Application of Q-learning-based reinforcement learning algorithm on a dual-arm crawling robot

Richard Desatnik
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
rdesatni@andrew.cmu.edu

Regan Kubicek
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
rkubicek@andrew.cmu.edu

Tuo Wang
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
tuow@andrew.cmu.edu

Abstract

This work presents a crawling robot with two one-degree of freedom (DOF) arms that learns to move forward in real-time using reinforcement learning. The crawling robot (Qbert) is fitted with a microcontroller and rotary encoders to measure the robot's velocity and provide rewards to a Q-Learning algorithm. The robot was tested on various surfaces, gradients, and a weighted sled. The reinforcement learning algorithm was ran for 100 iterations and a time of 4 min and 50 seconds to ensure convergence on a policy. Qbert clearly demonstrates a change in learned feedback that is expected in each experiment. Data in the form of a Q-table is collected in real time via Bluetooth. Qbert provides a platform that can be used in the classroom as a hands on project to teach reinforcement learning.

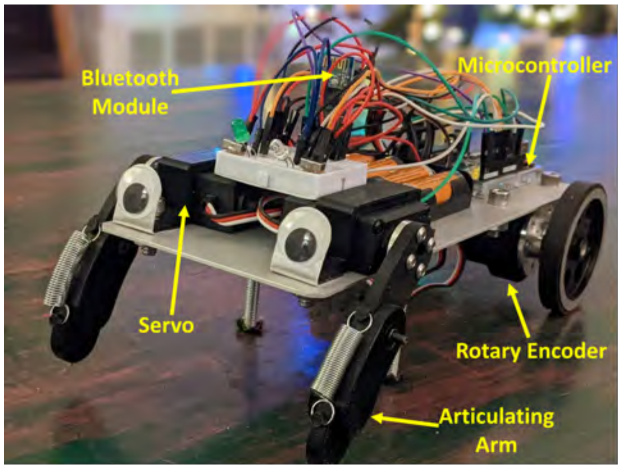


Figure 1. The hardware setup for Qbert is shown. The servos drive articulating arms. Rotary encoders record the forward or backward movement of the robot and report back distance for a reward.

1. Introduction

Reinforcement Learning (RL) is a class of machine learning algorithms that improves the agent's behaviors by iteratively upgrading its training policy. In reinforcement learning, developers devise a method of rewarding desired behaviors and penalizing undesired ones. Providing rewards and penalties during training ensures the agent seeks out a maximum reward, working towards an optimal policy. Through decades of development, RL has found it's way into a multitude of sectors such as computer systems, energy, recommender systems, finance, healthcare, robotics, and transportation to name a few [1].

In this work we employ a form of reinforcement learning referred to as Q-Learning. Q-Learning is a model-free reinforcement learning algorithm that determines values of actions at particular states within a state-space. The ”Q” in Q-Learning stands for ”Quality” and represents how beneficial an action is in gaining future rewards. At the heart of the Q-Learning algorithm is the Bellman equation that utilizes a value iteration update, essentially using a weighted average of an old value and new information. Q-learning has been classified as incremental Dynamic Programming (DP), as it iterates ultimately finding the optimal policy [6].

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma * maxQ(s_{t+1}, a) - Q(s_t, a_t)) \tag{1}$$

Equation (1) was used to update the values of Q. Where $Q(s_t, a_t)$ is the old value, α is the learning rate, r_t is reward,

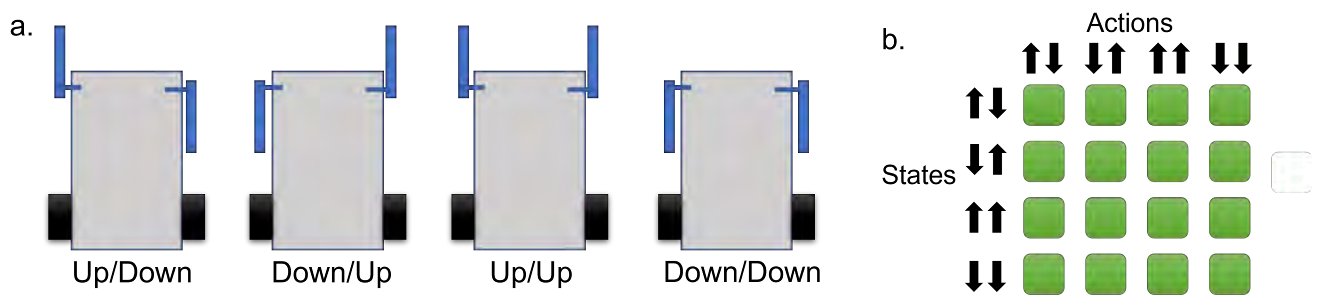


Figure 2. a. The four states are depicted based on various arm positions. b. The state-action space demonstrates the relationships between actions and states.

