# Approximating Optimal Values using Deep Q-Learning

Alex Pan
Githash: 2e5a7fe07632027fbbe436bdfb446063f266d346

## I. Introduction

### A. Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that aims to teach an *agent* to take *actions* that maximize its accumulated *reward* within an *environment*. The agent receives rewards by taking actions to transition to different *states* within the environment. The agent's goal is to predict the total expected future reward, or *value*, it could receive by being in any particular state. With this information, an agent can form a policy to decide which action to take at any state. The agent strives to learn an *optimal policy*, that is, the action(s) to take at any state that will maximize its accumulated reward.

States $s$ and associated rewards $r$ within the environment are usually modeled as a Markov decision process (MDP). This means that future states $s'$ *only* depend on the agent's current state $s$ and action $a$. Under a finite MDP, Bellman's optimally equation describes the value associated with each state-action pair if the optimal policy $\pi^*$ is followed at each state [1, pp64]:

$$Q_{\pi^*}(s,a) = \max_{a'} \sum_{s',r} p(s',r|s,a)[r + \gamma Q_{\pi^*}(s',a')]. \quad (1)$$

Here $p$ is the probability of transitioning to $s'$ and receiving $r$ after taking action $a$ at $s$. Future state-actions values are also discounted by $\gamma$. However, solving this recursive equation is often impractical as it presents as an $n$x$n$ system of nonlinear equations (where $n$ is the number of states in the MDP). Additionally, the agent may not have full knowledge of the MDP and know every state, action, and transition probability tuple.

### B. Q-Learning

Q-learning is a temporal difference (TD) method that practically approximates the optimal state-action (Q) value without directly solving (1). TD methods utilize the predicted values of future states to iteratively update the values of the current state [1, pp131]. The update method for Q-learning is as follows:

$$Q_(s_t,a_t) \mathrel{+}= \alpha[r_{t+1} + \gamma \max_{a'} Q_(s_{t+1},a') - Q_(s_t,a_t)] \quad (2)$$

The Q-value for a state $s_t$ at time-step t is updated by the value of the state it transitions to $s_{t+1}$, scaled by the learning rate $\alpha$. This update procedure has been shown to be stable and eventually converge to $Q_{\pi^*}(s_t,a_t)$ after a sufficient number of iterations. Q-learning is also an example of an "off-policy" algorithm, as it does not update based on the actual next action $a_{t+1}$. Instead, it uses the value associated with the maximizing action $a'$ as the target. This allows for Q-learning algorithms to train on and optimize states beyond what is currently being experienced by the agent.

| Parameter | Value |
|---|---|
| Replay Buffer Size | 200,000 |
| $\epsilon$-greedy | (1, 0.1) |
| Frame to stop linearly attenuating $\epsilon$ | 1000 |
| Learning Rate | 5x10^-5 |
| Copy weights to target NN every $m$ training steps | $m = 20$ |
| Select $\epsilon$-greedy action and train every $n$-th frame | $n = 4$ |

TABLE I: Hyper-Parameters used to train the agent described in the

### C. Deep Networks to Approximate Q-values

Tabular Q-learning is limited by its large memory requirements and lack of generality. In the case where the number of state-action pairs are small, Q-learning updates can be easily applied to stored state-action values. This is not computationally feasible if the state space is large or continuous. Updating values in a table also lacks generality since it cannot predict values of state-action pairs the agent has not experienced. These reasons make tabular-based Q-learning unsuited for environments with large state spaces.

As use-case solutions to these inefficiencies, various function approximation approaches have been developed to model $Q(s_t,a_t)$ as a continuous function. A more recent approach is to use deep neural networks as a way to approximate Q-values in a continuous state-space. The applicability of these methods was demonstrated on an Atari game by Mnih et al. They showed that their deep Q network (DQN) was able to develop a well-performing policy given only human-level inputs and outputs (frame-by-frame graphics and joy stick outputs) [2]. Notably, they demonstrated that introducing a "replay buffer" and an additional "target" network dramatically improved the stability and generality of DQN.

Here we demonstrate DQN's ability to find and learn a high-performing policy for the Lunar Lander gym environment developed by OpenAI. We test the efficiency and stability of DQN by analyzing the effect of different hyper-parameters on learning. Finally, we discuss how and why changes to the replay buffer and update frequency of the target network result in less stable training.

