Autonomous Decision-Making in Robots Using Neural Network Trees

Marwan Jabbour Dr. Joseph Vybihal
McGill University, Prometheus Lab
August 2021 – Comp 400

Abstract: *Robot decision-making operates in three main stages: interpreting the physical environment (sensor detection), processing that input (decision-making), and acting accordingly (motor commands). This paper focuses on simulating that feedback loop with a primary focus on the first and second stages. Previous research on this topic has explored a tree-like neural network with 6 sensor inputs and 7 neural networks; no explicit comparison to a non-tree like neural network has been made, and no path traversal simulations have been conducted. This paper uses data from 9 sensor inputs to create a new tree neural network with 11 individual neural networks with renewed focus on surface and direction classification. It makes explicit comparisons in performance to a non-tree like neural network and runs through simulations to see whether a robot can truly make sense of its environment in trying to reach its destination. The results reveal that, compared to a non-tree like neural network structure, the tree neural network yields higher accuracy rates, more customizability, and allows for new neural networks to be added more easily. However, this neural network tree guides the robot into an infinite loop in nearly half the simulations, and so the neural networks coupled with a path-finding algorithm generates the safest results.*

I. Acknowledgement

I wish to acknowledge the ongoing help provided by Dr. Joseph Vybihal for his crucial suggestions during the planning and development of this project. I would also like to credit Shaodi Sun and Hank Zhang for their previous work on the tree neural network project. I have also referenced the official code documentation for the Java Neuroph framework along with other scholarly articles.

II. Introduction & Motivation

a) Problem

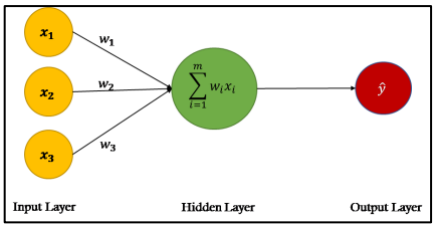
The Prometheus project attempts to create an autonomous robot that uses sensor data to perform actions in both known and unknown situations. In particular, the robot is placed on a square grid that corresponds to the actual floor. The grid is composed of uniform blocks of equal distance. The robot starts at a charging station and its ultimate objective is to reach a destination block. However, the robot has no knowledge of where the obstacles it might encounter on its way are. It is equipped with some ultrasonic sensors along with sensors for capacitance, voltage, etc. that it should make sense of in trying to reach the known destination without sacrificing its battery or hitting obstacles midway. The robot must make decisions along the way that will safely guide it to its destination.

The first level of the AI comprises of a neural network tree that accepts sensor inputs and feeds the corresponding outputs to the robot; it forms the basis of the decision-making process. The neural network must demonstrate correctness in comprehension of sensor data and performance of ensuing actions when many sensors are at play. Individual neural networks use data to make a prediction that can help guide the robot (e.g. Is the surface rough or smooth? Are there any obstacles in front of the robot, on its left, etc.)? Together, these neural networks will form a neural network tree that can perform higher level decisions (e.g. Should the robot be charged right now? Is the object that the robot is supposed to push movable?)

b) Questions

In trying to design the neural network tree, this paper answers several questions about the robot's feedback loop. *How should the neural network tree be designed? How does this tree compare, in accuracy and time, to a non-tree like structure? What are the advantages/drawbacks of each structure? Which structure would be better in allowing for future higher-level neural networks? How effective is this tree neural network in guiding a robot to its destination? Does the size of the grid matter? At what grid size can the robot no longer successfully reach its destination? Would a typical path-traversal algorithm (e.g. depth-first-search) be more effective in guiding the robot? How can a tree neural network work with such a path-traversal algorithm?*

III. Literature/Background



A preliminary appreciation of individual neural networks is crucial in our discussion of tree neural networks. The simplest neural network comprises of an individual, *building-block* neuron as shown in the image above. The input layer comprises of features from the outside world (e.g. csv file) and, hence, necessitates no computation. The neuron accepts the input x_1, x_2, x_3 and passes it on to the hidden layer. The hidden layer, while not connected to the world, performs computations on the features fed by the input layer, here through an activation function. The activation function $f\left(\sum_{i=1}^{3} w_i x_i\right)$ uses the individual weights w_1, w_2, w_3 (which are either excitatory (w>0) or inhibitory (w<0) connections), performs a computation, and sends information to the output layer (Activation Functions in Neural Networks, 2020).

There are several variants of the activation function: sigmoid ($\frac{1}{1+e^{-x}}$), tanh ($\frac{2}{1+e^{-2x}}$-1), RELU (max (0, x)), etc. each of which has a different value range. Essentially, they decide whether the neuron should be activated or not through a process called backward propagation. The loss function reveals how far the neural network's predictions are from the true labels, and backpropagation allows us to determine the gradient of this loss function in terms of the weights w_1, w_2, w_3 (Wood, 2020). In simpler terms, these weights change in the direction of the predictions of the neural network when it is trained.

Multilayer networks, in principle, implement linear discriminants in non-linearly mapped inputs. Their ability to learn this non-linearity stems from the introduction of hidden layers with neurons that feed other neurons rather than the output directly (Schmidt). These hidden layers amplify the separation capacity of the individual neuron by adding non-linearity and transforming inputs into possibly useful outputs for the next hidden layers. This adds more complexity to the model by capturing new, previously unnoticed, relationships between the features of the input.

The training process for multilayer networks also follows backward propagation. The error in the output layer is propagated back to the hidden layers in accordance with the weights. This process repeats over and over through cycles through the training data, known as epochs (Schmidt).

This paper investigates a new form of neural network structure called a tree neural network (treeNN). A tree neural network, not to be confused with a decision tree, operates through individual neural networks being fed into each other, much like individual inputs feed neurons. Whereas an individual multilayer network is able to answer questions directly based on sensor data, a tree neural network answers higher level questions that individual networks might not be able to. A tree neural network comprises of first-

level neural networks which accept input from sensors, perform computation, and feed the output into second-level neural networks. The second-level neural networks, potentially with input from basic sensors, feed into third-level neural networks, and so on.

In theory, this tree neural network can offer not only higher recognition power, but also finer generalization ability. For example, consider a neural network that attempts to predict if a robot can reach a certain destination based on its surface, battery levels, and detection of obstacles in its vicinity. Assume the robot can only reach its destination if its surface is smooth. However, the robot has no sensors to actually detect the surface it is on, and so must use other sensors such as an accelerometer and speedometer. In this case, we only care about the texture of the floor, not data from the accelerometer and speedometer, and so a separate neural network for this particular feature would make sense.

IV. Tools Used

For the machine learning component of this project, I employed the Java Neuroph Framework. Neuroph is a light-weight tool with an abundance of machine learning models and data preprocessing tools. In particular, I used the Multilayer Perceptron model for its similarity to a classic neural network. It is a feed-forward model with layers between the input and output layers. It is trained with backpropagation learning algorithm and is useful for classification, prediction, and approximation problems (Neuroph). For the simulations, I made of use of JPanel and JFrame for their simplicity and light-weight nature. I also used IntelliJ IDEA as my integrated development environment.

V. Methodology/Results

Steps

The project execution took the following steps:

Phase I:

1. Design the tree neural network (version 1)
2. Gather data for the first level neural networks, and implement them
3. Simulate data for the higher level neural networks, and implement them
4. Generate test cases and simulations for the individual neural networks

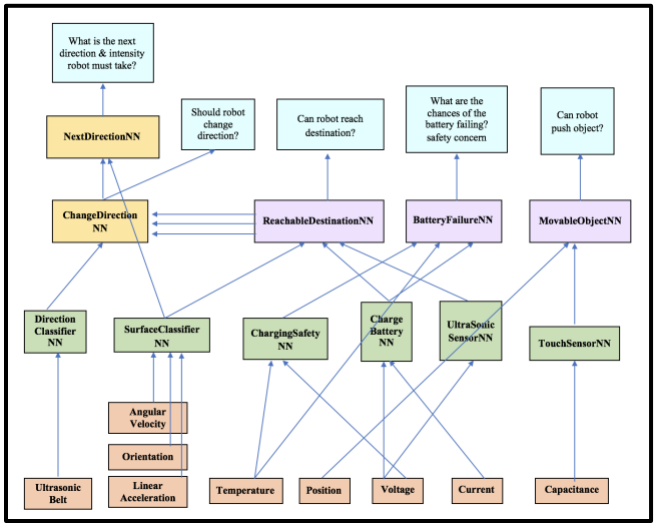
Phase II:

1. Design the non-tree like neural network (version 2)
2. Generate data for the non-tree like neural networks and implement them
3. Generate test cases for the non-tree like neural network
4. Compare version 1 and version 2

Phase III:

1. Design the DFS-based neural network (version 3)
2. Simulate the DFS neural network and version 1
3. Compare version 1 and version 3

a) Tree Neural Network: Version 1



The bottom row consists of the raw sensory inputs: ultrasonic belt, linear acceleration, orientation, angular velocity, temperature, position, voltage, current, and capacitance. These raw sensory inputs feed data into the green first-level neural networks: DirectionClassifierNN, SurfaceClassifierNN, ChargingSafetyNN, ChargeBatteryNN, UltraSonicSensorNN, and TouchSensorNN. The green neural networks feed the second-level neural networks: ReachableDestinationNN, BatteryFailureNN, and MovableObjectNN. In turn, ReachableDestinationNN and DirectionClassifierNN feed the third-level neural network ChangeDirectionNN. Along with SurfaceClassifierNN, ChangeDirectionNN feeds the fourth-level neural network NextDirectionNN. Here is a detailed summary of the individual neural networks, their input, and what they predict.

