CSE 4/546 Reinforcement Learning (Fall 2022)

Final Project Report

Team Members:

Faizuddin Mohammed – faizuddi

Ayesha Humaera –ayeshahu

Sai Charan Reddy Duvvuru – sduvvuru

Introduction:

One sort of machine learning is called reinforcement learning, and it includes a software agent making a decision on what action to do in an environment in order to accumulate rewards as quickly as possible. In order to determine the behaviors that result in the highest possible rewards, the learner makes use of approaches based on trial and error rather than procedures that are previously stated. Reinforcement learning is characterized by its distinctive traits, which include a search strategy based on trial and error and delayed rewards. The algorithms for reinforcement learning are evaluated and ranked according to the scores they get when asked to solve a problem that is dependent on an environment. In this study, a comparison is made between the amount of time spent by algorithms on a variety of environments that correlate to the environment. It has been noted that the performance of algorithms changes depending on the environment, but that these variations are equivalent to one another.

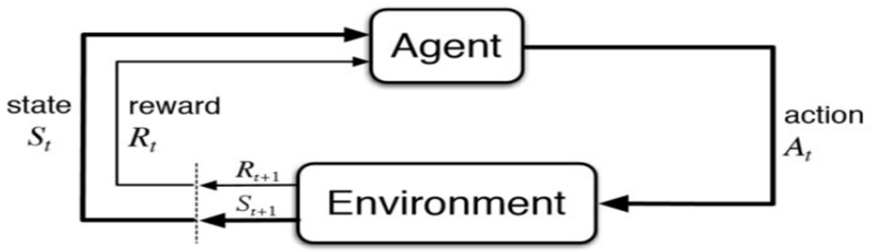


Figure 1: Reinforcement Learning simple architecture

Goal:

In 2013, a researcher at Google's DeepMind published a paper stating that they developed a system that uses Deep Reinforcement learning algorithms to play Atari games directly from pixels. It was possible to train the system using just the pixel data from an image or frame shown on the video game's screen as its input, therefore there was no need to explicitly design any game-specific rules or game-specific information.

For our final project we have compared 4 deep reinforcement learning algorithms and gave the result that which algorithm according to us works the best in the pong environment.

Environment:

Pong is a two-dimensional sports game that simulates table tennis. An in-game paddle is moved vertically across the left or right side of the screen by the user. They can compete with a player in charge of a second paddle on the adversary's side. Players strike a ball back and forth using the paddles. When one fails to pass the ball to the other, they score a point, and the objective is for each player to reach eleven points before the other.

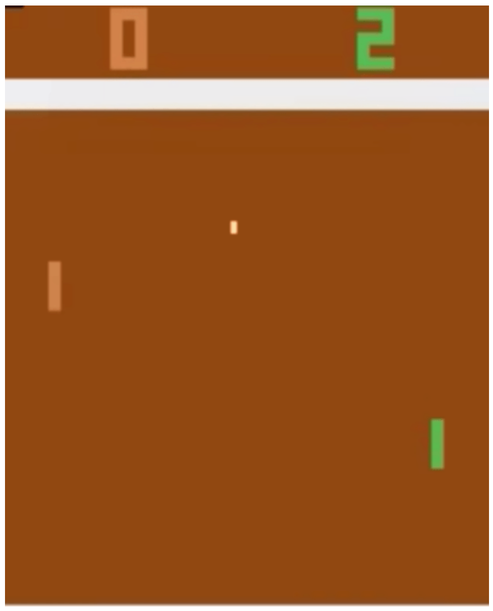


Figure 2: Pong Environment

Environment used: 'PongNoFrameskip-v4'

Action Space: 6 - {0, 1, 2, 3, 4, 5, 6}

> **{0, 1} - Do Nothing**
> **{2, 4} - Go Up**
> **{3, 5} - Go Down**

The list of algorithms we trained the environment on is:

· DQN

· DDQN

· Actor Critic

· Proximal Policy Optimization

Methodology:

We will be explaining how each algorithm described above works on the pong environment:

Proximal Policy Optimization:

Recent developments in the area of reinforcement learning have led to the development of a technique known as proximal policy optimization (PPO), which is an improvement on trust region policy optimization (TRPO). In 2017, this method was presented, and when it was implemented by OpenAI, it demonstrated exceptional performance.

