Mobile robot navigation using Artificial Neural Networks

Vinoth Vincent Raj

ABSTRACT:

This report describes the use of a simple artificial neural networks rather than a multi-layer neural network to direct the navigation of mobile robot in a familiar and trained environment. A single layer neural network trained with examples of values representing the typical static obstacles to be experienced by the robot and the necessary actions to be taken . The model represents the rules defining how the robot should move in order to make a circular path across the hall and to avoid a clash against the obstacle. A back propagation algorithm is used to train the network. The model has been implemented in robot as well as in simulation software. This report discloses the results of the model that demonstrates the ability of the model in handling different obstacles placed at random while in movement and the selection of the optimum parameters of the model. The development of techniques for autonomous mobile robot operation constitutes one of the major trends in the current research and practice in modern robotics.

LITERATURE REVIEW:

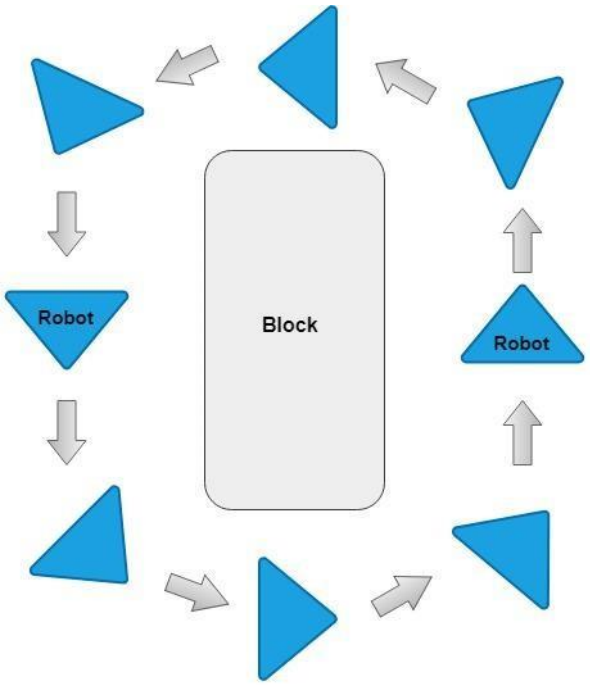
This section explains the literature analysis of the different techniques in navigation of mobile robot. The major problem in mobile robotics are obstacle avoidance and navigation in dynamic and static environment, which are being cracked by the diverse researchers in the last two decades.

Several researches have been made in the field of neural networks to solve the path-planning problem [1] Solutions to problems of the robot navigation were addressed by neural networks by Bekey [1]. For collision free methods, Domany and Hemmen [2] used recurrent neural network to avoid obstacles. A literature view of neural networks and its uses in mobile robotics have been explained by Zou et al. [3]. A feed forward neural network with multiple layers, which controls the steering degrees of angle of the robot automatically in the constant and moving environments, was designed by Singh & Parhi [4]. The block distances are the inputs to the neural network with four layers, and the steering angle is the output. Path planning in obstacle avoiding becomes more challenging when the robot is moving in a rapid inconstant environment. Using reinforcement learning and neural networks, Motlagh et al. [5] have explained the target following system, and block avoidance behaviours. Gavrilov & Lee [6] presented the robot navigation by using hybrid neural network. Hopfield neural network for path selecting and collision avoidance in the dynamic environment have been used by Glasius et al. [7]. Type-2 fuzzy neural network (IT2FNN) to solve the obstacle avoidance have been designed by Kim CJ, Chwa Din [7] Neural network for robot navigation in trained environment is presented by Chohra A [8]. A Jordan architecture of neural network solution in robot navigation is proposed by Tani J [9].Kohonen-type artificial neural network using a vision camera for navigation of mobile robot is designed by Mahmud et al. [10].

Matveeva & Teimoori consider the difficulty of navigation and guidance of a wheeled mobile robot towards a target based on the measurements concerning only the distance from the robot to the target.

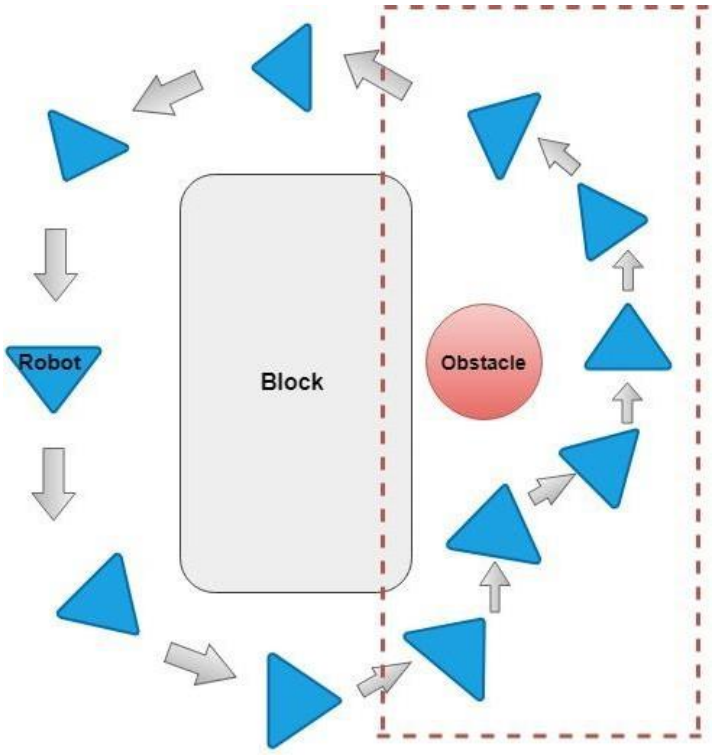
AIM AND OBJECTIVE:

The comprehensive aim of this coursework is to inspect the application of deep learning approaches to navigate mobile robots. In this report, a simple artificial neural network have been used to solve mobile robots navigation challenges. The aim of autonomous mobile robots is to navigate and move across physical objects without human intervention in real-world environments. The challenge is to navigate the mobile robot by identifying the obstacles in front of it and to navigate in circular path without colliding with the wall or obstacle. The data provided has covered different scenarios where the robot has successfully made navigation and able to move without colliding the obstacles. The different speed and velocity of the robot wheels is passed in the data according to its respective sensor reading. The aim of the robot demonstration is also shown in the figure below.



