An Experimental Analysis of the Deep Q-learning Algorithm

1st Farooq A. Khan
Computer Science dept.
Ryerson University
Toronto, CA
farooq1.khan@ryerson.ca

2nd Harun Abdi
Computer Science dept.
Ryerson University
Toronto, CA
harun.abdi@ryerson.ca

3rd Jessica Ip
Computer Science dept.
Ryerson University
Toronto, CA
ting.ip@ryerson.ca

4th Nicholas Dhanraj
Computer Science dept.
Ryerson University
Toronto, CA
nicholas.dhanraj@ryerson.ca

***Abstract*—In this paper, we perform the Deep Q-learning algorithm on the Car Racing Environment and the Pong Environment. We will show that the Deep Q-learning algorithm converges for these two environments as well as recommend ways to approach tackling the problem and how to improve our methodology.**

***Index Terms*—Deep Q-learning, Reinforcement Learning, Car Racing, Pong**

I. Introduction

Reinforcement learning (RL) is a machine learning method in which an agent learns to optimize its sequence of actions in order to maximize the total future reward it gains in its environment. The reinforcement learning agent does not start with perfect knowledge of its environment but learns through executing actions, receiving rewards, and making observations about the environment. In this report, we will examine how we applied reinforcement learning to solve the 'CarRacing-v0' and 'Pong-v0' environments provided by OpenAI Gym.

The Car Racing environment is a top-down car racing game in which the agent learns from pixels much like the Atari games that were trained in the Nature paper [6]. The Pong environment was used as a starting point to understand the nature of the Deep Q-learning algorithm. To gain a robust understanding of RL, and to have a comparative example, we worked on both environments in parallel.

A. Reason

The reason we chose to solve the Car Racing project was to investigate how the RL algorithms perform in more complicated environments. We wished to explore how the methods are generalized by observing how the model learns in this environment. The Car Racing environment was a good choice due to it being a challenging problem to solve, but not too overwhelming, like our original project, Halite IV[1] which had a much larger scale as well as more outcomes.

In conjunction with solving the Car Racing environment, we used the Pong environment as a starting point to understand the intricacies of how a RL problem is solved. This served as a great opportunity to see whether or not the convergence of the algorithm is dependent on the environment.

[1]See Appendix A for a detailed explanation of why we abandoned the project.

B. Lessons

Much of our understanding of the fundamentals of RL and Deep Q-learning is built from class lectures, practical experience from assignments, reading multiple papers, and working on this project. This project has made us more comfortable with applying RL algorithms to problems. We have learned a lot of core technical and character building skills through conducting experiments in training our models and perseverance through solving complex challenges.

Lessons learned include working in a collaborative environment on a big semester-long project and extrapolating the solution from one project to another. We ran into many issues during this project, such as having infeasible solutions to our first project idea, Halite IV, tracking down significant and complicated bugs, and having to face pressure and adversity of having to solve a problem that we have had limited exposure to. We have also learned how much of an impact incremental changes can have on an outcome. For one, hyper-parameter tuning is a tool that significantly cuts down our training time and gave us better results.

C. What is to follow?

This report will seek to define the Deep Q-learning algorithm and how it is structured to solve complex systems. We will also discuss the practical problem faced with setting up an environment and lastly, the results achieved.

II. Problem Statement

A. Car Racing Environment

The Car Racing environment is a top-down racing environment in which the agent must drive a car around a track to maximize the reward [7]. In order for an episode to terminate, a full lap, in which all tiles are visited, must be completed. These tracks are randomly generated at the beginning of each episode, and only a small portion is visible in a single frame. The longer the car takes to complete this task, the less reward it will receive. The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles in track [7]. The Reward in terms of time is defined as follows,

$$R(t) = 1000 - t/10 \tag{1}$$

where t is the number of frames it takes the car to reach the end of the track [1]. A sufficient solution, as stated by the documentation, is an agent that receives a total reward of 900 or above.

