Deep Q-Learning for Large Randomized Grid Worlds using LiDAR

1st Arthur Wang
College of Engineering
Boston University
Boston, U.S.A.
arwang@bu.edu

2nd Jason Ha
College of Engineering
Boston University
Boston, U.S.A.
jha24@bu.edu

***Abstract*—Robust mobile robot navigation is key for autonomous robots. When planning paths, many considerations have to be made such as physical constraints, finding the optimal path, avoiding obstacles and reacting to a dynamic environment. In this project, we seek to leverage a common strategy to teach robots tasks which is a deep Q-network (DQN). We train the DQN in a simple simulated environment modeled on real-world bird's-eye-view LiDAR point clouds. The grid world type environment is a 64x64 square grid and includes randomly placed obstacles and goal points. The goal is to allow the agent to reach any designated goal point within the environment regardless of the obstacle configuration.**

This project investigated several methods to improve the base DQN implementation for better obstacle avoidance. These included hyperparameter tuning, experience replay adjustments, different exploration strategies, different neural network (NN) architectures and inputs, and finally different learning strategies. Due to a combination of factors, Double Dueling Deep Q-Convolutional Neural Networks performed the best in terms of behavior generalization. This approach most effectively avoided obstacles while moving toward the goal point across various obstacle and goal point configurations.

***Index Terms*—Reinforcement Learning, Deep Q-Learning, Double Dueling Deep Q-Learning, Grid World, LiDAR, LIMO**

I. Introduction

Collecting and chasing balls during the practice of racket sports, such as table tennis, is a tedious task. Table tennis players can use hundreds to thousands of balls during practice or play and a portion of those will end up on the floor. This is especially vexing when the balls carry great velocity and spin, causing them to end up far away from the table.

We propose an autonomous robot to collect table tennis balls to automate this task. There exists many other ball collecting robots from both industry and research. One such product is TenniBot, an automated ball-picking robot designed for tennis courts [1]. It is capable of using unspecified, but likely more traditional, path planning algorithms to collect tennis balls. Additionally, Echo Robotics Golf Picker and Relox Range Picker are mainly targeted at golf driving ranges [2] [3].

Our platform is based on the AgileX LIMO Pro, a car-like robot with several features conducive for our application as can be seen in Figure 1. The LIMO Pro can be equipped with either traditional wheels, treads or mecanum wheels that enable omni-directional movement. It also has an EAI XL2 LiDAR which will be used to sense the environment. For this project, development and testing speed is accelerated by using a custom lightweight simulator that includes LiDAR with similar characteristics to the EAI XL2.

Fig. 1: AgileX LIMO Pro hardware schematic showing relevant sensors and hardware components such as the LiDAR [3].

Many robots, like the ones mentioned earlier, have autonomous path planning functions which use their LiDARs, RGB-D cameras or RGB cameras. A noticeable trend is the usage of traditional path planning algorithms such as A*, RRT and RRT*, genetic algorithms and many more [4]. While those methods are effective, their computational complexity increases rapidly depending on the environment size or how far into the future each path is calculated. They also require recalculation if the environment changes which, as mentioned before, can become quite expensive.

In this paper, we seek to adapt a Deep Q-Network (DQN) approach to this task of path planning for several reasons such as generalization, real-time, frame-by-frame path planning and more [5]. A classic problem in reinforcement learning is the grid world. It usually consists of a grid with locations marked as obstacles or goals. Usually, grid worlds are relatively small

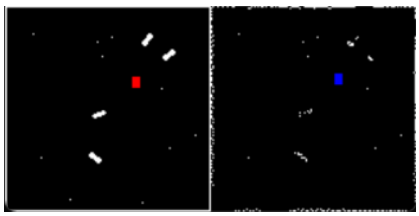


Fig. 2: Our 128 x 128 cell environment. Left shows the global map, the right shows the LiDAR point cloud. [3].

but we extend the grid world problem to a LiDAR occupancy map of size 64x64 and 128x128. Each cell in the grid map represents a 2 inch by 2 inch square covering a simulated 10.66 foot by 10.66 foot and 21.33 foot by 21.33 foot work space respectively as seen in Figure 2

This poses unique problems to basic DQN implementations. As the number of cells in the grid increases, the farther the robot can perceive. However, this also increases the number of possible states, complicating the learning process and raising the computational load on the neural network. As such, we leverage a property of LiDAR point clouds detailed in Subsection III-B to surmount this.

To address the greater need for learning and exploration ability, we utilize several methods to overcome these issues. These include, but are not limited to, time-depending environment modifications, and usage of more advanced DQN approaches such as Double Deep Q-Networks (DDQN) and Double Dueling Deep Q-Networks (D3QN) and a special input format.