Objective for the robot to detect edges and navigate

Detecting and avoiding obstacles can be quite challenging for the robot and the data taken for training has variety of situations tested with obstacles. It becomes necessary for our network to learn all the patterns and to mimic the same behaviour. The expected scenario when the robot is faced with obstacle is shown in the figure below



Expected Robot Behaviour to train

The objective and aim of the network is quite straight forward, next, the architecture of the network selected and parameter tuning and working of the model will be discussed in the upcoming sections

DATA PREPERATION:

Training a neural network to mimic the data with minimum loss is a iterative process and pre-processing data aids in achieving better results. Sophisticated algorithms will not work well on poor data therefore cleaning data becomes vital. Some of the data pre-processing checks are done in the data as part of data cleaning process. Firstly, the check for missing values are cleared with no missing values in training data. Secondly, the dataset comprises of 3 million training data since, the training data is captured in every milliseconds there is a high chance of getting duplicates in the training data. Neural network necessarily will not learn from duplicate data. Moreover, training with duplicate data will affect the network performance in terms of time complexity. Hence, duplicates in the data has been removed and the final data was left over with 3123 unique instances.

Neural network is sensitive to the scale of each features and it becomes hard for the network to equate different scaled features. Normalising data will make features to be more consistent with each other and allow the network to evaluate the data in an efficient way. In addition, it helps the data to be converged easier. Moreover, the sigmoid activation function, which was used in the neural network, also outputs between zero and one, which works well with the normalised data. Finally, fixed the outlier in the training data based on the quartile range calculation.

Once the data has been pre-processed, data were split into train, validation and test. The split ratio of train retains 70% of data and test and validation each retains 15 % of the entire data. After training the data, the output is again de-normalised and passed to the robot.

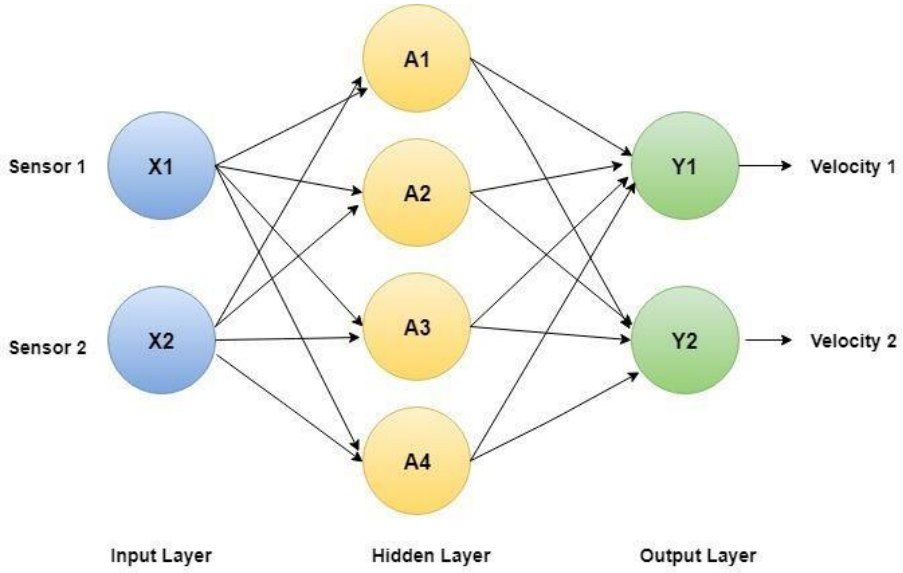
ARCHITECTURE:

The architecture of artificial neural networks ANN that connects neurons to other neurons in other layers forms a network, which can be perceived into three layers. The first one is an input layer, which has a size of the features of the input data. The data is forwarded to the hidden layer, which comprises of one or more neurons that can be selected independently and each neuron in hidden layer processes the data with suitable mathematical operations. The data, which was processed, is then forwarded to the output layer that computes the data by the use of activation function. The ANN system architecture can be feed forward or feedback ward type. The feed forward does not propagate back the output error to the hidden layer, and the data is allowed to move in a forward direction towards the output layer. Conversely, in back propagation the output error gradient could propagate backward.

Input Layer – Input layer passes the information from the actual data. The pre-processed data from the input is passed in a raw format in the input layer. It should also be noted that input does not compute any function or calculation, it just act as layer to pass the information to the next layer. The number of nodes in the input layer depends on the number of features of the input data. In our study, the robot has two inputs; they are the sensor reading one and sensor reading two.

Hidden Layer – The hidden layer does computations, which would not be the outcomes the model is expecting. They transfer information from the input layer to the output layer after performing the computations. In our problem, a network size with one hidden layer is used. Even though increasing the number of hidden layers enables the model to learn complex patterns in the dataset, it also causes slowness in learning and converging to global minimum. Four neurons are selected in the hidden layer as per our architecture.

Output Layer – Output layer computes the data from the hidden layer and computes the final output. This output will then be compared with the actual value and loss will be calculated. The neurons in the Output layer depends on the number of features to predict. In our problem, it is required to predict the values of the robot's two wheels. Hence, two neurons is selected in the architecture.



Artificial Neural Network Architecture

For the neural network robot, the input parameters are as follows.

Sensor 1 (45 degrees left) = x1

Sensor 2 (90 degrees front) = x2

The robot controller output is the motor velocity of the right and left wheels of the mobile robot

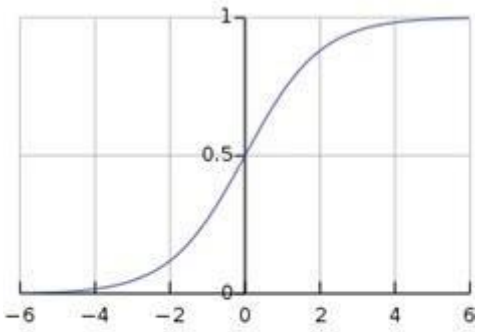
Motor speed (right wheel) = y1

Motor speed (left wheel) = y2

To achieve better performance, a single layered neural network is used. The neural networks were trained with 3000 patterns. These patterns represented typical scenarios that a robot may experience in real environments. An appropriate architecture is selected for the neural networks that has an input layer consisting of 31 nodes in the input layer, 17 nodes in the first hidden layer, and an output layer that indicates the corresponding velocity of the both wheels of the robot. The numbers of hidden neurons are empirically found with trial and error basis to be convergent of error to a minimum error rate.