Neural Network	Level	Input	Output
DirectionClassifierNN	Level 1	Ultrasonic belt	Is robot moving forward or turning right or turning left?
SurfaceClassifierNN	Level 1	Angular velocity Orientation Linear Acceleration	Is the surface the robot is moving on rough or smooth?
ChargingSafetyNN	Level 1	Temperature Voltage	Is it safe to charge the battery?
ChargeBatteryNN	Level 1	Voltage Current	Should the robot be charged?
UltraSonicSensorNN	Level 1	Voltage	Is the robot close to any walls/obstacles?
TouchSensorNN	Level 1	Capacitance	How far are the 2 whiskers – how hard is robot pushing?
ReachableDestinationNN	Level 2	SurfaceClassifierNN ChargeBatteryNN UltraSonicSensorNN	Can the robot reach its destination?
BatteryFailureNN	Level 2	ChargingSafetyNN ChargeBatteryNN Temperature	Will the battery likely fail at the current time?

MovableObjectNN	Level 2	TouchSensorNN Position (x2)	Is the object, if any, movable by the robot?
ChangeDirectionNN	Level 3	DirectionClassifierNN ReachableDestinationNN (x3)	Should the robot move forward, turn right, turn left, or turn 180 degrees?
NextDirectionNN	Level 4	ChangeDirectionNN SurfaceClassifierNN	Should the robot move forward, take a slight right turn, sharp right turn, etc.?

DirectionClassifierNN

The data for the DirectionClassifierNN has been obtained from the University of California, Irvine Machine Learning Repository. The data was collected as the SCITOS G5 was navigated through a room following the wall in a clockwise direction. The repository offers a more complex dataset, with 24 sensors attached around the robot; however, I only make use of the simplified dataset with 4 sensors. The 4 sensors are assumed to be at 60 degrees from each other, arranged around the waist of the robot.

The classes predicted are: move-forward, right-turn, and left-turn. I choose to ignore the dataset's distinction of sharp and slight right turns, as they are not provided for left turns, and assume a uniform surface throughout the robot's course. Here is a snippet of the dataset.

SD_front	SD_left	SD_right	SD_back	Class
1.344	0.496	2.843	0.692	forward
0.753	0.457	2.323	0.442	right
1.332	2.171	1.537	2.166	left

Class	Percentage
right-turn	54%
move forward	40%
left-turn	6%

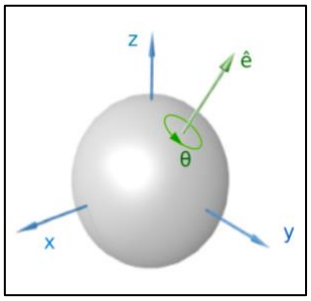
20 test cases have been generated for this neural network, the output of which is shown below.

SD_front	SD_left	SD_right	SD_back	Expected	Predicted	Outcome
1.687	0.445	2.332	0.429	right	right	true
0.798	0.663	2.469	1.242	right	right	true
1.567	0.459	1.82	0.554	right	right	true
0.79	0.762	2.789	1.172	right	right	true
0.761	0.49	1.754	1.145	right	right	true
0.769	0.783	1.564	0.668	right	right	true
0.811	0.435	1.423	0.394	right	right	true
2.0	0.515	1.957	1.689	forward	forward	true
0.831	0.761	1.441	0.769	right	right	true
0.88	0.708	1.25	0.513	right	right	true
0.846	0.446	1.925	1.339	right	right	true
0.967	0.946	2.725	1.302	left	right	false
1.54	0.452	1.839	1.025	right	forward	false
0.962	0.669	1.566	0.725	forward	forward	true
0.602	0.504	2.56	1.355	right	right	true
2.491	0.49	1.797	1.194	right	forward	false
0.823	0.789	1.319	2.603	right	right	true
0.808	0.536	1.693	1.299	forward	forward	true
1.252	0.597	1.567	0.924	right	right	true
0.763	0.648	3.02	1.624	right	right	true

Accuracy Rate: 17/20, 85%

SurfaceClassifierNN

The data for SurfaceClassifierNN has been obtained from the Department of Signal Processing at Tampere University where, in 2019, professors Heikki Huttunen and Francesco Lomio collected IMU sensor data by driving a small robot over various surfaces.



The 10 IMU sensor channels orientation (x, y, z, w), angular velocity (x, y, z), and linear acceleration (x, y, z) are used to describe how the robot is oriented as a quaternion. This quaternion is a vector of 4 elements that encodes spatial rotation and orientation in a 3D coordinate system.

The orientation channels refer to the angles (in radians), that the robot makes with the x, y, and z axes. Angular velocity refers to how quickly these angles are changing in relation to their axes, and linear acceleration refers to the distance traveled per unit of time. Since it is not possible to predict the surface without actually driving the robot on the surface, the dataset is structured per series and not per measurement. Specifically, there are 3809 series in the dataset, and 128 measurements per time series. Each of the 3809 series corresponds to one trial (and hence one floor type).

Feeding all 128 rows of data into the neural network is ineffective and costly; instead, feature engineering is used. In other words, knowledge of this dataset is used to construct explanatory variables, or features, that can be used to train the predictive model. Here I make use of purely statistical features. In the first round of feature engineering, I calculate the total, and in the second the mean, median, maximum, minimum, and standard deviation of all features. This results in (10+3) *5= 65 columns in the dataset.

The total of a feature is calculated by finding the square root of the sum of the individual features squared. For instance,

$$\text{total\_angular\_velocity} = \sqrt{\text{angular\_velocity\_X}^2 + \text{angular\_velocity\_Y}^2 + \text{angular\_velocity\_Z}^2}$$

Here is a snippet of the dataset (3809x65) after two rounds of feature engineering.

orientation_x mean	orientation_x median	orientation_x max	…	orientation_total std	Class
-0.758666	-0.758530	-0.75822	…	0.000042	fine concrete
-0.512057	-0.512035	-0.50944	…	0.000020	concrete

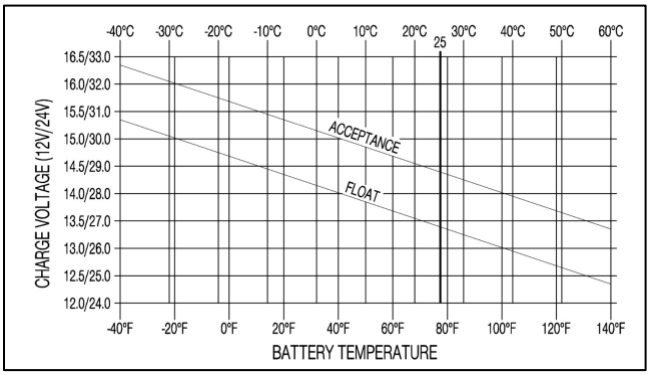
Class	Percentage
concrete	20.45%
soft_pvc	19.21%
wood	15.93%
tiled	13.49%
fine_concrete	9.53%
hard_tiles_large_space	8.08%
soft_tiles	7.80%
carpet	4.96%
hard_tiles	0.55%

Class	Percentage
low friction	75.83%
high friction	24.17%

In the context of this problem, the only attribute we care about is the texture of the surface, i.e. how much friction it causes. While it is possible to split the data into several levels of texture, a binary classification suffices. The high friction surfaces are selected to be *carpet* and *soft_pvc* and all other surfaces are low friction. Thus, the dataset can be classified as such. 20 test cases have been created for this neural network (found in test folder of project). Accuracy Rate: 19/20, 95%

ChargingSafetyNN

The data for ChargingSafetyNN has been collected from the AKCP website on battery temperature monitoring systems.



