Q-Learning in an Autonomous Rover

Marcus McGuire, Paul Morris, Washington Garcia, Shawn Martin, Nicolas Tutuianu & Elan Barenholtz

Robotics researchers often require inexpensive hardware that is freely distributable to the public domain in large numbers, yet reliable enough for use in different applications without fear of the hardware itself becoming a burden. In the past, researchers have moved towards robot simulations, in favor of the lack of hardware and ease of replication. In this paper we introduce an implementation of Q-Learning, a reinforcement learning technique, as a case study for a new open-source robotics platform, the Brookstone Rover 2.0. We utilize a Theano-based implementation of Google DeepMind's Deep Q-Learning algorithm, as well as OpenCV for the purpose of state-reduction, and determine its effectiveness in our rovers with a color-seeking "rover-in-a-box" task.

• • •

1 INTRODUCTION

In recent years, there has been enormous interest in the development and application of machine learning techniques. A major obstacle for real-world applications of machine learning is the design, implementation, and training of hardware to perform a specific task. In the example of a computer-controlled autonomous device, the investment in parts and the running time associated with training a machine learning algorithm in a robot can be inefficient. We propose a hardware-agnostic method of state space reduction to unify the state representation between a simple simulator and real-world agent, which allows for faster implementation of reinforcement learning algorithms on real-world agents. We apply this method to a physical optimal path problem using toy rovers and a reinforcement learning technique, Q-Learning, to examine its effectiveness against a statistical performance baseline.

1.1 FPV Rover

We have begun to utilize off-the-shelf first-person view (FPV) Rovers sold by the Brookstone manufacturing company. These tank rovers currently sell for $99 and come pre-equipped with motor operated tank treads for movement, a tilting forward-facing camera and Wi-Fi connectivity. Everything is housed in a plastic case which can be easily modified to attach secondary sensors and instruments. Power is supplied by six AA batteries.

Most importantly, each component can be independently controlled remotely through a Python API [5], developed by Simon D. Levy, which may be programmed directly into a variety of configurations housed on an independent computer. This ease of hardware setup and programming flexibility allows for more time to be spent on the implementation of the experiments rather than troubleshooting hardware.

1.2 The Current Study

In the current study, we utilized the tank treads and camera to use the rover as a Q-Learning agent, which will choose an optimal set of actions to achieve a predefined goal. The agent is implemented as in previous Deep Learning experiments by Google DeepMind [6], which were able to match or surpass human performance in several Atari 2600 games. The goal of this work was twofold: create a physical implementation of the Deep Q-Learning

algorithm presented by DeepMind, and also act as a springboard for future AI projects involving the Brookstone rover.

1.2.1 Q-Learning

Q-learning is formally defined as a model-free reinforcement learning algorithm that can be used to find an optimal action-selection policy for any finite Markov decision process [10]. This definition implies that the given problem model consists of an agent, states, and a set of actions for each state. It also implies that the agent does not attempt to learn a model of its environment; it only learns from its history of interactions with it. The only objective of the agent is to maximize the total reward. This reward is not received until the process has been completed or a terminal (final) state has been reached. Therefore, the agent determines which action is optimal in each current state by calculating which action has the highest long-term reward or the weighted sum of the expected rewards of all future steps. The Q-Learning equation is shown below [10].

1.3 World

Here, we implement a "rover-in-a-box" paradigm modeled after the operant conditioning chamber, or "Skinner Box", an apparatus used to study animal behavior. An animal subject is placed in a box and given different rewards whenever the subject interacts with some lever or mechanism inside the box [7]. The idea is to replicate this apparatus where the rover acts as the subject, and the inside of the box represents the rover's world. A different color of paper is placed on each side of the box, with a reward given whenever the rover sees a specific color. The overall goal is to learn the shortest path to the reward.

2 METHODS

2.1 Image Processing

The input image to the rover is very high-dimensional. In order to reduce the state space for Q-Learning, we used the OpenCV module to analyze and distil each frame of the rover's video feed into its key points. The image processing started

$$\underbrace{Q_{t+1}(s_t,a_t)}_{resulting\ Q-value} \leftarrow \underbrace{Q_t(s_t,a_t)}_{old\ value} + \underbrace{a_t(s_t,a_t)}_{learning\ rate} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{reward} + \underbrace{\gamma}_{discount\ factor} \underbrace{maxQ_t(s_{t+1},a)}_{estimate\ of\ optimal\ future\ value}}^{learned\ value} - \underbrace{Q_t(s_t,a_t)}_{old\ value} \right) \quad (1)$$

BRIEF DESCRIPTION

The resulting Q-value that is calculated is the expected total reward of taking an action (a_t) in a given state (s_t). This is the immediate reward received upon taking that action in that state (r_{t+1}) added by the discounted utility (γ) of the resulting state. Then, the difference is found between the resulting number of this and the previous Q-value, since this algorithm updates the Q-value during each instance rather than resetting it.

