ARTIFICIAL NEURAL NETWORKS ASSIGNMENT REPORT

A REPORT

submitted as part of the assignment

in

CE889-7-AU : Neural Networks and Deep Learning

by

ISHWAR VENUGOPAL

(1906084)

Group Members for Deep Learning project:
Chayabhan Limpabandhu
Mohammed A Muhammedali

School of Computer Science and Electronic Engineering
University of Essex

January 2020

ABSTRACT

Artificial Neural Networks were introduced as a method to approximate the actual working of the human brain. It has a wide range of applications in the day-to-day life of a modern human being, With increase in the number of researchers in this field, different types of artificial neural network architectures have been developed. In this work, we have implemented a feed-forward neural network with back-propagation on a mobile robot in the laboratory. The network was trained using the data collected from the sensors of the robot in a separate run. Also, we aimed to create a Deep Learning neural network architecture for a Kaggle competition which was focused on prediciting the sales value of Rossmann stores. With the data set available on the Kaggle platform, different data cleaning processes were tested and a deep learning model was implemented on the processed data set. With increase in the advent of data-driven technologies, artificial neural networks and deep learning techniques are widely sought after for solving complex problems.

CONTENTS

1. INTRODUCTION

The human brain is one of the most complex working systems in nature. It deals with numerous complex tasks that a human body does every instant [14, 9]. Several attempts have been made to mimic the working functionality of the human brain [18, 12, 25, 33, 16, 27]. One of the very initial attempts to recreating biological neurons in a computational atmosphere was aimed at trying to imitate only certain specific parts of the human nervous system, like dendrites, axons and cell bodies [17]. As the knowledge regarding the working methodology of the brain is limited, simplified mathematical models were used to understand the flow of information in the human brain. An interconnected artificial neuron system was one of the very first attempts towards achieving an artificial neural network to solve any computable function [18]. Most of these natural processes could be mimicked by coming up with a function which would result in an output value from a list of weighted input signals, based on some threshold condition or bias [1]. Even this extremely simplified model of the human nervous system was able to solve many complicated problems [1].

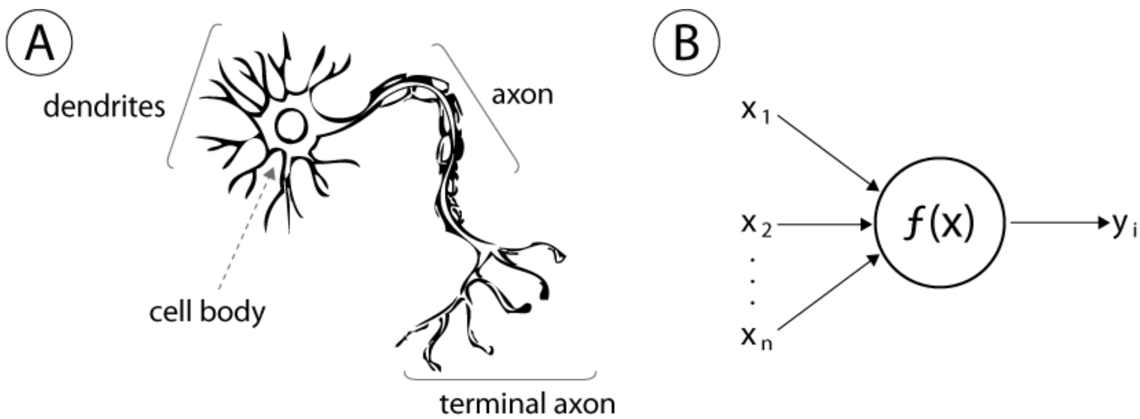

Figure 1.1: (A) represents a simple model of the human brain cell and (B) shows how an artificial neuron is represented in an Artificial Neural Network (ANN) system [17]

In this era where data is of prime importance, techniques similar to the working of the nervous system can be used to produce fruitful results which could have real-life applications. Artificial Neural Networks was one such approach. After the work published by [18] gave a whole new perspective to Artificial Intelligence and Neural Networks, further works were focused on developing on the idea and improving the model. In [12], the concept of a weighted synaptic response between neurons was introduced as a learning rule, which further helped in asserting importance of particular neurons over others in an artificial neural network system. The work done in [25] aimed at now answering the prime questions like how an information in a natural environment is sensed, stored and is made to influence the consequent actions. It led to the development of a probabilistic model for data storage. How an artificial neural network system learns as compared to the actual behaviour of natural nervous systems, was also an area of pioneer research. Several researchers have come up with improved versions of learning rules [33, 16]. The concept of back-propagation for the learning process was introduced in [27, 26]. It involved computing the new weights between neurons using techniques like gradient descent among others. Thereafter, researchers have come up with various models and variations of artificial neural network systems, which will be looked into in the next section.

1.1 Types of Neural Networks

There are different types of neural networks each suitable for some particular application [11]. The oldest and the simplest type of an artificial neural network is the perceptron. The single neuron perceptron then evolved to a multi layer perceptron (MLP) with the introduction of the concept of layers [24]. This kind of neural network has an input layer, hidden layers and an output layer; and has been widely used for applications in computer vision, natural language processing and acts as the basis for other neural network architectures.

For image and video processing, a different neural network architecture was developed which was called the convolution neural networks which implements convolution layers. This has given very productive results in the areas of computer vision and in finding patterns or characteristics of images [34].

The concept of using past instances of a state to predict its future gave rise to the

Recurrent Neural Network architecture. They work on temporal data by using techniques involving state matrices. This has been widely applied to to make Stock Market Predictions and time based data predictions [31].

The task of compressing data without the loss of quality gave rise to a different neural network architecture called autoencoders. With identical input and output layers, these techniques help preserve only the relevant features of an object and thus helping represent large amount of data in a much more compressed manner without losing any relevant information [5].

1.2 Deep Learning

In recent years, the researchers across the globe have been working on deep artificial neural networks. Such deep networks began to claim worldwide coverage after winning numerous competitions in pattern recognition and machine learning problems [29]. The major difference in a simple neural network and a deep neural network is in the fact that the term "deep" corresponds to more number of hidden layers as the name suggests.

Artificial Neural Network Models with many successive nonlinear layers of neurons started being developed in the 1960-70s [29]. Applying back propagation algorithm to multiple-layered deep artificial neural networks were a tough task at a time when the computation infrastructure was not as developed as it is in recent times. Deep Learning started becoming a technologically feasible technique with the help of unsupervised learning as demonstrated in [28] and [23]. In recent years, deep learning techniques attracted widespread attention when it outperformed other machine learning techniques by a large margin in various important applications [30, 32]. Deep learning artificial neural networks have also claimed to achieve human-like visual pattern recognition results in limited domains [8].

1.3 Applications of Neural Networks

The artificial neural network systems have a wide range of applications. It has been used widely in the medical field to detect a particular medical condition of a patient based on different features or symptoms shown by the patient [4]. To understand the diversity of its applications, we can also draw our attention towards recognizing the emotion in speech using neural networks [20]. As we can see, the applications are not limited. However we could broadly classify the applications of artificial neural networks as system modelling/approximation [6], classification/recognition [21, 22], optimisation [13, 15], behaviour based robotic control [7, 19] and financial forecasting [10].

2. NEURAL NETWORK ARCHITECTURE AND ALGORITHM

The following feedforward neural network with back-propagation was used for training the data from the sensor readings:

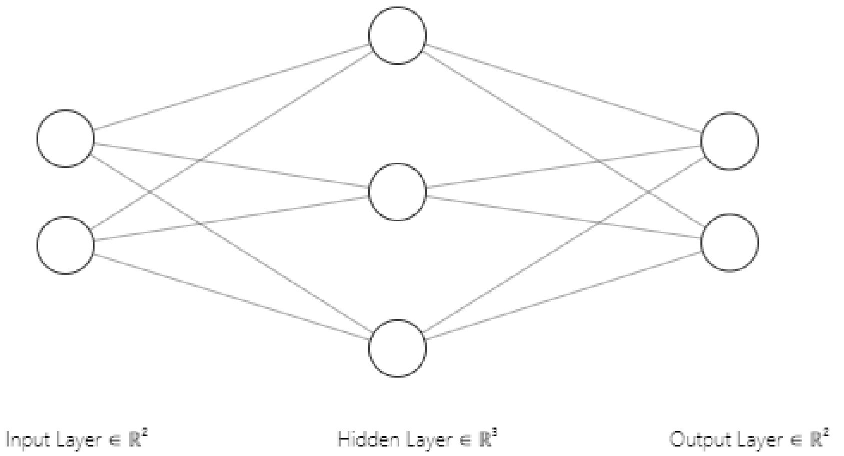

Figure 2.1: The neural network architecture with two input neurons, one hidden layer with three hidden neurons and two output neurons, used in this assignment [2]

The data corresponding to the left-edge following behaviour of the robot was collected for random orientations. This is a supervised learning problem. The left sensor reading and the front sensor readings formed the input variables, while the left motor speed and the right motor speed were the output variables. This data was then cleaned by removing duplicates and by normalising each column value to values between 0 and 1. For the normalisation procedure, the following equation was applied to each column:

$$New\ value = \frac{Old\ value - Minimum\ value}{Maximum\ value - Minimum\ Value} \tag{2.1}$$

Here, the maximum and minimum values correspond to the corresponding values in each column respectively. Normalisation between 0 and 1 was done because the activation function used for the hidden neurons and the output neurons are sigmoid functions. In order to facilitate the calculations during the feedforward process, the input required to be in this range of values. Before normalisation, the entire data was divided into training data (70 %), validation data (15 %) and the test data (15 %). This was done using the 'split' function available in Python under the 'numpy' package. This function randomly shuffles the rows before splitting. This ensures that is no bias towards any particular pattern of occurrence. The cleaned data looks similar to the sample shown in 2.2.