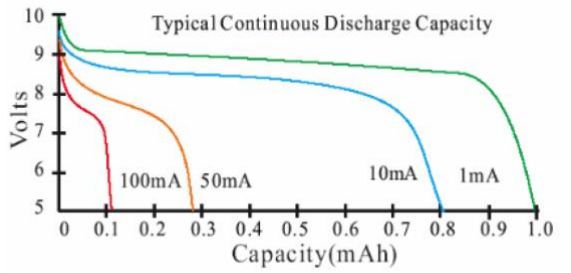
The neural network is trained on temperatures (in Celsius) and voltages that correspond to both acceptable (top-line) and unacceptable charging conditions. Essentially, the neural network measures if the voltage is acceptable, based on the current battery temperature. This neural network is part of the safety feature of the Prometheus robot; it offers no direct assistance in the locomotion of the robot. Nevertheless, it is useful when deciding, later on, if the robot should or should not be charged. 10 test cases have been generated for this neural network.

Voltage (V)	Temperature (C)	Class	Outcome
16.25	-39	safe	true
15.4	-7	safe	true
14.95	5	safe	true
14.3	28	safe	true
13.9	40	safe	false
13.25	-40	unsafe	true
13.75	-28	unsafe	true
14	-20	unsafe	true
14.5	-4	unsafe	true
16	30	unsafe	true

Accuracy Rate: 9/10, 90%

ChargeBatteryNN

The ChargeBatteryNN is also part of the safety feature of the Prometheus robot and aims to predict, based on current and voltage readings, if the robot needs to be charged or not. The data used for this neural network has been collected from the continuous discharge capacity graph shown below.



Here is a glimpse of the dataset ChargeBatteryNN has been trained on.

Input current (mA)	Voltage (V)	Remaining Capacity	Full Capacity	Class
1	10	0.0	1.0	charge needed
1	5.5	0.995	1.0	no charge
50	9.0	0.015	0.2825	charge needed

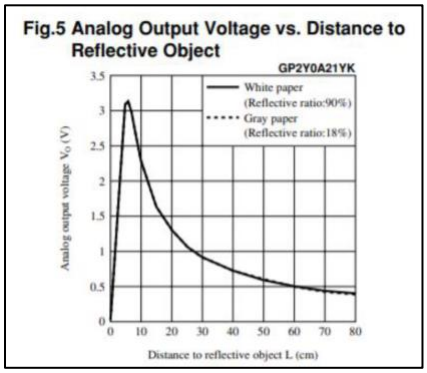
The robot is classified as requiring charge if, based on the input current and voltage, the remaining capacity is at most 50% of the full capacity. So, in the second row, the remaining capacity is 99.5% of the full capacity, and so charge is not required. Whereas in the first row, the remaining capacity is 0% of the full capacity (empty) and so charge is surely needed. When testing the neural network, only the voltage and current will be provided. In other words, the actual dataset the neural network is trained on does not include the capacities in the calculation. 17 test cases are provided for this neural network, as shown below.

Input current (mA)	Voltage (V)	Class	Outcome
1	9.75	charge needed	true
1	8.25	no charge	true
1	7.25	no charge	true
1	6.25	no charge	true
10	8.75	charge needed	true
10	7.25	no charge	true
10	6.25	no charge	true
10	5.25	no charge	true
50	9.15	charge needed	true
50	8.25	charge needed	true
50	7.25	no charge	true
50	6.25	no charge	true
50	5.25	no charge	true
100	8.75	charge needed	false
100	7.75	charge needed	false
100	6.75	no charge	true
100	5.75	no charge	true

Accuracy Rate: 15/17, 88.24%

UltraSonicSensorNN

Data for the UltraSonicSensorNN has been collected from SHARP datasheet for sensor GP2Y0A21YK. Internally, the transmission and the reception time of the pulse is used to calculate the distance.

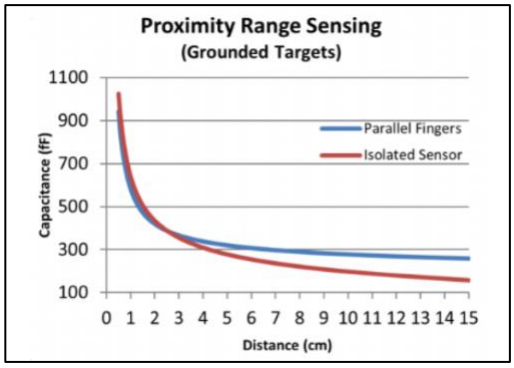


As the robot approaches the reflective object, i.e. obstacle, the voltage increases exponentially up to a certain point after which the sensor is no longer usable. In our problem, we assume distances greater than 13 cm to be far and those less than 13 cm to be close to the robot. 11 test cases have been created for this neural network as depicted below. Note that the close/far classes refer to the *potential* obstacles in front of the robot, not whether the destination is far or close.

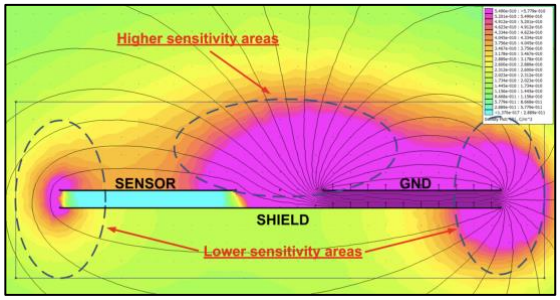
Voltage(V)	Class	Outcome
2.8	close	true
2.6	close	true
2.3	close	true
2.15	close	true
1.8	far	true
1.6	far	true
1.3	far	true
1.26	far	true
0.8	far	true
0.6	far	true
0.55	far	true

Accuracy Rate: 11/11, 100%

TouchSensorNN



The TouchSensorNN uses capacitance, in femtofarad, and the fringing electric fields to predict the distance, in cm, between the two whiskers of the sensor. The farther the two whiskers spread, the harder the robot pushes an object. Here, a binary classification suffices: *light/no touch (range<=7.5 cm), hard touch (range>7.5 cm)*. The parallel fingers topology is used over the isolated sensor because of its sensitivity with regards to the location of the nearest ground potential source. David Wang, from Texas Instruments, contends that "if the ground source is much smaller than the electrodes and at a distance much greater than the electrodes, the isolated sensor will be less sensitive at shorter sensing ranges than the parallel fingers" (Wang, 2015). Refer to the figure below for a simulation of the parallel fingers topology. The highest sensitivity is found in the region closest to the inner edges of the sensor where the density and intensity of the electric fields are at their highest. 10 test cases have been provided for the neural network.



Capacitance (fF)	Class	Outcome
600	light touch	true
420	light touch	true
360	light touch	true
207	hard touch	false
198	hard touch	false
189	hard touch	false
179	hard touch	false
163	hard touch	false
300	light touch	true
235	light touch	true

Accuracy Rate: 5/10, 50%

MovableObjectNN

A second-level neural network, MovableObjectNN takes two inputs from TouchSensorNN, along with two inputs for position, and outputs a binary outcome: *is the object pushed by the robot, if any, movable or not?* The two inputs correspond to the current location of the robot, along with its previous location. When the robot moves in any direction, we assume the initial position is 0 and the final position is 1. When the robot pushes against a wall, or obstacle, both initial and final positions are either 0 or 1.

The object is considered movable if there is no change in the TouchSensorNN, but a change in position. In other words, the neural network can only predict the object is movable if the robot is moving, and there is no sudden change in its push against the object. In all other cases, for safety, we assume the object is not movable. 16 test cases have been provided for this neural network.

TouchSensorNN (initial)	TouchSensorNN (final)	Position (initial)	Position (final)	Class	Outcome
0	0	0	0	immovable	true
0	0	0	1	movable	true
0	0	1	0	immovable	true
0	0	1	1	immovable	true
0	1	0	0	immovable	true
0	1	0	1	immovable	false
0	1	1	0	immovable	true
0	1	1	1	immovable	true
1	0	0	0	immovable	true
1	0	0	1	immovable	false
1	0	1	0	immovable	true
1	0	1	1	immovable	true
1	1	0	0	immovable	true
1	1	0	1	movable	true
1	1	1	0	immovable	true
1	1	1	1	immovable	true

Accuracy Rate: 14/16, 87.5%

BatteryFailureNN

Also a second-level neural network, BatteryFailureNN takes input from ChargingSafetyNN, ChargeBatteryNN, and temperature, and outputs a binary outcome: *is the battery likely to fail?* For instance, if it is unsafe to charge the battery and the battery needs to be charged, then it the battery will likely fail. However, if it is safe to charge the battery then the battery will not fail, regardless if needed or not. In extreme temperature, if it is unsafe to charge the battery, the battery is likely to fail, regardless if needed or not. This is because even if the battery does not need charge, the extreme weather conditions put the battery's health at risk.