Explaining each parameter in more detail:

- Learning rate ($a_t(s_t, a_t)$) - This determines how fast the newly acquired information will override the old information ($0 < \alpha < 1$). If the learning rate is 0, the agent will not learn anything. If it is 1, solely the most recent information would be considered. A constant closer to 0 is often used such as $\alpha(s, a) = 0.1$ or for all t.
- Discount factor (γ) - This influences the importance of how soon reward is received ($0 < \gamma \leq 1$). If γ is closer to 0, the agent will only consider short-term rewards. On the other hand, if γ is closer to 1, the agent will strive for a high-reward in the long-term. A constant closer to 1 is sometimes used; however, starting with a lower discount factor and increasing it toward a higher value can be used to accelerate the learning process.
- Estimate of optimal future value (Utility) ($maxQ_t(s_{t+1}, a)$) - The utility of a state is the Q-value of the best action from that state. From the next state, the agent considers all the actions and takes the maximum value.

with the raw video feed from the rover's built-in camera. The image was then converted from a red-green-blue (RGB) color space to a Hue-Saturation-Value (HSV) color space. This was done because HSV provided greater contrast between colors and allowed for easier definition of target color ranges. The OpenCV library was then used to make a mask of the image for each target color, discarding the pixels that do not fall in that color's HSV value range.

To perform noise reduction, we used opening and closing morphological operations. Both opening and closing operations use the operation dilation, where every 5x5 set of adjacent pixels that contains at least one active pixel, becomes a 5x5 set of all active pixels, and the operation erosion, where every 5x5 set of adjacent pixels that contains at least one inactive pixel, becomes a 5x5 set of all inactive pixels. Morphological opening performs an erosion followed by a dilation in order to remove small particles from the image. Morphological closing performs a dilation followed by an erosion in order to fill holes within larger objects in the image. The other method we used for noise reduction relies on the fact that the noise we were trying to eliminate was random, while the target object remained relatively constant. The noise was filtered out by comparing all the pixels in each mask to the pixels in the mask of the two frames before it and removing the pixels that were not present in all three of the masks.

After the noise was filtered out, OpenCV was used to find the contours of the objects in each of the masks. Then, OpenCV was used to find the center of the largest contour. This central point for each color is the main piece of information we used to determine the state. To determine the state from the central point, we broke the image into 3 equal portions: the far left third of the image, the middle third, and the far right. The x –coordinate of the central point was compared to the x value cutoffs for each third of the image to determine in which point the point was located. Each third was represented as either a 1 or a 0, depending on whether the central pointed was located in the third or not. For example, a state of [1,0,0] means there is an object of the target color with its center located in the left third of the rover's image. These three numbers that represent the state for a single target color were computed for each of the four target colors and combined into a single set of twelve numbers that constitute the rover's whole state.

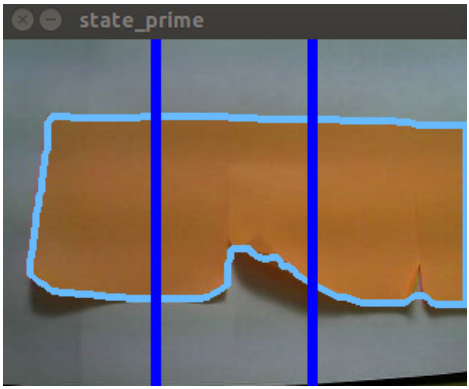

Figure 2: **The Rover highlighting an orange wall**

2.2 Simulation Method

Due to the physical limitations of robotics, a rover in a box being trained through reinforcement can only learn as fast as it can turn. Because of the large number of training iterations necessary for the rover to learn an optimal policy live training can be inefficient. Therefore, we developed a box simulation in order to learn an optimal policy for actions in the simulated world and test the effectiveness of "offline" training in a simulator. In reality, the rover in our example was able to observe an incredibly large number of states inside the "box" corresponding to the number and pixels in each frame from its input camera and the value of each pixel. It would be difficult to construct an accurate simulation of the images the rover would receive navigating inside the box due to the sheer number of possible states in the simulation. However, after image processing, the "state space" of the box example was greatly reduced to only essential information. This reduced state space was small enough to encode in a simulation as a digital "world". We designed this simulation to test whether a policy of actions trained in a "world" that accurately simulated the colors of the walls inside a box would translate to use in a real rover with the help of image processing.

