COSC 420
Project 4
April 7th, 2017

Artificial Neural Net Learning:
Implementation of Back-Propagation Software System

Abstract: In this project, the goal is to implement and examine feed-forward artificial neural networks and explore back-propagation learning algorithm. Neural networks will be trained on several data sets for solving several problems: XOR, Aproximation1, Aproximation2. We will observe, how different parameters for building networks, like number of hidden layers, number of nodes (neurons), number of epochs and learning rate will affect training, validation and testing performance.

1. Introduction

Feed forward artificial networks have such property that no sequence of connections among neurons forms a cycle, so basically, no neuron can feed back to itself. Thus information, flows only one direction, from input neurons all the way to the output nodes. Perceptron, pattern classification device, is a great example of feed-forward neural net.

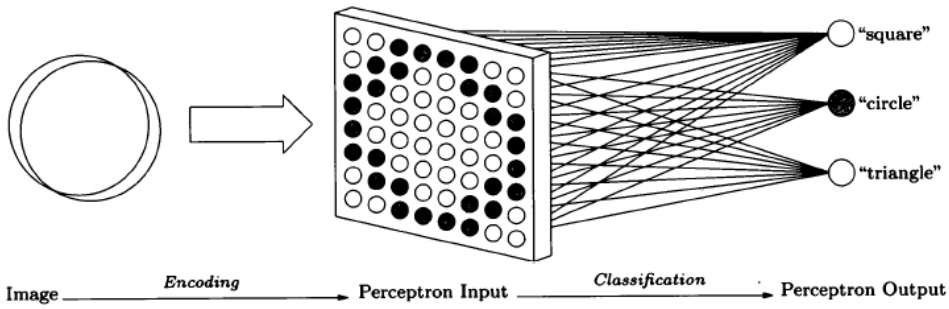


Figure 1. A stylized picture of perceptron. "The Computational Beauty of Nature"

Perceptron demonstrates how image encoding and classification goes one direction. here are single-layer and multiple-layer perceptron kinds of neural network. Single-layer one have inputs and outputs only, whereas multiple-layer also have one or more hidden layers in between.

2. Back-Propagation Algorithm

Back-propagation algorithm is the most commonly used for training multi-layer neural networks. The general idea behind this algorithm is run several data sets, compare output with expected value and compute the predefined error function. Then error function is fed back through the network and used for adjustment of all the weights to minimize the value of the error function.

Such operation is done several number of times until error is minimized to almost zero value, which indicates that network is trained and should solve testing problems correctly.

So, as it's been said, first we propagate input values forward through the network.

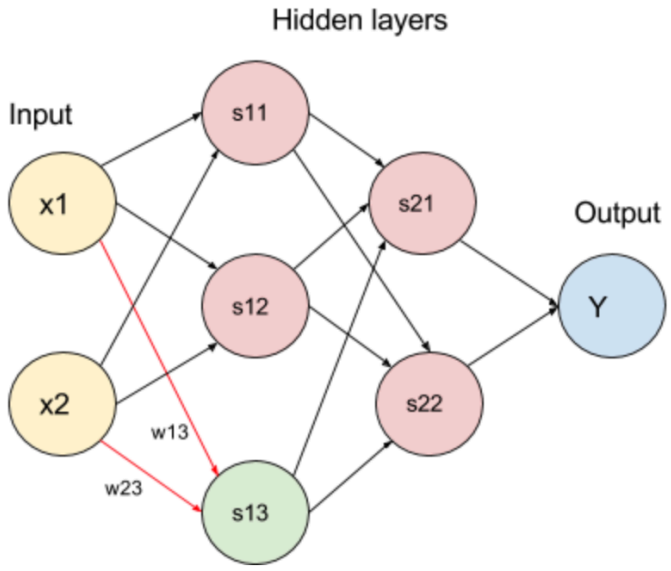


Figure 2. Feed forward propagation.