Left_Distance_Reading	Front_Distance_Reading	Left_Motor_Speed	Right_Motor_Speed
0.05210832	0.408575663	0.003194562	0.16279554
0.068115531	0.082615196	0.478225868	0.153340148
0.037113288	0.378305252	0.003194562	0.124500729
0.045668173	0.572484618	0.003194562	0.127529483
0.042550583	0.657287173	0.003194562	0.124500729
0.067049613	0.233616402	0.003194562	0.244609764
0.066919926	0.081032243	0.497291012	0.14523823
0.027682573	0.299750189	0.171624626	0.124500729
0.115542026	0.244899626	0.003194562	0.510141594

Figure 2.2: A Sample of the normalised training data

The first two columns in 2.2 corresponds to the input values and the other two columns corresponds to the output values. This would be the data we would be using for the training purposes and for testing the accuracy of our Artificial Neural Network model before using it on the robot itself.

2.1 Algorithm

The basic algorithm used in the training process can be demonstrated as follows:

2.1.1 Step 1: Creating the neural network design

The parameter to be tuned here was the number of hidden neurons required. The program was run with different number of hidden neurons ranging from 1 to 10 by keeping all other parameters the same [namely, η = 0.1, λ = 0.4 and α = 0.6].

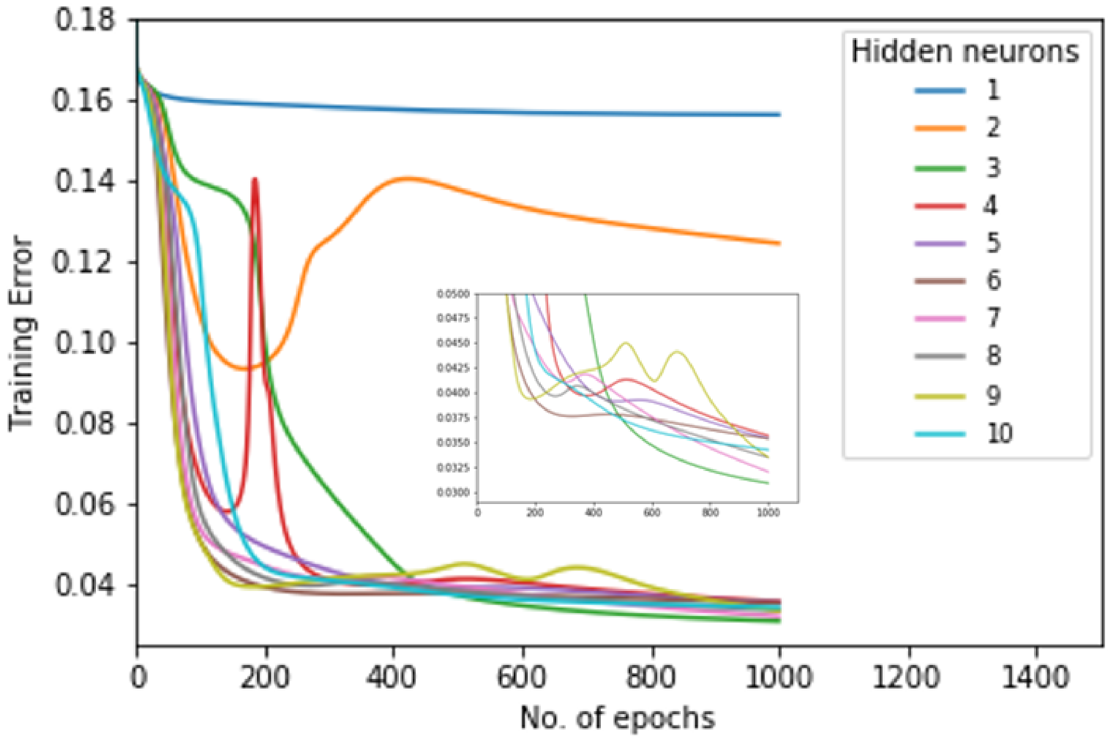

Figure 2.3: A plot between the average training error in each epoch versus the number of epochs, for different values of hidden neurons in a single layer (η = 0.1, λ = 0.4 and α = 0.6).

It was observed that the artificial neural network system with three hidden neurons gave the lowest training error [2.3]. Hence a design architecture with a single layer having three hidden neurons was selected. The output weights and the hidden weights were initialized as random numbers between 0 and 1. This was done using the 'random()' function in C++. This was intuitively a better way of initializing the weights instead of making all the values to a constant number like zero.

2.1.2 Step 2: Feed-forward process

We first calculate the net input (v_i) from the input neurons using the following formula:

$$v_i^h = \sum_{j=0}^{m} w_{ij}^h x_j \tag{2.2}$$

Here m is the number of inputs, x_j corresponds to each of the inputs and w_{ij}^h correspond to the hidden weights. Then we apply the Sigmoid activation function on the net input to obtain the value of each hidden neuron. It is done as follows:

$$\phi(v) = \frac{1}{1 + exp(-\lambda v)} \tag{2.3}$$

Here λ is the regularization parameter which is introduced to reduce over-fitting. Similarly we continue with the same process for finding the value for the output neurons.

$$v_i = \sum_{j=0}^{n} w_{ij} h_j \tag{2.4}$$

$$\phi(v) = \frac{1}{1 + exp(-\lambda v)} \tag{2.5}$$

where, n is the total number of hidden neurons and h_j corresponds to each of the hidden neurons. We have now completed our feed-forward process.

2.1.3 Step 3: Calculating the error in the output

We first calculate the simple error in the output obtained from the feed-forward process as compared to the actual output:

$$Error\ function = Actual\, output - Calculated\, output \tag{2.6}$$

For each of the output variables we calculate the Root-mean-square error and then take the average of the total number of rows. We then take the mean of both the average error values obtained for each of the outputs, which gives our average training error for that particular epoch. All the graphs plotted in this report have used this average training error as the data.

2.1.4 Step 4: Back-propagation

We now proceed towards finding the local gradient associated with each output neuron:

$$\delta_k(t) = \lambda.\phi(v_k(t)).[1 - \phi(v_k(t))].e_k(t) \tag{2.7}$$

Here e_k(t) corresponds to the associated error function. The local gradients of the hidden neurons are also calculated in a similar fashion:

$$\delta_k^h(t) = \lambda.\phi^h(v_k^h(t)).[1 - \phi^h(v_k^h(t))].[\sum_{k=1}^{l} \delta_k(t).w_{ki}(t)] \tag{2.8}$$

where l is the number of output neurons. The regularization parameter [λ] could also be tuned. In order to find the optimum λ value, different ensembles were run by varying lambda from 0.1 to 0.9. All other parameters were kept constant [namely, η=0.9 and α=0.6].

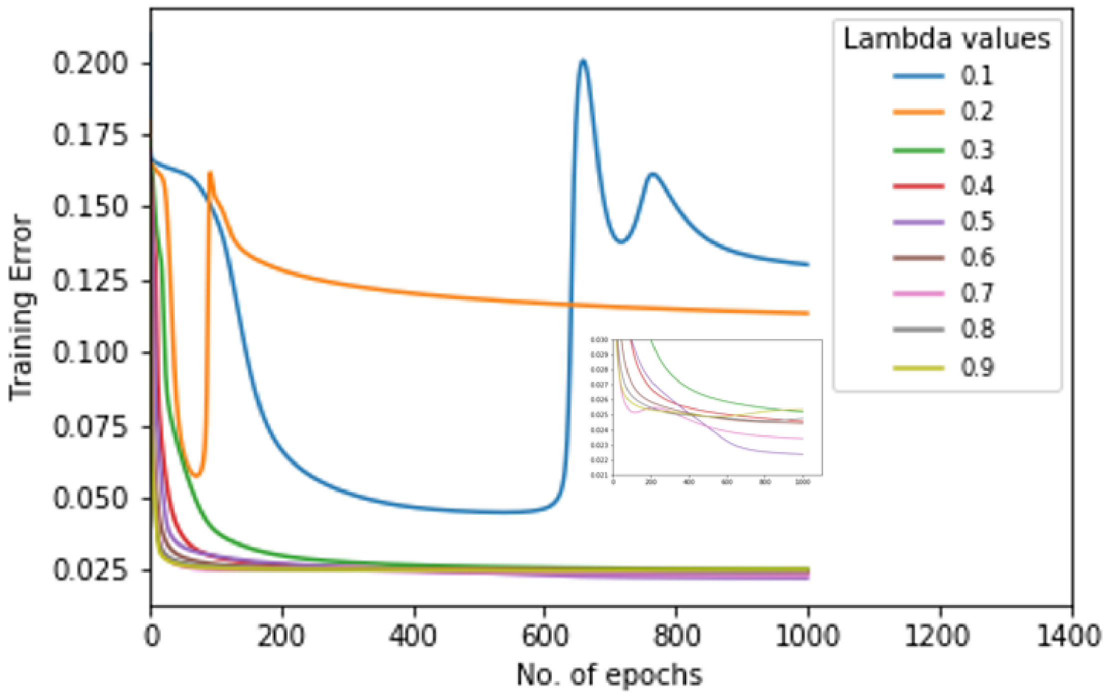

Figure 2.4: A plot between the average training error in each epoch versus the number of epochs, for different values of λ (η = 0.9, and α = 0.6).