II. Related Works

Deep Q-Learning was proposed by Mnih et al. to play Atari [6]. It is an extension of Q-learning that replaces the Q-table with a neural network (NN) [6]. A Neural network can be used as a universal function approximator and applied to Q-learning in this way. However, unlike the original Q-table approach, the DQN method does not guarantee to converge. Neural networks can catastrophically forget past lessons and cause learning divergence. Despite this, DQN promises greater generalization and flexibility over a Q-table. It uses the Bellman function to propagate rewards and update the Q values for learning. This equation can be seen in Equation 1 where the policy predicts the target Q values of the next state.

$$Q(s_t, a_t; \theta_{target}) = R_t + \gamma(maxQ(s_{t+1}, a; \theta_{target})) \quad (1)$$

The γ term represents the discount factor, which is a mathematical trick to converge infinite horizon problems by multiplying by values less than 1. As each iteration of the Bellman equation is computed, a γ value of less than one will diminish the rewards over time. By modifying this hyperparameter, it is possible to change how much the learning process values future rewards. A γ value closer to 0 makes the agent more myopic, focusing on immediate rewards, while a value less than but closer to 1 makes it more far-sighted, emphasizing long-term rewards.

Note that the θ is denoted to be the target network. We are using a second target network that remains fixed to help reduce overestimation when estimating the Q values for the next state. Once the Q values for each action are calculated, they can be used to generate the loss needed to perform error backpropagation on the policy neural network. A popular loss function is Huber Loss as seen in Equation 2 where "a" is $Q_{t\_policy} - Q_{t+1\_target}$. This original DQN paper serves as our baseline learning method, around which further improvements to our path planning policy is built.

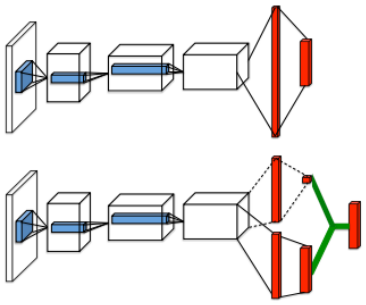


Fig. 3: Adapted from Figure 1 of Wang et al. [8]. Difference between single stream Q-network (top) and a dueling Q-network (bottom).

$$L_\delta(a) = \begin{cases} \frac{1}{2} * (a)^2, & for |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta), & otherwise. \end{cases} \quad (2)$$

To help with learning convergence, Hasselt et al. proposed using a second target network to create the Double DQN (D2QN) [7]. This necessitates a modification to the Bellman equation which can be seen in Equation 3.

$$\begin{aligned} &Q(s_t, a_t; \theta_{target}) = R_t + \\ &\gamma Q(s_{t+1}, argmaxQ(s_{t+1}, a; \theta_{policy}); \theta_{target}) \end{aligned} \quad (3)$$

The target network's Q values of the next state are selected using the actions outputted by the policy network. The rationale is that the original Bellman update overestimates the Q values by cyclically updating them and then using them to perform actions. This can explode the importance of undesirable actions and prevent learning the optimal behavior. The target network is setup to be fixed compared to the policy network, providing a reference point. Our project utilizes the double training method to further improve upon the base DQN's learning ability.

Wang et al. developed the concept of the Dueling Deep Q-Network (DuelingDQN) to improve the ability of DQNs by decoupling the estimation of state Value and action Advantage in the neural network [8]. This separation helps generalize learning across different actions without changing the learning process [8]. This architecture can be seen in Figure 3 and is used in our project for a few reasons. The Value stream learns the value of the input state without regard to the actions while the Advantage stream learns the advantages of doing actions to the input state. This is useful where multiple actions would have similar Q values such as our grid world application. For instance, if the goal is northeast of the agent, Manhattan distance would dictate that moving either up or right will yield the same reduction in distance, assuming no relevant obstacles affect that calculation. Both of those actions would theoretically have the same Q value. When combining Value and Advantage to calculate the Q values, the advantage scores of each action are mean-zeroed. As such, actions with similar advantage scores will have values closer to zero, diminishing their impact on the Q-value, as seen in Equation 4.

$$Q(s_t, a_t; \theta) = V(s_t) + A(s_t, a) - \frac{1}{n} \sum_{a} A(s_t, a) \quad (4)$$

The paper by Yin et al. uses a rewards based e-greedy exploration strategy with a Double Dueling DQN (D3QN) called RND3QN to better learn grid-based path planning around obstacles [9]. Their RND3QN method is able to increase success rates by over 60% in some tests. Although successful, this method proved difficult to extrapolate to a larger grid world in our simulations, which we discuss more in Subsection III-B. However, the learning stability and ability of D3QN over other DQN methods led us to adopt it as our ultimate method. This is especially useful given their usage of local mapping.