γ is the discount factor, and $maxQ(s_{t+1}, a)$ is an estimate of the optimal future value.

In this project, we fabricated a two-armed crawling robot (Fig. 1) to demonstrate the Q-Learning algorithm. Our robot, Qbert, moves forward using two articulating arms. The robot performs two-arm crawling motions similarly to swimming locomotion. It can employ two arms like the breaststroke or consecutive one-arm crawling like freestyle swimming. Rotary encoders provide feedback and reward the robot for moving forward. After several iterations, a max Q-value is acquired and implemented. Provided enough learning time an optimal policy is reached and carried out.

In the next sections, we describe related work on robot Q-Learning, our methods for designing tests, our experiments with hardware and learning algorithms, and we also discuss the limitations of robot learning algorithms and future works.

2. Related Work

Autonomous robots are becoming more ubiquitous due to the fact that they can make a decision based on their own analysis in a given complex environment. Q-learning, as a model-free learning algorithm, makes itself one of the practical approaches in this application. However, Q-learning has its limitation on slow convergence to the optimal solution. To address this limitation, Low, Ong, and Cheah introduced a partially guided Q-learning algorithm that accelerates the convergence when Q-values are initialized using flower pollination algorithm (FPA)[2]. To validate their algorithm, they applied the modified Q-learning algorithm in a robot path planning task in both a simulation environment and on a three-wheeled robot in the real world. The test results show that, compared with classic Q-learning, the computational cost of the FPA modified Q learning algorithm has been significantly improved.

In a more advanced setting, Sasaki, Horiuchi, and Kato applied Deep Q-network to robot behavior learning the simulation environment [4]. Deep Q-network (DQN), a combination of Q-learning and a Convolutional Neural Network (CNN) is one of the most famous methods of deep reinforcement learning. Previously, DQN showed superior game learning ability that outperformed human players in several games [3], and has the remarkable feature to learn a policy from high-dimensional image data through a CNN. In their work, the mobile robot learns to acquire good behaviors such as moving along the marked lines and avoiding collision by using high dimensional image data.

For a classroom demonstrator, Tokic, Ertel, and Fessler present a crawling robot with one two-DOF arm that learns to move forward [5]. They adapted a value iteration algorithm to the robot with a Markovian decision process (MDP). The crawling process is separated into four states which allows the robot to learn relatively quickly. In the end, their robot achieves good crawling performance and learning speed. This research aims to provide a robot as an educational platform where students can learn introductory-level knowledge of reinforcement learning. Based on their classroom demonstration, their robot was able to attract students' attention. Especially for an abstract, mathematically challenging problem like reinforcement learning, using a robot like this can be one of the best ways to motivate students to further learn RL. The work in this paper is based largely on the Tokic crawler.

3. Data

The data for our experiments is quantitative continuous data. For our experiments, the agent observes changes in time and position from the environment to obtain velocity which is used as the reward for the agent. The Arduino microcontroller receives position data from the encoder. The encoder with an attached wheel, records steps when rotated and sends a value in the form steps per revolution. We also make use of the millis() function in the Arduino library to keep track of time. The millis() starts to track the time when the Arduino is turned on. To process the data we take the difference between an initial and final encoder position per iteration so we can keep track of the distance traveled. When this value is divided by the change in time we are able to monitor the velocity of the Qbert robot. Each value