16 test cases have been provided for this neural network. Notice that the first (-45) and last (45) temperatures are considered extreme weather conditions and pose a greater risk to the battery.

Temperature (C)	ChargingSafetyNN	ChargeBatteryNN	Class	Outcome
-45	safe	must charge	unlikely failure	true
-45	unsafe	no need charge	likely failure	false
-45	safe	no need charge	unlikely failure	true
-45	unsafe	must charge	likely failure	true
-10	safe	must charge	unlikely failure	true
-10	safe	no need charge	unlikely failure	true
-10	unsafe	no need charge	unlikely failure	true
-10	unsafe	must charge	likely failure	true
20	safe	must charge	unlikely failure	true
20	safe	no need charge	unlikely failure	true
20	unsafe	no need charge	unlikely failure	false
20	unsafe	must charge	likely failure	true
45	safe	must charge	unlikely failure	true
45	safe	no need charge	unlikely failure	true
45	unsafe	no need charge	likely failure	true
45	unsafe	must charge	likely failure	true

Accuracy Rate: 14/16, 87.5%

ReachableDestinationNN

ReachableDestinationNN is the third second-level neural network. The logic behind this neural network is deciding whether the robot can or cannot reach the blocks neighboring it. The robot is equipped with three ultrasonic sensors and so has three ReachableDestination neural networks. The neural network processes whether the robot needs to be charged or not (ChargeBatteryNN), how far the block is (UltraSonicSensorNN), and how rough/smooth the surface is (SurfaceClassifierNN). The first and third inputs will be fed to all three ReachableDestination neural networks, while the second one depends on the obstacles in the front, left, and right sides of the robot.

7 test cases have been provided for this neural network. (Notice that since the data is simulated, the individual user should alter their expectations for SurfaceClassifierNN. In other words, the user should decide whether a rough/smooth surface should deter the robot from reaching its destination).

ChargeBatteryNN	UltraSonicSensorNN	SurfaceClassifierNN	Class	Outcome
must charge	close obstacle	low friction	unreachable	true
must charge	far obstacle	low friction	unreachable	true
must charge	far obstacle	high friction	unreachable	true
no need charge	close obstacle	low friction	unreachable	true
no need charge	close obstacle	high friction	unreachable	true
no need charge	far obstacle	low friction	reachable	true
no need charge	far obstacle	high friction	reachable	true

Accuracy Rate: 7/7, 100%

ChangeDirectionNN

ChangeDirectionNN can be considered the 'brain' of the neural network tree. This neural network decides what direction the neural network should take - continue on its current track, turn right, turn left, or turn 180 degrees- by accepting input from all three ReachableDestination neural networks. One very important thing to notice from the dataset is that if there are no obstacles in front of the robot, it will continue on that track. Instead, the dataset can be altered to allow for a random choice of direction in such a case. Both options are imperfect as will be seen later. 8 test cases have been created for this neural network.

ReachDestNN (left)	ReachDestNN (fwd)	ReachDestNN (right)	Class	Outcome
close	far	close	forward	true
far	far	close	forward	true
far	far	far	forward	true
close	far	far	forward	true
far	close	close	left	true
close	close	far	right	true
far	close	far	right	true
close	close	close	180 - turn	true

Accuracy Rate: 8/8, 100%

NextDirectionNN

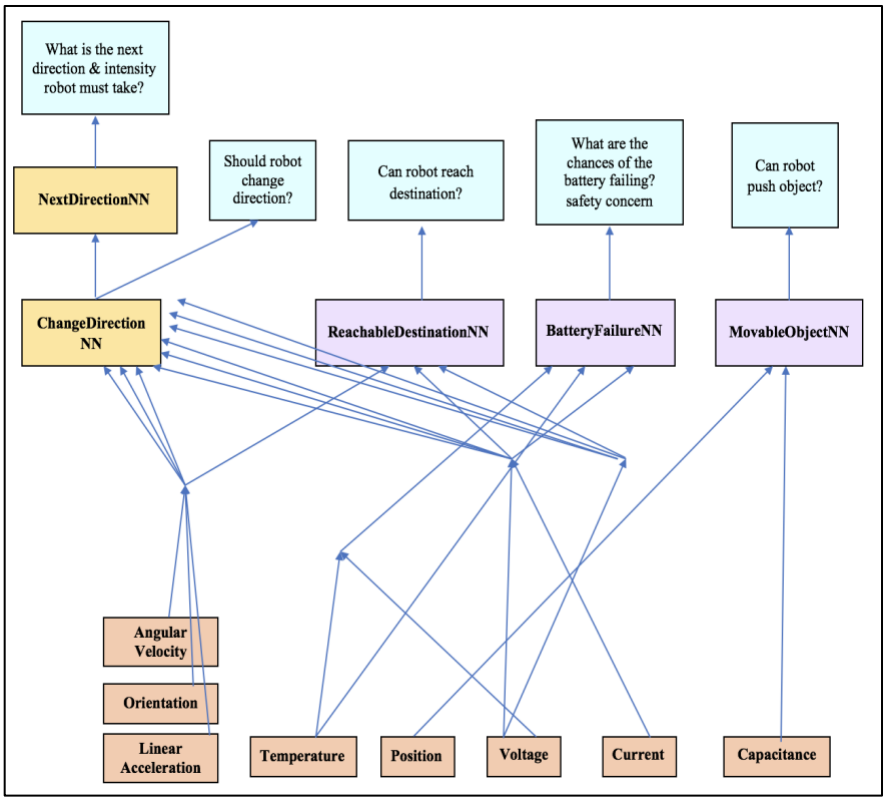
While ChangeDirectionNN decides the direction the robot should take, it makes no explicit distinction between rough and smooth surface types. While this may not affect forward motion, it can affect the intensity with which the robot should turn right and left. For instance, if on a high friction surface, the robot may need to turn with a higher intensity to avoid colliding with the obstacle in front of it. 6 test cases have been provided for this neural network.

ChangeDirectionNN	SurfaceClassifierNN	Class	Outcome
forward	high friction	forward	true
forward	low friction	forward	true
right	high friction	sharp right-turn	true
right	low friction	slight right-turn	true
left	high friction	sharp left-turn	true
left	low friction	slight left-turn	true

Accuracy Rate: 6/6, 100%

b) Neural Network: Version 2

The second variant of the neural network structure is simpler in structure and eliminates all first-level neural networks.



Raw data, rather than network processed input, is fed into higher-level neural networks. The neural networks are trained on larger datasets, with larger number of columns, and are expected to yield the same output. Here are the individual neural networks and their test cases.

ReachableDestinationNN (version 2):

The dataset for this neural network is created my merging the raw data for ChargeBatteryNN (voltage, current), UltraSonicSensorNN (voltage), and SurfaceClassiferNN (angular velocity, orientation, linear acceleration). This new dataset has 68, rather than 3, columns. The expected output is handled by the method giveMergedOutput, which is specific to each neural network.

7 test cases have been provided for this neural network. Note that the 65 columns for SurfaceClassiferNN are not shown; instead, the expected surface corresponding to that row of data is mentioned.

Current	Voltage (chargebattery)	Voltage (ultrasonic)	SurfaceClassifierNN (65 columns)	Class	Outcome
1	9.75	3	low friction	unreachable	true
1	9.3	1.6	low friction	unreachable	true
10	8.75	1.55	low friction	unreachable	true
1	8.25	2.95	high friction	unreachable	true
1	7.25	2.80	low friction	unreachable	true
1	6.25	1.15	low friction	reachable	false
1	5.25	0.95	low friction	reachable	false

Accuracy Rate: 5/7, 71.4%

BatteryFailureNN (version 2):

The dataset for this neural network is created my merging the raw data for ChargeBatteryNN (voltage, current), ChargingSafetyNN (temperature, voltage). This new dataset has 4, rather than 3, columns. 16 test cases have been provided for this neural network.

