Deep Reinforcement Learning For Autonomous Mobile Agent Indoor Navigation

Trevor Sherrard
Rochester Institute of Technology (RIT)
Dept. of Electrical Engineering
Rochester NY, USA
tws4129@rit.edu

***Abstract*—Autonomous agent navigation within an unknown environment is a topic of great interest to several robotics researchers and engineers alike. Many map-based particle estimation techniques exist to help solve this problem (including but not limited to various implementations of Simultaneous Localization and Mapping (SLAM) algorithms). Many of these SLAM algorithm implementations often rely on storing several copies of the environmental map representation within system memory, and require knowledge about the robot's drive kinematics and physical geometry. Here, a Deep Q-Learning Network (DQN) based implementation is proposed to allow an agent to learn how to navigate to a goal position without any prior knowledge of its environment making use of a punishment/reward paradigm. The model is trained using a multi-stage training environment paradigm, in which each training environment is more complex than the previous. The autonomous agent and resulting trained DQN model are evaluated in terms of performance in a physics-consistent simulation environment.**

***Index Terms*—DQN, machine learning, robot navigation, reinforcement learning**

I. Introduction

The challenge of an autonomous agent navigating through an unknown and dynamic environment is a challenging problem to solve. Various methods and techniques are available to robotics engineers and other technical professionals in the field. One commonly used method that is widely available in several open source software frameworks for mobile robotics is that of Simultaneous Localization and Mapping (SLAM) Gmapping [1]. This method was designed for use within the Robot Operating System (ROS) message passing framework. The algorithm takes in laserscan data from LIDAR devices, as well as odometry messages and produces an estimate of the agent's position within an occupancy gridmap representation of the environment [1]. This estimation is done using the aforementioned sensor data in conjunction with a Rao-Blackwellized particle filter [1]. This method relies on knowing robot geometry and holonomic constraints and has high space complexity, as each particle within the particle filter carries a copy of the estimated occupancy gridmap state. To overcome these constraints, it is desirable to teach an autonomous agent to navigate towards a goal by maximizing its reward function and minimizing its punishment function per each training episode.

This approach offers a distinct advantage over the traditional use of SLAM localization and mapping algorithms. The agent needs only understand what is "off limits" (i.e getting within 2 feet of obstacles or colliding with them) and what actions are desirable (navigate to within an acceptable radial distance of the goal, decrease velocity around dynamic obstacles). While the agent will still need to maintain the location of the goal state, it will not need to make use of error prone and spatially complex particle models for state estimation.

Firstly, a review of other novel and tangential academic works within this field will be examined as a motivation of this work. Secondly, an outline of the methodology used for the simulation based, episodic reinforcement training, model validation, and simulation to real world agent transition process will be examined. Lastly, experimental data outlining agent performance in the aforementioned simulation environment will be examined and conclusions about the overall viability of the methodology in question will be made.

II. Literature Survey

Map-free goal finding and environment navigation can be implemented for a single agent, or for several agents exhibiting swarming behavior. O. Tuzel et al., outline the application of leader based goal-finding navigation for swarm robotics [1]. The leader agents within the group knew the location of the goal, while the followers were naive to it [2]. Leaders were able to influence the overall group movement and trajectory by moving within the group as a whole [2]. It was found that increasing the number of leaders among the group itself led to the goal-finding task to be completed quicker [2]. While the problem being examined in this work does not involve swarm behavior, the work of Tuzel et al. is important in understanding rudimentary concepts of map-free navigation techniques.