The state-space of this environment is an RGB 96x96x3 frame of the game [7].

The action-space is a continuous tuple of steering, acceleration, and deceleration whose values range $(-1, 1)$; $(0, 1)$; $(0, 1)$ respectively.

B. Pong Environment

Pong is a top-down table tennis-themed game in which the agent must learn to move their paddle to hit the ball past the opponent's paddle in order to score, while preventing the opponent from scoring. An episode terminates when either player reaches a score of 21. In the Pong environment, a reward of +1 is achieved when the agent scores a point and -1 when the opponent scores a point. Otherwise it gets a reward of 0. A Pong episode's cumulative reward ranges between -21 and +21 [5].

The state-space of this environment is an RGB 210x160x3 frame of the game stored in an array where each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from $\{2, 3, 4\}$.

The action-space is the same six actions for any Atari game:

```
ACTIONS = [
        'NOOP', 'FIRE',
        'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE'
        ]
```

The 'NOOP', and 'FIRE', are the do nothing actions. Actions at index 2 and 4 make the paddle go up while 3 and 5 make the paddle go down.

III. Method and Models

A. Preprocessing

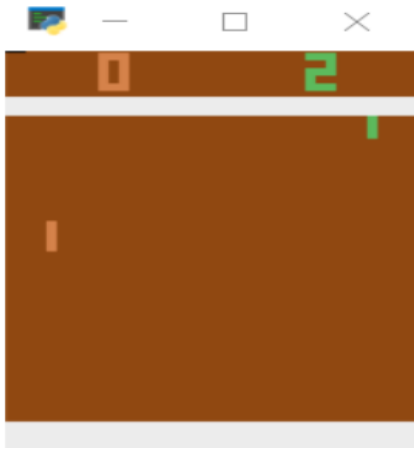
When preprocessing the environment we were concerned with two things. We aimed to balance the trade-off between the efficiency and the effectiveness of this step. The OpenAI Gym wrapper class allowed us to override the step function; however, for training, the preprocessing happens at each step, which if done inefficiently, would have significantly increased the amount needed for us to detect a signal to see if our agent is learning [1]. In our pipeline we must make the preprocessing step efficient while retaining as much information as possible from the state.

Our first preprocessing step was to grayscale the environment reducing the dimensions for both the state spaces of the Pong and Car Racing environments.

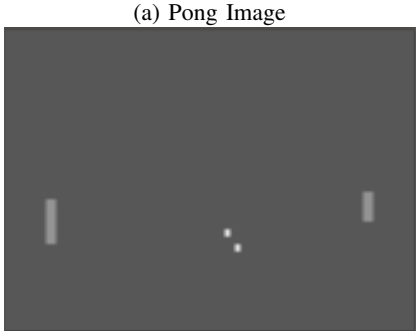
This following equation is how we grayscale the frame:

```
state = state[:, :, 0] * 0.299 + state[:, :, 1] *
    0.587 + state[:, :, 2] * 0.114
```

Our second preprocessing step was resize the frame to be smaller. Our third preprocessing step was to crop the frame, if needed. We have added these last two steps as we have learned that the model did not need the entire frame to learn.



(a) Pong Image



(b) Preprocessed Pong Image

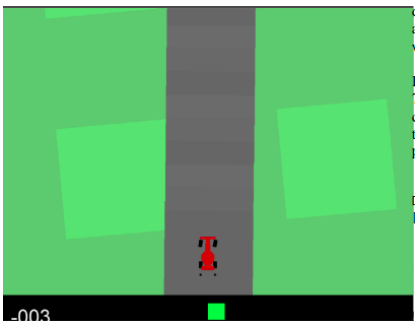
Fig. 1: Images of Pong Environment

1) Pong Preprocessing: The Pong environment is a 210x160x3 frame and after applying our grayscale formula, it was reduced to a 260x160x1 frame. We have also resized the Pong environment frame, while removing the score bar from the top. The shape of the state we used for training was 80x80x1 (Figure 1a and 1b).