## II. Methods

### A. Lunar Lander Environment

The game-objective is to gently land the ship on a landing pad at spatial coordinates (0,0). The agent can take 4 discrete actions to land the ship: do nothing or fire thrusters left, right, or up. The agent can receive up to +140 reward points for moving towards the landing pad. It receives an additional $\pm$100 reward

for coming to rest or crash landing. An additional +10 reward for each of the ship's 2 legs touching the ground. To incentive landing, each discrete firing of the up-thruster incurs -0.3 reward whereas firing left/right returns -0.03. The agent is considered to have "solved" the environment if it averages above 200 points over 100-consecutive episodes after training.

The continuous state vector for the environment is represented as the 8-tuple

$$(x, y, \dot{x}, \dot{y}, \theta, \omega, leg_L, leg_R),$$

where $(x, y)$ and $(\dot{x}, \dot{y})$ are the ship's spatial coordinates and velocity components, respectively. $(\theta, \omega)$ correspond to the ship's angle and angular velocity while $(leg_L, leg_R)$ are Boolean values that indicate whether the left and right legs are touching the ground. Average Q-values plotted in all subsequent figures were calculated by averaging the Q-values of a set of 50 initial states randomly instantiated by the environment wrapper.

### B. Deep Q Network Algorithm

We model the DQN algorithm presented in [2] with changes to the network and hyper-parameter selection. No additional preprocessing on input data was performed as the Lunar Lander environment provides the relevant state vector to the agent after each step. Alg. 1 outlines the key steps for this implementation of DQN-learning. Table 1 shows the hyper-parameters used for training. Training ended after 1000 episodes or when the average reward for the 50 most-recent episodes became more than 200 points.

*1) Replay Buffer:* Minh et al introduces a "replay buffer" to improve the stability of DQN. Briefly, the buffer stores a limited number of states that the agent has previously experienced. In this implementation, past experiences were stored as 4-tuples consisting of the current state, action, reward and future state $(s_t, a_t, r_t, s_{t+1})$. The number of stored tuples was 2x10$^5$. Once the buffer size is reached, the oldest entry is deleted and replaced with the most recently experienced state-action tuple.

*2) $\epsilon$-Greedy Actions:* The "greedy" action taken by the agent at a particular state $s$ is the one that maximizes $Q(s, a)$. The probability of choosing the greedy action is $(1 - \epsilon)$. The value of $\epsilon$ was initialized to 1 at the beginning of training. After every frame, $\epsilon$ was linearly attenuated until it reached 0.1 at frame 1,000. After this frame, $\epsilon$ was held at 0.1 until the end of training.

*3) Neural Network Design:* A fully-connected neural network (NN) with 2 hidden layers was used to ensure sufficient complexity in predicting Q-values. The input and output layers have 8 (size of state-vector) and 4 (number of actions) nodes, respectively. The first and second hidden layers consist of 64 and 128 nodes, respectively, and are all equipped with the ReLU activation function. Weights were randomly initialized before training. An additional target NN was used to stabilize training. This target NN has the same shape as the original network and was initialized with the same weights.

*4) Neural Network Training:* In each training instance, the original network was trained for 1 epoch with a batch of 32 4-tuples randomly sampled uniformly from the replay-buffer without replacement $(s, a, r, s') \sim U(D)$. The loss function used to adjust weights in the original NN was defined to reflect the Q-update method in (2):

$$\begin{aligned} L_i(\theta_i) &= E_{(s,a,r,s')\sim U(D)}[(y - Q_(s, a; \theta_i))^2], \\ &\text{where } y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-). \end{aligned} \tag{3}$$

Here, the target scalar $y$ is the reward plus the maximum Q-value for a given state predicted by the target network with parameters $\theta_i^-$. The loss is the mean-squared error between the target and predicted value. The target network parameters were only updated to have the same parameters as the original network $\theta_i$ after $m = 20$ training iterations. The learning rate $\alpha$ was set to 5x10$^{-5}$. Additionally, the discount rate $\gamma$ was set to 0.99. Finally, gradient descent was performed on the original network using RMSprop (momentum=0.95).

*5) Training and Greedy-Action Latency:* Latency was introduced to restrict the agent so that the training rate and number actions it could take were more comparable to a human player. Specifically, neural network training and $\epsilon$-greedy actions were only taken every 4th frame/step. The most recent $\epsilon$-greedy was taken otherwise. Experience tuples were still added to the replay buffer every frame. As an example, after 100 steps, the number of training iterations and $\epsilon$-greedy actions taken would be 25 each.