It was observed that the value of 0.5 gave the least training error. The next step was to calculate the delta weights:

$$\Delta w_{ki}(t) = -\eta \frac{\partial \varepsilon(t)}{\partial w_{ki}(t)} = \eta \delta_k(t) h_i(t) \tag{2.9}$$

$$\Delta w_{ki}^h(t) = -\eta \frac{\partial \varepsilon(t)}{\partial w_{ki}^h(t)} = \eta \delta_k^h(t) x_j(t) \tag{2.10}$$

To speed up the convergence, a momentum term (α) is added. This gives rise to the following equations:

$$\Delta w_{ki}(t) = \eta \delta_k(t) h_i(t) + \alpha \Delta w_{ki}(t-1) \tag{2.11}$$

$$\Delta w_{ki}^h(t) = \eta \delta_k^h(t) x_j(t) + \Delta w_{ki}^h(t-1) \tag{2.12}$$

The parameters η (learning rate) and α (momentum term) were tuned separately. At first, different ensembles were run by varying the value of eta from 0.1 to 0.9 and keeping all other parameters constant [λ = 0.4 and α = 0.6].

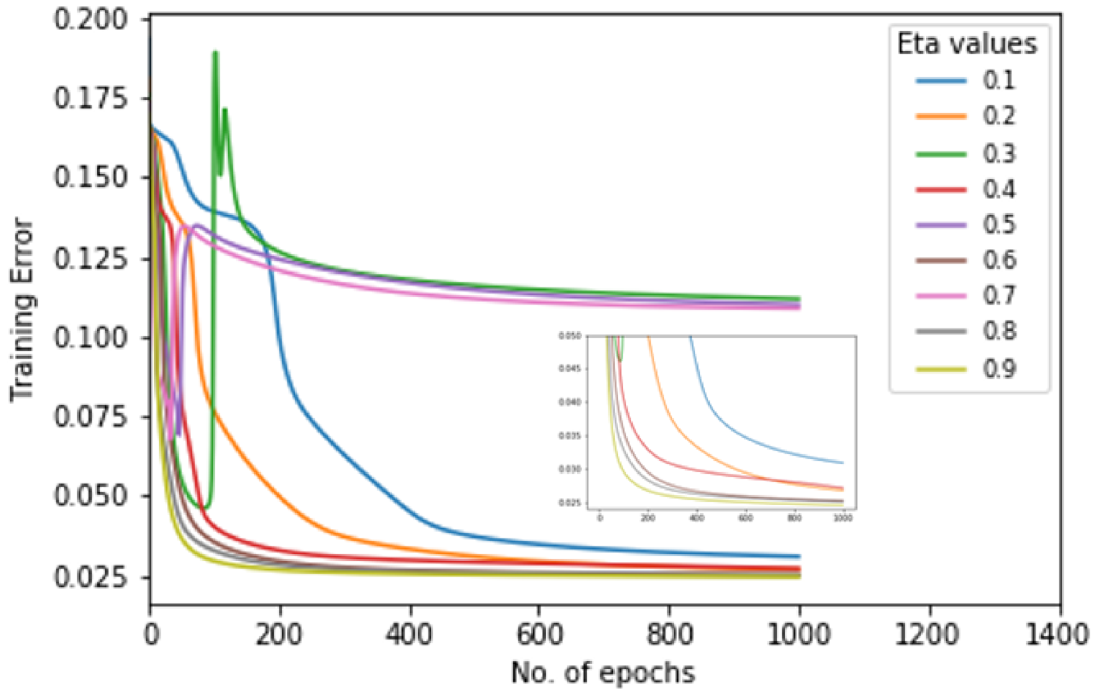

Figure 2.5: A plot between the average training error in each epoch versus the number of epochs, for different values of η (λ = 0.4 and α = 0.6).

It was observed that a value of 0.9 for η gave the least training error. A similar procedure was applied for tuning the α values. The results observed were as displayed in the [2.6].

The value of α that gave the least training error was 0.6.

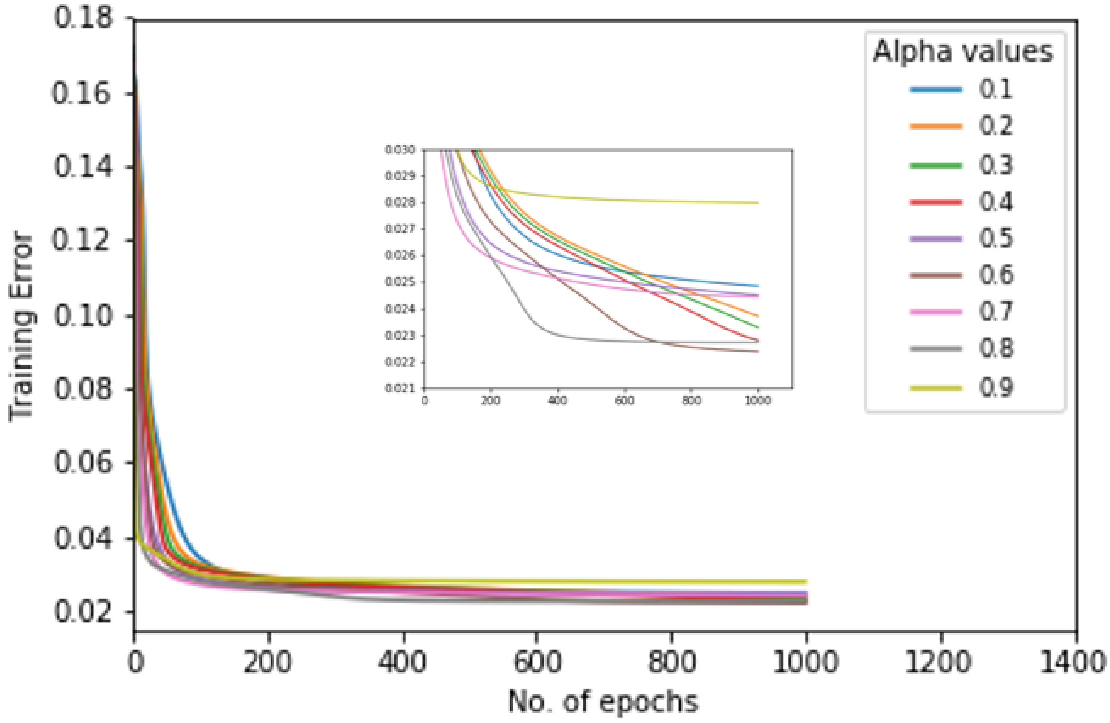

Figure 2.6: A plot between the average training error in each epoch versus the number of epochs, for different values of α (λ = 0.4 and η = 0.5).

2.1.5 Step 5: Weight Update

The next step was to update the weights in accordance to the delta weights and local gradients calculated earlier in this epoch. The weight update is done by adding the delta weights calculated in the previous steps to the current weights. This is done for both the output weights as well as the hidden weights.

2.1.6 Step 6: Repeat the process

The feed-forward and the back-propagation steps were carried out until a stopping criteria was reached. Ideally, the validation error would have increased at some point of time and would facilitate as the stopping criteria for the training process. But in this case, it was observed that the validation error did not have any steady increase in most of the cases. For a typical scenario with five hidden neurons (one layer), η=0.8, $\lambda = 0.5$ and $\alpha = 0.7$; the training errors and the validation errors formed a plot like [2.7]. As per this observation, 1000 epochs was seen as a good number for the total number of epochs in the training process.

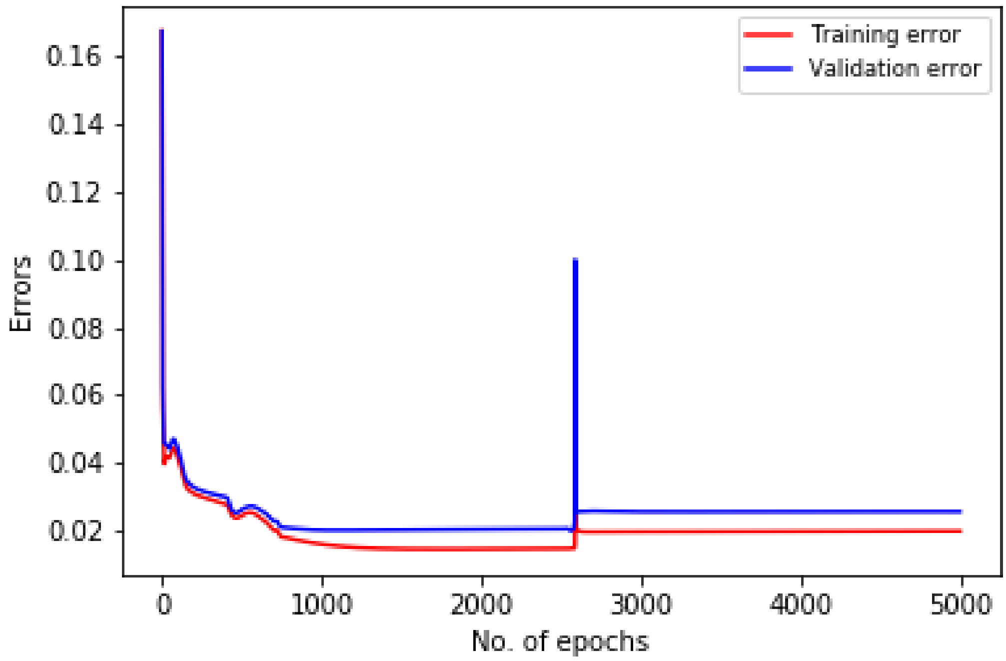

Figure 2.7: A plot between the average training error, validation error versus the number of epochs for the following values of the parameters: number of hidden neurons: 5 (one layer), η=0.8, λ = 0.5 and α = 0.7.