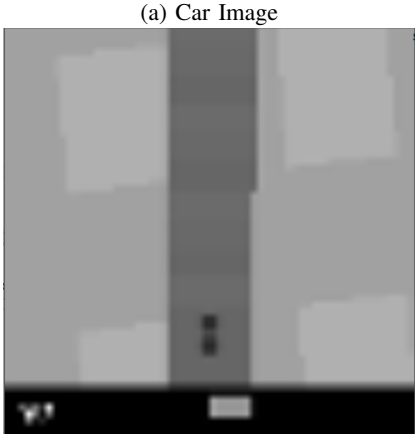
2) Car Racing Preprocessing: The Car Racing environment is a 96x96x3 frame and after applying our grayscale formula, it was reduced to a 96x96x1 frame. We have also resized the Car Racing environment, making the shape of the state we used for training a 60x60x1 frame (Figure 2).

Since we had started off with the Pong environment and modified it to accommodate the Car Racing environment, we had to make changes. The episodes in the Pong environment are guaranteed to terminate eventually as the score will get to 21.

Episodes are not guaranteed to terminate in the Car Racing environment as the agent needs to complete a lap around the track for the episode to terminate. Early on in training our model we realized that the episodes would not terminate as the agent would act randomly in the environment and not be able to complete a lap. This resulted in getting an extremely low negative reward because the agent would not be able to stay on the track, and once it is off the track it would not know how to get back on. Therefore we capped the evaluation of the car to a specific time limit, and we experimented with values



(a) Car Image



(b) Preprocessed Car

Fig. 2: Images of Car Environment

between 1000 and 3000 frames. [2] This threshold was helpful because the agent would learn how to stay on the track and accumulate positive rewards.

The last difference the Car Racing environment had was that it has continuous tuples of actions. We have down sampled the actions in the Car Racing environment to be discrete. We have decided to do this to simplify the problem due to our lack of understanding of implementing continuous action spaces, as well as knowing that the agent could converge with the following actions [1].

In terms of picking how many actions to take, we started off using 7 actions: accelerate, brake, turn-left, turn-right, do-nothing, combination of turn-left and accelerate, and combination of turn-right and accelerate. We decided to simplify the problem to the following 5 core actions:

```
Turn_left    = [-1,0,0]
Turn_right   = [1,0,0]
Brake        = [0,0,0.8]
Accelerate   = [0,1,0.2]
Do_Nothing   = [0,0,0]
```

B. Deep Q-learning

The goal of any self learning agent is to maximize the rewards by performing an action given a state. Traditional RL algorithms fall short in more complex environments for many reasons. Oftentimes the state-action pairs generate very large Q-tables, which require large storage. Function approximation of the Q-tables has given rise to the Deep Q-learning algorithm. A simpler RL algorithm could not be applied to the Pong and Car Racing environments due to their sheer complexity.

Applying the Deep Q-learning algorithm was the simplest approach to solving these problems because many of the traditional means of having a Q-table was astronomically infeasible. The state space alone is a staggering $256^{96 \cdot 96 \cdot 3}$ for the car racing environment [7].

The Deep Q-learning algorithm is a model-free RL algorithm invented to provide an estimation for the Q-values of the state-action space. It is a generalization of the Q-learning algorithm in that it approximates the Q-values with a function parameterized by some weights w, removing the need to tabularise the state-action pairs. An architecture of the Q-learning algorithm can be seen in Figure 3.

