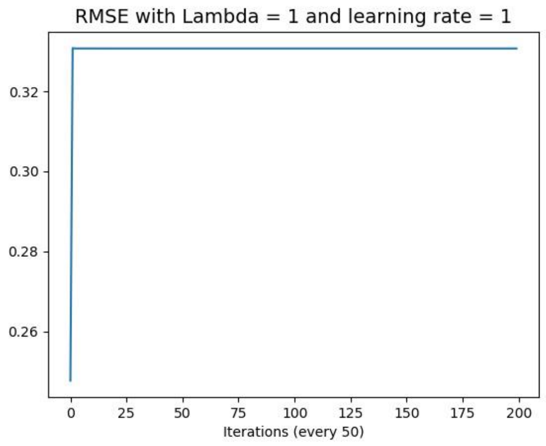


*Figure 12 - RMSE measured during training for lambda=1 and learning rate=1*

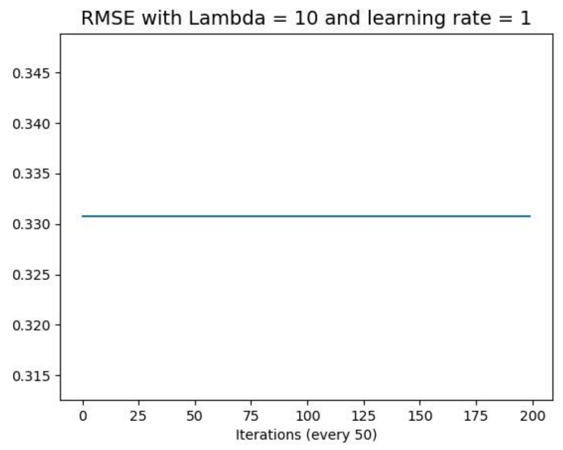


*Figure 13 - RMSE measured during training for lambda=10 and learning rate=1*

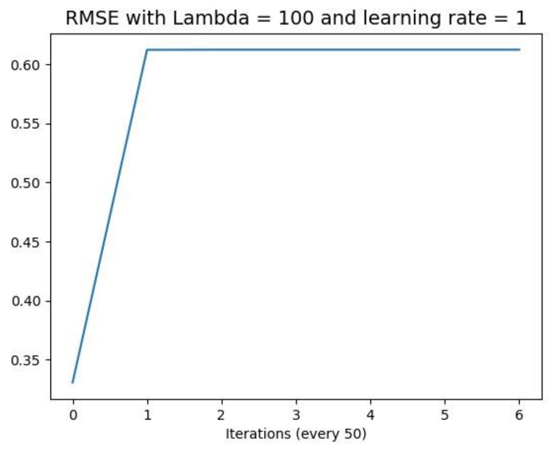


*Figure 14 - RMSE measured during training for lambda=100 and learning rate=1*

As it can be observed from the figures above, higher value of lambda is disadvantageous for the learning process. Since not very complex network has been implemented for the purposes of this assignment, it is not seriously prone to overfitting. Therefore, regularization process, which makes use the lambda parameter, is not very useful here. On the contrary, regularization of the learned weights may considerably harm the performance of this neural network, and prevent the learning process from convergence, which in particular can be noticed in Figures 9-14.

It is worth mentioning that Figures 4 and 8 both present RMSE measured every 50 iterations for learning rate and lambda set to 1 and 0.0001 respectively. Nevertheless, the corresponding plots are different. This comes from the fact, that these 2 set of plots (figures 1-6 and figures 7-14) come from separate runs of the program. Therefore, different initial weights were (randomly) generated before running the gradient descent algorithm and thus other weights were received during and after the training process.

Final set of experiments included comparisons of final RMSE values after the training has been finished, for varying values of selected hyper-parameter. Figures 15-17 present comparisons for different values of learning rate, lambda and number of iterations respectively. For better clarity of the presented plots, logarithmic scale over the x-axis has been applied.

