PAWS Navigation: Reinforcement Learning for Path Planning in Unknown Environments

Samantha Johnson

University of Maryland, College Park

ENPM690 – Robot Learning

# Abstract

words

# Introduction

PAWS (Psychological Assistance and Wellness System) is in development to be an in-home mental health robotic pet companion. This companion is geared toward helping individuals with moderate to severe mental health conditions, such as depression, anxiety, or PTSD, who would benefit from constant supervision and stimulation from an external source. The goal of the PAWS Bot is to aid in the recovery process and decrease the relapse rate for affected individuals, as well as reduce the high number of suicides that occur amongst this group.

PAWS is a wheeled mobile pet-like robot that is capable of interacting with its human companion through various sensors. One of the main development points for the PAWS Bot is enabling its ability to safely navigate in the human’s home. The robot needs the capability to respond to a human’s distress signal and traverse to their location quickly to assist. This navigation component is a complex problem to solve because the manufactured robot will have no information about the home in which it will exist. To address the complexity of needing to operate in an unknown environment, it was determined that a machine learning technique would be utilized to train the robot’s navigation component.

The machine learning method used for the assistance and navigation component of PAWS will be reinforcement learning. This will allow the robot to use sensors to acquire feedback from its environment in order to make decisions on where it should go. The algorithm chosen for usage in the navigation component is Double-DQN (Double Deep Q-Network), which combines the techniques of Q-learning with Neural Nets. The robot and simulation was originally created using CoppeliaSim (formerly VREM) and programmed using Python, through access with the simulation’s Remote API. Further work was done to simulate the robot in the OpenAI Gym environment due to issues working with CoppeliaSim.

# Background and Related Work

Robot path planning is a highly researched field which has resulted in many path planning algorithms that enable a robot to autonomously navigate to a location. Some well known algorithms like Dijkstra and A\* utilize a discrete grid of the environment and mark the vertices with a cost based on distance or a heuristic function. Both of these algorithms are complete, in that they will find an optimal path if it exists. In contrast there are sampling based algorithms such as RRT (Rapidly Exploring Random Trees) which creates a tree, expanding from the robot’s start position, until it reaches a goal. This algorithm does not necessarily find the optimal path, but will find a path to goal if one does exist. (<http://correll.cs.colorado.edu/?p=965> Path Planning page 75+)

The previously discussed path planning algorithms all have a major element in common; they require a known map of the environment. There are methods that can allow a robot to create a map of its environment for use in path planning, such as utilizing SLAM (Simultaneous Localization and Mapping), but for the PAWS robot it is desired that it can consistently act in an unknown environment without a map. There has been research done into methods that allow a robot to traverse an unknown environment but this field is a complicated and still evolving one. It has previously been proposed to use some application of Evolutionary and Hybrid algorithms to solve this path planning problem, but there is minimal research in the application of Reinforcement Learning to this area (mobile robots motion path planning in unknown environments.pdf).

Some research in path planning utilizing reinforcement learning was examined which utilized a DDQN (Double Deep Q-Network) as the planning algorithm. In one paper, “Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning”, a DDQN is applied to a mobile robot so it can plan in an unknown, dynamic environment. The robot uses lidar sensor data to get a localized view of its environment. After 40,000 training epochs it was shown that the error of the DDQN converged and the average reward was maximized. The trained net was successful in tests on a robot in the real world [#].

# Approach

Based on the promising results of research utilizing reinforcement learning with a DDQN for path planning in unknown environments, it seemed beneficial to utilize this algorithm for the development of the PAWS navigation system. The main objective for the PAWS Bot is to be able to navigate in an unknown environment to reach a goal in a timely manner. The secondary objective is to learn not to crash into obstacles. It is not the goal to necessarily find an optimal path, but a path that takes a shorter amount of time is preferable.

## The Robot

The PAWS Bot is a 2-wheeled circular robot that only performs planar motion which is limited to the cardinal directions; in other words, it is not allowed to move diagonally. The robot can only sense locally and receives a type of beacon signal denoting the location to which it needs to travel. It can determine the distance vector between itself and the beacon location, but it has no other knowledge of the rest of the map. The robot is designed to gather the information about the environment through mainly proximity sensors and feed this information into a decision making process to determine an appropriate action to take. There is no feed back once the robot takes its prescribed action so the control algorithm is open loop.