In figure 3, we start with two inputs and propagate through two hidden layers before calculating the output. We calculate sigmoidal function $g(x) = \frac{1}{1+\exp(-x)}$ value for each node, where x is determined by summation of products of all weights (coming into a node) and s_{ij} values from previous layer corresponding to those weights. We also add bias weight to it for weigh correction. So, we get $x_k = \sum_{i=0}^{prev\ nodes\ \#} s_{ij} w_{ik} + b_k$. So, in our example, we do x = $x_1 w_{13} + x_2 w_{23}$ + bias weight. And then calculate $g(x)$ for node colored in green, which is sigma value that is used in calculations for nodes in next layer.

After output is computed, we calculate the error function δ value. First layer in the end will take in account the difference between expected output and received one. And for every other node, we calculate $\delta_{s_{ij}} = g'(x_{ij}) \sum_k w_{kj} \delta_{s_{(i+1),j}}$. As we can see, delta function is from layer before is used for calculation of delta function for current node. Calculating the derivative of the sigmoidal function gives us information about how the output changes when the input changes. If its value is too large that means the input was near its threshold and little change may change $g(x)$, if it's too small, then value of $g(x)$ is really close to 0 or 1. Also, the sign of $g'(x)(t-y)$ will tell us which way neuron was wrong. Overall, this term will give us all the information needed for error correction.

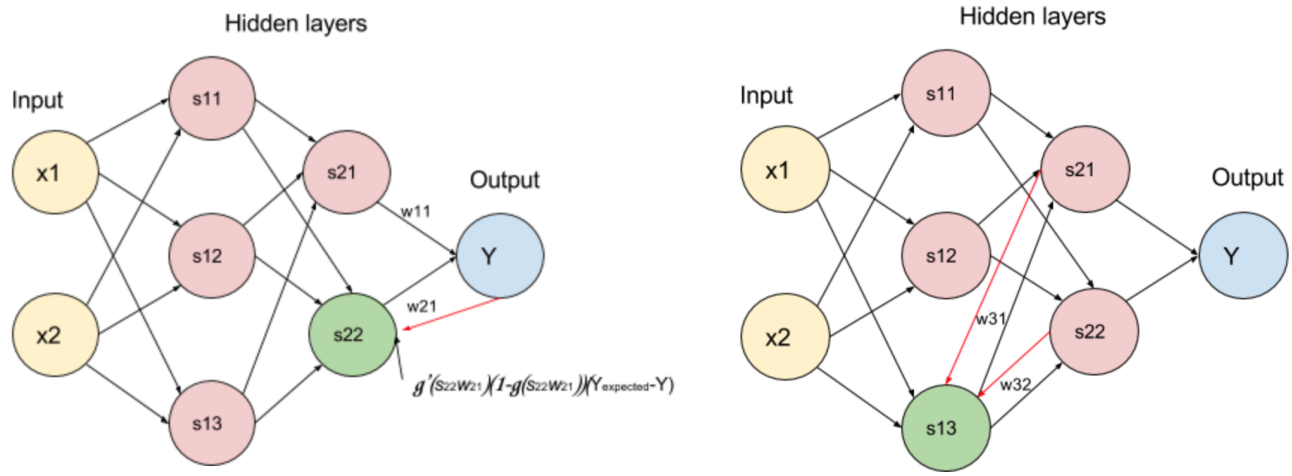


Figure 3. Back propagation start and moving one layer back.

We use delta function and learning rate value for computing adjusted weight.

3. Experiment Software

To conduct the experiment, I've written a simple program in Java programming language that takes 3 data set filenames as command line arguments and then asks user for network values.

```
Provide with: [problem name string][inputs number] [layers number] [epochs number] [learning rate]
Separate values by space
Aprox2 3 1 5000 0.7
type nodes number for each layer:

Layer i:
14
0.014225862420225823
```

Figure 4. Program run

In the program, I calculate data that goes into .csv files named after parameters chosen so it is easier to indicate which experiment was run by just looking at the file name.

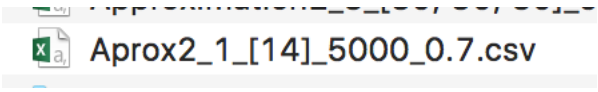


Figure 5. File name specifics

There are three phases in the program and experiment itself: network training, network validation and network testing. We set up a number of epochs (error correction iterations) for training and validation. For each epoch we train network on training data set, and then we validate the network using another dataset of 100 values by measuring the root mean square error. With the number of epochs, error value should decrease if network is training correctly. After all epochs we test the network using testing dataset. I decided to compare resulted value against expected one $\pm$(0.01+RMSE). If difference was less or equal then 0.01+RMSE, problem was solved correctly.

