University of Essex
School of Computer Science and Electronic Engineering

Assignment Report

Submitted as part of the requirements for:

CE889 Neural Networks and Deep Learning

Name: Ogulcan Ozer

Tutor: Prof. Hani Hagras

Date: 06 January 2019

Word Count:4081

Table of Contents

1 Literature Review

1 Robot Navigation Using Neural Networks

Mobile robots have been used in different applications such as exploring places that are dangerous for humans, carrying heavy loads in factories etc. And in these environments, obstacle avoidance, goal orientation and navigation in general, has the biggest importance for the safety of the robot and the safety of the people that are working in the same place with the robot.

A well trained neural network can provide this autonomy to a mobile robot. One of the early works [1], shows that a neural network can be trained in a reactive manner to respond to its environment in real-time using its sensors. [1] states that a neural network for navigation can be trained by providing different samples of desired actions, but it is also mentioned that in this way even though the robot can navigate, it displays oscillatory behaviour. And It is claimed that the reason for this behaviour is the insensitivity of the network to its past actions. Therefore, in [1], the network is modified similar to a recurrent neural network that also receives past readings as its inputs in the training process, which prevents the oscillations caused by the pure reactionary behaviour.

Another example is [2], which follows a more cautious approach. In this study, two neural networks and a principal component analysis(PCA) layer is used. First, the robot uses all its sensors to gather data about the environment. This data is passed through the PCA layer then fed to the first neural network to train it to find the free space for the robot to move. Then, [2] states that, the output of the first neural network is used together with goal information and fed to the second network for robot navigation. It is concluded that with this approach, the robot can navigate in known and unknown environments while avoiding both static and dynamic obstacles.

Lastly a more interesting study lets the robot learn by itself. [3] uses a neural network for navigation, and genetic algorithms to train the neural network. The genetic algorithm uses a fitness function that is a function of three variables, V: the average rotation speed of two wheels, deltaV: the difference between the speeds of the wheels and i: the activation value of the sensor with closest reading. The genetic algorithm works in real time and trains the network with its output. In the end [3] concludes that, a robot around 50^{th} generation performs optimally, displaying very smooth movement and perfectly avoiding any obstacle while continuing its movement in the environment.

2 Deep Neural Networks

Concept of deep learning has been around since 1940s, and it`s been renamed many times because of the influences of different researchers. [4] The reason it was not as popular like a classical multilayer perceptron with one hidden layer is because there were some problems like vanishing gradient, lack of computational power and the availability of big data for the training. But in the recent past, these problems were addressed, which proved that deep learning is a great machine learning tool for solving complex problems. As stated in [5], most machine learning techniques requires careful data processing, feature selection and expert knowledge. Because unlike deep learning, these techniques are not very successful when they are trained on raw input data. And this is one of the most important property of deep learning, the features are learned from the data. [5]

The problems like computational power and large datasets were addressed with time and the improvement of technology, but the solution to the vanishing gradient problem came from [6], a solution proposed by Geoffrey E. Hinton. In [6] Hinton showed that the learning process can be done one layer at a time and then the whole network can be connected and fine-tuned. These layers(autoencoders) are trained in an unsupervised fashion- one by one- and then connected to a classical MLP layer. [6,7] This approach resulted in deep learning gaining huge popularity in recent years. And deep learning techniques won many competitions- like IJCNN 2011, demonstrating for the first time, a computer can do image recognition tasks better than humans [8]- and proved that deep learning is very good at complex tasks like image recognition, natural language processing etc.

After its success, researchers focused on improving these deep learning techniques. Around 2010-2013 it was discovered that Rectified Linear Units (ReLUs) perform better than sigmoidal activation functions. [8] It is stated in [9] that ReLUs are a better model of neurons in human brain, and it is shown that a deep neural network with rectifying neurons can be trained without unsupervised pretraining.

2 Implementation

1 Robot Neural Network

Before the implementation of the network, the data collected from the robot was cleaned in the laboratory using MATLAB. Duplicate values and noise were removed. The noise mentioned here were the values in the input data that are bigger than 5000, which is the maximum value of a sensor reading. Reason for this noise in the data remains unknown.

Implementation of the neural network started with the functions for reading and manipulating the data. If there are no weight text files supplied in the working directory, the program starts in training mode. Read data function opens the input and target csv files and reads the data into their respec-

tive arrays. And at the same time, it keeps the records of the minimum and maximum values to be used later for normalization and un-normalization processes. After the data are read, shuffle function is called. This function shuffles an index array so that the separate input and target arrays can be shuffled the same way by looking at the index array. When the data are shuffled, they are normalized according to the min and max values recorded during read operation. Then the data are separated with percentages 70/15/15 as training, validation, test. Training of the neural network starts after these operations.

Training part was designed to be like a grid search, it tries different alpha, hidden node and learning rate values, but the values for these parameters can also be set as constants if desired. Before learning starts, a data struct is created and filled. This struct holds the input, output, hidden values, results, errors of the results and hidden, output weights also including their past delta values. All the operations are applied to this data struct. Initialization function is called with input, hidden and output node number values. This function basically creates the skeleton of the network based on the values passed to the function. Then it fills the weight arrays in the data struct with random values between [-1,1]. Initialized struct is then passed to the feed-forward function, which is a classical feed-forward operation using logistic activation function. After calculating the errors from returned struct, the backpropagation function is called. It calculates the new weights and records the past weights for momentum calculation. When one pass over whole training set is done, validation set is fed to the network by calling the validate function for RMSE calculation. Returned RMSE is used for cut-off checking. If the latest RMSE > 0.33 or if ((latest RMSE – min RMSE) > 0.15 and min RMSE < 0.1) the training process stops. Otherwise, if the latest RMSE < min RMSE, min RMSE is updated. Then the training data is shuffled again, and a new epoch starts. When the training stops completely, training error values, RMSE values, parameters of the network and weights are written to a transcript file for record keeping.

If there are hWeights and yWeights text files present in the running directory of the program, these weights are read- and the learning part is skipped completely- into a data struct. Then the mobile robot and its sensors are initialized. After the initialization, sensor (0,1) readings are normalized and put into the struct to be passed to the feed-forward function. Min and max values used in normalization process when running in feed-forward only mode are manually selected after creating a histogram of the input and target values. The outliers found in the histograms are trimmed and new min and max values are used. In the end, output is un-normalized and sent to the wheels, then the process repeated.

2 Deep Neural Network

First part of section 2.2 will provide a general description of the system built by Group 4 (Tomoko Ayakawa, Anton Bubnov, Vinoth Kolandaira and Ogulcan Ozer). Second part of section 2.2 will discuss the author's personal contribution to the model selection.

General Description (drafted by Tomoko Ayakawa)

Information in this section is common among the members of Group 4.

To note, several changes have been made after the demonstration on 12 December 2018. The major changes are that the model was changed from a normal Autoencoder to a Stacked Autoencoder, the size and depth of the network were adjusted, and additional data-pre-processing was made.

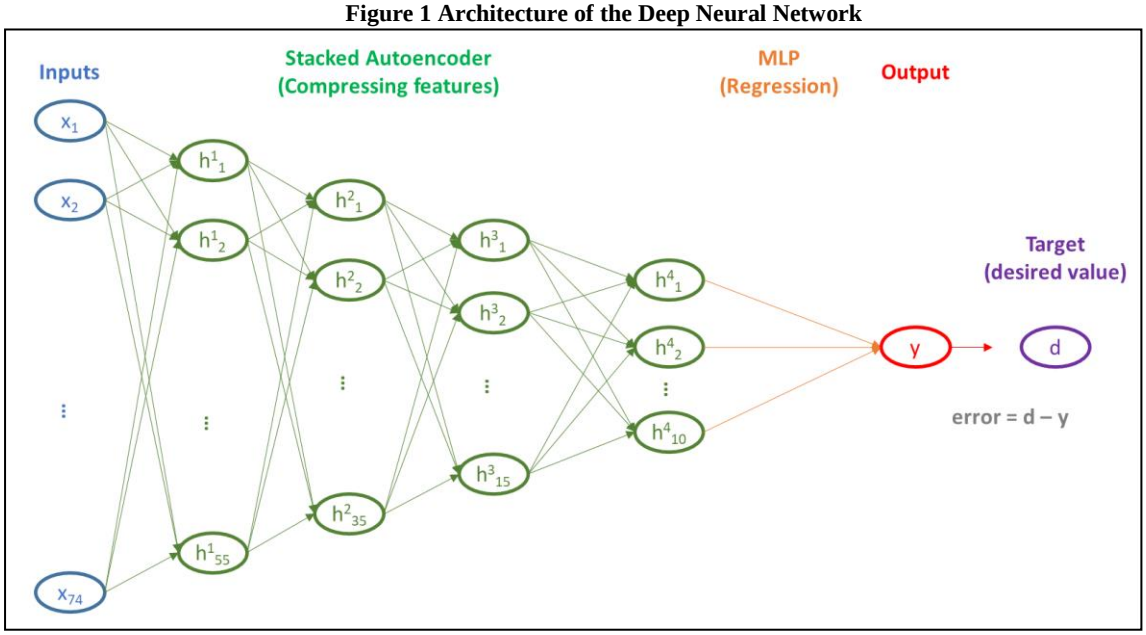
Tools

The system was built using `keras` packages with Python 3. This tool was selected because of the three reasons. Firstly, it is suitable for rapid experimentation because it was originally designed for that purpose [10]. Secondly, `keras` uses `TensorFlow` as its backend system, and `TensorFlow` is said to be one of the best tools for numerical computation [10]. Thirdly, `keras` is used along with `scikit-learn` (`sklearn`), and both are highly user friendly and easy to use.

