commit 030746be253aed7aeb0f19d0d8d25bc5803713c3

# Solving LunarLander-v2 with Deep Q-Network

Zhiyi Chen
*Computer Science Department*
*Georgia Institute of Technology*
zchen853@gatech,edu

***Abstract*— The Deep Q-Network (DQN) was proposed by Mnih et al. [1] in 2015, where it is presented as an algorithm that can learn to play Atari 2600 video games. It was shown that not only was DQN able to surpass the performance of all previous algorithms, it outperformed some best human players in most games. In this paper, DQN is used to solve another MDP problem, namely the LunarLander-v2, which is a rocket trajectory control problem. It was shown that with minor changes to the originally proposed configurations, DQN can reliably solve the Lunar lander problem.**

***Keywords—DQN, Markov Decision Process, Q-Learning, control policy***

## I. INTRODUCTION

Deep Q-Network (DQN) has been a fountain to recent deep Reinforcement Learning (RL) research ideas. This work aims to apply DQN on solving a control optimization MDP problem, LunarLander-v2. In addition, several hyperparameters in DQN are explored, in attempt to understand how they can affect the convergence of agent in solve the lunar lander problem. Section II discuss the background of the lunar lander problem. Section III discusses the origin of DQN, the Q-Learning, as well as DQN itself. Section IV lays out how DQN is implemented to solve the lunar lander problem. And Lastly Section V reviews and discusses the experiment results.

## II. BACKGROUND

The objective is to train an agent to learn a control policy that is able to land a rocket (lander) safely at a target location (landing pad). The agent is put to repeatedly interact with the environment E, in this case the Lunarlander-v2 simulation environment from OpenAI Gym, by taking a sequence of actions $a \in A$, and observing the resulted states $s_{t+1} \in S$ and corresponding rewards $R$, until a terminal state is reached.

For each episode, the lander starts out at some coordinate $(x, y)$ that is above the landing pad, which is always at coordinates $(0,0)$. The agent controls the motion of the lander at each step by selecting from one of the four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. At each time step, the environment provides the agent with an observation of the new state in low-dimensional representation, as well as a positive or negative reward gained entering the state. Specifically, the state is represented with a 8-tuple: $(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, leg_L, leg_R)$, which describes lander's position, velocity, orientation, angular velocity, and whether it has landed on left leg or right leg or both.

The goal is to land the lander at landing pad $(0,0)$, but landing outside of the pad is possible. The agent can obtain a reward in the range 100 – 140, for moving the lander towards the landing pad, depending the exact landing location. If lander moves away from the landing pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 points respectively. Each leg-ground contact is worth +10 points. Firing the main engine incurs a penalty of -0.3 points and firing the orientation engines incur a penalty of -0.03 points, for each occurrence. Fuel is infinite, but the agent is limited to take a maximum of 1000 steps per episode. If agent achieves 200 points on average over 100 consecutive episodes, the problem is consider solved.

This problem can be formulated as a Markov decision process (MDP) with continuous state space. The objective is thus finding a mapping function, i.e. policy pi, between states and actions, that maximizes the expected future discounted reward when the agent chooses actions based on the policy.

## III. RELATED WORK

### A. Q-Learning

The lunar lander problem described in previous section is an MDP, which is a typical formulation for reinforcement learning problems. Among many solutions to such problem, Q-learning, an off-policy TD (temporal difference) control algorithm proposed by Watkins [2], has particular importance. Q-learning is based on the concept of Q-function, $Q^{\pi}: S \times A \rightarrow \mathbb{R}$, related to the MDPs. Q-function defines the expected future discounted reward for taking action a in state s and then following policy $\pi$ thereafter. By applying Bellman operation, with three additional conditions, Q-function can converge to $Q^*$ for the optimum policy $\pi^*$. The optimum policy $\pi^*$ for an MDP problem can thus be obtained if $Q^*$ can be approximated.

In Q-Learning, an agent starts with an arbitrary Q-function. At each iteration, the agent takes arbitrary action in the environment and observes the reward and next state resulted from the action. The Q-function is then updated by applying Bellman operation according to (1).

