Reinforcement Learning with Deep Q-Networks

A thesis submitted in partial fulfillment of the requirements for the degree

Masters of Science, Computer Science.

Western Kentucky University

Bowling Green, Kentucky

By

Caleb Cassady

Reinforcement Learning with Deep Q-Networks

04/08/2022

Defense Date

Dr. Guangming Xing

Committee Chair

Dr. Huanjing Wang

Committee Member

Dr. Rong Yang

Committee Member

Ranjit Koodali

Associate Provost for Research & Graduate Education

Abstract

Reinforcement Learning with Deep Q-Networks

In the past decade, machine learning strategies centered on the use of Deep Neural Networks (DNNs) have caught the interest of researchers due to their success in complicated classification and prediction problems. More recently, these DNNs have been applied to reinforcement learning tasks with state-of-the-art results using Deep Q-Networks (DQNs) based on the Q-Learning algorithm. However, the DQN training process is different from standard DNNs and poses significant challenges for certain reinforcement learning environments. This paper examines some of these challenges, compares proposed solutions, and offers novel solutions based on previous research. Experiment implementation available at https://github.com/caleb98/dqlearning.

This thesis is dedicated to my parents, Pam and Scott Cassady, who constantly supported and believed in me throughout.

Contents

1 Introduction

As computational power has increased and become more readily available, machine learning research has shifted much focus toward the field of Deep Learning. Deep Learning refers to machine learning models which utilize multiple layers of processing in order to learn higher-level abstractions of data [7]. One of the most prominent models used in Deep Learning is the Deep Neural Network (DNN), which has proven successful in a wide range of supervised learning tasks such as image classification, natural language processing, speech recognition, and many more [7, 13]. DNNs have also been shown to perform well in Reinforcement Learning (RL) tasks such as playing Atari games [9, 8], first-person-shooter games [6], and Go [12]. DNN-based approaches to RL tasks often involve a modified Q-Learning algorithm utilizing the network to estimate action values in a given state. Such action-value approximators are called Deep Q-Networks (DQNs.)

The advantage offered by DQNs compared to previous RL approaches is in their ability to handle tasks with large or continuous state spaces. Where previous approaches were practical only for tasks with small state spaces or where carefully designed hand-crafted representations could be implemented, DQNs can leverage their ability to learn useful abstractions even from high-dimensional inputs [8]. This results in a model that can be applied to a wider range of tasks with better performance and less manual human design.

2 Background

2.1 Reinforcement Learning

Reinforcement learning (RL) encompasses a unique collection of tasks that are not only applicable to many real-world scenarios but also differ from standard machine learning domains in very fundamental ways. Generally, machine learning tasks are classified as either *supervised* (learning from labeled training data) or *unsupervised* (discovering labels from unlabeled data). RL, however, is unique in that it does not use labeled training data or seek to discover any labels. Rather, RL tasks focus solely on maximizing a reward obtained from one's environment [14]. Examples of RL tasks include playing board or video games, controlling robotic limbs, finding optimal paths through an environment, and many more. In fact, many learning tasks we face as humans every day can be modeled as RL tasks.

The general model for an RL task involves an agent which observes an environment and takes actions based on that observation. For simplicity, we assume the observation-action cycle to occur as discrete timesteps. To facilitate training an agent for an RL task, the agent is provided with a reward value after taking an action at each timestep. Positive rewards can be used to indicate that a desirable action was taken, while negative rewards can indicate less favorable actions. The goal of a machine learning algorithm in the context of RL is to learn a policy that maximizes the total reward over an entire run of the simulation (often called an episode).

Formally, we define an RL environment as a four-tuple (S, A, δ, R), where:

- S is a set of environment states.
- A is a set of actions available to the agent.
- $\delta(s, a) = s'$ is a transition function that returns the new environment state s' after taking action a in state s.
- $R(s, a) = r$ is a function that returns a reward value r for taking action a in state s.

The agent interacts with the environment in discrete timesteps $t = 0, 1, \ldots, T$, with T indicating the terminating timestep of the episode. At each timestep the agent selects an action a_t and the environment is updated to the new state using the transition function $\delta(s_t, a_t) = s_{t+1}$. The agent is then provided with a reward $r_t = R(s_t, a_t)$.

Since the objective is to maximize the reward across all timesteps – that is, the *expected return* – we also need a way to calculate this value. Naively, we could define this reward using a sum of the sequence of rewards:

$$R_t = r_t + r_{t+1} + \cdots + r_T$$

However, for some tasks where the agent is continually taking actions with no end state this approach will fail as the sum of all rewards will tend to infinity when $T = \infty$. This would prevent the agent from distinguishing actions that produce greater rewards more quickly. For this reason, we introduce a discount factor γ $(0 \leq \gamma \leq 1)$ which is used to weigh the importance of

future rewards relative to immediate rewards. The total expected return is then calculated instead as:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{T} \gamma^k r_{k+t}$$

If the agent knew the precise reward sequence it would obtain, no additional computations would be required. Of course, the reward values at each timestep depend on both the state of the environment and the action that the agent selects. Thus, we also define an *action-value* function that helps us approximate the values of specific state-action pairs. This action-value function is often called the *Q-function*, and the values we obtain from it are referred to as *Q-values*.

$$Q(s_t, a_t) = \mathbb{E}\left[r_t + \gamma \max_{a'} Q(s_{t+1}, a')\right]$$

That is, $Q(s_t, a_t)$ represents the expected immediate reward for taking action a_t in state s_t plus the product of the discount factor and the maximum expected return in the new state. This function is defined based on the intuition that the optimal way to maximize overall reward is to take actions maximizing the expected return at each timestep. This assumption will be correct as long as the Q-function is accurate [9]. The approach that many RL algorithms take, then, is to learn an accurate Q-function.

2.2 Q-Learning

Q-Learning is a reinforcement learning algorithm that learns Q-values through an iterative update process. The algorithm begins by creating a look-up table representing the Q-function [15]. Then, for each timestep t, it performs the following actions:

1. Observe the current state s_t
2. Select and perform the action a_t
3. Observe the new state s_{t+1}
4. Receive the immediate reward r_t
5. Adjust the Q-Table using the following formula:

$$Q_t(s_t, a_t) = (1 - \alpha)Q_{t-1}(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a_{t+1}} Q_{t-1}(s_{t+1}, a_{t+1}) \right]$$

We introduce α here as the learning rate. This learning rate indicates the proportion of old and new knowledge which should be accepted by the agent. Without a learning rate, the Q-values in the table will change erratically and never converge. However, by introducing a learning rate it can be proven that $Q_t(s, a) \rightarrow Q^*(s, a)$ as $t \rightarrow \infty$, where Q^* represents the optimal Q-table [15].