## The Environment

The environment for the PAWS Bot consists of free space, obstacles, and a goal. To simulate a changing environment the obstacles and initial robot location were kept static, but the goal location changed around the map. This allowed for a new look at the environment every episode iteration. The environment is confined to a set space and closed off by walls.

The reward structure for the environment was complicated to devise. It was determined that the robot should learn to avoid obstacles, traverse to a goal, and reach the goal in a timely manner. So it seemed a simple 1 or 0 reward function would not be sufficient to encompass the scope of the problem. To accomplish the determined goals a function was created that incorporates aspects of distance and time to calculate the reward as shown below.

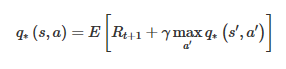
To summarize, if the robot reaches the goal state, it receives the goal reward, which is a large positive value to entice completion of the task. If the robot encounters an obstacle it receives the obstacle reward, which is a moderate negative value to encourage not crashing into obstacles. For all other spaces the reward is a composite of , which creates a positive value for moving towards the objective and a negative value if moving away. , which creates a higher value for being in spaces nearer to the objective and a negative value for being far away. And , which is a negative value that increases as the time limit reaches expiration, which encourages completing the task quickly. The α are weights which sum to one and allow for tuning of the reward function.

## The Agent State

The robot agent needs the capability to observe its local area for obstacles and needs knowledge of the goal it is trying to reach. Due to these needs a state was created that attempts to coalesce this information. The first component is 4 boolean proximity sensor inputs that sense at north, south, east, and west based on the robot’s point of view. They are either 1 or 0 depending on whether an object is sensed by that sensor. The second component is the vector pointing between the human location and robot’s location. This vector is rounded to 1 decimal place in an attempt to batch the state space into somewhat of a grid. The Z coordinate of the pointing vector is excluded due to the environment being planar in motion. The last component is the normalized time elapsed since the episode began which is a decimal between 0 and 1.

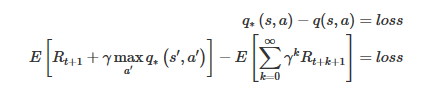
## Double Deep Q-Network

The DDQN used for the PAWS Bot is traditional, where the fundamental aspect is the Bellman Equation. This equation explains the optimal Q-function, which defines the optimal return for a relationship between states and actions. <https://deeplizard.com/learn/video/rP4oEpQbDm4>



4‑1: Bellman Equation

Bellman Equation: <https://deeplizard.com/learn/video/rP4oEpQbDm4>

The neural network in this algorithm functions to approximate this optimal Q-function by minimizing the loss between the values output from the network, and the target values. The neural network that is used in the DDQN algorithm has layers that are all densely connected and utilize ReLU activation functions. The input layer was created with an input size matching the state size for the agent. The output layer is composed of outputs which correspond to each of the 4 possible actions that the robot can take (forward, backward, left, right). There are 2 hidden layers connecting the input to the output layers.

Network Loss: <https://deeplizard.com/learn/video/0bt0SjbS3xc>

Calculating the target values for the loss function can be tricky because the target function is what is being predicted. Using the network being trained to also calculate the target values causes the neural net to essentially eat its own tail during training. To resolve this, it is common to introduce a target network which is used solely to calculate the target values of the Bellman Equation. This target network is then updated on some interval to match the weights of the trained network. The introduction of this target network is what makes this algorithm a *Double* DQN (<https://deeplizard.com/learn/video/xVkPh9E9GfE> ).

The DDQN also utilizes the Epsilon Greedy Strategy which pushes the agent to explore the environment early on in the training phase, to potentially find a more rewarding action than the one that is being chosen by the network. As the training phase goes on, the epsilon value decays, making it more likely to exploit the information it has learned through training ( <https://deeplizard.com/learn/video/mo96Nqlo1L8> ).

The last major component of the DDQN algorithm is called Experience Replay. This technique is accomplished by storing information into Replay Memory, which is then accessed in random batches to train the network. This method is used to reduce the correlation caused by training the network on consecutive samples (<https://deeplizard.com/learn/video/Bcuj2fTH4_4> ).

The above components get applied to create the DDQN algorithm which was utilized for the PAWS navigation system. This algorithm is generalized below.