One of the primary drivers for the use of Deep Reinforcement Learning (DRL) within the problem domain of robot navigation is that conventional deep learning techniques (DNN's) often fall short here. DNN's on their own require a massive amount of labeled data for their training process, which can be difficult to obtain in a timely manner within this problem domain [3]. Additionally, the DNN model must be trained completely, and then evaluated. If the model does not perform to the desired specifications, model parameters must be tuned and the entire training process must be repeated [3]. This is a computationally and temporally costly endeavour and is undesirable overall. The work of S. Han et al. investigates the effectiveness of using a Deep Q-Learning model for robot goal-finding and navigation [3]. Within this work, the model was trained through the use of a punishment/reward paradigm in which navigating to the goal position was rewarded, while collisions with obstacles were punished [3]. The robot was rewarded with a +4 weight upon finding the goal, and punished with a -1 weight upon a collision with an obstacle [3] and 0 in the case that neither of these cases occurred [3]. The action of navigating to the goal was rewarded more than the action of colliding with an obstacle was punished to allow the agent to prioritize moving towards to goal over minimizing obstacle collisions [3]. The agent was trained, and had its overall performance evaluated, within a virtual Gazebo environment [3]. This work is important as it illustrates the practicality of using DQN models for autonomous agent navigation within a previously unknown environment. This work will attempt to use a multi-stage training environment paradigm to achieve similar results. The effectiveness of the generate model used to control the virtual Turtlebot 3 agent will be evaluated in both the Gazebo simulation environment.

Another key work within the area of unsupervised, reinforcement learning for mobile robot navigation is that of N. Duo et. al [5]. The authors develop a reinforcement learning model for a physical TurtleBot using Proximal Policy Optimization (PPO) Algorithms [5]. This work uses a unique reward structure for the reinforcement learning process based off of the distance of the robot to the nearest obstacle, the distance to the target location and the time spent searching for the goal [5]. Using a more complex reward structure can allow for the agent to maximize rewards for safely, efficiently and quickly navigating to the goal location [5]. The reward model used in this work is not as complex as that of the model used by N. Duo et. al. However, the model proposed here does create incentives for the agent to stay away from static obstacles, thus increasing the safety of the overall system.

The work of B. Moridian et al. examines the use of Deep Q-Learning for use in the applications of rescue robotics path planning [4]. In these scenarios, environments are constantly changing, so effective and responsive path planning and obstacle avoidance is required. In these specific situations, traditional navigation and path planning methods can fall short in terms of performance [4]. The authors make use of Deep Q-Learning models fused with Long-Short Term Memory (LSTM) models to retain information obtained from their episodic training process [4]. The authors used two different vehicle kinematic models during their training processes to observe how this changed the overall effectiveness of learned robot navigation ability [4]. It was found that a model that included inertial damping for the forward/reverse direction and rotation did indeed perform differently than a model that exhibited no inertial dampening [4]. The presented work will attempt to further the investigation on the effects of different robot drive kinematics on overall agent goal-finding performance, as the work of Moridian et al. used a Husky AGV platform, while the work done here will make use of the Turtlebot 3 platform [4].

Overall, the major limitations of the current academic work on Deep Q-Learning for mobile robot navigation primarily include the comparison of performance of DQN models trained within a virtual environment in relation to the performance of the same model on a physical robot implementation. Additionally, the process of modifying DQNs and the relevant software stack from use within a simulation environment to use within a "real world" physical environment does not seem to be well studied and/or documented. The presented work will strive to investigate both of these points by training a DQN in simulation and making necessary modifications to the overall system to allow it to execute on a physical robot platform.

III. Method

As mentioned previously, the goal of this presented work is to create a Deep Q-Learning Network (DQN) that will allow an autonomous agent to navigate through a given environment to a designated goal. DQNs allow the benefits of standard reinforcement based Q-Learning and the benefits of DNNs to be combined into one paradigm. One benefit in particular is that DQNs do not have the same space complexity issues as Q-Learning models, because the Q-Table is represented as a DNN within a DQN model [3]. Ideally, the agent will be able to navigate to a given goal in an environment in which it has no prior knowledge of. Training of the agent takes place within a virtual simulation environment known as Gazebo. This is done because each training episode that results in the agent being punished, likely happens due to the agent colliding with walls or objects that have the potential to damage expensive equipment in a physical training environment.