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha_t[r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)] \qquad (1)$$

Where $s_t$, $\alpha_t$, $r_t$ are the state, action, and reward at time step $t$ and $\alpha_t$ is a step size smoothing parameter. The convergence of (1) to $Q^*$ is guarantee under three conditions: i) a Q-value is associated with each unique state-action pair, ii) the agent visits

each state and action infinitely often, and iii) $\alpha_t \to 0$ as $t \to \infty$ [4]. The satisfaction of iii) is rather straight forward, since alpha is less then zero, as t goes to infinity, alpha t approaches infinity within the Bellman operation. Condition ii) can be met by carrying out the update for a large enough number of iterations and allowing agent with certain probability to randomly pick actions. In the case of a finite state-action space, i) can be easily met by constructing a Q-table. In the cases of a continuous or large state space, Q-Learning uses function approximation to map between state-action to values. A caveat with using function approximation in Q-Learning is that the estimation error can result in divergence of the Q-function [3].

### *B. Deep Q-Learning*

A great advancement to traditional Q-Learning is the introduction of deep learning into the RL paradigm by Mnih et al. [1]. The new algorithm proposed is known as DQN, which has three major innovations towards the tradition Q-Learning. Firstly, it uses deep convolution network for function approximation. Secondly, it introduces the concept of experience replay, which stores a history of state, action, reward, and next state that are experienced by the agent. At each time step, a small random sample, called mini-batch is drawn from the replay to train the Q-network. Experience replay effectively reduces the autocorrelation of samples collected during online learning, and prevents falling into local minima or divergence. Thirdly, it introduced a target network to estimate the Q-values of the next state, that are used as targets when training the Q-network. The target network has the exact same architecture as the Q-network, but is only updated every lambda-steps with the weights of Q-network and held constant otherwise. Target network ensures that the target is fixed periodically, this prevents the Q-network to always chase an unstable target, which results in highly oscillated training errors. Algorithm 1 below is the DQN pseudocode copied from Mnih et al. [1]

**Algorithm 1** Deep Q-learning with experience replay

Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** episode 1, $M$ **do** Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in the emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **end for**
**end for**

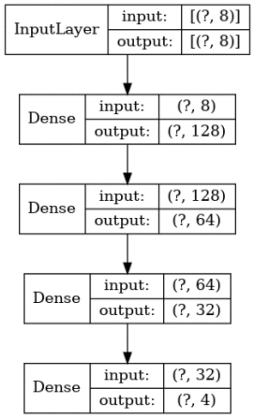
## IV. Implementation Details

This section lays out how DQN is implemented for the lunar lander problem. The setup of the DQN in Mnih et al. [1] has served as the reference for this work. As a result of the significantly different state space representations of the two environments, some modifications and simplifications are made to the original DQN setup.

The first notable difference is the abandoning of the preprocessing function phi used in the original setup. State space is presented as pixel images in the Atari environment, while in the LunarLander-v2, it is presented as 8-tuples. Function phi was used for image preprocessing, as well as for stacking every 4 consecutive game frames to form the final input to Q-network in the original DQN. This is because, in some Atari games, current state might not be fully observable from the current game frame, it has to be inferred from some previous frames, hence stacking 4 frames allows the agent to choose action with consideration of prior sequence of game frames. In the lunar lander problem, state representation gives the full information about the current state, thus function phi is not required.

In additional, since state space of LunarLander-v2 environment is presented in tuples, a much simpler network is used for Q-function approximation. Specifically, a fully connected 4 layers (128, 64, 32, 4) deep neural network with 8 input neurones is used instead of convolution network originally proposed. Figure x below shows the set up of the deep NN used in this work.

*Figure 1:Deep neural network design for Lunar Lander DQN*




Other base setup of the DQN hyperparameters for this work is summarized in table below.