1. Initialize the environment and agent
2. Copy the policy network to the target network
3. For each episode:
   1. Initialize the simulation
   2. For each time in max time:
      1. Select an action based on e-greedy strategy
         1. Random or from neural net
      2. Execute action in simulation environment
      3. Return reward obtained and next state
      4. Store last experience in replay memory
      5. Perform neural net training
         1. Select batch of memories to replay through policy network
         2. Utilize the target net to obtain values for the next state
         3. Calculate the loss between the output Q values and target Q values
         4. Gradient descent to minimize loss
      6. Decay epsilon
      7. After a set time update the target net weights to match the policy net weights

<https://deeplizard.com/learn/video/xVkPh9E9GfE>

# Implementation

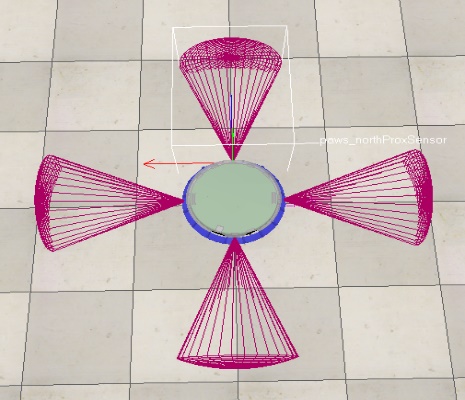
To design the PAWS navigation system the robot and algorithms were implemented via simulation, which allowed a controlled environment for training and testing. The original plan for these tests was to utilize the CoppeliaSim software to simulate the robot and the environment. But due to several issues the use of this software became difficult. After much tweaking and testing it was eventually decided to abandon CoppeliaSim and instead utilize a simulation library that was better suited for the fast pace of machine learning algorithms. The OpenAI Gym was chosen to use for simulation due to it being designed for reinforcement learning problems.

## Original Design - CoppeliaSim

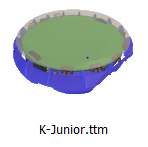
The simulation software utilized in the original design was CoppeliaSim (formerly V-REP). The algorithms were developed using Python which implemented the Keras and TensorFlow for the neural network portions. Various other libraries were used such as numpy and scipy for some of the calculations, as well as matplotlib for data plotting.

### Robot Design and Control

The designed robot was developed using an available model from the CoppeliaSim library. The K-Junior model shown below was modified to be slightly larger and the sensor placement was arranged so that the sensors pointed in the North, South, East, and West directions from the robot’s point of view. The range that the sensors could detect was modified to 0.7m, and the line of sight of the sensors was adjusted so they would each have sensing perimeters that were independent from one another.



‑: Robot with Sensor Locations



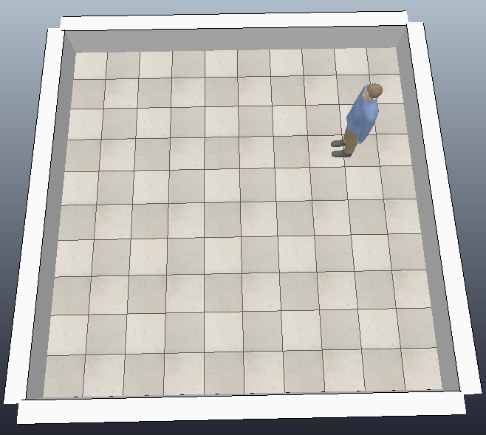
‑:K-Junior Model CoppeliaSim

The robot is controlled via Python modules that interface with the CoppeliaSim Remote API. The possible actions the robot can take are coded as forward or backward, in which the robot moves straight ahead or in reverse, and left or right, where the robot first rotates 90 degrees in the appropriate direction and then moves straight ahead. The necessary motor commands to complete these actions are transmitted to the robot via the API using the simxSetJointTargetVelocity command. The robot was designed to move in steps set to 0.5 m to emulate a grid like state space. The robot can detect when it collides with an object by querying the API for simxReadProximitySensor which returns a vector to the nearest object. If that distance is less than a set tolerance, the robot is determined to be in a collision state.