Fig. 3: Q-learning

What exactly are the Q-values? They are a representation of quality [3]. This action-value function must adhere to the *Bellman Equation* [6]. Through an iterative update the optimal Q-values, $Q^*(s', a')$, are achieved by maximizing the expected value of $r + \gamma Q^*(s', a')$ for all sequences of state action pairs.

$$Q_{i+1}(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q(s', a') | s, a] \tag{2}$$

As $i \to \infty$, we will get Q^*. The approximation is defined as follows,

$$Q(s, a; w) \approx Q^*(s, a) \tag{3}$$

In our research, we decided to follow the experimental results of the nature paper and implement a Convolutional Neural Network (CNN) for our function approximation. The architecture can be seen in Figure 4. We pass in a state to our CNN and it outputs a list of Q-values of all discrete actions. This is the same for both the Pong and Car Racing environment. This allowed us to reuse most of our code implemented for the Pong environment.

Fig. 4: Deep Q-learning

A reason why we gravitated towards the Deep Q-learning algorithm was because it generalized experiences to previously unseen states, i.e. it has the ability to generate agents that

[2] fps = 50 [7]

can act with experience in environments where they have no knowledge of the state-space [3]. Moreover, this is an extremely efficient idea because it saves on execution time. Because the neural network is very sporadic in its decision making when learning, we use a separate target-network to act in the environment. This target-network is a clone of the original Q-network with the exact same architecture and it is updated at regular intervals to adhere to more recent Q-values.

C. Architecture of the Network

A nonlinear function approximation method with parameters w, was used. Two models were constructed and used on both the Pong and Car Racing environments. Both models had a history buffer size of 4, a ReLU activation function and three convolution layers. The first model was a copy of the model featured in the nature paper [4], while the second model featured two additional fully connected layers. The Python Deep Learning library PyTorch was used to train the model on a GPU on the Google Cloud servers.[3]

D. Adam Optimizer

We went with the recommendation from Dr. Farsad on using the Adam Optimizer for our Neural Network. Additional research on the topic showed why the Adam Optimizer was used over Stochastic Gradient Descent. Adam provides an optimization algorithm that can handle sparse gradients on noisy problems [8]. This is ideal for our use case because initially our agent acted randomly. Adam is also computationally efficient, and the hyper-parameters require little tuning [8]. We did not perform any hyper-parameter tuning to the Optimizer.

E. Loss Function

Mean Squared Error was used to calculate the loss for a Deep Q-learning Network. The function calculates the loss by using the sampled actions from the target-network, and the Q-values of the actions taken with the q-network. These sampled Q-values were calculated as follows,

$$Q_{samp} = r + (1 - d) \cdot \gamma \max_{a'} Q_{target}(s', a') \tag{4}$$

where d, is either 1 when the environment has ended and 0 otherwise. If the environment has ended the immediate reward is given.

F. Double Deep Q-learning

We also experimented with Double Deep Q-learning (DDQN) because DQN is very good at overestimation [4]. Research has shown that the DDQN algorithm reduces overestimation and leads to better performance. This DDQN algorithm is similar to DQN; however, the loss function is calculated differently. The greedy policy is evaluated using the q-network, but using the target network to estimate its values [4].

$$Y = R + (1 - d) \cdot \gamma Q(S_{t+1}, \operatorname*{argmax}_{a'} Q(S_{t+1}, a'; w); w') \tag{5}$$

where Y is the Q-value samples of the DDQN, w are the weights of the q-network, and w' are the weights for the target network.

[3]Visit Appendix B for a detailed implementation of our NN architecture with PyTorch.

IV. Results and Discussion

A. Struggles

When running the Car Racing environment locally on a CPU for approximately 100,000 iterations we did not see any indication that the agent was learning. We believed it needed more training; hence, our next step was to use a GPU.

We used Google's Cloud computing services to train our model, but setting up our accounts was very time consuming as this was new to all our group members. Once our accounts were set up we also ran into two major errors when trying to train on the cloud. Our first issue was rendering the environment without having a display. We were able to fix this issue after conversing with the other group working on the Car Racing environment, who directed us to the `PyVirtualDisplay` library.