**Algorithm 1** Deep Q Learning with Buffer

Initialize replay buffer D to hold N steps
Initialize NN to approximate $Q$ with random weights $\theta$
Initialize target NN $\hat{Q}$ with weights $\theta^- = \theta$
Initialize $\epsilon = 1$
**for** each episode **do**
    Initialize environment; $s_t = s_1$
    **for** step $t$ in episode **do**
        With probability $\epsilon$ : Randomly select $a_t$
        Else: Select $a_t = \arg\max_a Q(s_t, a, \theta)$
        Perform $a_t$ in environment: receive reward $r_t$ and state $s_t$
        Store $(s_t, a_t, r_t, s_{t+1})$ in $D$
        Sample batch: $(s_d, a_d, r_d, s_{d+1}) \sim U(D)$
        Set target $y_d$ for each tuple in batch

$$y_d = \begin{cases} r_d \text{ if } s_{d+1} \text{ is terminal state} \\ r_d + \gamma \max_{a'} \hat{Q}(s_{d+1}, a'; \theta^-) \quad \text{otherwise} \end{cases}$$

        Gradient descent: $(y - Q(s_d, a_d; \theta))^2$ with respect to the NN with parameters $\theta$
        After every C training steps: copy $\theta^- = \theta$

## III. Results

### A. DQN Learning is Highly Stochastic

To show that DQN was able to learn a policy sufficient to solve the Lunar Lander game, DQN was first run using the parameters described in the methods and Table 1. Fig. 1a shows

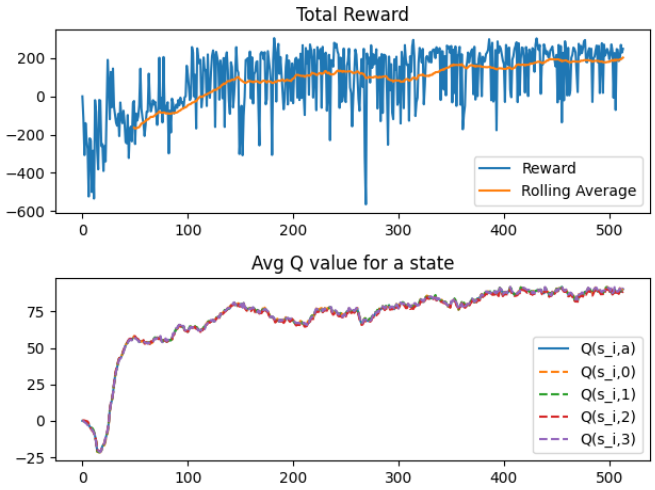


Fig. 1: Performance of agent while training (Table 1): a) The total accumulated reward after each episode/run of the agent. The orange line shows the rolling average of the 50 most recent episodes. Training ended at episode 514 when the rolling average $>= 200$. b) The average predicted Q-value for a set of 50 initial states after each episode. Dashed lines show the Q-value of each of the 4 actions available to the agent. For clarity, the average of the dashed lines is equal to the solid blue line.

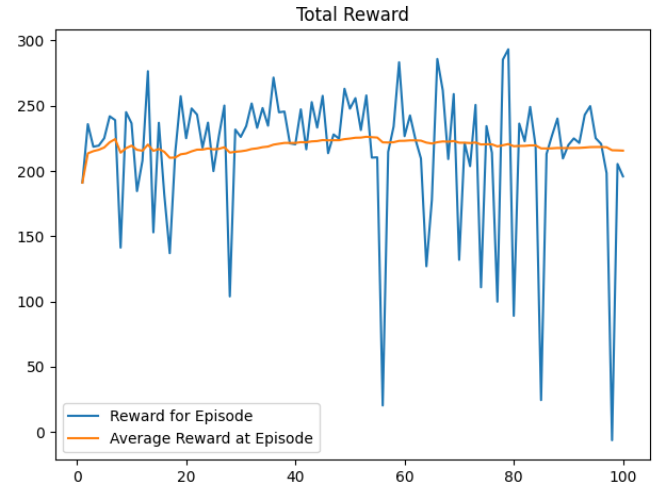


Fig. 2: The reward accumulated per episode of the agent after training (Table and Fig. 1). The orange line denotes the cumulative average after each episode.

the total reward and average Q-value across 514 episodes or 237,626 steps (see methods for termination criteria). The total reward initially accumulated by the agent each episode is highly stochastic. This is due to two reasons: 1) The agent performs near-random actions at this point 2) The environment has a high dynamic range of rewards and has specific criteria for successful completion.

