# Introduction

This report provides details and background for the artificial neural network (ANN) algorithms presented for CE889 assignments. The given individual task was for an ANN to learn left wall following behaviour from data collected from a robot running a left wall following algorithm. The group task was to have an ANN predict the store sales for a retail outlet for a given date using the data provided about the store.

In this report, we will look at a brief history of ANNs outlining the major developments in terms of ANN architectures and situations where such architectures have been used. Following this, we will look at the architecture used by the author for this assignment. We will then look at the various parameters that were selected to run the robot for the practical demonstration of ANN learning. We will review the reasons why the parameters were chosen for the task. The report then looks at parameters chosen for the group task and review the reasons behind the choice for these parameters.

In the next section we will look at the results achieved in both the tasks and review the performance of the ANNs used. Following this, we will look at potential areas of improvements for both tasks and possible alternatives that could have been used in terms of architecture.

# Review of Literature and Major Developments

# ANN Architecture Used for Assignments

## Individual Task: Left Wall Following Behaviour for a Robot

For this task, the ANN had to predict values for left motor speed (LMS) and right motor speed (RMS). The inputs consisted of values for left front sensor (LFS) and left back sensor (LBS). There were approximately 700 training instances provided for the task. In the demonstration, the robot read values for LFS and LBS from its sensors and the ANN provided the robot with LMS and RMS which were used to propel the robot.

From the description we can see that the ANN had to be trained for a regression task. Moreover, since we know the desired output for each training sample, this is a supervised learning problem. Since the number of inputs and outputs were both small, a multi-layer perceptron with error back-propagation (MLP – BP) was considered a reasonable choice for the ANN architecture. For the same reason, one hidden layer was considered sufficient for this task.

### Implementation of MLP – BP

The algorithm was implemented in C++. The implementation consisted of a single class ‘Net1’. The key components and processes of the algorithm as implemented are as follows:

* Data normalisation: As an initial step, all input and output data was normalised in the range [0, 1]. This was done so that no input dominates the learning process and learning takes place faster **[Reference needed].**
* Input layer and input nodes: As mentioned earlier, we have two inputs for each training instance, hence we have two nodes in the input layer. We add one input node with a value of 1 as a bias node. This is to ensure that we can access solutions which do not always pass through the origin. Input nodes were represented by a variable ‘**in\_vec**’ of the type Vector<double>.
* Hidden layer and hidden nodes: We have one hidden layer in our implementation. The number of hidden nodes at the design stage was not fixed so that we could experiment with varying values of hidden nodes for our task. In addition to hidden nodes whose values were calculated from input nodes, we have an extra bias node in the hidden layer with the default value of 1. Hidden nodes were represented by a variable ‘**hid\_vec**’ of the type Vector<double>. This representation was useful as it did not need us to know the number of hidden nodes in advance.
* Output layer and output nodes: We have two outputs for each training instance. Therefore we have two nodes in the output layer. These were represented by a variable ‘**out\_vec**’ of the type Vector<double>.
* Weights from input to hidden layer: There is a weight value connecting each input node to each hidden node. The weight values were initialised to random values in the range [-1, 1]. The weights were represented by a variable ‘**weights\_in\_h**’ of the type Vector<Vector<double≫.
* Weights from hidden to output layer: There is a weight value connecting each hidden node to each output node. The weight values were initialised to random values in the range [-1, 1]. The weights were represented by a variable ‘**weights\_h\_out**’ of the type Vector<Vector<double≫.
* Eta: used to determine the learning rate for the ANN, values in the range [0, 1]. Eta can be initialised at the time of creating a Net1 object.
* Alpha: used to determine the momentum term for the ANN, values in the range [0, 1]. Alpha can be initialised at the time of creating a Net1 object.
* Activation function: In our implementation a sigmoid activation function was used with the formula

**Φ(ν) = 1/(1 + e(-λ.ν) )**

Where, φ(ν) represents the activated value of ν and λ is a constant.