SIGMOID FUNCTION:

Introducing non-linearity function helps the model to learn a variety of data and to interpret between outputs. Our network uses sigmoid activation function in order to apply nonlinear separations and it is easy to compute the derivatives for sigmoid function. In addition, sigmoid function performs well in producing probability outputs between zero and one. Sometimes network may experience slowness to learn, as gradient will be very close to zero over a huge part while using sigmoid function.



Sigmoid function

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^{x}}{e^{x}+1}.$$

FORWARD PROPAGATION:

Forward propagation starts with the input layer where the input values are multiplied by the weights between the input and hidden layer. A product of input values and weights are made and passed into the neurons in the hidden layers excluding the bias in the hidden layer where a sigmoid activation function is applied to reduce the values of the input and weight products to a value between zero and one. This network architecture is capable of producing non-linear separation between the data. Hence, each neuron produces output, which is then passed as inputs to the output layer. Again, the same process of weights and products of the hidden layers outputs is taken as input to the output layer. The neurons in the output layer computes the weighted product with bias. A nonlinear activation function, sigmoid, is applied to produce the outputs in the output layer.

As per the problem definition, the network is expected to produce two outputs that is the two speeds for the right and left wheel of the robots. Training a neural network refers to a approach that repeatedly updates the weight of the connections between the input nodes in a specified number of epochs so that error between the actual and predicted data is minimized significantly

$$\textbf{Network formulation: } y_k = \varphi[\sum_{i=0}^{n} w_{ki} h_i] = \varphi[\sum_{i=0}^{n} w_{ki} \varphi^h (\sum_{j=0}^{m} w_{ij}^h x_j)]$$

$$x_0 = h_0 = +1, \quad w_{k0} \text{ and } w_{i0}^h \textbf{ are biases.}$$

LOSS FUNCTION:

In order to verify how the good the model has learned it becomes necessary to compare the predicted value with the actual value of training data. Since the problem is a regression problem, a root mean squared error has been implemented to check the loss between actual value and the predicted value by the network. It produces always a non-negative number and a rmse value near to zero could be considered as a good fit of the model. It is sensitive to outliers and it is related to the size of the squared error, hence large errors have huge impact on the loss value.

$$\textbf{Error function: } \varepsilon(t) = \frac{1}{2}\sum_{k=1}^{l} e_k^2(t) \ , \quad e_k(t) = d_k(s(t)) - y_k(s(t))$$

BACK-PROPAGATION:

In this report, the backpropagation algorithm is used, which deploys the gradient method to train the network. Through many experiments in trails, it was succeeded in gaining a much better performance. Once the loss function returns the loss, it becomes necessary to adjust the weights of the network according to the loss it has incurred. Hence, backpropagation uses gradient descent technique to minimise the error function. In simple words, the algorithm tries to find the weight value such that the cost function becomes minimum. Gradient descent helps the algorithm to decide whether to increase or decrease the weights by providing the correct direction. Gradient decent performs the following functions. It tries to find the relationship between the loss function and the output values. It takes the derivative of the loss function multiplied by the derivative of the activation function in the output layer multiplied by the regularisation term to compute the gradients for the output layer.

$$\frac{\partial \varepsilon(t)}{\partial w_{ki}(t)} = \frac{\partial \varepsilon(t)}{\partial e_k(t)} \cdot \frac{\partial e_k(t)}{\partial y_k(t)} \cdot \frac{\partial y_k(t)}{\partial v_k(t)} \cdot \frac{\partial v_k(t)}{\partial w_{ki}(t)}$$

$$= -e_k(t) \cdot \varphi'(v_k(t)) \cdot h_i(t) = -\delta_k(t) \cdot h_i(t)$$

$$\delta_k(t) = e_k(t)\varphi'(v_k(t)) = \lambda \cdot \varphi(v_k(t)) \cdot [1 - \varphi(v_k(t))] \cdot e_k(t)$$

For the hidden layer, the summation of the gradient calculated in the output layer multiplied by the sigmoid derivative of the inner layer and the respective weights of the neuron gives the gradient of the hidden layer. These gradients are used to add the already initialised weights. For the next iteration, a momentum value is added along with the gradient to the weights to converge to the global minimum faster.

$$\frac{\partial \varepsilon(t)}{\partial w_{ij}^{h}(t)} = \sum_{k=1}^{l} \frac{\partial \varepsilon(t)}{\partial e_k(t)} \cdot \frac{\partial e_k(t)}{\partial y_k(t)} \cdot \frac{\partial y_k(t)}{\partial v_k(t)} \cdot \frac{\partial v_k(t)}{\partial h_i(t)} \cdot \frac{\partial h_i(t)}{\partial v_i^{h}(t)} \cdot \frac{\partial v_i^{h}(t)}{\partial w_{ij}^{h}(t)}$$

$$= -\sum_{k=1}^{l} e_k(t) \cdot \varphi'(v_k(t)) \cdot w_{ki}(t) \cdot \varphi^{h}{}'(v_i^{h}(t)) \cdot x_j(t)$$

$$= -\sum_{k=1}^{l} \delta_k(t) \cdot w_{ki}(t) \cdot \varphi^{h}{}'(v_i^{h}(t)) \cdot x_j(t) = -\delta_i^{h}(t) \cdot x_j(t)$$

$$\delta_i^{h}(t) = [\sum_{k=1}^{l} \delta_k(t) \cdot w_{ki}(t)] \cdot \varphi^{h}{}'(v_i^{h}(t)) = \lambda \cdot \varphi^{h}(v_i^{h}(t)) \cdot [1 - \varphi^{h}(v_i^{h}(t))] \cdot [\sum_{k=1}^{l} \delta_k(t) \cdot w_{ki}(t)]$$

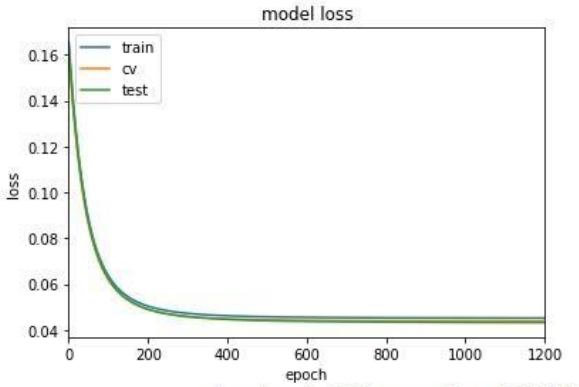
At every iteration as forward propagation yields output, loss function is calculated and based on the loss function gradients for the output and hidden layer weights are calculated and based on the gradient values the weights are updated at every iteration. This step continuous until a specific number of epochs is reached where the whole set is trained repeatedly to reach a minimum value in loss function.