Our first approach was to train the model for a day in search of a signal to know if the model is learning or not, but that did not come without challenges. Firstly, we had tested out our code on a test environment to find any bugs in our implementation. The agent was able to the optimal reward in our test environment. This gave us the confidence to integrate the Car Racing environment with our implementation of Deep Q-learning.

This leads us to second major problem. When running the Car Environment, training for anything more than a couple hundred thousand iterations, on a GPU, was giving us memory errors. After rigorously testing our code we had discovered a memory leak only specific to the Car Racing Environment. This was because the environment was generating tracks randomly for each evaluation stage; however, those tracks still remained in memory and were not being used. The fix we found was to close the environment at the end of each evaluation. Once these errors were fixed we were on track to run both the Pong environment and Car Racing environment in parallel.

B. Rookie Mistakes

We trained our model for approximately one million iterations, done over a couple of days stopping at every couple hundred thousand iterations to examine the results. Our results were very random and the model seemed as though it was not learning, after retracing our steps and studying the behaviour of the model we realized that we had corrupted our weights file midway through training.

The reason the weights were corrupted was because we discretized the environment into seven actions [4]. However after a couple hundred thousand iterations, we have reverted to the five actions mentioned in Section III-A2. Due to this error, the encoding of the five action were mapped to the encoding of the seven actions, making any progress we made obsolete. To mitigate such issues we stopped making major code changes in the middle of training.

C. Initial Car Racing Model

We trained the Car Racing agent for approximately 400k iterations. [5] Unfortunately, the model was not learning how to navigate the environment consistently. The model was getting high positive rewards, but was also getting very negative rewards in some episodes. These results are shown in Figures 5a and 5b. In Figure 5a, we observed an upward trend for the average reward during the evaluation process of training; however, after training the model for 400k iterations that trend disappeared and we were left with noisy data (Figure 5b). We concluded that the model was not learning.

We suspect the model failed because it only learned to accelerate, so it would get high rewards on tracks that are more straight, but would suffer on tracks with turns, however we are not sure why the agent did not learn how to turn. An obvious fix would be to have a constant speed which would mean the average rewards in theory will increase. This would also mean that our model will have one less action which, depending on the situation, could be less desirable.

D. Revisiting the Pong Model

Our initial model for the Pong agent used the same CNN architecture as the Car Racing model. Similarly, the Pong model was slow at convergence. We trained the Pong agent for five million iterations and observed a net gain of 4 reward points. We suspected the model would converge faster if there were more weights and biases to tune. Thus, we updated the model to a more complex Neural Network (NN) architecture.

The NN architecture contains two additional fully connected layers each with 512 nodes. Reference Section III-C for more detail about our implementation. We ran this model for 1.5 million iterations and we observed significant improvement in the convergence both in terms of average reward and time (Figure 6). [6] This led us to believe that the Car Racing agent would also perform better with this NN architecture.

Next we trained the model using DDQN. The same NN architecture from the previous experiment (Section III-C) was used; however, the loss function was modified.[7] We only trained this model for 800 thousand iterations and we observed similar results to our previous experiment (Figure 6).[8]

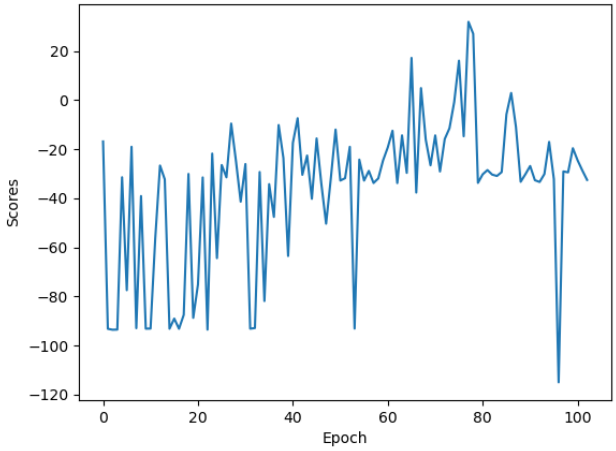
[4]These seven actions were the following: accelerate, brake, turn-left, turn-right, do-nothing, turn-left and accelerate, and turn-right and accelerate.