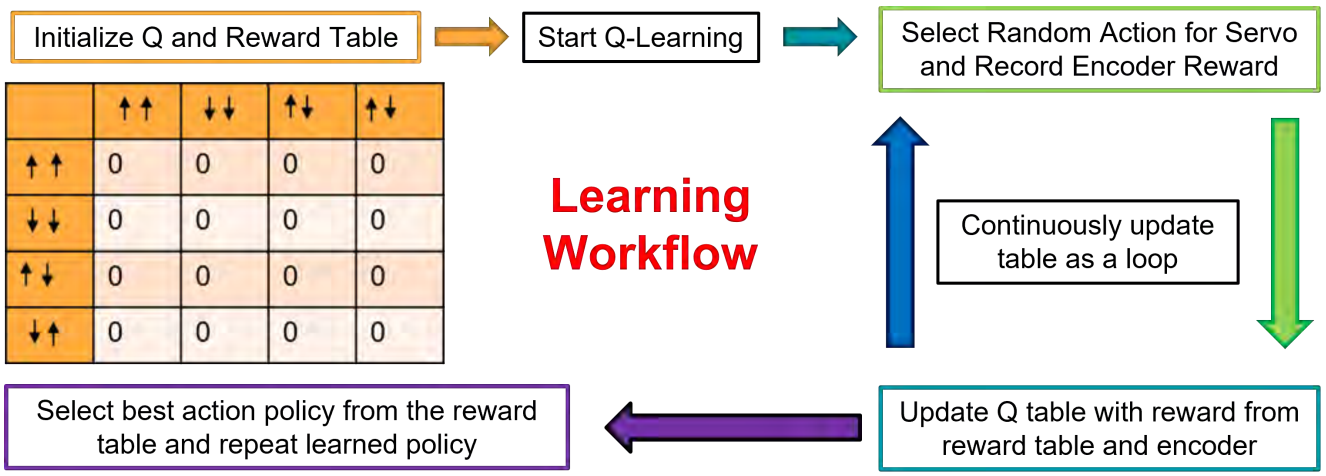

Initialize Q and Reward Table
Start Q-Learning
Select Random Action for Servo and Record Encoder Reward
Learning Workflow
Continuously update table as a loop
Select best action policy from the reward table and repeat learned policy
Update Q table with reward from reward table and encoder
↑↑
↓↓
↑↓
↓↑
0

Figure 3. The flowchart provides a visual of how the algorithm is implemented. The Algorithm starts in orange with the q-table initialization. Then loops through actions and updates table as illustrated by the blue and green arrows. Finally the table selects the policy converged on through value iteration as displayed by the purple arrow

of data provides either a reward or a penalty after each action, which is then used to update the Q-table. If the robot moves quickly then there is a high velocity giving a high reward and if the robot rolls slightly backwards, such as in the case of a back stroke then there is a negative reward effectively punishing the robot. The robot operates at roughly 83 actions and observations per minute of run time. To ensure we converge on an optimal policy given a change of environment we ran our tests for 4min and 50sec. This gives us a total of 400 actions per test and ensures the algorithm iterates through the Q-table 100 times before selecting a policy. In order to create truly random action selections for each test we use the Arduino random() command with an initial random seed starting from noise output of an analog pin.

4. Methods

4.1. Approach

Qbert utilizes Q-learning with value iteration. The reason we decided to use this technique is because we had discrete actions the robot could take and discrete positions the robot could be in. This limited us to having a small 4 by 4 Q-table. Since the table was so small we used value iteration to converge on a policy for the agent. We also did not have a virtual model of the system to use model based forms of reinforcement learning as we wanted to understand how the robot would perform directly from the physical environments we exposed the agent to.

4.2. Alternative Approaches

Q-learning with value iteration worked well with our experiment design. Other reinforcement learning methods would not have worked well for the conditions of our experiment. We did not used model based methods reinforcement learning as we did not have a model to accurately simulate the different surface terrains and contact forces the robot would be subjected to during the experiments. We did not want to use a form of model free deep neural networks, as the tests would have either required a simulated environment to run the vast number of tests required to converge on a proper policy. The run time between experiments would have also taken too much time for the battery life of the robot and would have required locations with much more space then where we had tested to use deep learning. Our robot also ran within a very tight state space with a limited number of discreet positions and actions. Therefore there was no need to use an algorithm that selects a policy on a continuous state-space.