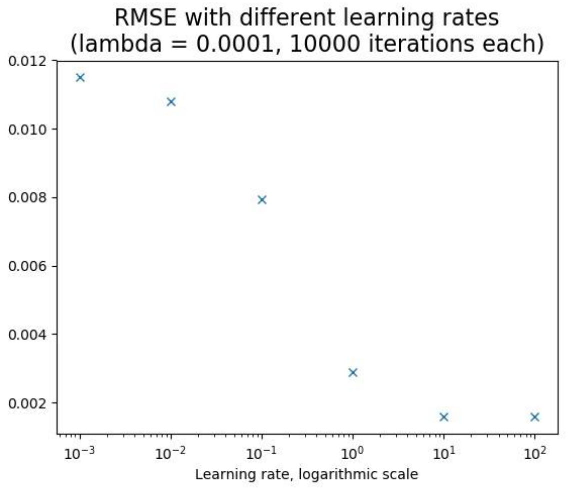


*Figure 15- Final values of RMSE for different learning rate values*

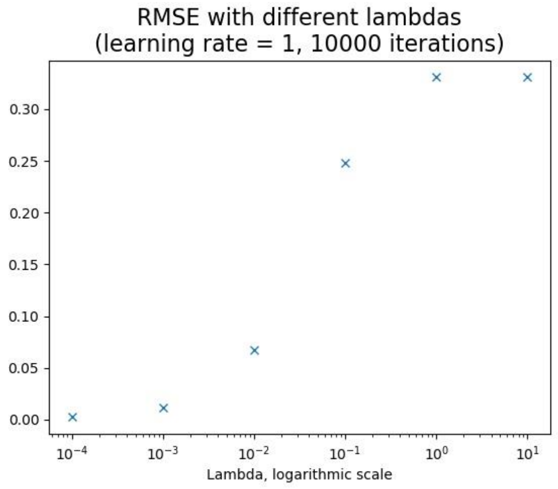


*Figure 16 - Final values of RMSE for different lambda values*

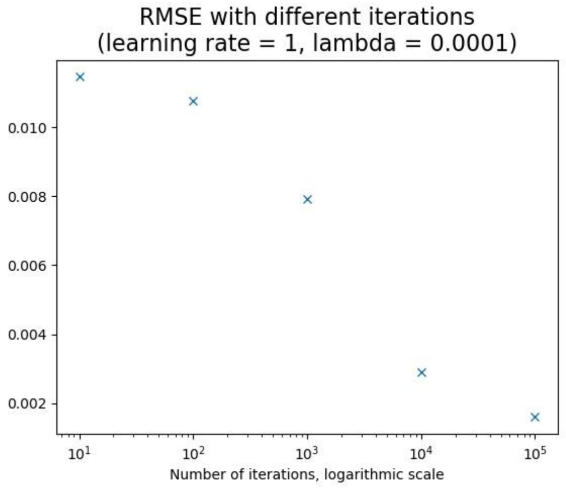
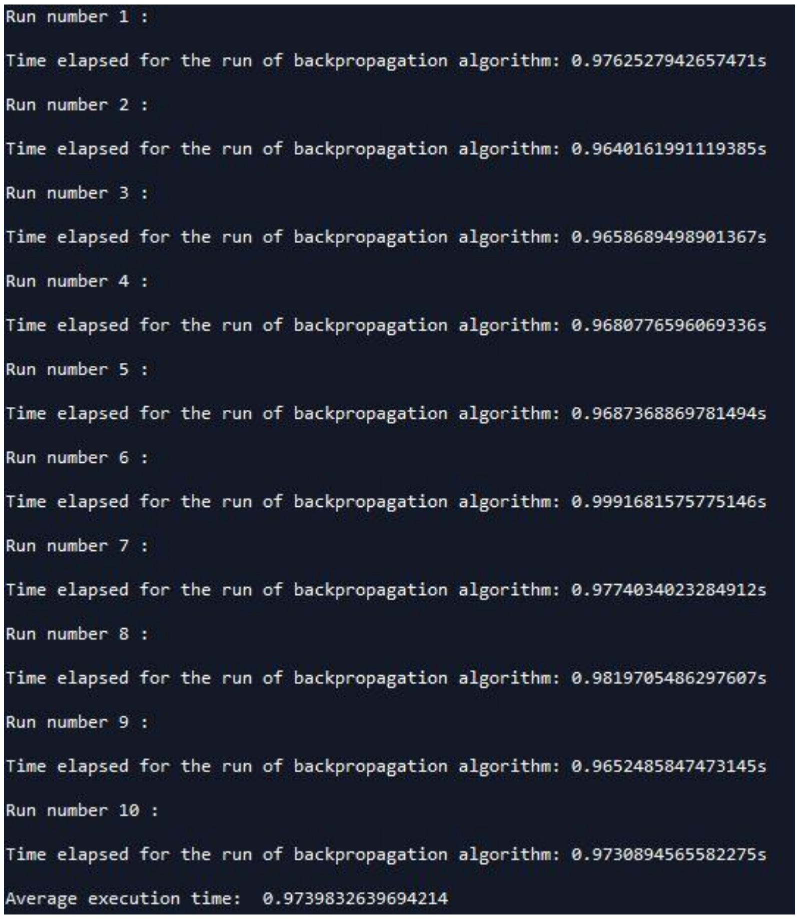