Architecture

Figure 1 describes the architecture of our Deep Neural Network. It reads 74 features and produces a single output. It employs Stacked Autoencoder and Multi-Layer Perceptron (MLP). The Stacked Autoencoder is composed of 4 layers and built layer-by-layer. It gradually reduces the feature dimensionality from 74 to 55, 35, 15, then to 10. Each layer is trained up to 10 epochs using Rectified Linear Unit (ReLu) as activation function. Adaptive moment estimation (Adam) optimization with the default learning rate 0.001 is applied.
The output layer uses 10 outputs of the autoencoder as its inputs to forecast the amount of sales. ReLu and Adam optimization with the default learning rate 0.001 are used for the MLP.

Figure 1 Architecture of the Deep Neural Network



Training Data

The training data contains 74 features. The historical data from training.csv (downloaded from [11]) and associated store information from store.csv (ditto) are jointed and pre-processed. The features names are as listed below.

1) Store
2) Mon
3) Tue
4) Wed
5) Thu
6) Fri
7) Sat
8) Sun
9) Customers (log)
10) Open
11) Promo
12) HolidayNone
13) PublicHoliday
14) ChristmadHoliday
15) EasterHoliday
16) SchoolHoliday
17) Year2013
18) Year2014
19) Year2015
20) Jan
21) Feb
22) Mar
23) Apr
24) May
25) Jun
26) Jul
27) Aug
28) Sep
29) Oct
30) Nov
31) Dec
32) Day 1
33) Day 2
34) Day 3
35) Day 4
36) Day 5
37) Day 6
38) Day 7
39) Day 8
40) Day 9
41) Day 10
42) Day 11
43) Day 12
44) Day 13
45) Day 14
46) Day 15
47) Day 16
48) Day 17
49) Day 18
50) Day 19
51) Day 20
52) Day 21
53) Day 22
54) Day 23
55) Day 24
56) Day 25
57) Day 26
58) Day 27
59) Day 28
60) Day 29
61) Day 30
62) Day 31
63) StoreType a
64) StoreType b
65) StoreType c
66) StoreType d
67) AssortmentBasic
68) AssortmentExtended
69) AssortmentExtra
70) CompetitionDistance (log)
71) DaysSinceCompetitionOpen (log)
72) Promo2
73) DaysSincePromo2Since (log)
74) PromoInterval

Table 1 depicts the scope of data used for the training, validation and testing.

Table 1 Training Data

Purpose		Training Data*				Kaggle Data (features only) *
		Training (80%)		Test (20%)		
		80%	20%			
Stack Autoencoder Training		✓		-	-	✓
MLP	Training	✓	-	-	-	-
	Validation	-	✓	-	-	-
Final Evaluation		-	-	✓	✓	-

※ Training Data : *training.csv* + *store.csv* downloaded from Kaggle site
※ Kaggle Data : *test.csv* + *store.csv* downloaded from Kaggle site

System Functions

The functions of the system are as follows:

1) Pre-process the training data and Kaggle testing data
2) Read the pre-processed training data and Kaggle testing data
3) Scale the features and the target using MinMaxScaler. MinMaxScaler normalises the data in the range of 0 and 1.
4) Split the training data into training (80%) and test datasets (20%)
5) Train a Stacked Autoencoder by repeating the followings (unsupervised):
   5-1) Train a layer up to 20 epochs, using the inputs as target values
   5-2) Compute outputs with the trained layer in order to use them as inputs of the next layer

   When all layers are trained, loss history of each layer is exported to a log file.
6 Assemble the Autoencoder layers and add MLP layers
7 7-1) Train the model up to 100 epochs, using the sales as target values (supervised). Validation is employed using 20% of the training data.
   7-2) Loss history (training and validation) is exported to a log file.
8 Evaluate the performance
   8-1) Make a prediction with the test dataset
   8-2) Replace small predicted values with the threshold value (0.0023)
   8-3) Compute RMSE
   8-4) Scale back the outputs and export them into a csv file
   8-5) Export the current configuration and RMSE to a log file
9 Make a prediction with Kaggle testing data
   9-1) Make a prediction using the trained model
   9-2) Replace small predicted values with the threshold value (0.0023)
   9-3) Scale back the outputs and export the results to a csv file

Performance

Figure 12 and 13 show the decrease of Mean Square Error (MSE) during the training of Stacked Autoencoder and the output layer respectively.

Figure 2 Decrease of Loss in Unsupervised Training of Stacked Autoencoder

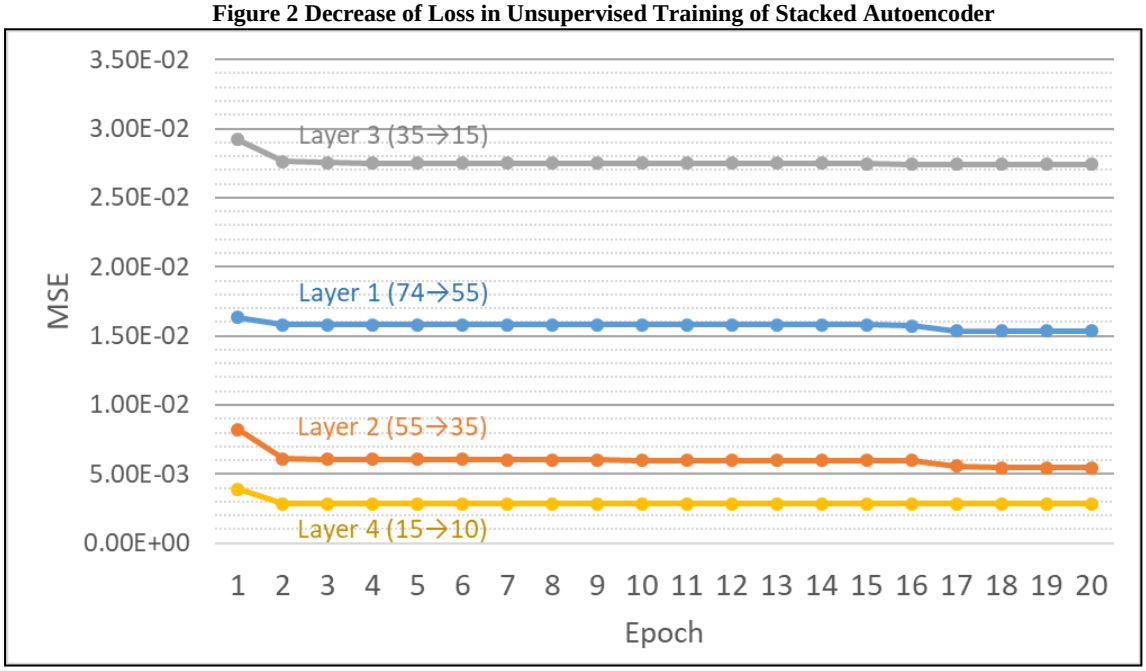
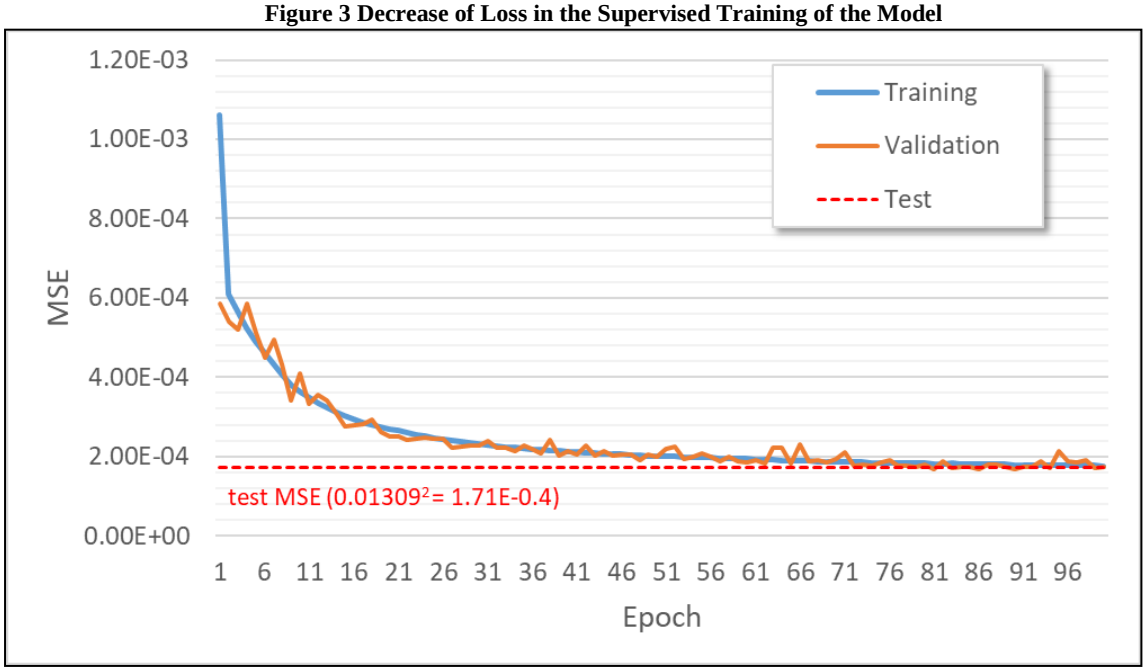