This approach, of course, requires enough memory to store the full Q-table, which may not be feasible for domains with sizable state spaces or where useful environment features cannot be extracted manually [8]. Relying on manual feature extraction is not ideal as it often requires a human expert to determine which features are useful. This may be subject to selection bias if the designer is unaware of abstract features which play a role in completing the task. To solve this, a function approximator for the Q-value is used instead of storing the values in a table. This function approximator frequently takes the form of a neural network [9].

2.3 Deep Q-Learning

Deep Q-Learning refers specifically to the application of deep neural networks (DNNs) to the Q-Learning algorithm. DNNs have a powerful ability to learn multiple levels of abstraction from data [7, 4] and are thus a suitable choice for making generalizations about the Q-function based on these learned abstractions. With sufficient data, these abstractions may even prove to be better representations than human-designed features [9].

Because deep Q-learning does not store an action-value table and instead uses a DNN as a function approximator, we use a parameterized Q-function $Q(s, a; \theta) \approx Q(s, a)$. Here, θ represents the parameters of the DNN. For training, we substitute the Q-Learning iterative update process with gradient descent minimizing the loss function:

$$L_t(\theta_t) = (y_t - Q(s_t, a_t; \theta_t))^2$$

Where y_i is:

$$y_t = r_t + \gamma \max_a Q(s_{t+1}, a; \theta_t)$$

This loss function is based on the intuition that we should expect the model to predict the Q-value at timestep t as the reward obtained from the action selected in that state, plus the maximum expected reward obtainable from the new state as predicted by the model. This does, in some sense, preserve the iterative update process used in Q-learning since the model parameters at the previous timestep θ_t are used to determine the new model parameters

θ_{t+1} in the next timestep.

Throughout the training process, we assume the best strategy the agent can take is to always select the action for which the Q-function approximates the highest value. However, in the early training phases, these Q-value estimations are a product of the model's initialization and not of learned experience. As a result, relying purely on the model's selected actions may not produce helpful training examples from which useful improvements can be discovered. To account for this, the Q-learning algorithm implements an ϵ-*greedy* strategy [9] where the model's action is selected with probability $1 - \epsilon$ and a random action is selected with probability ϵ. The value of ϵ is set near 1 at the start of training and then decays to a small value throughout training.

2.4 Improvements to Deep Q-Learning

Although Deep Q-Learning has proven successful in a variety of learning tasks, the default implementation is prone to errors such as value overestimation [3, 10] or catastrophic forgetting and poor selection of learning experiences [11, 2]. Value overestimation is a result of using the same network both for policy evaluation and for state predictions during the learning process. Catastrophic forgetting can be caused when the available size for replay memory is small and useful learning experiences are discarded to make room for newer, but less useful experiences. By increasing the size of replay memory, however, selecting good learning experiences can become a problem. This is because the default model has no way to distinguish useful experi-

ences from non-useful experiences. In this section, we introduce proposed methods of addressing these problems.

2.4.1 Selecting Good Learning Experiences

In the default Deep Q-Learning algorithm, learning experiences are sampled from replay memory with a uniform probability distribution. This ignores the fact that not all experiences are equally useful for the gradient descent process. Experiences that the current model predicted accurately will lead to a lower error and thus, a smaller change in network weights. Poorly predicted experiences, on the other hand, will have a higher error and result in the gradient descent step having a more dramatic impact on the network. Schaul et al. 2015 [11] introduces a method for weighting the probability of selecting experiences based on a priority metric, called prioritized experience replay (PER).

The primary concept of PER is to utilize the error computed for each experience during the training step as a metric for evaluating its priority. To prevent new experiences from being ignored completely, they are added to replay memory with maximum priority and later updated once used in the training process. A greedy PER implementation then selects the highest priority experiences for use during the training process.

However, Schaul et al. [11] warn against such a greedy implementation, citing sensitivity to noise spikes introduced early in the training process, errors in reward value approximation, and potential overfitting as issues that

may arise. Instead, they suggest using a sampling method in which the probability of selecting an experience is a function of its true priority value:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Here, p_i represents the priority assigned to experience i. α is a new parameter that may be tuned based on the desired impact of priority on replay memory sampling. Setting $\alpha = 0$ will result in the default uniform sampling behavior, while $\alpha = 1$ results in a weighting based completely off priority.

Two methods are supplied for calculating p_i. The first is "proportional prioritization" where $p_i = |\delta_i| + \epsilon$. Here, δ_i represents the training error for experience i and ϵ is a small positive constant that prevents experiences from have a zero probability of being selected when the error is zero. The second method is "rank-based" where $p_i = \frac{1}{\text{rank}(i)}$. Using this method, rank(i) is the rank of experience i in order of $|\delta_i|$.

The final modification PER makes to the Deep Q-Learning algorithm is to introduce importance-sampling weights. Because the original algorithm relies on selected training experiences having the same distribution as the expected environment experiences [11], PER can cause issues due to modifying the training experience distribution. The importance-sampling (IS) weights are calculated using the following formula:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

Where N represents the total number of experiences stored in replay memory, and β is a new parameter responsible for controlling how strongly the IS

weights influence the training process, with a value of $\beta = 1$ corresponding to full compensation. Schaul et al. 2015 [11] note that this value is often linearly annealed to 1 over the course of the training process.

2.4.2 Improving Q-value Approximation

The Q-Learning algorithm relies on its own prediction of state values to perform the iterative update step. By definition, this means that more accurate predictions will lead to more accurate updates, resulting in a faster-learning and higher performing resulting model. During the learning process, however, it is expected that the prediction error will be high since the model has yet to acquire any information about the environment or its rewards. This causes frequent overestimation issues which may lead to the agent learning a sub-optimal policy [3].

Hasselt et al. 2016 [3] attempt to mitigate this issue by using an approach known as Double Deep Q-Learning. In this approach, a second network θ^- is maintained and used for value estimations during the training process. This network is referred to as the target network. We then find the value of y_t used for computing loss as follows:

$$y_t = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta^-)$$

Periodically after a fixed number of training steps θ^- is updated with the weights of θ_t.

This does not completely remove the bootstrapping component of computing DQN loss [3], but it serves to improve the stability of q-value estimations by using a fixed target network for evaluating states during the training step. This can mitigate the overestimation problem seen in the standard Deep Q-Learning implementation because it is less susceptible to sporadic network updates which may occur in the online network early in the learning process.

Duryea et al. 2016 [1] explore the idea of using secondary networks to improve value estimations in the learning process further by introducing the Multi Q-Learning approach. In this approach, instead of using a single target network, the agent maintains a collection of n networks θ^0, θ^1, $\cdots$, θ^n. During each step of the training process, a single network θ^k is randomly selected and used for the greedy policy. However, during the training step, value estimations are computed using an average across all θ, excluding θ^k:

$$y_t = r_t + \gamma \frac{1}{n-1} \sum_{i=0, i \neq k}^{n} \left[Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta^k); \theta^i) \right]$$

By creating an arbitrary number of completely independent networks, this approach serves to fully decouple value estimation in the training step from action selection. In doing so, value estimation stability is improved [1] because large changes to a single network will not impact the average value estimate.

