Intelligent Systems

Final Report

Design and development of a Recurrent Neural Network for Path Planning Algorithm

Name	: P.K.P.Kumara
Date	: 15.12.2019

INTRODUCTION

Obstacle Avoiding and path planning algortihm for mobile robots have been researched since long time. In unmanned vehicle controlling path planning is a major issue. The purpose of the path planner is to compute a path from the start position of the vehicle to the goal to be reached. Main two objectives of path planning algorithms are obstacle avoiding and find a realible path(optimum if possible) to go it's final location [1] . Lot of conventional logical algorithms were presented for this problem. Even since there are complete solutions to 2D path planning, Aritificial intelligent techniques have take a major part in 2D and 3D path planing to reduce the computational time and memory [1].

Path planning for complex shaped obstacles were done by using aritificail annealing algorithm [1]. But this requires a plan view of the environment. Sivaram [2] presents an algorithm which is implemented using parallel distributed model of neural network with three activation functions to determine the next consecutive moves to the cells for the actor. This algorithm uses reinforcement learning with weights determined dynamically in each iteration.

Recurrent neural networks are widely use in path planning due to its memorizing capability which is not have in conventional multilayer network.

METHOD

First I used an alogrithm to create the training set data. Then used an Recurrent nueral network for the system because its memorizing feature which was not avaialable in conventional Neural netwrok. Recurrent Neural netwroks are used in most of sequential tasks. So I used a simple version of Recurrent neral network for illustrate the useability of the recurrent neral netwrok for path finding AI algorithm. I used an Elman Netwrok [3] for the following system.

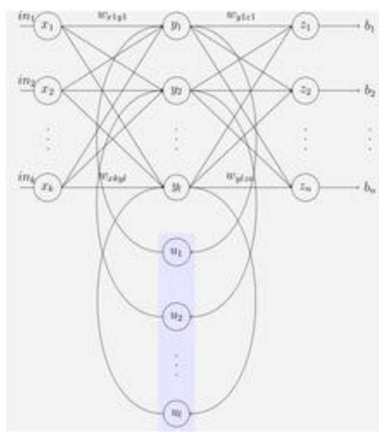


Figure 1 Simple Elman Network Architecture

Then I had to use 180 sensor readings for all 360 angles with 2 degree resolution and 2 inputs for position which was calculated by previous commands it took. Altogether 182 inputs and then I used another 3 layers including output layer with have 4 nodes for taking decisions to go forward, backward, right or left. 2nd layer consists of 100 neurons and 3rd layer consist of 50 neurons. The recurrent layer was added to 3rd layer which has another 50 neurons and values from the previous forward path.

