DQN Breakdown

Vihaan Akshaay Rajendiran, Zaiwei Chen, Siva Theja Maguluri

August 13, 2021

Abstract

Deep Q-Network(DQN) [1] is one of the early deep reinforcement learning methods proposed by DeepMind which combines reinforcement learning with a class of artificial neural networks to approximate a state-value function in a Q-learning framework. It used a Convoluted Neural Network architecture as a function approximator and uses three techniques (-namely Experience Replay, Target Network, Error Truncation) to overcome instability and divergence. The research below examines the importance of each of these factors for solving environments [2] based on control theory problems from the classic RL literature [3] (Acrobot, CartPole and MountainCar). The experiments are conducted with different combinations of these techniques present. Furthermore, each of these techniques were analysed exclusively on Acrobot environment and the common trends are presented.

Contents

1 Introduction

Human beings have the ability to learn several tasks through out their lives without having to be given explicit instructions. Thanks to the innate reward system in the brain, our body gets a feedback for every action that we take and helps us learn about our environment consciously or unconsciously. The body is designed in such a way as to cause discomfort and hunger when food is needed and makes one feel better after eating. This is a natural system inherently present in the body which helps one learn to eat.

1.1 Reinforcement Learning

Formalizing the setting that Reinforcement Learning Algorithms [3] work with, The decision maker is called the **Agent**. The agent continuously interacts with its surroundings called **Environment**. These interactions consist of an **Action** that the agent takes that alters the environment's **State**. The agent also receives a scalar **reward** for every step that it takes. The goal of the agent is to maximize the total reward it gets. To model the real world better, it would make sense for immediate rewards to matter more for the agent than future reward. The **Discount factor** ($0 \leq \gamma \leq 1$) is a constant that is multiplied to future rewards to make it less impactful. We define this new discounted cumulative reward as the **Return** and the goal of the agent is to maximize the Return from every state. A **Policy** can be thought of as a rule book that helps an agent to decide the action to take from a given state. We can consider that an Environment is solved if we find the optimal policy that helps the agent choose actions such that it receives the maximum return corresponding to each possible state in the environment.

This straightforward framing of the problem of learning from interaction to achieve a goal can be represented by a **Markov Decision Process** (MDPs) [3]. The interactions can be formalized as follows. The agent and environment interact at discrete time steps, $t = 0, 1, 2, 3, ...$ .At each time step t, the agent receives some representation of the environment's state, $S_t \in \mathcal{S}$ and with the help of a policy Π_t selects an action $A_t \in A(s)$. The Reward that the agent obtains at a particular step is denoted by R_t and the total discounted cumulative reward (Return) that the agent obtains from a particular state is given by G. The goal of the agent is to obtain the optimal policy π^* which helps

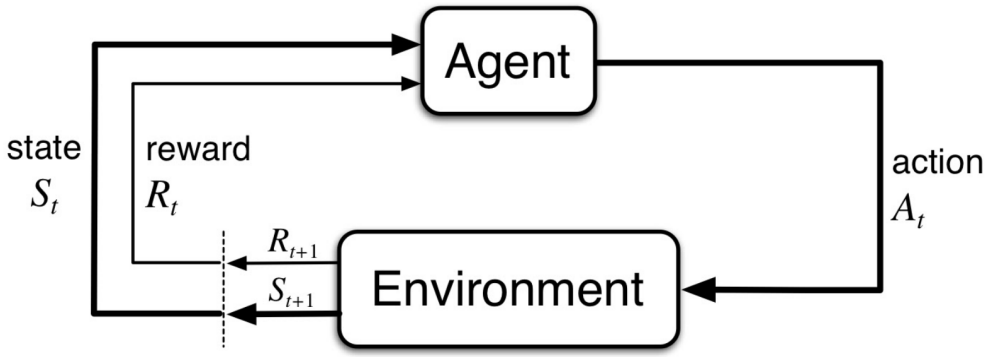


Figure 1: The Agent-Environment Interaction[3]

the agent choose actions in each state that returns the maximum return(G) possible.

1.2 Q-Learning

We start this section by introducing Q-Values. These are scalars we assign to each state-action pair. These are supposed to be the equivalent of the expected Return for the agent taking a certain action from a particular state. The advantage of this formulation is that, once the Q-Values converge to the values that we desire (optimal returns), just following the action that has the maximum Q-Value in each state, gives us the optimal policy to solve the environment.

Q-Learning [4] is an off-policy reinforcement learning algorithm that is used to find the best action for each state. For performing the Q-learning algorithm, we start with a **Q-Table** which is essentially a matrix of shape, states x actions, which has the corresponding Q-value in each of its entries. This becomes the reference table for the agent to choose actions.

The next step is for the agent to interact with the environment and update these Q-Values. The agent is designed as to first choose to **explore** more (choose random actions in each state) and update the Q-values. As the agent keeps gaining more experience, it is slowly made to **exploit** its knowledge and choose the action it thinks best.

With each of these interactions, we utilize the transition data (current state - S_t, current action - A_t, next state - S_{T+1}, and the reward R_t) to iteratively update the Q-Value (corresponding to state S_t and action A_t). The stochastic iterative update rule solving the Bellman equation is given

below:

$$Q^{new}(S_t, A_t) = Q(S_t, A_t) + \alpha(R_t + \gamma . \max_a Q(S_{t+1}, a) - Q(S_t, A_t)).$$

Where α is the control parameter/ learning rate, and γ is the discount factor.