2.5 Contributions of this Thesis

Of particular interest to this thesis is the combination of the Multi Q-Learning algorithm with Prioritized experience replay. Previous research has

shown both of these modifications to improve specific aspects of the Deep Q-Learning algorithm [3, 5, 11], but has yet to examine the interaction between them. This interaction is expected to have unique outcomes due to the Multi Q-Learning implementation maintaining multiple independent networks. These networks may produce different errors for the same transition when sampled from replay memory. As a result, the priority assigned to a given transition is highly dependent on the active network which selects the appropriate action for that step.

In this thesis, the approach of combining the Multi Q-Learning algorithm and PER with no modifications is referred to as the naive implementation. This implementation is based on the expectation that the dependence of a transition's priority on the currently active Multi Q-Learning network will not impede the performance gains offered by PER or Multi Q-Learning in standard circumstances. The efficacy of this approach is discussed in the analysis section, and potential alternatives are suggested.

3 Experiments

In this thesis, the experiments aim to compare the performance of the Deep Q-Learning algorithms using the various modifications introduced above. It also aims to examine the impact and compatibility of PER implementations of these algorithms. The test matrix included the standard algorithm, Double Q-Learning, Multi Q-Learning with four networks, and Multi Q-Learning with eight networks, with variants of each for both PER and non-PER im-

plementations. Each of these algorithms were tested in the OpenAI gym framework's CartPole-v1 and LunarLander-v2 environments. Learning rates of 0.1, 0.01, 0.001, and 0.0001 were tested for each algorithm and the best performing models were selected for analysis.

The CartPole and LunarLander environments were selected due to their frequent use as benchmarks for Reinforcement Learning algorithms and their varying reward distribution. In the CartPole environment, the agent is provided with a reward value of 1 for every step in which it can maintain the pole's balance. In the LunarLander environment, the agent receives negative rewards for each step in which it uses the lander's thrusters or crashes, and only receives a positive reward when either landing rod is in contact with the ground. Bonus points are awarded for landing within the designated goal.

Each model was trained for a total of 250 episodes and used an ϵ-greedy approach to action selection during the training process. The ϵ value was linearly annealed 1.0 to 0.01 over the course of $10,000$ training steps. A discount factor $\gamma = 0.99$ was used for each model. The replay memory size was limited to $250,000$ transitions, which was selected to be sufficiently large such that no model was required to overwrite previous transitions. During each step of the training process, the models were updated using a batch of 128 transitions sampled from the replay memory. For the Double DQN implementations, the target network was updated every 100 training steps in the CartPole environment and every 500 training steps in the LunarLander environment. For the Multi Q-Learning models, β was linearly annealed

from 0.4 to 1.0 over the course of the training process.

The underlying DQN implementation was fixed for both environments and consisted of a feed forward neural network containing a single hidden layer of 256 neurons using the ReLU activation function.

Test Matrix

Num. Networks	**No PER**	**PER**
1	Standard DQN	Standard DQN w/ PER
2	Double DQN	Double DQN w/ PER
4	MultiQ 4 Networks	MultiQ 4 Networks w/ PER
8	MultiQ 8 Networks	MultiQ 8 Networks w/ PER

3.1 Implementation

The neural network implementation used for this thesis was built in Python using the PyTorch and NumPy libraries. PyTorch is an open-source machine learning framework that manages computational graphs for neural networks and performs automatic gradient calculations. It is largely based on the Torch library for the C programming language. Its primary use in this experiment was for building the DQN, performing backpropagation, and calculating network loss using the framework's built-in Huber loss function

NumPy is a mathematical framework that adds support for many array and matrix-based operations. NumPy is open source and is a standard library for scientific computing. It was used in this experiment for the handling of episode data, including q-value predictions, rewards, and losses. It also fa-

cilitated the replay memory and PER implementations.

The full implementation and all supporting code can be found at the following link: https://github.com/caleb98/dqlearning

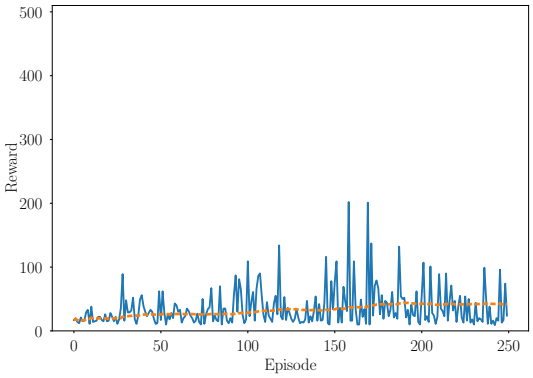
4 Results

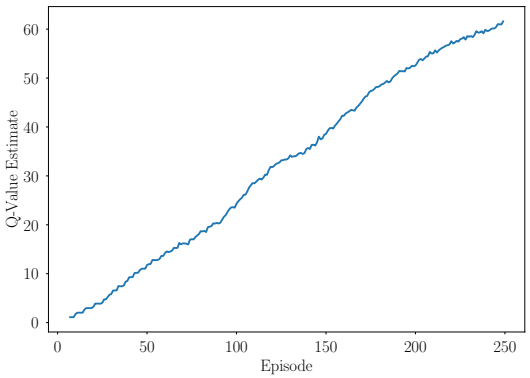
4.1 CartPole Environment

Data for each algorithm's performance in the CartPole-v1 environment is given in the figures below. Blue lines indicate the per-episode reward obtained by the agent during the learning process, while orange lines indicate the average reward obtained per episode from the previous 100 episodes.