Some related work can be found in Wang et al.'s paper on the kiwi fruit picking robot [10]. Similar to our simulated LIMO Pro agent, Wang et al. utilize a mobile robot and a LIDAR sensor for DQN. They are able to reduce the distance and time costs of covering a LIDAR point cloud map by constantly retraining on the most successful trajectories. Their results show a 31.56% reduction in distance and a 35.72% reduction in time needed to perform coverage in contrast to a classic Boustrophedon algorithm. While successful, their method required a pre-mapped environment and is not applicable to a 4-d movement type robot. However, their approach of training on successful trajectories to reinforce positive learning, coupled with training techniques from the next paper discussed, helped improve the performance of our D3QN.

Finally, Lei et al.'s work in creating a simulated Pygame environment to train a D2QN overlaps with our proposed approach [11]. Like Lei et al., we simulate a local dynamic environment and create a virtual LiDAR to train the our DQN, D2QN and D3QN policies. We adopt some of their training techniques as they work in a grid environment with randomly placed goal points and obstacles as well. The specific techniques are discussed in Subsection III-B. Additionally, they make use of a CNN to process pre-processed LiDAR point clouds padded with the goal position and agent position which we modify in our D3QN.

III. Methods

A. Simulator

The simulator is visually represented as a PyGame window with a resolution equal the size of the grid environment being used. All elements in the environment are represented by integer positions, while the agent's exact position and orientation are internally represented by floating-point numbers. This approach aligns with the occupancy grid representation of the world while maintaining floating-point precision for mathematical processes. This was particularly important when the agent's action space was modeled after a tank or car, where it could rotate clockwise and counter-clockwise and move forward. When the action space was converted to 4-directional, D-pad movement (Up, Down, Left, Right), the need for floating point pose precision became redundant. More details can be found in Subsection III-B.

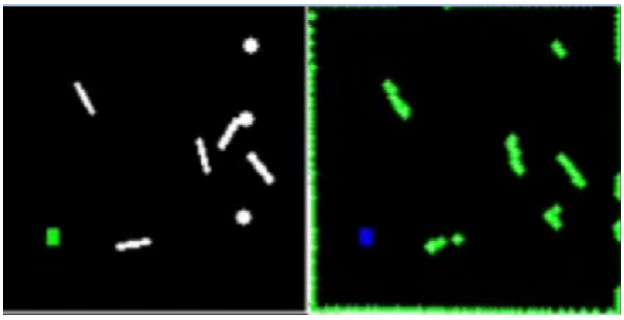


Fig. 4: Different views of the simulator environment. Global Map with Agent (Left), LiDAR Occupancy Map with Agent (Right)

The simulated LiDAR is modeled after the EAI XL2 LiDAR found on a LIMO Pro, providing a 2-D scan with 360 points. The frequency of each scan is set to once per environment step for simplicity, and the agent remains static while the scan takes place. To enhance modeling accuracy, Gaussian noise is applied to each detection, ensuring that the agent does not have perfect detections.

Obstacles are represented as circles/dots and lines, and are randomly generated. Circles or dots model items in the environment with a circular cross-section, such as posts, chair legs, people's legs, and similar objects. Lines are used to represent objects with some width such as boxes. The entire outer edge of the environment is also designated as a obstacle. These details can be seen in Figure 4 with obstacles and LiDAR view.

B. Deep Q-Learning & More

As already stated, our main point of comparison is the performance difference between a base DQN and the D3QN. It is expected that the D3QN would be able to perform better. The DQCNN architecture in Lei et al. [11] was used as a starting point, which can be seen in Figure 5. Instead of providing the occupancy grid as input to the model, the point cloud is reshaped into a 20x20x2 image where the channel dimension contains the relative X and Y coordinates of each

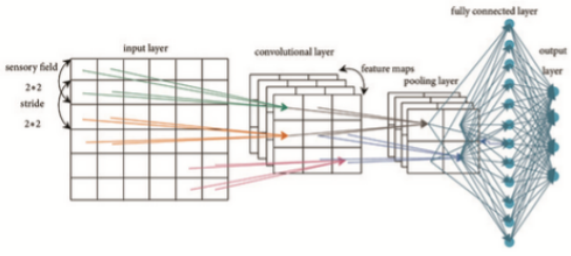


Fig. 5: Overall network architecture for our D3QN. Adapted from Figure 2 in Lei et al. [11].