PARAMETER TUNING:

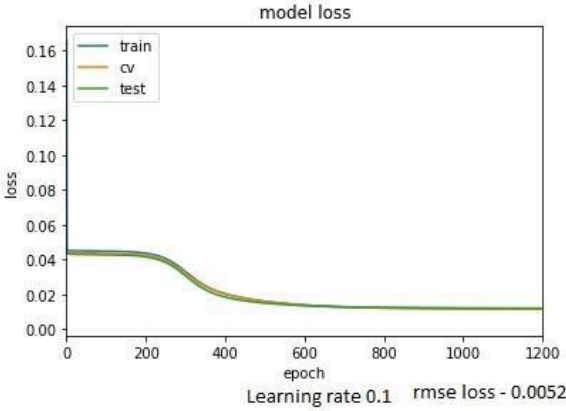
Choosing hyper-parameters is not straightforward unfortunately for any machine learning and deep learning algorithms. In addition, it relies on experimental outcomes rather than theory, and the best way to determine the optimum values is to try many different combinations of parameter values and evaluate the performance of the model. The neural network defined for this data has some hyper-parameters, which needs to be tuned in order to, obtain the minimized loss and they are learning rate, momentum, regularisation parameter and epochs. Justifications for choosing the optimal values are made in the upcoming paragraphs.

a) Learning rate:

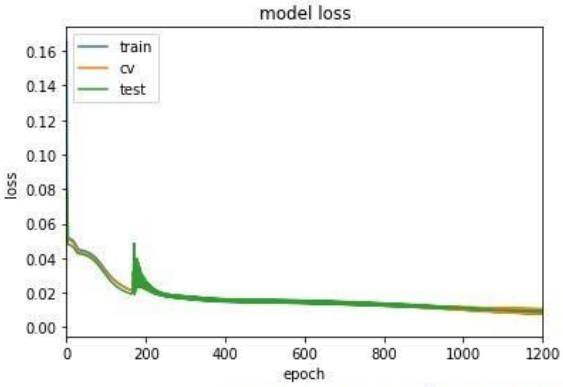
Learning rate is one of the most important parameter, which needs to be tuned in the neural network framework. Learning rate being large would make the network to learn very poorly and learning rate being small would make the network to learn very slowly. Learning rate values of 0.01 to 0.9 have been passed into the network and the rmse loss have been examined. Clearly learning rate of 0.3 achieved a low rmse loss when compared to other parameters. The parameter converges to global minimum and it started increasing in the greater parameter values



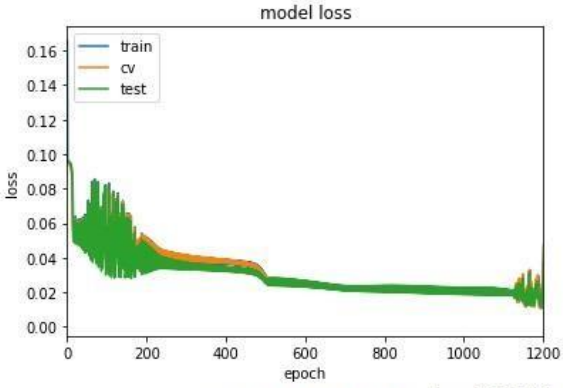
learning rate 0.01 rmse loss - 0.021600



Learning rate 0.1 rmse loss - 0.005239



Learning rate 0.3 **rmse loss - 0.003446**

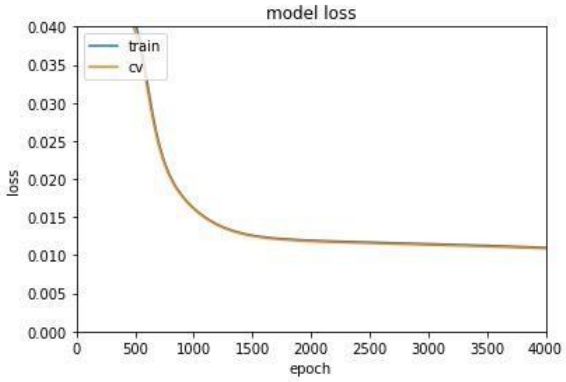


Learning rate 0.6 rmse loss - 0.010779

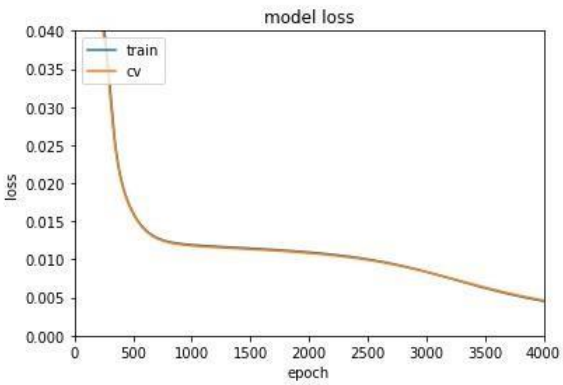
b) Regularisation:

Regularisation penalises the weight matrices of the hidden and output nodes to prevent overfitting. Regularisation parameter will added to the cost function. The regularisation parameter is called lambda, which represents the degree of regularisation. A lambda value of zero results in no regularisation and a large value of lambda corresponds to more penalising the weights. Bias at each layer will not be regularised. Increasing lambda significantly can make many neurons to zero or null and can certainly make high variance network to high bias network. So choosing the value of lambda is imperative.