*Table 1: Hyperparameter base setup for Lunar Lander DQN*

| Hyperparameter | Value | Description |
|---|---|---|
| minibatch size | 32 | Number of training cases over which each gradient descent update is computed. |
| max memory length | 1000 | Training cases are sampled from this number of most recent frames. |
| target network update frequency | 10 | The frequency with which the target network is updated with Q-network weights |
| discount factor (gamma) | 0.99 | Discount factor gamma used in the Q-learning update |
| learning rate | 0.0001 | The learning rate used by optimizer |
| Epsilon max | 1.0 | Initial value of epsilon-greedy exploration |
| Epsilon min | 0.1 | Final value of epsilon-greedy exploration |
| Epsilon greedy step | 1000 | The number of steps over which the max value of epsilon is linearly annealed to its min value |
| Epsilon random steps | 10000 | A uniform random policy is run for this number of steps before epsilon-greedy policy engage |
| seed | 42 | For randomness control |

Four hyperparameters are explored for different values in addition to the base setup values, they are: *replay memory size*,

*target network update frequency*, *exploration annealing steps* and *replay start size*. When a hyperparameter is tested for different values, all other hyperparameters are kept at their base setup value. The performance of the agent trained under base setup is used as the benchmark when evaluating performance of other hyperparameter settings.

A key constraint in this work is the limited computing power. Due to this limitation, convergence of the DQN can take more than 2 hours under certain hyperparameter settings. In order to allow for a quicker turn around and to explore more hyperparameter settings, each training process is limited to 600,000 steps, which, under different hyperparameter settings, are around 1100 to 1500 episodes. As a result, convergence of the DQN under some hyperparameter settings might not be observed within the limited training steps. However, the performance of the DQN agents at 600,000 steps do provides insights about the effects of hyperparameters.

## V. Results and Discussion

### A. Base setup

*Figure 2:In-training rewards of DQN with base hyperparameter setup*

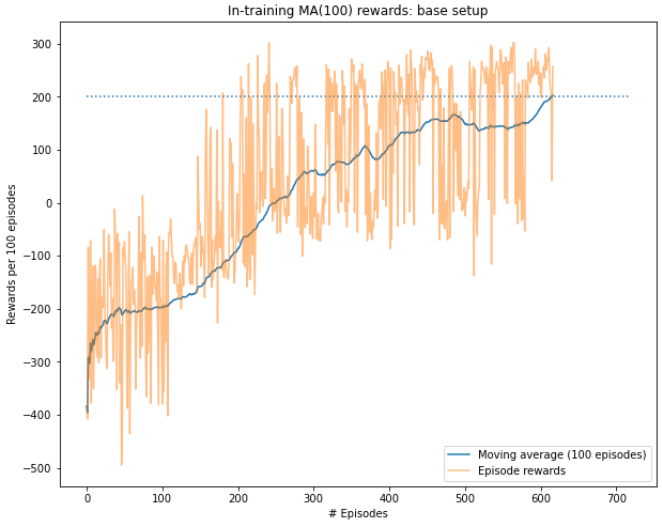



Figure 1 above shows the rewards obtained by agent during the training process using base setup hyperparameter values shown in table 1. Rewards obtained at each training episode for a total 600,000 training steps are shown in orange. The 100-episodes moving average reward (MA100) is shown in blue. The dotted line is the convergence criteria, which requires the agent to score 200 points on average for 100 consecutive episodes. The intersection between MA100 line and the dotted line signifies the problem is solved. Conversely, if no intersection between MA100 line and the dotted line, it means the problem is not solved within 600,000 training steps.

From Figure 1, it is observed that the episode rewards have fluctuated throughout the training process, however, moving average rewards increased steadily. DQN is consider converged at around 600$^{th}$ episode, where MA100 intersected with the dotted line.