Once we have done enough exploration of the environment, the Q-Table is expected to converge at optimal Q-Values. Once it has converged, the optimal policy to solve the environment would be to choose the action that has the maximum return. This can be achieved by choosing the action that has the maximum Q-value in a given state.

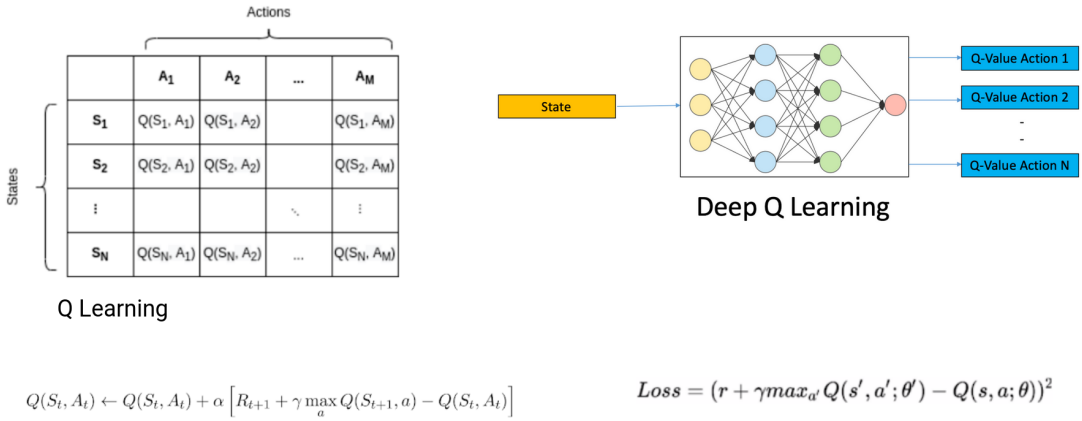
2 DQN

Q-learning is a very intuitive algorithm with a simple update rule and it is easy to obtain the policy corresponding to a Q-Table. Although Q-Learning works best with grid worlds and similar environments with limited states and actions, most real-life tasks do not follow the same trend. Even a simple task like balancing a pole on a cart (We'll specifically explore this environment later in the experiments) has infinite states. This causes the size of the Q-Table to be massive, which would lead to issues with memory as well as time taken to estimate each of the entries in the table with sufficient transitions. This 'curse of dimensionality' is one of Q-Learning's disadvantages.

Q-Learning also has issues with generality. Since it only saves values for certain states, if the agent encounters a new state, it as no clue which action to take.

This lead to the use of function approximators. In DQNs[1], instead of using a Q-Table, a deep neural network is used as function approximator. The input to the neural network is the current state, and the output is the corresponding Q-Values for each function.

In DQNs, let the parameters of the neural network be denoted by θ, (ie $Q(S_t, A_t) = Q(S_t, A_t; \theta_t)$). We want the predictions of the neural network to match the Bellman Equation. Therefore, the loss function can be defined as the mean squared error between the targets ($R_t + \max_A Q(S_{t+1}, A)$) and



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

Figure 2: Q-Table and Q-Network

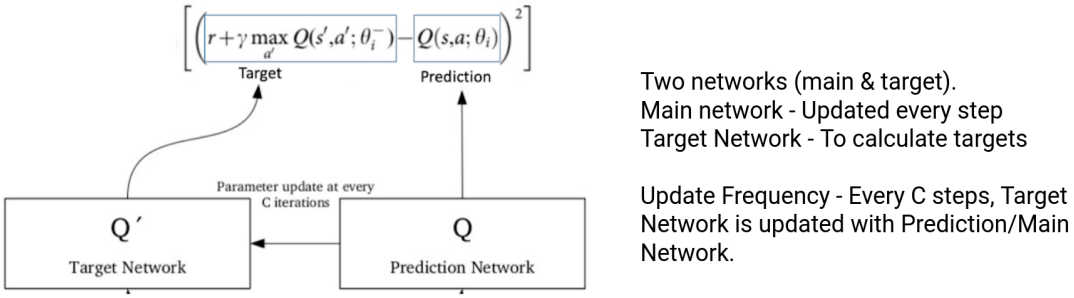


Figure 3: Q-Targets

the prediction ($Q(S_{t+1}, A_t)$) using stochastic gradient descent. The update rule is as follows:

$$\theta_{t+1} \leftarrow \theta_t + \alpha[(R_t + \max_A Q(S_{t+1}, A; \theta_t) - Q(S_t, A_t; \theta_t))\nabla_{\theta_t} Q(S_t, A_t; \theta_t)].$$

Although this representation allows neural networks to act as valid Q-Tables, it was observed that the neural networks were unstable and did not converge when trained with these properties. Additional techniques like 'Q-Targets','Experience Replay' and 'Truncation' were introduced by [insert paper name] to smooth out the learning and avoid divergence.