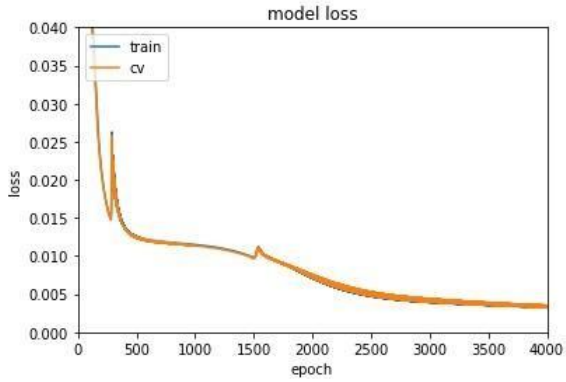
Regularisation Parameter Tuning:



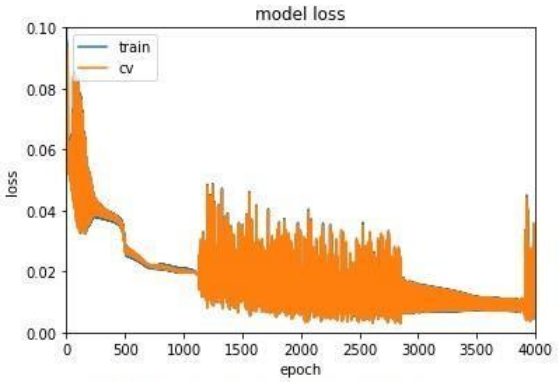
regularisation - 0.05 rmse loss - 0.010982



regularisation - 0.09 rmse loss - 0.006618



regularisation - 0.3 **rmse loss -0.003446**



regularisation - 0.6 rmse loss - 0.01913

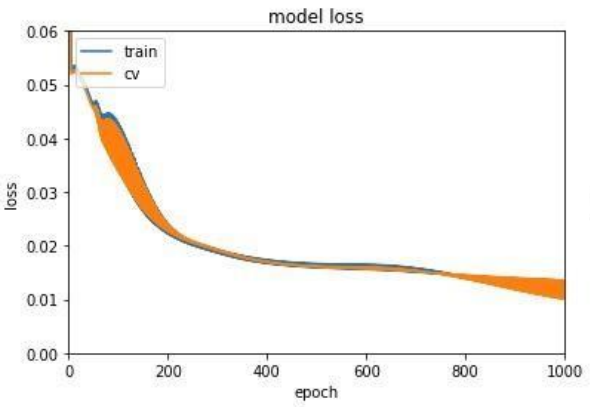
c) Momentum:

Momentum uses the gradient of the past iterations rather than the current step gradient to decide the direction to convergence. It is a moving average of the gradients, which is used to update the weights of the network. Momentum works better in our data as derivatives are noisy and it was estimated on a small batch of data. As gradient descent optimises a function by taking a small taking a small step in the gradient, it makes minimal improvement at every iteration and has higher probability to converge in a local minimum. Momentum with gradient descent speeds up the converge rate.

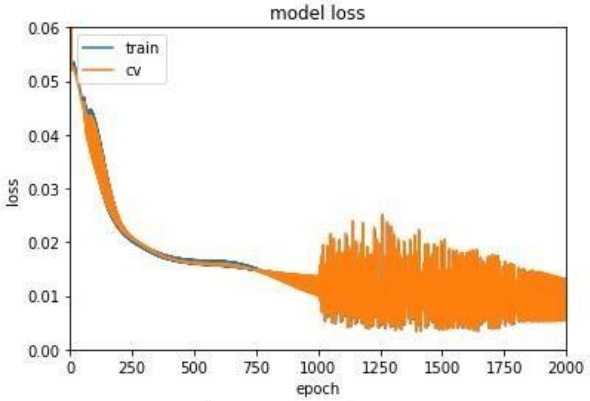
d) Epochs:

Epochs represents the count of times the network has trained the entire dataset. Selecting the number of epochs is straightforward and depends heavily on the neural network architecture and the complexity of the data. Less number of epochs lead to network not trained properly and with larger number of epochs, the validation accuracy might go down or make no progress in improvement. Decision has been made in our data set by examining the loss of the training and validation data. Epochs have been started with 1000 and gradually increased until it converges at a very low rate. After 4000 the epochs failed to converge significantly and the training has been stopped. The graph below explains the loss for the number of epochs and the epochs with 4000 shows the significant decrease in loss.

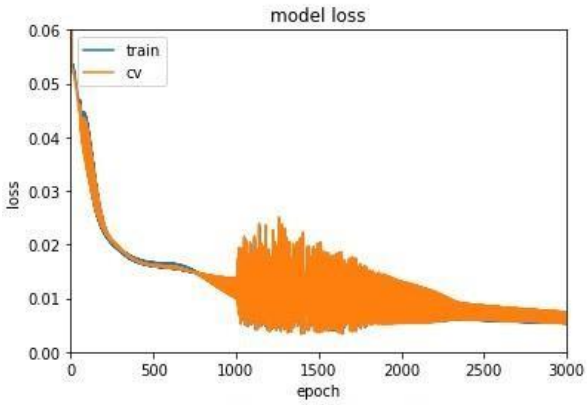
Epochs optimal value:



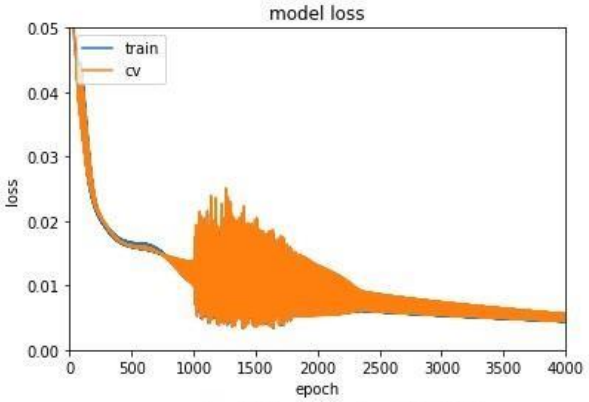
epochs -1000 rmse loss - 0.014530



epochs -2000 rmse loss - 0.007869



epochs -3000 rmse loss - 0.005629



epochs -4000 rmse loss - 0.004661

TESTING IN MOBILE SIMULATOR:

The model has been trained with the optimum parameters and a low root mean squared error has been achieved. Since, the model has been trained with different parameters values to reduce the loss function, it has reached its maximum training and training more would not result in a significant increase in the performance of the neural network. Hence the weights of the hidden layer and output layer along with biases have been extracted and passed into a simple feed forward c++ code as the respected weights of the feed forward neural networks hidden weights and output weights with the optimal biases. The code for connecting the robot has been provided in the Aria library, the sensor readings are pointed as input for the feed forward neural network, and the extracted weights of hidden and output layer in trained network are set as weights and bias. The robot when functioning reads the sensor values it was experiencing and passes into the feed forward neural network for every milliseconds The network receives the sensor readings as input and computes the velocity speed for the motor through the optimum trained weights and biases.