The simulation was implemented in Python and consisted of three parts. The simulation had a world made up of states and transitions, an iteration loop that simulated a rover's movement throughout a box, and an implementation of the Q-Learning algorithm written in Google's TensorFlow machine learning library. The world's states were linked by

a list of transitions, where each transition corresponded to an action. For instance, if turning left in state A would lead to state B, the world would contain a transition ((A, Left) => B) and would change to state B whenever the simulated rover made a left turn in state A. The iteration loop observed a simulated rover's current state, obtained an action from the Q-Learning algorithm's learned policy, took the action, obtained a new state and reward from the world, and updated the Q-Learning algorithm's expected reward based on the new state and action. In this process, the Q-Learning algorithm learned to predict the reward it would receive for taking a certain action in a certain state. After a sufficient number of iterations, the Q-Learning implementation was expected to learn a policy of actions for navigating the states in the simulated world. The success of the policy trained from the simulation would be based on how closely the actions recommended by the policy compare to the actions necessary to take the shortest path from a starting position to the target wall, or "goal state". The metric we used to analyze the simulation results was the difference between the number of states the trained policy would pass through when told to take the shortest path to the goal state and the actual minimum number of states necessary to reach the goal state.

2.3 Rover Implementation

The rover implementation is written in Python and builds off the Brookstone Rover 2.0 API created by Simon Levy [5]. The implementation is split into separate modules to control different aspects of the program. A diagram of the different modules is shown below:

Following the algorithm presented by DeepMind [6], several states are taken with random-policy and stored in replay memory. At each time step, the state consists of a 4x3 matrix representing the four colors of the box and in which three portions of the screen they are located. In an ideal rover box scenario, there are never more than two colors present on the screen at a time. Once the replay memory is filled, the rover loops for an arbitrary amount of iterations in what is called the learning phase. At each iteration, the Q-Learning equation (1.2.1) is emulated by optimizing the weights of a neural network. In this experiment, it is necessary to define a beginning state and a goal state. In our experiments, we define the pink color as the goal (terminal state) and assign it a constant