Temperature (C)	Voltage (chargingSafetyNN)	Current (mA)	Voltage (chargebattery)	Class	Outcome
-45	16.5	10	9.0	unlikely failure	false
-45	13.0	1	5.5	likely failure	true
-45	16.5	1	8.5	unlikely failure	false
-45	13.25	1	10.0	likely failure	true
-10	15.4	1	9.5	unlikely failure	false
-10	15.4	1	7.0	unlikely failure	false
-10	14.25	1	6.5	unlikely failure	true
-10	14.0	10	9.6	likely failure	false
20	14.5	10	9.0	unlikely failure	true
20	14.5	10	7.0	unlikely failure	true
20	16.25	10	6.5	unlikely failure	true
20	16.3	10	8.5	likely failure	false
45	13.75	50	9.25	unlikely failure	true
45	13.75	50	6.0	unlikely failure	true
45	15.5	50	5.5	likely failure	false
45	16.0	100	9.0	likely failure	false

Accuracy Rate: 8/16, 50%

MovableObjectNN (version 2):

The dataset for this neural network is created my merging the raw data for TouchSensorNN (capacitance), and position. The two inputs correspond to the current location of the robot, along with its previous location. When the robot moves in any direction, we assume the initial position is 0 and the final position is 1. The object is considered movable if there is no change in the TouchSensorNN, but a change in position. The new dataset is also made of 4 columns. 16 test cases have been provided for this neural network.

Capacitance (fF) (initial)	Capacitance (fF) (final)	Position (initial)	Position (final)	Class	Outcome
9	9	0	0	immovable	true
9.5	9.5	0	1	movable	false
10	10	1	0	immovable	true
10.5	10.5	1	1	immovable	true
11	7.5	0	0	immovable	true
12	7.0	0	1	immovable	true
12.5	6.5	1	0	immovable	true
13	6	1	1	immovable	true
5.5	13.5	0	0	immovable	true
5	14	0	1	immovable	true
4.5	14.5	1	0	immovable	true
4	15	1	1	immovable	true
3.5	3.5	0	0	immovable	true
3	3	0	1	movable	false
2.5	2.5	1	0	immovable	true
2	2	1	1	immovable	true

Accuracy Rate: 14/16, 87.5%

ChangeDirectionNN (version 2):

While the original ChangeDirectionNN takes its input from 3 ReachableDestination neural networks, this version takes the 3 sets of raw data fed into ReachableDestinationNN. The raw data for ChargeBatteryNN (voltage, current), UltraSonicSensorNN (voltage), and SurfaceClassiferNN (angular velocity, orientation, linear acceleration) are all fed, 3 times, into this neural network. So, this neural network has (3+65) *3=204, rather than 3, columns in its dataset.

12 test cases have been generated for this neural network. Again, note that the 65 columns for SurfaceClassiferNN are not shown; instead, the expected surface corresponding to that row of data is mentioned. Also note that the columns are abbreviated as: C_i for current, Vch_i for voltage fed into ChargeBatteryNN, Vus_i for voltage fed into UltraSonicSensorNN, S_i for expected output from SurfaceClassifierNN, where i=1 is the left sensor, i=2 is the front sensor, and i=3 is the right sensor.

C_1, C_2, C_3 (mA)	Vch_1, Vch_2, Vch_3 (volt)	S_1, S_2, S_3	Vus_1	Vus_2	Vus_3	Class	Predicted	Outcome
1	6	low friction	3	1.15	2.25	forward	forward	true
1	5.5	low friction	0.95	0.75	2.25	forward	forward	true
10	5.5	low friction	0.5	0.75	1.15	forward	forward	true
10	5.25	high friction	2.8	1.25	1.15	forward	forward	true

50	6.5	low friction	1.15	2.80	2.95	left-turn	forward	false
50	6.25	low friction	1.30	3.0	3.0	left-turn	forward	false
50	5.5	low friction	2.95	2.80	1.25	right-turn	forward	false
100	6.25	low friction	0.6	1.8	0.7	right-turn	forward	false
100	5.75	low friction	2.75	2.80	1.25	right-turn	forward	false
100	5.0	low friction	0.5	1.95	0.5	right-turn	forward	false
100	4.75	low friction	3.0	3.0	3.0	180-turn	forward	false
100	4.5	low friction	2.5	2.75	2.5	180-turn	forward	false

Accuracy Rate: 4/12, 33.3%

NextDirectionNN (version 2):

NextDirectionNN is not changed for this neural network structure. An alternative would be combining the previous dataset for ChangeDirectionNN with that of SurfaceClassifierNN, and training the neural network. However, since the accuracy of ChangeDirectionNN is very low, adding an extra layer would be ineffective.

a) Analysis

a) Version 1 vs Version 2

To compare both neural network structures, several factors must be taken into account. First and foremost, the individual accuracies of the neural networks must be compared. Then, the training times must be studied. And finally, we must analyze how useful the results of the neural networks are for future studies.

To measure the training times, I use the built in Java System.currentTimeMillis() method which returns the current time in milliseconds. Here, I only take into account the time the neural network needs to train on the dataset. This does not include any form of data preprocessing, such as in the case for SurfaceClassifierNN, as we can assume these datasets have been built and are ready. Here is a snippet of the code used to measure training time.

```
@Override
public void trainNN() {
    DataSet trainSet=this.generateTrainingSet();
    long start = System.currentTimeMillis();
    aNeuralNetwork.learn(trainSet);
    long finish = System.currentTimeMillis();
    long timeElapsed = finish - start;
}
```

The accuracy is directly extracted from the test cases shown previously. Ideally, there would be a larger dataset for testing our data; however, I make sure to encompass all possible classes for the neural networks. Below is a detailed comparison of the time and accuracy of version 1 and version 2 of the neural networks. It is worth noting that the training time for the version 1 neural networks reflect the time needed to learn the dataset these neural networks are trained on. However, in a real-life simulation, raw data is first fed into the first-level neural networks, and that output is fed into the higher-level neural networks. So, the column total time encompasses the time needed to train, not only the current neural network, but also its lower level neural networks. The individual times for the neural networks are found below the table. The same code was used to measure these neural networks.

Neural Network (version 1 vs version 2)

Neural Network	Accuracy (V1)	Training Time ms (V1)	Total Time (V1)	Accuracy (V2)	Training Time ms (V2)	Shorter Time	Higher Accuracy
BatteryFailureNN	87.5%	2371	3023	50%	772	V2	V1
MovableObjectNN	87.5%	1395	1740	87.5%	699	V2	V1=V2
ReachableDestinationNN	100%	54	26693	71.4%	2188	V2	V1
ChangeDirectionNN	100%	832	27525	33.3%	6466	V2	V1

ChargeBatteryNN=626ms, ChargingSafetyNN= 26ms, SurfaceClassifierNN= 25555ms, TouchSensorNN= 345ms, UltraSonicSensorNN=458ms

In terms of accuracy, it is clear that version 1 performs substantially better. All version 1 neural networks have accuracies greater than or equal to those from version 2. In fact, the most critical neural network, ChangeDirectionNN, which is considered the brain of the robot, reveals a 66.7% increase in accuracy with the first version. The second version is biased and tends to predict most ‘next directions’ as forward. This would lead the robot to crash into obstacles.

Interestingly, MovableObjectNN has the same accuracy in both versions. This is a result of a bias in the training, as both neural networks tend to predict that the object is not movable. The same problem occurs with the second version of ReachableDestinationNN as it tends to predict that the destination is not reachable in most cases. BatteryFailureNN version 2 has no explicit bias in its results but is unpredictably inaccurate.

However, it is critical to point out that the accuracies shown do not reflect the precision of the neural networks. In other words, if the dataset itself is biased, then these accuracies do not reflect the performance of the neural networks themselves, but the datasets they were trained on. I tried to avoid any bias in the training set by creating larger datasets with a larger number of combinations of raw data. However, covering all possible inputs is difficult and costly, both in terms of space and time. For instance, consider version 2 of ReachableDestinationNN which accepts 204 columns of input in its training. Ideally, the training set should then allow for all combinations of these columns. Assume falsely that we only have 10 options for each column; i.e. 10 options for current, 10 options for voltage, etc. The dataset should then contain 10*10*10…=10^204 rows. A dataset with this number of rows is unrealistically large and, assuming the program does not crash, would take substantially longer to be learned. Hence, it must be acknowledged that the training process itself is flawed, and is bound to contain bias.

The first version of the neural network is also flawed and should be used very carefully. This is because the datasets of the higher-level neural networks are much smaller than those of the first-level neural networks. So, there is not much room for error. For instance, consider ChangeDirectionNN, the 'brain' of the neural network tree. This neural network has only 4 classes in its dataset: forward, right-turn, left-turn, and 180-turn. A false prediction can easily lead the robot to collide with an obstacle. This means that the errors of the lower level neural networks are propagated to the higher-level neural networks. For instance, assume UltraSonicSensorNN were biased and only predicted far obstacles. If the surface were of low friction and the battery needed no charging, then ReachableDestinationNN might then predict that the destination is always reachable. ChangeDirectionNN would accept a left input of far, a forward input of far, and a right input of far. Based on the training set, it will continue to move forward and not turn in any direction. This will then lead the robot to crash, even though ChangeDirectionNN, itself, has a very high accuracy.