During network training, we read 200 sets of inputs and expected outputs for the problem. For validation part, dataset has 100 lines of information. Finally, testing data has 50 values.

4. Experiment Flow and Problem

There were there problems I approached. First one was traditional XOR problem, that I've done for simply for testing my software. It also helped me to observe what parameters to look at when adjusting program input values for the problem. The, there were two approximation problems done.

4.1. XOR

Exclusive-or is a logical operation that outputs *true(1)* only when input values differ. In this problem, I generated my own datasets for 2 inputs and 1 output. I tried multiple values for all: epochs, number of layers and nodes within, and learning rates (LR).

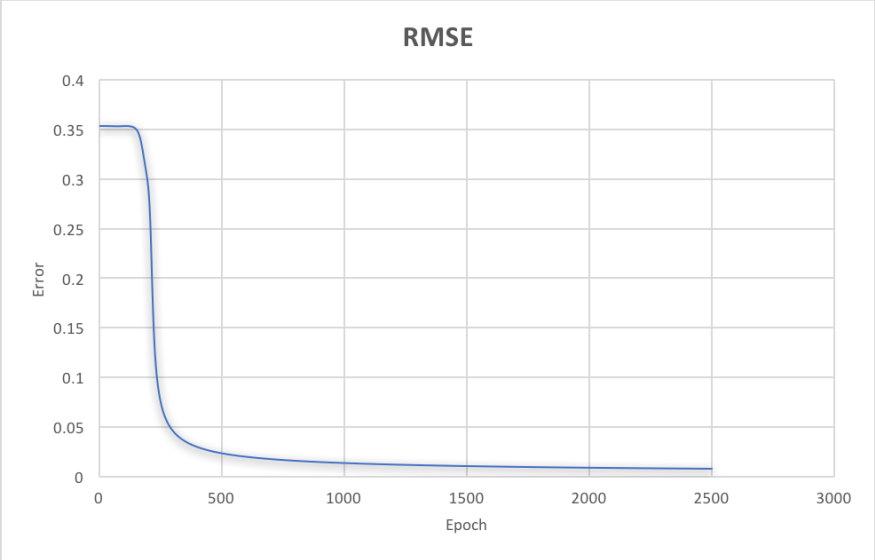


Figure 6. RMSE graph for XOR problem with 2500 epochs, 0.1 LR and 1 hidden layer with 3 nodes

We can observe, how error drops around 250th epoch and then keeps becoming less. Network received from this configuration (2500 epochs, 0.1 LR and 1 hidden layer with 3 nodes). I've also tried high learning rates like 0.8 and 1.2, then added more layers, but seems like this problem gets solved easily with all sorts of parameters. Let's look at accuracy. Figure 7 demonstrates computed values, we can see that for all the expected 1s we received >= 0.99, whereas for all expected 0s, we received <= 0.001.

-----	testing network	----
Expected	Computed real	Is solved
1	0.998407236	yes
0	0.001098682	yes
0	0.001698484	yes
1	0.998407236	yes
1	0.998369267	yes
0	0.001098682	yes
1	0.998369267	yes
0	0.001698484	yes
1	0.998407236	yes
1	0.998407236	yes
0	0.001698484	yes
0	0.001098682	yes
0	0.001698484	yes
1	0.998407236	yes
1	0.998369267	yes
1	0.998369267	yes
1	0.998407236	yes
0	0.001098682	yes
0	0.001098682	yes

Figure 7. XOR computed values

4.2. Approximation 1

First approximation problem is following:

$$f(x,y) = \frac{1+\sin(\pi x/2)\cos(\pi y/2)}{2}, \quad x \in (-2,2), y \in (-2,2).$$

All three data sets were provided. I started with 1 layer with 15 nodes. At lower learning rate (0.1) I received pretty big MRSE of 0.4 after 2000 epochs.