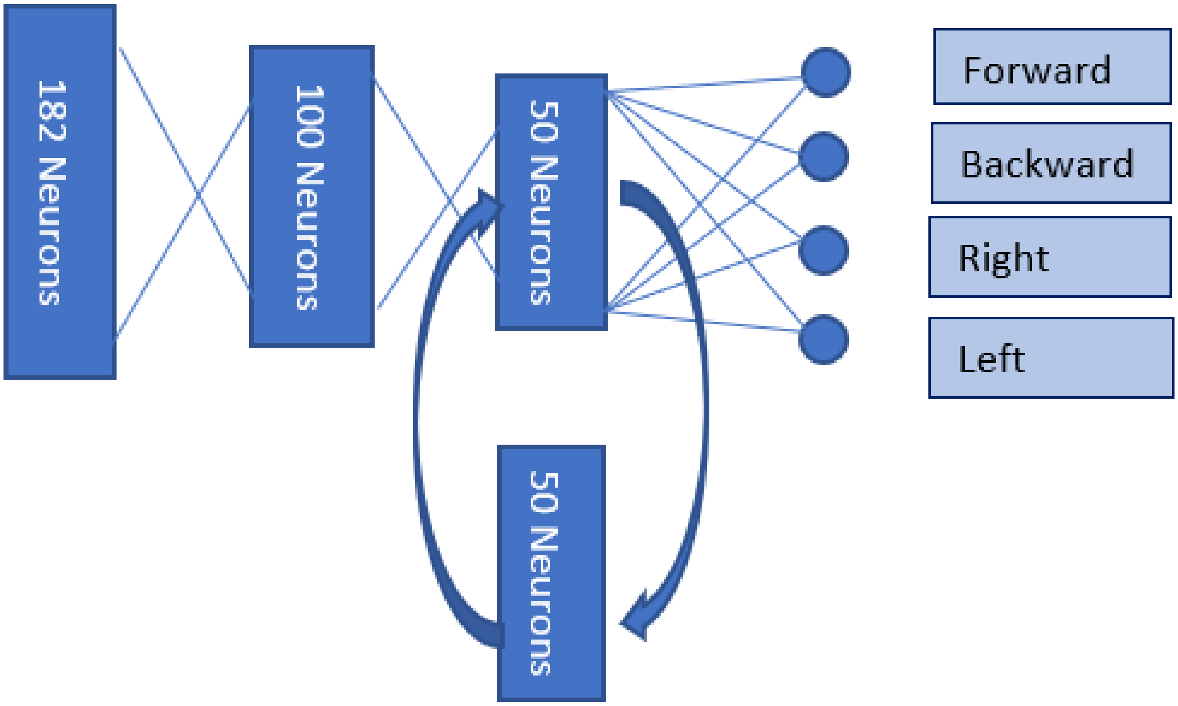


Figure 2 Designed Netwrok architecture

Final values from this network taken into calculate the probabality of executing each task using log probability function. Then the highest probability task will execute and the cost will calculate as log of the executed nodes value. If the probability is 1 cost will 0. Then back propagation has used to train the neral network.

Create Algorithm for getting sensor valus for given map

```
import numpy as np
import matplotlib.pyplot as plot

mapArrayFromView=[]
Forward_path=[]

F="Forward"
R="Right"
L="Left"
B="Backward"

mapArrayFromView.append([[1,1,1,1,1,1,1,1,1,1],
          [1,0,0,0,0,0,0,0,0,1],
          [1,0,1,1,1,1,1,1,0,1],
          [1,0,0,0,0,0,0,0,1,1],
          [1,1,0,0,0,0,0,0,0,1],
          [1,0,0,0,0,0,0,1,1,1],
          [1,0,1,1,1,1,1,0,0,1],
          [1,0,0,0,0,0,0,0,0,1],
          [1,0,0,0,0,0,0,0,0,1],
          [1,1,1,1,1,1,1,1,1,1]])
Forward_path.append([F,F,F,R,F,F,L,F,F,R,R,R,R,R,R,R])

mapArrayFromView.append([[1,1,1,1,1,1,1,1,1,1],
          [1,0,0,0,0,0,0,0,0,1],
          [1,0,1,1,1,1,1,0,0,1],
          [1,0,0,0,0,0,0,0,1,1],
          [1,1,0,0,0,0,0,0,0,1],
          [1,0,0,0,0,0,0,1,1,1],
          [1,0,1,1,0,1,1,0,0,1],
          [1,0,0,0,0,0,0,0,0,1],
          [1,0,0,0,0,0,0,0,0,1],
          [1,1,1,1,1,1,1,1,1,1]])
Forward_path.append([F,R,R,R,F,F,F,R,R,R,F,F,F,R])


mapArray=[mapArrayFromView[i][::-1] for i in range(0,len(mapArrayFromView))]

initialPosition=[1,1]
curr_position=[1,1]
```

```python
def lidarValues(Position,mapArray):
    lidarRange=360
    distanceMatrix=[]
    startangle=0
    for theeta in range(0,lidarRange,2):
        angle=startangle+theeta
        distanceMatrix.append(distance(Position,angle,mapArray))
    return distanceMatrix


def distance(Position,angle,mapArray):
    i=1
    j=1
    theetaD=angle
    theetaR=(theetaD*(np.pi)/180)
    l_from_sin=0
    l_from_cos=0
    while True:
        #avoid divide by 0 error
        if (theetaD==0) or (theetaD==180):
            l_from_sin=100000
            l_from_cos=(i-0.5)/np.cos(theetaR)
        elif (theetaD==90) or (theetaD==270):
            l_from_cos=100000
            l_from_sin=(j-0.5)/np.sin(theetaR)
        else:
            l_from_cos=(i-0.5)/np.cos(theetaR)
            l_from_sin=(j-0.5)/np.sin(theetaR)

        #checking for obstacle
        if abs(l_from_cos)>= abs(l_from_sin):
            if mapArray[Position[1]+(int(np.sign(l_from_sin))*(j))][Position[0]+(int(np.sign(l_from_cos))*(i-1))]==1:
                return abs(l_from_sin)
            else:
                j=j+1

        if abs(l_from_cos)<=abs(l_from_sin):
            if mapArray[Position[1]+(int(np.sign(l_from_sin))*(j-1))][Position[0]+int(np.sign(l_from_cos))*(i)]==1:
                return abs(l_from_cos)
            else:
                i=i+1
        if abs(l_from_cos)==abs(l_from_sin):
            if mapArray[Position[1]+int(np.sign(l_from_sin))*(j)][Position[0]+int(np.sign(l_from_cos))*(i)]==1:
                return abs(l_from_sin)


def lidarPlot(distanceValues,Startangle=0):
    plot.axes(projection='polar')
    plot.title('Circle in polar format:r=R')

    for i in range(0,len(distanceValues)):
        plot.figure()
        plot.polar((Startangle+i)*np.pi/180*360/len(distanceValues),distanceValues[i],'o')
    plot.draw()
```