[5]This training was done approximately over a day stopping at regular intervals to examine the results

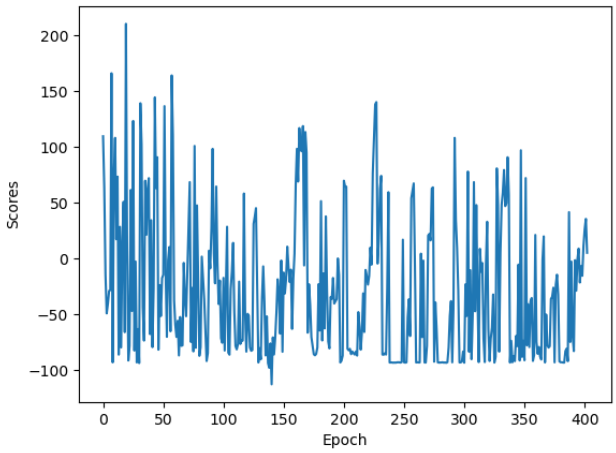
[6]An evaluation of the agent can be viewed here.

[7]Refer to Section III-F for the modified loss function.

[8]Unfortunately we have not trained the model long enough to extrapolate whether the modification to the loss function had a significant effect on the behaviour.



(a) Average reward over 100k iterations of training



(b) Average reward over 400k iterations of training

Fig. 5: Average Rewards of Car Racing Agent over 400k Training steps

E. The Car Model Revisited

We began training the Car Racing environment using the modified NN in parallel (with the Pong environment) to replicate the positive results we had observed in the Pong environment, as mentioned in Section IV-D. We trained the new model for approximately 1.6 million iterations. [9] The agent was learning as it was consistently receiving positive rewards from the track. This can be seen in Figure 7, which shows the agent is driving consistently within the environment. This consistent behaviour can be seen here.

The above video shows that the agent was able to stay on the track and turn when needed, unlike our previous iteration of the model which would only accelerate and perform random actions. The agent tends to favour the edge of the track rather

[9]This training was done over a 2-3 days stopping at regular intervals to examine the results.

