Deep Reinforcement Learning for Super Crate Box and Atari Games

Aaron Koenigsberg
koeni296@umn.edu

Brandon Voigt
voigt227@umn.edu

Chen Hu
huxxx853@umn.edu

1 Introduction

For this project we attempted the novel application of Deep Reinforcement Learning (DRL) to the game Super Crate Box [1]. Due to a myriad of difficulties stemming from the lack of a convenient training environment for the game, we determined partway through that best option for our project was to switch focus. Instead we implemented a Deep Q-Network (DQN) in PyTorch which learns to play games from the OpenAI Gym library [2]. We used two refinements of the basic DQN algorithm called Double DQN and Dueling DQN, and tested our implementations against the game Breakout.

The study of how intelligent agents can be implemented in a game has practical applications to the game industry as a whole. Many games include AI players that can work with or against a human player; however, these AI players often lack a realistic feel. Usually, the behavior of the AI is dictated by a simple algorithm that eventually becomes predictable. By creating an agent with a neural network, we can make our agent feel more unique and realistic. Additionally, DRL agents can surpass human performance, so they can be used to create adversaries which are stronger than another human could be. Even though Super Crate Box and the Gym environment games are relatively simple, the DRL algorithms that we explored can also be applied to more complex games.

This paper summarizes the background related to DQNs, then reports on the successes and failures of our attempts with DRL relating to Super Crate Box and the OpenAI Gym environments.

2 Background

Reinforcement Learning (RL) is a branch of machine learning in which agents seek to maximize their cumulative reward in a given environment. They accomplish this by taking different actions depending on the state of the environment [3]. RL has been applied to many research areas including games, self-driving cars, finance, healthcare, and robotics [4].

One type of RL algorithm is called Q-learning, in which the goal is to estimate the function Q, which maps the current state and possible actions to future rewards. Maximizing Q is therefore the same as maximizing the cumulative reward [3]. For a set of states S and set of actions A, with $s_i \in S$ and $a_i \in A$,

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

[5]. There are techniques for learning Q without using deep learning; one such technique is learning a Q-table. A Q-table is simply a map from a discrete set of states and actions to a reward. The state must be discretized (if it is not already) to obtain a finite set of state-action pairs. Q is then learned as the agent explores its environment and experiences rewards for actions. However, as the dimensionality of states and actions increases, the size of the Q-table increases exponentially. This

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

causes problems with memory usage and algorithmic instability, so the naive Q-table approach may not be effective for large-scale problems [3].

An alternative method which does not explicitly learn Q is to use traditional deep learning techniques. One could record human experts playing a game, obtaining a large dataset of game states and "correct" actions given that state. Then, using traditional neural networks and that dataset, one could train a model to output an action based on that data. Although this technique could theoretically train AI to play a game, it also has a few pitfalls. For one, the AI will be limited in performance to the skill of the player from whom the data came – it would be scarcely impressible to achieve better than human-level performance. Secondly, the AI may not generalize well to new situations that it has not seen before. For instance, a new level in the game may have a new feature which the AI has never been exposed to.

2.1 Deep Q-Networks

Compared to the incapability of traditional Q-learning methods and deep learning techniques, the merge of those two methods, which is called deep-Q-network (DQN) method, helps to avoid some of their issues. This technique was introduced by DeepMind Technologies in 2013, when they published a breakthrough paper which applied deep neural networks to seven different Atari games. Their AI was better than any other contemporary AI at six of these games, and even surpassed human experts at three of the games [6]. The DQN method accepts any type of input, but commonly uses only the pixels and game scores as inputs. Then it uses deep learning techniques to attempt to map these inputs to their Q values for each possible action. Compared to traditional model-free RL methods, which are either quickly made obsolete as the dimensionality of states and actions increases, or need to collect a huge amount of data from human experts, DQN scales effectively and also generalizes effectively [7]. Because of these advantages, we are using DQN as our fundamental method.