$$R = \begin{cases} r = -1.0, & \text{if } collided. \\ r = max(1.0, init\_man\_dist), & \text{if } goal\_reached. \\ r = min(1.0, init\_man\_dist \div (cur\_man\_dist + 1)) * 0.02, & \text{if } cur\_man\_dist < prev\_man\_dist. \\ r = -0.02, & \text{if } cur\_man\_dist \geq prev\_man\_dist. \end{cases} \quad (4)$$

$$R = \begin{cases} r = min(-1.0, -init\_man\_dist), & \text{if } collided. \\ r = max(1.0, init\_man\_dist), & \text{if } goal\_reached. \\ r = min(1.0, init\_man\_dist \div (cur\_man\_dist + 1)) * 0.02, & \text{if } cur\_man\_dist < prev\_man\_dist. \\ r = -0.02, & \text{if } cur\_man\_dist \geq prev\_man\_dist. \end{cases} \quad (5)$$

detection. This allows the model to take in input from any size grid world without increasing input complexity and size. Since the point cloud only contains 360 points, the rest of the image is padded with 40 copies of the agent's relative position to the goal. This contrasts with Lei et al.'s input, which is padded with 20 copies of the goal position and 20 copies of the agent's position. Both approaches present the model with the relative positions of obstacles and knowledge of the goal position with respect to the agent, so either approach should work.

Several other model architectures were tested as well due to errors in programming causing incoherent learning, which are further discussed in Section IV. The different architectures trialed include a simple fully connected neural network for debugging, and an LSTM for memory of the path taken. Additionally, different inputs were tested including sequences of 3 states being input at a time to either the neural network, CNN, or LSTM. However, the most successful model architecture combines the CNN with the dueling architecture, as seen in Figure 6a. Note the large number of parameters; this was used, along with other methods, to prevent model catastrophic forgetting that was experienced in earlier iterations.

Initially, the rewards for the D3QN were sparse, granting a reward only upon reaching the goal. Otherwise, the agent was penalized a flat amount for moving and a larger amount for colliding or running out of time. This approach conflicted with Yin et al.'s reward-based e-greedy implementation [9]. Essentially, if the episode rewards increase, the e value is decreased, scaled by fraction of total time steps completed as seen in Equation 6.

$$\epsilon = (\epsilon_{init} - \epsilon_{min}) * max(\frac{N - n_{step}}{N}, 0) + \epsilon_{min} \quad (6)$$

Due to the large grid representation, it was very rare for the agent to reach the goal, making positive rewards extremely sparse. As a result, the average episode rewards never increased, preventing the reward-based e-greedy method from decreasing the e value.

Another method attempted to allow sparse rewards to work was our own implementation of Hindsight Experience Replay (HER) [12]. In addition to the original HER method, which substitutes the final state of an episode for the original goal point and recalculates rewards, we tested using intermediate states to replace the goal point and recalculate rewards. This would increase the proportion of successful positive rewards in the replay buffer to increase learning speed and ability. However, quantifying the success of this method was difficult, as it was used during a period when programming issues caused artificially poor performance.

Ultimately, we decided to provide densely shaped rewards based on the original Manhattan distance to the goal and the current Manhattan distance as seen in Equations 4 for DQN and 5 for D3QN and Algorithm 1. The reason was because this provided more rewards with a better measure of agent progress. The difference in reward functions was a mistake during training, which skews the episode rewards for the DQN higher than they should be. The actual average episode rewards should be negative due to the large amount of collisions and time-outs for the DQN. This can be seen later in Section IV. In either case, the reward is capped to positive and negative 20.

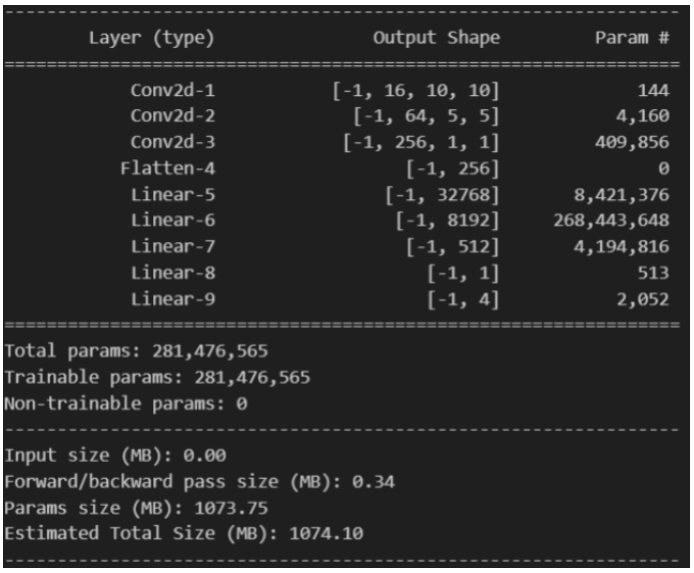
To complement the dense rewards and improve learning speed, the environment also froze every n episodes. In theory, this would allow the agent to encounter the goal position more, especially as the agent begins to start using expert actions more than random ones. This would increase the proportion of goal state rewards in the replay buffer.

