MCOMD2AIC Artificial Intelligence Computing Assessment 2

Nathan Shuttleworth

Contents

[Introduction 2](#_Toc71377647)

[Solution 1 3](#_Toc71377648)

[Solution 2 6](#_Toc71377649)

[Comparison of Solutions 7](#_Toc71377650)

[Fairness of Testing 7](#_Toc71377651)

[Time - Based Criteria 8](#_Toc71377652)

[Compilation times – epoch 8](#_Toc71377653)

[Speed of solution 8](#_Toc71377654)

[Training time 9](#_Toc71377655)

[Flexibility/ Adaptability 9](#_Toc71377656)

[System Resources used 9](#_Toc71377657)

[Appendices 10](#_Toc71377658)

[Code for Solution 1 10](#_Toc71377659)

[References 12](#_Toc71377660)

# Introduction

The purpose of this investigation is to develop two separate solutions to help a robot navigate a maze within an allotted amount of time. The robot itself has 3 sensors, 1 that measures the distance between itself and an obstacle in front of it, and 2 more that measure whether they are touched or not. I handled this investigation by developing the two solutions independently, and then comparing them to see which was better regarding several factors, including but not limited to: Time taken, resources used and flexibility.

# Solution 1

The first solution to this investigation involves manually finding a set of weights to help the robot navigate around the maze, and for it to get round at least a quarter of the maze within 10 minutes. I intended this solution to fulfil the requirements of the ‘intermediate solution’. To do this, I created an excel spreadsheet and worked out what the robot should do with each combination of sensor, as well as the output combination that was already coded into the solution:

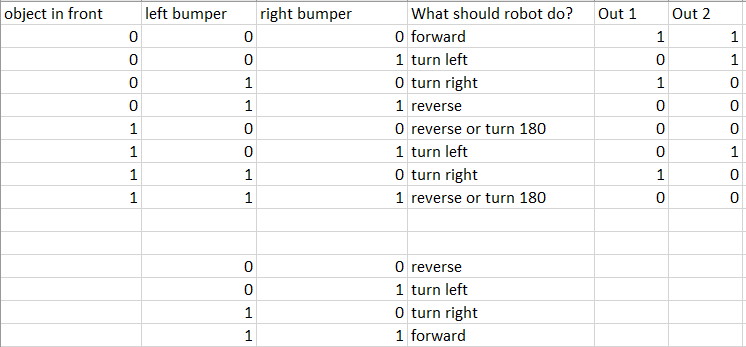


Figure 1- Table for sensors and decisions

What I next did is put together a truth table with the X values and the outputs I was trying to achieve again using excel. The values of X1, X2 and X3 are the same as the object in front, left bumper, and right bumper values from Figure 1 respectively.

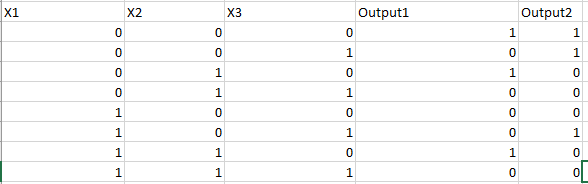


Figure 2- Truth table for X and output values

Now that these values were all clearly laid out, I used a neuron to find the weights that would allow this to work with the robot. I created it in Excel using guidance from ‘Building a simple neural network in EXCEL (Turner, S.)’, as well as using the McCulloch-Pitts model (Turner, S.). 2 sets of weights need to be discovered, since there are 2 outputs to find. Once the network had been set up, I started plugging in numbers to find weights and biases so that the outputs would become equal to the desired outputs. The full neuron looks like this:

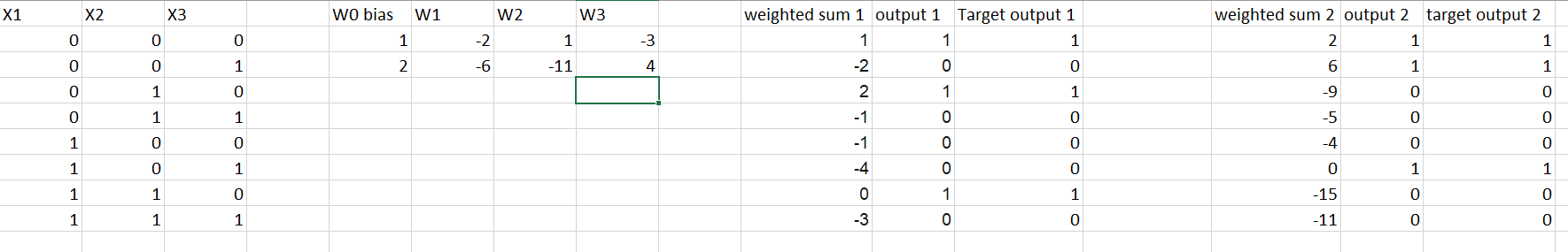


Figure 3- Full neuron

The 2 sets of weights I ended up finding that worked were:

|  |  |  |
| --- | --- | --- |
| **Weight\Set** | **Set 1** | **Set 2** |
| W0(bias) | 1 | 2 |
| W1 | -2 | -6 |
| W2 | 1 | -11 |
| W3 | -3 | 4 |

x

I then implemented these weights into the existing code, changing the second block to this:

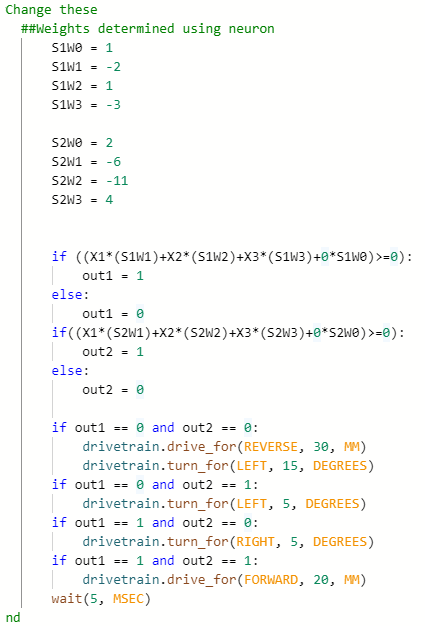


Figure 4 - 2nd block of code

Notice how the ‘if’ statements are implementations of the McCulloch-Pitts equation. After trying to run this, the robot successfully navigated a small portion of the maze, however it would get trapped in spaces and would continuously move in an anti-clockwise circle, never turning right after reversing. I fixed this by changing the robot’s behaviours for when all of the sensors are getting an input, both out1 and out 2 are 0. The angle at which the robot turns was changed from 15 degrees to 110. This makes it so that the robot primarily does large left turns, and then smaller right turns to correct itself when it hits a wall at an angle. With this change made, the robot can now navigate at least a half of the maze within 10 minutes.

The final code for the robot can be seen in appendix 1.

# Solution 2

For the second solution, I wanted to the robot to be able to teach itself how to find a way around the maze, it would learn the weights to navigate the maze by itself as opposed to me calculating them. This should theoretically allow the robot to fully move around the maze, instead of getting stuck. I aimed for this solution to fulfil the ‘Advanced Solution’ as detailed in the brief.

Before starting to create anything, I looked at the starter code and thought about how the weights could be ‘learned’ iteratively. In this solution, there are several sets of weights. Due to the way backpropagation works, I could not use the same 2 arrays of desired outputs that I had used in the previous solution. Insteatd, I created an array of desried outputs for each direction the robot could move in (forwards, reverse, left, right), and also 2 arrays of weights for each direction as well. I defined these as global arrays at the beginning of the code.

The weights are initially assigned randomly in a separate function that runs only once. They are generated in the range -1<=x<=1.

I based my code off of a version of a neurone in python (Turner, S.). Firstly, the input values are used to work out which direction the robot shou;ld be travelling in, in accordance with the truth table for directions and responses. Next, the net values are calculated, and then if the net values are above 0, the outputs are set to 1, otherwise they are set to 0.