Generate Training Set

```
def createTrainingSet(mapArray,Forward_path,initialPosition):
    Xx=[]
    Yy=[]
    for m in range(0,len(Forward_path)):
        Xx.append([])
        Yy.append([])
        curr_position=initialPosition
        for i in range(0,len(Forward_path[m])):
            l_values=lidarValues(curr_position,mapArray[m])
            Xx[m].append(l_values+curr_position)
            Y=decode_Output(Forward_path[m][i])
            Yy[m].append(Y)
            curr_position=nextPosition(curr_position,Y)
            if mapArray[m][curr_position[1]][curr_position[0]]==1:
                print("error")
                print("Check map and forward path:"+str(m))
                while (True):
                    pass

    return Xx,Yy

def nextPosition(prev_position,Y):
    if Y[0]==1:
        curr_position=[prev_position[0],prev_position[1]+1]
    elif Y[1]==1:
        curr_position=[prev_position[0]+1,prev_position[1]]
    elif Y[2]==1:
        curr_position=[prev_position[0]-1,prev_position[1]]
    elif Y[3]==1:
        curr_position=[prev_position[0],prev_position[1]-1]
    return curr_position

def decode_Output(direction):
    if direction=="Forward":
        return [1,0,0,0]
    elif direction=="Right":
        return [0,1,0,0]
    elif direction=="Left":
        return [0,0,1,0]
    elif direction=="Backward":
        return [0,0,0,1]

Xx,Yy=createTrainingSet(mapArray,Forward_path,initialPosition)
print (len(Xx),len(Yy))
print (len(Xx[0]),len(Yy[0]))
print (len(Xx[0][0]),len(Yy[0][0]))
```

Network Architecture

```
network_architecture = [
    {"layer_size": 182, "activation": "none"},
    {"layer_size": 100, "activation": "sigmoid"},
    {"layer_size": 50, "activation": "relu_RNN"},
    {"layer_size": 4, "activation": "relu"}
]

RNN_layer=2


def init_parameters(network_architecture,RNN_layer, seed = 3):
    np.random.seed(seed)
    parameters = {}
    number_of_layers = len(network_architecture)

    for l in range(1, number_of_layers):
        parameters['W' + str(l)] = np.random.randn(
            network_architecture[l]["layer_size"],
            network_architecture[l-1]["layer_size"]
            ) * 0.01
        parameters['b' + str(l)] = np.zeros((network_architecture[l]["layer_size"], 1))

    parameters['U' +str(RNN_layer)] = np.random.randn(
            network_architecture[RNN_layer]["layer_size"],
            network_architecture[RNN_layer]["layer_size"]
            ) * 0.01
    return parameters

def sigmoid(Z):
    S = 1 / (1 + np.exp(-Z))
    return S

def elu(Z, a=0.1):
    R = np.array(Z, copy = True)
    R[Z <= 0]= a*(np.exp(Z)-1)

def relu(Z):
    R = np.maximum(0, Z)
    return R

def softmax(Z):
    return np.exp(Z) / sum(np.exp(Z))

def sigmoid_backward(dA, Z):
    S = sigmoid(Z)
    dS = S * (1 - S)
    return dA * dS

def elu_backward(dA, Z,a=0.1):
    dZ = np.array(dA, copy = True)
    dZ[Z <= 0] = a*np.exp(Z)
    return dZ

def relu_backward(dA,Z):
    dZ = np.array(dA, copy = True)
    dZ[Z <= 0] = 0
    return dZ
```