In terms of time, the second version of the neural network tree performs better, as expected, in all neural networks. However, it is worth noting that these times reflect the cumulative training times, not the time needed to predict a certain output. If, in our design, we choose to train the neural network on every single block, then version 2 performs significantly better. However, if we choose to train the neural networks only once, and then use them throughout the robot's entire course, then these times are irrelevant. Prediction time is insignificant and exhibits no difference with both neural networks.

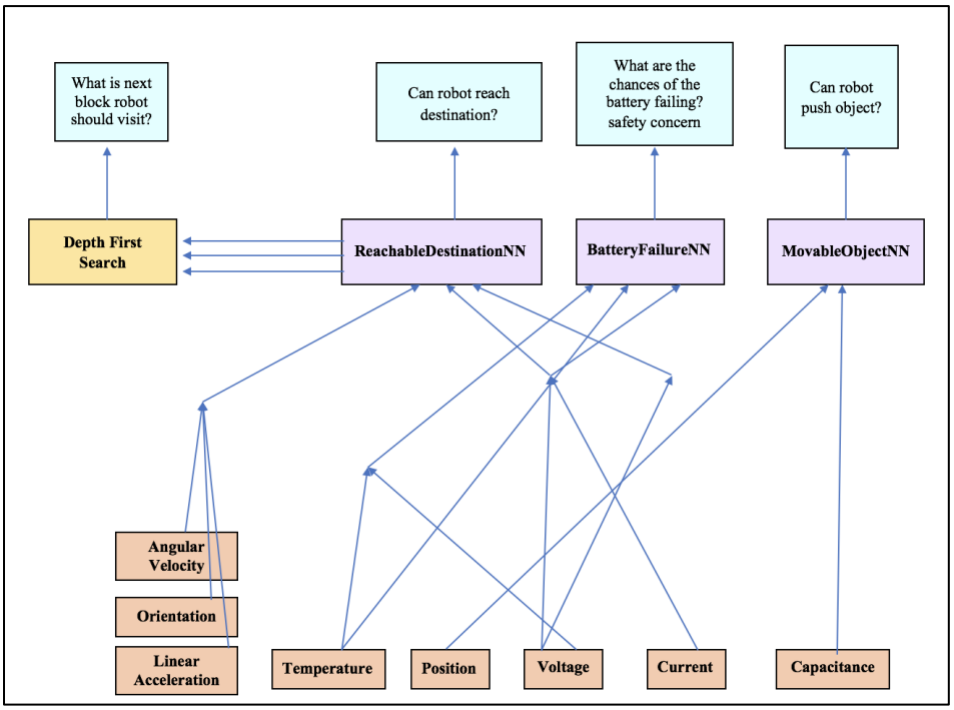
Finally, both neural network structures must be compared in terms of the information they provide for future neural networks. For instance, assume we now want a neural network that accepts visual data to predict the surface type. We find that this neural network performs significantly better than the current SurfaceClassifierNN and choose to use it instead. With the first neural network tree version, we can simply delete the current neural network and replace it with the new visual neural network. This will not significantly alter the higher level neural networks. For example, ReachableDestinationNN is only told if the surface is rough or smooth, and so nothing will change in its dataset. The same is true for NextDirectionNN. However, if we use the second neural network structure, then the data sets for the higher level neural networks must change. A new method must be found to merge the data from the new surface classifier and the raw data originally present.

Now assume we want to add a neural network that predicts how much time the battery has left to survive based on how hard the robot is pushing on an object, how rough the surface is, and whether the battery needs to be

charged or not. With the first version, this new neural network can accept input from TouchSensorNN, SurfaceClassifierNN, and ChargeBatteryNN. With the second version, these first-level neural networks do not exist. Therefore, such a neural network is not feasible, and new data must be collected.

In short, the first version provides more customizability and more easily allows new neural networks to replace current ones. This is made possible by the encapsulation and information-hiding the first version provides. It also allows for more streams of logic, as new connections can be made from the current neural networks. With the second version this is not possible, as new data must be recollected.

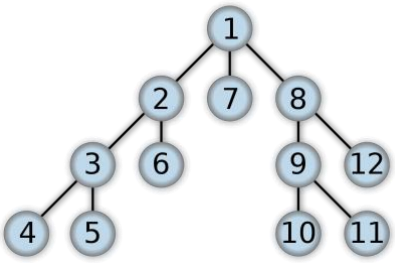
Neural Network Tree: Version 3



While the first version neural network tree may seem ideal, it is somewhat flawed. Specifically, ChangeDirectionNN works by always assuming a forward position if no obstacles are in the robot's way; the robot only turns if it absolutely must. One approach may be creating a new dataset that allows for a random selection of direction in case there are no obstacles in front of the robot, but this solution is imperfect.

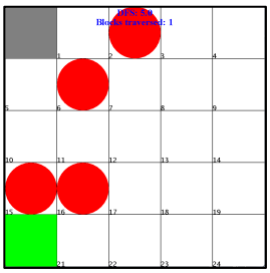
Instead, we introduce a path-planning algorithm that, instead of ChangeDirectionNN, acts as the 'brain' of the neural network tree. This path-planning algorithm guides the robot to its destination while also making use of the lower level neural network outputs.

In my simulations, I make use of the depth first search algorithm, abbreviated as DFS. Depth first search works by starting at a root node and exploring the grid as far as possible. Once the robot cannot go any deeper in its search, it backtracks, checks for unvisited blocks and traverses them. This algorithm is guaranteed to locate the robot's destination.

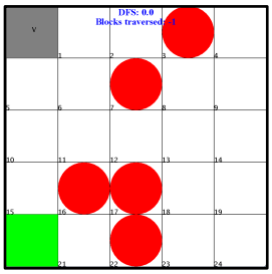


To test this neural network tree, I created several simulations of different sizes and with different obstacles/destinations for the robot to traverse. The idea is that the neural network tree that guides the robot to its destination with the fewer number of blocks is more efficient. Refer to these simulations below.

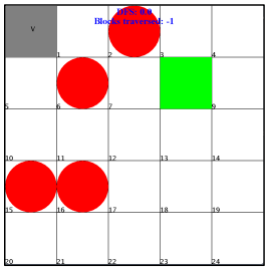
Simulation 1
Size: 5x5
Obstacles: 4
Destination: block 20
version 1: 15 blocks
version 3: 31 blocks



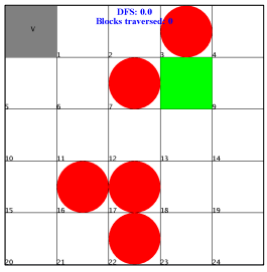
Simulation 2
Size: 5x5
Obstacles: 5
Destination: block 20
version 1: 9 blocks
version 3: 5 blocks



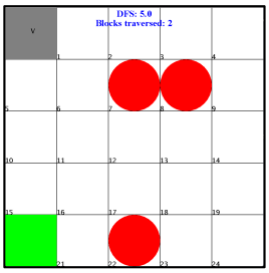
Simulation 1.b
Size: 5x5
Obstacles: 4
Destination: block 8
version 1: 25 blocks
version 3: 15 blocks



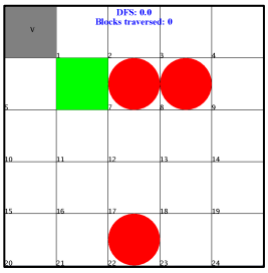
Simulation 2b
Size: 5x5
Obstacles: 5
Destination: block 8
version 1: infinite blocks
version 3: 21 blocks



Simulation 3
Size: 5x5
Obstacles: 3
Destination: block 20
version 1: infinite loop
version 3: 5 blocks

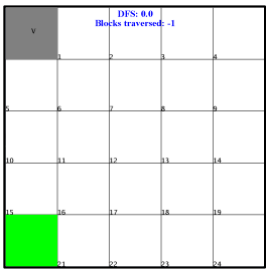


Simulation 3b
Size: 5x5
Obstacles: 3
Destination: block 6
version 1: infinite loop
version 3: 19 blocks