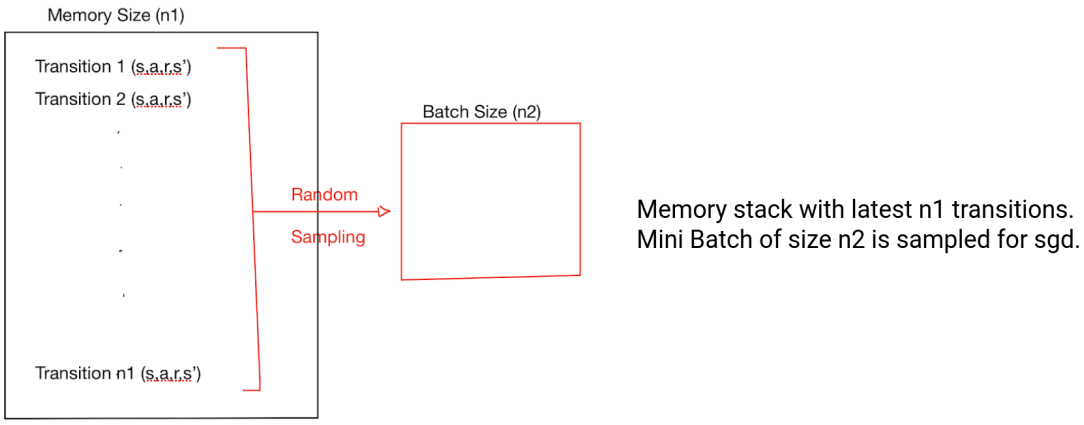


Figure 4: Experience Replay

2.1 Q-Targets

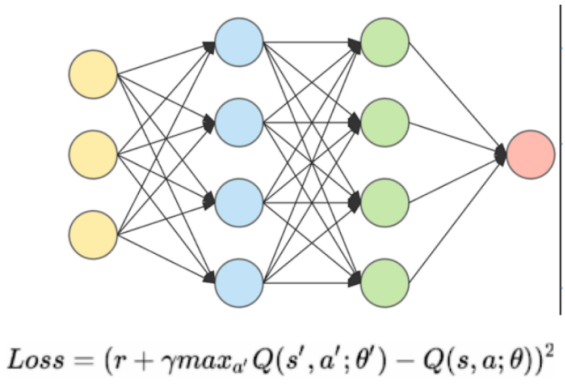
Revisiting the update equation, it can be noticed that we use the same neural network for the prediction as well as target. This is similar to learning how to shoot a specific target but the target keeps moving. This would lead to instability. To tackle this, DQN proposes Q-Targets [1]. The idea is to use another parallel neural network which is a copy of the main network. The main network is used for the predictions and the parallel target network is used to calculate the targets. That way, the targets are strong and stationary. Every 'C' steps, we update the target network with the weights of the main network. We denote this technique with the notation $\pm Q$, where $+Q$ would mean it being implemented and $-Q$ would mean, the technique is dropped. For our main experiments, we use an update frequency of 20 when Q targets are present. (i.e. we update the target network every 20 steps)

2.2 Experience Replay

In Reinforcement Learning, the samples we obtain (the trajectories in our case) are not independently and identically distributed [5]. Adjacent trajectories are highly correlated and the data distribution changes as the agent learns. To tackle this, the paper suggests to use a main data basket to hold a lot of trajectories and to sample few trajectories randomly from it regularly to

perform mini-batch stochastic gradient descent to update the main network. This uses samples in a more efficient manner and can help in stability. We denote this technique with the notation $\pm E$, where $+E$ would mean it being implemented and $-E$ would mean, the technique is dropped. For our main experiments, We use Experience Replay with a Memory size of 10,000 samples and draw mini-batches of size 64.

2.3 Truncation



$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

Gradient is clipped to +1/-1 for stability

Figure 5: Error Gradient Truncation

When training neural networks with the method mentioned above, it is possible that the network can diverge because of huge gradients. Since application of neural networks as function approximators here in RL is notorious for divergence, gradient truncation is also suggested to improve the stability of the neural network. We denote this technique with the notation $\pm T$, where $+T$ would mean it being implemented and $-T$ would mean, the technique is dropped. For our main experiments, we use a truncation limit of ± 1 on the update gradients.

3 Experiments & Results

The first experiment was to try and understand the impact of each of the three mentioned techniques(namely Q,E & T) on the convergence of the DQN algorithm on three standard control theory problem from the classic RL Literature [2] - (namely Acrobot, MountainCar and CartPole). We even analyze each of the properties further more on acrobot(one of the environments) to obtain more insights.

3.1 Experiments on Acrobot

3.1.1 Environment - Acrobot

In the Acrobot environment, the agent tries to swing up a double pendulum by applying either clockwise or anti-clockwise torque at the second joint and keep the tip above a certain line. The episode runs for a fixed amount of 500 steps and the agent receives -1 for each step where the tip is below the target line.

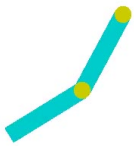


Figure 6: Environment - Acrobot

- The state space is a 6-dimensional vector - cos(theta1) sin(theta1) cos(theta2) sin(theta2) thetaDot1 thetaDot2.
- Three possible actions - clockwise/anticlockwise/no torque at the joint connecting the two links.

- The agent receives -1 reward for every step with the tip below the target and receives 0 for every step the tip is above the target.

3.1.2 8 Variations

Setup - There are a total of 8 variants/agents possible - ($\pm Q \pm E \pm T$) Each of these agents were deployed on the Acrobot Environment. Each agent was allowed to run in the environment for 30 times. The number of episodes they took to solve the environment, the scores each episode and the total time taken was stored. The running average of last 100 episodes was used to check if the agent has solved the environment. Although there was no official goal for acrobot, the average score for considering acrobot to be solved was taken to be -85.0 for the sake of the experiment.
Observations The plot below compares the total number of episodes taken for each of the variation to solve the environment (mean and standard deviation is shown with red and green respectively). There was an upper limit (10000 episodes) on the total number of episodes the experiments were run for. The cases in the graph which have no standard deviation show that the variation ran until the upper limit on episodes and still was not able solve the environment for all the 30 runs. We shall call this the standard plot configuration and all the plots present in the rest of the report will have a similar configuration.