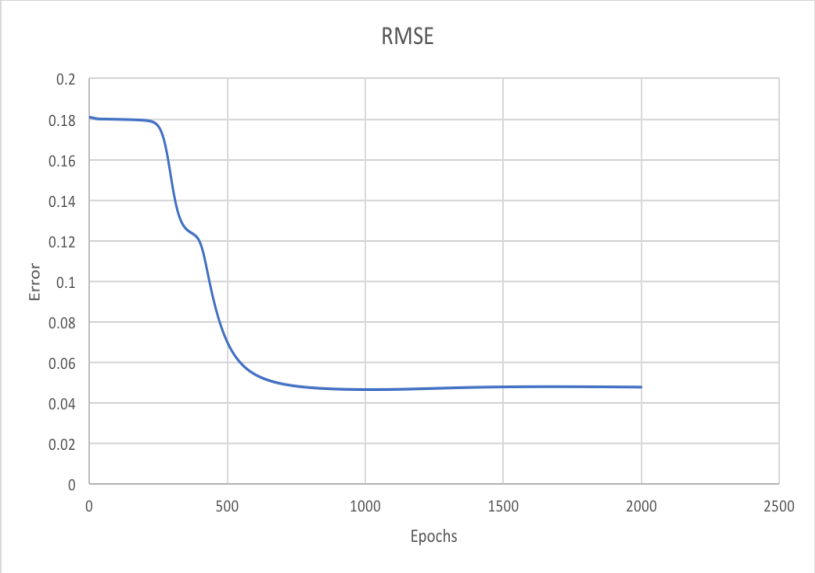


Figure 8. Approximation 1. 1 layer

In testing phase, only 66% problems got solved correctly. We can observe a very slight increase in error after 1500 epoch which suggests that we over fit the network. Next, I decided to increase learning rate up to 0.8 along with number epochs increased to 10000. It dropped RMSE error by almost 0.02 and bumped correctness percentage up to 78%.

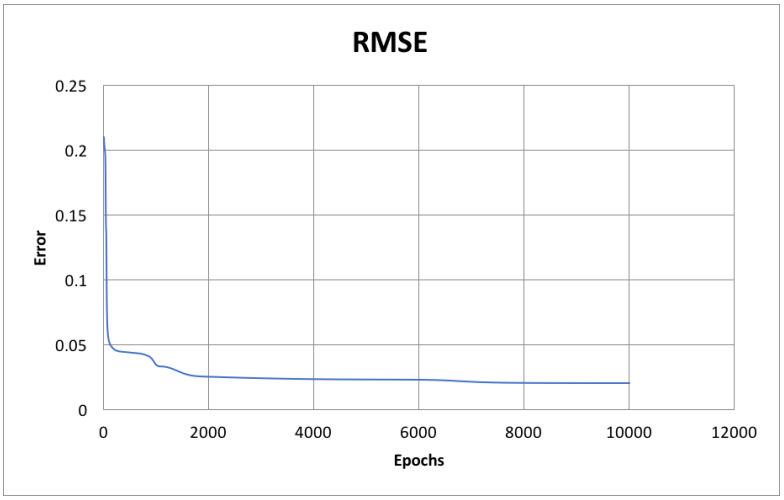


Figure 9. Approximation 1. 1 layer

We can observe, how error keeps lowering at this stage. Next, I increased number of nodes for the only hidden layer to 33. For different epoch and LR values I received the following results in error:

1. Epoch 15000, LR 0.4 – RMSE 0.014, Accuracy 86%
2. Epoch 15000, LR 0.8 – RMSE 0.021, Accuracy 76%
3. Epoch 50000, LR 0.3 – RMSE 0.013, Accuracy 84%