The reward per episode, however, it still rather stochastic at later episodes given that $\epsilon \rightarrow 0.1$. Observing the lunar lander at these later episodes reveal that the agent continues to fire right and left engines despite successfully landing in the goal. A reluctance to terminate may indicate that the agent may not have an accurate predictions of terminal state values that are not frequently visited.

Fig. 1b shows the correlation between the Q-value prediction of a set of initial states and the accumulated reward over time. This is unsurprising as the total expected reward for initial states should be high if there is greater accumulated reward. Interestingly, the Q-values for each individual action all closely follow the cumulative average for these initial states. This suggests that the action taken at the beginning of the game does not affect the total expected reward. This makes intuitive sense as there are many chances for corrective action before termination. Finally, Fig. 2 shows that an agent trained with these parameters was able to solve Lunar Lander with an average score of 218 points at the end of 100 different episodes.


Fig. 3: Performance of agents with varying replay buffer sizes while training a) The rolling average (window size = 50) of total accumulated reward after each episode/run. Training was terminated when the rolling average reward $>= 200$ or when episodes=1000. This occurred at episode 1000, 590, and 514 for agents trained with buffer sizes of $2x10^3$, $2x10^4$, and $2x10^5$, respectively. b) The average predicted Q-value for a set of 50 initial states after each episode.

### *B. Smaller Replay Buffers Make Training Less Stable*

To determine the effect of the replay buffer size on learning, we tested two additional training runs using buffer sizes of $2x10^3$ and $2x10^4$ frames. For perspective, $10^3$ frames corresponds to on order $10^1$ episodes. All other parameters were fixed according to Table 1. As shown by Fig. 3.a, reducing the buffer size leads to less consistent rewards and longer training times. In this case, size=$2x10^3$ was not able to meet the training criteria before 1,000 episodes.

The less consistent performance of smaller buffer sizes is supported by the predicted Q-values. Fig. 3.b shows that Q-values for a small buffer size of $2x10^3$ vary sinusoidally across episodes. This suggests that the predictive network is optimizing for recently visited states. As the small buffer overwrites old experiences, the network returns a less accurate prediction for less frequent or past states. This is why the ”frequency” of this Q-value oscillation seems negatively correlated with the buffer

size. A larger buffer allows for continued sampling of these past states during batch training. This allows the network to not bias its optimization for recently visited states and results in more stable predictions.

Interestingly, the 2 agents trained with smaller replay buffer sizes overestimate the average Q-value for initial states relative to the agent with buffer size 2x10$^5$. Despite this, agents with size 2x10$^4$ and 2x10$^5$ perform very similarly during training.

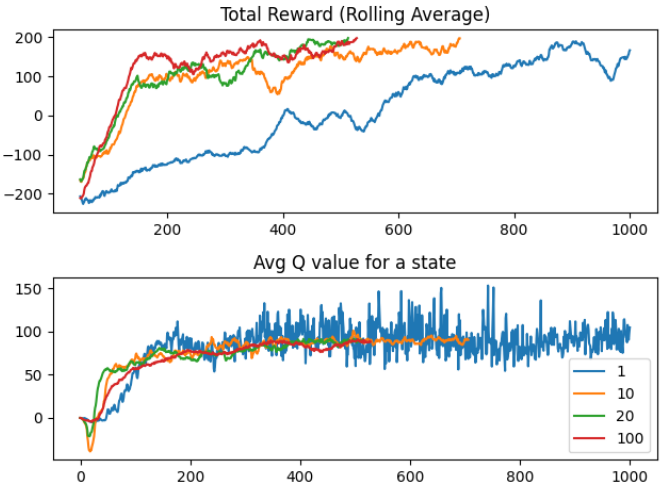


Fig. 4: Training performance of agents with varying target network update frequencies a) The rolling average (window size = 50) of total accumulated reward after each episode/run. Training terminated at episode 1000, 707, 514, and 529 for frequencies of 1, 10, 20, and 100, respectively. b) The average predicted Q-value for a set of 50 initial states after each episode.