The agent in question is implemented within the Gazebo simulation environment as a TurtleBot 3 mobile robot using manufacturer generated Universal Robot Descriptor Files (URDF). The virtual representation of the Turtlebot 3 agent has access to LIDAR and odometry data. The TurtleBot 3 move base software was also setup to allow the agent to receive movement commands from an external ROS node and actuate said commands within the environment. The episodic training environment used here made use of heavily modified versions of the ROS packages developed by ROBOTIS for machine learning and robot simulations within Gazebo. The 12,000 training episodes conducted overall were split across three different training environments. At the end of the training process of a given stage, the next training process loads the generated model weights and continues the process. The first 3,000 episodes were conducted in a very simple stage with only a few simple obstacles. The second 3,000 episodes were conducted in an environment that had long corridors, tight corners and small doorways, while still being easy to navigate. The final 6,000 training episodes were conducted within a scale model of the third floor of the Kate Gleason College of Engineering (KGCOE) at the Rochester Institute of Technology. This progression of training stages was designed in an effort for the agent to learn how to navigate in a simple environment, and then extend what the model learned to a more complex environment, before finally testing the learned behavior within the final validation stage. For the first two stages, the learning rate was set to 0.0025, and 0.00025 for the last stage. These three training environments can be seen in figure 1. The overall training process summary can be seen within table I.

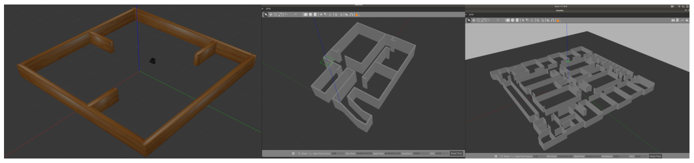

Fig. 1. Training Environments

Stage	Intended Complexity	Epsisodes Conducted Within Stage	Learning Rate During Episodes
1	Not Very Complex	3,000	0.0025
2	Somewhat Complex	3,000	0.00025
3	Extremely Complex	6,000	0.00025

TABLE I
EPISODIC TRAINING PARADIGM SUMMARY

In terms of the calculation of the reward, the TurtleBot 3 agent is punished heavily for colliding with any obstacle and the given training episode is stopped at this point. If the TurtleBot 3 agent is able to navigate to the goal state without any collisions, it is heavily rewarded. Again, when (and if) the agent finds the goal state within a given training episode, that episode terminates. The agent can also be punished (to lesser degrees than hitting an object) during the training episode. The overall calculation of a reward at a given time step can be seen within the algorithm within figure 2.

Algorithm 1 Reward Calculation

```
procedure CALCREWARD($GoalDist$, $ScanData$, $AngSpeed$, $State$)
    global $NumTurns$
    $ScanPunish = 0$
    $SpinPunish = 0$
    for each float $i$ in $ScanData$ do
        if $i$ less than 0.3 then
    $ScanPunish = ScanPunish + 0.01$
        end if
    end for
    if abs($AngSpeed$) greater than 1.3 then
    $NumTurns = NumTurns + 1$
    end if
    if $NumTurns$ greater than 30 then
    $SpinPunish = 5$
    $NumTurns = 0$
    end if
    if $State$ is not goal and $State$ is not collision then
    $reward$ = -$ScanPunish$ - $4 * GoalDist$ - $SpinPunish$
    return $reward$
    end if
    if $State$ is goal then
    $reward = 200$
    return $reward$
    end if
    if $State$ is collision then
    $reward = -200$
    return $reward$
    end if
end procedure
```

Fig. 2. Reward Calculation Algorithm

Using the reward calculated from the algorithm above at a given time step, an estimate of the action value function is calculated from the Bellman equation. The update equation for Q-Learning can be seen below in equation one. Note that γ in equation one is the discount factor. This parameter can be tuned to change the agent's preference of pursuing actions that increase future utility [6]. This parameter was set to 0.95 for the duration of all training episodes. This was done in an effort to continually value future rewards to encourage exploratory behaviors. a is the action taken within state s that leads to state s' [6]. The learning rate within equation one is represented as α [6]. The Q-Functional (written as Q() in equation one) uses these aforementioned parameters to compute the expected reward for the current or future state-action pairs [6].