Neural Network Forward Path

```
def L_model_forward(X,R_prev, parameters, network_architecture):
    forward_cache = {}
    A = X
    number_of_layers = len(network_architecture)
    forward_cache['A' + str(0)] = A

    for l in range(1, number_of_layers):
        A_prev = A
        W = parameters['W' + str(l)]
        b = parameters['b' + str(l)]
        activation = network_architecture[l]["activation"]
        if (l==RNN_layer):
            U=parameters['U' + str(l)]
            Z, A = linear_activation_forward(A_prev, W, b, activation,U,R_prev)
        else:
            Z, A = linear_activation_forward(A_prev, W, b, activation)
        forward_cache['Z' + str(l)] = Z
        forward_cache['A' + str(l)] = A

    AL=softmax(A)

    return AL, forward_cache

def linear_activation_forward(A_prev, W, b, activation,U=0,R_prev=0):
    if activation == "sigmoid":
        Z = linear_forward(A_prev, W, b)
        A = sigmoid(Z)
    elif activation == "relu_RNN":
        Z = linear_forward_RNN(A_prev, W, b, U,R_prev)
        A = relu(Z)
    elif activation == "elu":
        Z = linear_forward(A_prev, W, b)
        A = elu(Z)
    elif activation == "relu":
        Z = linear_forward(A_prev, W, b)
        A = relu(Z)

    return Z, A

def linear_forward(A, W, b):
    Z = np.dot(W, A) + b
    return Z

def linear_forward_RNN(A, W, b, U,R):
    Z = np.dot(W, A)+np.dot(U, R) + b
    return Z

def compute_cost(AL, Y):
    m = AL.shape[1]
    logprobs = np.multiply(np.log(AL),Y)
    cost = - np.sum(logprobs) / m
    cost = np.squeeze(cost)
    return cost
```

Back Propagation

```python
def L_model_backward(AL, Y,R_prev, parameters, forward_cache, network_architecture):
    grads = {}
    number_of_layers = len(network_architecture)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
    dAL= AL-Y
    dA_prev = dAL
    for l in reversed(range(1, number_of_layers)):
        dA_curr = dA_prev
        activation = network_architecture[l]["activation"]
        W_curr = parameters['W' + str(l)]
        Z_curr = forward_cache['Z' + str(l)]
        A_prev = forward_cache['A' + str(l-1)]

        if (l==RNN_layer):
            U_curr=parameters['U'+str(l)]
            dA_prev, dW_curr, db_curr,dU_curr = linear_activation_backward(dA_curr, Z_curr, A_prev, W_curr, activation,R_prev,U_curr)
            grads["dU" + str(l)] = dU_curr
        else:
            dA_prev, dW_curr, db_curr,dNU = linear_activation_backward(dA_curr, Z_curr, A_prev, W_curr, activation)

        grads["dW" + str(l)] = dW_curr
        grads["db" + str(l)] = db_curr
    grads["dU" + str(RNN_layer)] = dU_curr
    return grads

def linear_activation_backward(dA, Z, A_prev, W, activation,R_prev=0,U=0):
    dU=0
    if activation == "relu":
        dZ = relu_backward(dA, Z)
        dA_prev, dW, db = linear_backward(dZ, A_prev, W)
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, Z)
        dA_prev, dW, db = linear_backward(dZ, A_prev, W)
    elif activation == "elu":
        dZ = elu_backward(dA, Z)
        dA_prev, dW, db = linear_backward(dZ, A_prev, W)
    elif activation == "relu_RNN":
        dZ = relu_backward(dA, Z)
        dA_prev, dW, db,dU = linear_backward_RNN(dZ, A_prev, W, R_prev, U)
    return dA_prev, dW, db, dU

def linear_backward(dZ, A_prev, W):
    m = A_prev.shape[1]
    dW = np.dot(dZ, A_prev.T) / m
    db = np.sum(dZ, axis=1, keepdims=True) / m
    dA_prev = np.dot(W.T, dZ)
    return dA_prev, dW, db

def linear_backward_RNN(dZ, A_prev, W, R_prev, U):
    m = A_prev.shape[1]
    n = R_prev.shape[1]
    dW = np.dot(dZ, A_prev.T) / m
    dU = np.dot(dZ, R_prev.T) / n
    db = np.sum(dZ, axis=1, keepdims=True) / m
    dA_prev = np.dot(W.T, dZ)
    return dA_prev, dW, db, dU

def update_parameters(parameters,RNN_layer, grads, learning_rate,network_architecture):
    L = len(network_architecture)

    for l in range(1, L):
        parameters["W" + str(l)] = parameters["W" + str(l)] - learning_rate * grads["dW" + str(l)]
        parameters["b" + str(l)] = parameters["b" + str(l)] - learning_rate * grads["db" + str(l)]

    parameters["W" + str(RNN_layer)] = parameters["W" + str(RNN_layer)] - learning_rate * grads["dW" + str(RNN_layer)]

    return parameters
```