The mathematical equation of PPO is shown below:

$$L^{CLIP}(\theta) = \bar{E}_t[min(r_t(\theta)\bar{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\bar{A}_t)]$$

Figure 3 : Proximal Policy Optimization

The following lines explain how the policy proximal optimization work:

- PPO is a policy gradient optimization algorithm, basically, in each step of the algorithm there is an update to the existing policy to seek improvement on the selected parameters
- The update will not be too large. It achieves this by effectively "clipping" the update area to a very restricted range, which guarantees that the update is not too extensive and that the old policy is not too different from the new policy.
- The difference between the future discounted sum of rewards on a certain state and action and the value function of that policy is called the **advantage function**.
- The importance sampling ratio is used for update
- And then epsilon is the hyperparameter which denotes the limit of the range within which the update is allowed

The pseudocode for implementing the PPO algorithm using the Actor critic style:

```
$Input:\ initial\ policy\ paramters$
$for\ iteration = 1, 2, ...do$
$for\ actor = 1, 2, ....do$
$Run\ policy\ \pi_{\theta_{old}}\ in\ environment\ for\ T\ timesteps$
$Compute\ Advantage\ estimates\ \bar{A}_1, ..., \bar{A}_t$
$end\ for$
$Optimize\ surrogate\ L\ wrt\ \theta,\ with\ K\ epochs\ and\ minibatch\ size\ M <= NT$
$\theta_{old} \leftarrow \theta$
$end\ for$
```

Figure 4: Pseudocode for PPO

Observation:

Applying PPO on pong took a long time to converge(approximately 522 episodes and each episode had 9 timesteps) we see that for at least 60-70 episodes the actions were random and there was no increase in the reward and then the agent begins understanding what to do and the average mean reward starts decreasing from -21 we trained it for 522 episodes and got a mean reward of 16

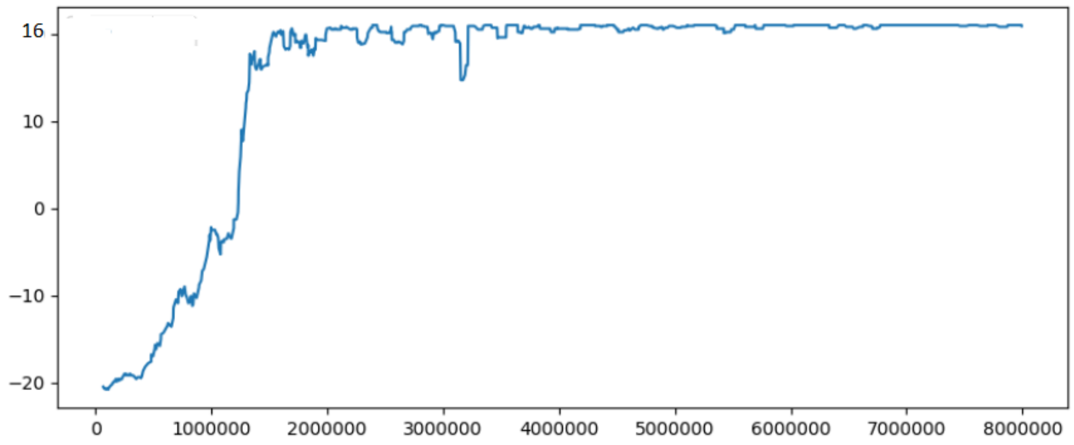


Figure 5: PPO on Pong

In the graph we see that a few times the reward goes down but it doesn't fluctuate very far away from the original policy as we clip the update function.

<u>Actor Critic:</u>

A Temporal Difference(TD) variation of Policy gradient is Actor-Critic. It has the Actor and Critic networks. The actor chose the appropriate course of action, and the critic told him or her how effective it was and what needed to change. The policy gradient technique is the foundation for the actor's learning. Comparatively, reviewers assess the actor's performance by computing the value function.