After this, the backpropagation takes place. The sum squared errros for each weight is calculated in comparison to the desired outputs, and then delta values are calculated by multiplying this by the learning coefficient. After some tweaking, I found that the optimum learning coefficeitne for this system was 400. The weights are then updated by adding the delta values to their current value.

Finally, the inputs are used to work out what the robot needs ot do in terms of moving. This has not been significantly changed from the base code.

In order to give the system some time for the weights to be trained, I have set it so that the process must run 20,000 times first before the robot is given the option to move. Whilst this does take up some of the time needed, I believe that it is necessary to create a more efficient solution. The training data is in the section after the nets and outputs are calculated. It works by changing the output values and the direction variable to be explicitly the ones from the truth table, the robot is essentially ‘fooled’ into thinking it is moving in a certain direction. It then runs many iterations with this training data, so as to get the robot to improve its weights for each of the directions before it actually starts to move around the maze.

The final code for this solution can be seen in Appendix 2.

# Comparison of Solutions

## Fairness of Testing

To accurately compare the two solutions, I first needed to define certain things to ensure that the testing was fair. In the brief, one of the criteria for the different levels of solutions is the distance the robot travels in 10 minutes, whether it is ¼, ½ or completing the maze. I interpreted the first two as the robot’s tracks covering ¼ or ½ of the area of the maze, and the last one as the robot getting from the start to the end of the maze.

The brief also talks about comparing the system resources used for each solution. Since for the first one different software was used, and the fact that all of the code being used is very lightweight, it will be difficult to accurately measure the use of resources. A couple of things I have noticed is that my second solution, which is more complex, takes longer to load. As well as this, for the advanced program the clock runs slower than real time. This is likely due to it being more complex.

One of the ways I will try to measure the use of resources is to have no other (non-essential) programs open on my computer except for the Chrome tab that will be open running the Vex robotics tab and code. I will then, using the stopwatch on my phone, time the amount of time taken for the code to load up upon initially pressing start. I will also use the in-built timer alongside the one on my phone to time how long it takes for the robot to navigate the maze. This will give me both 2 ways of measuring the epoch, as well as seeing the difference in times will give me an indication as to which code uses fewer resources and is more light weight. I will also consider what programs have been used to get to that point with the solution, for instance if external excel spreadsheets had to be used.

## Time - Based Criteria

### Compilation times – epoch

#### Solution 1

The compilation time for this solution was 1.98 seconds.

#### Solution 2

The compilation time for this solution was 2.12 seconds.

#### Comparison

Whilst the second solution took slightly longer to load in, the difference is too low to be an indicator of performance.