2.2 Implementation on the robot

The task at hand was to implement a left-edge following behaviour on a robot using a feed-forward process with the weights obtained from the training process illustrated above. The robots used in the laboratory for this purpose are the ones shown in Figure 2.8 with eight sonar sensors which has also been illustrated in the diagram.

The sensors at -90^o and -50^o have been combined and used as the left sensor reading, whereas the sensors at -30^o and -10^o have been combined and used as the front sensor reading. The readings are first obtained from the robot in real-time and then normalised. The final weights from the training process are applied over these normalized inputs to obtain normalized output values. These values are then de-normalized to obtain the final left motor speed and the right motor speed. This process is carried out as long as the robot

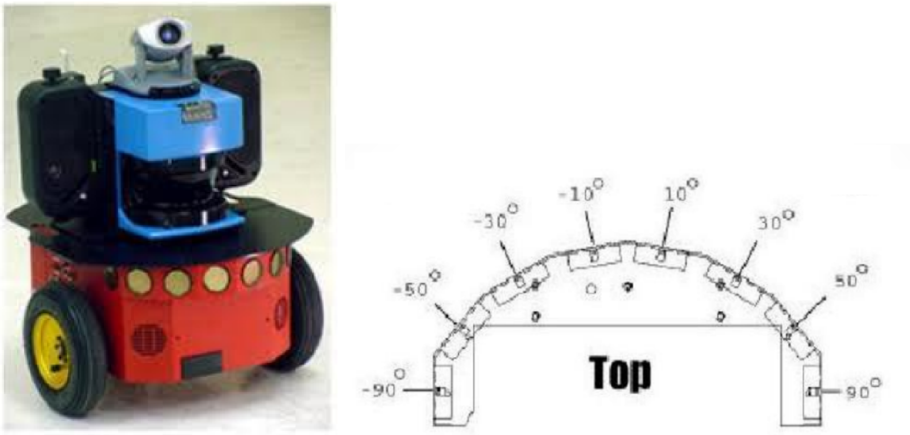

Figure 2.8: (a) Mobile robot - Pioneer which has been used for the experiments (b) The orientation of the sonar sensors in front of the robot

is turned on, and is ideally expected to follow a left edge following behaviour similar to the scenario when the data was collected.

3. DEEP LEARNING ARCHITECTURE

The task at hand was to work in groups of three and submit an entry into the Kaggle Deep Learning competition for the problem posed by Rossmann stores. Rossmann stores had been tasked with predicting their daily sales for up to six weeks in advance. Store sales are influenced by many factors, including promotions, competition, school and state holidays, seasonality, and locality. With the data provided on their Kaggle webpage [3], the task was to apply a suitable deep learning model to predict the sales.

The task was divided among three of us, and we worked separately on different data cleaning processes and my task in particular was to build a suitable deep learning model using tensor flow. The entire task can be broadly divided as data cleaning followed by building and testing the Deep Learning model.

3.1 Data Cleaning

The data is divided in to three files, namely 'train.csv' (which contains the historical data including Sales), 'test.csv' (which has historical data excluding Sales) and 'store.csv' (which contains supplemental information regarding the stores).

The raw data supplied in the training data have the following columns enlisted as features [[3]]:

- Id - an Id used to identify a (Store, Date) duple within the test set
- Store - a unique Id for each store
- Sales - the turnover for any particular day

- Customers - the number of customers on a particular day
- Open - an indicator for whether the store was open
- StateHoliday - indicates a state holiday
- SchoolHoliday - indicates if the (Store, Date) was affected by the closure of public schools
- StoreType - Different store models: a, b, c, d
- Assortment - describes an assortment level: a = basic, b = extra, c = extended
- CompetitionDistance - distance in meters to the nearest competitor store
- CompetitionOpenSince[Month/Year] - gives the approximate year and month of the time the nearest competitor was opened
- Promo - indicates whether a store is running a promo on that day
- Promo2 - Promo2 is a continuing and consecutive promotion for some stores
- Promo2Since[Year/Week] - describes the year and calendar week when the store started participating in Promo2
- PromoInterval - describes the consecutive intervals Promo2 is started, naming the months the promotion is started anew.

We had to predict the 'Sales' column for the testing data. The training and testing data were cleaned in different steps as demonstrated in the following sentences:

```
In [68]: store.isnull().sum()

Out[68]: Store                          0
         StoreType                      0
         Assortment                     0
         CompetitionDistance            3
         CompetitionOpenSinceMonth    354
         CompetitionOpenSinceYear     354
         Promo2                         0
         Promo2SinceWeek              544
         Promo2SinceYear              544
         PromoInterval                544
         dtype: int64
```

Figure 3.1: Identifying the null values in the data

1) The null values in the training and testing data were identified and replaced with the mode value (i.e. the value that occurred the most) of the column [3.1].

2) As our deep learning neural network architecture takes only numeric values, we had to convert all the columns with type 'object' to numeric [[3.2], refer Appendix for the code].

```
In [76]: store.dtypes

Out[76]: Store                          int64
         StoreType                     object
         Assortment                    object
         CompetitionDistance          float64
         CompetitionOpenSinceMonth    float64
         CompetitionOpenSinceYear     float64
         Promo2                         int64
         Promo2SinceWeek              float64
         Promo2SinceYear              float64
         PromoInterval                 object
         dtype: object
```

Figure 3.2: Identifying the data types of all columns

3) The challenging part was the conversion of the 'Date' column without the loss of information. This was done by converting the 'year-month-day' format to numbers after removing the '-' symbol separating the year, month and the date [3.3].

```
In [86]: train['Date'] = train['Date'].map(lambda x: ''.join(x.split('-')))
         train.head()
```

Out[86]:

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
0	1	5	20150731	5263	555	1	1	0	1
1	2	5	20150731	6064	625	1	1	0	1
2	3	5	20150731	8314	821	1	1	0	1
3	4	5	20150731	13995	1498	1	1	0	1
4	5	5	20150731	4822	559	1	1	0	1

Figure 3.3: Converting the dates to numeric values

3.2 Deep learning network structure

A deep neural network architecture was implemented using the TensorFlow and Keras package in Python. We used the Sequential model and functions like Dense and Activation for further implementation tasks. After reading from the cleaned data set, a network structure similar to Figure.3.4 was implemented [refer Appendix for code]. The number of hidden layers and the number of hidden neurons in each layer was varied and tried upon with.

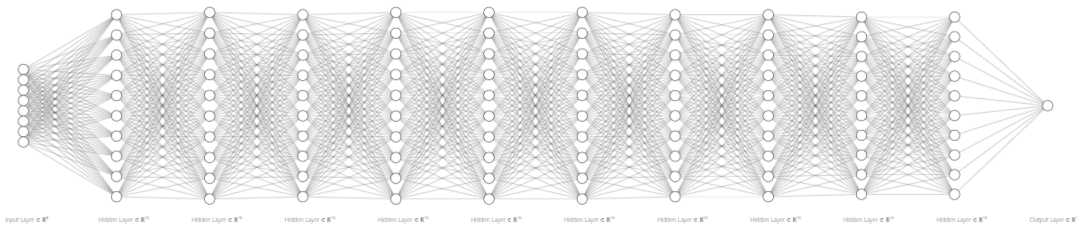

Figure 3.4: The basic structure of the Deep neural network implemented [2]

The activation function used in the final implementation was the ReLU (Rectified Linear Unit) Activation function for all the hidden layers, and Linear activation function for the output layer.

The error metrics used is RMSprop because using the Root-mean-square error gives more weight to large errors (as each error is squared) and thus gives a better pathway in upcoming epochs to reduce that error and reach an optimum stage. The efficiency of the system in each epoch was monitored using the accuracy measure as well as the mean square error measure. This was the basic architecture used to approach the problem. A sample run of the deep learning neural network system is shown in Figure.3.5. The fitted model was then used to predict the 'Sales' column of the test data.