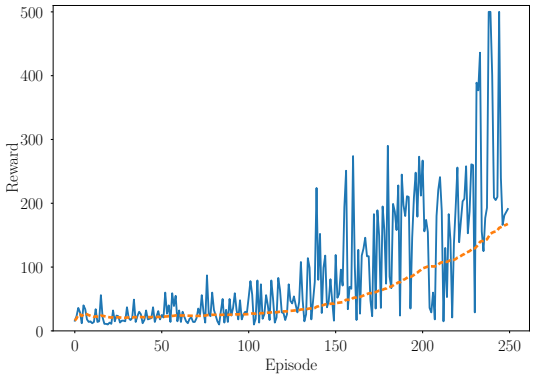
Standard DQN ($\alpha = 0.001$)

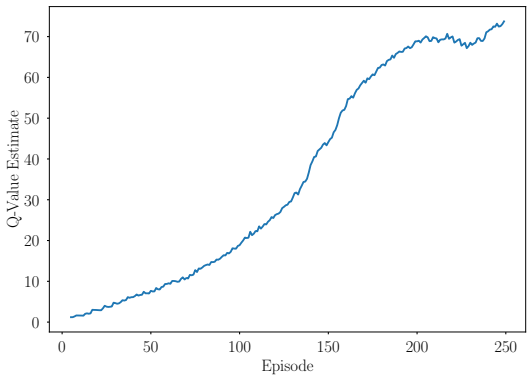
Regular





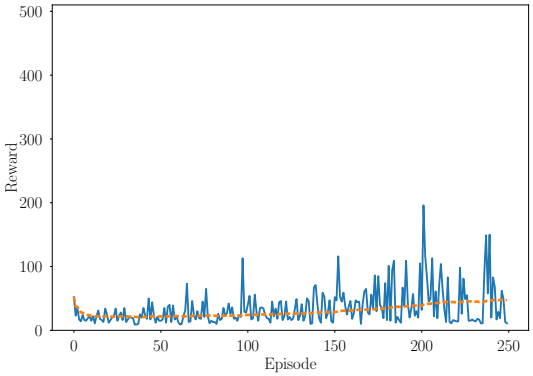
With PER

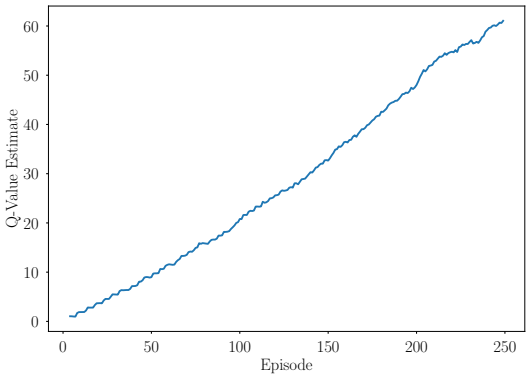




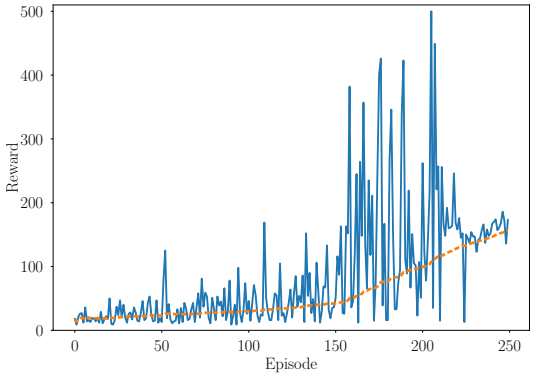
Double DQN ($\alpha = 0.001$)

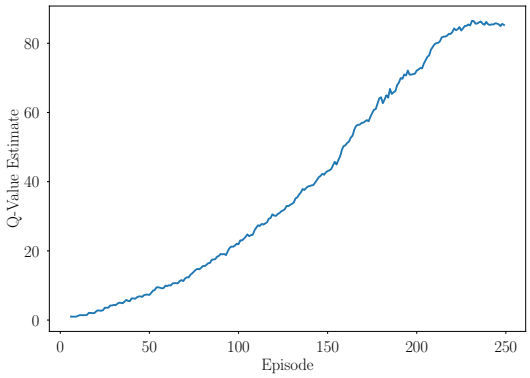
Regular





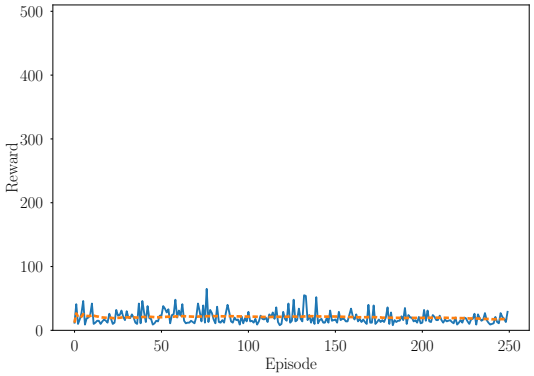
With PER



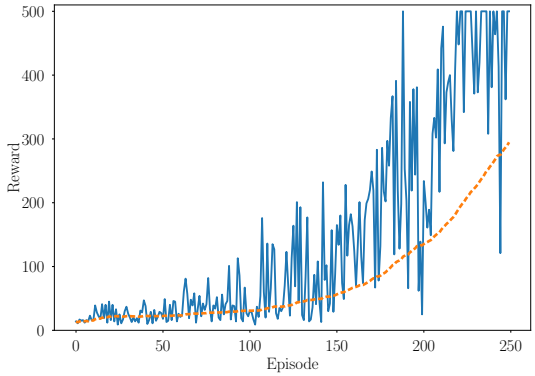


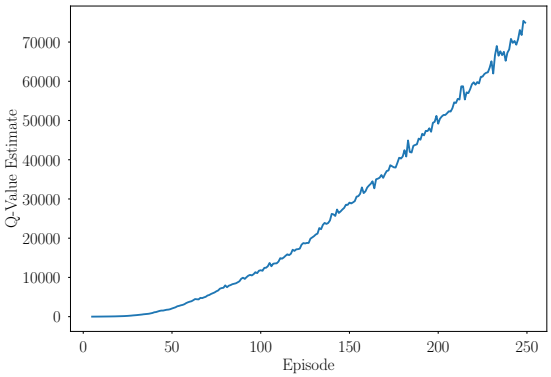
MultiQ 4 Networks ($\alpha = 0.01$)