*Figure 3: Rewards scored by trained DQN (base setup) on 100 test episodes*

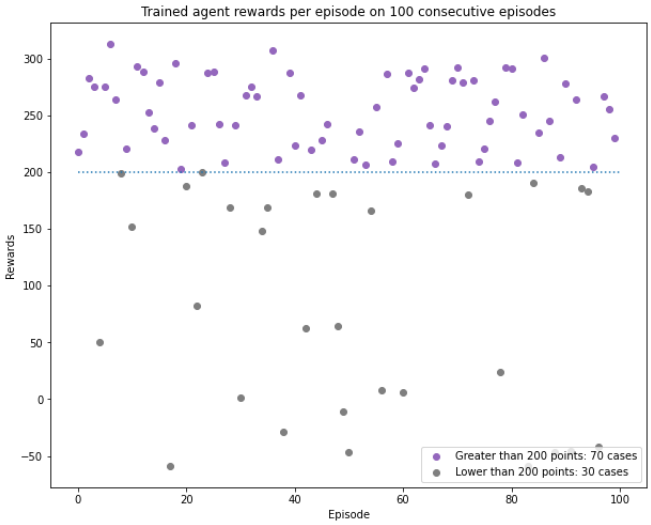
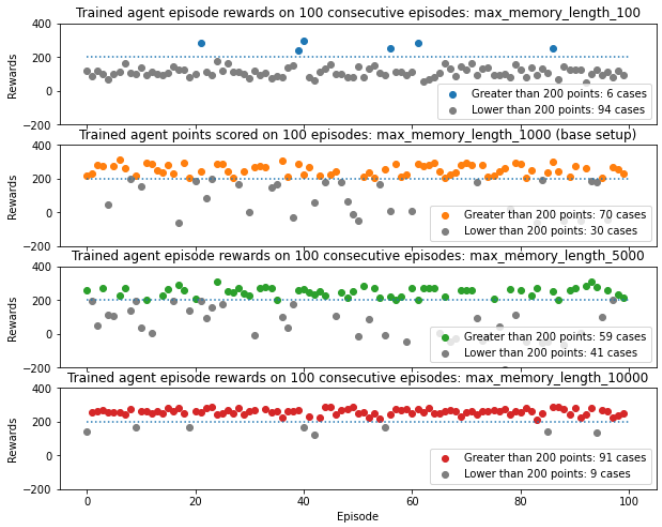



Figure 3 above shows rewards scored by the trained DQN agent for 100 consecutive test episodes. The trained agent was able to complete 70 episodes with reward higher than 200. It is worth noting that the agent scored between 150 to 200 points for about half of the remaining 30 episodes, and scored negative points for only 5 episodes. Which seems to imply that, for most of the time it failed, the agent was actually trying to move the rocket towards the landing pad, however, the rocket was likely to have either landed slightly off the landing pad, or crashed to the ground near the landing pad. Overall, the test result shows the agent was able to learn a policy that can land the rocket onto landing pad with a moderate success rate (70% of the time). One reason for the occasional failures at final execution is that the agent might not have seen the "perfect landing" enough times, since the training was stopped immediately after the agent scores above 200 points on average for 100 episodes. As a result, the Q-function might not be optimized for the final states, where the lander is about to make contact with ground. Hence, the performance on test episodes can potentially be improved by training the agent for more episodes, or by enforcing a stricter convergence criterion.

## B. *Experiment: Max memory length*

*Figure 4:In-training rewards scored by DQN trained with different max memory length*


*Figure 5: Trained DQNs (different max memory length) test episodes performances*




In this experiment, 3 additional values are tested for the hyperparameter *max memory length*, while all other parameters are kept at base setup values.

Figure 4 above shows the 100-episodes moving average (MV100) rewards scored by DQN agents during training when trained with different *max memory length*. It is observed that DQN did not converge within 600,000 training steps when *max memory length* was set to 100 training steps. A possible reason for this is that the memories stored in the replay are too recent (in this environment, each episode consists around 500-700 training steps, which is larger than max memory length of 100 steps), meaning that the agent was not able to see past experiences from episodes other than the current episode, thus it is not "exploiting" the past experience well enough.

Another interesting observation to make is when *max memory length* was set to 5000. From the corresponding MV100 plot, it seems the agent almost converged at around 350$^{th}$ episode, but suddenly diverged from the optimum policy thereafter, and only converged much later at around 800$^{th}$ episode. An informed guess to the reason could be that at round the 350$^{th}$ episode, only very few episodes with score greater than 200 have been experienced. Thus, even though MV100 is approaching 200, the agent was never adequately trained with enough data points of success landing. As a result, it took some wrong actions and eventually diverged. Of course, another huge possibility is simply the randomness of the run. Potentially the agent may converge around the 350$^{th}$ episode if it is trained again.