4.3. Hardware Setup

The body of the robot is a rigid aluminum frame and two 4 AA battery packs. The bottom battery pack powers the microcontroller and servos. The microcontroller used in this case is an Arduino Uno. Two Taiss/AB 2 phase incremental rotary encoders were mounted on the rear of the robot to report back distanced traveled at a resolution of 600 pulses per revolution. Two 0.15kg servos were placed on the front of the robot. These servos were powered by the 4 AA battery pack on top. Articulating arms were attached to the servos. The articulating arms used a revolute joint and a tensile spring to restore position after bending. This causes the limb to push in a single direction as the robot crawls. To make better contract with changing terrain rubber feet were added to give Qbert more traction. Two LEDs are placed in the front to indicate the state of the algorithm between tests. The red LED signifies the robot is learning while the green LED lets the user know the test is complete and that the robot is repeating the learned policy. An HC-05 Blue-

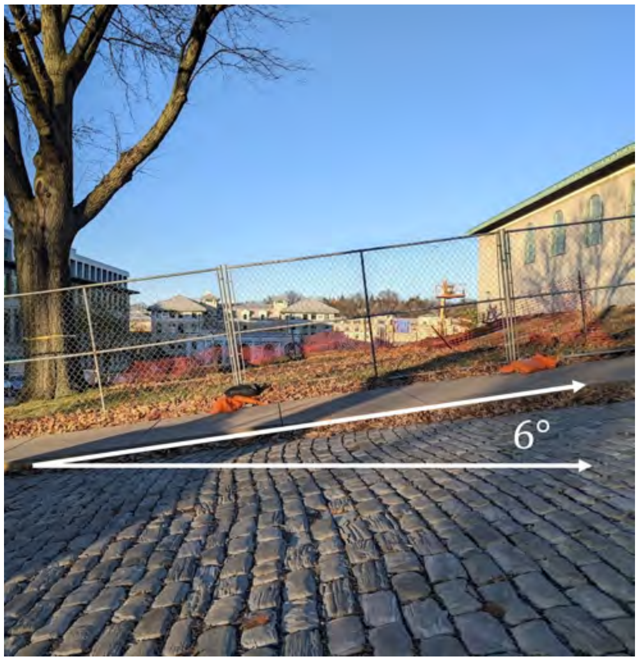


Figure 4. Hill with steep concrete gradient, used for incline and decline experiments.

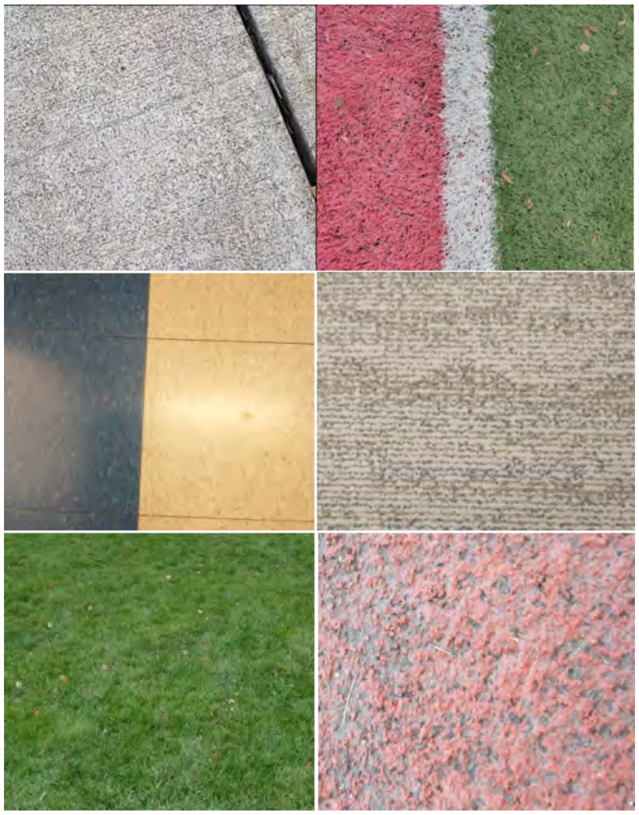


Figure 5. The images of the different terrain selected, in order white concrete, turf, linoleum, carpet, grass, and rubber track

tooth module was connected to the microcontroller to wirelessly broadcast serial data in real time this allows the user to run tests and obtain data while the robot is running. The full hardware setup of the agent can be seen in Fig.1 While between trials the Android the app Bluetooth Terminal by Juan Sebastian and Ochoa Zambrano was used to generate the Q-table display on our phones. The total weight of the untethered Qbert robot is 0.915kg. For weighted tests we used an aluminum sled weighing 58g and with a foam bottom. The weighted sled can be seen in Fig. 8

4.4. The Algorithm

Qbert uses a Q-learning algorithm written in C++ in the Arduino IDE. The program starts by turning on a red LED to signify to the user the robot has started. The program starts by initializing a 4X4 Q-table and a 4X4 reward table. The 4X4 Q-table represents the possible combination of position states and actions the robot can be in. The 4X4 reward table can be used for selectively rewarding or punishing actions for a given position state and is the same size as the Q-table. In setup there is also a 4X4 action table created, this is used for the robot to match selected actions with states on the Q-table. The empty Q-table is set via Bluetooth to the user's phone to verify the initialization process was successful and the Q-table is indeed empty (Fig. 9a).

