Assignment 2, Vishwajeet Karmarkar

1. The motion of robots in the data-set can be modelled using a single wheel bicycle model. The model used is the same as the one in Assignment 0, hence details are skipped for brevity. On comparing the output from the model with the actual state of the robot from groundtruth data, it can be concluded that it is a poor representation of the behaviour of the robot. The positional error can be seen in the figures below.

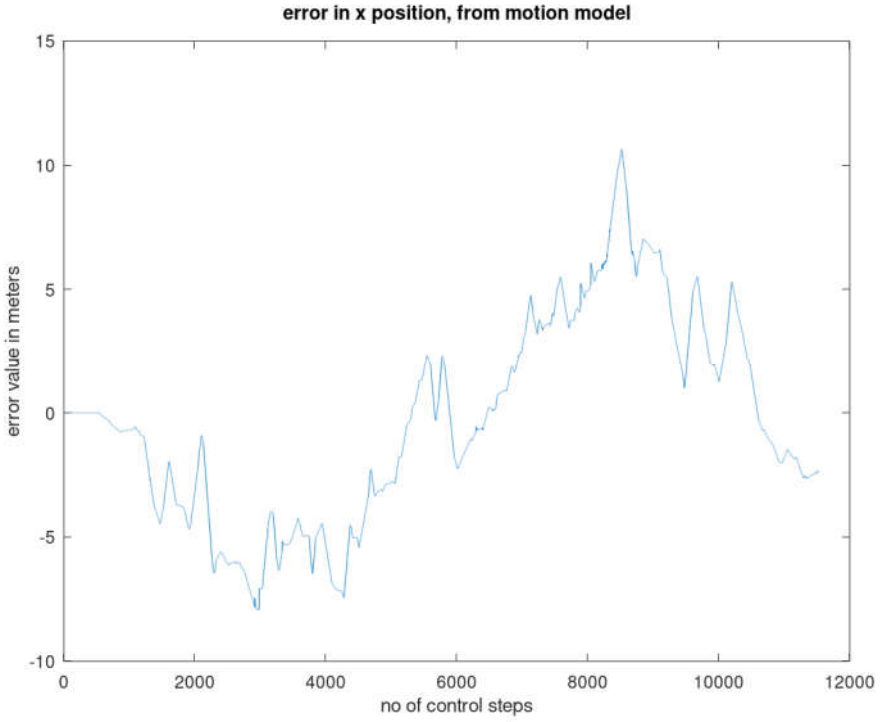


**Figure 1: error in x position**

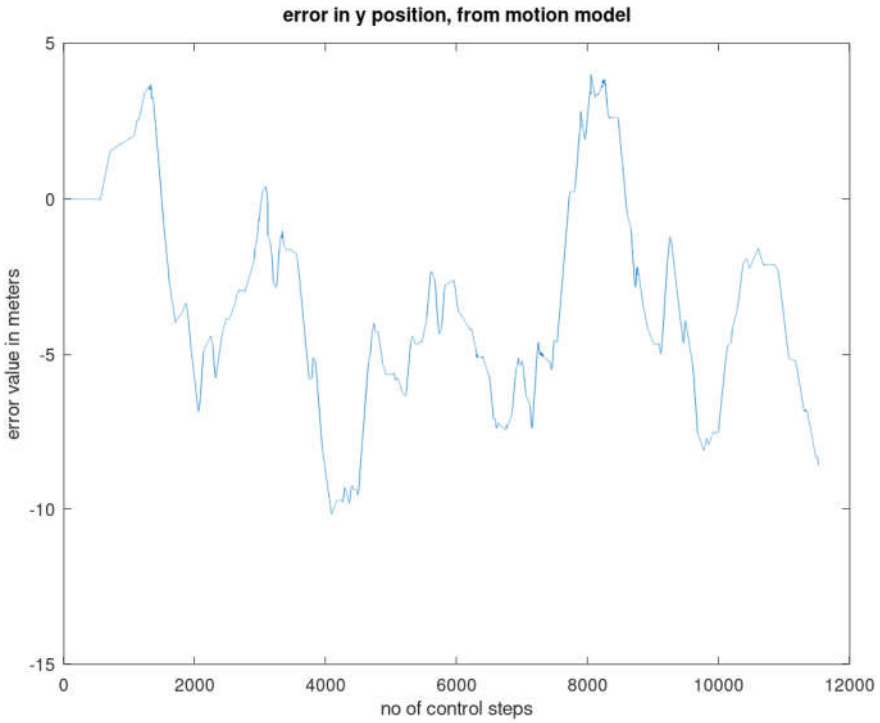


**Figure 2: error in y position**

Multiple reasons can be thought of for the poor performance of the model. Aspects like wheel alignment, friction, dimensional variances (for eg different diameter wheels), wind-draft/air resistance and many more have an impact on this variation between expected and the real output.