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (R(s) + \gamma * max(Q(s',a')) - Q(s,a)) \quad (1)$$

[6]

A deep learning model was constructed using the python Keras API in which the behavior needed to navigate each environment could be learned. This model consisted of an input layer, three dense layers and an output layer. Dropout layers were also added in-between dense layers to prevent over-fitting. The input layer had an input shape of 364. This corresponded to 360 for scan data points, one for the calculated reward, one for the action taken, one for angular velocity and one for linear velocity. The overall model architecture can be seen in figure 3.

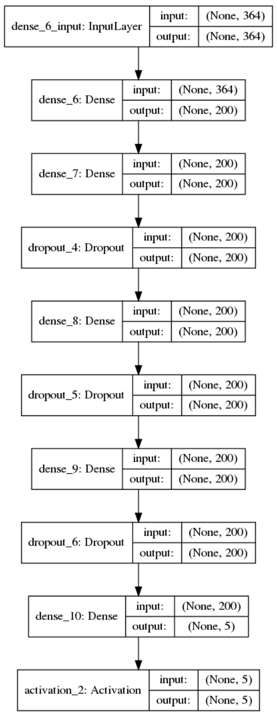
dense_6_input: InputLayer | input: (None, 364) | output: (None, 364)
dense_6: Dense | input: (None, 364) | output: (None, 200)
dense_7: Dense | input: (None, 200) | output: (None, 200)
dropout_4: Dropout | input: (None, 200) | output: (None, 200)
dense_8: Dense | input: (None, 200) | output: (None, 200)
dropout_5: Dropout | input: (None, 200) | output: (None, 200)
dense_9: Dense | input: (None, 200) | output: (None, 200)
dropout_6: Dropout | input: (None, 200) | output: (None, 200)
dense_10: Dense | input: (None, 200) | output: (None, 5)
activation_2: Activation | input: (None, 5) | output: (None, 5)

Fig. 3. Deep Q-Learning Model Architecture

The activation functions within each layer were chosen arbitrarily, but early testing indicated that Rectified Linear

Unit Functions (ReLU) were effective as hidden layer activation functions. The choice of the number of neurons within the hidden dense layers was also chosen mostly arbitrarily. However, the number of neurons within the hidden layers was chosen to be within the model input shape and the output shape (i.e. between 364 and 5). Three hidden layers were used to attempt to learn complex latent features within the episodic training data. The overall model was updated with the obtained episodic training data when a given reward was received and at the end of the given training episode. The output layer consisted of five neurons exhibiting a linear activation function. These output neurons each correspond to one of the possible discretized actions that the agent could take. In this sense, the model was attempting to solve a regression problem mapping the aforementioned state variables to a given action. Table II outlines this mapping.

Output Layer Value	Corresponding Action
0	full left motion
1	partial left motion, partial forward motion
2	full forward motion
3	partial right motion, partial forward motion
4	full right motion

TABLE II
OUTPUT LAYER VALUE TO ACTION TAKEN MAPPING

IV. RESULTS AND DISCUSSION

The model was validated using the third and final stage representing a scale model of the third floor of KGCOE. To begin the model validation procedure, the model weights obtained from the training process were loaded into the model. From this, a random valid goal state was selected from the available goal states. The agent was then allowed to attempt to move towards the goal state. The outcome of this test episode is recorded and the overall process is repeated 10 times. If the model allows the agent to reach the goal state with a 60% success rate or better, then the model can be said to be validated. This procedure can be seen within table III.

Step #	Description
1	Initialize model and third floor KGCOE Gazebo environment
2	Load model weights from previous training episodes
3	Select a valid goal state
4	Allow robot to navigate towards goal state
5	Record outcome
6	Repeat steps three through five for 10 trials
7	If 60% of trials resulted in an outcome where the goal was reached, the model is validated. Otherwise, it is not.

TABLE III
MODEL VALIDATION PROCEDURE

It can be seen in table IV that the agent had a 30% goal finding rate after the fist 3,000 training episodes. At this point, the model cannot be said to be validated. This is expected, as at this point the agent has only been trained for 3,000 episodes within a very simple environment. The next 3,000 training episodes were completed on the second training stage, and the final 6,000 training episodes were allowed to complete on the final training stage. The model procedure outlined above was repeated using the model weights obtained from all 12,000 training episodes. The results from this process can be seen in table V.