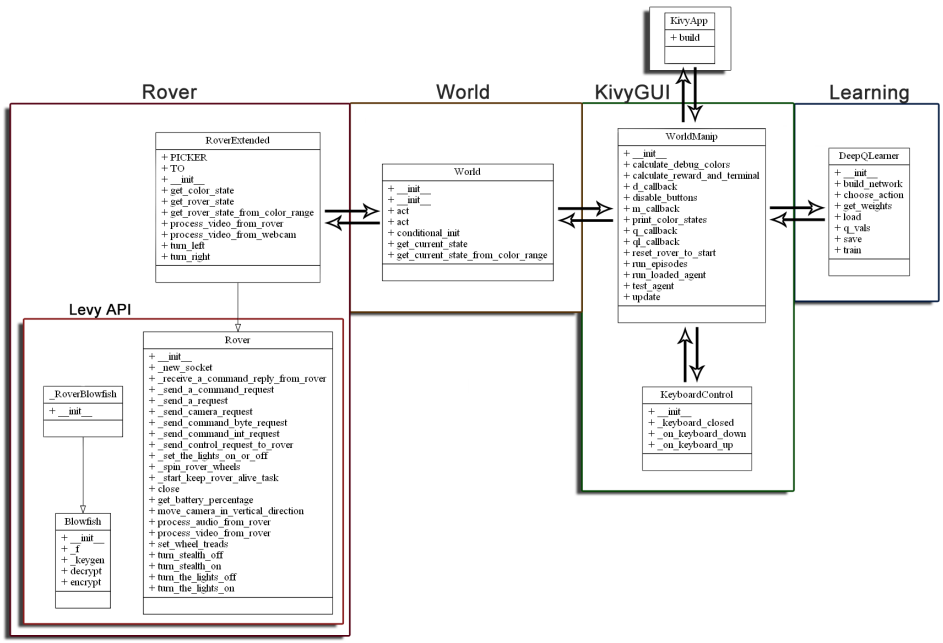

Figure 3: **UML Diagram of the Rover Implementation**

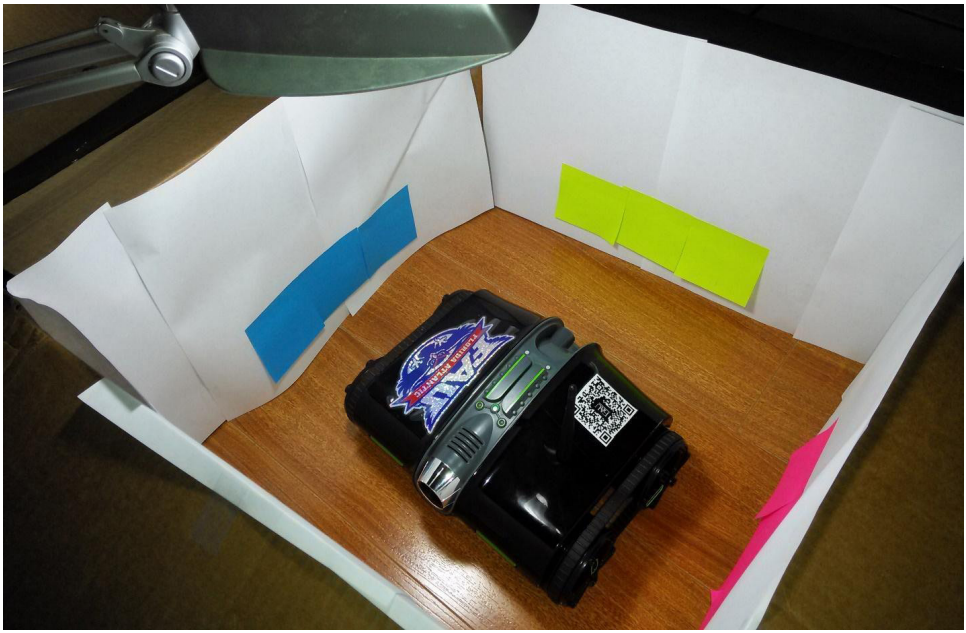

Figure 4: **Skinner Box environment, the rover's "world", consisting of pink, green, blue, and orange (not visible) sticky notes**

positive reward. The beginning state is defined as whatever color lies on the opposite side of the pink wall.

The primary modules are labeled Rover, World, KivyGUI and Learning. Rover controls the low-level functionality to communicate with the rover and is composed of the Simon Levy rover API. The Learning module defines the Deep Q-Learning agent, using Theano [1] and Lasagne [4] to handle the training and Neural Network updates respectively. The Q-Learning module is based off code published online [9], which follows the algorithm described by DeepMind [6] and implemented by Nathan Sprague [8]. The world is an abstraction of the Skinner Box world and is responsible for obtaining information processed by the rover and sending it to the learning model within KivyGUI. KivyGUI uses the Kivy GUI programming framework [5] to define a Graphical User Interface (GUI) for the human operator to interact with, which in turn manipulates the World module through different modes available to the user.

The structure of our program is inspired by previous work by Korjus, who attempted to recreate the DeepMind experiments [3]. In their implementation, Atari Learning Environment (ALE) was used as the central processing unit between the human interface and the Q-Learning agent. We mimic this with our World module, which in practice relayed the information between the Rover and Learning modules. By treating the agent as an inhabitant of the World, it was much easier to envision what power each object should have over the agent. In our case, the human operator has control over the World, which in turn has control over the agent.

Several rover control modes were made possible with this organization, which in our latest release includes Manual Mode, OpenCV Debug Mode, and Q-Learning Mode. The human operator can choose to control the world directly (Manual Mode) or let the Q-Learning module run through the training (Q-Learning Mode). OpenCV Debug mode was created as a tool to help us find the most accurate Hue-Saturation-Value color codes to use in our OpenCV implementation. This structure also opens up the opportunity for future work with the rovers.