The DQN is one of the more studied areas of reinforcement learning, and as such multiple improvements to the standard DQN exist. Three such improvements are Double DQN [8], Dueling DQN [9], and prioritized experience replay [10].

2.2 Double DQN

The Double DQN differs from the original DQN method proposed by Deepmind in that it uses both the target network and the online network in its target function. Formally, we can denote the value function for deep Q-learning as $Q(s, a, \boldsymbol{\theta})$, where s is a state, a is an action, and $\boldsymbol{\theta}$ denotes the parameters of the network. The target function Y_t of the original DQN can be written as

$$Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a, \boldsymbol{\theta}_t).$$

At each step, we use a greedy policy by choosing to take the action that maximizes Q. So this function is equivalent to

$$Y_t = R_{t+1} + \gamma Q(S_{t+1},\ \text{argmax}_a Q(S_{t+1}, a, \boldsymbol{\theta}_t),\ \boldsymbol{\theta}_t).$$

However, this causes problems because we use the same network to choose an action and to evaluate its value. It can be shown that this approach leads to an overly optimistic estimate of Q, which can cause suboptimal results in practice [8]. The Double DQN approach decouples the action selection step from the evaluation step by using a different network for each step. This can be written as

$$Y_t = R_{t+1} + \gamma Q(S_{t+1},\ \text{argmax}_a Q(S_{t+1}, a, \boldsymbol{\theta}_t),\ \boldsymbol{\theta}_t^{'})$$

where the online network parameters $\boldsymbol{\theta}_t$ are replaced by the target network parameters $\boldsymbol{\theta}_t^{'}$ for the evaluation step.

2.3 Dueling DQN

The Dueling DQN method attempts to increase the sample efficiency of a standard DQN by partitioning Q into two sections. The first section is the value of the state itself, and the other section is the advantage given by each of the actions, were they to be taken. These two sections are implemented as separate sets of fully-connected layers. The estimated state value is a scalar $V(s, \boldsymbol{\theta}, \boldsymbol{\beta})$, and the estimated advantage is a vector $A(s, a, \boldsymbol{\theta}, \boldsymbol{\alpha})$, where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ denote the parameters of the hidden

layers for the advantage and the state value, respectively. These are combined in the last module of the network to estimate Q as

$$Q(s, a, \boldsymbol{\theta}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = V(s, \boldsymbol{\theta}, \boldsymbol{\beta}) + \left(A(s, a, \boldsymbol{\theta}, \boldsymbol{\alpha}) - \max_{a' \in A} A(s, a', \boldsymbol{\theta}, \boldsymbol{\alpha}) \right).$$

This approach encodes the knowledge that some states may be more valuable then others, which allows the network to learn this information more effectively. No matter which action is taken, the state value of the network will always have a gradient. Normally with a DQN, only the taken action will have a gradient, so the dueling DQN improves the sample efficiency of the algorithm [9].

2.4 Prioritized Experience Replay

Prioritized experience replay attempts to increase sample efficiency by adjusting the method by which experiences are chosen for replay. In certain environments, the reward may be distributed in such a way that the network benefits more from replaying certain experiences over others. Therefore, intelligently selecting which experiences to replay should increase the speed at which a model can learn [10].

2.5 Deep Neuroevolution

One last way to train a neural network, which differs significantly from the previously mentioned methods, is a traditional genetic evolution method dubbed deep neuroevolution [11]. This method generates a population of agents, tests them, then selects the best from each generation to seed then next. By generating the new population based off the elite performers in the previous generation, the population should converge to elite performers.

For Super Crate Box we attempted to implement a Double DQN and the deep neuroevolution moethds, before we decided to focus on the OpenAI Gym environment Breakout game. For Breakout we implemented the Double DQN and Double Dueling DQN methods.