Training the model to fit the data ¶

```
model.fit(x,y,epochs=250,batch_size=50)
```

```
WARNING:tensorflow:From c:\python37\lib\site-packages\keras\backend\tensorflow_backend.py:1033: The name tf.assign_add is depre
cated. Please use tf.compat.v1.assign_add instead.

WARNING:tensorflow:From c:\python37\lib\site-packages\keras\backend\tensorflow_backend.py:1020: The name tf.assign is deprecate
d. Please use tf.compat.v1.assign instead.

Epoch 1/250
1017209/1017209 [==============================] - 17s 17us/step - loss: 110584745.1261 - acc: 7.9335e-04 - mean_squared_error:
110584745.1261
Epoch 2/250
1017209/1017209 [==============================] - 17s 17us/step - loss: 14823616.5325 - acc: 1.2190e-04 - mean_squared_error:
14823616.5325
Epoch 3/250
1017209/1017209 [==============================] - 17s 16us/step - loss: 14823450.1465 - acc: 1.3763e-04 - mean_squared_error:
14823450.1465
```

Figure 3.5: A sample run for the deep learning neural network that was implemented.

4. RESULTS AND DISCUSSIONS

4.1 Neural Network Implementation

With reference to the tuning procedures described in the last section, the final parameters were set as follows:

- The total number of epochs for the training procedure = 1000
- Number of hidden neurons=3
- $\eta = 0.9$
- $\lambda = 0.5$
- $\alpha = 0.6$

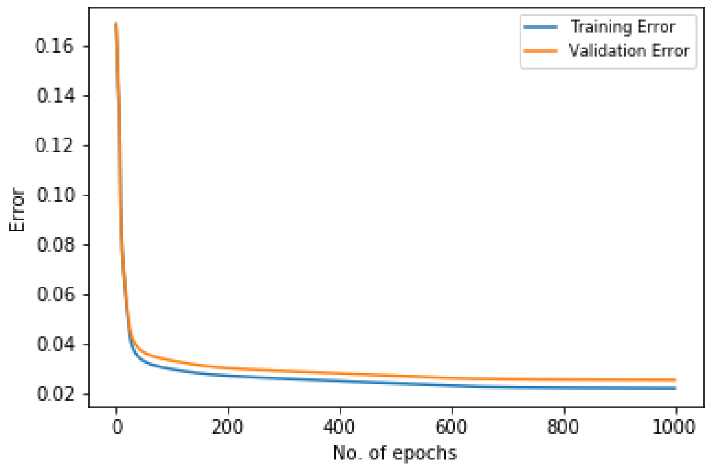

Figure 4.1: The training and validation errors with 3 hidden neurons (single layer), $\eta = 0.9$, $\lambda = 0.5$ and $\alpha = 0.6$

A typical run of the training process with these parameters gave an average testing error of 0.10622. The plot between the training error and the validation error versus the number of epochs is shown in Figure.4.1. It can be observed that the average validation error in each epoch is always higher than the average training error in that particular epoch. The errors started of from around 0.16 and reduced gradually before saturating at a value near 0.02. The final output weights and the hidden weights were stored to a csv file.

To implement this architecture on a robot in real-time, the final stored weights were read into the program and a feedforward process was implemented in real-time. The combination of sensors used as the input were finalised after trying out different sensors in a trial-and-error fashion. The robot did not show a smooth turn at the corners. It was closer to the wall than it is supposed to be while turning at the corners. This can be due to the fact that there is a certain amount of testing error associated with the model used, which in turn makes the robot keep a distance from the wall which in reality is much closer than it is meant to be. Snapshots for the robot motion as observed on the simulator are shown in Figure.4.3.

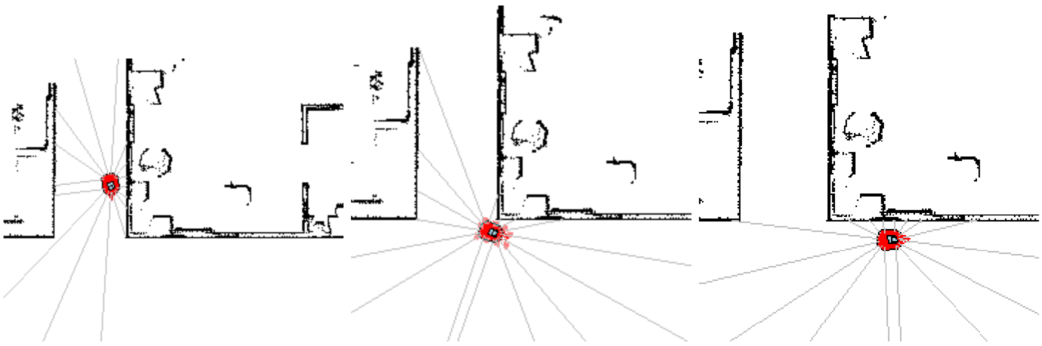

Figure 4.2: Screenshots from a trial run on the simulator for a left-edge following behaviour (from left to right)

The neural network model implemented in this experiment did not have the best of performances. This can be due to many reasons:

- The error metrics used during the training process may not be the optimum technique for this problem
- Having just a single hidden layer could have influenced the error obtained in the final implementation. Multiple hidden layers might have improved the performance.

4.2 Deep Learning Competition

We tried running the previously discussed architecture with different number of hidden neurons and different number of layers. But the obtained accuracy and the quality of predictions were very low. The final accuracy obtained in the final epoch was 1.3173e-04. The mean square error after the final epoch was 14823410.1406.

Predicting the sales value for the cleaned test dataset

In [10]:
```
result=df1
result['Sales']=model.predict(x1)
result.to_csv('prediction.csv')
result.head()
```

Out[10]:

	Id	Store	DayOfWeek	Date	Open	Promo	StateHoliday	SchoolHoliday	Sales
0	1	1	4	20150917	1.0	1	0	0	1.073729e+09
1	2	3	4	20150917	1.0	1	0	0	1.073729e+09
2	3	7	4	20150917	1.0	1	0	0	1.073729e+09
3	4	8	4	20150917	1.0	1	0	0	1.073729e+09
4	5	9	4	20150917	1.0	1	0	0	1.073729e+09

Figure 4.3: Sample output of the predictions obtained from the model

When submitted on Kaggle, it gave a Private score of 201193.79710 and a Public score of 191686.80088. This poor performance could be due to different reasons:

- The data processing method we used for dates was not every efficient. Instead of converting the dates to numeric by removing the '-' symbol, we should have split the date values as month, year and day. This would have resulted in a much better way to preserve the data without any loss of the properties of the data. In our method, we actually lacked the cyclic property present in dates. For example, the distance between December and January should be one. But as per our method, the distance value is much different than in reality.
- More efficient ways should have been used to convert columns with data type 'object' to a numeric data type.
- It can be observed that the data cleaning methodology that we select has a huge impact on the final results obtained.

- From the results we have obtained, we should be able to conclude that RMSProp might not have been the best error metric to use for this problem.

5. CONCLUSIONS

A feed-forward neural network with back-propagation was modelled and trained on a data collected from the sensor readings of a robot. This fitted model was then used to run the robot in real-time and get a left edge following behaviour. It was seen that the proper tuning of the parameters gave a very low training input. The robot seemed to work within the error limit that the network architecture obtained.

In the Deep Learning competition, a deep neural network architecture was implemented using TensorFlow and Keras. Even though the results obtained from the model was not up to the mark, this provided an insight into how deep neural network models work and how such models could be manipulated to obtain results in the desired error limit.

5.1 Future directions

The work carried out in this assignment can be taken forward by trying out the neural network training process with multiple layers and with multiple combinations for the number of hidden neurons in each layer. Adding a bias for the calculations can help achieve a better grasp on the final performance. Also, proper normalisation and de-normalisation procedures can be applied while implementing on the robot, which can take into consideration the real time errors generated by the robot. Better performance can also be generated by combining the reading of the laser sensors as well along with the current sonar sensor readings. With reference to the Deep Learning Project, more methods of cleaning and effectively processing the data can be investigated.

BIBLIOGRAPHY

[1] The differences between artificial and biological neural networks. `https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks`

[2] Nn-svg. `http://alexlenail.me/NN-SVG/index.html`.

[3] Rossmann store sales. `https://www.kaggle.com/c/rossmann-store-sales/data`.

[4] Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. Artificial neural networks in medical diagnosis. *Journal of Applied Biomedicine*, 2013.

[5] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. *JMLR: Workshop and Conference Proceedings*, 2012.

[6] D. S. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2, 1988.

[7] M. Carreras, J. Batlle, P. Ridao, and G. N. Roberts. An overview on behaviour-based methods for auv control. *IFAC Proceedings Volumes*, 8 2000.

[8] D. C. Ciresan, U. Meier, J. Masci, Gambardella, L. M., and J Schmidhuber. Flexible, high performance convolutional neural networks for image classification. *Intl. joint conference on artificial intelligence*, 2011.

[9] R.S.J Frackowiak. *Human Brain Function*. Elseveir Academic Press, 2004.

[10] Edward Gately. *Neural Networks for Financial Forecasting*. John Wiley Sons, Inc.605 Third Ave. New York, NYUnited States, 1995.

[11] Simon Haykin. *Neural Networks and Learning Machines*. Pearson Prentice Hall, 2009.

[12] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. New York: Wiley, 1949.

[13] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimisation problems. *Biological Cybernetics*, 52, 1985.

[14] Mark H Johnson. Development of human brain functions. *Biological Psychiatry*, 54, 12 2003.

[15] S. Kirhpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983.

[16] T. Kohonen. Self-organised formation of topologically correct feature maps. *Biological Cybernetics*, 43, 1982.

[17] Juxi Leitner. From vision to actions: Towards adaptive and autonomous humanoid robots. 05 2015.

[18] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity,. *Bulletin of Mathematical Biophysics*, 5, 1943.