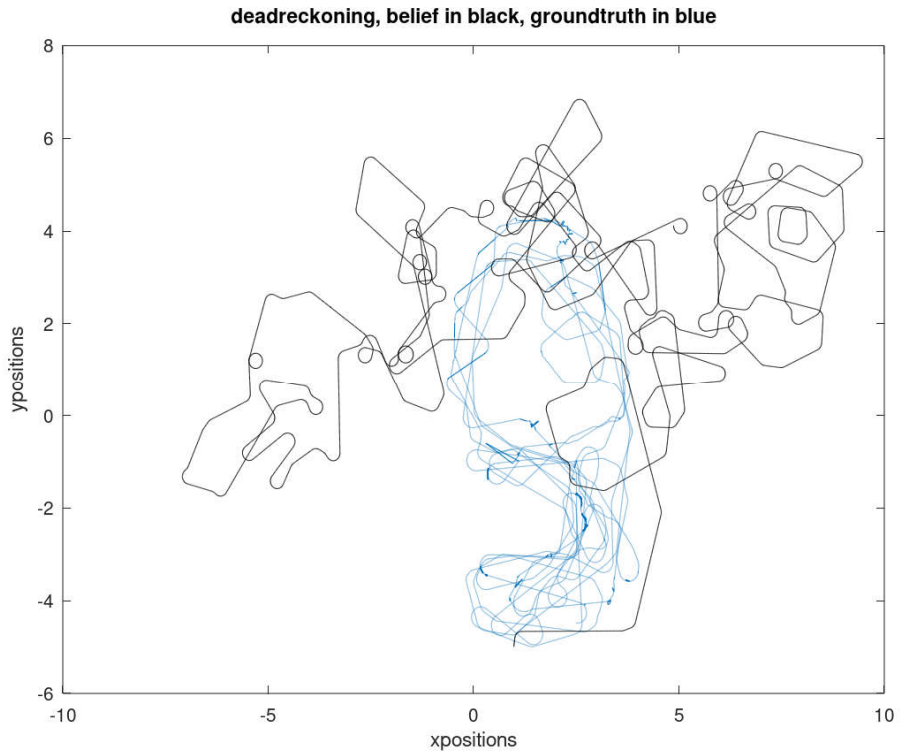


Figure 3: dead-reckoning

This leads to the formulation of an interesting question: "Can past data from the robot's motion be used in a machine-learning setting to learn the robot's state transition, without us explicitly modelling the physics of the world"? With this in mind the same data that is fed to the motion model, is also fed to the M.L algorithm, which will in turn generate the 'ML model'.

$$x_{out}, y_{out}, \theta_{out} = f(v, \omega, x, y, \theta)$$

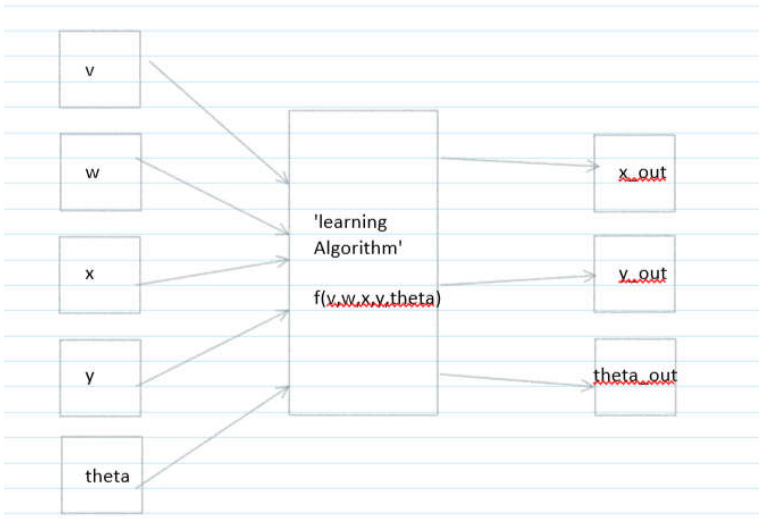


Figure 4

It is expected that the same variables that were fed to the motion model will be sufficient for the M.L model. Note on data generation: The cost function used in the learning aim is the 'sum of errors squared'. In the choice of variables there is a mismatch of representation as the heading angle is in a polar co-ordinate system, while the velocity and position are in Cartesian co-ordinates. Directly taking squared error of polar co-ordinates will give wrong costs due to the periodicity of polar-coordinates. Thus heading is expressed as the sin and cosine of heading in the algorithm's input and output. This helps maintain the same cost function across all the data dimensions. The output sines and cosines can be combined to get back a heading angle. Thus the M.L function takes the form:

$$f(v, \omega, x, y, \sin(\theta), \cos(\theta))$$

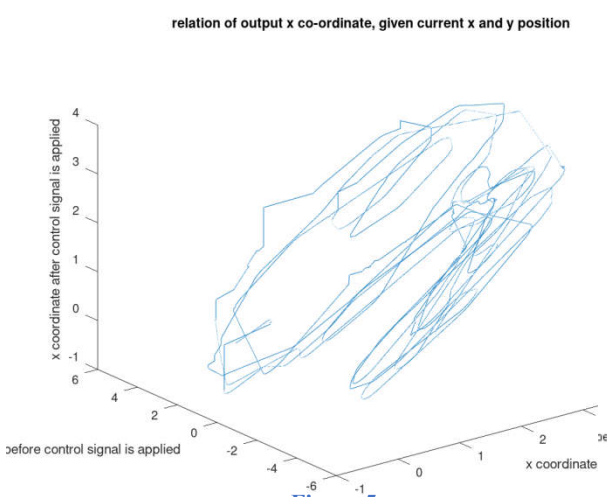


Figure 5

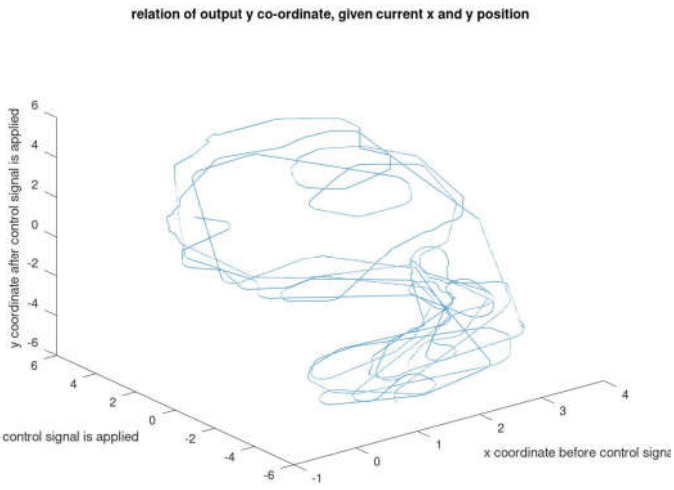


Figure 6

2. As can be seen from the images 5 and 6, there is a non-linear relationship between the output and the variables it is dependent on. The images only show the dependence on x and y position due to plotting limits, however when the other variables (v,w,theta) are added the relationship would be complicated. Neural Networks are thus chosen to be the algorithm of choice as they are a powerful tool to learn non-linear systems and are very flexible for a 'multi-input', 'multi-output' type system. Confidence in the performance is also gained from Pomerlau's ALVINN system[1], which learnt to drive a vehicle using neural networks. The ALVINN system took road images as input, and steering direction as output. The training phase involved a human providing steering data, against the road images. A neural network approach should thus be sufficient to solve the problem at hand.

[2]Neural Networks are said to be inspired from the working of the brain, in which neurons receive electrical impulses from other neurons or 'sensors' in the body, perform some 'computations' and send forward an output electrical impulse.

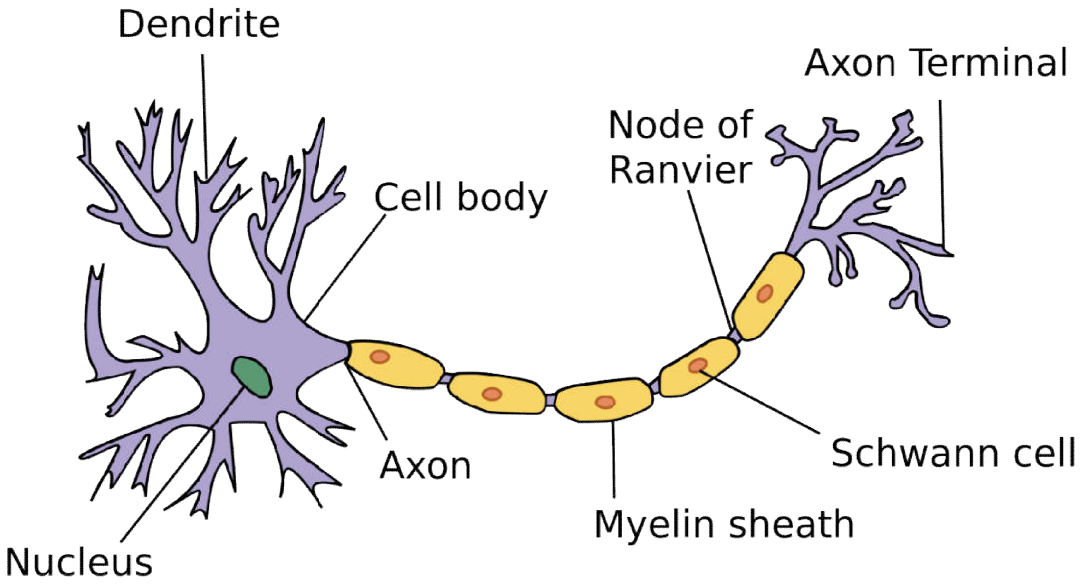


**Figure 7: Structure of a neuron. Input is taken in from the 'dendrite' and the output is given from the 'axon' [4]**

Carrying forward this idea, a neural network is made of nodes which simulate neurons. Each node can have multiple inputs feeding into it, and can feed-forward into other nodes. A representation of a sample neural network is shown above. The image below shows a network with 4 inputs, 1 output and with 1 hidden layer containing 15 nodes. An output from one layer before being fed to a simultaneous layer is multiplied by 'weights'. The weight signifies the importance of the output of that particular node on the network in front of it.