### C. Frequently Updating Target Network Slows Training

To illustrate the increase in training stability due to the implementation of a target network, 3 additional training runs with different target update frequencies were performed. Of note, a target update frequency of 1 was one such chosen value. This corresponds to updating the target network parameters after every training iteration. This is also a limiting case; an update frequency of 1 is the same as if the target network did not exist at all. Again, all other training parameters were fixed according to Table 1.

Fig. 4 shows that more frequent target network parameter updates makes training unstable. An update frequency of 1 does not meet the training criteria within 1000 episodes. Additionally, the predicted Q-values have vary significantly episode-to-episode. However, the accumulated reward over time steadily increases. If given more time, this agent would most likely meet the training criteria. This is evidenced by Fig. 4b, which shows that all agents begin to converge to the same Q-value prediction for the set of tested initial states.

Interestingly, training update frequencies of 20 and 100 both meet the training criteria after a similar number of episodes (514 and 529 respectively). Additionally, Fig. 4b shows that an update frequency of 20 results in initially better Q-value predictions than an update frequency of 100. Despite this, Fig. 4a shows that the agent with an update frequency of 100 initially performs better. This discrepancy may be due to a serendipitous sequence of $\epsilon$-greedy actions. More training runs with different random seeds may be needed to confirm these results.

### D. Similar Learning Outcome with Faster $\epsilon$-Attenuation

Three parameters control the probability of selecting the $\epsilon$-greedy action: the initial and final fixed values of $\epsilon$, and the speed at which $\epsilon$ is linearly attenuated each frame/step until it reaches the final fixed value ($\epsilon$ = 0.1). Here, we only examine how the attenuation speed affects learning. Three additional agents that linearly attenuated $\epsilon$ for 1x10$^2$, 1x10$^4$, and 5x10$^4$ steps were trained and compared. Again, 10$^3$ steps roughly corresponds to 10$^1$ episodes.

The impact of faster attenuation on learning drastically diminishes according to Fig. 5. Agents that attenuated $\epsilon$ for 1x10$^2$, 1x10$^3$, 1x10$^4$, and 5x10$^4$ steps met training criteria and terminated after training for 552, 514, 560, and 859 episodes, respectively. With the exception of steps= 5x10$^4$, all the accumulated rewards and predicted Q-values for each agent are similar. The agent with the fastest attenuation speed (steps = 1x10$^2$), however, had noticeably higher accumulated rewards and predicted Q-values initially. This may be because less exploration results in more greedy state-action pairs to be stored in the replay buffer. This in turn means that predicted Q-values corresponding to these greedy pairs are updated more frequently. More accurate predictions and higher chances of taking the greedy action then leads to more reward (initially).

Interestingly, training outcomes vary significantly for attenuation slower speeds (steps >= 1x10$^4$). Fig. 5 shows that the number of episodes required for training increased ~50% for the agent trained with steps = 5x10$^4$ compared to other agents. This is because the large amount unique state-action pairs observed due to increased exploration requires more training iterations to sufficiently update each prediction. Fig. 5b shows that the rapid increase of the average Q-value occurs at later episodes for agents with slower attenuation speeds. These results suggest that the lunar lander environment may not require much exploration to solve.

## IV. Discussion

### A. Limitation of Results

In summary, DQN is able to train an agent to solve the lunar lander environment. We show that the replay buffer size is positively correlated with shorter and more stable training. Additionally, the target network update frequency that minimizes training time is between 20-100 iterations. Finally, the speed at which exploration is linearly attenuated was also correlated with shorter training.

However, the reproducibility of these results are limited. A major weakness of this study is that each agent with a unique combination of hyper parameters was only trained once. DQN consists of several components that are probabilistic: $\epsilon$-greedy actions, instantiated initial states, and neural network initialization. Despite setting a consistent seed for the gym environment and Tensorflow, training runs using the same hyper parameters differed (sometimes significantly). One solution to this is to simply record rewards and Q-value predictions for several