Figure 3 Decrease of Loss in the Supervised Training of the Model



The trained model achieves the prediction accuracy of 0.01309 in Root Mean Square Error (RMSE). Table 2 shows RMSE computed on the system as well as the scores of the submission of the Kaggle testing data. Evaluation metrics used by Kaggle is Root Mean Square Percentage Error (RMSPE).

Table 2 Evaluation Metrics

	RMSE	Kaggle Score (RMSPE)	
		Private	Public
Best Score	0.01309	0.23061	0.21808
Average of Three Run	0.01344	0.23493	0.22086

※ Private Score : final score, computed based on 50% of submitted data
※ Public Score : computed based on the remaining 50% of submitted data

Personal Contribution

Personal contribution to the project- task assigned- was the pre-processing of the competition data. Manipulation of the data was done using python, using Pandas library. After communicating with team members about how the input features should look and how should we fill the missing values, many different functions were implemented. Using this python script, different training and testing combinations of outputs can be gathered. The script was designed to be both imported and used from the command line. Process.py script can be called from command line with the arguments: test, train, store, merge, test2train, all. Test command processes test data to be in same dimensions with the train data, and outputs 'processedtest.csv'. This output is only for prediction, because the important part of processedtest is, customers and sales columns are 0, which are to be predicted after the model is trained. Train command processes training data. Fills empty rows with either mean values or -1 if they are categorical data. Also separates the Date column as year, month, day columns. Store com-

mand processes the extra store data provided. Again, the empty rows are filled with mean values or -1 if they are categorical data. Test2train function processes 'processedtest.csv' to be used as a training data. Fills the Customers and Sales columns with mean values according to which store the row belongs. Merge command merges extra store information found in the 'store.csv' to each of the 'processedtrain.csv','processedtest.csv', 'test2train.csv' in case team members wants to compare the results with or without the extra store information. Lastly, all command does everything mentioned above, and processes the data for each occasion.

Another contribution was the python script dist.py. Which was used to visualise the data and distributions, so that we could see which features influence the sale numbers and which features have distinctions that would help the network perform better.

3 Parameters

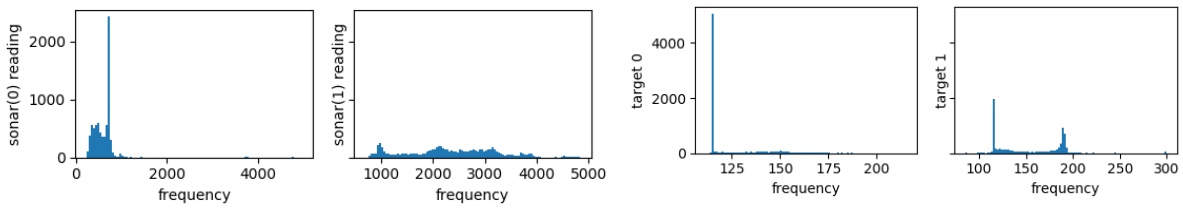
1 Robot Neural Network Parameters

Best parameters- which resulted in best weights that are used in the demonstration- for the mobile robot neural network were decided by performing an exhausting grid search. Parameters used in the grid search were:

- Number of hidden nodes, from 2 to 8.
- Alpha, from 0.01 to 0.06, with 0.01 increments.
- Learning rate, from 0.1 to 0.6, with 0.05 increments.

The results acquired from the grid search were examined and the weights of the runs with smallest RMSEs were tested on the mobile robot. And the weights with the min RMSE value 0.0694136 had the best performance in the MobileSim tests. Parameters of this run were recorded as: lambda = 0.5, alpha = 0.04, learning rate = 0.45. After this, multiple runs were performed using only these weights, and even though some of the results had smaller RMSE values, none of them performed as good as the first weights.

After the weights were chosen, histograms of the sonar readings and target values were created. And the outliers in the histogram were excluded by adjusting the min and max values used in the normalization and un-normalization process.

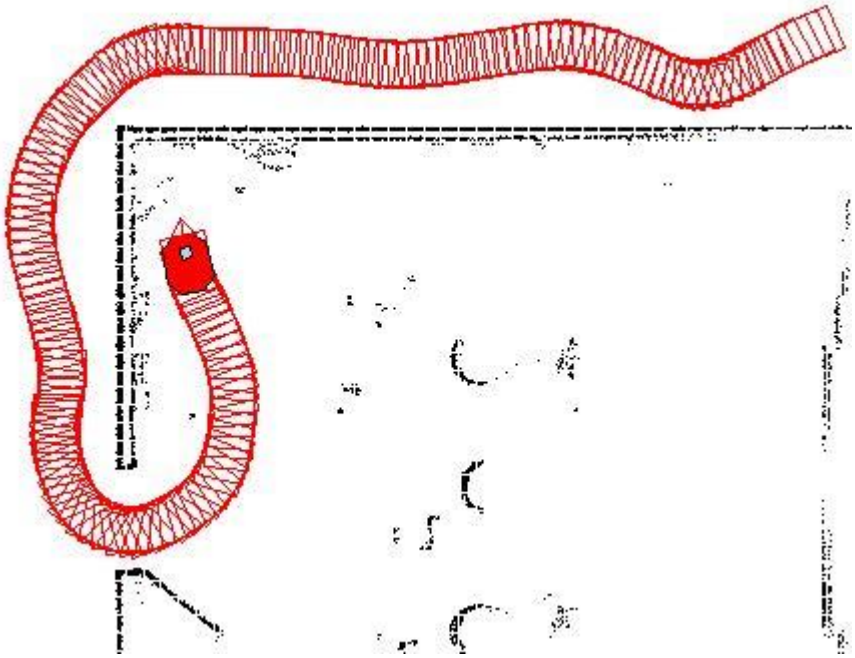


Sonar 0 min and max values were changed from (230.63, 4965.3) to (230.63, 1000), sonar 1 min and max values were kept the same. Target 0 min and max values were changed from (111.23, 215.82) to (90, 300), Target 1 min and max values were changed from (84.862, 300) to (90, 300).