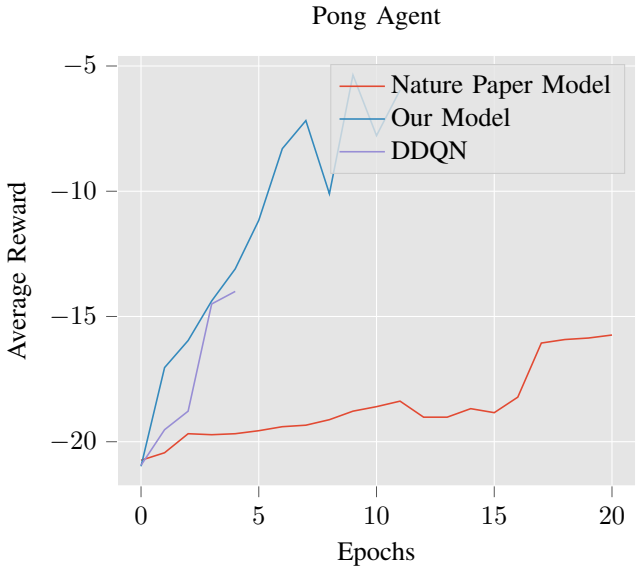


Fig. 6: Results of new Model of NN

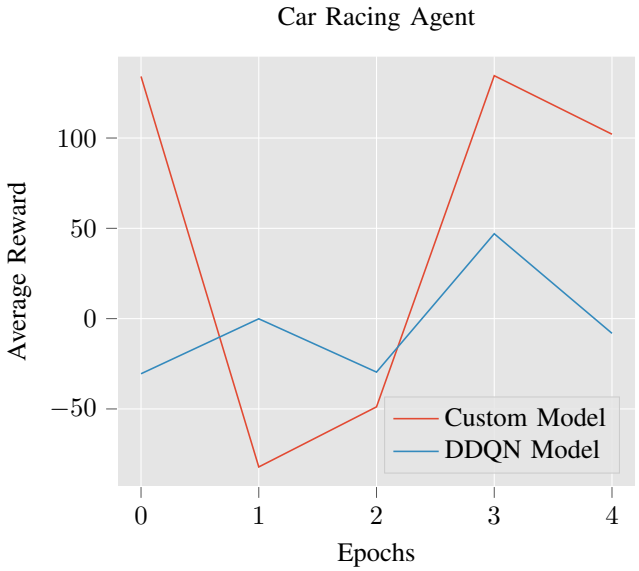


Fig. 7: Last 400k iterations of Car Racing environment

than the middle, the reason the agent prefers this behaviour is not clear to us. Although the model has a positive behaviour, there is tremendous room for improvement.

Similar to the Pong environment, we trained the Car Racing model using DDQN;[10] however, we attained inconclusive results. We believe this model is not mature enough and needs more iterations (Figure 7).

F. Summary Of Progress

Table I compares the average rewards of our agents in their respective environments. As you can see, while neither agent can compare a human's performance, our second model has greatly improved in comparison to the results from our first nature implementation.

Agent	Car Racing	Pong
Random	-100	-21
Human	700	15
Our Nature Implementation	-20	-10
Nature Model with 2 additional Fully Connected Layer	120	-5.9
DDQN	-15	-14

TABLE I: Reward comparison of environments and agents

G. What Next?

We suspect the car model had limited range in the actions it could take, only being able to do one type of movement at a time. The model could only accelerate, brake, turn-left, turn-right, or do nothing. This made the model suffer on turns as it could not turn and accelerate simultaneously. We believe reverting to the initial 7 actions, mentioned in Section III-A2 would help mitigate this issue as the car would be able to handle turns faster. Using discrete actions in this environment has made the agent act rigid, another experiment would be using continuous actions, which also has the potential to make the agent act more smoothly.

Unfortunately, our time was cut short when experimenting with the Pong and Car Racing environment. We hoped to implement and experiment more with the DDQN. If we were to repeat this experiment again given our current understanding, we would look deeper into the Dueling Double Deep Q-learning algorithm as well.

V. Implementation and Code

Our solution to this problem came from a combination of reading research papers [2] [3], and optimizing the provided code for assignment 3 to best suit the environments. The assignment 3 code was very helpful as it provided us with a Replay Buffer and the implementation of Deep Q-learning algorithms. We have also referenced the Car Racing leaderboard papers published on the OpenAI Gym site. [11]

Python packages used include PyVirtualDisplay which allowed us to train our models on the cloud, OpenAI Gym which allowed us to extend our understanding of the Pong environment and Car Racing environments, and PyTorch which is what was used to implement the Deep Q-learning algorithms.

References

[1] P. Aldape and S. Sowell, "Reinforcement Learning for a Simple Racing Game", Stanford University , 2018.
[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", Nature, 2015.
[3] J. P. Rodriques, M. A. Henriques and V. Mendes, "Developing an Intelligent Self-Driving Race Car using Deep Reinforcement Learning", TÉCNICO LISBOA, 2019.

[10] The Car Racing environment shared the same NN architecture and loss function as the Pong environment and was trained for 800 thousand iterations.

[11] OpenAI Gym Leaderboard

[4] H. van Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning", Nature, 2015
[5] Ksitiz. "Deep Reinforcement Learning for Atari games aided with human guidance", Stanford.
[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. "Human-level Control Through Deep Reinforcement Learning", Nature, 2015.
[7] OpenAI gym, CarRacing-v0. https://gym.openai.com/envs/CarRacing-v0/
[8] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization", Cornell University, 2017.