Behavior pre-cloning was also tested to see if it could improve the agent's performance. We trained the agent on a dataset of expert actions in varied environments before allowing it to do reinforcement learning. While successful, it was ultimately unused as part of our best performing D3QN implementation, as the D3QN was able to learn the same behaviors without it.

In conclusion, the methods used for our main experi-

Algorithm 1 Goal Euclidean and Manhattan Distance Generation

1: **Input:** t_{cur} (current timestep), t_{tot} (total timesteps), g_w (grid width)
2: **Output:** Distance to Goal in Euclidean and Manhattan Distance
3: $EDist \leftarrow rand(10, 10 + (min(0.25 * t_{tot}, t_{cur})/(0.25 * t_{tot})) * g_w/3)$
4: $\theta \leftarrow rand(-\pi, \pi)$
5: $XDist \leftarrow EDist * cos(\theta)$
6: $YDist \leftarrow EDist * sin(\theta)$
7: $MDist \leftarrow abs(XDist) + abs(YDist)$



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 10, 10]	144
Conv2d-2	[-1, 64, 5, 5]	4,160
Conv2d-3	[-1, 256, 1, 1]	409,856
Flatten-4	[-1, 256]	0
Linear-5	[-1, 32768]	8,421,376
Linear-6	[-1, 8192]	268,443,648
Linear-7	[-1, 512]	4,194,816
Linear-8	[-1, 1]	513
Linear-9	[-1, 4]	2,052

Total params: 281,476,565
Trainable params: 281,476,565
Non-trainable params: 0

Input size (MB): 0.00
Forward/backward pass size (MB): 0.34
Params size (MB): 1073.75
Estimated Total Size (MB): 1074.10

(a) Model Hyperparameters

Hyperparameter	Value
T-Steps	1,000,000
Buffer Size	8,000,000
Exp. Fraction	0.3
Exp. Final e	0.15
n-steps	3
Batch Size	512
Gamma	0.999
Hard Update Frequency	50,000

(b) Training Hyperparameters

Fig. 6: The left image shows model architecture and summary of our D3QN and the right shows the training hyperparameters used for both the DQN and D3QN.

ment are DQN and D3QN using standard experience replay, e-greedy exploration, densely shaped rewards, 4-directional movement, and a CNN model trained in a simulated randomized environment. This environment included features such as obstacle freezing, timestep-scaled goal spawn distance and noise obstacle detections. Additional methods explored include Hindsight Experience Replay (HER), behavior pre-cloning, LSTM models, and sequential states.

IV. Results

A. Experiment Setup and Questions

Our main experiment was comparing the performance of a DQN and a D3QN agent using similar neural network architectures and the best learning hyperparameters found for each. Additionally, the overall learning ability is analyzed on several test environments. Secondary experiments included analyzing the learning process for methods to prevent catastrophic forgetting and handle sparse rewards.

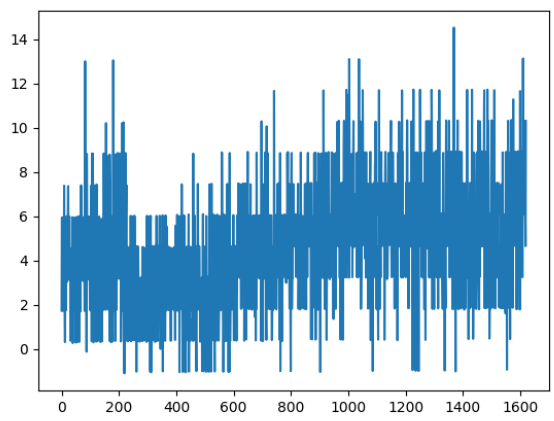
Additionally, original proposal sought to answer the following questions:

1) What movement strategies do DQN or re-DQN learn to perform with their policy when driving between waypoints? Wang et al. report a reduction in coverage distance and time spent; how does DQN or re-DQN minimize distance or time in our application[1]. Does it prefer straight paths or curved and what margin of collision does it provide for? To answer this we will plot the paths the policy takes and qualitatively analyze them.
2) How well does the policy deal with dynamic obstacles such as a walking person or an object that suddenly drops into the LIDAR FOV? What sort of reaction time does the policy possess. To investigate this we will test the simulation trained policy by adding, removing and moving occupancy as the simulated LIMO performs actions. This is relevant for people that might walk into the robot's FOV.
3) What challenges are there with transferring the policy from the simulated environment to the physical LIMO? Does the LIMO experience more collisions with obstacles than in the simulation? To test this we will attempt to use the policy on a physical LIMO vehicle if time permits. We will then compare the amount of collisions the policy experienced in physical testing versus similar simulated environments? We will then attempt to fine-tune the performance of the policy such that it performs acceptable in the real world. We will measure how many episodes it takes to converge to acceptable behavior.