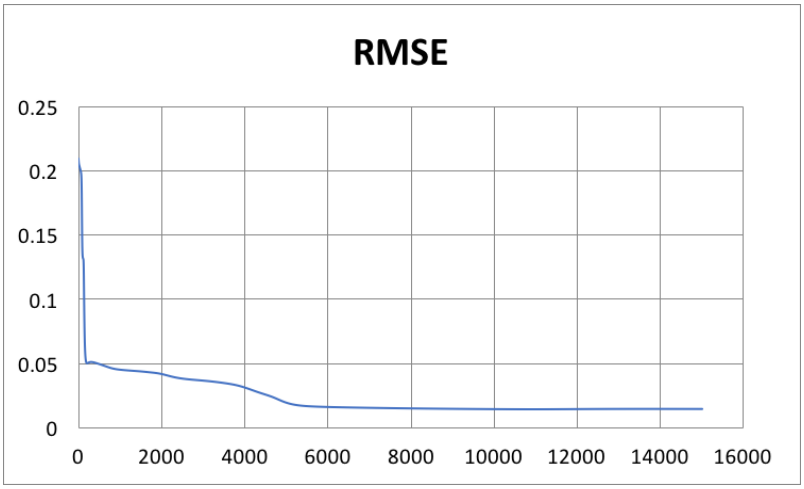


Figure 10. Approximation 1. 1 layer

This graph demonstrates network that has 86% accuracy.
Next, I increased number of layers to 2 and received following data:

1. Nodes number 10 10, epoch 2500, LR 0.1 – RMSE 0.04, Accuracy 70%
2. Nodes number 10 10, epoch10000, LR 0.1 – RMSE 0.025, Accuracy 80%
3. Nodes number 16 16, epoch 10000, LR 0.3 – RMSE 0.04, Accuracy 86%
4. Nodes number 16 16, epoch 2500, LR 0.7 – RMSE 0.012, Accuracy 78%
5. Nodes number 16 16, epoch 50000, LR 0.7 – RMSE 0.009, Accuracy 84%
6. Nodes number 20 20, epoch 10000, LR 0.4 – RMSE 0.013, Accuracy 82%
7. Nodes number 20 20, epoch 50000, LR 0.9 – RMSE 0.019, Accuracy 96%
8. Nodes number 20 20, epoch 100000, LR 0.4 – RMSE 0.008, Accuracy 90%

As we can see form these results, lowest error doesn't guarantee highest accuracy rate. For example, 5 and 7 have similar parameters, same number of epochs. RMSE Error in #5 is much better, but accuracy percentage is 14% lower. Thus, higher learning with higher number of nodes in layers and with higher epoch may boost network accuracy.

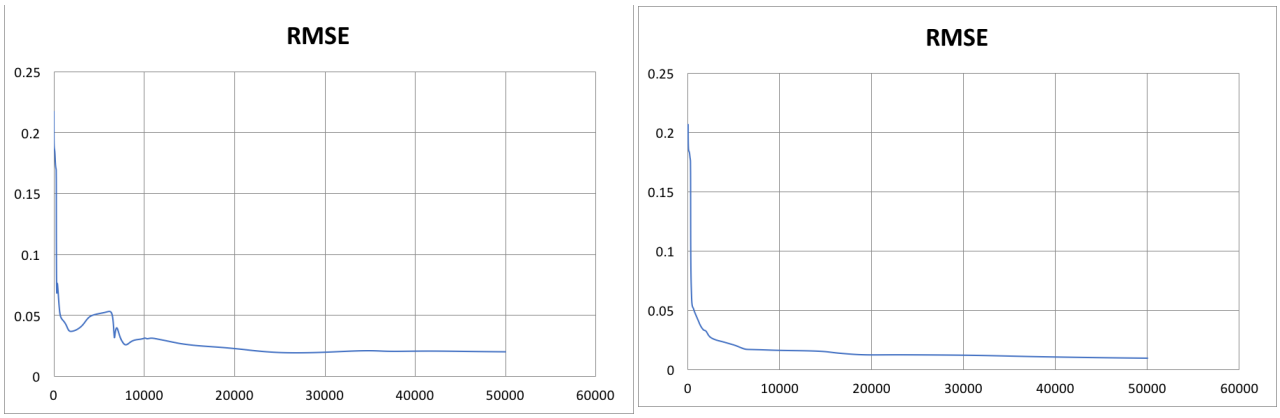


Figure 11.Approximation1. Higher LR and number of Nodes vs. Lower LR and number of nodes

We also can see how first graph for network with better accuracy is inconsistent in error change vs. the other network with lower accuracy percentage.