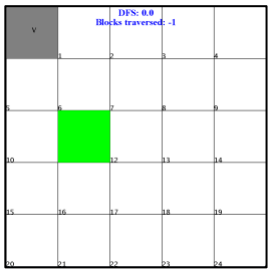


Simulation 4
Size: 5x5
Obstacles: 0
Destination: block 20

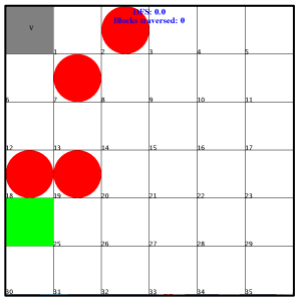
version 1: 13 blocks
version 3: 5 blocks



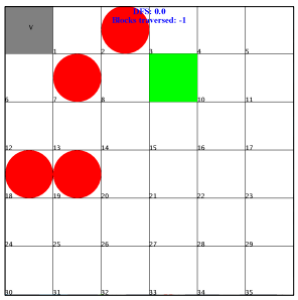
Simulation 4b
Size: 5x5
Obstacles: 0
Destination: block 11
version 1: infinite loop
version 3: 24 blocks



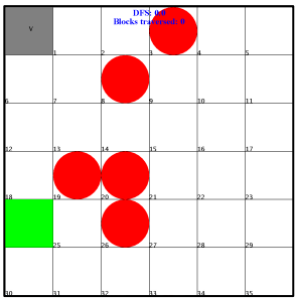
Simulation(6x6): 1
Size: 6x6
Obstacles: 4
Destination: block 24
version 1: 19 blocks
version 3: 49 blocks



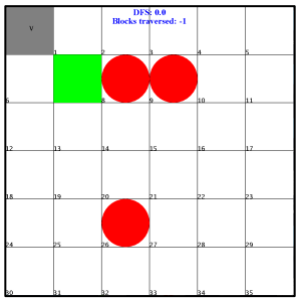
Simulation(6x6): 1b
Size: 6x6
Obstacles: 4
Destination: block 9
version 1: infinite loop
version 3: 25 blocks



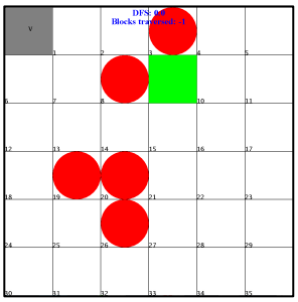
Simulation(6x6): 2
Size: 6x6
Obstacles: 5
Destination: block 24
version 1: 9 blocks
version 3: 5 blocks



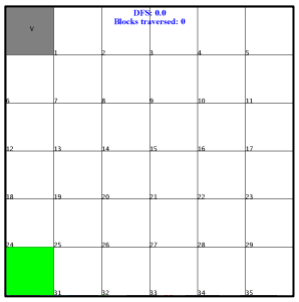
Simulation(6x6): 3b
Size: 6x6
Obstacles: 3
Destination: block 7
version 1: infinite loop
version 3: 31 blocks



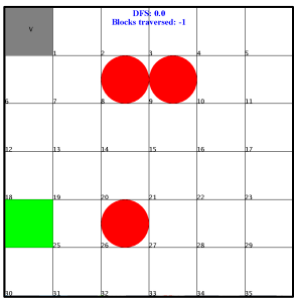
Simulation(6x6): 2b
Size: 6x6
Obstacles: 5
Destination: block 9
version 1: infinite loop
version 3: 25 blocks



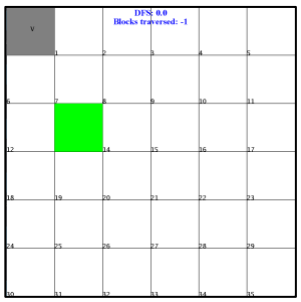
Simulation(6x6): 4
Size: 6x6
Obstacles: 0
Destination: block 30
version 1: 16 blocks
version 3: 6 blocks



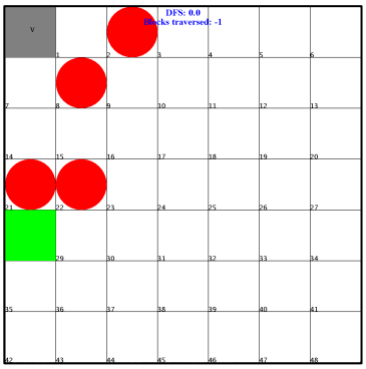
Simulation(6x6): 3
Size: 6x6
Obstacles: 3
Destination: block 24
version 1: 17 blocks
version 3: 5 blocks



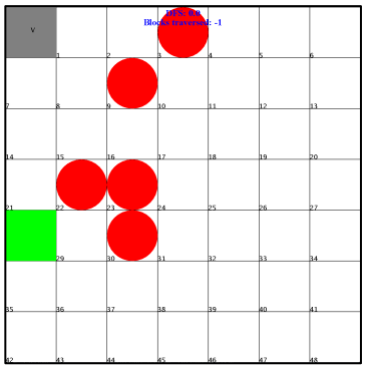
Simulation(6x6): 4b
Size: 6x6
Obstacles: 0
Destination: block 13
version 1: infinite loop
version 3: 34 blocks



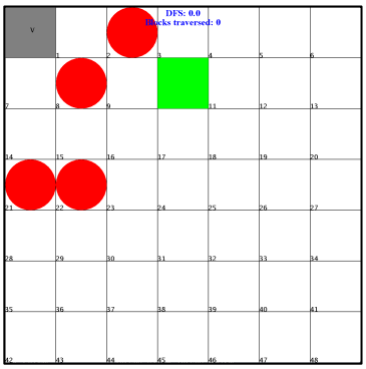
Simulation(7x7): 1
Size: 7x7
Obstacles: 4
Destination: block 28
version 1: 23 blocks
version 3: 79 blocks



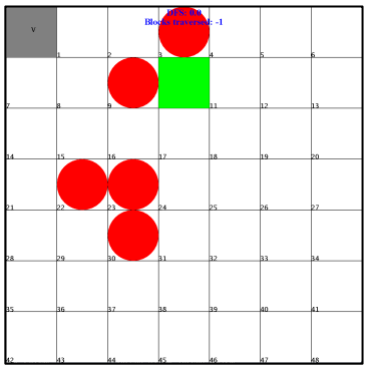
Simulation(7x7): 2
Size: 7x7
Obstacles: 4
Destination: block 28
version 1: 9 blocks
version 3: 5 blocks



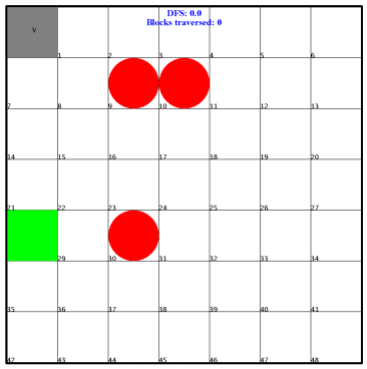
Simulation(7x7): 1b
Size: 7x7
Obstacles: 4
Destination: block 10
version 1: infinite loop
version 3: 33 blocks



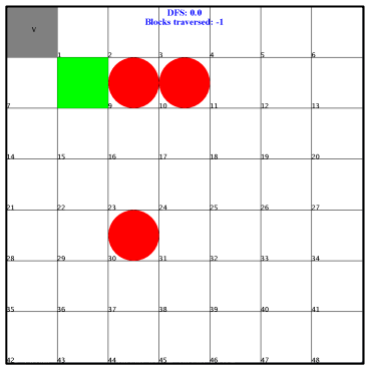
Simulation(7x7): 2b
Size: 7x7
Obstacles: 4
Destination: block
version 1: infinite loop
version 3: 33 blocks



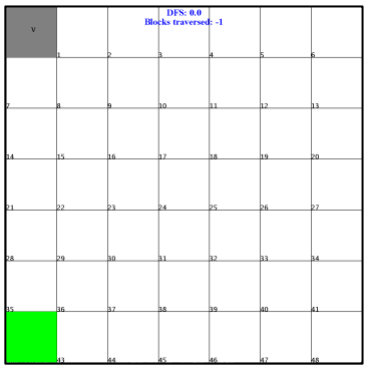
Simulation(7x7): 3
Size: 7x7
Obstacles: 3
Destination: block 28
version 1: 21 blocks
version 3: 5 blocks



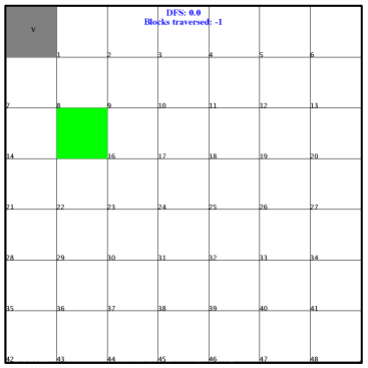
Simulation(7x7): 3b
Size: 7x7
Obstacles: 3
Destination: block
version 1: infinite loop
version 3: 35 blocks