4 Experiments and Results

1 Robot Neural Network

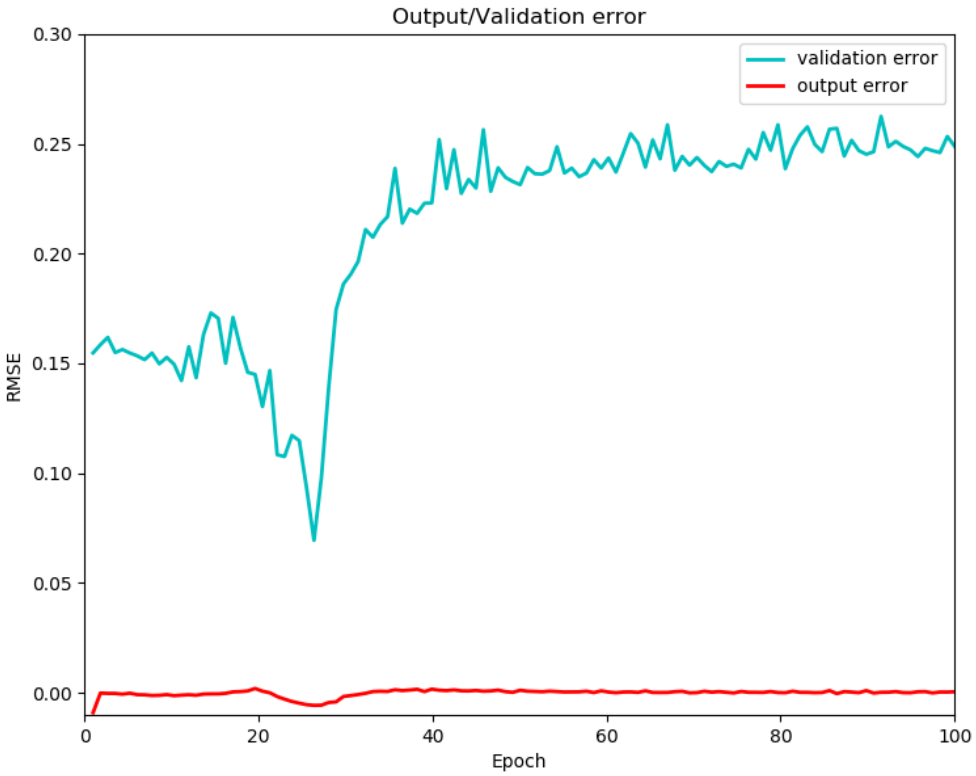
Performance of the mobile robot in the experiments done in MobileSim was successful and satisfactory. The robot could follow a left edge properly and did not have any problems taking sharp turns.



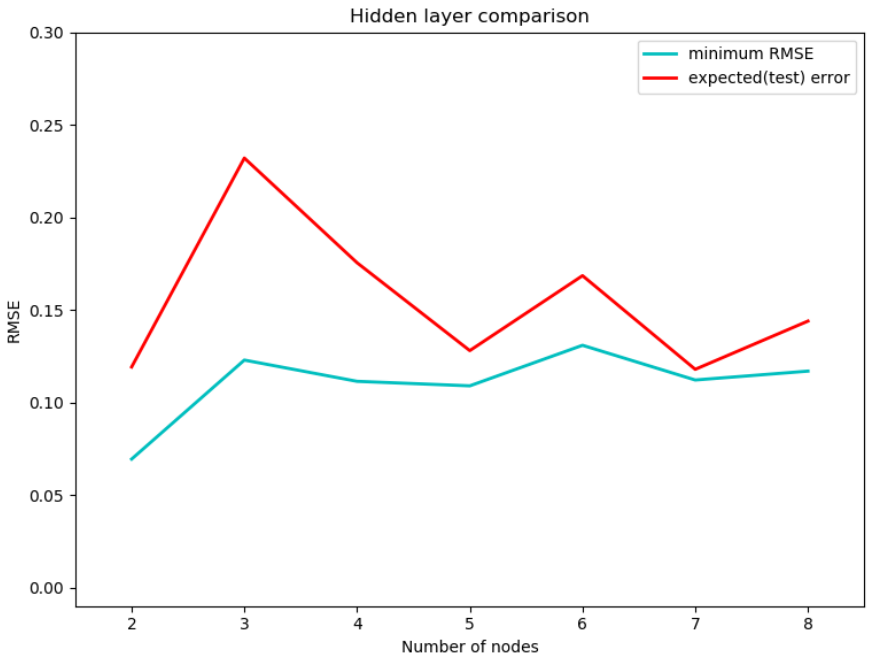
Mobile robot following the left edge, using the latest implementation of the neural network with chosen weights.

The output and validation error graph for the best weights can be seen below. In this result, the cut-off lines in the code were disabled for demonstration purposes.

	Lambda	**Alpha**	**Learning Rate**	**RMSE**
Best Weights	0.5	0.04	0.45	0.0694136



Even though gradient descent with momentum tries to prevent getting stuck in local minima, the cut-off function also allows 0.15 error difference breathing room for the training process, so that the network can keep learning for a little more time, in case the min RMSE value is in local minima. Also, the errors of hidden layers with different numbers of nodes were tested.



Each trial uses the same best parameters acquired during the experiment and repeated for 100 epochs. As it can be seen from the graph the lowest error is acquired from the network with two hidden nodes, and the previous testing and observations confirms this. But there is also an important factor to be considered. In all the experiments it was discovered that the weights with lower error did not always perform as desired and better than the weights with higher error. Throughout the trials, many weights with errors lower than 0.0694136 were found. Some of them behaved like a left edge follower with just worse performance, but some of them did not show any sign of left edge following behaviour at all. And the reason for this is believed to be the RMSE being an average error of the system, that having a lower error does not necessarily lead to a better left edge following behaviour.

2 Deep Neural Network

The best performing network in the experiments was the stacked autoencoder connected to an MLP without any hidden layers using ReLU as the activation function.

Model	Encoder Layers	Activation	Hidden Nodes	RMSE	Kaggle Private	Kaggle Public
Stacked Autoencoder	55, 35, 15, 10	relu	None	0,01309	0,23061	0,21808

There are also the results of some previous trials with different models

Stacked Autoencoder	15, 10, 5, 3	relu	5	0,02165	0,26467	0,24097
Stacked Autoencoder	15, 10, 5, 3	relu	4	0,02096	0,27160	0,24645
Stacked Autoencoder	15, 10, 5, 3	relu	8	0,02164	0,27686	0,24796
Stacked Autoencoder	15, 10, 5, 3	relu	3	0,02193	0,27301	0,25147
Stacked Autoencoder	15, 10, 5, 3	relu	2	0,02045	0,27926	0,25204
Stacked Autoencoder	15, 10, 5, 3	sigmoid	3	0,02147	0,27909	0,25260
Stacked Autoencoder	15, 10, 5, 3	sigmoid	5	0,02147	0,28460	0,26217

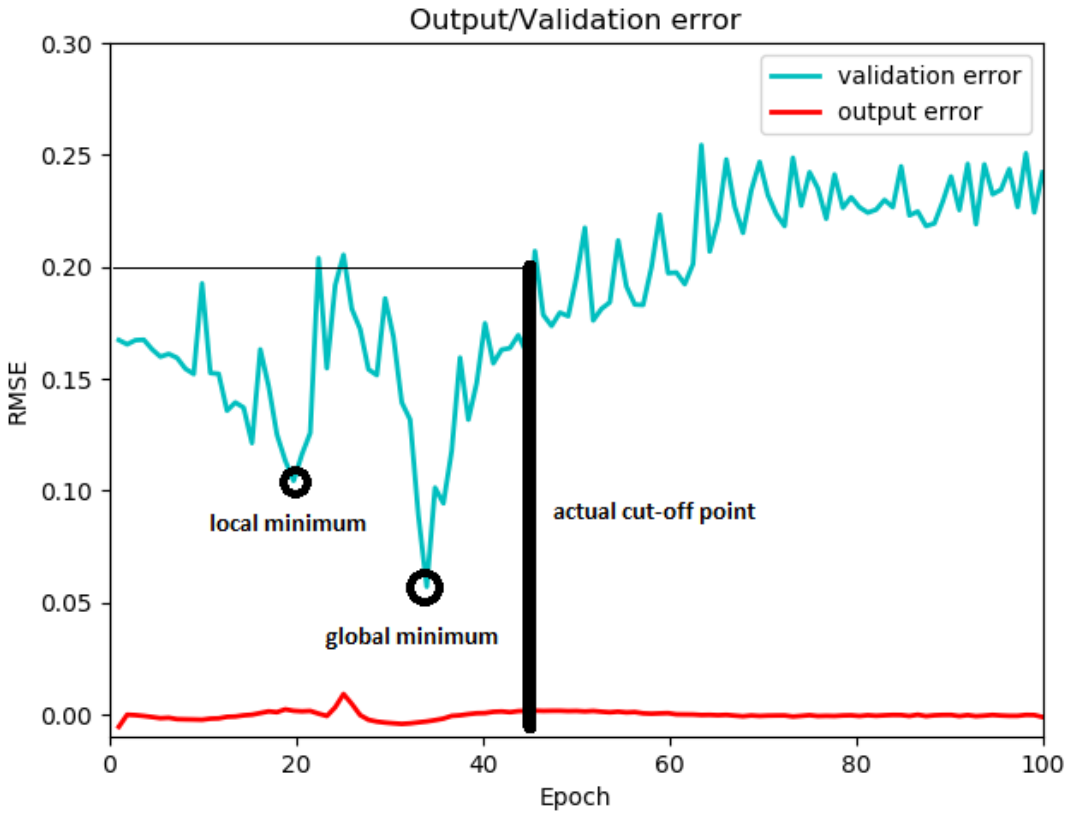
It can be seen from the results that; the number of hidden nodes does not seem to make much difference. However, we can say that the models using ReLU as their activation function overall perform better than the models using sigmoid.