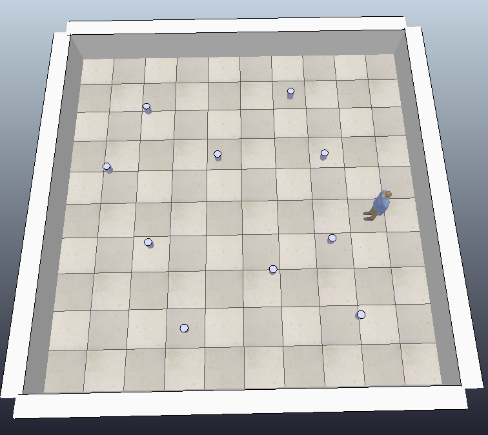
### The Environment

The environment was created in CoppeliaSim and was chosen to be a simple planar floor. The size was either 5m x 5m or 2.5m x 2.5m depending on the tests being run. The obstacles are composed of walls to set the boundaries of the state space and cylindrical posts that the robot can detect and collide with. There were 2 versions of the environment created, a larger one with obstacles, and a smaller one that contained only the goal and walls. The goal that the robot needs to travel to is marked by a human figure that is moved around the environment every new simulation episode.

Initial training and testing was attempted to be done in the larger 5m x 5m state space with obstacles. Due to many issues encountered making training difficult, the state space size was reduced to the 2.5m x 2.5m space without obstacles to see if promising results could be produced.



5‑3: Map 2.5m x 2.5m



5‑4: Map 5m x 5m

### The Algorithm

The DDQN was implemented via Python and followed the algorithm outlined in the previous section. The original plan for the algorithm was to develop the pieces from scratch; this includes the algorithm for the neural network portion of the DDQN. The code for the neural net was originally written using Python with ReLU activation functions and fully connected layers. Due to many issues encountered during the training of the DDQN, it was opted to use the open source library Keras to handle the neural network component instead. This helped eliminate one of the variables of the DDQN algorithm in hopes that it would be easier to narrow down what was going wrong in the training process.

The parameters used in the DDQN are as follows:

**Training Settings**

EPISODES = 500

TIME\_LIMIT = 100

MEMORY\_CAPACITY = 1000

BATCH\_SIZE = 35

TARGET\_UPDATE\_COUNT = 25

**Hyperparameters**

ALPHA = 0.05

DISCOUNT\_RATE = 0.95

EPSILON = 1

EPSILON\_DECAY = 0.995

EPSILON\_MIN = 0.01

The reward weights and values were iterated many times throughout training, but they generally settled around the following values:

**Reward Settings**

GOAL\_REWARD = 100 (or 10)

NOT\_SAFE\_REWARD = -1

REWARD\_DISTANCE\_WEIGHT = 0.4

REWARD\_CLOSE\_WEIGHT = 0.6

REWARD\_FREE\_SPACE = 1

REWARD\_TIME\_DECAY = -0.5

### Data Collection

Throughout training data was being collected and plotted at set intervals. The values that were collected are as follows:

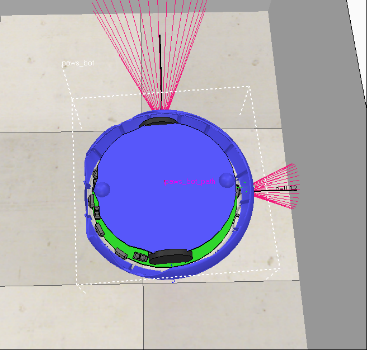
* ***Distance vs. Time****:* Shows the path that the robot took over the course of training. A separate plot was saved for every episode.
* ***Error vs. Time:*** Shows the average RMS error calculated every training loop. A separate plot was saved for every episode.
* ***Time vs.*** ***Episode:*** Shows the amount of time it took the robot to reach the done state in each episode. A plot was saved at the end of training.
* ***Reward vs.*** ***Episode:*** Shows the average reward accumulated per episode. A plot was saved at the end of training.

To allow for easier testing, the values of the weights in the neural net were saved off once training was complete. This allowed for any net to be loaded during the testing phase so comparison of results could be made

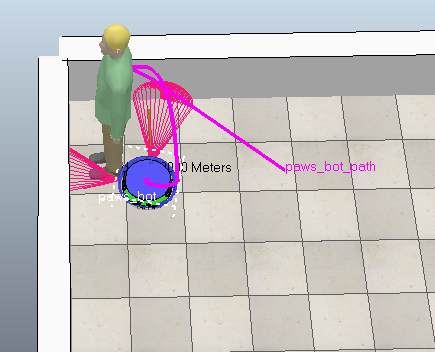
### Problems

Training utilizing CoppeliaSim as the simulation software proved to be extremely difficult. One of the downsides to using this software was the amount of time it took to run the episodes (~1 hour per 10 Episodes). The steps applied to the robot happen in real time which made training a high number of iterations very time consuming. The ability to vary the training and hyperparameters was limited due to the amount of time it took to run. Increasing the motor speed did not help speed up the time because it instead caused the robot to travel much farther than anticipated due to the rate at which commands are communicated through the API.