Train Nueral Network using generated traing sets

```
def L_layer_model(Xx, Yy, network_architecture,RNN_layer, learning_rate = 0.0075, num_iterations = 20, print_cost=False):
    np.random.seed(1)
    costs = []

    # Parameters initialization.
    parameters = init_parameters(network_architecture,RNN_layer)


    # Loop (gradient descent)
    for i in range(0, num_iterations):
        for n in range(0,len(Xx)):
            R_prev= np.zeros(
                (network_architecture[RNN_layer]["layer_size"],1)
                )
            cost_i=[]
            for j in range(0,len(Xx[n])):
                X=np.asarray(Xx[n][j]).reshape(len(Xx[n][j]),1)
                Y=np.asarray(Yy[n][j]).reshape(len(Yy[n][j]), 1)

                # Forward propagation
                AL, forward_cache = L_model_forward(X,R_prev, parameters, network_architecture)

                # Compute cost
                cost = compute_cost(AL, Y)

                # Backward propagation
                grads = L_model_backward(AL, Y,R_prev, parameters, forward_cache, network_architecture)

                # Update parameters
                parameters = update_parameters(parameters,RNN_layer, grads, learning_rate,network_architecture)

                R_prev=forward_cache['A' + str(RNN_layer)]
                cost_i.append(cost)
        # Print and graph the cost for 50 steps of all iterations
        if print_cost and i % (num_iterations/50) == 0:
            print("Total cost at iteration %i: %f" %(i, sum(cost_i)))
            costs.append(sum(cost_i))

    # plot the cost
    plot.figure()
    plot.plot(np.squeeze(costs))
    plot.ylabel('cost')
    plot.xlabel('iterations (per tens)')
    plot.title("Learning rate =" + str(learning_rate))
    plot.draw()

    return parameters
```