If we look at our best performing result and compare it to the others in the Kaggle Public leader board, our model sits around 2816th place in the leader board. And this result does not look good. Because If we look at some other team`s kernels that have better results, we can see that some of them use- algorithms that are much simpler than DL- random forests, classical MLPs etc.

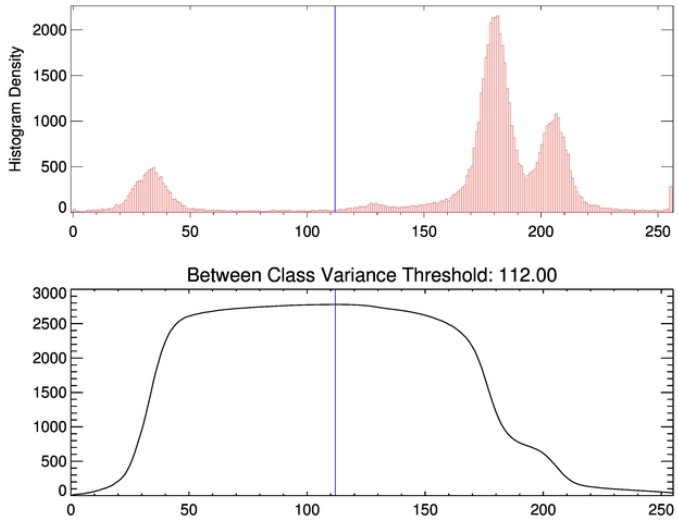
The reason for such result can be separated into two parts. We might be using an approach that is unnecessarily complex for the solution of the problem. Or we have significant optimization problems with our models. Either way, we have a problem that needs to be examined more carefully.

5 Strengths and Weaknesses

Since the system was implemented with C++ from scratch, and C++ being derived from C, the training process is fast. Also, since C++ has pointers, it makes the data manipulation easier and faster while using functions. It removes the overhead of copying the data in every single operation, which makes a great contribution to the speed of the system. The flexible cut-off was also useful. It prevents the training process from stopping early, so that it can try to find the global minimum for a little longer. An example from a different trial can be seen below.



Even though the flexible cut-off helps to find the global minimum, it is also a weakness in terms of the training time. And another factor to think about is that the 0.15 error difference was chosen after a lot of testing and observations. It performs great for this specific problem and data, but it is highly unlikely that it can generalize. Another weakness of the system is processed min max values for scaling. The min and max values are manually adjusted after creating a histogram of the data and detecting the outliers.



taken from [12]

For future improvement, I believe the detection of these outliers can be automated by using the Otsu`s method, which is an algorithm used in computer vision to separate bimodal distributions. As it can be seen in the graph, Otsu`s method returns a threshold value between two peaks. Depending on the data and application, min or max value can be replaced with the threshold value for automated outlier exclusion.

6 Conclusions

The result in the demonstration was satisfactory. System performed as expected and correctly. But as mentioned before, getting a low error from training does not necessarily lead to correct behaviour. Thus, the real-life performance of the system needs to be confirmed manually either in simulation or on an actual mobile robot, which I believe is a big problem. For a mobile robot, training solution like [3] - if can be afforded- is better suited for practical systems. Because the trained neural network indirectly minimizes the error according to the movement of the robot with various fitness values, which also penalizes the collisions.

In the case of the deep learning, the results we have are not satisfactory. I believe we have a lot of things to improve. But our results in general still confirm some of the things mentioned before. As stated in [8,9], ReLUs provide better results for the deep learning models. And as shown in [8], deep learning is generally used for very complex problems, which might not be the best tool for the solution of this problem.

7 References

[1] P. K. Pal and A. Kar, "Mobile Robot Navigation Using a Neural Net," presented at the IEEE International Conference on Robotics and Automation Bhabha Atomic Research Centre, Bombay, 1995.

[2] W. Wahab, "Autonomous Mobile Robot Navigation Using a Dual Artificial Neural Network ", ed. Faculty of Technology, University of Indonesia Depok, Indonesia Department of Electrical Engineering, 2009.

[3] F. Dario and M. Francesco, "Automatic Creation of an Autonomous Agent Genetic Evolution of a Neural Network Driven Robot " presented at the Third International Conference on Simulation of Adaptive Behavior Brighton, UK, 1994.

[4] G. Ian, B. Yoshua, and C. Aaron, *Deep Learning*. Cambridge , MA: MIT Press, 2017.

[5] Y. LeCun, Y. Bengio, and G. Hinton. (2015) Deep learning. *NATURE*. 436-444.

[6] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation,* vol. 18, pp. 1527–1554, 2006.

[7] G. E. Hinton and R. R. Salakhutdinov. (2006) Reducing the Dimensionality of Data with Neural Networks *SCIENCE*. 504-507.

[8] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," The Swiss AI Lab IDSIA, Manno-Lugano Switzerland 2014

[9] X. Glorot, A. Bordes, and Y. Bengio," Deep Sparse Rectifier Neural Networks," Fort Lauderdale, FL, USA, 2011.

[10] M. Opala, "Deep Learning Frameworks Comparison – Tensorflow, PyTorch, Keras, MXNet, The Microsoft Cognitive Toolkit, Caffe, Deeplearning4j, Chainer", *Netguru.co*, 2018. [Online]. Available: https://www.netguru.co/blog/deep-learning-frameworks-comparison. [Accessed: 27 Nov 2018].

[11] “Rossmann Store Sales”, Kaggle, [Online]. Availbale: https://www.kaggle.com/c/rossmann-store-sales/data. [Accessed: 27 Nov 2018].

[12] (2012, January). *Separating Bimodal Distributions with Otsu Threshold* [Online]. Available: http://www.idlcoyote.com/code_tips/otsu_threshold.php.