### Speed of solution

#### Solution 1

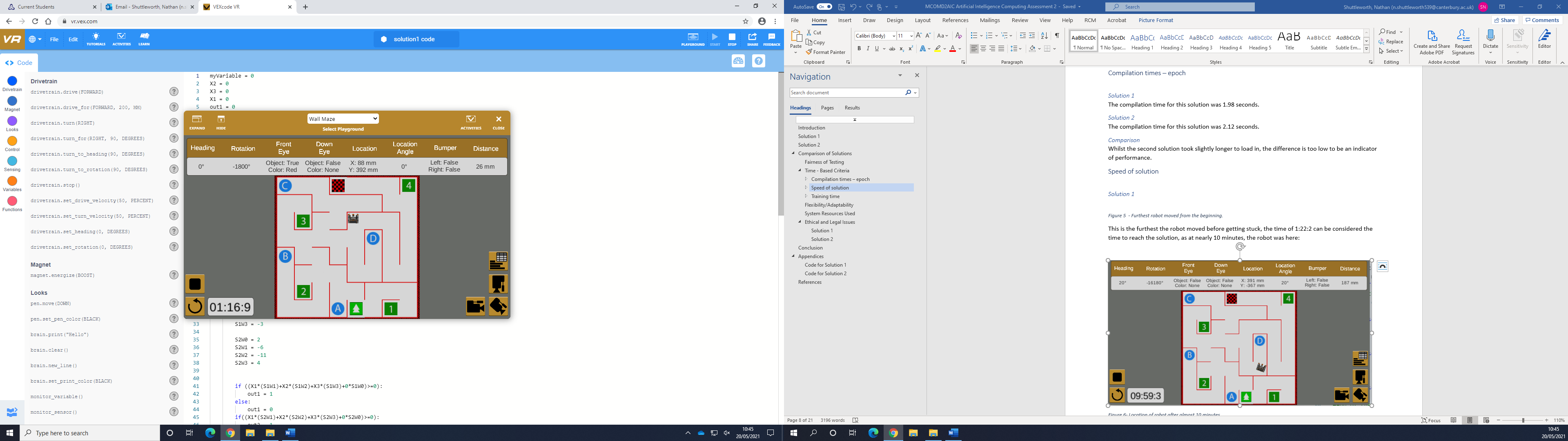


Figure 5 - Furthest robot moved from the beginning.

This is the furthest the robot moved before getting stuck, the time of 1:16:9 can be considered the time to reach the solution, as at nearly 10 minutes, the robot was here:



Figure 6- Location of robot after almost 10 minutes

For the first solution, the difference between time shown on Vex and the time I recorded was negligible, the only reason for any difference would be me starting my timer slightly out of time.