Next a learning length is selected, this is used to determine the number of times the program will iterate through the whole table. In the case of our tests we chose 100 iterations. A random action is then selected. The number selected then enters a series of if/else statements which drives the servos to select a particular motion based on the ac-

```
Initialize Q-table
Initialize Reward Matrix
Initialize Actions Table

Loop
    For all s ϵ S
        For all a ϵ A
        Q(s,a) ← E[r|s,a] + γ*reward + P
    V(s) ← max_a Q(s,a)
Until V(s) Converge
Loop
    Converged Policy
```

Figure 6. The algorithm for value iteration of the q table is displayed.

Conditions	Act1	Reward1	Act2	Reward2	Act3	Reward3	Act4	Reward4
Control								
Control	N/A	All Neg.	N/A	All Neg.	N/A	All Neg.	N/A	All Neg.
Terrain								
Linoleum	2	110692	0	414526	3	482576	0	379397
Carpet	2	82129	0	311148	3	360263	0	287763
Concrete	2	80035	0	290559	3	331258	0	263432
Track	2	87321	0	313546	3	353554	0	284316
Turf	2	16946	0	71609	3	81561	0	65672
Gradient								
Incline Con.	2	87948	0	303756	3	344584	0	268063
Decline Con.	2	171010	0	619392	3	713341	0	565217
Flat Con.	2	98500	0	346355	3	394387	0	312764
Sled								
20g	2	35201	0	136976	3	152727	0	120237
50g	2	30135	0	118731	3	132568	2	108102
80g	2	24101	0	95280	3	110483	0	85762
200g	2	484	0	18947	3	20328	2	19411

Figure 7. Table of all learned policies and decided actions for terrain, gradient, and weighted sled.

tion selected. There are overarching functions to measure changes in distance or velocity of the encoders. These functions are used to select different observations from the environment. After an action is selected and an observation is recorded and the Q-table is updated using the Bellman equation. The Q-table uses the saved reward that already exists at that location, a discount factor gamma on the new reward, and a reward value from the reward table to update the new value in the Q-matrix. If an action is repeated which results in no movement, a harsh punishment is inflicted in the Q-table to discourage repeated actions.

After all iterations of the learning length are complete the Q-table is finished updating and the final Q-table with the updated values are sent to the user. The red LED turns off and the green LED turns on signifying to the user that a policy has been selected and that the agent is ready to exploit what has been learned. The final component of the algorithm is selecting the highest value positions and actions from the Q-table as the policy for the robot. These actions continue in a while loop as the robot cycles through the prescribed motions. If the motions for all positions are the same then the robot is designed to insert a new state in the middle of the matrix in case there are repeated consecutive actions occurring. A flowchart provides a visual of how the algorithm is implemented (Fig. 3).

5. Experiment

For Qbert, we wanted to understand how the Q-table and optimal policies would be affected given different environments and conditions. For each environment we let the Qbert robot operate through a full 400 actions before selecting a policy. After 400 observations were reached we inspected the Q- table and identified the selected actions for the optimal policy. In our tests we wanted to select surfaces with different coefficients of friction and gradients to ensure we were adapting to varying circumstances. As a control we ran the robot suspended with no changes in velocity. This was used to verify that the robot was not selecting a policy inherited by the algorithm itself. When suspended the robot would only be punished for repeated selected actions resulting in a random matrix of negative values (Fig. 9b.) The first environment was a hill that provided a 6 deg. gradient to test incline and decline performance. To inspect changes in gradient on the Q-table we used the same surface material, in our case concrete, on flat ground and an equal but opposite change in ascent angle. The hill used for this experiment is seen in Fig. 4 Next, Qbert was tested on various surfaces; concrete, turf, linoleum, rubber, grass, and carpet. The reason for selecting these surfaces was to obtain a wide variety of coefficients of friction. All tests were successful with the exception of grass, as the robot was unable to move. The six surfaces tested can be observed in Fig. 5. The final experiment were policies generated by having a weighted sled on a linoleum floor. The aluminum sled carried 20g, 50g, 80g, and 200g and can be seen in Fig. 8. In these experiments we removed the rubber feet to have a plastic to linoleum foot contact. This would encourage slipping of the feet while the agent dragged the sled.