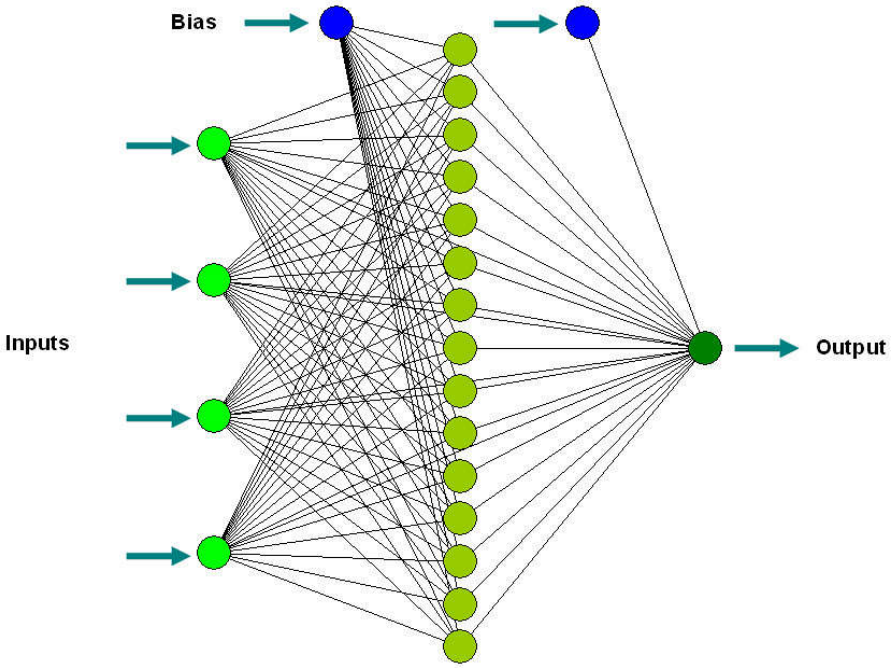


Figure 8

Each node in the hidden layers performs a computation on the input data to give outputs. This computation is referred to as an 'activation function'. Activation function for a simple 'perceptron' network is represented by a simple threshold function. [2]

$$o(x_1, x_2, \ldots . x_n) = \begin{cases} 1, if\ \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots \theta_n x_n > 0 \\ -1, \quad otherwise \end{cases} \quad (\theta\ are\ the\ weights)$$

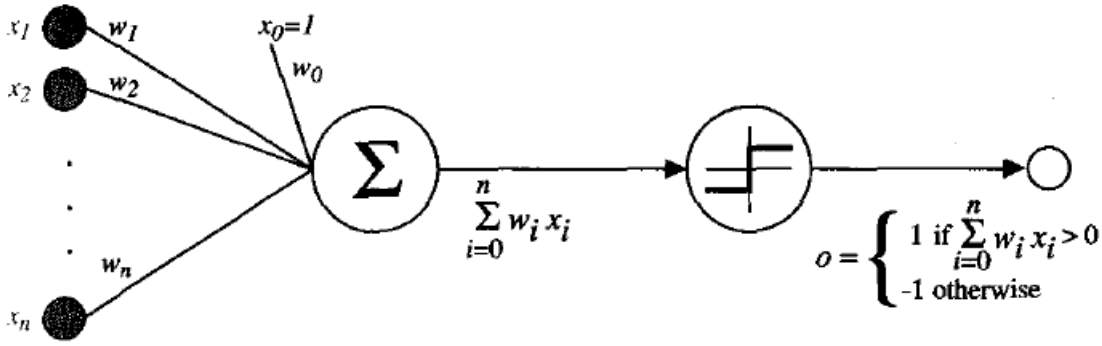


Figure 9

However this type of network can only classify linear examples. For non-linear classification or regression problems a non-linear activation function is required. Common types of activation functions include 'sigmoid', 'tanh' , 'Relu'. These functions also are differentiable, with easy formulations of the derivative.

$$sigmoid\ activation = \frac{1}{1 + e^{-x}}$$

$$\tanh activation = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$relu\ activation = \log(1 + e^{x})$$

Backpropogation algorithm:

This algorithm is used to train the network so as to find weights that can represent the data. The central idea behind back-propogation is to use partial derivatives of the cost with respect to each weight, to reduce the error in each node. An ideal, error-free node will give exactly the amount of 'activation' so as to predict the output correctly.

Mathematical representation:

The activation function used is ‘sigmoid’. The cost function used is the squared error function. For a 2 layered neural network as shown in image 10.

$$Y(x) = Sigmoid\ function. \frac{dy}{dx} = Y(x) \times \big(1 - Y(x)\big)$$

$$J(\theta) = \frac{1}{2}(y - a_4^2)$$

$$\frac{\partial J}{\partial \theta_3} = \frac{\partial J}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial \theta_3}$$

$$let\ \delta_4 = \frac{\partial J}{\partial a_4} \times \frac{\partial a_4}{\partial z_4}$$

$$then\ \frac{\partial J}{\partial \theta_3} = \delta_4 \times \frac{\partial z_4}{\partial \theta_3}$$

Similarly for layer 3

$$\frac{\partial J}{\partial \theta_3} = \frac{\partial J}{\partial a_4} \times \frac{\partial a^4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial \theta_2}$$

$$let\ \delta_3 = \frac{\partial J}{\partial a_4} \times \frac{\partial a^4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3}$$

$$then\ \frac{\partial J}{\partial \theta_3} = \delta_3 \times \frac{\partial z_3}{\partial \theta_2}$$

Similarly

$$\frac{\partial J}{\partial \theta_2} = \delta_2 \times \frac{\partial z_2}{\partial \theta_1}$$

Where

$$\delta_2 = \frac{\partial J}{\partial a_4} \times \frac{\partial a^4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2}$$