*Figure 17 - Final values of RMSE for different number of iterations values*

As it can be observed from the figures above, higher learning rate is generally advantageous for the learning process (it cannot be too high though, as already discussed). Higher number of iterations also seems beneficial there. The regularization parameter (lambda) values on the other hand helps the learning process the most at its lowest values.

## Average Execution Time

The following figure shows the execution time of each run 10 times. Then it shows the average execution time of these runs. We can see that on average, it takes a little bit less than 1 second to run the algorithm using the hyper-parameters cited above.



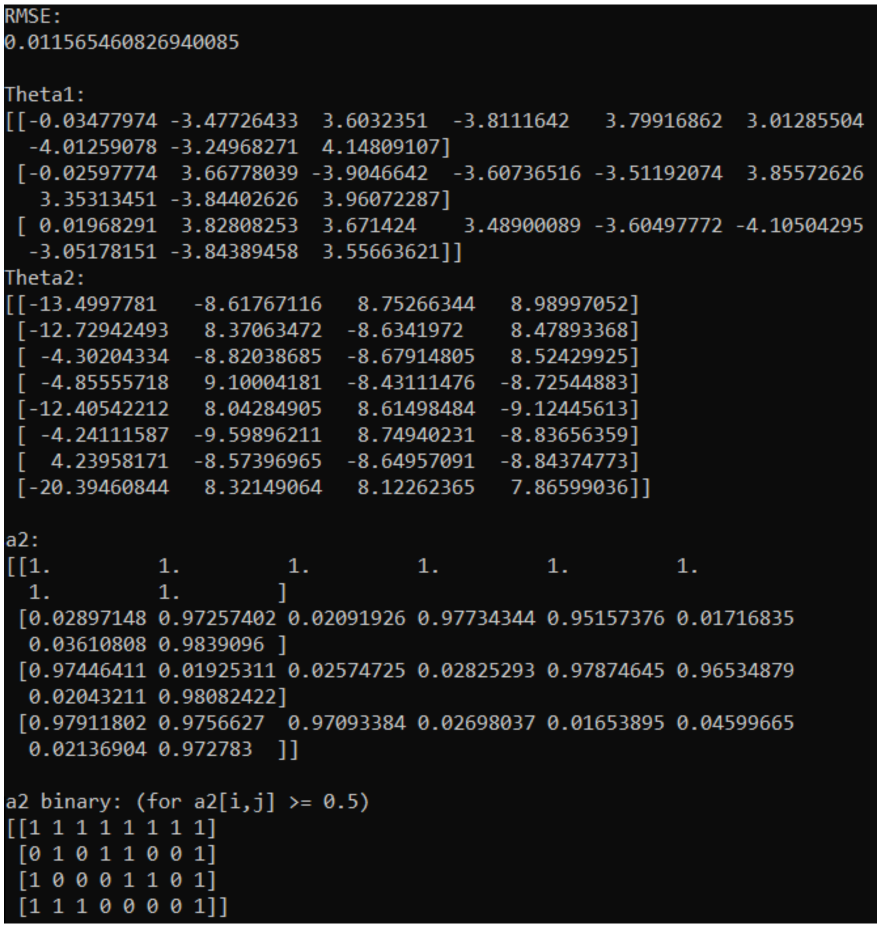
```
Run number 1 :

Time elapsed for the run of backpropagation algorithm: 0.9762527942657471s

Run number 2 :

Time elapsed for the run of backpropagation algorithm: 0.9640161991119385s

Run number 3 :

Time elapsed for the run of backpropagation algorithm: 0.9658689498901367s

Run number 4 :

Time elapsed for the run of backpropagation algorithm: 0.9680776596069336s

Run number 5 :

Time elapsed for the run of backpropagation algorithm: 0.9687368869781494s

Run number 6 :

Time elapsed for the run of backpropagation algorithm: 0.9991681575775146s

Run number 7 :

Time elapsed for the run of backpropagation algorithm: 0.9774034023284912s

Run number 8 :

Time elapsed for the run of backpropagation algorithm: 0.9819705486297607s

Run number 9 :

Time elapsed for the run of backpropagation algorithm: 0.9652485847473145s

Run number 10 :

Time elapsed for the run of backpropagation algorithm: 0.9730894565582275s

Average execution time:  0.9739832639694214
```