8 Appendix

```
/*_____________________________________________________________________
__________
# CE889 Neural Networks and Deep Learning | Ogulcan Ozer. | 11 December
2018
      UNFINISHED. See lines 192-202-414.

_____________________________________________________________________
________*/
/*---------------------------------------------------------------------
----------
#     CE889 Left edge following multilayer perceptron using gradient
descent,
      with momentum.

      Usage: Program has 2 modes.
      1: Feed-forward only -> Looks for 'yWeights.txt' and
'hWeights.txt' in the
      running directory. If they exist, assumes proper weights are
provided and
      starts running on feed-forward mode using the sonar reading as
inputs and
      provided files as weights of the network.

      2: Training -> If 'yWeights.txt' and 'hWeights.txt' do not exist
in the
      running directory, looks for 'inputClean.txt' and
'targetClean.txt' If
      they exsist, checks the dimensions of the data. Each newline in
the
      input file are assumed to be features, and each newline in the
output file
      are assumed to be predictions. According to the data dimensions,
input and
      output nodes are created. starts the training of the network.

      Provide the files according to the desired output and run the
executable.

---------------------------------------------------------------------
--------*/
#include <limits>
#include <cstddef>
#include<iostream>
#include<fstream>
#include<string>
#include <vector>
#include <string>
#include<cmath>
#include <algorithm>
#include <random>
#include "Aria.h"

using namespace std;
```

```
/*-------------------------------------------------------------------------
----------
#     Variables and structs
-------------------------------------------------------------------------
--------*/
int HIDDEN = 5;//Number of hidden nodes
double LEARN = 0.0;//Learning rate of the NN
double ALPHA = 0.0;//Alpha value used in momentum
double LAMBDA = 0.5;//Lambda value used in logistic function.

vector<int> indexValues;//Index record keeping for data shuffle.
vector<double> shuffleTemp;//Temporary vector for creating new training
data from shuffled indexes.
vector<vector<double>> dataInput;//Training data
vector<vector<double>> dataTarget;//Labels or targets for training data
vector<vector<double>> validateInput;//Validation data
vector<vector<double>> validateTarget;//Labels or targets for
validation data
vector<vector<double>> testInput;//Test data
vector<vector<double>> testTarget;//Labels or targets for test data

 //Minimum and maximum values for each input and target, to be used in
normalization and un-normalization:
vector<double> inputMin;
vector<double> inputMax;
vector<double> targetMin;
vector<double> targetMax;

//Records of output errors and validation errors.
vector<vector<double>> eValidate;

//For recording minimum RMSE
double minRMSE = INT_MAX;//

//Data struct for a single input object:
typedef struct Data {

      vector<double> x; //Input values
      vector<double> y; //Output values
      vector<double> h; //Hidden values
      vector<double> d; //Target values/labels
      vector<double> e; //Errors
      vector<vector<double> > w; //Hidden-Output weights
      vector<vector<double> > wH; //Input-Hidden weights
      vector<vector<double> > wMoment; //Past delta weights
      vector<vector<double> > wHmoment;; //Past hidden delta weights

} Data;
//////////////////////////////////////////////////////////////////////
//////////
/*-------------------------------------------------------------------------
----------
#     Function definitions
-------------------------------------------------------------------------
--------*/

//Read the sample data from a .txt file.
int read_data(string input, string target);
```

```
//Normalize the data for each input and target.
void normalizeData(void);

//UN-Normalize the data for each input and target.
double unNormalizeData(double d, int node);

//Returns the avarage rmse of all the outputs.
double get_rmse(vector<vector<double>> errorset);

//Seperate data as 70/15/15
void seperate_data(void);

//Initialize single Input with associated weights.
Data init_data(int inputNodes, int hiddenNodes, int outputNodes);

//Set the error values of a data object
void set_errors(Data *data);

//Run validation data on given weights, nested inside a data object.
double validate(Data *data);

//Run test data on given weights, nested inside a data object.
double test(Data *data);

//Feedforward operation.
void feed_forward(Data *data);

//Backpropogation operation.
void backpropogate(Data *data);

//Shuffle the training data
void shuffleData(void);




/*----------------------------------------------------------------------
----------
#     Main Program
----------------------------------------------------------------------
--------*/
int main(int argc, char **argv)
{
      ifstream File;
      Data minData;
      File.open("yWeights.txt");//Check if there are any weights
provided.
      if (!File.is_open())///////
      {
            cout << "Weights could not be found. Training..." << endl;

            if (read_data("inputClean", "targetClean") == 1)//If there
are no weights, start the training.
            {
                  shuffleTemp.resize(dataInput[0].size());
                  indexValues.resize(dataInput[0].size());
                  for (int i = 0; i < dataInput[0].size(); i++) {
```

```
                        indexValues[i] = i;
                  }
                  shuffleData();//Shuffle
                  normalizeData();//Normalize
                  seperate_data();//Seperate the data 70/15/15

                  //Get the sizes of input and output nodes from the
read data.
                  int inValue = dataInput.size() + 1;
                  int outValue = dataTarget.size();

                  //Resize the shuffle and index arrays to seperated
data.
                  shuffleTemp.resize(dataInput[0].size());
                  indexValues.resize(dataInput[0].size());
                  for (int i = 0; i < dataInput[0].size(); i++) {
                        indexValues[i] = i;
                  }


                  //Do- for different hidden nodes, alpha values and
learning rates.
                  for (HIDDEN = 3; HIDDEN < 9; HIDDEN++) {
                        ALPHA = 0.1;
                        for (int d = 0; d < 6; d++) {
                              ALPHA = ALPHA + 0.01;
                              LEARN = 0.4;
                              for (int o = 0; o < 10; o++) {
                                    LEARN = LEARN + 0.05;


                                    //Initialize the weights of the data
structs:
                                    Data test1 = init_data(inValue,
HIDDEN, outValue);//Struct for feedforward input.
                                    minData = init_data(inValue, HIDDEN,
outValue);//Struct for record keeping - minimum weights

                                    //Initialize the error history
vector.
                                    eValidate.resize(2);
                                    eValidate[0].resize(0);//for
                                    eValidate[1].resize(0);
                                    minRMSE = INT_MAX;

                                    //Do hundred epochs * 1 / (LEARN +
0.4) :
                                    for (int epoch = 0; epoch < 100 * (1
/ (LEARN + 0.4)); epoch++)
                                    {


                                          double passErr = 0;
                                          for (int i = 0; i <
static_cast<int>(dataInput[0].size()); i++)
                                          {
                                                //Initialize features and
target values and biases.
```

```
                                                test1.x[0] = 1;
                                                test1.h[0] = 1;
                                                test1.x[1] =
dataInput[0][i];/*Needs to be corrected for initializing the values one
by one. Should be doing it dynamically*/
                                                test1.x[2] =
dataInput[1][i];
                                                test1.d[0] =
dataTarget[0][i];
                                                test1.d[1] =
dataTarget[1][i];

                                                //Do one pass on training
data:
                                                feed_forward(&test1);
                                                set_errors(&test1);
                                                backpropogate(&test1);

                                                passErr = passErr +
((test1.e[0] + test1.e[1]) / 2);//sum the errors

                                          }
                                          /**************END OF ONE
PASS**************/


                                          //Get validation error for the
last pass, and add it to the error records with the average output
errors.
                                          passErr = passErr /
dataInput[0].size();
                                          double val = validate(&test1);
                                          cout << "Validation error:" <<
val << endl;
                                          eValidate[0].push_back(val);

      eValidate[1].push_back(passErr);

      ////////////////////////////////////////


                                          //Cutoff point
                                          if (((val - minRMSE > 0.18) &&
minRMSE < 0.1) || (val > 0.33)) {
                                                break;
                                          }
                                          //Save the weights with minimum
rmse value in this pass
                                          if (minRMSE > val)
                                          {
                                                minRMSE = val;

                                                for (int y = 0; y <
outValue; ++y)
                                                {
                                                      for (int h = 0; h <
HIDDEN; ++h)
                                                      {
```

```
      minData.w[y][h] = test1.w[y][h];
                                                }