Some notable observations -

- Experience Replay - Cases 1,2,5 $\&$ 6 have Experience Replay and the other four cases don't. We can see that the latter diverge. So we can say Experience Replay appears to be vital for convergence for this setup.

- Q-Targets - Out of the cases that converged, comparing cases 1 $\&$ 2 (Q-Targets enabled) with cases 5 $\&$ 6 (corresponding cases with Q-Targets disabled), the former seems to perform better than the latter. This shows that Q-Targets help in convergence for this setup.

- Truncation - The results aren't consistent with Truncation. This can be further observed with results from the following sections as well.

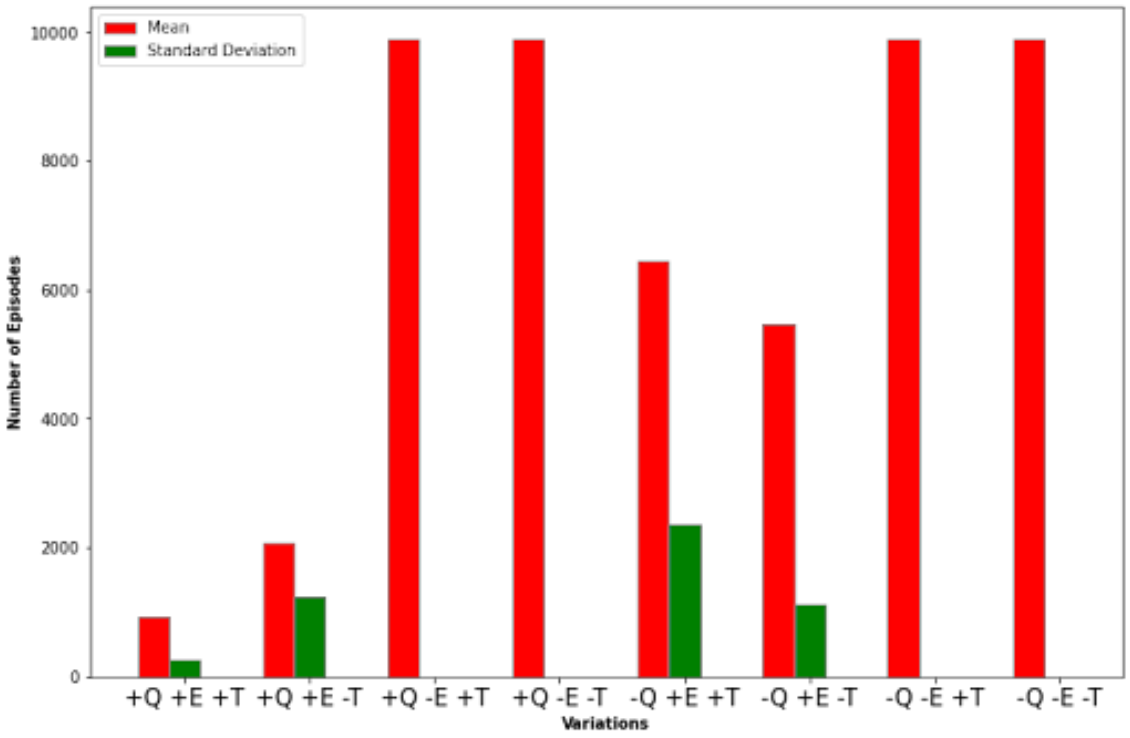


Figure 7: 8 Variations on Acrobot Environment

3.1.3 Varying Q - Targets

Setup - To understand the effect of having Q-Targets better, the following experiment was carried out. We varied the frequency of updating the target network with the main network weights. Only a small subset of frequencies were tried to understand the trend in the initial frequencies. The cases considered below are carried out in the presence of experience replay and truncation similar to the previous experiments with the update frequencies of 1,5,10 and 20 (corresponding to cases Q1,Q2,Q3 and Q4).

Observations - The plot follows the standard configuration as mentioned above. We can see that in the lower frequencies, the agent seems to clearly perform better with increasing frequency.

This plot is one of the smoothest plots in the whole experiment. It is intuitive to expect the performance to deviate and go the other way as the frequency increases to larger numbers.

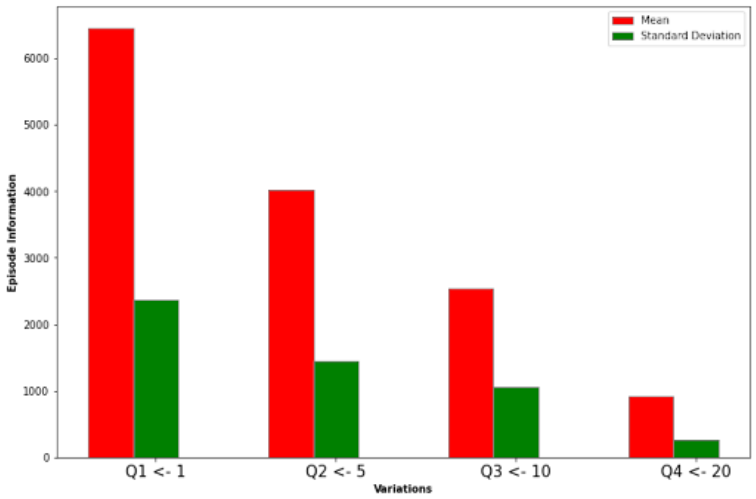


Figure 8: Varying Q-Targets on Acrobot Environment

3.1.4 Varying Experience Replays

Setup - Experience Replay has two main components (Memory of size n1 & Mini-Batch of size n2). The first one being Memory, which refers to the huge database which stores the latest 'n1' number of experiences. The algorithm samples 'n2' number of experiences from memory in random and performs a semi-stochastic gradient descend to train the main network. To understand Experience Replay better, the following experiment was carried out by having various configurations of the Experience Replay. (namely (10k,64),(5k,64),(10k,32),(5k,32) & (1,1) where each tuple (a,b) represents the configuration with a memory of size 'a' and a mini batch of size 'b').