3 Super Crate Box

In order to apply DRL to Super Crate Box, our first major task was to set up a basic RL environment for the game. Our approach was modelled on the functionality of the OpenAI Gym library [2], which allows the user to directly track the following information:

1. Current game state
2. Set of possible actions
3. Reward function
4. Whether or not the game is finished.

For Super Crate Box, the current state, score/reward, and "game over" status are all represented by the pixels of the game screen, an example of which is shown in Figure 1. The possible actions – moving left, moving right, jumping, and shooting – are input as key presses. Therefore, an agent playing the game must be able to examine the pixels of the screen in order to obtain the 4 values listed above, and must be able to emulate key presses in order to control the character.

3.1 Successes

Because Super Crate Box does not having an existing implementation of an analogous Gym environment, it is difficult to work with for a few different reasons which we will go over in the Continuing Issues section. Before that, we would like to highlight the things that went well.

3.1.1 Implementation of Agents

As a baseline, we were able to implement a simple agent that chooses an action randomly, and can interact with the game window using the Win32 API [12] to emulate key presses. Next, we

Figure 1: An example image from the game: Super Crate Box

implemented a full DQN pipeline which examined the pixels of the screen in order to obtain the current game state, current score, and whether the game was over or not. Also, the DQN was able to select actions based on the current game state, so the entire DQN pipeline was functional. However, we were not able to train the agent to achieve better than random performance.

3.1.2 Score Extraction

We wanted to reward our agent for collecting crates, since that is the primary score metric of the game. However, since the score is represented as pixels on the screen, extracting it can be tough. We tried OCR using the Python interface for Google's Tesseract software [13], but it had difficulties with the font and low resolution. Because Super Crate Box is a pixel-art, low-resolution game, the font is blocky and this likely caused issues for Tesseract. We attempted to preprocess the image by resizing and applying a Gaussian blur to make this font more recognizable. This did not improve Tesseract's ability to extract the digits. Finally, we attempted to allowlist only digits and change Tesseract's configuration settings – this too was unsuccessful, so we decided to try another route at getting the score.

Before we present our current solution, allow us to introduce two more confounding factors. Score extraction is made more complicated due to "screenshake" and enemy interference. Screenshake is a technique for game developers to emphasize the impact of in-game animations, where the entire screen is rapidly translated by just a few pixels, giving the impression that the game world is shaking. This causes issues for our purposes because the score is shaking as well, which means that on any given frame, the score is not necessarily at the exact same pixel coordinates. The second confounding factor is the introduction of enemies to the game world. When enemies are added to the game, they fall into the world through an entrance where the score is located, so any score extraction algorithm needs to be robust to irrelevant pixel changes, such as an enemy dropping through the score.

Our solution to the score extraction problem relied on taking multiple cropped frames as a group and making comparisons among them to discard irrelevant information. When a new frame differed in a significant way from the other frames in an area near the score, we assumed that the score changed, and registered that the agent had collected a single crate. Our algorithm was reasonably robust to both false positives and false negatives, but sometimes a particularly poorly placed enemy would cause a misreading.

3.1.3 Action Set

In order for the agent to meaningfully travel and interact with the environment, we needed to fully represent its actions in an action set, from which the agent could select its actions at each state. Unfortunately this wasn't as simple as we initially thought, due to two constraints. First, the agent must be able to hold down a button across multiple states, and the agent must be able to perform multiple actions at once.

In order to fully traverse the environment, the agent must be capable of jumping at least a certain height. This means that at each new state, the agent must have the ability to hold down the jump

button. If the agent re-presses the jump button instead of holding it down, it will jump many times but only very short jumps instead of being able to jump once at a higher height. Additionally the agent needs to be able to move left and right while jumping, or else it will fall directly down, instead of being able to jump up and onto another platform. The agent will likely also benefit from being able to attack while moving, and possibly attack while moving while jumping!

We attempted to satisfy both constraints by pausing the agent for a fraction of a second while it was holding down a button. The game would continue uninterrupted, carrying out the agent's selected action. Once the time had passed, the agent would be unpaused and allowed to select a new action. The down side of this was that the agent had to commit for that amount of time to perform that action, even if that meant it would surely get a game over. We switched to allowing the agent to select any number of actions using a pseudo-boolean array, where if the action's index was true, the button would be pressed/held, and if it was false, the button would be released. However this is not traditional in terms of standard DQNs, so we ran into trouble implementing it.