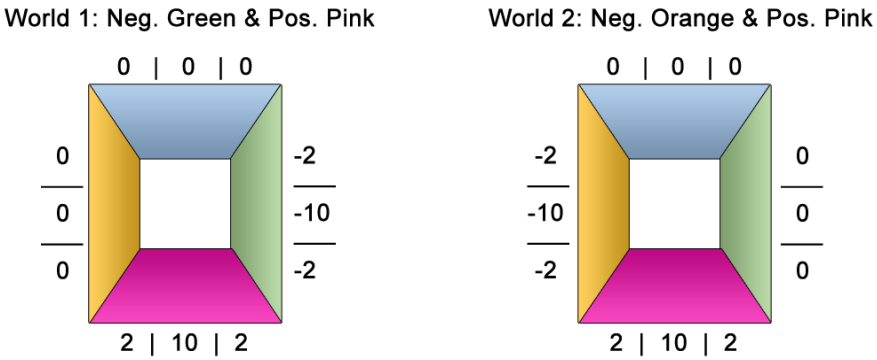

Figure 5: **A model of the rover's world with the rewards it received at each location**

2.4 Materials

The rover's physical world was created using a cardboard box and pieces of neon colored sticky-notes:

An advantage of the Skinner Box scenario is that the true optimal path is known beforehand. Based on the reward assignments, it is often easy to see what the optimal path will be and what sum of rewards Ro is expected. A good approximation of the optimal path occurs when the sum of rewards Ro is maximized: where k is the number of moves before finding pink, and is thus minimized.

Flipping the rewards to generate two unique worlds ensures the agent can learn regardless of the model's input. Since the action selection is limited to only left or right, it is possible for a model to have the rover only go a single direction such that the pink card is always found, regardless of reward. With this reward combination, and given the previous definition of Ro, a good approximation of the optimal path is one which avoids the color with negative rewards and takes the minimum number of moves. The speed of the rover motors is kept constant such that four complete moves are required to flip the rover to the opposite side of the box. By intuition, the optimal path, starting from the blue wall, will always take four moves.

$$R_o = max(\sum_{t=0}^{k} R_t) \qquad (2)$$

For the scope of this paper, we focus on the number of learning phase iterations and their effect on the final model. A physical limitation of the experiments was the amount of time taken for the rover to act, which caused high iteration experiments to become unfeasible, due to battery life. We observe if an optimal path could be found despite limited iterations. After the learning phase completes, the model is tested by setting the rover to the blue wall and allowing it to act according to the information collected.

A run ends when the rover either finds the pink card or when it has reached the upper threshold for the number of moves, 12. If the rover did not find the pink wall in 12 moves or less, no path was found. The number of moves and optimality is recorded for each run. The number of times a path was found is compared against the number of times an optimal path was found, to visualize how accuracy is affected by the number of iterations.

3 RESULTS

3.1 Simulation Results

Our Neural Network Q-Learning implementation was tested in a simulation of the box with twelve possible states. After two thousand training iterations at a learning rate of 1e-3 for each iteration, the length of the path the network learned to take was evaluated against the actual shortest path. The problem was evaluated at multiple solution states

# State Changes for Shortest Path	Average Path Across 100 Runs	Difference
2 State Changes	2 State Changes	0 State Changes
4 State Changes	4.56 State Changes	0.56 State Changes
6 State Changes	6 State Changes	0 State Changes

Table 1: **Simulation Results over 100 runs for Q-Learning Agents**

# State Changes for Shortest Path	Average Across 100 Runs (Random)	Difference (Random/Q-Learning)
2 State Changes	26.5 State Changes	24.5 State Changes
4 State Changes	39.71 State Changes	35.71 State Changes
6 State Changes	45.07 State Changes	39.07 State Changes

Table 2: **Simulation Results over 100 Runs for Random Agent, and Comparison to Q-Learning Agent**

where the shortest path length fell into one of three categories, as shown in Table 1. The shortest path length is defined as the minimum number of state changes needed to take the shortest path.

The simulated Q-Learning algorithm performed perfectly when the shortest path was of length two or six on all one hundred runs. The algorithm still performed well on four-length paths, but occasionally found an eight-length path (meaning it turned the other direction around the circular twelve-state world.) This result seems to occur because the 1e-3 learning rate only propagated a significantly discounted reward to states two to four states away from the terminal reward. Because the discounted reward was similar at both four and eight steps away from the goal state, the network saw the same reward turning left and turning right in that position. At length six, this issue did not occur because the distance between left and right are both equal at a path length of six.