Another huge issue experienced with this software was its tendency to introduce anomalies that caused the robot to get flung around the map when the model was loaded, which typically ended with the robot being flipped upside down. This resulted in the training halting unexpectedly which ruined several overnight trainings that were planned. It took quite a bit of time to figure out what was causing the anomaly since it happened randomly, but the flipping was eventually observed and was attempted to be mitigated through code (i.e. killing the episode if this happens, changing the order and timing of moving models, etc.). These methods were all unsuccessful in resolving the issue. A snapshot of this anomaly is shown below. It shows the robot resting upside down after the resetting of the environment, as well as the erratic path it took, where it bounced off the wall, which is shown in pink.



5‑5: Upside Down Robot



‑:Error Anomaly

## Modified Design – OpenAI Gym

Regardless of the issues with CoppeliaSim, there was still a desire to test the created DDQN algorithm. A modified design was chosen which utilized OpenAI Gym and a third party environment called gym-maze which was listed in the Gym documentation (<https://github.com/openai/gym/blob/master/docs/environments.md#third-party-environments>). The DDQN algorithm was the same, and written in Python with the Keras and TensorFlow libraries used for the neural network portions.

### The Robot and Environment

The environment used was a 2D gridded map with a start and goal position connected via a maze. The map remained static throughout training and the goal position did not move. This specific environment was chosen because it seemed closest to the original environment that was planned and was easy to integrate into the existing DDQN code. Also the maze environment simulated walls/obstacles which was a desired part of the environment.

Due to the simplicity of the environment the robot is represented via a colored block. The robot can move north, south, east, or west which matched the original plan for the robot, so the action space remained the same. In this environment the robot only knows its current position on the map as grid coordinates. There is no sensor information to ingest which caused the state to reduce to only the 2 components.

The original reward function the simulation returned was a step function which was not sufficient for verifying the DDQN algorithm that was created. So the environment’s standard reward function was switched out for the one originally developed.

### The Algorithm

The same DDQN algorithm as previously described was used for the tests. All of the parameters remained the same aside from the training parameters, which were modified to account for the ability to run more episodes.

**Training Settings**

EPISODES = **1000**

TIME\_LIMIT = **250**

MEMORY\_CAPACITY = 1000

BATCH\_SIZE = 35

TARGET\_UPDATE\_COUNT = 25

### Data Collection

The data collection method in these tests was the same as previously described. The only parameter not collected was Distance vs. Time after determining that this data was not very valuable. Also an additional parameter collected was the reward using the original step reward function, so a comparison could be made between the two.

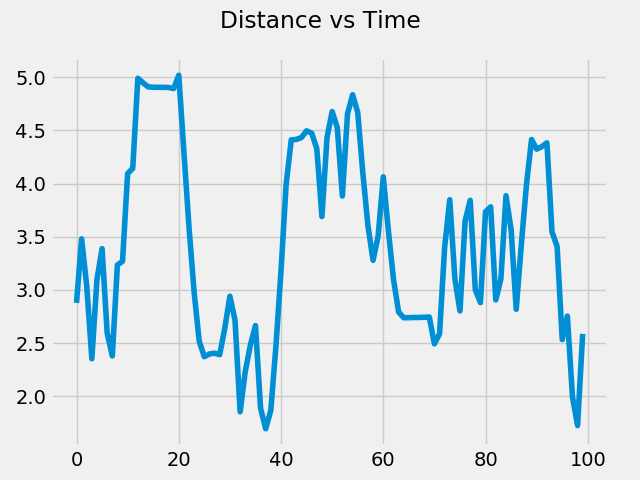
* ***Error vs. Time***
* ***Time vs.*** ***Episode***
* ***Reward vs.*** ***Episode***
* ***Reward Step vs Episode***