* Lambda: used as a parameter for sigmoid activation function, values in the range [0, 1]. Lambda can be initialised at the time of creating a Net1 object.
* Feed Forward Step: Now that we have all the essential ingredients for our ANN implementation in place, we can look at how each training sample is processed.
  + At first, all the raw values for each hidden node are calculated. The raw value for each hidden node is the sum of input nodes multiplied by their respective weights connecting the input node to the hidden node.
  + The raw values are then converted to activated values using the sigmoid activation function described earlier. The bias hidden node is not connected to any input node and its value is always 1.
  + The raw values for each output node are calculated in a similar manner, summing up the product of activated value for each hidden node along with the weight connecting to a particular output node.
  + The raw values for each output node is then converted to an activated value using the sigmoid activation function.
  + Finally an error value for each output node is calculated by taking the difference of actual output values for that particular training instance and the activated output values.
  + ***Practical implementation:*** These steps were implemented using a function ‘get\_v’ for getting raw value and then using a function ‘activated\_h’ to get the activated value for each node. Detailed code for all functions is in Appendix A. The result of each step is a square error term, which is the sum of square of errors of both output nodes. These error terms are added cumulatively and are used to calculate mean squared error terms at the end of each epoch. These functions were used in another function – feed\_forward() which was responsible for implementing the feed forward behaviour.
* Error Back-Propagation Step: Once we have the errors for each output node, we then back-propagate these errors to update the weight values, this is the crux of the ANN algorithm, as updated weights reflect the learning from the errors in the output. The steps involved in back-propagation are as follows:
  + For each output node, a local gradient term is calculated by using the following formulae:

δ(o) = error(o) x activated\_value(o) x [1 – activated\_value(t)), where δ(o) represents local gradient for the node o.

* + For each output node, a value Δ is calculated which represents the amount by which that particular weight needs to be changed.
  + Δ for an output node is calculated by summing over a product of eta (learning rate as discussed earlier), local gradient for the node (δ) and the activated value of each hidden node connected to the output node.
  + Once Δ has been calculated, it is adjusted by adding to it a product of previous Δ for that node and the momentum term (alpha). This is done to ensure that the weight update retains learning from the previous training sample. For the first sample, value of previous Δ is zero and hence no update takes place.
  + New weights for a node are calculated by adding the previous weight and Δ value calculated in the previous step.
  + Next step is to calculate the updated weights for the hidden nodes. For each hidden node a local gradient term is calculated by using the following steps:
    - First a sum value, denoted by sum(h) is calculated as sum of product of original weights and local gradient of the connected output node.
    - Then, δ(h) = sum(h) x activated\_value(h) x [1 – activated\_value(h)), where δ(o) represents local gradient for the node h, sum(h) is the value calculated in the previous step and activated\_value(h) is the activated value for the hidden node during the feed forward step.
  + Finally, a Δ value is calculated for each hidden node in exactly the same way as the Δ value for output node, except that the values are calculated using connections from input nodes and the corresponding weights.
  + Each weight is now updated as previously by adding Δ to the previous weight. This completes the back-propagation step for a training sample.
  + **Practical implementation**: These steps were implemented using four different functions. update\_local\_grad\_output() and update\_weights\_h\_out() carried out the calculations and weight updates for the output nodes, while the corresponding function for hidden nodes was performed by update\_local\_grad\_hidden() and update\_weights\_in\_h(). Details can be found in Appendix A.
* Loop through all samples: The feed forward and back-propagation step is repeated for each training sample. Once all the training samples are completed, this marks the end of one epoch. At the beginning of each epoch, the total data available was split into training and test sets. Training set was allocated 500 samples and test set was allocated 200 samples. At first, it was done after randomly shuffling the whole data set. However, random shuffling produced erratic MSEs for training and test set. Hence, a choice was made to preserve the order of the data and training was carried out on first 500 training samples with the remaining 200 samples acting as test data. This choice resulted in a much smoother curves for training and test set MSEs. At the end of each epoch, MSEs for training and test sets are calculated and written to a text file. This functionality was implementd through a function loop\_through(). Multiple epochs were run using a function run\_epochs(), which was also responsible for saving network weights and writing MSE data to text files on disk.
* We end training when the number of epochs is greater than 100 and the MSE for the test set is greater than MSE for the previous test set.

## Group Task: Prediction of Retail Sales for Rossmann Stores

The group task involved training a deep neural network (DNN) on data provided for a retail firm which has 1115 stores. Data was presented as store specific (type of store, distance from nearest competition etc.) and for each day (number of customers, sales, promotions running etc.). Since the task was to predict sales figures, it was a regression task for the algorithm.

The challenge was for the DNN to predict sales for unseen data. Unlike the ANN, DNN was not coded from scratch. It was allowed for DNN to make use of one the existing implementations.

### Implementation of Deep Learning Assignment

The DNN implementation followed the following steps:

* As an initial task, the group decided to create a shared repository on GitHub, so that the group members could keep reporting/sharing their progress on a central platform.
* Our first step in creating the DNN was to understand the data and convert it into a usable form for the DNN implementation. There were three main issues with data – using non-numeric data, missing data and extracting meaningful information from fields. These were handled as described below:
  + There were some data fields which were in a non-numeric format. For example ‘store type’ was represented by a text value. We converted all non-numeric fields to numeric values, using integers to represent categorical variables. In case of dates, it was considered that the important piece of information in the context of predicting sales for a particular date was the number of days between a given date value and the current date. Hence all dates were converted to integer values representing difference in days.
  + In case of missing data, there were two fields which had missing information for some stores. First missing information was about promotion details. In this case it was decided to use a 0 value to indicate if promotion was not running. In second case, information was missing about competition stores opening dates. Since we had decided that we were going to use the difference in terms of number of days as the relevant information in case of dates. Such fields were replaced by a zero value as well.
  + Some fields, like the date of sales were not used directly as the date unique for each date/store combination and was not going to repeat in future. Therefore we decided to use the month field from the date to capture the seasonality from the sales data, if there was any.
* Following the data manipulation, we explored a few options from pre-existing implementation libraries for DNN. Amongst these, we focused on Deep Learning for Java (DL4J) library and Matlab’s implementations. After spending some time exploring DL4J, it became clear that the implementation was more suitable for classifier type problems and that implementation was not well developed yet to support regression type problems to the same extent. Therefore we decided to train our DNN with Matlab, even though Matlab does not offer as customisable a platform as DL4J, it was considered satisfactory for our requirements.
* Given the nature of the problem, the group decided on a feed-forward MLP with Error Back-Propagation Network as the network architecture.
* We started with training a network of 1 hidden layer with 40 nodes to get a frame of reference for accuracy and time spent in training.
* We trained several networks with different number of hidden layers. We experimented with 2, 3 and 4 hidden layers for this task. However, given that we had 18 input variables, it was deemed unnecessary by the group member to train networks with more hidden layers.
* Given our final accuracy rates and considering the time spent in training each network, we decided on the performance of a network with 3 hidden layers and 15 nodes in each hidden layer as the best performing DNN. A justification and discussion of parameters and results is presented in the next section.

# Choice of Parameters

## Individual Task: Left Wall Following Behaviour for a Robot

For this task, the main parameters to be decided upon were

* Number of hidden nodes: The starting point for ANN training was taken as twice the number of input nodes. Since the number of input nodes was 2, we started with 4 nodes. It was observed that the network learned faster if the number of nodes was increased. However, the number of weights to be trained also increased with an increase in the number of nodes. For example, with 2 hidden nodes, the number of weights to be trained would be (3 x 2) + (3 x 2) = 12, whereas, with 4 hidden nodes, the number of weights to be trained was (3 x 4) + (5 x 2) = 22. It was decided that we needed a minimum of 5 training samples for each weight **[REFERENCE NEEDED]**. Since we had 500 samples for training, it limited us to having a maximum of 100 weights in the network, which gave us an upper bound of roughly 15 hidden nodes (3 x 15) + (16 x 3) = 93 weights. In the end, we observed a good balance between speed of convergence and time taken to train with 10 hidden nodes, which was chosen as the final choice for the network.
* Rate of Learning (Eta): Eta represents the speed at which network traverses through the search space **[Reference???]**. A very high Eta value is likely to miss the optimal solution, whereas a very small Eta value is likely to take a long time to reach convergence. Therefore, we need a balance between the two extremes. Let us compare the error curves for two values of Eta – 0.2 vs 0.01, while holding Lambda and Alpha constant at 0.5 and 0.9 respectively. It can be seen that at Eta value of 0.2, the test error rate is completely erratic and it would be very difficult for us to find a natural stopping point for algorithm. Whereas for Eta = 0.01, we have a smooth curve for test and train error, and we can see that test error started rising after roughly 250 iterations, which is where the training was stopped.

Since, we were prioritizing better learning over speed, we decided to use an Eta value of 0.01 in our final algorithm.

Figure

Figure

* Momentum rate (Alpha): Momentum rate represents the weightage that we are giving to the weights learned from the prior examples **[REFERENCE].** It was noticed in training that lower alpha values tended to take a longer time to train. For example, for Alpha = 0.5, our network could not converge in 1000 iterations, whereas for Alpha = 0.75, the network took roughly 350 iterations to train as shown in the figure below. In our example, we wanted to preserve learning from previous training to a high degree, hence we have kept the Alpha as 0.9.

Figure

* Lambda: Lambda represents the steepness of the sigmoid activation curve **(Reference)**. From a practical perspective, we notice that higher values of Lambda tend to give an uneven start to the test error (i.e. the test error rises in the first few iterations), hence we have kept the value of Lambda to be low at 0.1 which eliminates the problem of an uneven start. Overall, the results are not too sensitive to values of Lambda, so a higher value can also be chosen as long as the first few iterations are ignored. The following two figures illustrate this point. For low values of Lambda, a further advantage that was noticed was that the test error reduces at a gradual pace throughout the training while the rate of reduction can be erratic for higher values of Lambda. Lower rates of Lambda tend to take longer time to train. However, in this case it did not pose too much of a problem as the training still finished in roughly 500 iterations for low values of Lambda (0.1).

Figure

Figure