5.1. Results

After experiments were carried out in various environments, all of Qbert's actions converged on the same policy

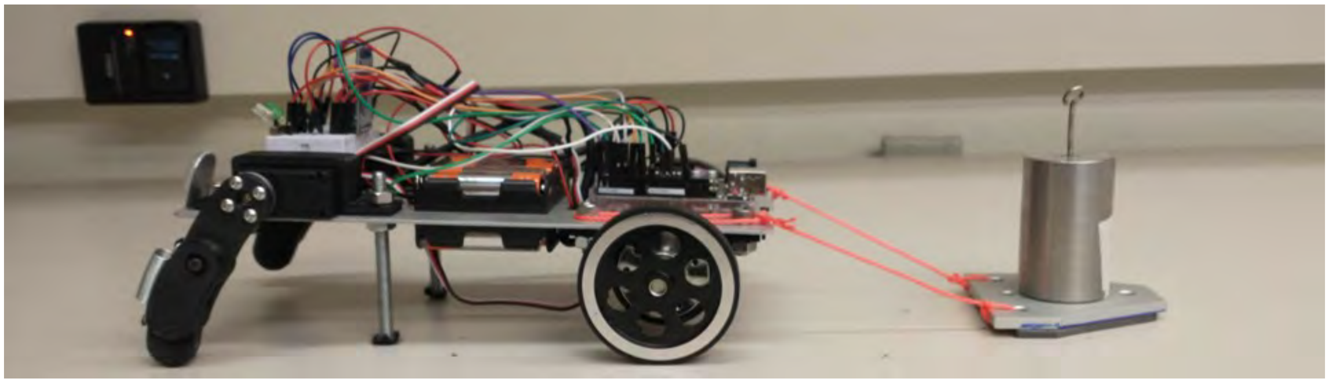
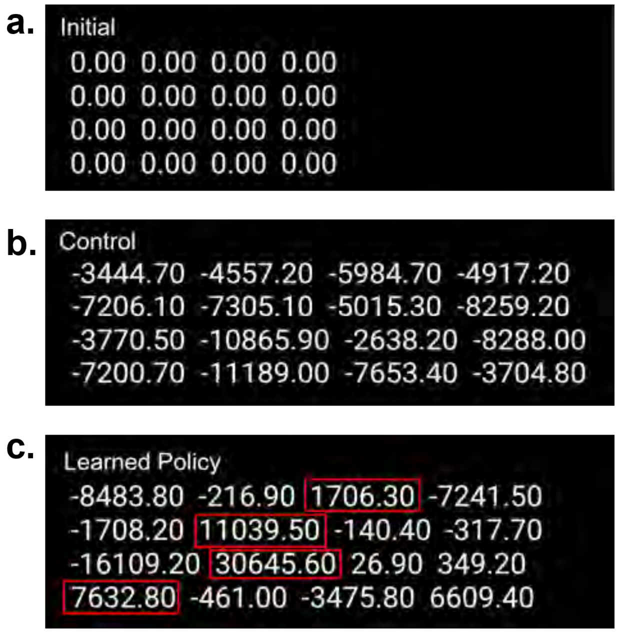


Figure 8. A sled was attached to Qbert to explore other learning policies.



a. Initial

0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00

b. Control

-3444.70	-4557.20	-5984.70	-4917.20
-7206.10	-7305.10	-5015.30	-8259.20
-3770.50	-10865.90	-2638.20	-8288.00
-7200.70	-11189.00	-7653.40	-3704.80

c. Learned Policy

-8483.80	-216.90	$\boxed{1706.30}$	-7241.50
-1708.20	$\boxed{11039.50}$	-140.40	-317.70
-16109.20	$\boxed{30645.60}$	26.90	349.20
$\boxed{7632.80}$	-461.00	-3475.80	6609.40

Figure 9. a. The starting Q-Table. b. A Q-Table generated without movement for control. c. Q-Table showing learned policy (outlined in red).

