# 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. 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 was 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 was 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]
* Alpha: used to determine the momentum term for the ANN, values in the range [0, 1]
* 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]