# Results and Analysis

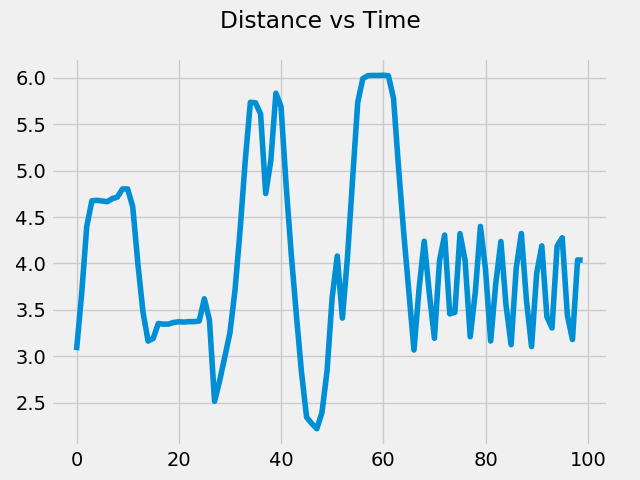
## CoppeliaSim – Original Design

### Training

Due to the amount of time it took an episode to run and the many errors experienced, the longest training run that was accomplished using CoppeliaSim was 40 episodes on the 2.5m x 2.5m map. During this training run the movement of the robot seemed to have a common feature of oscillating its distance to the goal. Which was the robot moving in “circles” on the map, constantly turning right or left.

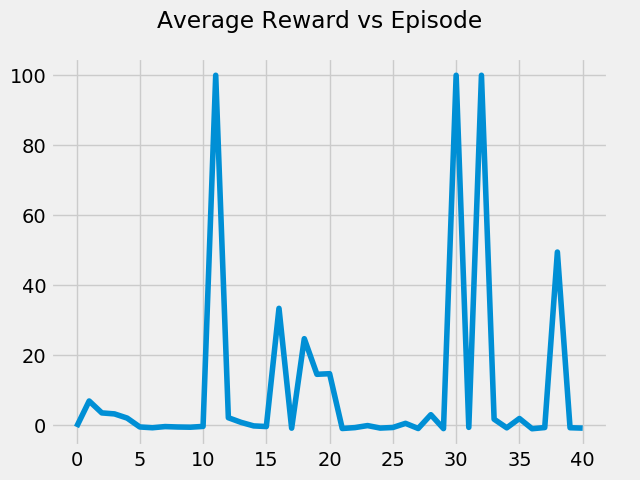


6‑1:Distance vs Time Episode 37

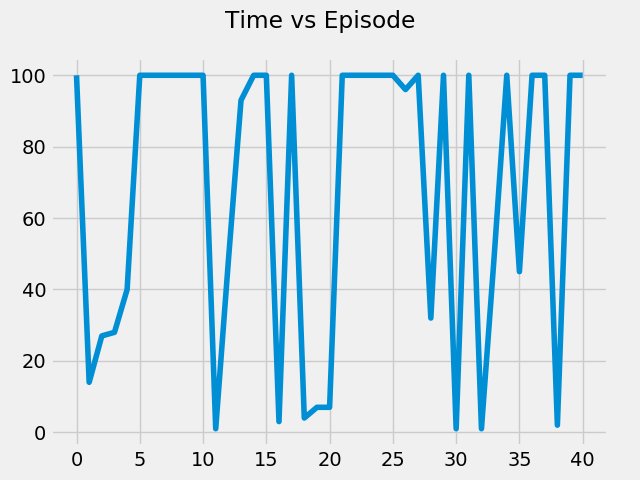


6‑2:Distance vs Time Episode 40

During this training the robot was able to reach the goal 15 times, which can be determined by looking at the vertices on the Max Steps per Episode plot below. The average reward per episode was variable throughout the training.