[19] Yong-Kyun Na and Se-Young Oh. Hybrid control for autonomous mobile robot navigation using neural network based behavior modules and environment classification. *Autonomous Robots*, 15, 2003.

[20] J. Nicholson, K. Takahashi, and R. Nakatsu. Emotion recognition in speech using neural networks. *Neural Computing Applications*, 12 2000.

[21] Guobin Ou, Yi Lu, and Murphey. Multi-class pattern classification using neural networks. *Complex SystemsPattern Recognition*, 40, 1 2007.

[22] Eddy Patuwo, Michael Y. Hu, and Ming S. Hung. Two-group classification using neural networks. *Decision Sciences*, 7 1993.

[23] Ranzato, M. A., F. Huang, Y. Boureau, and Y. LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. *In Proc.computer vision and pattern recognition conference*, 2007.

[24] Martin Riedmiller. Advanced supervised learning in multi-layer perceptrons — from backpropagation to adaptive learning algorithms. *Computer Standards Interfaces*, 16, 1994.

[25] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 1958.

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations of back-propagation errors. *Nature*, 323, 1986.

[27] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986.

[28] J Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4, 1992.

[29] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61, 1 2015.

[30] Schölkopf, Burges, C. J. C., and A. J Smola. *Advances in kernel methods—support vector learning*. Cambridge, MA: MIT Press., 1998.

[31] Ah Chung Tsoi and Andrew Back. Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15, 1997.

[32] V Vapnik. *The nature of statistical learning theory*. New York: Springer, 1995.

[33] B. Widrow and Jr. M. E. Hoff. Adaptive switching circuits. *IREWESCON Convention Record*, 1960.

[34] Hyeon-Joong Yoo. Deep convolution neural networks in computer vision. *IEIE Transactions on Smart Processing Computing*, 4, 02 2015.