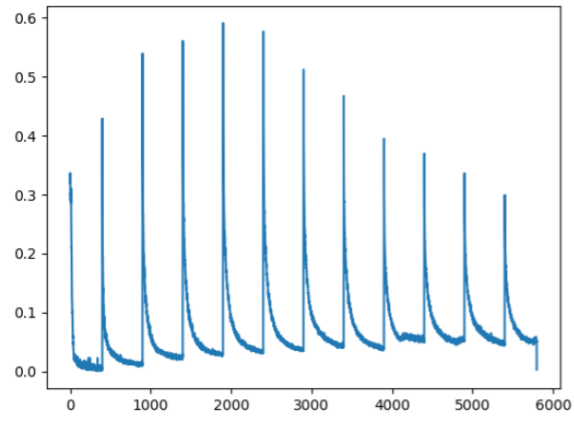
However, due to time constraints, only a few could be fully addressed while most had to be modified. They are as follows:

1) **What movement strategies do DQN or re-DQN learn to perform with their policy when driving to the goal and avoiding obstacles?**
2) **Since we use 4-directional movement, does the policy prefer completing all horizontal or vertical movements before switching? Or does it prefer to "zigzag" its way to the goal position.**
3) **How many learning time-steps or episodes does it take for the DQN to converge vs. the D3QN. What are the rewards at convergence?**

The main metrics for analyzing each method's performance is its graphs of average episode rewards and episode losses

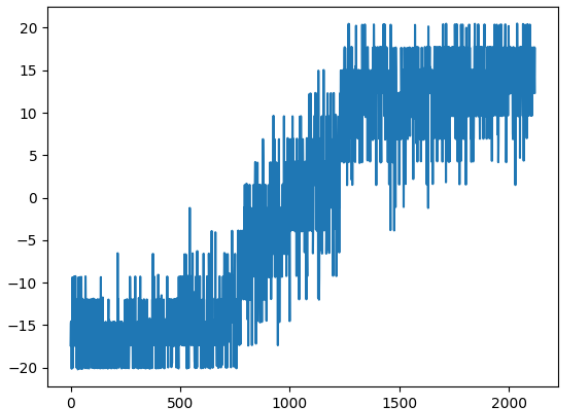


(a) Episode Rewards

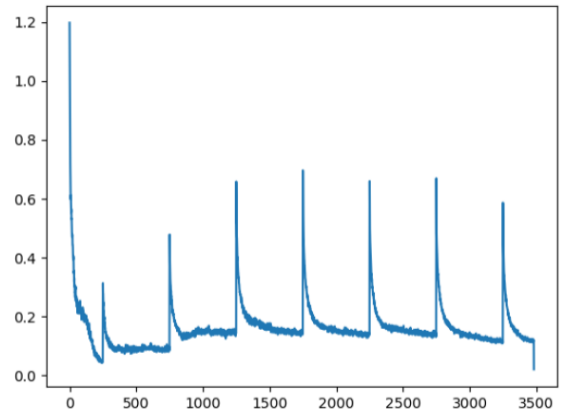


(b) Loss

Fig. 7: Training episode rewards and loss using basic DQN. Collision punishment was -1.0 and goal reward was scaled by original goal distance. Consequently, reward graph is skewed positive.



(a) Episode Rewards



(b) Loss

Fig. 8: Training episode rewards and loss using D3QN. Collision punishment was the negative of the goal reward and the goal reward was scaled by original goal distance.

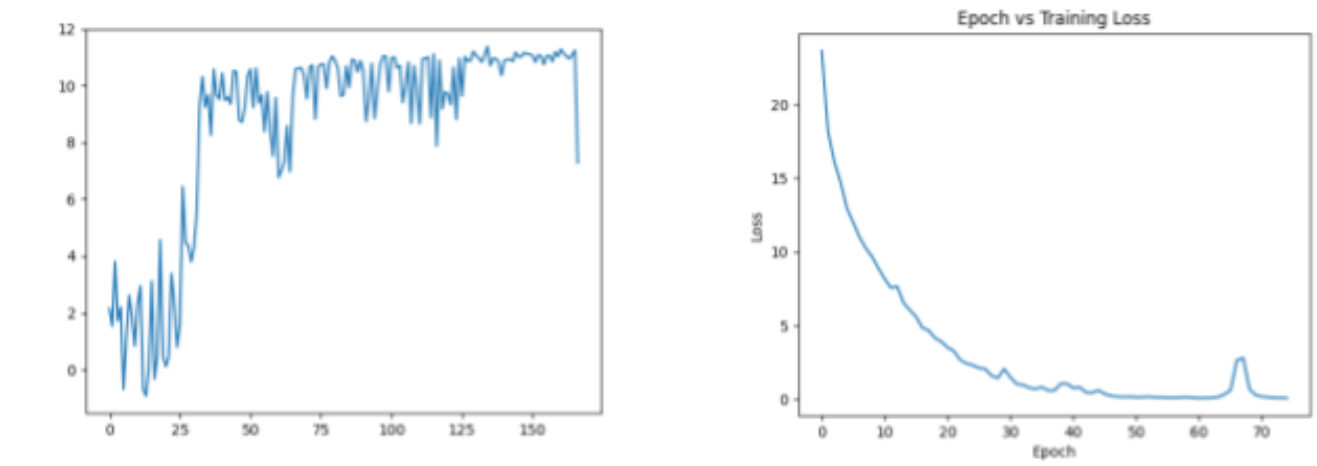
during training. For the main experiment, the average of policy evaluation rewards are used as well. For Question 1 and 2, the movement strategy and behavior is qualitatively analyzed. For question 3, this is answered by the learning graphs.