*Figure 18 - Information on execution time*

## Learned Weights Interpretation

The only input provided to the neural network for training were 8 8-element-vectors of the form [1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], …, [0,0,0,0,0,0,0,1]. For each training sample, the expected output was exactly the same as the given input. The values of learned weights were such that the difference between the output reconstructed from input (referred as predicted output) and expected output was minimized. In addition to the input and output layers, the implemented neural network also contains a hidden layer consisting of 3 nodes (plus a bias node). Using the weights from first layer, by multiplying their representative matrix with the input matrix, initial values for each input at each hidden node (excluding the bias node) were received. Then, by transforming the results independently via sigmoid function, the activation values at hidden nodes were obtained. Analogous process was performed using the newly computed activations (with bias unit included) and the matrix representing the weights from second layer.

The learned weights enabled a certain format of hidden layer node activations which contributed to reconstruction of the input as predicted output. A concrete example is presented in figure 18.



```
RMSE:
0.011565460826940085

Theta1:
[[-0.03477974 -3.47726433  3.6032351  -3.8111642   3.79916862  3.01285504
  -4.01259078 -3.24968271  4.14809107]
 [-0.02597774  3.66778039 -3.9046642  -3.60736516 -3.51192074  3.85572626
   3.35313451 -3.84402626  3.96072287]
 [ 0.01968291  3.82808253  3.671424    3.48900089 -3.60497772 -4.10504295
  -3.05178151 -3.84389458  3.55663621]]
Theta2:
[[-13.4997781   -8.61767116   8.75266344   8.98997052]
 [-12.72942493   8.37063472  -8.6341972    8.47893368]
 [ -4.30204334  -8.82038685  -8.67914805   8.52429925]
 [ -4.85555718   9.10004181  -8.43111476  -8.72544883]
 [-12.40542212   8.04284905   8.61498484  -9.12445613]
 [ -4.24111587  -9.59896211   8.74940231  -8.83656359]
 [  4.23958171  -8.57396965  -8.64957091  -8.84374773]
 [-20.39460844   8.32149064   8.12262365   7.86599036]]

a2:
[[1.         1.         1.         1.         1.         1.
  1.         1.        ]
 [0.02897148 0.97257402 0.02091926 0.97734344 0.95157376 0.01716835
  0.03610808 0.9839096 ]
 [0.97446411 0.01925311 0.02574725 0.02825293 0.97874645 0.96534879
  0.02043211 0.98082422]
 [0.97911802 0.9756627  0.97093384 0.02698037 0.01653895 0.04599665
  0.02136904 0.972783  ]]

a2 binary: (for a2[i,j] >= 0.5)
[[1 1 1 1 1 1 1 1]
 [0 1 0 1 1 0 0 1]
 [1 0 0 0 1 1 0 1]
 [1 1 1 0 0 0 0 1]]
```