Finally, I observed 3 layers architectures:

1. Nodes number 10 10 10, epoch 15000, LR 0.8 – RMSE 0.019, Accuracy 84%
2. Nodes number 32 32 31, epoch15000, LR 0.4 – RMSE 0.04, Accuracy 82%

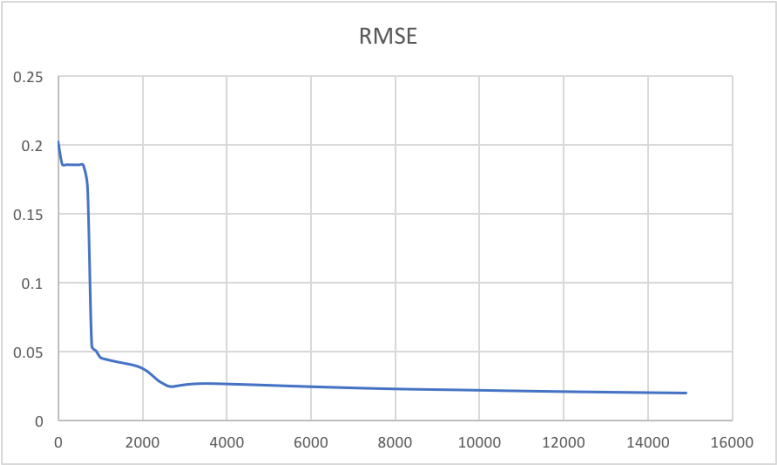


Figure 12. Approximation1. 3 layers

Results were similar for small nodes number and for bigger number. Therefore, the most optimal results were achieved with 2 layer architecture after 50000 epochs with higher LR.

4.3. Approximation 2

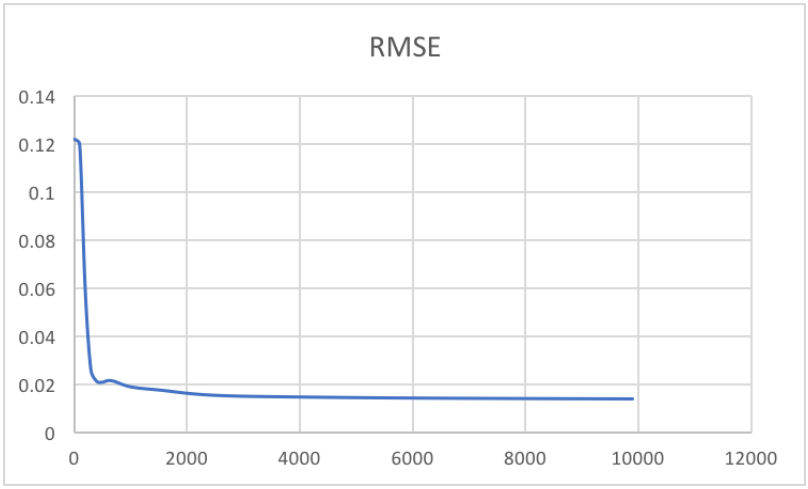
$$f(x,y,z) = \frac{3}{13}\left[\frac{x^2}{2} + \frac{y^2}{3} + \frac{z^2}{4}\right], \quad x \in (-2,2), y \in (-2,2), z \in (-2,2).$$

There were more inputs in following problem. We had 3 inputs x, y, z and one output. Because this problem was very similar to previous one, I tried to pay more attention to parameters that had more nodes in hidden layers along with high/medium LR and larger number of epochs.