For our J and activation values

$$\delta_4 = (a_4 - y)$$

$$\delta_3 = \delta_4 \times \theta_3 \times Y(z_3) \times (1 - Y(z_3))$$

$$\delta_2 = \delta_3 \times \theta_2 \times Y(z_2) \times (1 - Y(z_2))$$

Forward pass:

$$z_1^2 = \theta_{10}^1 \times 1 + \theta_{11}^1 \times a_1^1 + \theta_{12}^1 \times a_2^1 + \theta_{13}^1 \times a_3^1$$

$$a_1^2 = sigmoid(z_1^2)$$

$$z_1^2 = \theta_{20}^1 \times 1 + \theta_{21}^1 \times a_1^1 + \theta_{22}^1 \times a_2^1 + \theta_{23}^1 \times a_3^1$$

$$a_2^2 = sigmoid(z_2^2)$$

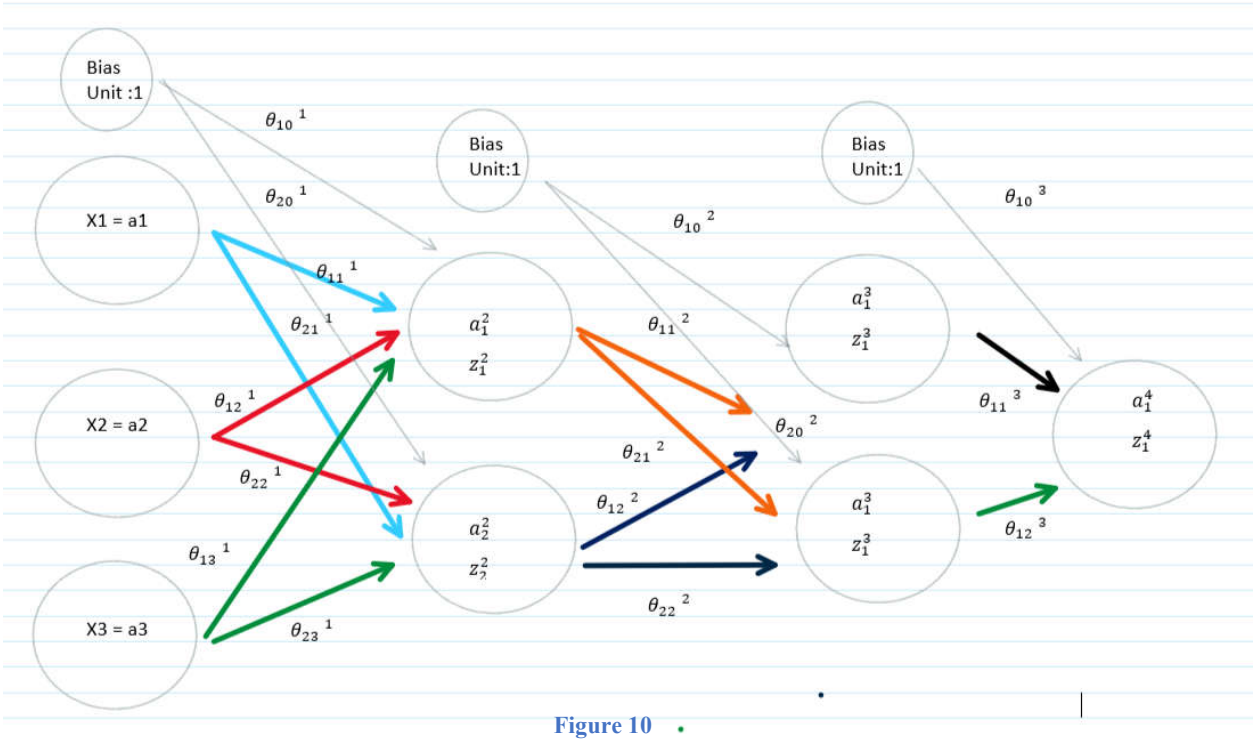


Figure 10

A forward pass as shown above is performed for 1 training example, then $\delta$ and gradient values are calculated. This is repeated for the entire training example set. Stochastic gradient descent can then used to update the weights of each node, wherein the values of gradient are averaged over all training examples, and updated over each training example cycle.

To speed up convergence, momentum rule is used instead of pure gradient descent. Momentum rule is used to add a weight to the previous value of partial derivatives, which helps the system get out of local minima's and speeds up the optimization.

$$v_t = \mu * v_{t-1} + \alpha * \frac{\partial J}{\partial \theta}$$

$$\theta_t = \theta_{t-1} - v_t - updated\ weight\ value$$

The main parameters in the neural network is the learning rate alpha, momentum constants, number of hidden layers and the nodes per hidden layer.

1) To determine number of hidden layers, a single layer trial run was performed and it was found to be giving good results. Thus a shallow network is chosen.
2) Increasing the number of units in the hidden layer, helps performance of the algorithm, as it gives the system the ability to form more inter-relations between the inputs, thus giving a scope to learn data with complicated relations. Increasing nodes however leads to an increase in run-time and so nodes were only increased till the performance improved. Performance metrics for 5000 iterations with same learning rate and momentum values are shown below.

| Number of nodes | MSE on x-co-ordinate, test data | MSE on y-co-ordinate, test data | R^2 on x-co-ordinate, test data | R^2 on x-co-ordinate, test data |
|---|---|---|---|---|
| 10 nodes | 0.0034395 | 0.0083828 | 0.99745 | 0.99880 |
| 30 nodes | 0.0015219 | 0.0026156 | 0.99887 | 0.99887 |
| 60 nodes | 0.0026006 | 0.0030743 | 0.99837 | 0.99947 |

It can be seen that there is a decent difference between the performance for 10 nodes and 30 nodes. Plots are skipped for brevity, but the difference is also visible from looking at the plots (similar plots to ones demonstrated later in the document). It also appears that the 30 node iteration performed better than 60 on some variables. This can be accounted to the variation in the initial weights, which are

randomized. In conclusion there seems to be no gain from a 30 node to a 60 node system and hence 30 nodes are selected.