3.2 Rover Results

The primary goal of the learning phase is to construct a model that will guide the rover to an optimal path. The variables of the Q-Learning function, the hyper-parameters, were originally set according to the findings by Nathan Sprague [8]. A variety of unsuccessful preliminary runs were conducted in order to find a good combination of hyper-parameters that led to the pink card, although the results are not presented here. After several tests, the hyper-parameters were chosen as below:

One obstacle encountered was ensuring that the rover learned a path which was significantly better than taking random actions. The rover's accuracy is measured by its probability of finding an optimal path. For taking only random moves, the probability, P(r), is calculated using the four-step optimal path. Since the rover may only choose from two actions, there is 50% probability to make a correct move at each time step:

$$\begin{aligned} P(r) &= 100\% \cdot P(t_1) \cdot P(t_2) \cdot P(t_3) \cdot P(t_4) \\ &= 100\% \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2}. \\ &= 6.25\% \end{aligned} \tag{3}$$

where tn is the nth time step in the rover's world.

This percentage is used as the baseline for comparison against the Q-Learning results. However, we wanted to ensure that the rover met both conditions of the optimal path condition: that the number of moves is minimized, and that total rewards are maximized. To ensure both possibilities were tested, we utilize two unique worlds as discussed in

Learning Rate	Batch Size	Replay Size	Epsilon	Discount
$a_t = 0.005$	4 batches	16 iterations	$\epsilon = 0.2$	$\gamma = 0.9$

Table 3: **Ideal hyper-parameters for our rover experiments**

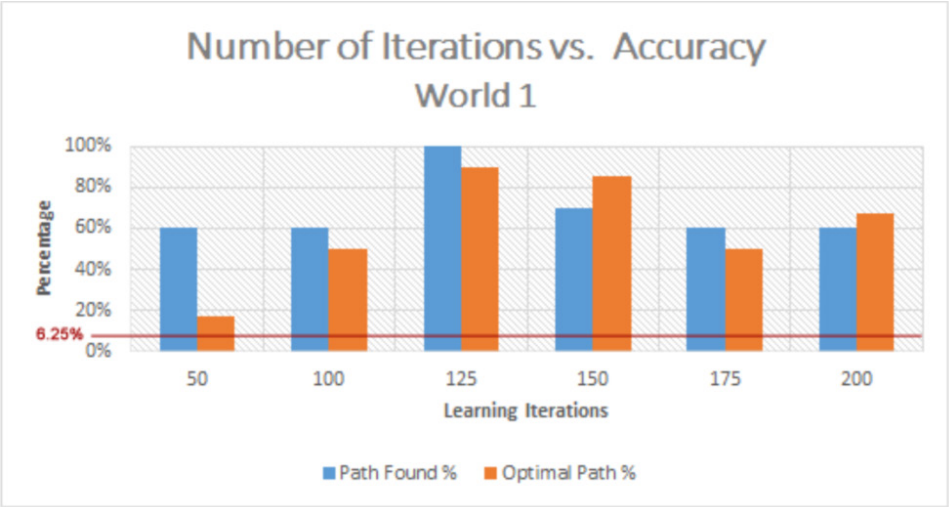

Figure 6: **Average of 10 runs for World One experiment on each iteration amount**

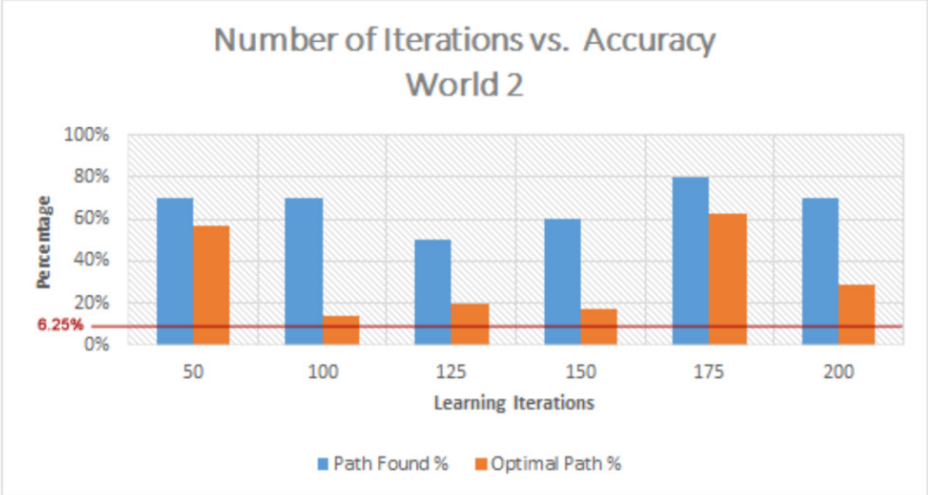

Figure 7 **Average of 10 runs for World 2 experiment on each iteration amount**

section 2.2. Ten separate runs were conducted after training for each iteration amount, and the average calculated to find how often a path is found.