Upon testing in the mobile simulator, the robot performs decently well, where it was able to detect the edges of the wall and it makes sharp turns. Two use cases were tested in the mobile simulator to check if the robot was able to mimic what it has learned in the training period. Top right turn and bottom right turn. As the figure shows below, the robot mimics the behaviour of the trained data.

The training of the network is satisfactory by checking the results obtained in the mobile simulator. The same has to be tested in a real time scenario in where the actual data has been trained to check how the robot performs in a real time scenario.

TESTING IN REAL ENVIRONMENT:

The connection has been coded to connect to the robot. A similar environment has been setup where the robot has been trained initially to gather the input data. A rectangle shaped wall is placed and the robot is placed near the wall in such a way that that the sensor is able to pass the values to the network for the wall placed in left hand side. Now that the robot is placed in its appropriate position, the code has been executed and the robot was able to pass the sensor reading to the network and the network computes the velocity of the robot wheels for each sensor values. In testing in real environment, the robot did made a one complete circle without colliding with obstacles in the first round. However, in the second round, the robot has detected the obstacle and while crossing the obstacle it assumes that as a turn in the wall and collides vertically against the wall. It is a common and expected scenario as the robot was not trained using such scenarios.

The performance of the robot confirms that a simple artificial neural network would demonstrate the finest results in mimicking a behaviour, provided, it has been trained to its optimum level. A simple neural network with one hidden layer and four neurons with a sigmoid activation function provides satisfactory results in demonstrating a simple robot navigation in detecting edges and making a circle without clashing with the obstacles

CONCLUSION:

In this report, the performance of the simple artificial neural network for mobile robot navigation has been tested in real environment. The results proved that a neural network with one hidden layer and four neurons performed extremely well upon testing in real environment and in simulator. It proves that real time online training would yield satisfactory results in navigating robots when regularisation and backpropagation with momentum is applied while training the network weights. The main challenge in this problem was to detect the edges and obstacle and reduce and adjust the velocity speed for both right and left wheels so that that the robot avoids collision and makes a smooth turn across the wall, which was achieved effectively through this method. In future, providing data reading of different kind of obstacles and training to its optimum level would help tune the network to continuously navigate through the environment without any controllers.

REFERENCES:

1. Bekey, G. A. & Goldberg, K.Y. (1993). Neural Networks in Robotics. Kluwer Academic Publishers, ISBN 0- 7923-9268-X, Boston

2. Domany, E.; Hemmen, J.L. & Schulten, K. (1991). Models of Neural Networks. Springer Verlag, ISBN 3-540-51109-1, Berlin

3. Zou AM, Hou ZG, Fu SY, (2006) Tan M Neural Networks for Mobile Robot Navigation: a Survey. Advances in Neural Networks-ISNN, China, pp. 1218-1226.

4. Singh MK, Parhi DR (2011) Path Optimisation of a Mobile Robot Using an Artificial Neural Network Controller. International Journal of Systems Science 42(1): 107-120

5. Motlagh O, Nakhaeinia D, Tang SH, Karasfi B, Khaksar W (2014) Automatic Navigation of Mobile Robots in Unknown Environments. Neural Computing and Applications 24(7): 1569-1581.

6. Gavrilov AV, Lee S (2007) an Architecture of Hybrid Neural Network Based Navigation System for Mobile Robot. IEEE Seventh International Conference on Intelligent Systems Design and Applications, Brazil, pp. 587-590

7. Glasius R, Komoda A, Gielen SC (1995) Neural Network for Path Planning and Obstacle Avoidance. Neural Networks 8(1): 125- 133.

8. Chohra, A.; Sif, F. & Talaoubrid, S. (1995). Neural Navigation Approach of a Mobile Robot in a Structured Environment.

9. Tani.J (1996). Model-based Learning for Mobile Robot Navigation from the Dynamical Systems Perspective. IEEE Trans. on Syst ISSN 1083-4419

10. Mahmud F, Arafat A, Zuhori ST (2012) Intelligent Autonomous Vehicle Navigated by Using Artificial Neural Network. IEEE International Conference on Electrical and Computer Engineering, Bangladesh, pp. 105-108.

11. Xiao H, Liao L, Zhou F (2007) Mobile Robot Path Planning Based on Q-ANN. IEEE International Conference on Automation and Logistics, China, pp. 2650-2654.

12. Prof. Hani Hagras Neural Network Architecture and Learning , Moodle, University of Essex

APPENDIX:

Python Code:

```
import pandas as  pd
from sklearn.utils import shuffle
import numpy as np
from random import shuffle
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from math import *
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import mean_squared_error
neurons = 4
epochs = 4000
regularisation_parameter= 0.2
learning_rate=0.4
momentum=0.1
def load_file():


    df = pd.read_csv("M:\\nntrainingdata_copywoheader.csv", header=None)
    df.columns = ['X_1', 'X_2', "Y_1", "Y_2"]
    return df
def remove_duplicates(df):
     df = df.drop_duplicates(keep=False, inplace=True)
     return df

def get_dimensions(df):
       print(df.head())
       print(df.shape)
       print(df.describe())

def fix_outliers(df):

      df.loc[df["X_1"] > 1500, "X_1"] = 1500
```

```
    #  df.loc[df["X_2"] > 4500, "X_2"] = 4500
      return df

def normalise(df):

      df_min = df.min()
      df_max = df.max()
      df_std = df.std()
      ##Applying normalisation in the data frame
      normalized_df= (df - df.min()) / (df.max() - df.min())
      return normalized_df,df_min,df_max,df_std
In [29]:
def Split_train_test_cross_validation(df):

      # Seperating reposne and Predictor varibales

      X = df.iloc[:,[0,1]]
      Y = df.iloc[:,[2,3]]

      #Splitting data into train , split and validation

      X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.15, rand
om_state=1)
      X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=
0.30, random_state=1)

      print(X_train.shape , y_train.shape)
      print(X_val.shape , y_val.shape)
      print(X_test.shape, y_test)

      #Applying Transpose to pass it to neural network framework
      X_train = np.array(X_train).T
      y_train = np.array(y_train).T

      X_val = np.array(X_val).T
      y_val = np.array(y_val).T

      X_test = np.array(X_test).T
      y_test = np.array(y_test).T

      return X_train,y_train,X_val,y_val,X_test,y_test
def updated_file_dimension(neurons,X_train,y_train):
        x_Shape = X_train.shape[0]
        y_shape = y_train.shape[0]
        neurons = neurons
        print("Input", x_Shape)
        print("Neurons", neurons)
        print("Output", y_shape)
def sigmoid(x):
    return 1/(1+ np.exp(-x * regularisation_parameter))

def sigmoid_derivative(x):
    return sigmoid(x)*sigmoid(1-(x))

def test_predict(weight_dict, X_test):

    wh, bh  = weight_dict["weights_hidden"], weight_dict["bias_hidden"]
    wo, bo = weight_dict["weights_output"], weight_dict["bias_output"]



    zh = np.dot(wh, X_test) +bh
    ah = sigmoid(zh)  #activation layer

    zo = np.dot(wo, ah) + bo
    ao_test = sigmoid(zo)
```

```
    #rmse_test = np.mean(np.square(y_test - ao_test))
    return ao_test
        def initialise_Weights():

            #Set seed to repeat the same values
            np.random.seed(144)
            weights_hidden = np.random.rand(4,2)
            bias_hidden = np.zeros((4,1))

            weights_output = np.random.rand(2,4)
            bias_output = np.zeros((2,1))

            #gradient momentum initillizations
            dw1_last = np.zeros((4,2))
            dw2_last = np.zeros((2,4))

            weight_dict = { "weights_hidden": weights_hidden, "bias_hidden": bias_hi
    dden, "weights_output": weights_output, "bias_output": bias_output, "dw1_last":dw1_
    last, "dw2_last": dw2_last}

            return weights_hidden,bias_hidden,weights_output,bias_output,dw1_last,dw
    2_last,weight_dict
        def forward_backward(X_train, y_train,X_val,y_val,weight_dict):

                ##Forward Propogation Training
                wh, bh  = weight_dict["weights_hidden"], weight_dict["bias_hidden"]

                wo, bo = weight_dict["weights_output"], weight_dict["bias_output"]
                dw1_last, dw2_last = weight_dict["dw1_last"], weight_dict["dw2_last"
    ]

                zh = np.dot(wh, X_train) + bh
                ah = sigmoid(zh)  #activation layer
                zo = np.dot(wo, ah) + bo
                ao = sigmoid(zo)

                error = y_train - ao
                rmse = np.mean(np.square(y_train - ao))

                ##Backward Propogation
                gradient_output = regularisation_parameter *    ( error * sigmoid_der
    ivative(zo))
                gradient_hidden = regularisation_parameter * np.dot(wo.T, gradient_o
    utput) * sigmoid_derivative(zh)

                delta_w1 = np.dot(gradient_hidden, X_train.T)
                delat_w2 = np.dot(gradient_output, ah.T)
                delta_b1 = np.sum(gradient_hidden, axis=1, keepdims=True)
                delta_b2 = np.sum(gradient_output, axis=1, keepdims=True)

                # update the weights with the derivative (slope) of the loss functio
    n
                wh += learning_rate*delta_w1 + momentum * dw1_last
                wo += learning_rate*delat_w2 + momentum * dw2_last
                bh += learning_rate*delta_b1
                bo += learning_rate*delta_b2

                weight_dict = { "weights_hidden": wh, "bias_hidden": bh, "weights_ou
    tput': wo, "bias_output": bo, "dw1_last":dw1_last, "dw2_last": dw2_last}


                return rmse,weight_dict
        def Validation_Data(weight_dict, X_val, y_val):

            wh, bh  = weight_dict["weights_hidden"], weight_dict["bias_hidden"]
            wo, bo = weight_dict["weights_output"], weight_dict["bias_output"]
```

```
            dw1_last, dw2_last = weight_dict["dw1_last"], weight_dict["dw2_last"]

            zh = np.dot(wh, X_val) + bh
            ah = sigmoid(zh)
            zo = np.dot(wo, ah) + bo
            ao_val = sigmoid(zo)

            #rmse_cv = sqrt(mean_squared_error(cv_a2, y_cv))
            rmse_cv = np.mean(np.square(y_val - ao_val))
            return rmse_cv
        df = load_file()
        remove_duplicates(df)
        #get_dimensions(df)
        fix_outliers(df)
        import sys
        df_n,df_min,df_max,df_std = normalise(df)
        In [39]:
        X_train,y_train,X_val,y_val,X_test,y_test = Split_train_test_cross_validatio
    n(df_n)

        [469 rows x 2 columns]
        Initialise Weights:¶
        We will initialise the weights with zerosrandom numbers and bias with zeros

        In [40]:
        #print(X_train.shape , y_train.shape)
        #print(X_val.shape , y_val.shape)
        #print(X_test.shape, y_test)

        #print(x_train.head())
        #print(y_train.head())
             #Applying Transpose to pass it to neural network framework
        #X_train = np.array(X_train).T
        #y_train = np.array(y_train).T
        In [41]:
        updated_file_dimension(neurons,X_train,y_train)
        Input 2
        Neurons 4
        Output 2
        In [42]:
        weights_hidden,bias_hidden,weights_output,bias_output,dw1_last,dw2_last,weig
    ht_dict = initialise_Weights()
        In [43]:
        rmse_trainingloss = []
        rmse_validationloss = []
        test_loss = []
        for i in range(1,epochs):


            rmse,weight_dict = forward_backward(X_train, y_train,X_val,y_val,weight_
    dict)

            rmse_trainingloss.append(rmse)
            rmse_validation = Validation_Data(weight_dict, X_val, y_val)
            rmse_test = Validation_Data(weight_dict, X_test, y_test)
            rmse_validationloss.append(rmse_validation)
            test_loss.append(rmse_test)
            if i % 200 == 0:
                    print("Epoch %i:\n Training RMSE-
    loss = %f      Validation data RMSE-loss = %f" %(i,rmse, rmse_validation))
        In [20]:
        #y_predicted.head()

        y_predicted =test_predict(weight_dict, X_test)
        y_predicted = pd.DataFrame(y_predicted.T,columns=["Y_1", "Y_2"])