The solution we settled on was a traditional DQN which selected a single action, where the set of actions included certain combinations of key presses. Since the agent needed to be able to jump and move at the same time, the action space we defined was $\{Left,\ Right,\ Jump,\ LeftAndJump,\ RightAndJump,\ Shoot\}$. It seemed to work in terms of the agent's expressibility. However, this action space did not allow the agent to move and attack at the same time, since incorporating this ability would have resulted in a fairly large action space.

3.1.4 Reward Function

Another problem that must be overcome for all RL implementations is determining an appropriate reward function. The main metric of success in Super Crate Box is the collection of crates. This serves as your final score when the player inevitably dies. Therefore we certainly wanted to reward the agent when a crate is collected. We settled on a base reward of 1 for the frame in which a crate is collected. In order to encourage crate collection, and avoid training an agent whose only goal was to survive, we applied a reward of $-.01$ for each frame where the agent did not collect a crate. If this penalty did not exist, the agent would potentially spend a large amount of time between trying to get crates, since spending one second would be the same reward as spending ten seconds. Finally, we wanted to apply a steep penalty when the agent is killed by an enemy, since that ends the game. We initially decided on a penalty of -100. However, tuning the reward function is often a challenging problem, and we suspected that a different reward function could have led to better results.

3.1.5 Deep Neuroevolution

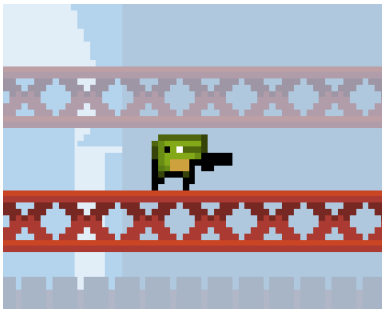
We also explored the idea of using a deep neuroevolution method instead of the double DQN. With this method, we were able to train an agent to survive in the game for a long period of time. Unfortunately, the elite agents we found tended to use a naive strategy of staying in the same place while repeatedly jumping and shooting. So when we tried to adjust these agents to move around the screen and actually collect boxes, this approach was unsuccessful.

3.2 Continuing Issues

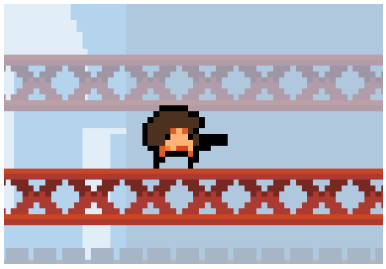
There were a few issues with Super Crate Box that we were not able to overcome and contributed to our decision to switch focus. In this section we go over a few of them.

3.2.1 Character Model Randomization

One issue that we were not able to disable in the game was the variation in character models. Each time the game begins, it selects a different visual which represents the player character, as shown in Figure 2. Unfortunately we cannot disable this, so our DQN would have to learn that the character can look different, which may have been one of the issues that made it more difficult to create an effective model. While we believed this to be within the realm of possibility with our DQN, other issues compounded on this uncertainty and ultimately made us decide to change focus.



(a) A character model that looks like a crocodile



(b) A possible character model that looks like a man

Figure 2: Two possible character models

3.2.2 Choosing a Reward Function

We considered a few different ways of adjusting the reward function to improve the agent's ability. First, we could consider the distance to the next crate as a factor in the reward function. This would introduce the challenge of extracting the agent's position from the game world, but it would result in a reward function which could guide the agent toward getting more crates. Another potential consideration was the number of enemies, since having more enemies on the screen means more chances of dying. Again, the difficulty in including this information would be extracting it from the pixels of the screen, but it would allow us to encourage the agent to attack enemies, which is good for the long term survival and performance of the agent.