Trial #	Starting Position (x,y)	Goal Position (x,y)	Outcome
1	(0.6, 0.0)	(3, -4)	Reached Goal
2	(0.6, 0.0)	(13, 3)	Collision
3	(0.6, 0.0)	(15, 3.5)	Collision
4	(0.6, 0.0)	(9, 0)	Collision
5	(0.6, 0.0)	(5, 4)	Collision
6	(0.6, 0.0)	(4, -3)	Reached Goal
7	(0.6, 0.0)	(11, 2)	Collision
8	(0.6, 0.0)	(-12, 1)	Collision
9	(0.6, 0.0)	(4, -4)	Reached Goal
10	(0.6, 0.0)	(3, -17)	Collision

TABLE IV
MODEL VALIDATION RESULTS AFTER 3,000 TRAINING EPISODES

Trial #	Starting Position (x,y)	Goal Position (x,y)	Outcome
1	(0.6, 0.0)	(3, -4)	Collision
2	(0.6, 0.0)	(13, 3)	Collision
3	(0.6, 0.0)	(15, 3.5)	Collision
4	(0.6, 0.0)	(9, 0)	Collision
5	(0.6, 0.0)	(5, 4)	Collision
6	(0.6, 0.0)	(4, -3)	Collision
7	(0.6, 0.0)	(11, 2)	Collision
8	(0.6, 0.0)	(-12, 1)	Collision
9	(0.6, 0.0)	(4, -4)	Collision
10	(0.6, 0.0)	(3, -17)	Collision

TABLE V
MODEL VALIDATION RESULTS AFTER 12,000 TRAINING EPISODES

It can be seen in table V that the agent was unable to navigate to any of the goal states after the model had been trained on 12,000 training episodes and thus, the overall model cannot be validated. There are several reasons for which this may have happened. The first being that the overall DQN model architecture is not suited for this specific problem domain. Another reason may be that the reward structure was incorrectly designed. This could have led to the agent learning behaviors that maximized its utility but did not solve the problem at hand. Another reason this may have happened may be due to the training process overall. Perhaps the agent is unable to transfer what it has learned in the first two training stages to be used during the third training stage. Or perhaps the behaviors learned in the third training stage have overwritten the ones learned in the earlier training stages, leading to undesirable behaviors being learned. Given that the agent's ability to find the goal decreased during training episodes conducted on the second and third stages, it is likely that the problem can be isolated here.

In order to begin to diagnose where problems begin to arise, one could complete the training episodes on the first two stages and perform the validation procedure seen above. If the results from this procedure are equal to or better than those obtained from the validation procedure run using weights from the first 3,000 training episodes, then the problem can be isolated to the third and final training stage. If the results do not improve, then there is likely a bigger problem at hand (i.e. the model structure or reward structure is not designed correctly for the given problem). A third possibility pertaining to poor model performance is that of the increasing complexity of the training stages. The jump in environmental complexity between stages 1 and 2 and stages 2 and 3 may be too large. That is, the increased complexity of a training environment as compared to the previous may be large enough that learned behaviors from the previous stage are no longer valid in the new training environment. This variable could be eliminated by adding additional training stages with more gradual increases in complexity. It should be noted that one reward structure was used for all three training stages. Multiple reward structures may be needed depending on the overall training environment. For example, perhaps the negative reward received from being far away from the goal needs to be scaled differently depending on the overall size of the training stage.

One interesting finding about the overall reward structure will be examined. It was found after the first training stage was completed, the agent was spinning in circles in the center of the stage. This was happening as it learned to minimize punishments incurred by hitting the walls by not moving all together. At this point, the reward structure was modified to include a negative reward for spinning in place more than 30 times in order to disincentivize this behavior. Because the reward structure was changed, the overall training process needed to be restarted.