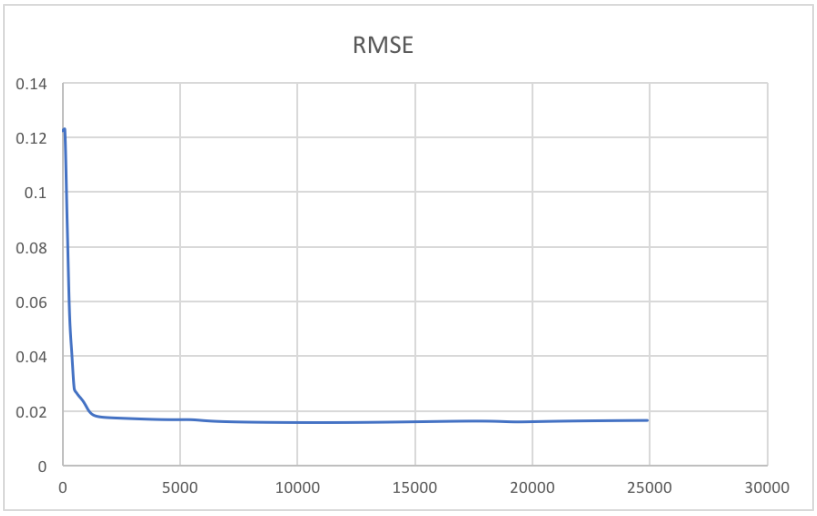
I still started with 1 hidden layer architecture, and received following results:

1. Nodes number 6, epoch 25000, LR 0.3 – RMSE 0.016, Accuracy 92%
2. Nodes number 11, epoch15000, LR 0.45 – RMSE 0.010, Accuracy 80%
3. Nodes number 11, epoch 40000, LR 0.45 – RMSE 0.0099, Accuracy 84%
4. Nodes number 12, epoch10000, LR 0.3 – RMSE 0.013, Accuracy 88%
5. Nodes number 12, epoch 50000, LR 0.4 – RMSE 0.012, Accuracy 88%
6. Nodes number 12, epoch 50000, LR 0.8 – RMSE 0.0097, Accuracy 82%
7. Nodes number 14, epoch 5000, LR 0.7 – RMSE 0.011, Accuracy 98%
8. Nodes number 14, epoch 10000, LR 0.8 – RMSE 0.011, Accuracy 88%

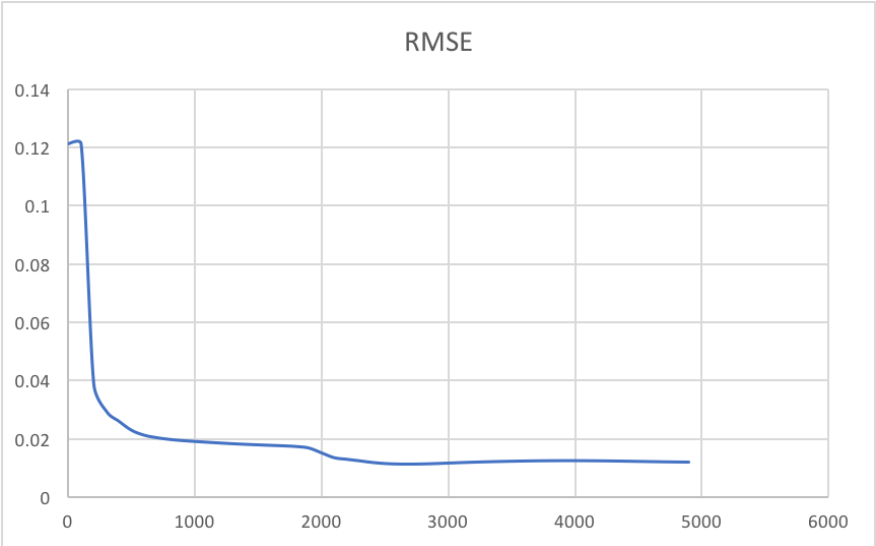
Following chart demonstrates an error for first parameter set where accuracy is 92%. Number of epochs is 25000, but it seems to be pretty stable.



We lowered number of epochs here, but encreased number of nodes and performance seems to be more stable, but accuracy rate is lower.