Observations - The plot follows the standard notation as mentioned above.

The following observations can be made from the comparison plot.

- Experience Replay helps in convergence. This is clear from comparing E1, E2, E3 & with E5. The first four have experience replay and the fifth has a configuration that works similar to an experiment with no experience replay. The first four converges and the last one does not.
- The size of mini-batch(n2) is significant. On comparing E1 and E3 against E2 and E4, we can see that more samples for obtaining the averaged out gradient to vary the weights seem to help the algorithm learn better.

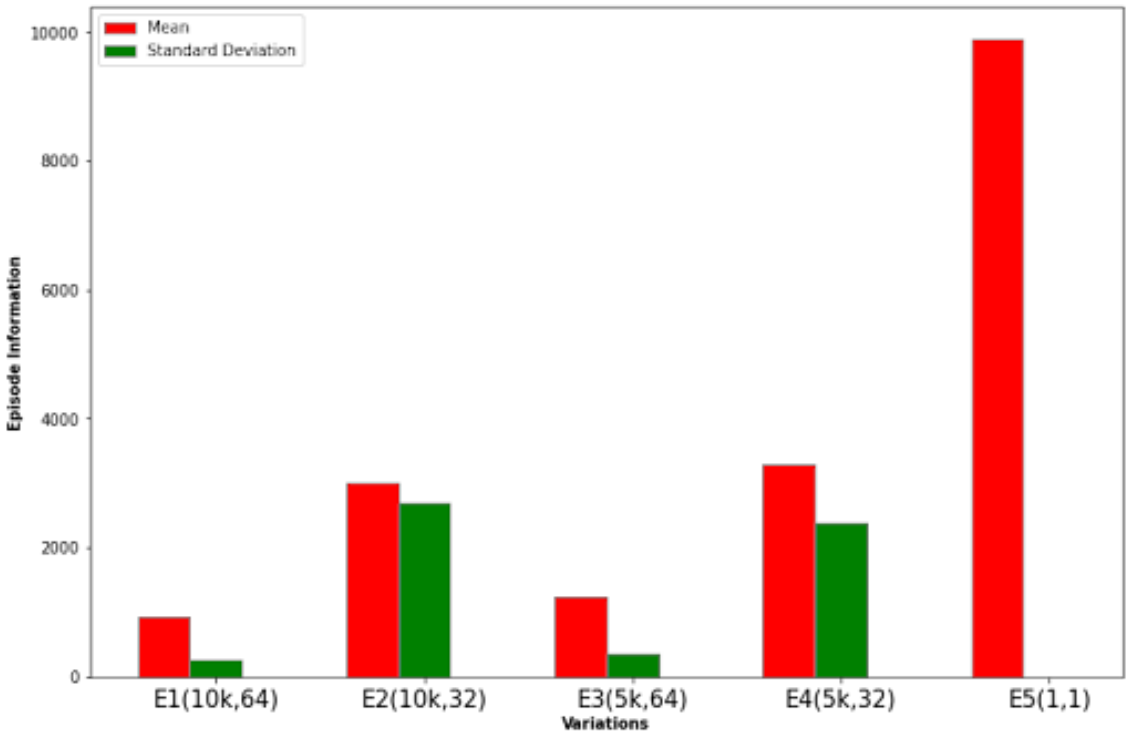


Figure 9: Varying Experience Replay Configurations on Acrobot Environment

- The size of the memory(n1) is not as significant for the following cases. Although a very small number would make a difference, over a certain threshold, having 5k or 10k size seem to not have a big impact on the learning. This can be observed by comparing the results of configuations E1 and E2 against E3 and E4.

3.1.5 Varying Truncation Levels

Setup - Truncation is just a limit on the gradient that is generated, as semi-stochastic gradient descent is performed on the main network with the mini-batch is obtained from the memory. We perform the experiment with different truncation levels (namely T1, T2, T3 and T4 with limits on the gradients as $\pm 1, \pm 5, \pm 10, \pm 20$ respectively).

Observations - The plot follows the standard notation as mentioned above.

Although there seems to be a benign effect of truncation on stability for this setup, the results aren't conclusive in other setups and environments as the report provides below.