Figure 5 shows the performance of the four DQN agents on 100 consecutive episodes. DQN with *max memory length* set to 10,000 is the best performer. However, an interesting observation is the performance of the DQN with *max memory length* equal to 100. Although it did not have many success landings, it scored above 0 for all episodes. This shows that although its Q-function has not converged yet, but the performance is very consistent.

In summary, this experiment suggests setting max memory length too small will hinder the agent's ability to exploit past experience, hence leading to slower convergence.

## C. *Experiment: Epsilon random steps*

*Figure 6: In-training rewards scored by DQN trained with different epsilon random steps*

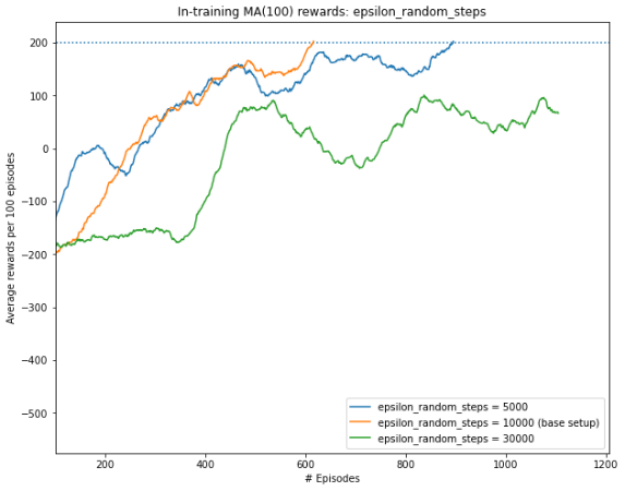
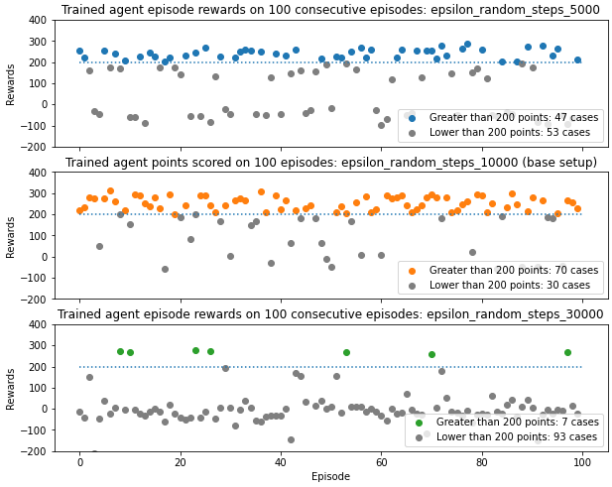



*Figure 7:Trained DQNs (different epsilon random steps) test episodes performances*



In this experiment different values for *epsilon random steps* are explored, this hyperparameter controls how many steps random action (exploration of the state-action space) must be taken prior to applying epsilon-greedy. Figure 6 illustrate the 100-episodes moving average rewards scored during training by three DQN agents. It is observed that when *epsilon random steps* is set to 30,000, the DQN agent failed to converge within 600,000 training steps, while the other two DQN agents successfully converged. However, performance results on testing episodes illustrated in figure 7 shows that the DQN agent trained with *epsilon random steps* set to 5000 failed on more than half of the 100 testing episodes even though it converged during training. This observation suggests that higher *epsilon random steps* value might result in slower convergence of the DQN. On the other hand, a low *epsilon random steps* value might result in agent sees only limited state-action space, and overfits to training data, hence not performing well on testing data.

### D. *Experiment: Epsilon greedy steps*

In this last experiment, different values for *epsilon greedy steps* are explored, which controls the linear annealing rate of the epsilon value. A higher value for *epsilon greedy steps* results in a slower annealing rate for epsilon, which means the likelihood of choosing random action in epsilon-greedy policy reduced at a slower pace through time. Effectively, random actions are more likely to be chosen even at later training episodes. Vice versa.