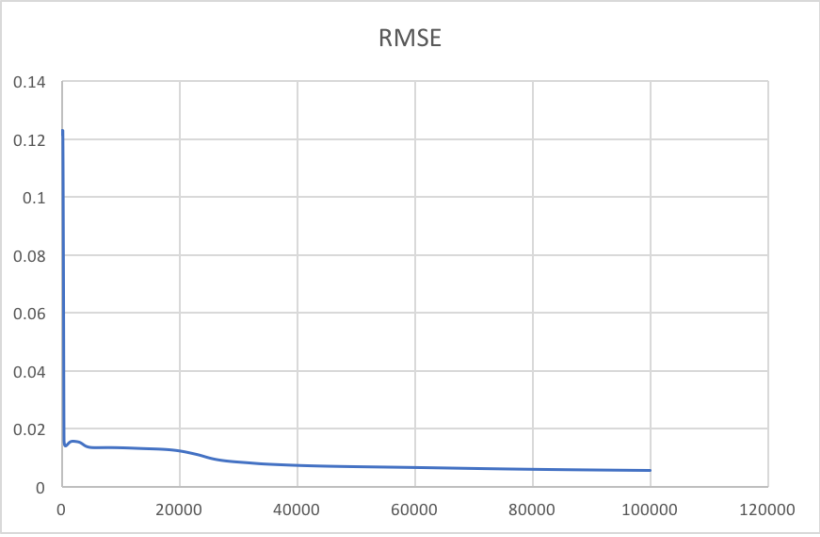
Finally, we encreased LR and number of nodes and epochs and hit 98%. Out 0f 50 problems only one was solved incorrectely. But the answer was only 0.3 off. The graph is below, we can see how error keeps dropping with number of epochs.



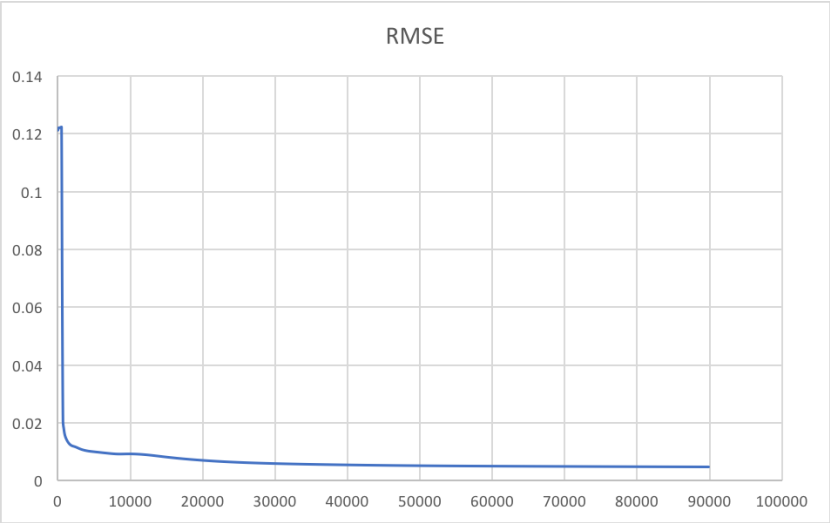
In 2 layer structures, the picture was following:

1. Nodes number 12 12, epoch10000, LR 0.7 – RMSE 0.010, Accuracy 80%
2. Nodes number 12 12, epoch 100000, LR 0.7 – RMSE 0.0056, Accuracy 98%
3. Nodes number 12 12, epoch 120000, LR 0.7 – RMSE 0.0053, Accuracy 98%

After changing value for a while a realized, that the most optimal was to have 12 nodes in each layer and have a LR of 0.7. I ran 10000 and it went well. I increased number of epochs up to 100,000 and achieved 98% accuracy. With 120,000 epochs error dropped a bit but accuracy stayed the same. I went for 150,000 epochs and there was over fit. So, 100,000 epochs was optimal to bring accuracy to almost perfect score. Graph below illustrates how error kept decreasing.



And I tried several 3 layer architectures, the best I was able to achieve was with 30,30,30 structure after 90,000 epochs and learning rate 0.83. It was the lowest error received in the whole project – 0.004. But result in accuracy was different, only 94%. The graph is below.



We can observe how error decreases smoothly.

5. Conclusion

Overall, in this project, there were 3 problems examined with training neural networks using back propagation algorithm. After doing over 100 of experiments with parameters switching, I realized that higher LR and larger number of epochs increase a chance for higher accuracy in problem testing. Also, sometimes it depends in number of nodes in architectures. For 1 layer, optimal number is about 12. In 2 layer is about 16, and 3 layer is about 20.