3) The learning rate alpha determines how fast the system converges. A high alpha leads to overshooting (no convergence), while a very low alpha value leads to extremely slow learning. Various alpha values were tried and the corresponding Cost vs Iterations plots helped get an estimate of the learning rate (0.05). Similar methodology was used to tune the momentum parameters.

A sample dataset was generated to test the neural nets behaviour, with results shown in the image below. The dataset was generated using the following non-linear function.

$$y = x^2 + e^x + x + 0.1 \times random\_number$$

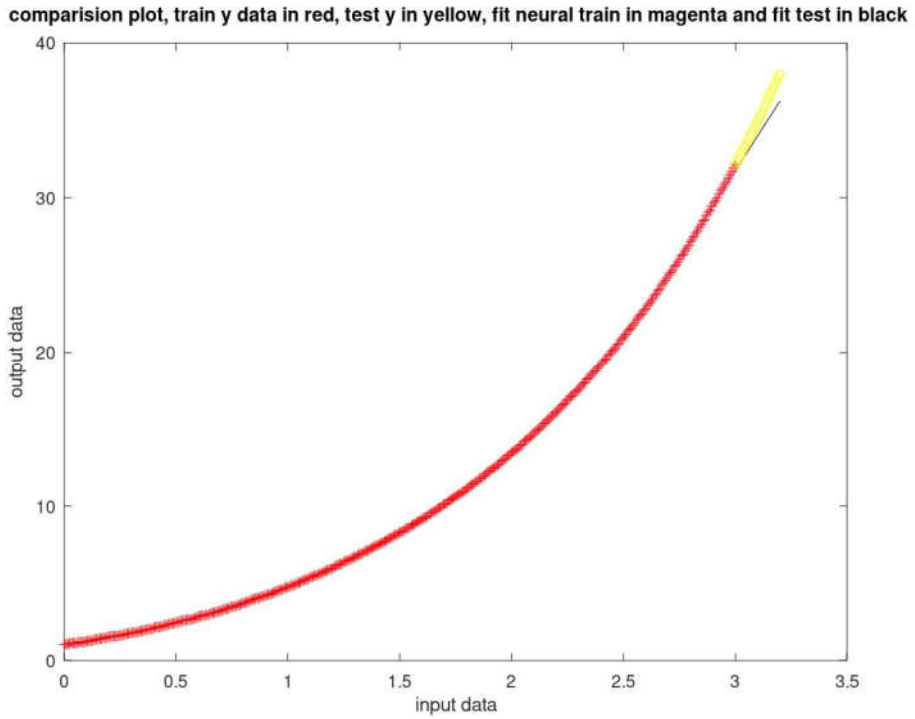


Figure 11

It can be seen that from the image that the expected output in the training part fits the data-points extremely well. The performance reduces on the test set, but it still seems sufficient to give a good estimate of the function.

3.