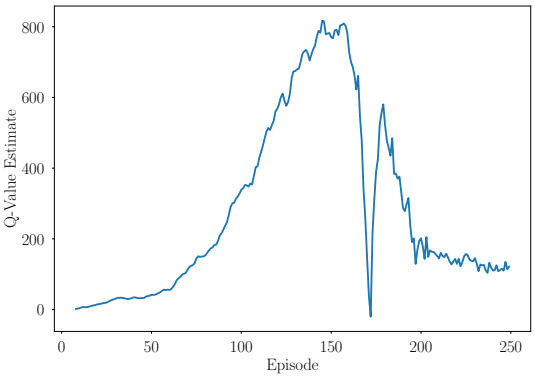
Regular



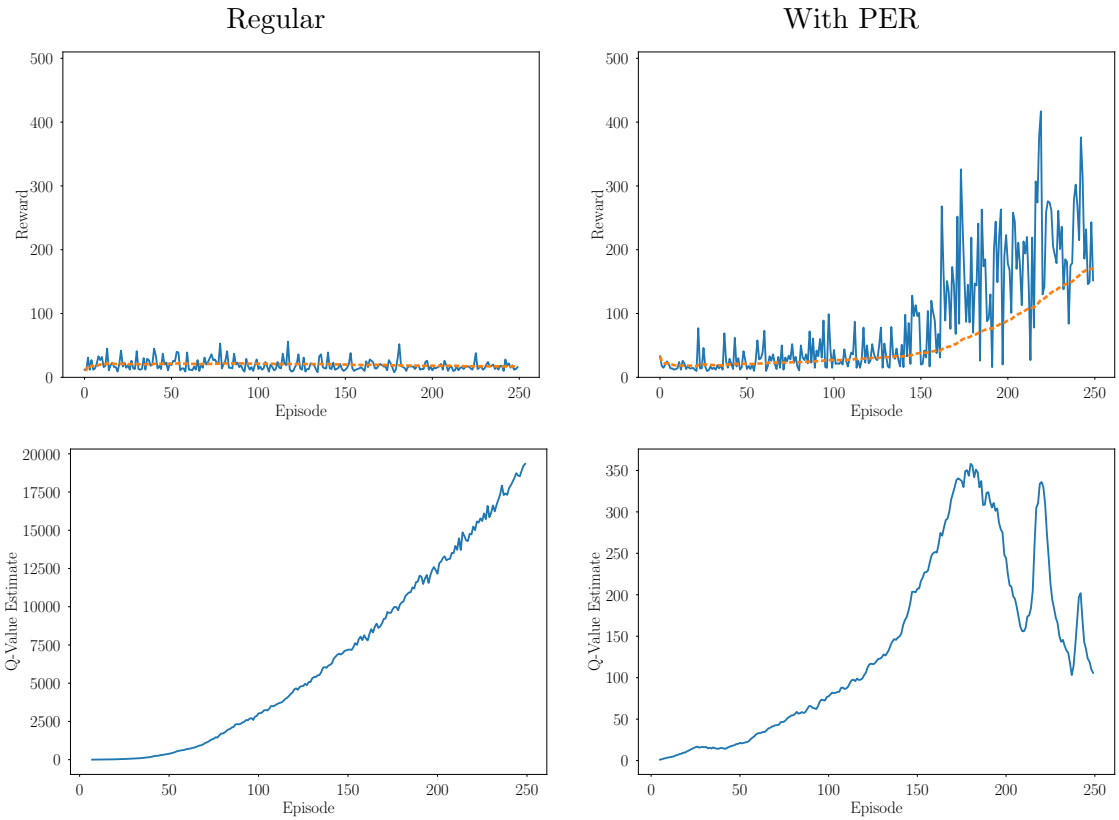
With PER







MultiQ 8 Networks ($\alpha = 0.01$)



Mean Reward across Final 50 Episodes

Standard		**Double DQN**		**MultiQ4**		**MultiQ8**	
No PER	**PER**	**No PER**	**PER**	**No PER**	**PER**	**No PER**	**PER**
35.84	201.32	48.94	169.4	16.94	396.72	16.48	210.14

For the pole balancing task, every model saw significant performance increases when using Prioritized Experience Replay compared to their regular counterparts. The best performing model was the Multi Q-Learning model implemented with four backing networks and using PER, which produced a perfect score in the environment as early as episode 189 during training and

had a final 50 episode average reward of 396.72.

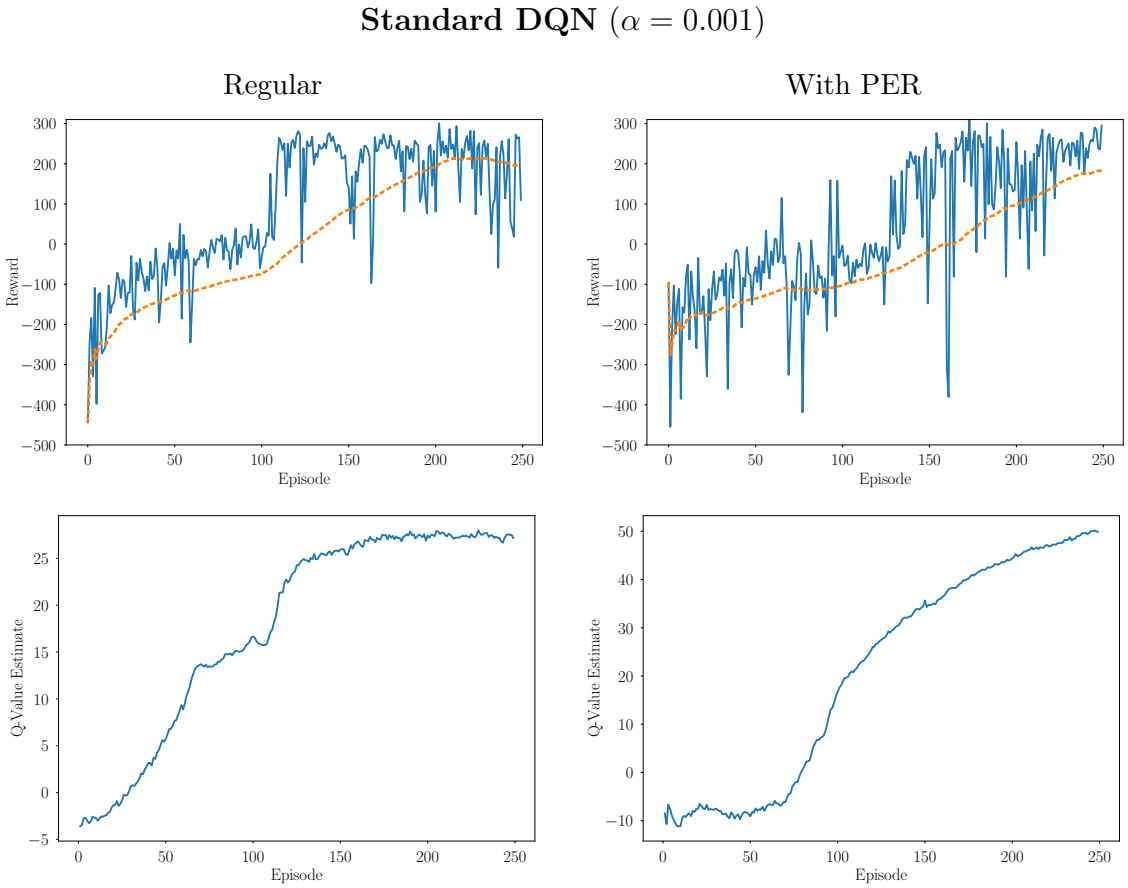
The eight network variant of Multi Q-Learning with PER also achieved strong performance with a final 50 episode average reward of 210.14, though it never successfully solved the task during the training process. This performance was similar to the standard DQN algorithm with PER, which had a final 50 episode average reward of 201.32 and was able to achieve a perfect score of 500 during 3 episodes of the training process. Surprisingly, the Double DQN model showed the worst performance with PER enabled. However, among the non-PER variants, it performed the best with a final 50 episode average reward of 48.94.