While these seemed like promising methods to improve the performance of our DQN, due to the inconveniences of working with the raw pixel input, it wasn't feasible. For instance, locating the player character on the screen was a difficult task. One tactic would be to scan all the pixels for a specific color, assuming there is a unique color in the player model. However, due to the issue above where the character model can vary, we could not assume anything about the appearance of the player. Another tactic would be to keep track of the player model as a separate representation of the game state, which we would update based on the action the agent selects and the world geometry. However this was also infeasible due to the amount of game logic that we would need to duplicate, as well as the possibility of drift compounding slowly.

3.2.3 Frame Per Second Lock

The final nail in Super Crate Box's coffin was the framerate lock. Since Super Crate Box has to run as a program on the screen, its framerate is locked to 60 FPS. Even for very simple games, a huge number of frames may be needed before DRL can achieve good performance. For example, DeepMind trained their DQN on 10,000,000 frames [6]. Super Crate Box is by no means complicated, but it is more complicated than a game like Breakout, for instance. Therefore at a conservative estimate of 10,000,000 frames, at 60 frames per second, the training time would be two straight days. Furthermore, this assumes the DQN is running at 60 frames per second and that it lines up perfectly with the refresh rate of the monitor. Unfortunately that is likely not the case, and causes the projected training time to inflate even more. Due to this inconvenience, the development of the DQN would be extremely problematic, since we would not be able to observe whether or not our DQN is working until more than two days after we tried to test it.

4 Atari

In light of the inadequacies of Super Crate Box, our group decided to shift our focus to the Atari Gym environment, where we implemented Double DQN and Double Dueling DQN agents for the game Breakout. We used examples from the PyTorch docs [14] and Keras docs [15] as a starting point, but the implementation was our own.

The DQN approach requires tuning a number of hyperparameters, including the batch size, number of frames used for exploration, learning rate, size of the replay memory, and how frequently to update

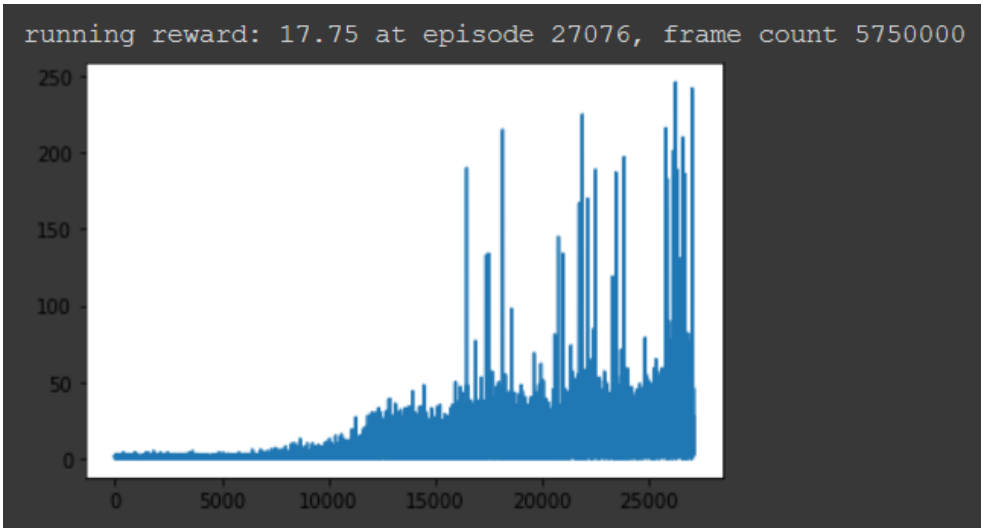


Figure 3: Double DQN agent for Breakout. Y-axis is the score earned on a given life, X-axis is the episode.

the target network. In most cases, we were able to use the hyperparameters described by DeepMind. However, due to out-of-memory issues, we reduced the size of the replay memory from 1,000,000 to 100,000. We also found that it was beneficial to gradually reduce the size of the learning rate over time.