The results are summarized in the tables above. As Figure 6 shows, accuracy increases with iteration amount but then falls off after 125 iterations. In most runs, the rover managed to find an optimal path, but sometimes the path found was not optimal. For 100 iterations and above, the Q-Learning model performed well above the baseline of P(r).

Figure 7 shows that accuracy is not as high for World 2, with a dip in the middle of the graph at 125 iterations. Accuracy increases at the lower and upper iteration counts, with the best performance at 175 iterations. At 175 iterations, the Q-Learner performed much higher than the baseline of P(r).

An interesting observation of the experiments is how often an optimal path was found for World 1 versus World 2. For World 1, the model gave a high rate of optimal output, but for World 2 a new model with the same iterations and same environment setup failed to yield a similar rate of optimal paths.

Despite the dip, the model performed well for World 1 with 125-150 iterations, with a slight drop off in accuracy as the iteration amount increased. A similar trend is visible in World 2, where instead the accuracy drops off at around 200 iterations. World 2 has a visible reduction in optimality compared to World 1. During testing, the rover had a tendency to focus on whichever path it found first for the duration of testing.

Another observable effect was an oscillation of the rover in specific corners of the world until the run ended, which was most common during tests with low and high iteration amounts. This behavior was unexpected for high iteration counts, and we believe over-training became a factor as our learning rate was kept constant. The introduction of a learning rate reduction schedule may alleviate this behavior in future experiments.

4 CONCLUSION

Similar to experiments by Google's DeepMind, we were able to successfully implement a Q-Learning agent, which allowed our rover to learn. Over many iterations, results show that our rover is capable of finding the shortest path seamlessly and accurately. Regardless of implementation, both implementations performed well above the baseline accuracy of a 6.25% random policy. This being said, the simulation and rover implementation of the "rover-in-a-box" problem both have their unique advantages. While both implementations were capable of producing accurate results, the simulation was able to produce an accurate Q-Table and trained neural network significantly faster than the latter, due to physical constraints such as time, battery life, and computer speed as opposed to rover movement.

These findings could be used in a magnitude of ways. An effective use would be to allow these two implementations to work in tangent. To build the Q-Table in a simulation, and then feed the Q-Table into a working rover. By recreating the environment in a simulation, this reduces the training time and improves efficiency. Regardless, this proof of concept simply does not do Q-Learning enough justice. Q-Learning, as well as concepts from Google's DeepMind could be implemented to do extraordinary things from allowing the creation of self-driving cars, to enabling the revolution of artificial intelligence as a whole.

REFERENCES

1. Deeplearning. Theano 0.8.2 Documentation. Retrieved May 23, 2016 from http://deeplearning.net/software/theano/
2. Kivy. Kivy: Cross-platform Python Framework for NUI. Retrieved June 5, 2016 from http://kivy.org/
3. Korjus, K., Kuzovkin, I., Tampuu, A., and Pungas, T. 2014. *Replicating the Paper "Playing Atari with Deep Reinforcement Learning."* Faculty of Mathematics and Computer Science at the University of Tartu Technical Report MTAT.03.291. University of Tartu, Tartu, Estonia.
4. Lasagne. Lasagne. Retrieved May 23, 2016 from https://github.com/lasagne/lasagne
5. Levy, S.D. 2013. How I hacked the Brookstone Rover 2.0. (September 2013). Retrieved May 23, 2016 from http://isgroupblog.blogspot.com/2013/09/how-i-hacked-brookstone-rover-20.html
6. Mnih, V., et al. 2013. Playing Atari with Deep Reinforcement Learning. Neural Information Processing Systems (NIPS) Deep Learning Workshop

7. Skinner, B.F. 1938. The behavior of organisms: an experimental analysis. Appleton-Centory-Crofts, Inc. New York, NY.

8. Sprague, N. 2014. *Parameter Selection for the Deep Q-Learning Algorithm*. James Madison University. Harrisonburg, VA.

9. Spragunr. spragunr/deep_q_rl. Retrieved May 23, 2016 from https://github.com/spragunr/deep_q_rl

10. Watkins, C. 1989. Learning from Delayed Rewards. Ph.D. Dissertation. King's College, Wilkes-Barre, PA