By observing the q-value estimation figures, we can see that the failure of the Multi Q-Learning models to learn the environment without PER was the result of q-value estimations exploding. In their PER variants, there is still a tendency to overestimate the q-values during the middle of the training process. However, this overestimation is corrected by the end of training and q-value predictions appear to start converging.

4.2 LunarLander Environment

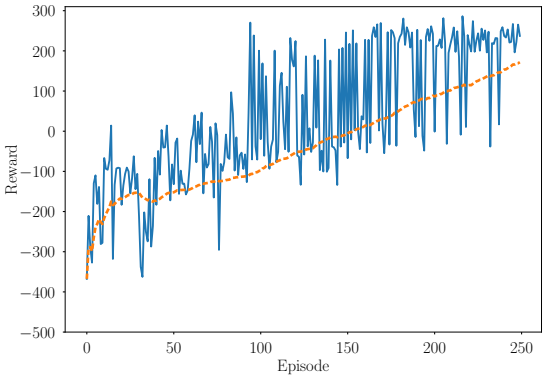
Data for each algorithm's performance in the LunarLander-v2 environment is given in the figures below. As before, blue lines indicate the per-episode reward obtained by the agent during the learning process, while orange lines indicate the average reward obtained per episode from the previous 100

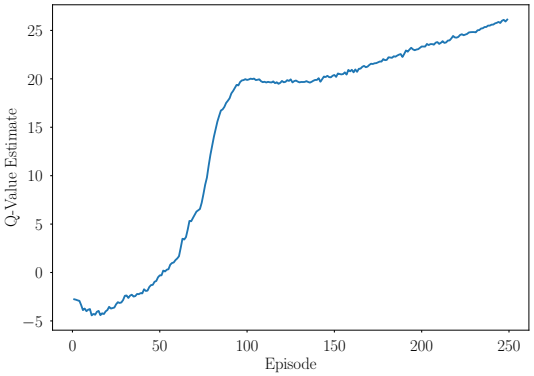
episodes.


Standard DQN ($\alpha = 0.001$)
Regular
With PER
Reward
Episode
Q-Value Estimate
Episode
Reward
Episode
Q-Value Estimate
Episode

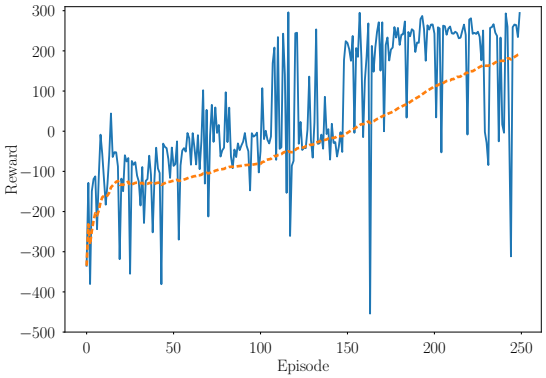
Double DQN

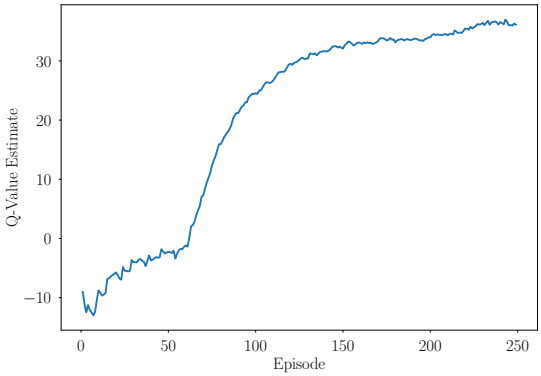
Regular ($\alpha = 0.0001$)





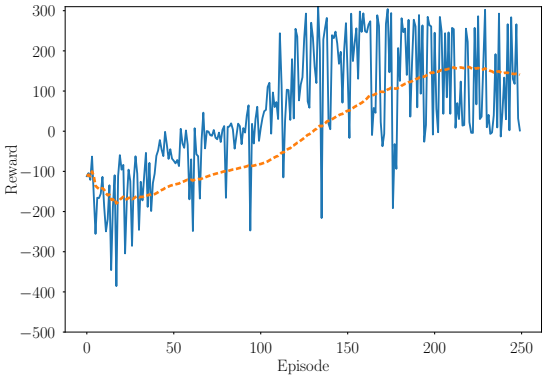
With PER ($\alpha = 0.001$)

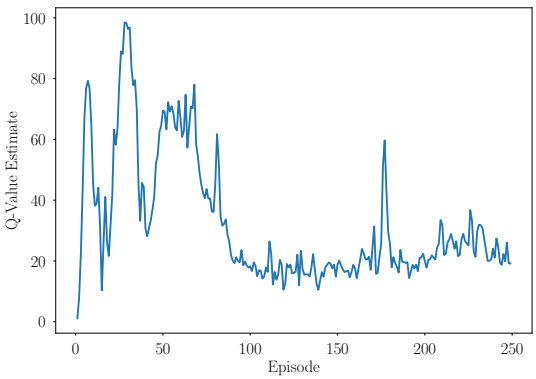




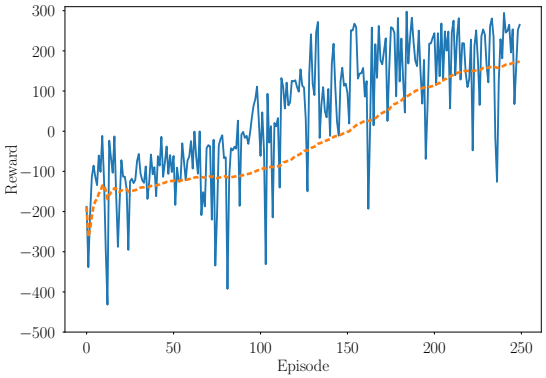
MultiQ 4 Networks ($\alpha = 0.01$)