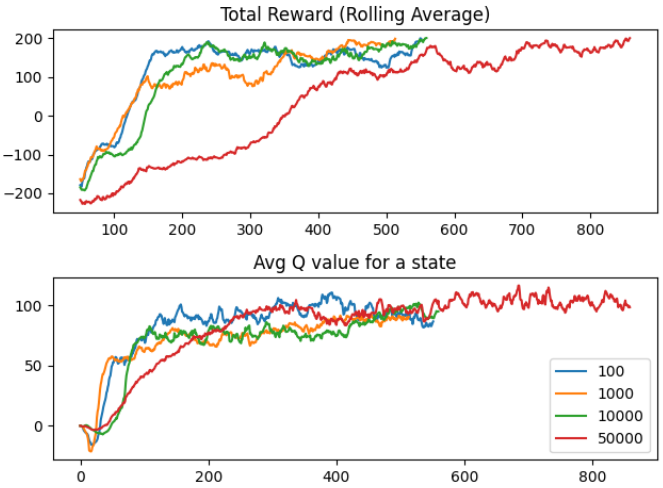


Fig. 5: Performance of agents that vary in the number of step $\epsilon$ was linearly attenuated until $\epsilon = 0.1$ a) The rolling average (window size = 50) of total accumulated reward after each episode/run. Training terminated at episode 552, 514, 560, and 859 for final $\epsilon$-attenuation at 1x10$^2$,1x10$^3$, 1x10$^4$, and 5x10$^4$ steps, respectively. b) The average predicted Q-value for a set of 50 initial states after each episode.

separate training runs. These metrics can then be aggregated and averaged across runs.

An additional limitation was the choice and range of hyper-parameter values to test. For example, the final fixed value of $\epsilon$ most likely has more impact on training results than the speed of $\epsilon$ attenuation (see Fig 5). Furthermore, only one limiting case was tested for the target network update frequency. It would also be informative to know at which training frequency $> 100$ where training becomes intractable.

Finally, these results would be strengthened if the actual performance of each agent (Fig. 2) was reported along with the training performance metrics. The training termination criteria of obtaining an average reward of at least 200 points (for the 50 most recent episodes) within 1000 episodes was an arbitrary choice to prevent over-training and limit run times. This termination does not necessarily guarantee that the agent will meet the goal of averaging over 200 points across 100 episodes after training.

### B. DQN Stability

The novelty of Mnih el al's DQN algorithm is the introduction of the replay buffer and target neural network. We show in the results how each affect the stability of learning. Although we explain why the replay buffer improves training in the results, we did not justify why the removal of the target network (update frequency of 1) caused such stochastic predictions.

A neural network requires a target value to compare its own output to. At a high level, the network uses the difference between the target and its prediction to decide how much each network parameter should be updated to minimize this difference. In a supervised learning context, the target is a label or value provided by a third-party expert. In RL, the true Q-value $Q_{\pi^*}(s_t, a_t)$ is not known a prioi. Q-learning addresses this by letting the target be the current value function plus the reward. The target value for each update is then a different value for each training iteration. Therefore, unlike traditional supervised learning, training is done on a *moving* target. To address the inherent instability of this process, Minh et al introduce an additional target network that holds the target value "still" temporarily. Our results show the stability advantages this choice provides.

### C. Optimal Value Approximation

There is evidence to show that the values learned by the presented agents do not well approximate optimal values. $Q_{\pi^*}(s_t, a_t)$ is defined to be the *maximum* expected discounted reward (1). Therefore, the Q-value for the initial state(s) represent all the final discounted return after an episode. In the case of the lunar lander environment, the transition probability associated with each state-action pair is equal to 1. Assuming that an agent following the optimal policy can complete an episode in $\sim 50$ frames and obtain a reward of 200, the optimal value for an initial state can be roughly approximated to be $(\gamma = 0.99)^{50}(200) \sim 120$. In all figures, average Q-values for the same set of initial states never seem to reach $\sim 120$. Despite this, all graphs still seem to have an upwards trajectory. Perhaps after more training time, they will converge to optimal values.

### D. Future Directions

As previously described, additional experiments are necessary to strengthen the results shown. However, DQN has many other hyper parameters that can impact learning. For example. the learning rate is known to dramatically impact the convergence of DQN. Additionally, the shape of the fully-connected neural network could be downsized to improve training speeds.

To fully recreate Minh et al's methodology, a convoluted neural network that takes only the pixels rendered from the environment as input could be implemented. Significant preprocessing of these images would be necessary, but it would go to show that DQN can truly learn from human-level input.

## V. References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2nd Ed. MIT press, 2020. Url: http://incompleteideas.net/book/the-book-2nd.html.

[2] 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