```

```
        y_predicted["Y_1"] = df['Y_1'].min() + y_predicted["Y_1"]*(df["Y_1"].max() - df["Y_1"].min())
        y_predicted["Y_2"] = df['Y_2'].min() + y_predicted["Y_2"]*(df["Y_2"].max() - df["Y_2"].min())


        y_val = pd.DataFrame(y_val.T,columns=["Actual_Y_1", "ACtual_Y_2"])

        y_val["Actual_Y_1"] = df['Y_1'].min() + y_val["Actual_Y_1"]*(df["Y_1"].max() - df["Y_1"].min())
        y_val["ACtual_Y_2"] = df['Y_2'].min() + y_val["ACtual_Y_2"]*(df["Y_2"].max() - df["Y_2"].min())
        y_val.reset_index()
        y_predicted.reset_index()
        #compare = pd.concat([y_val, y_predicted], axis=1, sort=False)
        compare = pd.concat([y_val, y_predicted], axis=1)
        plt.plot(rmse_trainingloss)
        plt.plot(rmse_validationloss)
        plt.plot(test_loss)
        plt.title("model loss")
        plt.ylabel("loss")
        plt.xlabel("epoch")
        plt.legend(["train", "cv","test"], loc="upper left")
        plt.xlim(0, 1200)
        plt.show()
```

savefig("foo.png")

C++ Code:

```
// ConsoleApplication2.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

#include <Aria.h>
#include <iostream>
#include <vector>

using namespace std;

// 0.2 is our lambda value for regualrisation

double sigmoid_function(double x)
{
    return 1 / (1 + exp(-x * 0.2));
}

int main(int argc, char **argv)
{
    int leftSpeed = 100;
    int rightSpeed = 100;
    double Sensor0, Sensor1;


    /// Update the optimal weights for the hidden layer and bias at the last column

    /// Intiliase third input as 1 so that we multiply (W * x) + (1 * bias)

    double Weights_hidden[4][3] = {
        { -67.22442179,  -33.82059205 , 36.71329891 },
        { -23.55719207,  -63.26829714, 8.36408098 },
        { -43.45512204,   42.53989317, 4.3626676 },
        { -101.12989911,   22.00611327, 1.86436707 } };

```

```
35.
36.     /// Update the optimal weights for the hidden layer
37.     double Weights_output[2][5] = {
38.         { 39.93776553,  26.8948194 , -27.98823371,  39.63620943, -38.35517485  },
39.         {-10.57528398, -18.11828294, -11.41810711,  -6.36536815,9.75833391}
40.         };
41.
42.     double readings[3];
43.
44.     // create instances
45.     Aria::init();
46.     ArRobot robot;
47.     ArPose pose;
48.     // parse command line arguments
49.     ArArgumentParser argParser(&argc, argv);
50.     argParser.loadDefaultArguments();
51.
52.     ArRobotConnector robotConnector(&argParser, &robot);
53.     if (robotConnector.connectRobot())
54.         std::cout << "Robot connected!" << std::endl;
55.
56.     robot.runAsync(false);
57.     robot.lock();
58.     robot.enableMotors();
59.     robot.unlock();
60.
61.
62.     while (true)
63.     {
64.         //Getting the senor readings from the Aria robot
65.
66.         Sensor0 = robot.getSonarReading(0)->getRange();
67.         Sensor1 = robot.getSonarReading(1)->getRange();
68.
69.         //Implementing a simple forward propagation
70.
71.         //Getting inputs from the sensors and storing it in a variable:
72.
73.         readings[0] = Sensor0;
74.         readings[1] = Sensor1;
75.         //since we added bias in the third column of weights we will declare the th
    ird input wich will multiply the third column to one (to just add the bias)
76.         readings[2] = 1;
77.
78.         //Normalise the inputs as we will be passing it to a sigmoid_function funct
    ion.. Update the maximum and minimum values of each reading from the python code
79.         readings[0] = (readings[0] - 230.42) / (1500.000 - 230.42);
80.         readings[1] = (readings[1] - 630.17) / (4740.19 - 630.17);
81.
82.         //Initialise a variable to store the activation function values for each la
    yer
83.         double weighted_sum[4];
84.         double activation_values[5];
85.         double outputlayer_weighted_sum[2];
86.         double motor_velocity[2];
87.
88.
89.         //Forward Propogation for Hidden layer to calculate the activations of hidd
    en neurons and store in activation values:
90.
91.         for (int i = 0;i<4;i++)
92.         {
93.             // Initialise all the values to zero at first before storing the weight
    ed sum
94.             weighted_sum[i] = 0;
95.             //
```

```
96.             for (int j = 0;j < 3;j++)
97.             {
98.                 //Do the weight times input multiplication and store it in weight s
    um variable
99.                 weighted_sum[i] += readings[j] * Weights_hidden[i][j];
100.                }
101.                activation_values[i] = sigmoid_function(weighted_sum[i]);
102.
103.            }
104.
105.            //since we added bias in the third column of weights we will decalre
    the third input wich will multiply the third column to one (to just add the bias)

106.            activation_values[4] = 1;
107.
108.
109.            for (int i = 0;i<2;i++)
110.            {
111.                // Initialise all the values to zero at first before storing the
    weighted sum
112.                outputlayer_weighted_sum[i] = 0;
113.                for (int j = 0;j < 5;j++)
114.                {
115.                    //Do the weight times input multiplication and store it in w
    eight sum variable
116.                    outputlayer_weighted_sum[i] += activation_values[j] * Weight
    s_output[i][j];
117.                }
118.                motor_velocity[i] = sigmoid_function(outputlayer_weighted_sum[i]
    );
119.            }
120.
121.            motor_velocity[0] = 115.000 + motor_velocity[0] * (196.76 -
    115.000);
122.            motor_velocity[1] = 92.88 + motor_velocity[1] * (300.00 - 92.88);
123.
124.            // Set the speed
125.            robot.setVel2(motor_velocity[0], motor_velocity[1]);
126.
127.            // sleep the thread
128.            ArUtil::sleep(200);
129.
130.        }
131.
132.        robot.lock();
133.        robot.stop();
134.        robot.unlock();
135.        // terminate all threads and exit
136.        Aria::exit();
137.
138.    }
```