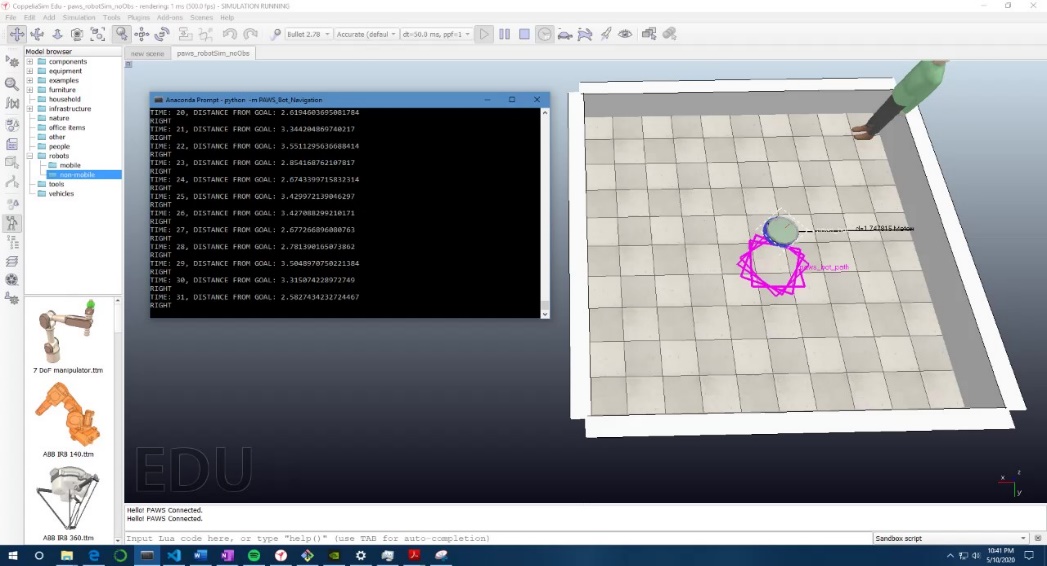
6‑3: Average Reward per Episode



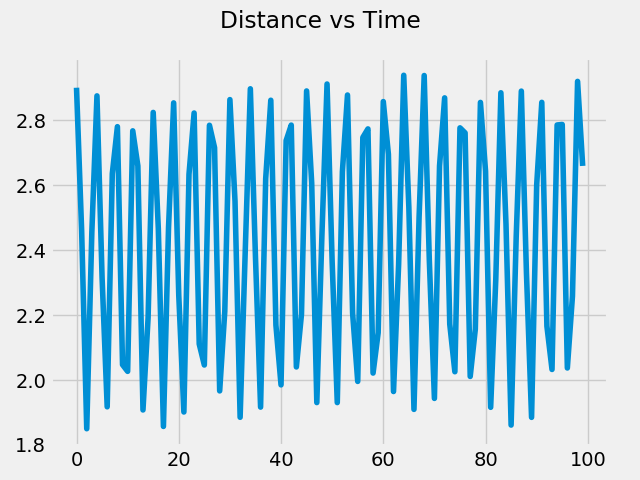
6‑4: Max Steps per Episode

### Testing

Testing was conducted on the network trained on the 40 episodes and in none of the test runs was it able to reach the goal (test accuracy = 0%). The robot continuously oscillated around its starting point turning in a circle until the time elapsed. This behavior is shown in the two images below.



6‑6: Test Run Capture



6‑5:Distance vs Time Test Run

### Analysis

The amount of training steps accomplished using the CoppeliaSim method was so low that it makes it hard to analyze the results. Similar work done in this area displays the need for thousands of training episodes on a static map (PlanningWithDeepReinforcementLearning.pdf) and tens of thousands of episodes on a dynamic map (Dynamic Path Planning of Unknown Environment .pdf) before the network converges to something reasonable.

The first major realization is that CoppeliaSim may not be the best simulation software for the training portion of the DDQN algorithm. The training should have been done in an environment that can render significantly faster. It was also discovered that there is a library called PyRep (<https://github.com/stepjam/PyRep>) which states to improve the speed of interaction with the simulation compared to using the Python remote API. Unfortunately the lack of familiarity with CoppeliaSim and simulation environments in general made it hard to introduce more efficient simulation means.

The state space for the robot to train in may have also been too large. Binning the distance vector between the robot and the goal was most likely a good approach, but increments of 0.10 may have still created too many possible states for the robot to be in when obstacle proximity and time were also added into the mix.

## Gym – Modified

### Training

The Gym model was trained for 1000 Episodes which produced the following results:

### Testing

### Analysis

# Conclusions

fml

# Future Work

There is still much work that needs to be done on the PAWS Navigation System before it can be considered marketable. The next goal is to implement the desired environment in Gym so that a more accurate representation of the state can be tested with this algorithm. This would also allow tens of thousands of training epochs to be ran which would most likely help the solution converge. It would be beneficial to run the training in the custom Gym environment and then run the tests in CoppeliaSim to simulate the robot running in the real world.

# Bibliography

Lei, X., Zhang, Z. & Dong, P. Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning. *Journal of Robotics* **2018,** 1–10 (2018).