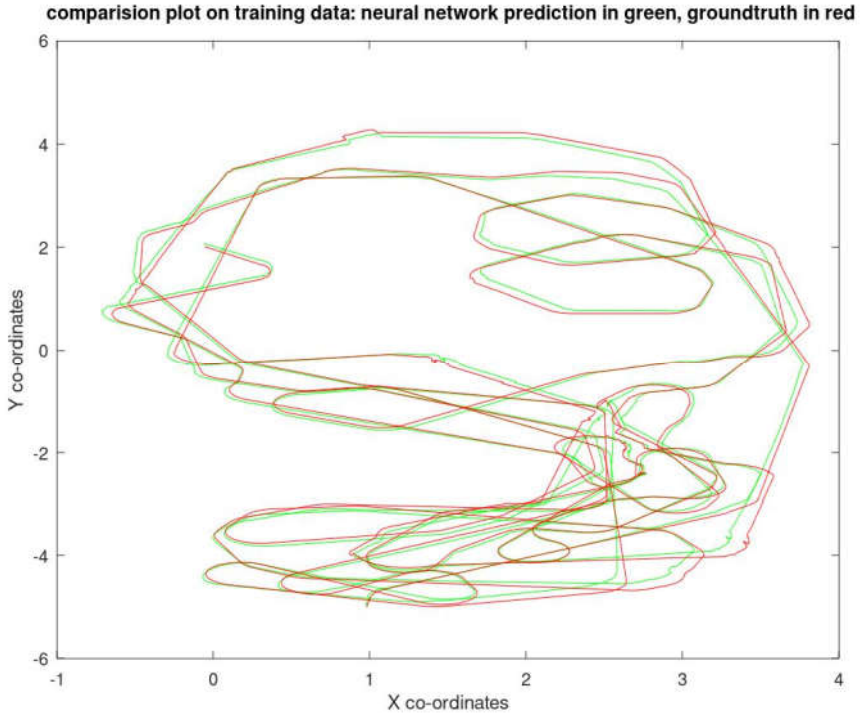


Figure 12

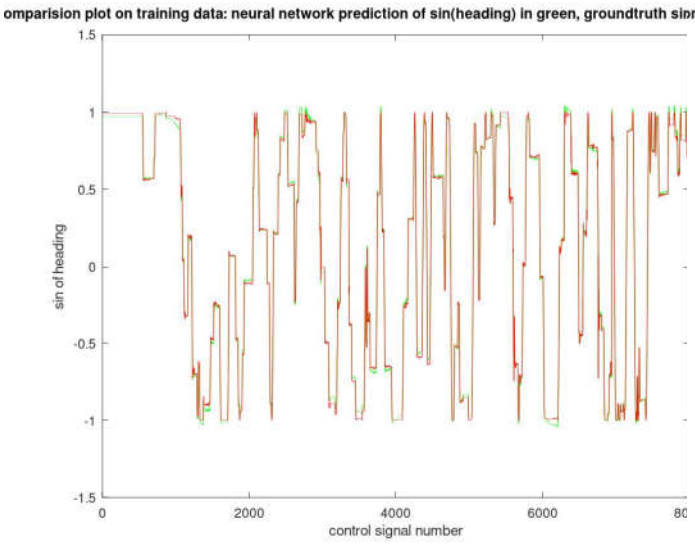


Figure 13

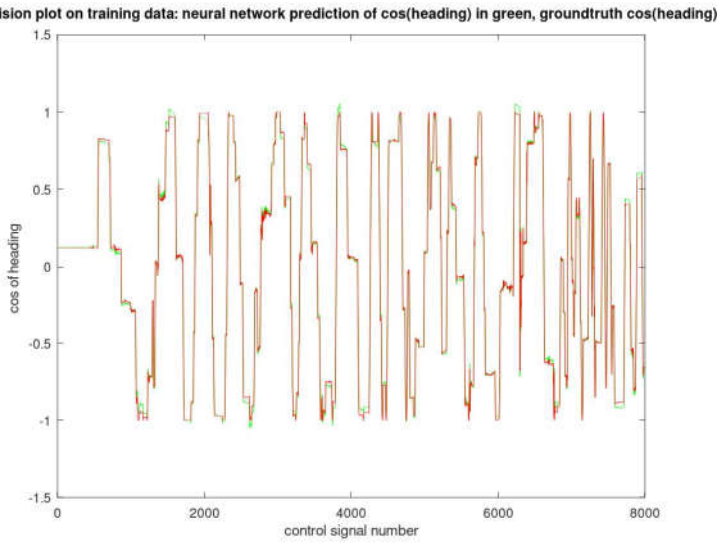


Figure 14

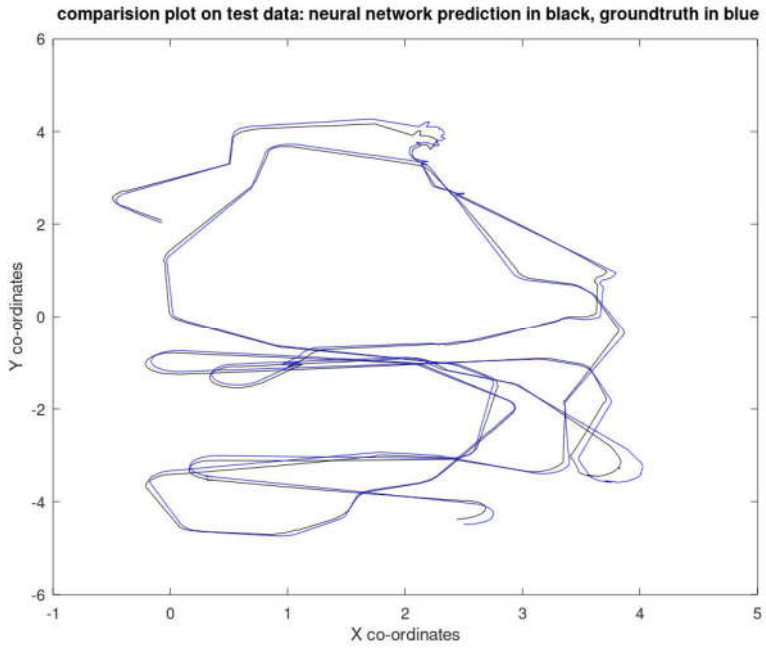


Figure 15

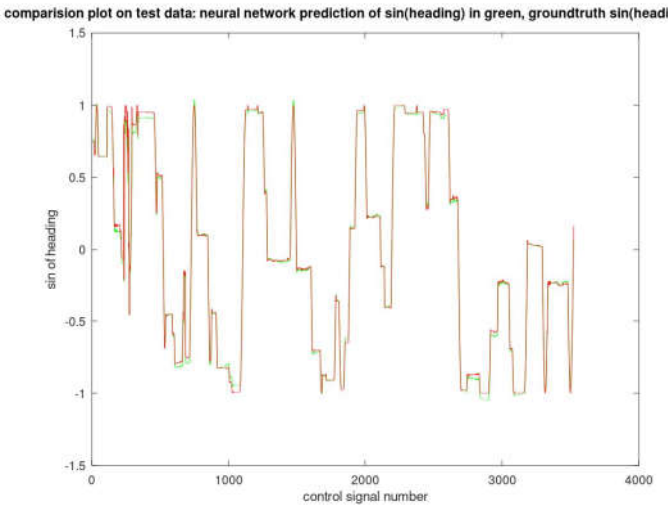


Figure 16

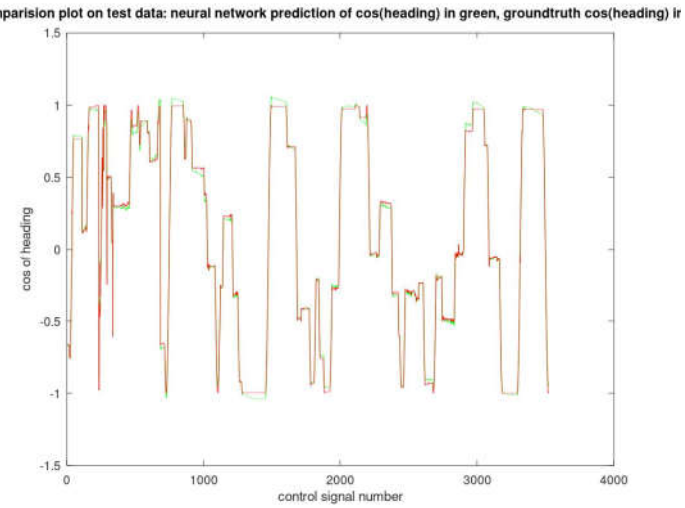


Figure 17

The dataset has been split into training data and test data, and performance is analysed for both the datasets. The training data contains 8000 examples and testing set contains 3523 examples (70-30 split). Figure 12 shows the comparison between model's estimated position output(in green) vs the actual positions(in red).

Just on a visual analysis it appears that the system has learnt the motion model quite well and is able to predict the motion of the robot fairly accurately. Figures 13 and 14 show the comparison of the sin and cosine values for training data. It appears that the sin values fits much better than the cosine values, but overall the fit appears pretty good. (neural net predictions in green, actual values in red).

The next set of 3 images show the same results, but on the test dataset. It's seen that the fit isn't as close as that on the training data, but still quite accurate. Other than visual inspection the following methods give a quantifiable way to analyse the fit of the data.

The first method is using a Mean Squared error to analyse the quality of the regression fit. Another way is the R^2 method. Mean square, as its name suggest, gives an error estimate of the expected values w.r.t the actual values. Lower the mean-squared error, better the estimate of the model. R^2 tries to see how well the fit describes the variability of the data about its mean; If R^2 is 100% then that implies that the fit describes the variability perfectly. [3]

$$MSE = [\sum_{i=1}^{N} (f_i - y_i)^2 / N]$$

$$(z_{f_i} - z_{o_i}) = difference\ between\ predicted\ and\ actual, N$$

Coefficient of determination R^2 :

$$SS_{tot} = \sum_i (y_i - \bar{y})^2$$

$$SS_{res} = \sum_i (y_i - f_i)^2$$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

$$SS_{tot} = total\ sum\ of\ squares, SS_{res} = residual\ sum\ of\ squares$$

MSE for training set =

| MSE_x_coordinate | MSE_y_coordinate | MSE_sin(heading) | MSE_cos(heading) |
|---|---|---|---|
| 0.0015219 | 0.0026156 | 0.0017775 | 0.0021752 |

$R^2$ for training set =