Data for DQN training is provided in real-time by the simulator environment as detailed in Subsection III-A. On the other hand, data for behavior pre-cloning was collected from the simulator environment when the agent was controlled by an expert. The collected expert datasets contained 5812 and 2386 LiDAR scans for single state input and sequential stacked states respectively. Data collection, training and evaluation was carried out on both Boston University's Shared Computing Cluster and Personal Work Devices.

B. Results & Analysis

1) Question 1: Movement Strategies: The DQN does not perform well and jitters in place. This most likely is a local policy minima issue where the agent believes that this is optimal. However, when it does move, it seems to get uncomfortably close to obstacles and occasionally collides with them. Since the performance is poor, the movement is best described as a mix of half random and half expert actions.

In contrast, the D3QN performs much better but also jitters a little under certain circumstances. However, it seems to grant a slightly larger margin when passing by obstacles. The path



(a) Episode Rewards

(b) Classification Training Loss

Fig. 9: Training episode rewards and classification loss using basic DQN with behavior pre-cloning. This was without using the scaled dense rewards.

it takes is optimal a portion of the time, as measured by the Manhattan Distance, meaning the exact number of steps matches the Manhattan distance.

2) Question 2: Zigzag vs. Continuous Action Sequence: The action space for both DQN and D3QN use a 4-directional movement style. When coupled with Manhattan distance, moving to a goal northeast of the agent can be accomplished by either "zigzagging" by alternating up and right movements or by completing all upward movements first followed by all rightward movements (or vice versa), assuming no obstacles are present.

There did not appear to be any clear pattern in how the DQN chooses between action sequences. This is especially evident when the goal is directly inline with the agent, as it often zigzags even when a straight path is possible.

In contrast, the D3QN applies its zigzagging and continuous movement proportions more logically. When the goal lies directly along the agent's X or Y axis, the agent typically moves continuously in that direction. Conversely, it uses diagonal movements more frequently when the goal is off-axis.

3) Question 3: Convergence Time: Looking at Figure 7, the time it takes to converge to a suboptimal policy is around 1000 episodes for the DQN. The reward it converges to is around 5.5 but this is skewed as mentioned earlier in Subsection III-B.

For the D3QN, the convergence time is approximately 1600 episodes at an average episode reward of 14 which is much better than the DQN.

4) Experimental Analysis: As established in Question 1 and 2, the baseline DQN has extremely poor performance. The agent learns to jitter in place for most environment cases. Looking at the average episode rewards in Figure 7a, it can be seen that the policy is not substantially better than when it began learning. Due to the aforementioned error in reward magnitude in Subsection III-B and Equation 4, the graph is skewed more positive than it should be. Originally, we wondered if this impacted the learning and consequently the performance of the policy. After much consideration, we decided that even if a large negative reward for collision was provided, the behavior would not be different. This is because the policy opted to minimize its punishment by maximizing its episode time steps and then timing out rather than immediately colliding with a wall for a small punishment. During evaluation across 10 environments, the average episode reward it achieved is -0.117, indicating it typically moved briefly before timing out. Overall, its performance is considered poor in all cases.

In comparison, the D3QN performed very well as can be seen in Figure 8. It started with substantially negative average episode reward possible of approximately -17 and ended with almost the highest possible average episode reward of 14. This is impressive because each action reduces the reward by 0.02 and episode reward is capped at positive 20. As such, the agent is able to reach the goal the most of the time using a reasonable path during training. When evaluated using the same environments as the base DQN, the D3QN was able to achieve an average reward of 26.309. This is calculated without the reward cap.

In general, D3QN went through many iterations with respect to training techniques. Initially, this method also struggled with achieving a higher average reward. The combination of factors that enabled this better performance are, in no particular order of importance: larger model size, frozen environments, and much larger replay buffer. Unfortunately, there was not enough time to perform a proper ablation study of these components.

