# 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.
* 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. 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.
  + 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.