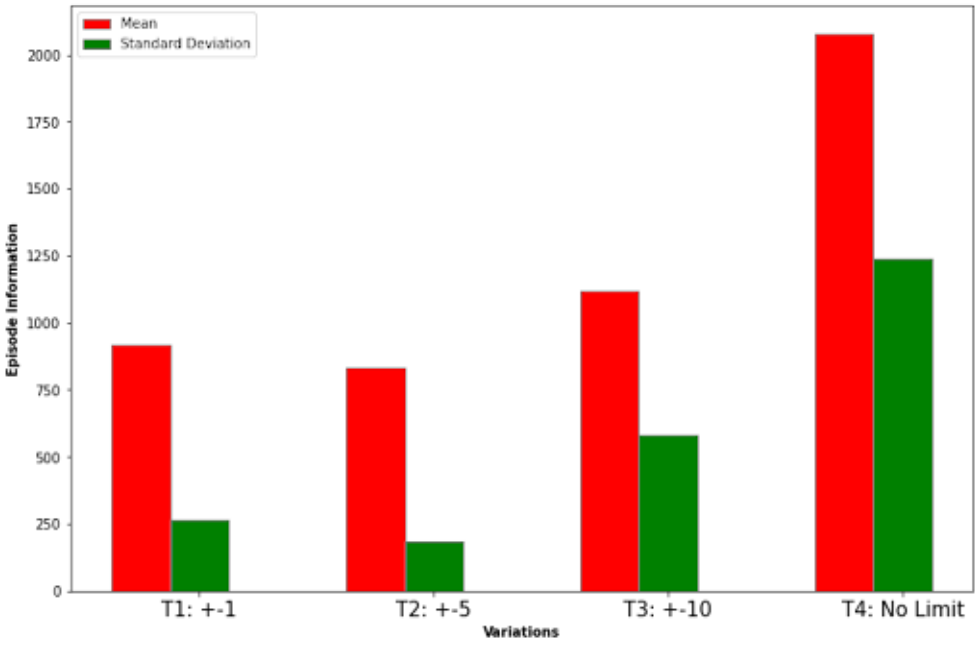


Figure 10: Varying Truncation Levels on Acrobot Environment

3.2 Experiments on MountainCar

3.2.1 MountainCar - Environment

In the Mountain Car environment, the agent tries to drive up a car located inbetween two hills, over the hill on the right. The catch is that the car does not have enough power to go up the hill in a single shot. The car has to go up and down the hills to gain momentum and then pass the hill on the right.

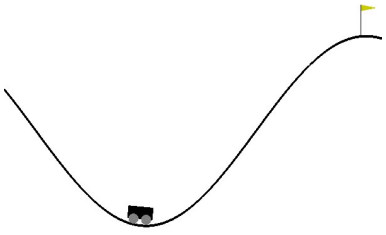


Figure 11: Environment - MountainCar

- The state space is a 2-dimensional vector - Position and Velocity of the car.

- Three possible actions - push left/no push/ push right.
- The agent receives -1 reward for every step until it reaches the top of the right hill.

3.2.2 8 Variations

Setup - Similar to the 8 variations of the algorithm ($\pm Q \pm E \pm T$) experimented with the acrobot environment earlier, we perform the experiment in the exact similar configurations. The input and output layer is configured with 2 and 3 nodes respectively so as to match the state space dimension and the possible actions for the mountaincar environment. Following official documentation, the goal for mountaincar environment was set to be -110.0.

Observations - The plot follows the standard notation as mentioned above.

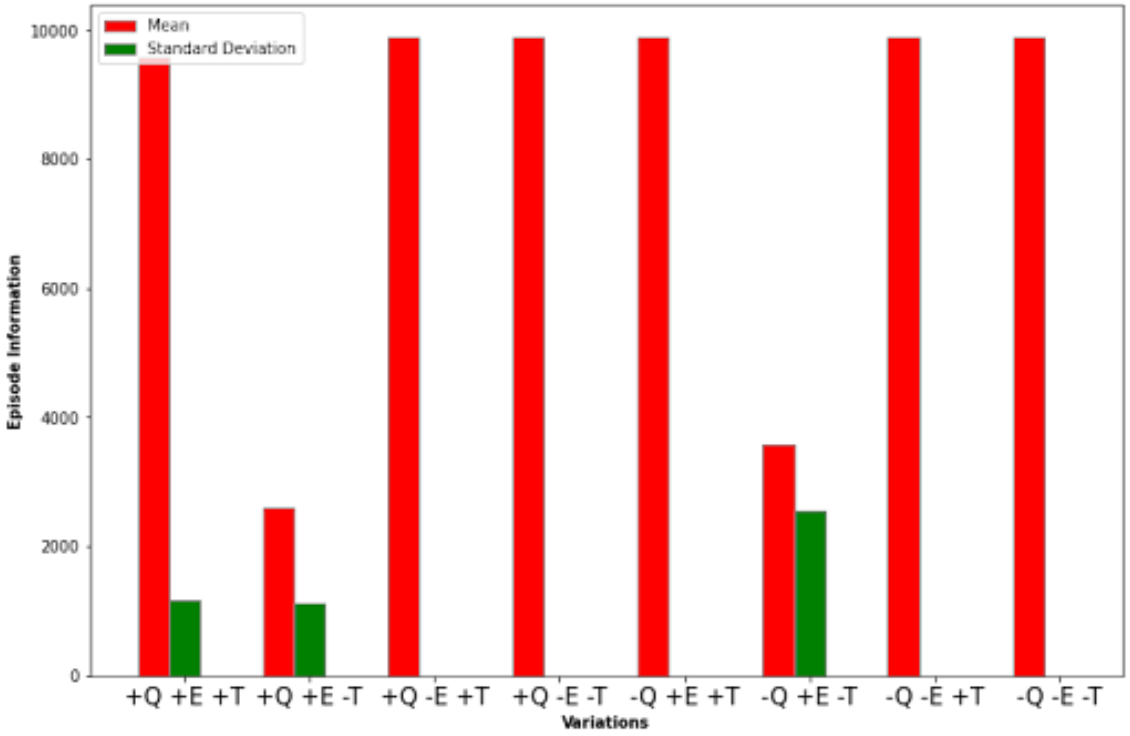


Figure 12: 8 Variations on the Mountaincar Environment

The very first observation is that the graph that we have obtained from Mountaincar environment is evidently different from the results we had obtained from acrobot case. This goes to show how the performance of an algorithm isn't universal and highly depends on the nature and

dynamics of the environment. This becomes even more clear after the inspection of results from Cartpole environment to be discussed later in the report.