Simulation(7x7): 4
Size: 7x7
Obstacles: 0
Destination: block 42
version 1: 19 blocks
version 3: 7 blocks



Simulation(7x7): 4b
Size: 7x7
Obstacles: 0
Destination: block 15
version 1: infinite loop
version 3: 46 blocks



Simulation:
Version 1 vs Version 3

Size	Obstacles	Blocks Traversed (version 1)	Blocks Traversed (version 3)	Blocks Traversed (min-path)	Less blocks
5x5	5	9	5	5	version 3
		∞	21	7	version 3
	4	15	31	9	version 1
		25	15	7	version 3
	3	∞	5	5	version 3
		∞	19	3	version 3
	0	13	5	5	version 3
		∞	24	4	version 3
6x6	5	9	5	5	version 3
		∞	25	7	version 3
	4	19	49	9	version 1
		∞	25	7	version 3
	3	17	5	5	version 3
		∞	31	3	version 3
	0	16	6	5	version 3
		∞	34	4	version 3
7x7	5	9	5	5	version 3
		∞	33	7	version 3
	4	23	79	9	version 1
		∞	33	7	version 3
	3	21	5	5	version 3
		∞	35	3	version 3
	0	19	7	5	version 3
		∞	46	4	version 3

Notice that for every size and obstacle number, there are two versions of the simulation, each with a different destination. For example, for a grid of size 5x5, there are two simulations with 5 obstacles in the exact same positions. The only difference is the location of the destination.

b) Version 1 vs Version 3

The first thing to notice from this table is that the version 1 neural network tree has a success rate of only 12/24=50%. This means that for half of the simulations, the robot gets eventually stuck in an infinite loop. This can be attributed mainly to the dataset on which ChangeDirectionNN is trained.

ChangeDirectionNN assumes a forward direction if the robot is not required to turn. The robot, then, has no incentive to explore the grid it is placed on, unless it absolutely has to. In fact, one thing to notice from the table above is that the second simulation with zero obstacles fails for all grid sizes. This is because once the robot moves against the wall, it will remain moving against the wall until it reaches a corner. The robot does not explore the center of the grid and, thus, fails in locating its destination.

However, this does not imply that the robot is guaranteed to locate a destination along a wall. In fact, this can only be proven in simulation 5x5 with 3 obstacles. Although the destination is in the corner of the grid, the robot will never reach it. This is because, when selecting a new direction, ChangeDirectionNN favors right-turns over left-turns. When the simulation starts, the robot turns right, moves forward to the right wall, turns right again, moves forward to the bottom wall, turns right again, moves forward to the obstacle, turns right, moves forward to the obstacles, turns left, and moves forward to the left wall. When

the robot reaches the left wall, it will select a right-turn rather than a left-turn due to ChangeDirectionNN. This forces the robot into an infinite loop.

However, this also does not imply that the robot cannot locate destinations not in corners or along the walls. This is shown in the second simulation of size 5x5 and 4 obstacles. Out of all 12 simulations with destinations not against a wall, this is the only one the robot locates. By chance, after perceiving an obstacle in its way, the robot moves downward and finds the destination on its course.

Another detail to notice from the table is that the robot locates its destination faster with version 1 of the neural network tree in all the first simulations with 4 obstacles. However, this is a consequence of the depth-first-search algorithm and is simply a matter of chance. DFS will go as deep as possible on the right side of the grid before exploring the left side of the grid in all 3 simulations with 4 obstacles.

Another detail to extract from the table is that for a fixed number of obstacles and a fixed destination, an increase in grid size corresponds to a greater, or equal, number of traversed blocks. For these simulations, then, the increase in size, does not affect the logic of the neural network, especially since the obstacles are in the same locations.

The table also reveals that depth-first-search is far from being closest to the minimum path algorithm. With the coincidental exception of the first 5-obstacle simulations, DFS performs worse than the minimum path. For instance, in the first 7x7 grid with 4 obstacles, DFS performs more than 8 times worse than the minimum path.

Nevertheless, compared to version 1 of the neural network tree, the version 3 neural network structure is expectably more reliable and safer to use. While both algorithms seem to be safe from collision with obstacles, only version 1 is susceptible to leading the robot into an infinite loop.

b) Conclusion/Discussion

a) Summary

In an attempt to simulate the Prometheus robot's decision-making, this paper introduces a tree-like neural network that allows for sensor detection, input processing, and command delivery. This neural network design is successful in allowing higher level decision making. The paper establishes that, compared to a neural network structure with no first-level neural networks, the tree neural network yields higher accuracy rates. Through the encapsulation of individual neural networks, it also allows room for customizability and more easily allows new neural networks to replace current ones. Nonetheless, the version 2 neural network design is much faster than its counterpart, as the data is fed directly and without preprocessing into the higher-level neural networks.

This paper also presents a third solution to the Prometheus problem by employing a neural network tree coupled with a path-finding algorithm. While the original neural network tree and this version both protect the robot from collision with obstacles, only this version guarantees the robot will reach its destination. This first version is at risk of getting the robot stuck in an infinite path mainly due to the dataset on which ChangeDirectionNN is trained. With the first version, the robot is more likely-but is not guaranteed-to locate the destination if it is along one of the walls of the grid.

b) Future work

While the third design of the neural network tree is safe, it is not effective. To correctly compare ChangeDirectionNN and a path-finding algorithm, like DFS, reinforcement learning must be introduced. Reinforcement learning is a closed-loop problem and works by rewarding desired behaviors and punishing undesired ones. Sutton and Barto state, "the learner is not told which actions to take, as in many forms of machine learning, but instead must discover which actions yield the most reward by trying them out" (Sutton & Barto, 2015). In the context of Prometheus, this method of learning would guide the robot out of an infinite loop. The undesired behavior could be both an increase in blocks traversed and the traversal of blocks already visited.

This new neural network should then be compared to not only depth-first-search but also other path-finding algorithms. For example, the bug finding algorithm works by guiding the robot towards its

destination and following the obstacles in its way until it can safely reach its destination. A* search algorithm can also be tested by exploring adjacent blocks until the destination can be reached.

One important to detail to account for is that A* search algorithm can only be used partially with the current robot. Since the current robot only has 3 ultrasonic sensors, blocks behind the robot and on its diagonals, cannot be accounted for when exploring adjacent blocks. An alternative to this would be to allow the robot to travel along its diagonals and compare the performance before and after adding this feature. The addition of moving obstacles should also be considered in future works on this topic.

c) Impact on Society

The impact of tree neural networks is extensive, especially in the field of robotics and video games. While individual neural networks allow programs to recognize patterns and solve general problems, tree neural networks allow for higher-level decision making. This can be used, for instance, with self-driving cars that perceive the world through cameras, radar, and LiDAR (Light Detection and Radar) sensors. Visual data mining, coupled with a tree neural network, can theoretically offer basic motor commands to the car – similarly to how Prometheus works. This technique can also be used with robot vacuum cleaners that learn and remember floor plans.

References

Neuroph. (n.d.). *Multi Layer Perceptron*. Multi layer perceptron. http://neuroph.sourceforge.net/tutorials/MultiLayerPerceptron.html.

Schmidt, A. (n.d.). *Multilayer Neural Network*. Multilayer neural network. https://www.teco.edu/~albrecht/neuro/html/node18.html.

Sutton, R. S., & Barto, A. G. (2015). *Reinforcement Learning: An Introduction*. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

Wang, D. (2015, April). *Capacitive Proximity Sensing Using the FDC1004*. Texas Instruments. https://www.ti.com/lit/an/snoa928a/snoa928a.pdf?ts=1628203903068&ref_url=https%253A%252F%252Fwww.google.com%252F).

Wood, T. (2020, September 2). *Backpropagation*. DeepAI. https://deepai.org/machine-learning-glossary-and-terms/backpropagation.

Images

Depth-first search. (n.d.). https://en.wikipedia.org/wiki/Depth-first_search#/media/File:Depth-first-tree.svg.

Femm Simulation of the Electric Flux Density for the Parallel Fingers Topology. (2015). Texas Instruments. https://www.ti.com/lit/an/snoa928a/snoa928a.pdf?ts=1628203903068&ref_url=https%253A%252F%252Fwww.google.com%252F.

Multilayer Perceptron. (2018). https://towardsdatascience.com/power-of-a-single-neuron-perceptron-c418ba445095.