| x_coordinate | y_coordinate | sin(heading) | cos(heading) |
|---|---|---|---|
| 0.99887 | 0.99962 | 0.99674 | 0.99499 |

MSE for test set =

| MSE_x_coordinate | MSE_y_coordinate | MSE_sin(heading) | MSE_cos(heading) |
|---|---|---|---|
| 0.0025450 | 0.0034131 | 0.0026013 | 0.0046155 |

$R^2$ for test set =

| x_coordinate | y_coordinate | sin(heading) | cos(heading) |
|---|---|---|---|
| 0.99840 | 0.99941 | 0.99462 | 0.99077 |

As we can see, the training set has very low MSE value, and an R^2 value almost touching a 100%. This suggests a surprisingly good estimate of the outputs from the training data. Upon first inspection it might

appear that the model has overfit the training data, however when the neural net is applied on test data its performance is also excellent. We can see that there is an increase in MSE, and a decrease in R^2 compared to the training data; however it still gives a good estimation of the robot's state. It can be concluded that the model has satisfactorily learnt the behaviour of the robot in its current environment.

A few observations and points from literature about neural nets:

1) Backpropogation helps create new features which are not explicitly defined in the input layers which can help create complicated representations. However this can also mean that weights are being used to fit idiosyncrasies of training example that are not representative of the general distribution of examples, which implies over-fitting.
2) Using neural nets in this example doesn't help us gain any explicit understanding of the behaviour of the robot. Even though 'new features' are being created in the hidden layer, we do not get an understanding of what these values imply.
3) Neural networks have slow training times, however don't take much time for making new predictions, once the weights are learnt.

[1]Dean Pomerleau,(1992) "Neural Network Perception for mobile robot guidance", School of computer Science, Carnegie Mellon University.

[2]Tom M Mitchell "Machine Learning"

[3]Barnston, A., (1992). "Correspondence among the Correlation [root mean square error] and Heidke Verification Measures; Refinement of the Heidke Score." Notes and Correspondence, Climate Analysis Cente

[4]"Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program ., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1474927