Behavior cloning was also relatively successful as seen in Figure 9. Using the expert, single frame, dataset, the policy learned to navigate to the goal quite well. In Figure 9a, it can

be seen that the policy has some dips in the average episode rewards. This is likely due to the environment changing in ways that the model did was not acquainted with. As the policy adapts to the live environments, the rewards converge tightly to around 11. For the different reward structure used on this test, this was quite good. However, during evaluation the cloned policy also jittered and fell short of reach the goal as well.

Testing with the LSTM model yielded no apparent benefit, however this could be due to using smaller layers and a smaller replay buffer. The same applies to input state stacking with and without the LSTM.

Overall, the proposed D3QN method outlined in the conclusions of Subsection III-B performed much better than the base DQN method as hypothesized.

V. Conclusion

From our project, we concluded that learning to navigate a 128x128 grid world is challenging for our agent without certain performance enhancers. A grid of this size hinders the robot's ability to consistently reach goals and earn positive rewards through random movements in a sparse reward format. Additionally, randomized obstacles further complicate training when the environment changes with each training episode. This can cause the agent to accrue too many negative rewards in the replay buffer to effectively learn to seek out positive rewards. To address this issue, we found it helpful to periodically freeze the environment. Through our experiments with both DQN and D3QN, the results confirmed that D3QN with certain training enhancements outperforms DQN, as anticipated, by learning the correct behaviors more effectively.

For future work, we plan to explore different learning methods, such as Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). These are especially relevant because the original action space was a discretized representation of a car. However, this proved difficult for the original DQN to learn and was abandoned in favor of 4-directional movement. Using PPO or SAC can enable continuous action spaces which are key to smooth driving. For the LIMO Pro, 4-directional movement is possible using the mecanum wheels. However, the movement is not very precise and slippage may occur. As such, being able to transition to using continuous steering and throttle values will enable more accurate locomotion.

Additionally, a more sophisticated exploration strategy should be considered. Freezing the environment for a set of episodes is used to compensate for large distances to the goal causing infrequent successes and constant collisions. Using an exploration strategy beyond just random movement could help such as randomly selecting an action based on some heuristic.

Furthermore, we aim to introduce dynamic obstacles, as all our current tests are conducted in a static environment. This was one of the originally stated environment components that was left out due to the need to simplify the simulation and learning process.

As we continue to develop our method, we also plan to apply our work to a real LIMO robot and attempt to transfer the policy to reality. This would necessitate the inclusion of other autonomous modules such as SLAM to make sure agent movements are precise like in the simulator. Additionally, a visual goal detection module using CNNs will need to be developed to create a modular autonomy pipeline.

References

[1] Tennibot.
[2] E. Robotics.
[3] R. Robotics.
[4] K. Karur, N. Sharma, C. Dharmatti, and J. E. Siegel, "A survey of path planning algorithms for mobile robots," *Vehicles*, vol. 3, no. 3, pp. 448–468, 2021.
[5] A. I. Panov, K. S. Yakovlev, and R. Suvorov, "Grid path planning with deep reinforcement learning: Preliminary results," *Procedia Computer Science*, vol. 123, pp. 347–353, 2018. 8th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2017 (Eighth Annual Meeting of the BICA Society), held August 1-6, 2017 in Moscow, Russia.
[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.
[7] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.
[8] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016.
[9] Y. Yin, Z. Chen, G. Liu, and J. Guo, "A mapless local path planning approach using deep reinforcement learning framework," *Sensors*, vol. 23, no. 4, 2023.
[10] Y. Wang, Z. He, D. Cao, L. Ma, K. Li, L. Jia, and Y. Cui, "Coverage path planning for kiwifruit picking robots based on deep reinforcement learning," *Computers and Electronics in Agriculture*, vol. 205, p. 107593, 2023.
[11] X. Lei, Z. Zhang, and P. Dong, "Dynamic path planning of unknown environment based on deep reinforcement learning," *Journal of Robotics*, vol. 2018, no. 1, p. 5781591, 2018.
[12] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," 2018.

APPENDIX

Work for this project was done in order of creating the simulator and then testing and debugging the DQN algorithms. The simulator was completed with input from both of us. This included the simulated LiDAR, environment update functions, and spawning algorithms. The base code for the DQN algorithm is borrowed from our Homework 3 implementation.

Arthur handled the implementation of new models, hindsight experience replay, training DQN and D3QN on the SCC and hyperparamter tuning.

Jason was in charge of debugging the simulator, collecting expert data and porting and integrating the behavior cloning code from Homework 1.

The presentation and report were completed jointly with an approximately equal contribution. Arthur wrote the theory and methods section and most of the results. Jason completed the introduction, parts of the results, the conclusion and future work sections.