The architecture of the neural network itself also plays a key role, of course. For this problem, we used three convolutional layers, followed by a sequence of multiple hidden layers. As described in the Background section, the Double DQN and Dueling DQN use different sets of hidden layers, which result in a different estimate of Q that is then used to optimize the model. Otherwise, we were able to use the same approach for both of these methods.

Two additional refinements of the environment were necessary to train our agents effectively. We implemented a frame stacking mechanism similar to that described by DeepMind, in which the current state of the game is represented as a sequence of the last four observed frames, rather than just the most recent frame. Additionally, while a game of Breakout is not done until the player loses 5 lives, we considered each life lost to be the end of an episode. This created a more direct relationship between surviving in the game and getting a higher reward. In the following discussion, the episode number and the reward are based on a single life in Breakout.

5 Results

For Breakout, we had success training DQN agents in PyTorch. We found that the Double DQN and Double Dueling DQN methods performed similarly, with the Double DQN performing slightly better on average. This lines up with our expectations, since the benefits from the Dueling method are most noticeable when there is a large number of choices for a given environment, and Breakout only has a few possible actions. In the Dueling DQN paper [9], they also had a decreased performance specifically on Breakout.

Figure 3 shows the results for the Double DQN. We can see that there is a clear performance increase as the agent trains. The figure shown is after 5,750,000 frames, and performance could be expected to continue to increase. However, due to the time limit on Google Colab (even with the Pro version) we were not able to run the simulation for longer than that. The average reward per life at this point was 17.5, but we can see that the highest performers were much higher than that, over 200 in some cases. So there is a large disparity between the highest performers and the lowest performers. Presumably with more training, this disparity would go away.

Figure 4 shows the results for the Dueling DQN. As expected, these results are similar to those of the simpler Double DQN. The average reward was slightly lower at this point, but the best performer

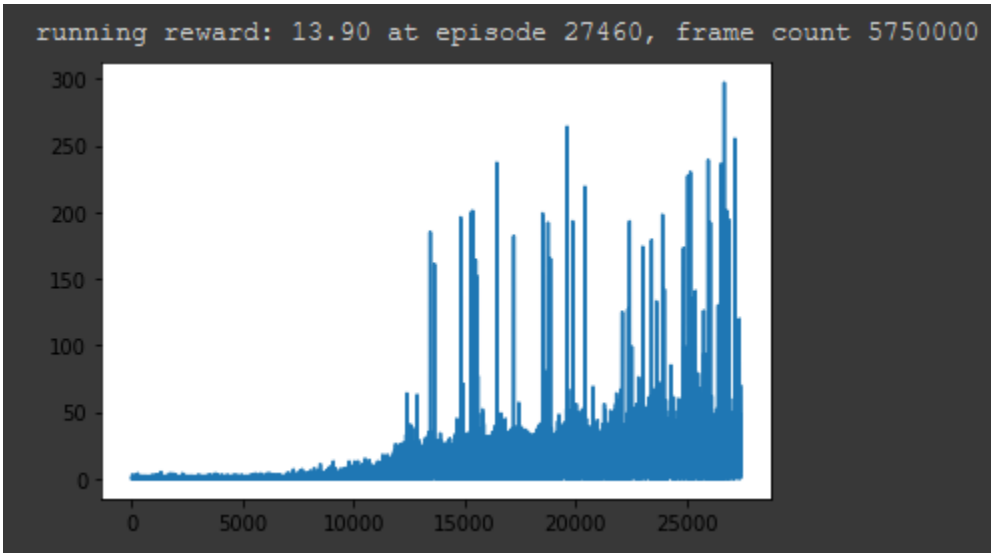


Figure 4: Dueling DQN agent for Breakout. Y-axis is the score earned on a given life, X-axis is the episode.

achieved a reward of nearly 300. Again, these results would likely stabilize over time. While the Dueling DQN did not perform better for Breakout, we would expect to see improved results if we applied this method to a game with a larger action space.