                                          }
                                          for (int h = 0; h <
HIDDEN - 1; ++h)
                                          {
                                                for (int x = 0; x <
inValue; ++x)
                                                {

      minData.wH[h][x] = test1.wH[h][x];
                                                }
                                          }
                                    }


                                          shuffleData();//Shuffle



                                    }
                                    /**************END OF HUNDRED
EPOCHS**************/

                                    //Get Expected Error
                                    double testError = test(&minData);

                                    /*-----------------------------------
-----------------------------------------
                                    #     Output data to a .txt file for
later evaluation
                                    -----------------------------------
-----------------------------------------*/
                                    std::ofstream ofs;
                                    ofs.open("test.txt",
std::ofstream::out | std::ofstream::app);
                                    ofs << "Number of hidden nodes: " <<
HIDDEN - 1 << endl;
                                    ofs << "Lambda: " << LAMBDA << endl;
                                    ofs << "Alpha: " << ALPHA << endl;
                                    ofs << "Learning Rate: " << LEARN <<
endl;
                                    ofs << "Minimum RMSE for this run: "
<< minRMSE << endl;
                                    ofs << "Expected Error for this run:
" << testError << endl;
                                    ofs << "ValErrors: ";
                                    for (int i = 0; i <
eValidate[0].size(); i++)
                                    {
                                          ofs << eValidate[0][i] << ",";
                                    }
                                    ofs << endl;
                                    ofs << "Errors: ";
                                    for (int i = 0; i <
eValidate[1].size(); i++)
                                    {
```

```
                                    ofs << eValidate[1][i] << ",";
                              }
                              ofs << endl;
                              ofs.close();
                              cout << "Validation min RMSE reached:
" << minRMSE << endl;
                              ofs.open("yWeights.txt",
std::ofstream::out | std::ofstream::app);

                              for (int y = 0; y < minData.w.size();
++y)
                              {
                                    for (int h = 0; h <
minData.w[0].size(); ++h)
                                    {
                                          ofs << test1.w[y][h] <<
",";
                                    }
                                    ofs << endl;
                              }
                              ofs.close();
                              ofs.open("hWeights.txt",
std::ofstream::out | std::ofstream::app);
                              for (int h = 0; h <
minData.wH.size(); ++h)
                              {
                                    for (int x = 0; x <
minData.wH[0].size(); ++x)
                                    {
                                          ofs << minData.wH[h][x]
<< ",";
                                    }
                                    ofs << endl;
                              }
                              ofs.close();

                        }
                  }
            }
      }
      else
      {
            cout << "Could not read the file/s.";
      }
}
else//Run the robot if there are weights provided.
{

      vector<vector<double> > w; //Hidden-Output weights
      vector<vector<double> > wH; //Input-Hidden weights
      double temp = 0;
      vector<double> values;

      // Read the weights for output.
      while (File >> temp)
      {
            values.push_back(temp);
            if (File.peek() == ',')
```

```
                        File.ignore();
                  if ((File.peek() == '\n') || (File.peek() == '\r'))
                  {
                        w.push_back(values);
                        values.clear();
                  }

            }

            File.close();

            //Read the weights for hidden layer.
            File.open("hWeights.txt");
            values.clear();
            temp = 0;
            while (File >> temp)
            {
                  values.push_back(temp);
                  if (File.peek() == ',')
                        File.ignore();
                  if ((File.peek() == '\n') || (File.peek() == '\r'))
                  {
                        wH.push_back(values);
                        values.clear();
                  }

            }
            minData = init_data(wH[0].size() + 1, wH.size() + 1,
w.size());
            for (int y = 0; y < w.size(); ++y)
            {
                  for (int h = 0; h < w[0].size(); ++h)
                  {
                        minData.w[y][h] = w[y][h];
                  }
            }
            for (int h = 0; h < wH.size(); ++h)
            {
                  for (int x = 0; x < wH[0].size(); ++x)
                  {
                        minData.wH[h][x] = wH[h][x];
                  }
            }
            File.close();
      }
      /*-----------------------------------------------------------------
----------------
      #     Main loop for robot feedforward
      -----------------------------------------------------------------
--------------*/

      //Initialize robot.
      Aria::init();
      ArRobot robot;
      ArPose pose;
      ArSensorReading *sonarSensor[8];

      //Read command line args.
```

```
	ArArgumentParser argParser(&argc, argv);
	argParser.loadDefaultArguments();
	argParser.addDefaultArgument("-connectLaser");
	ArRobotConnector robotConnector(&argParser, &robot);

	//Set normalization values
	inputMin.push_back(230.63);
	inputMax.push_back(1000);
	inputMin.push_back(673.37);
	inputMax.push_back(4835.3);
	targetMin.push_back(90);
	targetMax.push_back(300);
	targetMin.push_back(90);
	targetMax.push_back(300);
	//

	if (robotConnector.connectRobot()) {
		std::cout << "Robot Connected !" << std::endl;
	}

	robot.runAsync(false);
	robot.lock();
	robot.enableMotors();
	robot.unlock();

	//Feed-forward only loop.
	while (true)
	{
		//Read sonar values.
		double sonarRange[8];
		for (int i = 0; i < 8; i++)
		{
			sonarSensor[i] = robot.getSonarReading(i);
			sonarRange[i] = sonarSensor[i]->getRange();

		}
		//Set inputs of the struct from sonar readings and set
biases.
		minData.x[0] = 1;
		minData.h[0] = 1;/*Needs to be corrected for initializing
the values one by one. Should be doing it dynamically*/
		minData.x[1] = (sonarRange[0] - inputMin[0]) / (inputMax[0]
- inputMin[0]);
		minData.x[2] = (sonarRange[1] - inputMin[1]) / (inputMax[1]
- inputMin[1]);

		//Feed-forward
		feed_forward(&minData);

		//Set the speed of the robot.
		robot.setVel2(unNormalizeData(minData.y[0], 0),
unNormalizeData(minData.y[1], 1));

		//Command line output for important information.
		cout << "SONAR 0 = " << sonarRange[0] << endl;
		cout << "SONAR 1 = " << sonarRange[1] << endl;
		cout << "OUTPUT 0 = " << minData.y[0] << endl;
		cout << "OUTPUT 1 = " << minData.y[1] << endl;
```

```
            cout << "UNORM 0 = " << unNormalizeData(minData.y[0], 0) <<
endl;
            cout << "UNORM 1 = " << unNormalizeData(minData.y[1], 1) <<
endl;

            //Sleep/Delay for 100 msec.
            ArUtil::sleep(100);

      }
      return 0;

}//End of robot feed-forward loop.


/*-------------------------------------------------------------------------
----------
#     Functions
-------------------------------------------------------------------------
--------*/

//Initialize single Input with associated weights.
Data init_data(int inputNodes, int hiddenNodes, int outputNodes)
{
      Data data;

      //Initialize according to the dimensions of the data.
      data.x.resize(inputNodes);
      data.y.resize(outputNodes);
      data.h.resize(hiddenNodes);
      data.d.resize(outputNodes);
      data.e.resize(outputNodes);

      srand(time(NULL));

      data.wHmoment.resize(hiddenNodes - 1);
      data.wH.resize(hiddenNodes - 1);
      for (int h = 0; h < hiddenNodes - 1; ++h) {
            data.wH[h].resize(inputNodes);
            data.wHmoment[h].resize(inputNodes);
      }
      for (int h = 0; h < hiddenNodes - 1; ++h) {
            for (int x = 0; x < inputNodes; ++x) {
                  data.wH[h][x] = (((double)rand() / (RAND_MAX) * 2) -
1);

            }
      }
      data.wMoment.resize(outputNodes);
      data.w.resize(outputNodes);
      for (int y = 0; y < outputNodes; ++y) {
            data.w[y].resize(hiddenNodes);
            data.wMoment[y].resize(hiddenNodes);
      }
      for (int y = 0; y < outputNodes; ++y) {
            for (int h = 0; h < hiddenNodes; ++h) {
                  data.w[y][h] = (((double)rand() / (RAND_MAX) * 2) -
1);
```

```
			}
		}
		return data;
}

//Feedforward operation.
void feed_forward(Data *data) {

		vector<double> vH;
		vH.resize(static_cast<int>(data->h.size()) - 1);//Hidden node
sum.
		vector<double> vK;
		vK.resize(static_cast<int>(data->y.size()));//Output sum.



		//Feed forward - 1 . (Finding each vH value, vH < hidden node
size - 1, since h0 = bias)
		for (int h = 0; h < static_cast<int>(data->h.size() - 1); h++) {
			double tmp = 0;
			for (int x = 0; x < static_cast<int>(data->x.size()); x++) {

				tmp = tmp + data->x[x] * data->wH[h][x];

			}

			vH[h] = tmp;
		}

		//Apply logistic function to hidden layer sums.
		for (int h = 1; h < static_cast<int>(data->h.size()); h++) {

			data->h[h] = 1 / (1 + (exp(-(LAMBDA*(vH[h - 1])))));


		}
		//Feed forward - 2 . (Finding each vK value)
		for (int y = 0; y < static_cast<int>(data->y.size()); y++) {
			double tmp = 0;
			for (int h = 0; h < static_cast<int>(data->h.size()); h++) {

				tmp = tmp + data->h[h] * data->w[y][h];

			}
			vK[y] = tmp;

		}
		//Apply logistic function to output sums.
		for (int y = 0; y < static_cast<int>(data->y.size()); y++) {