Some notable observations -

- Experience Replay - All cases but 2 & 6 diverge. Specifically looking at cases 3,4,7 & 8, we can tell that the network diverges for cases where there is no experience replay. This stays consistent to our previous observations on how Experience Replay is Important for convergence.
- Q-Targets - Out of the cases that converged, comparing cases 1 & 2 (Q-Targets enabled) with cases 5 & 6 (corresponding cases with Q-Targets disabled), the former seems to perform better than the latter. This shows that Q-Targets again help in convergence for this setup as well, similar to our previous observations from cartpole.
- Truncation - Here, truncation seems to hurt the performance of the agent. We can see how the results aren't consistent with Truncation, and sometimes even counter-intuitive. This can be further observed with results from the following sections as well.

3.3 Experiments on CartPole

3.3.1 CartPole - Environment

In the Cart Pole environment, the agent tries to balance a pole on a cart by applying unit force either on right or left direction. The episode terminates when the pole falls for more than 12 degrees from the vertical or if the cart goes out of the frame.

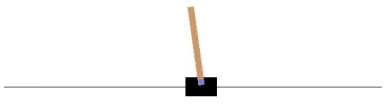


Figure 13: Environment - Cartpole

- The state space is a 4-dimensional vector -Cart position, Cart velocity, Pole angle, Pole angular velocity.
- There are two possible actions - push left or push right
- The agent receives +1 reward for every step until the episode terminates

3.3.2 8 Variations

Setup - Similar to the 8 variations of the algorithm ($\pm Q \pm E \pm T$) experimented with the acrobot environment earlier, we perform the experiment in the exact similar configurations. The input and output layer is configured with 4 and 2 nodes respectively so as to match the state space dimension and the possible actions for the cartpole environment. Following official documentation, the goal for cartpole environment was set to be 195.0

Observations - The plot follows the standard notation as mentioned above.

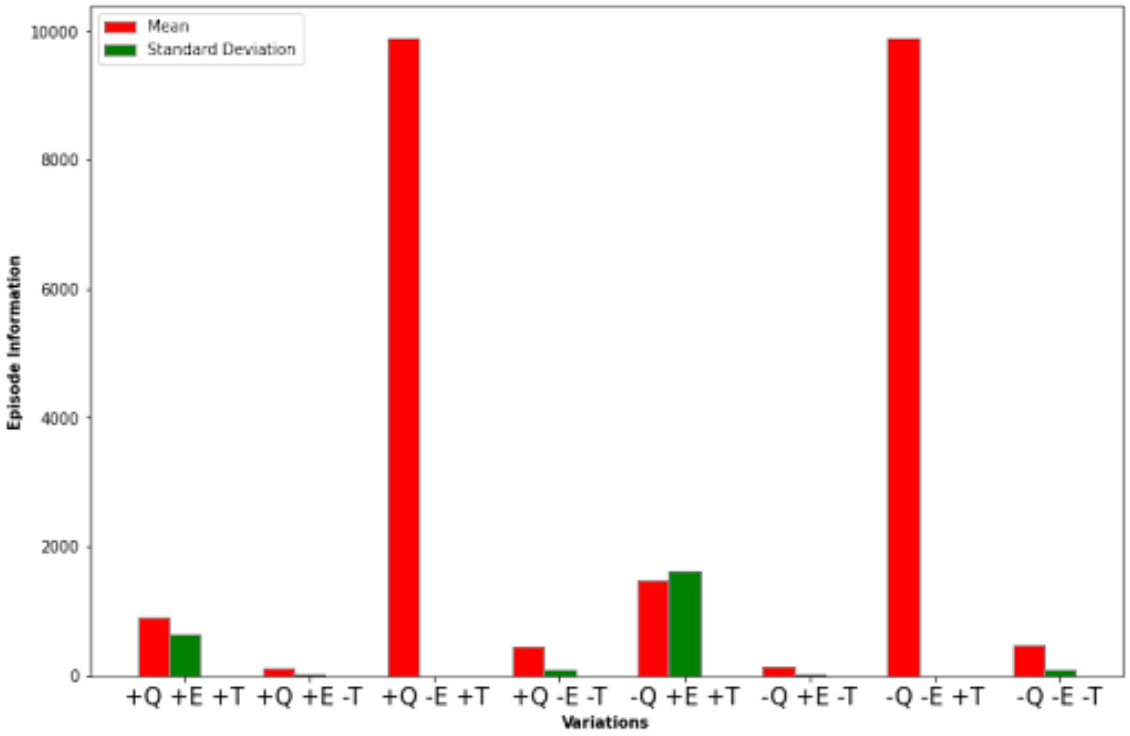


Figure 14: 8 Variations on the Cartpole Environment

The following graph is the most deviant and counter-intuitive of the three graphs. This can

be because of the fact that DQNs performance on CartPole isn't very consistant. Although DQNs solve CartPole with ease, if one plots the scores the agent scores with episodes, it can be seen how shaky the agent learns the environment. It tends to learn very quick and take a sudden turn to perform quite bad in just the span of a few hundred episodes. A single run of each of this variant and the performance graphs (Scores vs episodes) can be found in the appendix. Trying to comment on the trends we've looked at so far and comparing with this graph,