From figure 8 it is observed that the higher the value for *epsilon greedy steps*, the longer it takes for DQN to converge, although convergence of DQNs, when *epsilon greedy steps* is set to 10,000 and 100,000, was not actually observed in 600,000 training steps.

*Figure 8:In-training rewards scored by DQN trained with different epsilon greedy steps*

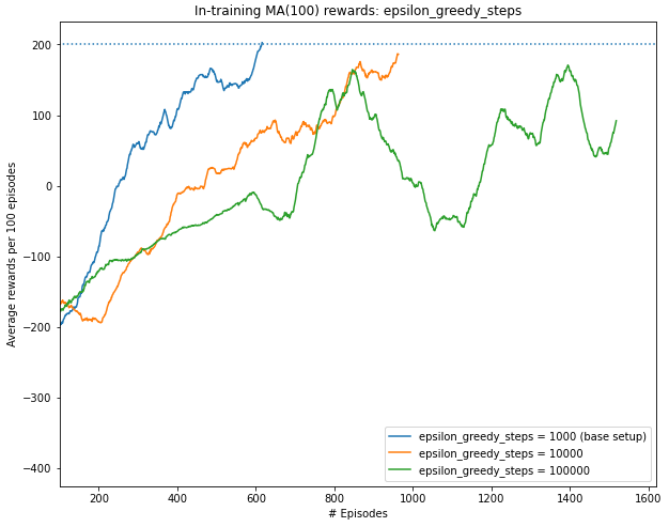
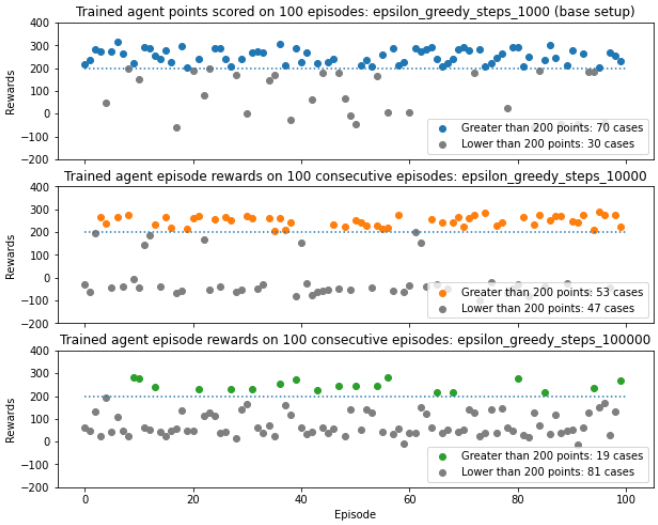



*Figure 9:Trained DQNs (different epsilon greedy steps) test episodes performances*




## VI. Conclusion

In this experiment, DQN was used to solve the lunar lander problem. Effects of three hyperparameters on DQN were explored. It was observed that if *max memory length* is set too small, agent's ability to exploit past experience might be hindered, hence resulting in slower convergence. *Epsilon random steps* needs careful tuning as setting it with a value too high might results in delayed convergence, while setting it with a value too low might result in overfitting to training data. Lastly, it is observed that the higher the value of *epsilon greedy steps,* the longer it might take for DQN to converge.

It is important to note that in order to achieve a quicker turn around time for getting experiment results, all experiments performed are limited to 600,000 training steps. As a result, convergence was not observed for some DQNs. Evaluating those non-converged DQNs on testing episodes does not reflect the true performance. In future when time permits, those non-converged DQNs will be re-trained without limiting training steps, or with a much higher limit, and then re-evaluated on 100 consecutive testing episodes. In addition, due to time limitation, hyperparameters were not explored thoroughly. In the future, hyperparameters should be explored with a larger range.

## References


[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529-533, 2015.

[2] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's Colleage, Cambridge, 1989.

[3] Leemon Baird et al. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the twelfth international conference on machine learning,*, pages 30-37, 1995.

[4] Roderick, M., MacGlashan, J., Tellex, S.: *Implementing the deep Q-network*, November 2017.