#### Solution 2

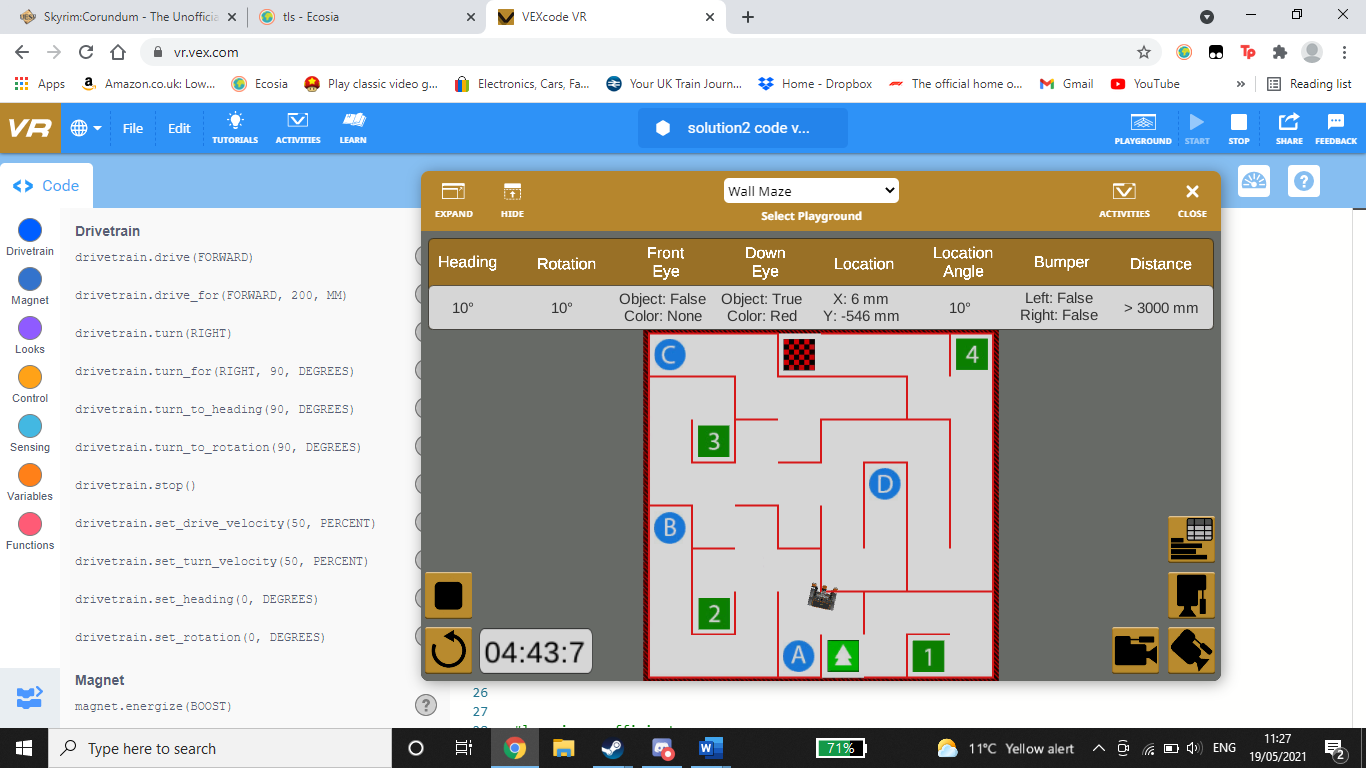


Figure 7 - Furthest location reached by robot

The robot unfortunately did not move as far as I had hoped, it became stuck inside the wall after 4:43 minutes, and remianed there.

#### Comparison

In regard to distance travelled, solution 1 was more successful. However, if the second one had been given more time to train (more than 10 minutes) I believe that it could have worked better and not caused itself to become stuck in walls, it would respond better to sensor inputs and would have avoided this.

### Training time

#### Solution 1

Since Solution 1 was developed by me using a neuron created in a spreadsheet, it was not trained in the traditional way. I came across the weights by constantly trying different combinations of weights until I got the correct sets. This took about 10 minutes for both neurons, since 2 sets of weights had to be found.

#### Solution 2

Solution 2 trains itself, using training data to get the weights to be appropriate enough for it to start moving, and then after a certain number of iterations the robot will begin to move. It will then use the data it receives from its sensors to help improve the calculated weights. The dedicated training time ended up being about 1:14.5 minutes as recorded by vex, and 1:15.71 minutes as recorded by me using a stopwatch.

#### Comparison

Whilst solution 2 spent less time dedicated to training, it is ultimately hard to give a direct comparison since if a machine were calculating the weights in the same process I did for solution 1, they would have been found much faster. In conclusion, I deem solution 2 to be better in this regard since it can continue training forever, constantly backpropagating to find the best weight values, whereas solution 1 is stuck with the values that I found.

## Flexibility/Adaptability

Both of these solutions would be equally adaptable to other similar mazes, just with a different layout. That is because the weights that are being learned/changed relate to how the robot process data from its sensors, whereas learning how to actually navigate the maze. On the one hand, this is good because other mazes of a similar design (square walls) will be easily navigated by the robot. However, if the robot where to be put in the maze with round/circular walls, it would struggle since it has been optimised for moving at right angles. This would be the case for both solutions.

## System Resources Used

The difference between the time recorded by VEX for solution 1 and the time I recorded for it using my stopwatch was negligible. The difference that was there can be put down to human error (me not pressing start on both at exactly the same time). However, for solution 2, the difference in time grew as the program ran, being an almost 6 second difference by the end. To me, this says that the code for solution 2 used more resources since it caused the VEX application to run slower, which is reflected in its timer not being accurate.

## Ethical and Legal Issues

### Solution 1

There are not many ethical and legal issues with regards to this solution. There is not a high amount of resources or power being used, and there are only two pieces of software being used (Microsoft Excel and VEX). However, if this were to be scaled up more issues would become apparent. Professional licenses for Excel would have to be acquired, and the strain of a large number of users all using the VEX website at once could cause it to crash. Also, using two pieces of software instead of one is likely to use more power and be worse for the environment.

### Solution 2

As with the last solution, there are not many ethical and legal issues involved. However, if this solution were to be scaled up, it would likely that the VEX software would use more resources, since its code is more complex and uses more iterations and does many more operations, since it is backpropagating its weights.

# Conclusion

In conclusion, the best Solution I developed in terms of meeting the task requirements was solution 1. This is because it made it further around the maze and did not use as many system resources. However, solution 2 was more complex and so if it had the chance to be trained for longer (and I had more time to fine tune its training), it would have been the better solution as it would have been able to reach the end of the maze. Both solutions are equally adaptable and carry much the same legal and ethical issues.