Appendices

A. NEURAL NETWORK CODE

A.1 Training Process

```
#include <Aria.h>
#include <stdio.h>
#include <iostream>
#include<conio.h>
#include <ctime>    // For time()
#include <cstdlib>  // For srand() and rand()
#include<fstream>
#include<string>
#include<sstream>

using namespace std;

//Setting the parameters of the neural network
const int no_input = 2, no_output = 2, no_hneuron = 3, no_epochs =
    1000;
const double eta = 0.9, lambda = 0.3, alpha = 0.6;

class neuron
{
public:
  double value; //to store the value of a neuron
  double error; //to calculate the error for each predicted output
  double wh[no_input], w[no_hneuron]; //the hidden weights and the
    output weights
  double delta_wh[no_input], delta_w[no_hneuron]; //delta weights
  double delta_wh_old[no_input], delta_w_old[no_hneuron]; //to save the
```

```
delta weights of previous time step
double lgrad_hid, lgrad_out; //local gradients
void initialize_weights(int layer) //function to initialize all the weights, delta weights, local gradients and errors
{
  int i,j;
  for (j = 0;j < no_input;j++)
  {
    if (layer == 2)
      wh[j] = (double(rand()) / double(RAND_MAX)); //Random number between 0 and 1
    else
      wh[j] = 0;
    delta_wh[j] = double(0);
    delta_wh_old[j] = double(0);
  }
  for (j = 0;j < no_hneuron;j++)
  {
    if (layer == 3)
      w[j] = double(rand()) / double(RAND_MAX); //Random number between 0 and 1
    else
      w[j] = 0;
    delta_w[j] = double(0);
    delta_w_old[j] = double(0);
  }
  lgrad_hid = double(0);
  lgrad_out = double(0);
  error = double(0);
}
double activation(string func,double netinput) //function to return a value after applying the activation function
{
  double activated;
  if (func == "sigmoid")
    activated = 1 / (1 + exp(-lambda*netinput));
  else
    activated = netinput;
```

```
    return activated;
  }
};
vector<vector<double>> read_training_data(string filename) //function
    to read the training data from the csv file
{
  vector<double> row;
  vector <vector<double>> alldata;
  ifstream file(filename);
  string line, word;
  getline(file, line); //reading each line as a string
  while (getline(file, line))
  {
    row.clear();
    stringstream ss(line); //for breaking each line into words
    while (getline(ss, word, ','))
    {
      row.push_back(stod(word));
    }
    alldata.push_back(row);
  }
  return alldata;
}

void main()
{
  vector<vector<double>> training; //to save all the training data
  vector<vector<double>> input, output; //to save the input and output
    values in the training data
  vector<vector<double>> validation; //to save all the validation data
  vector<vector<double>> v_input, v_output; //to save the input and
    output values in the validation data
  vector<vector<double>> test; //to save the training data
  vector<vector<double>> test_input, test_output; //to save the input
    and output values on the testing data
  int i, j, r, k;
  double sum,train_error,val_error,test_error;
  int epoch, total_rows,val_total_rows,test_total_rows;
```

```
vector<vector<double>> wh_avg, w_avg; //Variables to store the average hidden and output weights in each epoch
srand(time(0)); //initializing a seed (with the system time) to generate a random number

epoch = 0;
ofstream errorfile_train("training_errors.csv"); //creating a file to save the training errors
ofstream errorfile_val("validation_errors.csv"); //creating a file to save the validation errors
ofstream errorfile_test("test_errors.csv"); //creating a file to save the testing errors
errorfile_train << "No. of Epochs,Training Error" << endl; //the first row with the column names
errorfile_val << "No. of Epochs,Validation Error" << endl; //the first row with the column names
errorfile_test << "Final Average Error" << endl; //the first row with the column names

ofstream h_weightfile("finalhiddenweights.csv");//file to save the final hidden weights
ofstream o_weightfile("finaloutputweights.csv");//file to save the final output weights
h_weightfile << "Hidden Neuron Number,Input neuron number,Weight Value" << endl;
o_weightfile << "Output Neuron Number,Hidden neuron number,Weight Value" << endl;

training = read_training_data("finaltrainingdata.csv"); //reading all the training data
for(i=0;i<training.size();i++) //a loop to save the inputs and outputs separately from the training data
{
  input.push_back(vector<double>());
  input[i].push_back(training[i][0]);
  input[i].push_back(training[i][1]);
  output.push_back(vector<double>());
  output[i].push_back(training[i][2]);
```

```
    output[i].push_back(training[i][3]);
  }
  total_rows = training.size(); //determining the total number of rows
   in the training data

  validation = read_training_data("finalvalidationdata.csv"); //reading
    all the validation data
  for (i = 0;i<validation.size();i++) //a loop to save the inputs and
   outputs separately from the validation data
  {
    v_input.push_back(vector<double>());
    v_input[i].push_back(validation[i][0]);
    v_input[i].push_back(validation[i][1]);
    v_output.push_back(vector<double>());
    v_output[i].push_back(validation[i][2]);
    v_output[i].push_back(validation[i][3]);
  }
  val_total_rows = validation.size(); //determining the total number of
    rows in the validation data

  test = read_training_data("finaltestdata.csv"); //reading all the
   validation data
  for (i = 0;i<test.size();i++) //a loop to save the inputs and outputs
    separately from the validation data
  {
    test_input.push_back(vector<double>());
    test_input[i].push_back(test[i][0]);
    test_input[i].push_back(test[i][1]);
    test_output.push_back(vector<double>());
    test_output[i].push_back(test[i][2]);
    test_output[i].push_back(test[i][3]);
  }
  test_total_rows = test.size(); //determining the total number of rows
    in the validation data

  vector<vector<neuron>> hidden; //hidden neurons
  vector<vector<neuron>> predicted; //neurons for predicted outputs
  vector<vector<neuron>> v_hidden; //hidden neurons for validation data
```

```
vector<vector<neuron>> v_predicted; //predicted outputs for validation data
vector<vector<neuron>> test_hidden; //hidden neurons for validation data
vector<vector<neuron>> test_predicted; //predicted outputs for validation data

for (i = 0;i < total_rows;i++) //a loop to initialize each hidden neuron and predicted output neuron
{
  hidden.push_back(vector<neuron>());
  for (j = 0;j < no_hneuron;j++)
  {
    hidden[i].push_back(neuron());
    hidden[i][j].initialize_weights(2);
  }
  predicted.push_back(vector<neuron>());
  for (j = 0;j < no_output;j++)
  {
    predicted[i].push_back(neuron());
    predicted[i][j].initialize_weights(3);
  }
}
for (i = 0;i < val_total_rows;i++) //Initializing hidden and predicted output neuron for validation data
{
  v_hidden.push_back(vector<neuron>());
  for (j = 0;j < no_hneuron;j++)
  {
    v_hidden[i].push_back(neuron());
    v_hidden[i][j].initialize_weights(2);
  }
  v_predicted.push_back(vector<neuron>());
  for (j = 0;j < no_output;j++)
  {
    v_predicted[i].push_back(neuron());
    v_predicted[i][j].initialize_weights(3);
  }
```

```
}
for (i = 0;i < test_total_rows;i++) //Initializing hidden and
 predicted output neuron for testing data
{
  test_hidden.push_back(vector<neuron>());
  for (j = 0;j < no_hneuron;j++)
  {
    test_hidden[i].push_back(neuron());
    test_hidden[i][j].initialize_weights(2);
  }
  test_predicted.push_back(vector<neuron>());
  for (j = 0;j < no_output;j++)
  {
    test_predicted[i].push_back(neuron());
    test_predicted[i][j].initialize_weights(3);
  }
}

for (r = 0;r < total_rows;r++) //Initialize the vectors for the
 average weights
{
  for (i = 0;i<no_hneuron;i++)
  {
    wh_avg.push_back(vector<double>());
    for (j = 0;j<no_input;j++)
      wh_avg[i].push_back(0);
  }
  for(i=0;i<no_output;i++)
  {
    w_avg.push_back(vector<double>());
    for (j = 0;j < no_hneuron;j++)
      w_avg[i].push_back(0);
  }

}
while (epoch < no_epochs) //the main loop that loops over each epoch
{
  for (r = 0;r < total_rows;r++) //Loops over each row in an epoch
```

```
{
  if(r!=0) //conditions to transfer the weight values from each row to the next
  {
    for (k = 0;k < no_hneuron;k++)
      for (i = 0;i < no_input;i++)
        hidden[r][k].wh[i] = hidden[r - 1][k].wh[i];
    for (k = 0;k < no_output;k++)
      for (i = 0;i < no_hneuron;i++)
        predicted[r][k].w[i] = predicted[r-1][k].w[i];
  }
  else if(r==0)
  {
    if(epoch!=0)
    {
      for (k = 0;k < no_hneuron;k++)
        for (i = 0;i < no_input;i++)
          hidden[r][k].wh[i] = hidden[total_rows-1][k].wh[i];
      for (k = 0;k < no_output;k++)
        for (i = 0;i < no_hneuron;i++)
          predicted[r][k].w[i] = predicted[total_rows - 1][k].w[i];
    }
  }

  for (k = 0;k < no_hneuron;k++) //a loop to find the value of each hidden neuron
  {
    sum = 0;
    for (i = 0;i < no_input;i++)
    {
      sum = sum + (hidden[r][k].wh[i] * input[r][i]); //net input
    }
    hidden[r][k].value = hidden[r][k].activation("sigmoid", sum); //applying activation function
  }
  for (k = 0;k < no_output;k++) //a loop to find the value of each predicted output
  {
```

```
      sum = 0;
      for (i = 0;i < no_hneuron;i++)
      {
        sum = sum + (predicted[r][k].w[i] * hidden[r][i].value); //
net input
      }
      predicted[r][k].value = predicted[r][k].activation("sigmoid",
sum); //applying activation function
    }
    for (i = 0;i<no_output;i++) //a loop to determine the errors in
output and their local gradient
    {
      predicted[r][i].error = output[r][i] - predicted[r][i].value;
      predicted[r][i].lgrad_out = lambda*predicted[r][i].value * (1 -
 predicted[r][i].value)*predicted[r][i].error;
    }
    sum = 0;
    for (i = 0;i<no_hneuron;i++) //finding the local gradient of each
 hidden neuron
    {
      for (j = 0;j<no_output;j++)
      {
        sum += predicted[r][j].lgrad_out* predicted[r][j].w[i];
      }
      hidden[r][i].lgrad_hid = lambda*hidden[r][i].value * (1 -
hidden[r][i].value)*sum;
    }
    for (i = 0;i < no_output;i++) //a loop to calculate the delta
weights for output neurons
    {
      for (j = 0;j < no_hneuron;j++)
      {
        if(r!=0)
        {
          predicted[r][i].delta_w_old[j] = predicted[r-1][i].delta_w[
j];
          predicted[r][i].delta_w[j] = (eta*predicted[r][i].lgrad_out
 * hidden[r][j].value) + (alpha*predicted[r][i].delta_w_old[j]);
```

```
        }
        if(r==0)
        {
          if(epoch!=0)
          {
            predicted[r][i].delta_w_old[j] = predicted[total_rows-1][i].delta_w[j];
            predicted[r][i].delta_w[j] = (eta*predicted[r][i].lgrad_out * hidden[r][j].value) + (alpha*predicted[r][i].delta_w_old[j]);
          }
        }
      }
    }
    for (i = 0;i < no_hneuron;i++) //a loop to calculate the delta weights for the hidden neurons
    {
      for (j = 0;j < no_input;j++)
      {
        if (r != 0)
        {
          hidden[r][i].delta_wh_old[j] = hidden[r - 1][i].delta_wh[j];
          hidden[r][i].delta_wh[j] = (eta*hidden[r][i].lgrad_hid*input[r][j]) + (alpha*hidden[r][i].delta_wh_old[j]);
        }
        if (r == 0)
        {
          if (epoch != 0)
          {
            hidden[r][i].delta_wh_old[j] = hidden[total_rows - 1][i].delta_wh[j];
            hidden[r][i].delta_wh[j] = (eta*hidden[r][i].lgrad_hid*input[r][j]) + (alpha*hidden[r][i].delta_wh_old[j]);
          }
        }
      }
    }
```

```
   for (i = 0;i < no_hneuron;i++) //Updating the weights for the hidden neurons
     for (j = 0;j < no_input;j++)
     {
       hidden[r][i].wh[j] = hidden[r][i].wh[j] + hidden[r][i].delta_wh[j];
     }
   for (i = 0;i < no_output;i++) //Updating the weights of the output neurons
     for (j = 0;j < no_hneuron;j++)
       predicted[r][i].w[j] = predicted[r][i].w[j] + predicted[r][i].delta_w[j];
 }

 train_error = 0;
 for (r = 0;r < total_rows;r++) //Calculating the total average training error
 {
   sum = 0;
   for (i = 0;i < no_output ; i++)
   {
     sum += (predicted[r][i].error)*(predicted[r][i].error);
   }
   sum = sum / double(no_output);
   sum = sqrt(sum);
   train_error += sum;
 }
 train_error = train_error / double(total_rows); //Average Training error

 //=======Storing the final epoch weights for the validation data ========//
 for (i = 0;i<no_hneuron;i++)
   for (j = 0;j<no_input;j++)
     wh_avg[i][j] = hidden[total_rows-1][i].wh[j];
 for (i = 0;i<no_output;i++)
   for (j = 0;j<no_hneuron;j++)
     w_avg[i][j] = predicted[total_rows-1][i].w[j];
```

```
//===========Validation Data ===============//
for(r=0;r<val_total_rows;r++)
{
  for (k = 0;k < no_hneuron;k++) //a loop to find the value of each hidden neuron
  {
    sum = 0;
    for (i = 0;i < no_input;i++)
    {
      sum = sum + (wh_avg[k][i] * v_input[r][i]); //net input
    }
    v_hidden[r][k].value = v_hidden[r][k].activation("sigmoid", sum); //applying activation function
  }
  for (k = 0;k < no_output;k++) //a loop to find the value of each predicted output
  {
    sum = 0;
    for (i = 0;i < no_hneuron;i++)
    {
      sum = sum + (w_avg[k][i] * v_hidden[r][i].value); //net input
    }
    v_predicted[r][k].value = v_predicted[r][k].activation("sigmoid", sum); //applying activation function
  }
  for (i = 0;i<no_output;i++) //a loop to determine the errors in output
  {
    v_predicted[r][i].error = v_output[r][i] - v_predicted[r][i].value;
  }
}
val_error = 0;
for (r = 0;r < val_total_rows;r++) //Calculating the total average epoch error
{
```

```
    sum = 0;
    for (i = 0;i < no_output; i++)
    {
      sum += (v_predicted[r][i].error)*(v_predicted[r][i].error);
    }
    sum = sum / double(no_output);
    sum = sqrt(sum);
    val_error += sum;
  }
  val_error = val_error / double(val_total_rows); //Average Validation error

  epoch += 1; //incrementing the epoch number
  cout << "Epoch Number: " << epoch << " , Training Error: " << train_error << " , Validation Error: " << val_error << endl; // Displaying the epoch errors on the screen
  errorfile_train << epoch << "," << train_error << endl; //Writing the training errors to a csv file
  errorfile_val << epoch << "," << val_error << endl; //Writing the validation errors to a csv file
} //The training and validation session ends here


//=======Storing the final epoch weights for the testing data ========//
for (i = 0;i<no_hneuron;i++)
  for (j = 0;j<no_input;j++)
  {
    wh_avg[i][j] = hidden[total_rows - 1][i].wh[j];
    h_weightfile << i << "," << j << "," << wh_avg[i][j] << endl;
  }

for (i = 0;i<no_output;i++)
  for (j = 0;j<no_hneuron;j++)
  {
    w_avg[i][j] = predicted[total_rows - 1][i].w[j];
    o_weightfile << i << "," << j << "," << w_avg[i][j] << endl;
  }
```

```

//===========Testing Data ===============//
for (r = 0;r<test_total_rows;r++)
{
  for (k = 0;k < no_hneuron;k++) //a loop to find the value of each hidden neuron
  {
    sum = 0;
    for (i = 0;i < no_input;i++)
    {
      sum = sum + (wh_avg[k][i] * test_input[r][i]); //net input
    }
    test_hidden[r][k].value = test_hidden[r][k].activation("sigmoid", sum); //applying activation function
  }
  for (k = 0;k < no_output;k++) //a loop to find the value of each predicted output
  {
    sum = 0;
    for (i = 0;i < no_hneuron;i++)
    {
      sum = sum + (w_avg[k][i] * test_hidden[r][i].value); //net input
    }
    test_predicted[r][k].value = test_predicted[r][k].activation("sigmoid", sum); //applying activation function
  }
  for (i = 0;i<no_output;i++) //a loop to determine the errors in output
  {
    test_predicted[r][i].error = test_output[r][i] - test_predicted[r][i].value;
  }
}
test_error = 0;
for (r = 0;r < test_total_rows;r++) //Calculating the total average epoch error
{
```

```
    sum = 0;
    for (i = 0;i < no_output; i++)
    {
      sum += (test_predicted[r][i].error)*(test_predicted[r][i].error);
    }
    sum = sum / double(no_output);
    sum = sqrt(sum);
    test_error += sum;
  }
  test_error = test_error / double(test_total_rows); //Average Validation error
  cout << "Average Testing Error: " << test_error << endl;
  errorfile_test << test_error << endl;
  getch();
}
```