Testing for test set and triang set

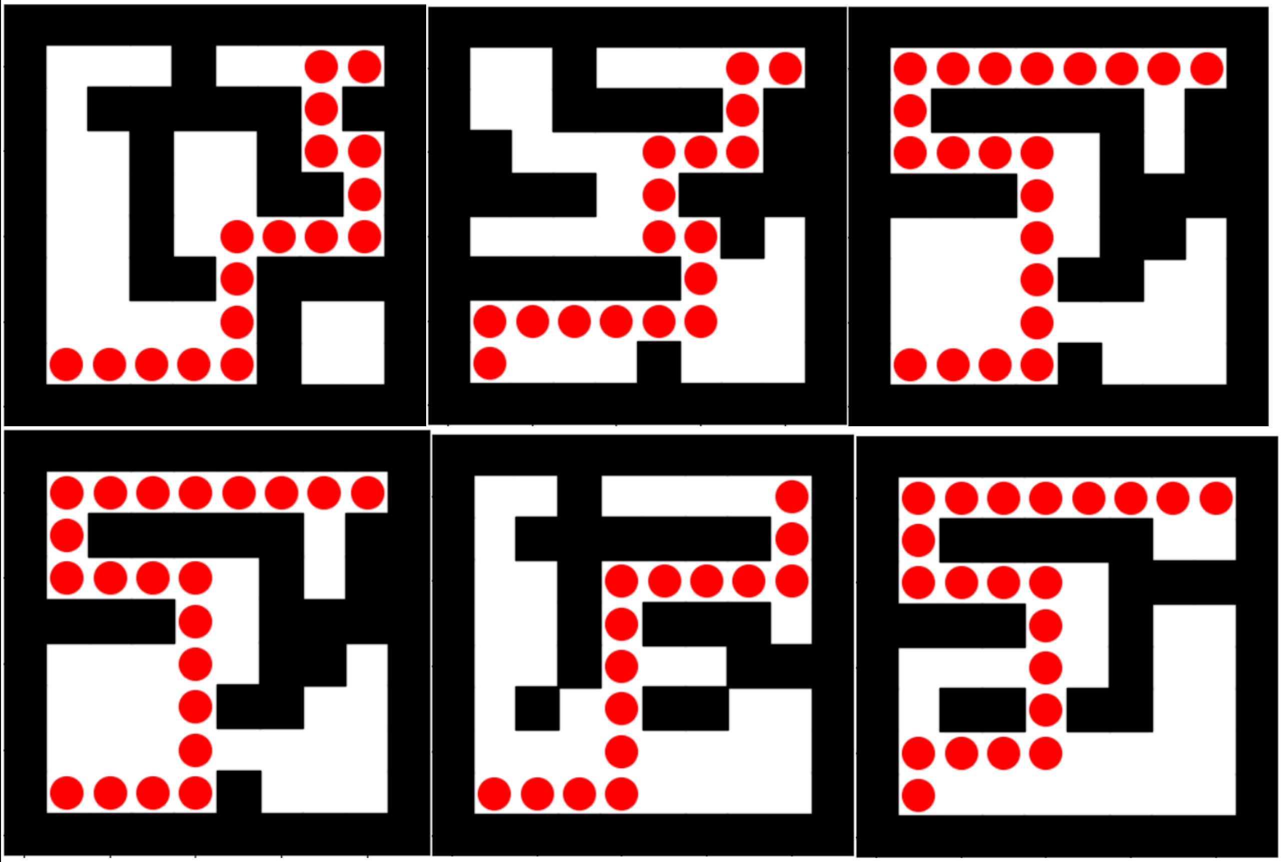
```
def testing(initialPosition, parameters, network_architecture,Testmap):
    Xt=[]
    Yt=[]
    positionMatrix=[]
    i=0
    R_prev= np.zeros(
                (network_architecture[RNN_layer]["layer_size"],1)
                )
    curr_position=initialPosition
    print(Testmap)
    while (i<50):
        positionMatrix.append(curr_position)

        if Testmap[curr_position[1]][curr_position[0]]==1:
            print("failed")
            break
        if (curr_position[1]==8) and (curr_position[0]==8):
            print ("Succeed")
            break
        l_values=lidarValues(curr_position,Testmap)
        Xt.append(l_values+curr_position)
        X=np.asarray(Xt[-1]).reshape(len(Xt[-1]),1)
        AL,forward_cache=L_model_forward(X,R_prev, parameters, network_architecture)
        print (AL)
        Y=get_Output(AL)
        print (Y)
        Yt.append(Y)
        #lidarPlot(l_values)
        print (curr_position)
        curr_position=nextPosition(curr_position,Y)
        R_prev=forward_cache['A' + str(RNN_layer)]
        i=i+1
    return positionMatrix


def testMapPlotValues(initialPosition,trained_parameters,network_architecture,testMap):
    positionMatrix=testing(initialPosition, trained_parameters, network_architecture,testMap)
    xP=[pM[0] for pM in positionMatrix]
    yP=[pM[1] for pM in positionMatrix]
    plotMap=np.where(testMap==np.amax(testMap))
    xM=plotMap[1]
    yM=plotMap[0]
    plot.figure()
    plot.gca().set_aspect('equal', adjustable='box')
    plot.plot(xM, yM, 's', color='Black', markersize=27);
    plot.plot(xP, yP, 'o', color='red',markersize=20);
    plot.draw()


def get_Output(AL):
    Y=[0]*4
    index_of_max_element=np.where(AL == np.amax(AL))[0][0]
    Y[index_of_max_element]=1
    return Y
```