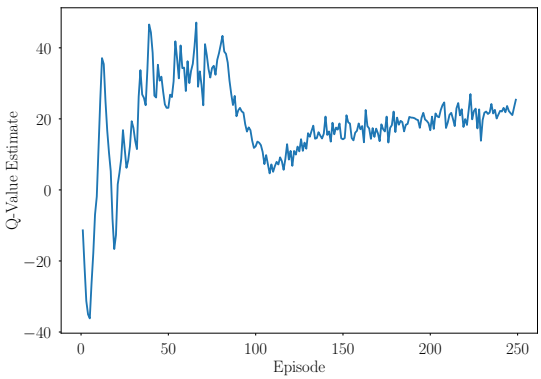
Regular



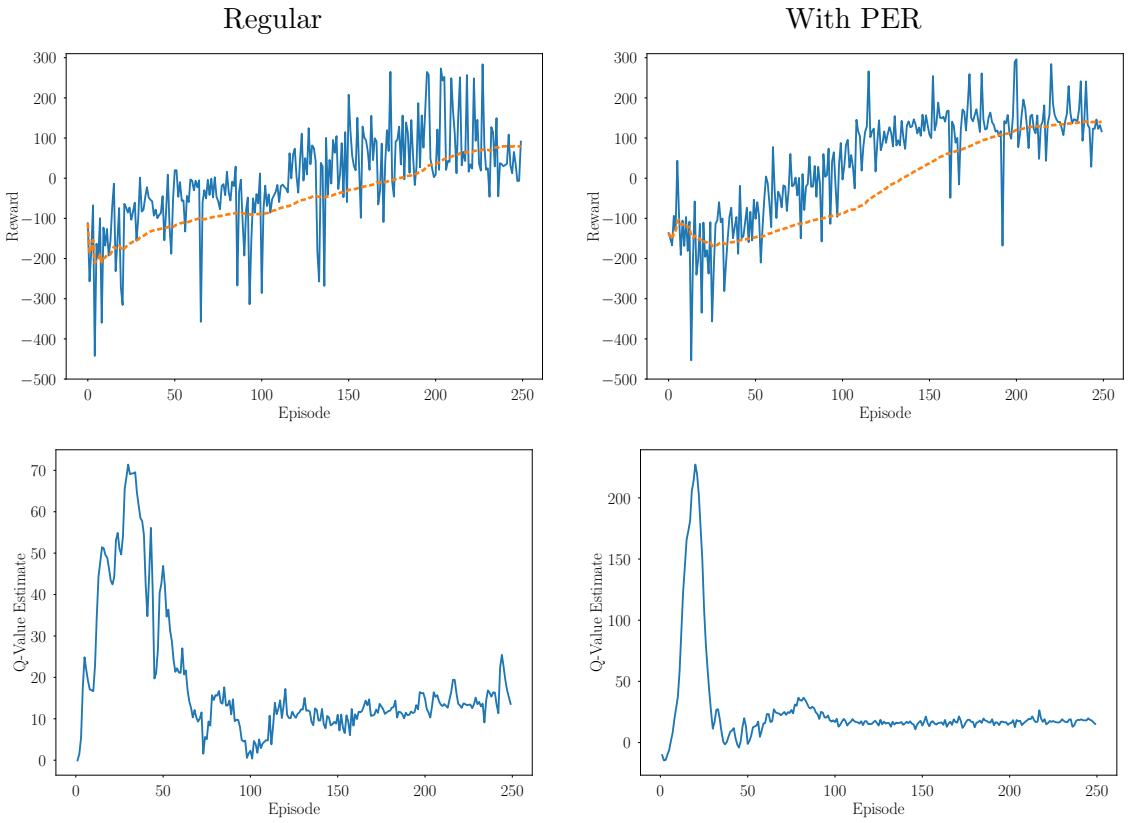


With PER





MultiQ 8 Networks ($\alpha = 0.01$)



Mean Reward across Final 50 Episodes

Standard		**Double DQN**		**MultiQ4**		**MultiQ8**	
No PER	**PER**	**No PER**	**PER**	**No PER**	**PER**	**No PER**	**PER**
196.19	206.56	202.05	186.56	115.55	186.79	84.50	144.82

For the LunarLander environment, results were much less varied. All models were able to successfully complete the task without crashing at least once, though the model with the highest performance was, surprisingly, the Standard DQN implementation using PER with a final 50 episode average of 206.56. Both Double DQN variants also produced strong models, with final 50 averages of 202.05 and 186.56, respectively. The Multi Q-Learning model

with 4 networks and PER was comparable to the Double DQN PER model, with a final 50 average of 186.79. The Multi Q-Learning model with 8 networks was not able to achieve the same performance and finished with a final 50 average reward of 144.82.

Although PER did not strictly improve performance for each model in the lunar lander environment – as was the case in the pole balancing environment – there was a notable improvement in performance stability in the models which did implement PER.

5 Discussion

5.1 Best Performing Models

The strongest performing model for the CartPole environment was the Multi Q-Learning model using 4 networks and PER. It was able to successfully solve the task as early as episode 189 during the training process and demonstrated reliable performance in the episodes that followed. Compared to the non-PER equivalent, which was unable to show any signs of learning throughout the training process, this was a significant improvement.

The strongest performing model for the LunarLander environment was the Standard DQN with PER. That being said, performance for each model in the LunarLander environment was much more similar compared to the CartPole environment. The non-PER variant of the Standard DQN and Double

DQN both achieved performances within 6% of the Standard DQN PER model. Further, every model showed signs of improvement over the course of the training process, unlike the CartPole environment where the Multi Q-Learning models showed no improvement.

The CartPole results generally matched the expectations for each algorithm's behavior. PER showed significant performance gains for each model, and the added stability of the Multi Q-Learning models allowed them to find more reliable and robust solutions earlier in the training process. The inability of the Multi Q-Learning models to solve the task without PER was caused by an exploding q-value estimation, which can be observed in the q-value estimation figures for these models. Given that these same models had strong performance when PER was implemented, the results suggest that this overestimation issue was a result of poor training experience selection for the non-PER variants.

The LunarLander results, on the other hand, had more unexpected outcomes. For one, the addition of PER was not a global improvement – the non-PER Double DQN model performed better than its PER variant. Even for the Standard DQN model, PER's relative performance gain was much smaller than in the CartPole environment. A possible cause for this behavior is if these models had relatively little variance in the error of their q-value predictions. When this is the case, the priority-weighted sampling of experiences from replay memory would roughly match that of a random sample. This could occur when each of the possible states that the agent can be in are of

near-equal importance. This is plausible for the LunarLander environment since a successful landing requires precise inputs for the entire duration of the episode. A single imprecise input could potentially put the lander into a state where successfully landing is no longer possible.

5.2 Impact of PER

One notable observation that can be made from the training data is how PER impacted the resulting performance of each model. In the CartPole environment, PER showed universal improvement among all the tested models. For the Standard and Double DQN models, PER significantly improved the resulting performance by a factor of 5.62 and 3.46, respectively. The performance gains for the Multi Q-Learning variants were even more significant, as these models showed no signs of improvement over the course of the training process without PER. However, when PER was implemented, the Multi Q-Learning variants demonstrated stronger performance than both the Standard PER and Double DQN PER implementations.