			data->y[y] = 1 / (1 + (exp(-(LAMBDA*vK[y]))));;


		}

		//
		//End of Feedforward.
		return;
```

```
}

//Backpropogation
void backpropogate(Data *data) {

	//Initialize local gradients.
	vector<double> minH;
	minH.resize(static_cast<int>(data->h.size()));
	vector<double> minKsum;
	minKsum.resize(static_cast<int>(data->h.size()));
	vector<double> minK;
	minK.resize(static_cast<int>(data->y.size()));

	//Initialize delta weights.
	vector<vector<double>> deltaW;
	deltaW.resize(static_cast<int>(data->y.size()));
	for (int y = 0; y < static_cast<int>(data->y.size()); ++y) {
		deltaW[y].resize(data->h.size());
	}
	vector<vector<double>> deltaWh;
	deltaWh.resize(static_cast<int>(data->h.size() - 1));
	for (int h = 0; h < static_cast<int>(data->h.size() - 1); ++h) {
		deltaWh[h].resize(data->x.size());
	}

	//Calculate local minK gradient. (output gradient)
	for (int y = 0; y < static_cast<int>(data->y.size()); y++) {

		minK[y] = LAMBDA * data->y[y] * (1 - data->y[y]) * data-
>e[y];

	}
	//Sum of minK to be used in minH gradient.(hidden node gradient)
	for (int h = 0; h < static_cast<int>(data->h.size()); h++) {
		for (int y = 0; y < static_cast<int>(data->y.size()); y++) {

			minKsum[h] = minKsum[h] + (minK[y] * data->w[y][h]);

		}
	}
	//Calculate local minH gradient.
	for (int h = 1; h < static_cast<int>(data->h.size()); h++) {

		minH[h - 1] = LAMBDA * data->h[h] * (1 - data-
>h[h])*(minKsum[h - 1]);

	}
	//Calculate delta Wk for each weight, with moment.
	for (int y = 0; y < static_cast<int>(data->y.size()); y++) {
		for (int h = 0; h < static_cast<int>(data->h.size()); h++) {

			deltaW[y][h] = (LEARN * minK[y] * data->h[h]) + (ALPHA
* data->wMoment[y][h]);
			data->wMoment[y][h] = deltaW[y][h];

		}
	}
	//Calculate delta Wh for each weight, with moment.
```

```
	for (int h = 0; h < static_cast<int>(data->h.size() - 1); h++) {
		for (int x = 0; x < static_cast<int>(data->x.size()); x++) {

			deltaWh[h][x] = (LEARN * minH[h] * data->x[x]) +
(ALPHA * data->wHmoment[h][x]);
			data->wHmoment[h][x] = deltaWh[h][x];
		}
	}
	//Calculate new Wh weights.
	for (int h = 0; h < static_cast<int>(data->h.size() - 1); h++) {
		for (int x = 0; x < static_cast<int>(data->x.size()); x++)
		{
			data->wH[h][x] = data->wH[h][x] + deltaWh[h][x];

		}
	}
	//Calculate new Wk weights.
	for (int y = 0; y < static_cast<int>(data->y.size()); y++) {
		for (int h = 0; h < static_cast<int>(data->h.size()); h++) {

			data->w[y][h] = data->w[y][h] + deltaW[y][h];

		}
	}


}

//Shuffle the data
void shuffleData(void)
{
	//Get a random number generator and shuffle the index numbers.
	auto rng = std::default_random_engine{};
	std::shuffle(std::begin(indexValues), std::end(indexValues),
rng);


	//Shuffle both training and target values according to index.

	for (int i = 0; i < dataInput.size(); i++) {//Do for training.
		for (int j = 0; j < dataInput[0].size(); j++) {

			shuffleTemp[j] = dataInput[i][indexValues[j]];

		}
		dataInput[i] = shuffleTemp;



		shuffleTemp.clear();
		shuffleTemp.resize(dataInput[i].size());
	}
	for (int i = 0; i < dataTarget.size(); i++) {//Do for target.
		for (int j = 0; j < dataTarget[0].size(); j++) {

			shuffleTemp[j] = dataTarget[i][indexValues[j]];

		}
```

```
            dataTarget[i] = shuffleTemp;
            shuffleTemp.clear();
            shuffleTemp.resize(dataInput[i].size());
      }
}

//Seperate function for setting errors, since feedforward will also be
used in real life, without target values to evaluate an error.
void set_errors(Data *data)
{

      for (int y = 0; y < static_cast<int>(data->y.size()); y++) {

            data->e[y] = data->d[y] - data->y[y];

      }

}

//Run validation on current weights.
double validate(Data  *data)
{
      vector<vector<double>> errors;
      errors.resize(data->y.size() * 2);
      for (int y = 0; y < data->y.size() * 2; y++) {
            errors[y].resize(validateTarget[0].size());
      }
      //Validate for each data in validation set.
      for (int v = 0; v < validateInput[0].size(); v++) {
            for (int i = 0; i < validateInput.size(); i++) {
                  data->x[i] = validateInput[i][v];
            }
            for (int o = 0; o < validateTarget.size(); o++) {
                  data->d[o] = validateInput[o][v];
            }

            feed_forward(data);
            set_errors(data);

            for (int o = 0; o < data->y.size() * 2; o = o + 2) {
                  errors[o][v] = data->y[o / 2];
                  errors[o + 1][v] = data->d[o / 2];
            }
      }
      return get_rmse(errors);
}

//Run test on current weights.
double test(Data *data)
{
      vector<vector<double>> errors;
      errors.resize(data->y.size() * 2);
      for (int y = 0; y < data->y.size() * 2; y++) {
            errors[y].resize(testTarget[0].size());
      }
      //Test for each data in test.
      for (int v = 0; v < testInput[0].size(); v++) {
            for (int i = 0; i < testInput.size(); i++) {
```

```
                data->x[i] = testInput[i][v];
            }
            for (int o = 0; o < testTarget.size(); o++) {
                data->d[o] = testInput[o][v];
            }

            feed_forward(data);
            set_errors(data);

            for (int o = 0; o < data->y.size() * 2; o = o + 2) {
                errors[o][v] = data->y[o / 2];
                errors[o + 1][v] = data->d[o / 2];
            }
      }
      return get_rmse(errors);

}

//Read the data from .txt files and get the min and max values at the
same time, for normalization.
int read_data(string input, string target) {

      double temp = 0;
      int c = 0;
      vector<double> values;


      ifstream File;
      File.open(input + ".txt");
      if (!File.is_open())
      {
            cout << "It failed" << endl;
            return 0;
      }
      double min = 9999999.0;
      double max = numeric_limits<double>::lowest();
      while (File >> temp)
      {
            if (temp <= min)
                min = temp;
            if (temp >= max)
                max = temp;
            values.push_back(temp);
            if (File.peek() == ',')
                File.ignore();
            if ((File.peek() == '\n') || (File.peek() == '\r')) {

                dataInput.push_back(values);
                inputMin.push_back(min);//Keep record of min and max
values while reading
                inputMax.push_back(max);//
                min = 9999999.0;
                max = numeric_limits<double>::lowest();
                values.clear();
            }


      }
```

```
	File.close();// End of reading input file.
	temp = 0;
	c = 0;
	values.clear();
	File.open(target + ".txt");
	if (!File.is_open())
	{
		cout << "It failed" << endl;
		return 0;
	}

	min = 9999999.0;
	max = numeric_limits<double>::lowest();
	while (File >> temp)
	{
		if (temp <= min)
			min = temp;
		if (temp >= max)
			max = temp;
		values.push_back(temp);
		if (File.peek() == ',')
			File.ignore();
		if ((File.peek() == '\n') || (File.peek() == '\r')) {

			dataTarget.push_back(values);
			targetMin.push_back(min);//Keep record of min and max
values while reading.
			targetMax.push_back(max);//
			min = 9999999.9;
			max = numeric_limits<double>::lowest();
			values.clear();
		}


	}
	File.close();//End of reading output file.

	return 1;
	//
	// End of reading training-target data from the file.

}

//Normalize the data for each input and target.
void normalizeData(void) {

	for (int i = 0; i < static_cast<int>(dataInput.size()); i++) {
		for (int j = 0; j < static_cast<int>(dataInput[i].size());
j++) {

			dataInput[i][j] = (dataInput[i][j] - inputMin[i]) /
(inputMax[i] - inputMin[i]);

		}
	}
	for (int i = 0; i < static_cast<int>(dataTarget.size()); i++) {
		for (int j = 0; j < static_cast<int>(dataTarget[i].size());
j++) {
```

```
                  dataTarget[i][j] = (dataTarget[i][j] - targetMin[i]) /
(targetMax[i] - targetMin[i]);

            }
      }


}

//UnNormalize the given value according to its output.
double unNormalizeData(double d, int node)
{
      return (targetMin[node] + (d * (targetMax[node] -
targetMin[node])));
}

//Returns the avarage rmse of all the outputs.
double get_rmse(vector<vector<double>> errorset)
{
      int nsize = static_cast<int>(errorset.size()) / 2;
      vector<double> sum;
      sum.resize(nsize);
      //Rmse for each output.
      for (int y = 0; y < static_cast<int>(errorset.size()); y = y + 2)
{
            for (int e = 0; e < static_cast<int>(errorset[y].size());
e++) {

                  sum[y / 2] = sum[y / 2] + pow((errorset[y][e] -
errorset[y + 1][e]), 2);

            }
            sum[y / 2] = sqrt(sum[y / 2] / errorset[y].size());
      }
      double avg = 0;
      //Average rmse for all outputs.
      for (int s = 0; s < nsize; s++) {
            avg = avg + sum[s];
      }
      avg = avg / nsize;

      return avg;
}

//Seperate data as 70/15/15
void seperate_data(void)
{

      validateInput.resize(dataInput.size());
      testInput.resize(dataInput.size());
      validateTarget.resize(dataTarget.size());
      testTarget.resize(dataTarget.size());

      int tr = dataInput[0].size() / 100 * 70;//Divide 70
      int vl = dataInput[0].size() / 100 * 15;//          15
      int ts = dataInput[0].size() - (tr + vl);//      and rest
```

```
	for (int i = 0; i < dataInput.size(); i++) {
		validateInput[i].resize(vl);
		testInput[i].resize(ts);
	}
	for (int t = 0; t < dataTarget.size(); t++) {
		validateTarget[t].resize(vl);
		testTarget[t].resize(ts);
	}

	for (int i = 0; i < dataInput.size(); i++) {
		for (int j = tr; j < dataInput[i].size(); j++) {

			if (j < (tr + vl))
				validateInput[i][j - tr] = dataInput[i][j];
			if ((j >= (tr + vl)) && (j < (tr + vl + ts + 1)))
				testInput[i][j - (tr + vl)] = dataInput[i][j];
		}
		dataInput[i].resize(tr);
	}

	for (int i = 0; i < dataTarget.size(); i++) {
		for (int j = tr; j < dataTarget[i].size(); j++) {

			if (j < (tr + vl))
				validateTarget[i][j - tr] = dataTarget[i][j];
			if ((j >= (tr + vl)) && (j < (tr + vl + ts + 1)))
				testTarget[i][j - (tr + vl)] = dataTarget[i][j];
		}
		dataTarget[i].resize(tr);
	}


	return;
}
/*-------------------------------------------------------------------------
----------
#	End of program
-------------------------------------------------------------------------
--------*/
```