Some notable observations -

- Experience Replay - Comparing case 1 against 3, 2 against 4, 5 against 7 and 6 against 8, it is clear that Experience Replay is important for convergence and helps the agent perform better.
- Q-Targets - Comparing the first four cases with the last four, we can see that the whole graph looks replicated with very little change. Here, Q-Targets seem to not have a huge effect.
- Truncation - Comparing every alternate cases, (ie ones with truncation and the corresponding cases without truncation) we can see that truncation seems to hurt the performance of the agent again.

4 Conclusion and Future Work

Very similar to how we as humans find mastering different tasks takes different amounts of time and how different people find the same task at different difficulties and require different amounts of time to master, similarly our variants also don't have a solid performance record or a fixed graph when tested on different environments. Even for the same environments, different variants seem to have advantages over the other variants in different attributes. Although there is so much ambiguity in trying to figure out exactly how and where each of these proposed features work, some consistent trends were observed across all the environments that were tested on.

- **Experience Replay:** In most cases, Experience Replay turned out to be vital for convergence. Cases where it wasn't absolutely necessary, it did help the convergence and enabled the

agent to learn faster.

- **Q-Targets:** Q-Targets seem to generally help in convergence. On the lower frequencies, it seems that updating the Q-Network less often helps. Although after a limit it is obvious that it will increase again, the tests showed that it decreased steadily till 20 and most probably wouldn't suddenly increase in the vicinity.
- **Truncation:** The performance of Truncation has not been very consistent with environments or variations in the truncation level.

Considering how Truncation is associated more with the stability of the neural network than the RL framework, more experiments of the similar kind with varying neural network architecture can be carried out. Also, the DQN algorithm offers more hyper parameters like the learning rate, epsilon decay, discount factor that can be varied and tested upon various environment to get more interesting insights.

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] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[5] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, USA, 1992. UMI Order No. GAX93-22750.

A Appendix - DQNs with CartPole

A.1 Setup

For the first trial, all possible combinations ($\pm Q, \pm Eand \pm T$) of these techniques was tested on the Cartpole environment to try and look for obvious patterns **(8 variations on the cartpole)**.

A.2 Observations

The plot of the average scores(averaged for 100 episodes) is presented below.

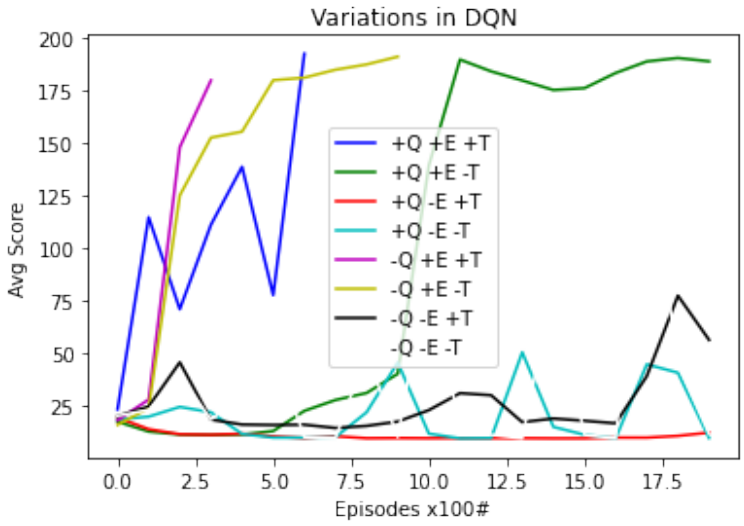
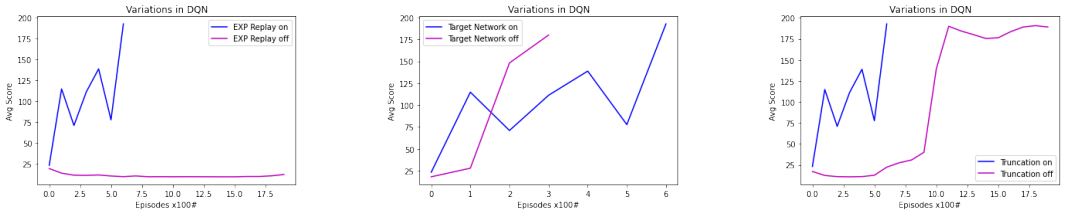


Figure 15: 8 Variations on Cartpole - 1 iteration each



(a) Experience replay present and absent (b) Q-Targets present and absent (c) Truncation present and absent

Figure 16: Comparing performance of DQN on CartPole w.r.t specific features.

It was evident that since the DQN Algorithm has various randomized elements involved, it is difficult to come to a conclusion with just one run of the algorithm. It was also observed to be in-

consistent with the trends and seemed to change every time the experiment was conducted. The graphs also serve as a quick example to explain how DQN learns the CartPole. It can be seen from the avg score how the agent learns and unlearns the environment multiple times before a successful solve. Although DQN was able to solve the CartPole with ease, it was difficult to understand and observe consistent trends and properties of how the DQN learns with this environment.