In all, this investigation has shown me how backpropagation can be an effective tool for machine learning, however it requires a lot of fiddling to make it an effective solution. Whilst manually finding the weights was easier, it does not have nearly as much power as backpropagation, and on a larger scale would be very inefficient.

# Appendices

## Code for Solution 1

myVariable = 0

X2 = 0

X3 = 0

X1 = 0

out1 = 0

out2 = 0

def when\_started1():

global myVariable, X2, X3, X1, out1, out2

while True:

#First block that should stay the same

if distance.get\_distance(MM) < 30:

X1 = 1

else:

X1 = 0

if left\_bumper.pressed():

X2 = 1

else:

X2 = 0

if right\_bumper.pressed():

X3 = 1

else:

X3 = 0

#end of the first block that should stay the same

##Change these

##Weights determined using neuron

S1W0 = 1

S1W1 = -2

S1W2 = 1

S1W3 = -3

S2W0 = 2

S2W1 = -6

S2W2 = -11

S2W3 = 4

if ((X1\*(S1W1)+X2\*(S1W2)+X3\*(S1W3)+0\*S1W0)>=0):

out1 = 1

else:

out1 = 0

if((X1\*(S2W1)+X2\*(S2W2)+X3\*(S2W3)+0\*S2W0)>=0):

out2 = 1

else:

out2 = 0

if out1 == 0 and out2 == 0:

drivetrain.drive\_for(REVERSE, 30, MM)

drivetrain.turn\_for(LEFT, 110, DEGREES)

if out1 == 0 and out2 == 1:

drivetrain.turn\_for(LEFT, 5, DEGREES)

if out1 == 1 and out2 == 0:

drivetrain.turn\_for(RIGHT, 5, DEGREES)

if out1 == 1 and out2 == 1:

drivetrain.drive\_for(FORWARD, 20, MM)

wait(5, MSEC)

#end

## Code for Solution 2

import random

myVariable = 0

X2 = 0

X3 = 0

X1 = 0

out1 = 0

out2 = 0

#weight arrays. 2 sets of weights for each direction

Forwardw1 = [0,0,0,0]

Forwardw2 = [0,0,0,0]

Leftw1 = [0,0,0,0]

Leftw2 = [0,0,0,0]

Rightw1 = [0,0,0,0]

Rightw2 = [0,0,0,0]

Reversew1 = [0,0,0,0]

Reversew2 = [0,0,0,0]

#desired outputs

desoutForward = [1,1]

desoutLeft = [0,1]

desoutRight = [1,0]

desoutReverse = [0,0]

#learning coefficient

lc = 400

#robot direction:

direction = "forward"

def initialWeightSetup():

global Forwardw1, Forwardw2, Leftw1, Leftw2, Rightw1, Rightw2, Reversew1, Reversew2

#randomly determine initial weights

for i in range(4):

Forwardw1[i] = random.randrange(-1,1)

Forwardw2[i] = random.randrange(-1,1)

Leftw1[i] = random.randrange(-1,1)

Leftw2[i] = random.randrange(-1,1)

Rightw1[i] = random.randrange(-1,1)

Rightw2[i] = random.randrange(-1,1)

Reversew1[i] = random.randrange(-1,1)

Reversew2[i] = random.randrange(-1,1)

def when\_started1():

global myVariable, X2, X3, X1, out1, out2

while True:

#First block that should stay the same

if distance.get\_distance(MM) < 30:

X1 = 1

else:

X1 = 0

if left\_bumper.pressed():

X2 = 1

else:

X2 = 0

if right\_bumper.pressed():

X3 = 1

else:

X3 = 0

#end of the first block that should stay the same

##Change these

## Determine direction robot is intending to travel

if(X1==0):

if(X2==0):

if(X3==0):

direction = "forward"

else:

direction = "left"

elif(X3==0):

direction = "right"

else:

direction = "reverse"

elif(X2==0):

if(X3==0):

direction = "reverse"

else:

direction = "left"

elif(X3==0):

direction = "right"

else:

direction = "reverse"

if(direction == "forward"):

net1=Forwardw1[1]\*X1+Forwardw1[2]\*X2+Forwardw1[3]\*X3+Forwardw1[0]

net2=Forwardw2[1]\*X1+Forwardw2[2]\*X2+Forwardw2[3]\*X3+Forwardw2[0]

elif(direction == "left"):

net1=Leftw1[1]\*X1+Leftw1[2]\*X2+Leftw1[3]\*X3+Leftw1[0]

net2=Leftw2[1]\*X1+Leftw2[2]\*X2+Leftw2[3]\*X3+Leftw2[0]

elif(direction == "right"):

net1=Rightw1[1]\*X1+Rightw1[2]\*X2+Rightw1[3]\*X3+Rightw1[0]

net2=Rightw2[1]\*X1+Rightw2[2]\*X2+Rightw2[3]\*X3+Rightw2[0]

elif(direction == "reverse"):

net1=Reversew1[1]\*X1+Reversew1[2]\*X2+Reversew1[3]\*X3+Reversew1[0]

net2=Reversew2[1]\*X1+Reversew2[2]\*X2+Reversew2[3]\*X3+Reversew2[0]

if net1 >=0:

out1 = 1

else:

out1 = 0

if net2 >=0:

out2 = 1

else:

out2 = 0

# this is the training data.

if(myVariable < 15000):

out1 = 1

out2 = 0

direction = "right"

if(myVariable < 10000):

out1 = 0

out2 = 1

direction = "left"

if(myVariable < 5000):

out1 = 1

out2 = 1

direction = "forward"

#for each desired output calculate the error

for loop in range (2):

ForwardError1 = 0.5\*((desoutForward[loop] - out1)^2)

ForwardError2 = 0.5\*((desoutForward[loop] - out2)^2)

LeftError1 = 0.5\*((desoutLeft[loop] - out1)^2)

LeftError2 = 0.5\*((desoutLeft[loop] - out2)^2)

RightError1 = 0.5\*((desoutLeft[loop] - out1)^2)

RightError2 = 0.5\*((desoutRight[loop] - out2)^2)

ReverseError1 = 0.5\*((desoutReverse[loop] - out1)^2)

ReverseError2 = 0.5\*((desoutReverse[loop] - out2)^2)

#use the error to calculate changes in weights and change weights

if(direction == "forward"):

deltaS1W0=lc\*ForwardError1

Forwardw1[0]=Forwardw1[0]+deltaS1W0

deltaS1W1=lc\*ForwardError1\*X1

Forwardw1[1]=Forwardw1[1]+deltaS1W1

deltaS1W2=lc\*ForwardError1\*X2

Forwardw1[2]=Forwardw1[2]+deltaS1W2

deltaS1W3=lc\*ForwardError1\*X3

Forwardw1[3]=Forwardw1[3]+deltaS1W3

deltaS2W0=lc\*ForwardError2

Forwardw2[0]=Forwardw2[0]+deltaS2W0

deltaS2W1=lc\*ForwardError2\*X1

Forwardw2[1]=Forwardw2[1]+deltaS2W1

deltaS2W2=lc\*ForwardError2\*X2

Forwardw2[2]=Forwardw2[2]+deltaS2W2

deltaS2W3=lc\*ForwardError2\*X3

Forwardw2[3]=Forwardw2[3]+deltaS2W3

elif(direction == "left"):