ACKNOWLEDGMENT

We would like to acknowledge our TA Matthew Kowal and our professor Dr. Nariman Farsad who played a significant role in helping us complete this project. Mathew guided us throughout the semester giving us feedback on how to move forward with our research project. He helped us greatly with both technical and moral support, guiding us with new ideas and a positive attitude.

Dr. Farsad helped us with the initial implementation of the Deep Q-learning algorithm and the initial implementation of the Pong environment. He was always available and willing to make accommodations for extra office hours.

APPENDIX A
HALITE IV

Initially, our group had chosen to work on Kaggle's Halite IV environment, a spaceship game that pitted 2 to 4 algorithms against one another in a competition to collect the most halite. [12] The goal of the agent is to maximize the reward, which in this case was to gather the most halite. Since Halite is a competition, the player with the most halite at the end of the game, which lasts 400 rounds, is the winner.

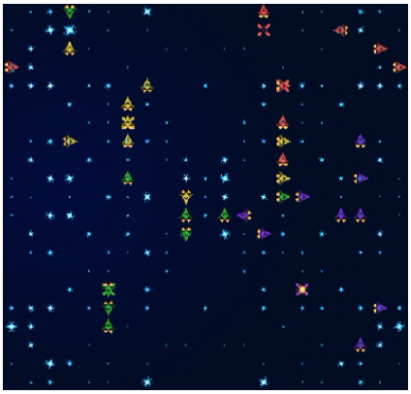


Fig. 8: Screenshot of Halite game

At the beginning of the project we were ambitious as we were early on in the semester, but as the semester went on and we had learnt more we had run into multiple problems and had realized as a team that this environment was too complicated. In Halite, each ship had 4 movement actions and the ability to mine halite. When enough halite was gathered more actions appeared, as ships got the option to create shipyards, and shipyards with enough halite could spawn new ships. With more and more ships, the state space would get more and more complex. In addition, the Halite game was meant to be played against different opponents, each with different strategies. Unlike the Car Racing and Pong environments, this meant that the Halite environment was unpredictable, making it hard to learn a suitable strategy for winning.

Our first intuition was to greatly simplify the problem, as we have shrunken it to be a single player game instead of a competition, and had disabled the ability of creating new shipyards and spawning new ships, meaning now there would be only one player with a single ship freely mining halite without any competition, but even then the environment was very difficult to work with and we could not find many relevant papers and demonstrations to guide us. As such, we decided to abandon the Halite project for the Car Racing environment.

[12] A screenshot of the game can be seen in Figure 8.

APPENDIX B
NEURAL NETWORK

Below is a more detailed output of CNN from the nature paper.

```
Sequential(
 Conv2d(4, 32, kernel_size=8, stride=4, padding=122)
    )
 ReLU()
 Conv2d(32, 64, kernel_size=4, stride=2, padding=41)
 ReLU()
 Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
 ReLU()
 Flatten(start_dim=1, end_dim=-1)
 Linear(in_features=409600, out_features=512, bias=
    True)
 ReLU()
 Linear(in_features=512, out_features=6, bias=True)
)
```

Below is our custom model.

```
Sequential(
 Conv2d(4, 32, kernel_size=8, stride=4, padding=122)
 ReLU()
 Conv2d(32, 64, kernel_size=4, stride=2, padding=41)
 ReLU()
 Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
 ReLU()
 Flatten(start_dim=1, end_dim=-1)
 Linear(in_features=409600, out_features=512, bias=
    True)
 ReLU()
 Linear(in_features=512, out_features=512, bias=True
    )
 ReLU()
 Linear(in_features=512, out_features=512, bias=True
    )
 ReLU()
 Linear(in_features=512, out_features=6, bias=True)
)
```