Lastly, it must be mentioned that the model was unable to be verified on a physical representation of the TurtleBot 3 platform due to the COVID-19 outbreak and the closure of academic facilities at Rochester Institute of Technology. Because the values produced at the output layer of the model are mapped to linear and angular velocities of a robot rather than wheel velocities, the overall machine learning portion of this project can be used with any physical agent that can be controlled using Twist ROS topics. This design flexibility allows this software architecture to be used to train and actuate a physical agent implementation other than that of the TurtleBot 3 platform.

V. Future Work

One of the main next steps for this endeavor would be to isolate and fix the problem(s) causing the poor performance of the agent's goal obtaining functionality. Once this is completed, several other sub-problems could be examined and many more extensions to the overall work could be made. For example, the doors within the constructed scale model of the third floor of KGCOE could be animated to become dynamic obstacles. Additionally, dynamic obstacles could be added in the main hallways and corridors to simulate students, faculty and other staff as they walk around. As previously mentioned, one of the other major next steps of this undertaking would be to extend the trained model for use on a physical agent implementation.

Another possible area of future work could be within the realm of model comparison. That is, one could use the multi-training stage episodic process outlined here to train and compare different DQN model architectures. In particular, the use of a DQN incorporating convolutional layers should be investigated as this would potentially extract latent features from the spatial and temporal relationships in the episodic training data which may improve overall performance.

VI. Conclusions

A deep reinforcement learning technique was examined as a means to teach an autonomous agent to navigate a given indoor environment. The training process was unique as it employed the use of a sequential, multi-stage training environment paradigm, where each new stage increased in complexity. Unfortunately, the agent's overall efficacy of finding goals after being trained for 12,000 training iterations over three training stages was bad (0% goal finding rate) when the obtained model weights were evaluated using the aforementioned validation procedure. Interestingly enough, the goal finding rate seen after all training episodes was less than the goal finding rate obtained when the model was evaluated after 3,000 training episodes conducted on the first stage. The reason for this could lie in the model and reward structures themselves, or perhaps the increase in complexity between training environments is too great for behaviors learned in the previous training stage to be applicable in the more complex environment. More work is required to determine the reason for the poor performance observed here.

The presented work is important because it outlines the potential efficacy of training an autonomous agent to navigate a type of environment it has no prior knowledge of. If the agent is trained within a sequence of environments representing the end-goal environment in a way where the learned behavior can be validated, then the model could potentially be used in a real world scenario. This may be useful for the design of autonomous agents operating in extremely dynamic and unpredictable environments such as combat zones and disaster relief situations. In these environments, traditional techniques such as various SLAM algorithms are prone to error accumulation. That being said, perhaps this technique could be paired with the traditional SLAM techniques to assist the agent in goal finding activities within incredibly complex and dynamic environments.

Acknowledgments

I would like to thank my Artificial Intelligence Explorations professor, Dr. Jamison Heard for his insightful feedback throughout this academic undertaking.

References

[1] G. Grisetti, C. Stachniss, and W. Burgard, "SLAM GMapping," OpenSLAM.org, 2005. [Online]. Available: https://openslam-org.github.io/gmapping.html. [Accessed: 03-Mar-2020].

[2] O. Tuzel, G. M. D. Santos, C. Fleming, and J. A. Adams, "Learning Based Leadership in Swarm Navigation," Lecture Notes in Computer Science Swarm Intelligence, pp. 385–394, 2018.

[3] S.-H. Han, H.-J. Choi, P. Benz, and J. Loaiciga, "Sensor-Based Mobile Robot Navigation via Deep Reinforcement Learning," 2018 IEEE International Conference on Big Data and Smart Computing (BigComp), 2018.

[4] B. Moridian, B. R. Page, and N. Mahmoudian, "Sample Efficient Reinforcement Learning for Navigation in Complex Environments," 2019 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR), 2019.

[5] N. Duo, Q. Wang, Q. Lv, H. Wei, and P. Zhang, “A Deep Reinforcement Learning Based Mapless Navigation Algorithm Using Continuous Actions,” 2019 International Conference on Robots Intelligent System (ICRIS), 2019.

[6] S. J. Russell and P. Norvig, Artificial intelligence: a modern approach. Upper Saddle River, NJ: Pentice Hall, 2010.