deltaS1W0=lc\*LeftError1

Leftw1[0]=Leftw1[0]+deltaS1W0

deltaS1W1=lc\*LeftError1\*X1

Leftw1[1]=Leftw1[1]+deltaS1W1

deltaS1W2=lc\*LeftError1\*X2

Leftw1[2]=Leftw1[2]+deltaS1W2

deltaS1W3=lc\*LeftError1\*X3

Leftw1[3]=Leftw1[3]+deltaS1W3

deltaS2W0 = lc\*LeftError2

Leftw2[0] = Leftw2[0]+deltaS2W0

deltaS2W1=lc\*LeftError2\*X1

Leftw2[1]=Leftw2[1]+deltaS2W1

deltaS2W2=lc\*LeftError2\*X2

Leftw2[2]=Leftw2[2]+deltaS2W2

deltaS2W3=lc\*LeftError2\*X3

Leftw2[3]=Leftw2[3]+deltaS2W3

elif(direction == "right"):

deltaS1W0=lc\*RightError1

Rightw1[0]=Rightw1[0]+deltaS1W0

deltaS1W1=lc\*RightError1\*X1

Rightw1[1]=Rightw1[1]+deltaS1W1

deltaS1W2=lc\*RightError1\*X2

Rightw1[2]=Rightw1[2]+deltaS1W2

deltaS1W3=lc\*RightError1\*X3

Rightw1[3]=Rightw1[3]+deltaS1W3

deltaS2W0 = lc\*RightError2

Rightw2[0] = Rightw2[0]+deltaS2W0

deltaS2W1=lc\*RightError2\*X1

Rightw2[1]=Rightw2[1]+deltaS2W1

deltaS2W2=lc\*RightError2\*X2

Rightw2[2]=Rightw2[2]+deltaS2W2

deltaS2W3=lc\*RightError2\*X3

Rightw2[3]=Rightw2[3]+deltaS2W3

elif(direction == "reverse"):

deltaS1W0=lc\*ReverseError1

Reversew1[0]=Reversew1[0]+deltaS1W0

deltaS1W1=lc\*ReverseError1\*X1

Reversew1[1]=Reversew1[1]+deltaS1W1

deltaS1W2=lc\*ReverseError1\*X2

Reversew1[2]=Reversew1[2]+deltaS1W2

deltaS1W3=lc\*ReverseError1\*X3

Reversew1[3]=Reversew1[3]+deltaS1W3

deltaS2W0 = lc\*ReverseError2

Reversew2[0] = Reversew2[0]+deltaS2W0

deltaS2W1=lc\*ReverseError2\*X1

Reversew2[1]=Reversew2[1]+deltaS2W1

deltaS2W2=lc\*ReverseError2\*X2

Reversew2[2]=Reversew2[2]+deltaS2W2

deltaS2W3=lc\*ReverseError2\*X3

Reversew2[3]=Reversew2[3]+deltaS2W3

if(myVariable > 20000):

if out1 == 0 and out2 == 0:

drivetrain.drive\_for(REVERSE, 30, MM)

drivetrain.turn\_for(LEFT, 15, DEGREES)

if out1 == 0 and out2 == 1:

drivetrain.turn\_for(LEFT, 5, DEGREES)

if out1 == 1 and out2 == 0:

drivetrain.turn\_for(RIGHT, 5, DEGREES)

if out1 == 1 and out2 == 1:

drivetrain.drive\_for(FORWARD, 20, MM)

myVariable = myVariable + 1

# for i in range(4):

# brain.print('\n' , [i+ 1] , Forwardw1[i] , ', ' , Leftw1[i], ', ' , Rightw1[i], ', ' , Reversew1[i])

wait(5, MSEC)

#end

vr\_thread(initialWeightSetup())

vr\_thread(when\_started1())

# References

Scott Turner (2018) ‘Building a simple neural network in EXCEL’. Available at: https://youtu.be/yieX99Y\_Phg (Accessed: 27 April 2021).

Scott Turner (2021) ‘Introduction to Neural Networks 1’. *Artificial Intelligence Computing (MCOMD2AIC).* Available at: <https://learn.canterbury.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_14250_1&content_id=_2813242_1> (Accessed 28 April 2021)

Scott Turner (2021) ‘Python neurone (no official name)’. *Activity 9.1 (MCOMD2AIC).* Available at: <https://learn.canterbury.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_14250_1&content_id=_2843350_1> (Accessed 28 April 2021) Used the first bit of code, not the second.