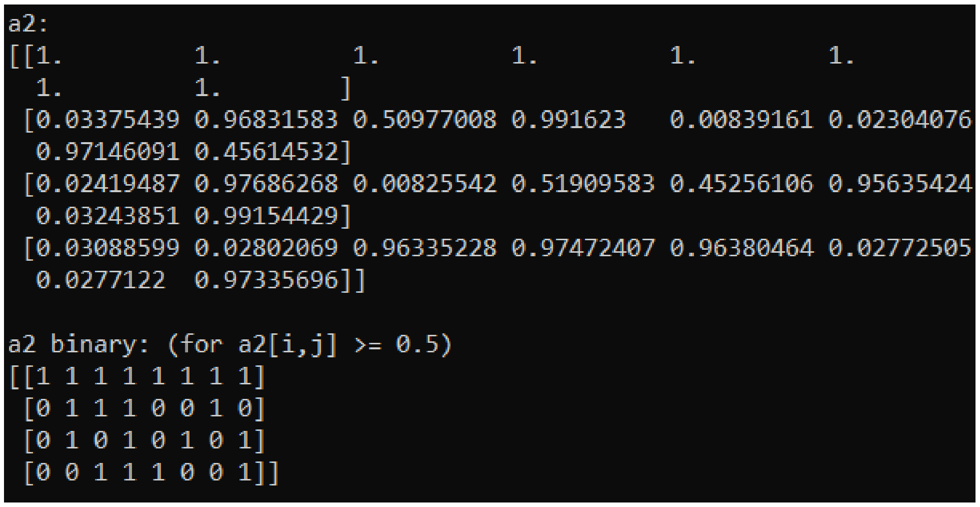
*Figure 19 - Exemplary (trained) weights and corresponding hidden layer activations*

The activation values at hidden layer ("a2") for each input are close to 0 or 1 (values of exactly 1 represent the bias unit). After approximation to binary format ("a2 binary"), it can be observed that for each training sample (i.e. for each column), these values represent different sequences of 0 and 1 digits. Furthermore, when 1s coming from bias units are skipped, it can be seen that each sequence corresponds to a (distinct) binary representation of the numbers from 0 to 7.

The neural network “does not know” what 0 or 1 are. It aims to use them (or to be more precise, values close to them) as hidden representation which combined with the learned weights ("Theta1" and "Theta2") will enable already mentioned input reconstruction (to a certain degree though, since measured error has never been 0).

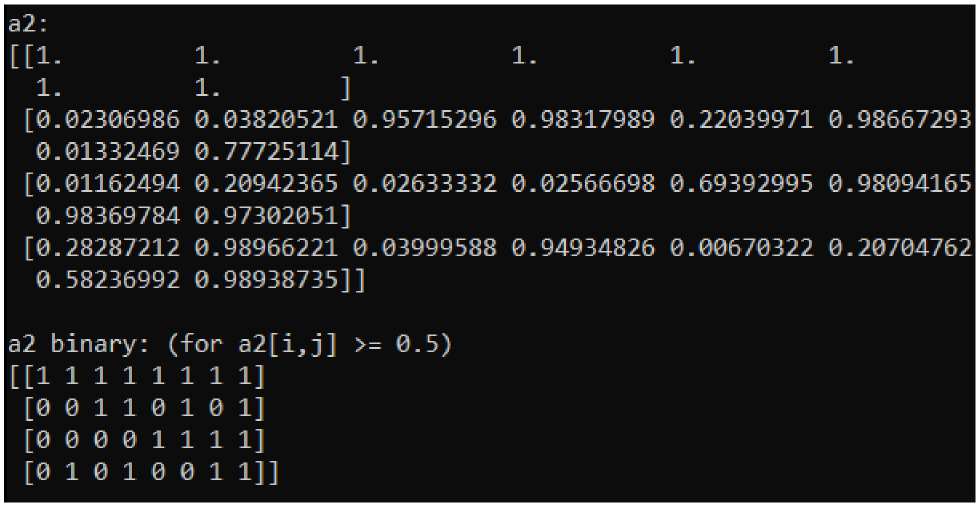
What is worth mentioning here is the fact, that in order to represent eight different binary values, exactly three digits are needed. Therefore, in can be assumed that the network tries to create identification encodings for each of eight distinct inputs, aiming at their reconstruction at the output layer. The whole process involves the learned weights.

An important fact is that since the weights are randomly initialized, their final values after training will be different per each run of the program. Therefore, the activation values of hidden layer units will also be different each time, which implies that the binary sequences representing numbers from 0 to 7 can be reached in different order for the same set of training samples. Nevertheless, each of these 8 sequences will always be present (without repetition). Three different runs with three different "a2 binary" matrix values are shown in figures 20a-c.