with the exception of the control tests. The optimal action policy produced by the algorithm in Fig. 6 was [2 0 3 0]. This produces a swimming like motion where the robot iterates through moving each arm back and forth. The control test with no input from the encoders did not converge on a policy as there was no observations from the environment as the agent explores. Although the policy converged to the same actions on the Q-table, the Q-tables themselves demonstrated wildly different values depending on the conditions. This shows that the algorithm and Qbert's actions adapted to the given environment. With respect to the different gradient tests, the Q-table on the incline produced smaller rewards. This is a result of the low velocities generated as the agent was fighting gravity by going uphill. Qbert weighs 0.915kg. When climbing or descending the 6deg gradient in Fig. 4 the agent is effectively assisted or hindered with an additional 1N of force. While in decline, the Q-table produced the highest values as rewards became enlarged from the robot rolling down hill and taking large strides. As seen in Fig. 4.4 the final rewards from the decent were two times larger than the accent. The flat ground produced results between the incline and decline tables. With the varying surfaces the changes in Q-table seemed to be based on whether the agent was able to move through the environment easily or with difficulty. The turf produced the lowest rewards and the largest punishments as Qbert struggled to move in the soft rubber pellets, dragging itself along. Motions that would traditionally give rewards would sometimes push the robot backwards in the turf. The linoleum produced the largest reward values, with the actions selected having the values [110692, 414526, 482575, 379397] from Fig. 4.4. This makes sense as Qbert's rubber feet and the rubber wheels had made the best traction with this particular surface. Concrete, track, and carpet all had similar selected actions values. This highlights how widely changing surface conditions changed the values in the Q-table, displaying how the agent's rewards adapted to different terrain. When a weighted sled was added and the rubber on the bottom of the feet were removed we did begin to see a slight change in policy. When the sled was weighed down with 200g and 50g the policy switched from [2,0,3,0] to [2,0,3,2]. This slight change is suspected to be for the constant shifting motion the robot would make when attempting to move the sled. This side to side motion combined with low rewards resulted in the final 0 changing to a 2. The Q-tables also incrementally had lower rewards as the weight was increased. Full Q-tables from all tests, and the Arduino code can be found in the provided supplemental data.

6. Discussion

The policy that was consistently settled upon between most trials was [2,0,3,0] from Fig. 4.4. This policy was converged on no matter the terrain, gradient, or weight carried for a given test. Our team believes this is a consequence of having such a small state-space of actions. By confining the Q-table to 4 distinct positions and 4 distinct actions the agent did not have a wide variety of policies worth converging to. This results in the same policy for all circumstances. However as seen from the control test as well as the variety of results from the Q-tables themselves we see that the agent is clearly adapting to it's given environment as the observations between tests change drastically. To better illustrate and test the adaptability of reinforcement learning future work can consist of robots that have much larger degrees of freedom and as a result much larger state-spaces. Future designs can also take in more observations from the environment to further adapt to changing circumstances.

7. Conclusion

Our team created the Qbert reinforcement learning robot. Qbert uses a Q-learning algorithm that provides rewards and penalties to reach an optimal policy. The algorithm is carried out in real time on a microcontroller. Rotary encoders provide feedback and a reward is given based on velocity of the robot. Qbert used value iteration to explore it's environment and to form an optimal policy. To verify the Q-table was truly adapting to the given environments we tested on various surfaces, gradients, and weights. Our results showed that the Q-table would modify itself to changing situations and that the optimal policy selected was [0,2,0,3] in all cases. This robot is a good demonstrator and provides a hands on application for students to build upon reinforcement learning theory.

8. Contributions

Regan, Richard, and Tuo help design the robot. Regan fabricated the robot. Richard wrote the code. Tuo, Regan, and Richard conducted experiment and all contributed to the paper.

References

[1] Y. Li. Reinforcement learning applications. *arXiv preprint arXiv:1908.06973*, 2019.

[2] E. S. Low, P. Ong, and K. C. Cheah. Solving the optimal path planning of a mobile robot using improved q-learning. *Robotics and Autonomous Systems*, 115:143–161, 2019.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[4] H. Sasak, T. Horiuchi, and S. Kato. A study on vision-based mobile robot learning by deep q-network. *Proceedings of the SICE Annual Conference*, 2017.

[5] M. Tokic, W. Ertel, and J. Fessler. The crawler, a class room demonstrator for reinforcement learning. *Proceedings of the Twenty-Second International FLAIRS Conference*, 2009.

[6] C. J. WATKINS and P. DAYAN. Q-learning. *Machine Learning*, 8:279–292, 1992.