Execute the program

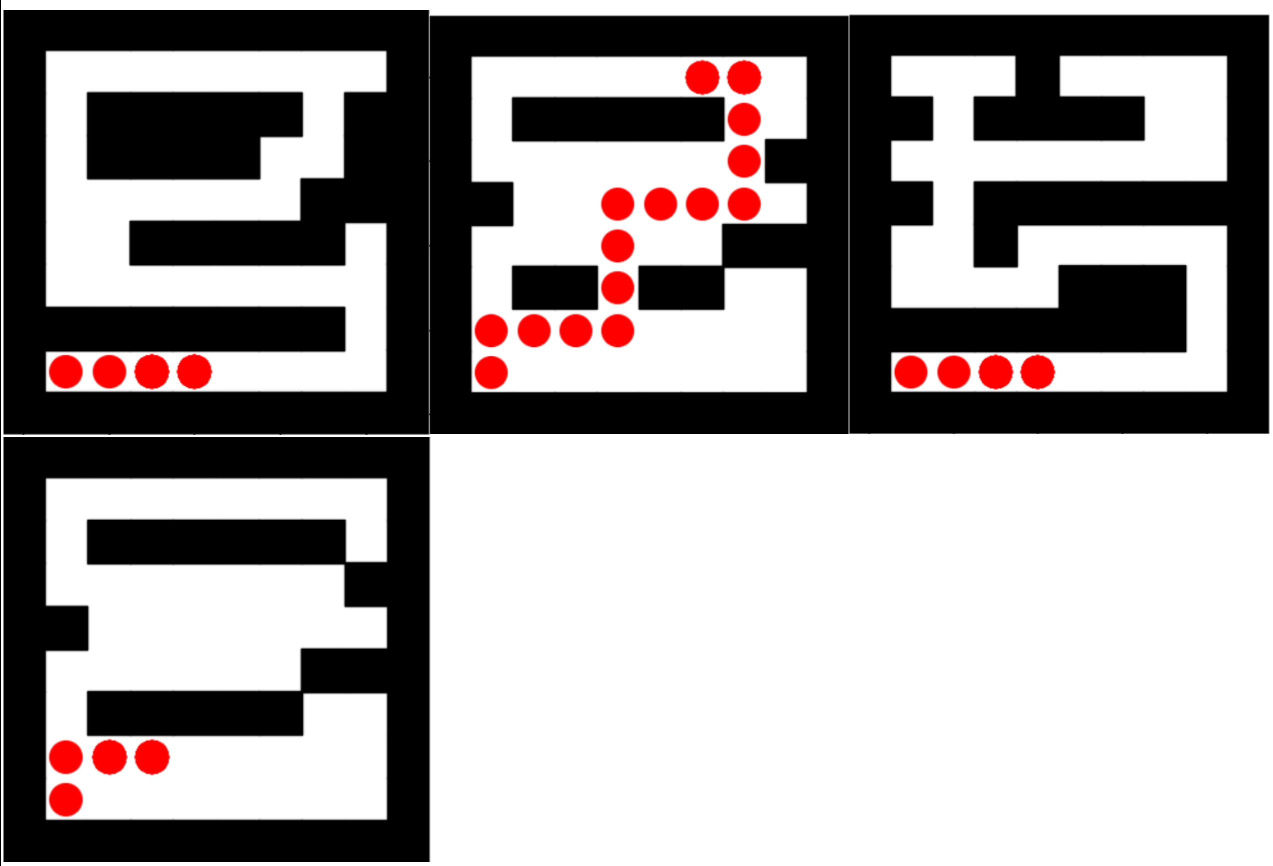
```
trained_parameters=L_layer_model(Xx[2:], Yy[2:], network_architecture,RNN_layer, learning_rate = 0.0075, num_iterations = 200, print_cost=True)

for maps in mapArray:
   testMapPlotValues(initialPosition,trained_parameters,network_architecture,maps)
plot.show()
```