A video of our partially trained agent can be viewed at the following URL:

https://youtu.be/NCgkmgzbw7o

The working codebase can be viewed here:

https://github.umn.edu/voigt227/TheABC

6 Conclusion

In this paper, we presented our attempt to create a gameplay agent for the game Super Crate Box by using Deep Reinforcement Learning. We first set up an RL environment similar to the functionality of the OpenAI Gym library and then tried to implement both the DQN method and the genetic evolution method to train the agent. Due to the lack of training time, character model randomization and other difficulties, we were not able to successfully train an agent which can play the game. Instead we switched our focus to the OpenAI Gym game Breakout. By using two refinements of the basic DQN algorithm, the agent was able to achieve an average reward of 17.75 and maximum reward of ≈ 250 with the Double DQN method, and an average reward of 13.90 and maximum reward of ≈ 300 with the Dueling DQN method.

One major takeaway is that while DRL is a promising and exciting field, it also comes with its own unique set of challenges. The implementation of any DRL agent requires tuning many hyperparameters. It also requires defining an appropriate reward function – a process which is domain-specific and necessitates trial and error. Additionally, training takes a long time, with millions of frames required for Atari games, and the training process introduces non-determinism. A recent paper [16] highlighted some of the issues with reproducibility in DRL, and emphasized the need for standardized experimental techniques and evaluation metrics.

With more time and better methods for extracting the environment of Super Crate Box, it might still be possible to train a DRL agent to play the game. However, while we did not succeed in this domain, we were at least able to reproduce some of DeepMind's successes on Breakout. In the process, we explored some recent advances in DRL, learned more about the challenges associated with it, and ultimately gained a better understanding of how DRL works in practice.

References

[1] *Super Crate Box*, (2010-2011). http://supercratebox.com

[2] *Gym*, (2016). https://gym.openai.com/docs

[3] Kaelbling, L. P., Littman, M.L., and Moore, A.W. "Reinforcement Learning: A Survey." *Journal of Artificial Intelligence Research* 4 (1996): 237–85. https://doi.org/10.1613/jair.301

[4] Li, Y. "Reinforcement Learning Applications." *Medium*. Medium, (2019). https://medium.com/@yuxili/rl-applications-73ef685c07eb

[5] "Q-learning." *Wikipedia*, (2020). https://en.wikipedia.org/wiki/Q-learning

[6] Mnih, V., Kavukcuoglu, K., Silver, D. *et al*. "Playing Atari with Deep Reinforcement Learning." (2013). https://arxiv.org/pdf/1312.5602v1.pdf

[7] Mnih, V., Kavukcuoglu, K., Silver, D. *et al*. "Human-Level Control Through Deep Reinforcement Learning." *Nature* **518**, 529–533 (2015). https://doi.org/10.1038/nature14236

[8] van Hasselt, H., Guez, A., Silver, D. "Deep Reinforcement Learning with Double Q-learning." (2015). https://arxiv.org/abs/1509.06461

[9] Wang, Z., Schaul, T., Hessel, M., *et al.* "Dueling Network Architectures for Deep Reinforcement Learning" (2015). https://arxiv.org/abs/1511.06581

[10] Schaul, T., Quan, J., Antonoglou, I., Silver, D. "Prioritized Experience Replay" (2015). https://arxiv.org/abs/1511.05952

[11] Petroski Such, F., Madhavan, V. *et al.* "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning" (2017). https://arxiv.org/abs/1712.06567

[12] *PyWin32*, (2019). http://timgolden.me.uk/pywin32-docs/contents.html

[13] *Google Tesseract OCR*, (2006). https://opensource.google/projects/tesseract

[14] "Reinforcement Learning (DQN) Tutorial." *PyTorch*, (2020). https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[15] "Deep Q-Learning for Atari Breakout." *Keras*, 2020. https://keras.io/examples/rl/deep_q_network_breakout

[16] Henderson, P., Islam, R., Bachman, P. *et al.* "Deep Reinforcement Learning that Matters" (2019). https://arxiv.org/abs/1709.06560