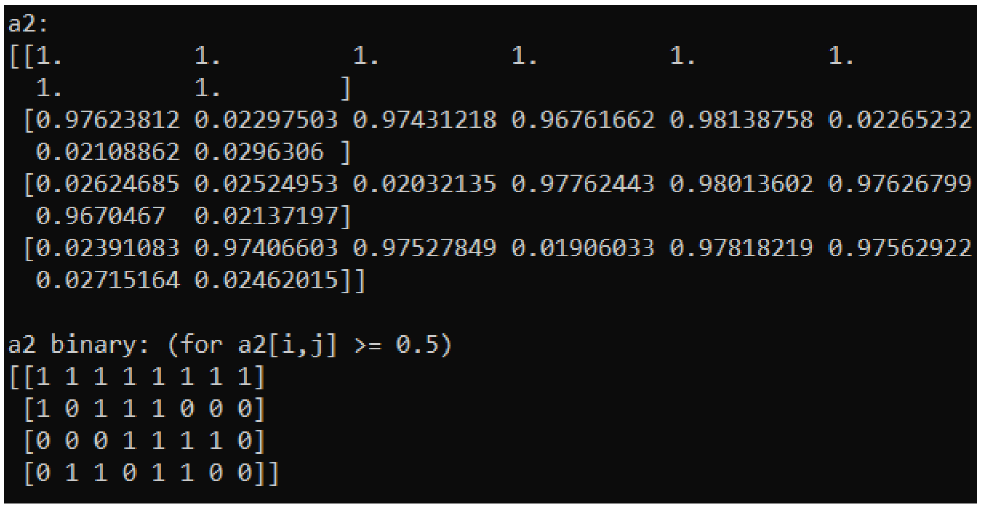
```
a2:
[[1.         1.         1.         1.         1.         1.
  1.         1.        ]
 [0.03375439 0.96831583 0.50977008 0.991623   0.00839161 0.02304076
  0.97146091 0.45614532]
 [0.02419487 0.97686268 0.00825542 0.51909583 0.45256106 0.95635424
  0.03243851 0.99154429]
 [0.03088599 0.02802069 0.96335228 0.97472407 0.96380464 0.02772505
  0.0277122  0.97335696]]

a2 binary: (for a2[i,j] >= 0.5)
[[1 1 1 1 1 1 1 1]
 [0 1 1 1 0 0 1 0]
 [0 1 0 1 0 1 0 1]
 [0 0 1 1 1 0 0 1]]
```

*Figure 20a - Examplary hidden layer activation values and their binary approximations (1)*



```
a2:
[[1.         1.         1.         1.         1.         1.
  1.         1.        ]
 [0.02306986 0.03820521 0.95715296 0.98317989 0.22039971 0.98667293
  0.01332469 0.77725114]
 [0.01162494 0.20942365 0.02633332 0.02566698 0.69392995 0.98094165
  0.98369784 0.97302051]
 [0.28287212 0.98966221 0.03999588 0.94934826 0.00670322 0.20704762
  0.58236992 0.98938735]]

a2 binary: (for a2[i,j] >= 0.5)
[[1 1 1 1 1 1 1 1]
 [0 0 1 1 0 1 0 1]
 [0 0 0 0 1 1 1 1]
 [0 1 0 1 0 0 1 1]]
```

*Figure 20b - Examplary hidden layer activation values and their binary approximations (2)*



```
a2:
[[1.         1.         1.         1.         1.         1.
  1.         1.        ]
 [0.97623812 0.02297503 0.97431218 0.96761662 0.98138758 0.02265232
  0.02108862 0.0296306 ]
 [0.02624685 0.02524953 0.02032135 0.97762443 0.98013602 0.97626799
  0.9670467  0.02137197]
 [0.02391083 0.97406603 0.97527849 0.01906033 0.97818219 0.97562922
  0.02715164 0.02462015]]

a2 binary: (for a2[i,j] >= 0.5)
[[1 1 1 1 1 1 1 1]
 [1 0 1 1 1 0 0 0]
 [0 0 0 1 1 1 1 0]
 [0 1 1 0 1 1 0 0]]
```

*Figure 20c - Examplary hidden layer activation values and their binary approximations (2)*