RESULTS

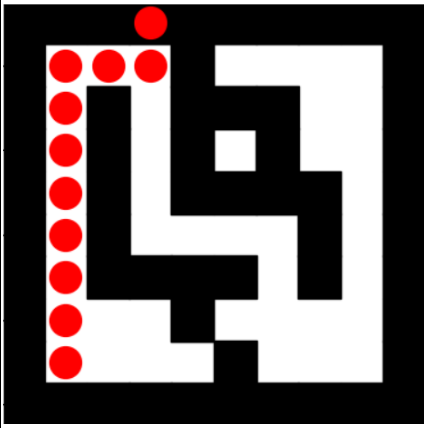
Successfully achieved test results



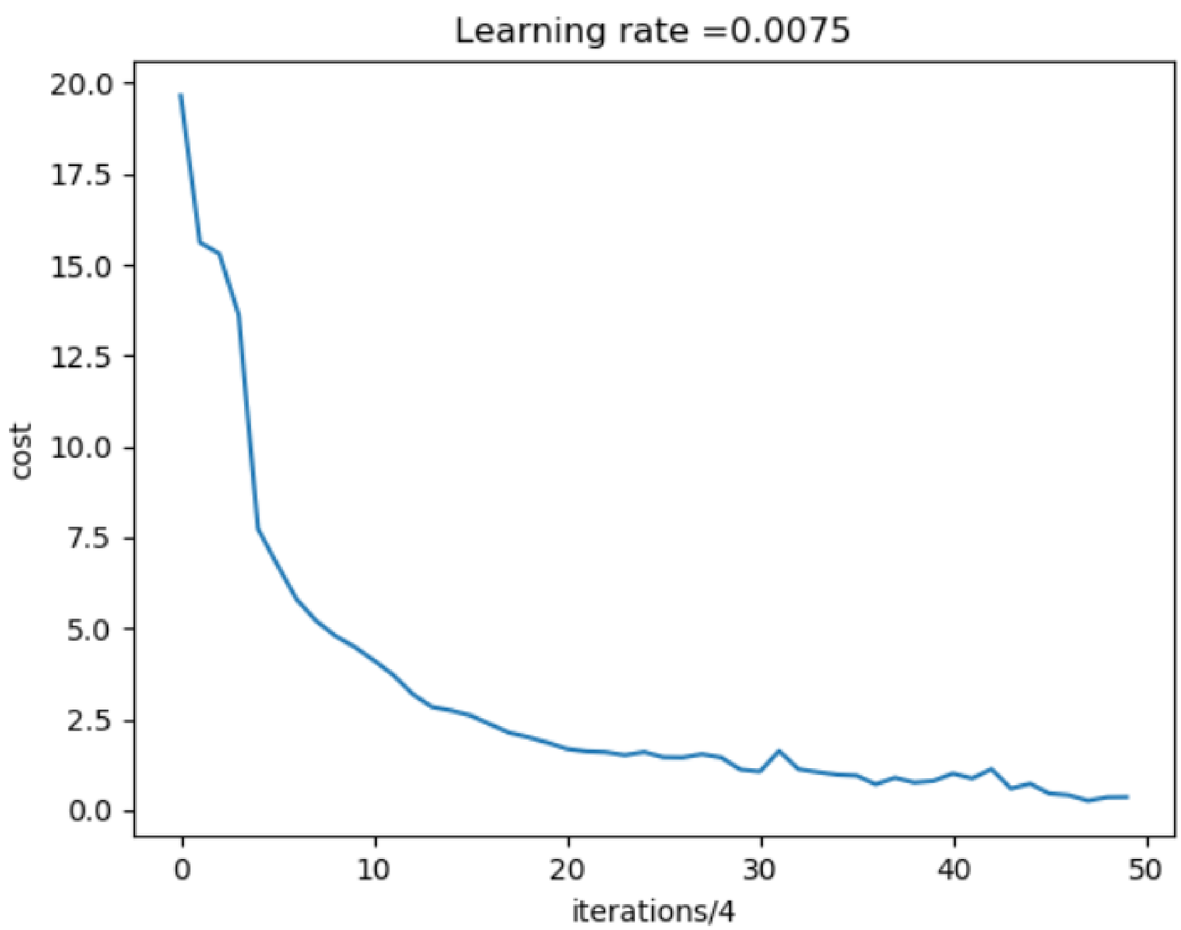
Ended with looped decision



Ended with collision



Cost values were reducing through the 200 iterations. Accuracy for the training set itself was 60% for 11 maps. Results are more than satisfied with the such a small traing set. If we increase the traing set, the Accuracy will be more higher.



REFERENCES

[1] R. A. i. M. R. D. A. T. (. I. 9.-9.-3.-9.-7. I. A. f. h.-. Valeri Kroumov and Jianli Yu (2011). Neural Networks Based Path Planning and Navigation of Mobile Robots.

[2] M.P.Sivaram Kumar, S.Rajasekaran, "A Neural Network based Path Planning Algorithm for Extinguishing Forest Fires," *IJCSI International Journal of Computer Science,* vol. 9, no. 2, pp. 563-568, March 2012.

[3] Elman, Jeffrey L. (1990). "Finding Structure in Time". Cognitive Science. 14 (2): 179–211.