In the LunarLander environment, the performance impact of PER was small or negligible for the Standard and Double DQN models , but still showed significant improvements for the Multi Q-Learning models. Unlike in the CartPole environment, however, both Multi Q-Learning models did show improvements over the course of the training process even without the use of PER. Based on this data, it can be concluded that the relative performance

gains of implementing PER are larger for Multi Q-Learning-based models. This interaction is further discussed in the relevant section below.

5.3 Multi Q-Learning Observations

In both the CartPole and LunarLander environments, increasing number of networks used from 4 to 8 for the Multi Q-Learning algorithm decreased the resulting performance at the end of the 250 episode training cycle. This is an expected result as including more networks is meant not to improve the speed of training, but rather the stability [1]. When observing the performance consistency and stability between Multi Q-Learning with 4 networks and 8 networks, we can see that this trend does follow. The q-value estimations reflect this trend, as well, as the 8 network variants had less noisiness in general than the 4 network variants.

One observation worth taking into account in the learning rate used by the Multi Q-Learning variants. In both the CartPole and LunarLander environments, the optimally performing Multi Q-Learning models were found using $\alpha = 0.01$. In general, the other DQN variants performed optimally at $\alpha = 0.001$. This suggests that, in general, Multi Q-Learning algorithms are capable of using larger learning rates. This is likely due to the increased stability offered by using multiple individual networks. Even if a single network overcorrects due to the learning rate being too high, the model as a whole is not compromised in the same way the Standard or Double DQN models

would be.

5.4 Multi Q-Learning Interactions with PER

As mentioned previously, the interaction between Multi Q-Learning and PER was of particular interest in this thesis due to the lack of previous research on the topic. Based on the results, we conclude that the naive implementation of Multi Q-Learning with PER is a viable method and retains both the performance improvements of PER and the stability improvements of Multi Q-Learning. This is largely noticeable in the CartPole environment where the Multi Q-Learning models failed to show any signs of improvement without PER. The trend holds for the LunarLander environment as well, though less drastically so.

Multi Q-Learning's relatively low performance without PER is likely caused by the fact that Multi Q-Learning is effectively training multiple networks simultaneously. Without PER, the rate of each individual network's improvement is limited by how frequently the random sampling of experiences from replay memory selects good transitions. The same can be said for the Standard and Double DQN models; however, the impact of PER is different in Multi Q-Learning due to the way q-value predictions are calculated during the training step and how PER calculates an experience's priority.

Recall that, in Multi Q-Learning, q-value predictions are calculated by se-

lecting the action the currently active network would take in a given state, then finding the mean q-value estimation for each of the inactive networks for that action in the state. This naturally means that any error in the q-value estimation represents an experience that is beneficial not only to a single network but to all networks in the Multi Q-Learning model. The result when using the naive implementation is that, with each training step, the selected transitions benefit the quality of the full model rather than a single network within the model.

6 Conclusion & Future Research

The results discussed above highlight the effects of applying various popular modifications to the Deep Q-Learning algorithm in two standardized testing environments. It also provides insight into how modifications which apply to different parts of the DQN algorithm may have different results when they are implemented together (such as the case of Multi Q-Learning and PER). Although plausible explanations for some of the phenomena present in this experiment's results are given, further research is warranted to discover the precise ways in which these interactions occur.

For one, the relative performance impact of each DQN algorithm modification compared to the Standard DQN performance varied greatly between the CartPole and LunarLander environments. This is not entirely surprising due to the difference in reward distribution between these environments.

However, gaining a deeper understanding of how reward distribution affects each algorithm can help to determine when certain modifications should be implemented.

Another interesting outcome was the interaction between PER and the Multi Q-Learning algorithm. Because PER utilizes the model's prediction error to prioritize specific experiences within the replay memory, it was not guaranteed that performance impacts would be as significant in the Multi Q-Learning algorithm where q-value predictions are a function of multiple neural networks. Although the naive approach was successful in this thesis's experiments, there are certainly other approaches that warrant additional research. For example, the replay memory data structure could be modified to maintain separate collections of transitions for each network in use and only sample from the active network's pool. Alternatively, a given transition could be stored with multiple priorities for each network. It is suggested that future studies examine such novel PER approaches to determine if better methods exist for determining a transition's value under the Multi Q-Learning algorithm.

References

[1] Ethan Duryea, Michael Ganger, and Wei Hu. "Exploring Deep Reinforcement Learning with Multi Q-Learning". In: *ICA* 07.04 (2016), pp. 129–144. ISSN: 2153-0653, 2153-0661. DOI: 10.4236/ica.2016.74012. URL: http://www.scirp.org/journal/doi.aspx?DOI=10.4236/ica.2016.74012 (visited on 11/19/2021).

[2] Ian J. Goodfellow et al. *An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks*. Mar. 3, 2015. arXiv: 1312.6211 [cs, stat]. URL: http://arxiv.org/abs/1312.6211 (visited on 03/20/2022).

[3] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (1 Mar. 2, 2016). ISSN: 2374-3468. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10295 (visited on 11/04/2021).

[4] Matthew Hausknecht and Peter Stone. *Deep Recurrent Q-Learning for Partially Observable MDPs*. Jan. 11, 2017. arXiv: 1507.06527 [cs]. URL: http://arxiv.org/abs/1507.06527 (visited on 11/10/2021).

[5] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. Oct. 6, 2017. arXiv: 1710.02298 [cs]. URL: http://arxiv.org/abs/1710.02298 (visited on 11/10/2021).

[6] Guillaume Lample and Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning". In: *Thirty-First AAAI Conference*

on Artificial Intelligence. Thirty-First AAAI Conference on Artificial Intelligence. Feb. 13, 2017. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14456 (visited on 10/21/2021).

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (May 28, 2015), pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539. URL: http://www.nature.com/articles/nature14539 (visited on 10/19/2021).

[8] Volodymyr Mnih et al. "Human-Level Control through Deep Reinforcement Learning". In: *Nature* 518.7540 (Feb. 26, 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236. URL: http://www.nature.com/articles/nature14236 (visited on 11/10/2021).

[9] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. arXiv: 1312.5602 [cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 10/19/2021).

[10] Mohammed Sabry and Amr M. A. Khalifa. *On the Reduction of Variance and Overestimation of Deep Q-Learning*. Oct. 14, 2019. arXiv: 1910.05983 [cs, stat]. URL: http://arxiv.org/abs/1910.05983 (visited on 03/20/2022).

[11] Tom Schaul et al. "Prioritized Experience Replay". In: (Nov. 18, 2015). URL: https://arxiv.org/abs/1511.05952v4 (visited on 11/19/2021).

[12] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 28, 2016), pp. 484–489. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature16961. URL: http://www.nature.com/articles/nature16961 (visited on 10/21/2021).

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html (visited on 03/20/2022).

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.

[15] Christopher Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8 (May 1, 1992), pp. 279–292. DOI: 10.1007/BF00992698.