A.2 Implementation on Robot

```
#include <Aria.h>
#include <stdio.h>
#include <iostream>
#include<conio.h>
#include<fstream>
#include<string>
#include<sstream>

using namespace std;

int main(int argc, char **argv)
{
  int i, j;
  //========Parameters for the neural network===============//
  const int no_input = 2, no_output = 2, no_hneuron = 3;
  const double eta = 0.9, lambda = 0.5, alpha = 0.6;
  //=====================================================//
  double wh[no_hneuron][no_input], w[no_output][no_hneuron]; //hidden weights and output weights
  vector<double> row; //to read each row from the file
```

```
string line, word; //to reach the each line and word
double input[no_input], v[no_hneuron], h[no_hneuron], y[no_output]; //input, output, hidden neurons and net input variables
double lms_speed, rms_speed; //final speeds

ifstream file_h("finalhiddenweights.csv"); //reading hidden weights
getline(file_h, line); //reading each line as a string
while (getline(file_h, line))
{
  row.clear();
  stringstream ss(line); //for breaking each line into words
  while (getline(ss, word, ','))
  {
    row.push_back(stod(word));
  }
  i = row[0];
  j = row[1];
  wh[i][j] = row[2]; //saving the hidden weights
}

ifstream file_o("finaloutputweights.csv"); //reading output weights
getline(file_o, line); //reading each line as a string
while (getline(file_o, line))
{
  row.clear();
  stringstream ss(line); //for breaking each line into words
  while (getline(ss, word, ','))
  {
    row.push_back(stod(word));
  }
  i = row[0];
  j = row[1];
  w[i][j] = row[2]; //saving output weights
}

//==========Initialization of robot===============//
Aria::init();
ArRobot robot;
```

```cpp
ArArgumentParser argParser(&argc, argv);
argParser.loadDefaultArguments();
ArRobotConnector robotConnector(&argParser, &robot);
if (robotConnector.connectRobot())
  cout << "Robot Connected!" << endl;
robot.runAsync(false);
robot.lock();
robot.enableMotors();
robot.unlock();
ArSensorReading *sonarSensor[8];
int sonarRange[8];
//==============================================//
while (true)
{
  //getting sonar readings
  for (i = 0; i < 8; i++) {
    sonarSensor[i] = robot.getSonarReading(i);
    sonarRange[i] = sonarSensor[i]->getRange();
  }
  //saving normalized inputs
  input[0] = double(min(sonarRange[2],sonarRange[3]))/double(5000); //left front sensor
  input[1] = double(min(sonarRange[0],sonarRange[1]))/double(5000); //left back sensor
  //cout << "Inputs: " << input[0] << " " << input[1] << endl;

  for(i=0;i<no_hneuron;i++)
  {
    v[i] = 0;
    for(j=0;j<no_input;j++)
    {
      v[i] += (wh[i][j])*input[j]; //Calculating net input
    }
    h[i]= 1 /( 1 + exp(-lambda*v[i])); //Applying activation function for the hidden neurons
  }
  for (i = 0;i<no_output;i++)
  {
```

```
      v[i] = 0;
      for (j = 0;j<no_hneuron;j++)
      {
        v[i] += (w[i][j])*h[j]; //Calculating net input value
      }
      y[i] = 1 /( 1 + exp(-lambda*v[i])); //Applying activation
   function for output neuron
    }
    //De-normalizing the outputs
    lms_speed = (y[0])*250;
    rms_speed = (y[1])*250;
    robot.setVel2(lms_speed, rms_speed); //setting robot speeds as per
   the output obtained

    cout << " Output speeds: " << rms_speed << " " << lms_speed << endl
   ;

  }

  // Stopping the robot
  robot.lock();
  robot.stop();
  robot.unlock();
  // terminate all threads and exit
  Aria::exit();
  return 0;
}
```

B. DEEP LEARNING CODE

B.1 Data Cleaning

```
import pandas as pd
import numpy as np

# #### Reading from store.csv file
store=pd.read_csv('store.csv')
store.head()

# #### checking for null values
store.isnull().sum()

# #### Finding the mode values in the columns having null in them

store['CompetitionDistance'].mode()

store['CompetitionOpenSinceMonth'].mode()

store['CompetitionOpenSinceYear'].mode()

store['Promo2SinceWeek'].mode()

store['Promo2SinceYear'].mode()

# #### Replacing the null values with the mode values

cols2 = ["CompetitionDistance"]
for col in cols2:
```

```
    store[col].fillna(9, inplace=True)
cols3 = ["CompetitionOpenSinceMonth"]
for col in cols3:
    store[col].fillna(9, inplace=True)
cols4 = ["CompetitionOpenSinceYear"]
for col in cols4:
    store[col].fillna(2013, inplace=True)
cols5 = ["Promo2SinceWeek"]
for col in cols5:
    store[col].fillna(14, inplace=True)
cols6 = ["Promo2SinceYear"]
for col in cols6:
    store[col].fillna(2011, inplace=True)
cols7 = ["PromoInterval"]
for col in cols7:
    store[col].fillna("null", inplace=True)
store.isnull().sum()

store.dtypes

# #### Changing all datatypes of columns to either int or float

store['StoreType'] = pd.to_numeric(store['StoreType'],errors='coerce').
    fillna(0).astype(np.int64)
store['Assortment'] = pd.to_numeric(store['Assortment'],errors='coerce'
    ).fillna(0).astype(np.int64)
del store['PromoInterval']
store.head()

store.to_csv("cleaned_store.csv")

# #### Reading the trainn.csv file for cleaning

train=pd.read_csv('train.csv')
train.head()

# #### Finding the columns with the null values

```

```
train.isnull().sum()

train.dtypes

train=train.fillna(0)

# #### Processing the date values with the '-'

train['Date'] = train['Date'].map(lambda x: ''.join(x.split('-')))
train.head()

# #### Changing columns with type object to numeric

train['Date'] = pd.to_numeric(train['Date'],errors='coerce')
train['StateHoliday'] = pd.to_numeric(train['StateHoliday'],errors='
    coerce').fillna(0).astype(np.int64)
train.dtypes

train.isnull().sum()
train.to_csv("cleaned-train.csv")

df=pd.merge(store, train, on="Store")
df.head()

cols=df.columns.tolist()
temp=cols[11]
cols[11]=cols[16]
cols[16]=temp
cols
df=df[cols]
df.head()

df.to_csv("merged_training.csv")

# #### Cleaning the test data

test=pd.read_csv('test.csv')
test.head()
```

```python

test['Date'] = test['Date'].map(lambda x: ''.join(x.split('-')))
test['Date'] = pd.to_numeric(test['Date'],errors='coerce')
test['StateHoliday'] = pd.to_numeric(test['StateHoliday'],errors='coerce').fillna(0).astype(np.int64)
test.head()

test.dtypes

test.to_csv('cleaned_test.csv')
```

B.2 Deep learning architecture

```python

# ### Trying to implement a neural network model with tensorflow
import  tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
import pandas as pd

# #### Reading data from the cleaned dataset for train
# #### And identifying the columns which will be treated as inputs and outputs

df=pd.read_csv("cleaned-train.csv")
x=df.iloc[:,0:8]
y=df.iloc[:,8:]

model=Sequential()  #Initializing the model
model.add(Dense(100,input_shape=(8,))) #Adding a hidden layer with 100 neurons taking 8 inputs
model.add(Activation("relu")) #Applying the rectifier activation function
model.add(Dense(100))
model.add(Activation("relu"))
model.add(Dense(100))
model.add(Activation("relu"))
```

```
model.add(Dense(100))
model.add(Activation("relu"))
model.add(Dense(100))
model.add(Activation("relu"))
model.add(Dense(100))
model.add(Activation("relu"))
model.add(Dense(100)) #Adding another hidden layer with 100 neurons
model.add(Activation("relu"))
model.add(Dense(50))
model.add(Activation("relu"))
model.add(Dense(100))
model.add(Activation("relu"))
model.add(Dense(50))
model.add(Activation("relu"))
model.add(Dense(1))
model.add(Activation("linear"))


# #### Defining the optimization parameters and the metrics to display

model.compile(optimizer='rmsprop',loss='mse', metrics=['accuracy','mse'
    ])

# #### Training the model to fit the data

model.fit(x,y,epochs=250,batch_size=50)

df1=pd.read_csv("cleaned_test.csv")
x1=df1.iloc[:,0:8]

# #### Predicting the sales value for the cleaned test dataset

result=df1
result['Sales']=model.predict(x1)
result.to_csv('prediction.csv')
result.head()
```