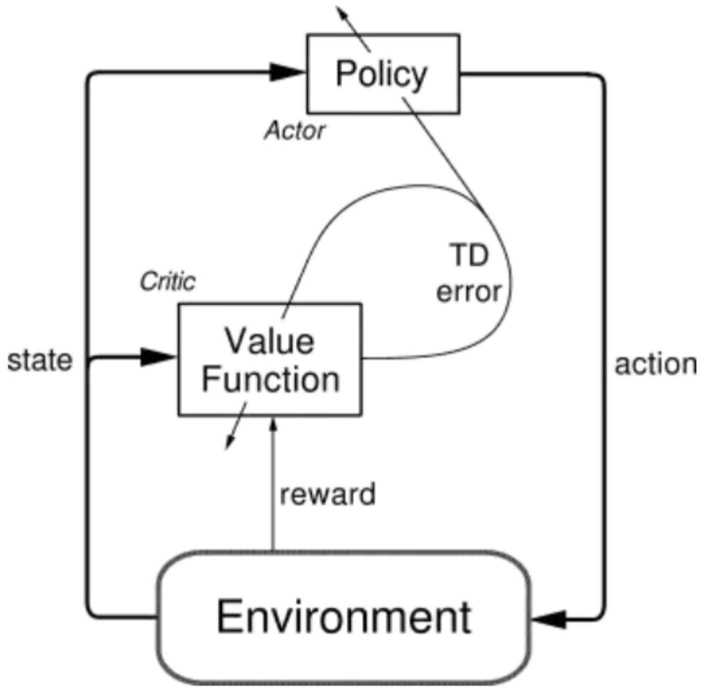


Figure 6: Actor Critic

We can use a critic to estimate the action-value function:

$$Q_w(s, a) \approx Q_{\pi_\theta}(s, a)$$

Actor-critic algorithms follow an approximate policy gradient

$$\nabla_\theta J(\theta) \approx E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)]$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$$

Observation:

Applying Actor-Critic on pong took a long time to converge(approximately 70k+ rounds) . We see that after 70k+ rounds the running mean is around -10. As Actor Critic takes a long time to converge and there is no clipping to the update policy there will be high variance to the update policy. So after a certain time the rewards might increase or they might fall down in value.

```
[<matplotlib.lines.Line2D at 0x7f15a1dfa1c0>]
```

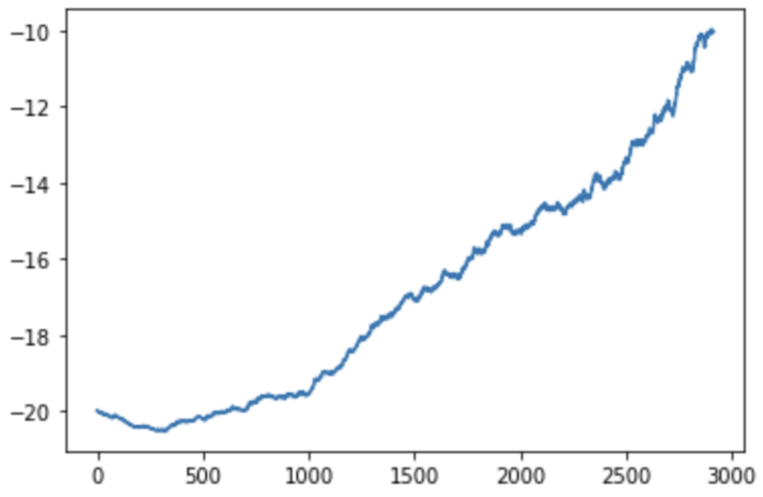


Figure: Actor Critic on Pong

<u>Deep Q-Network:</u>

In the context of a Q-Learning framework, an approximation of a state-value function using a neural network is referred to as a Deep Q-Network (DQN). In the instance of Atari Games, the inputs consist of numerous frames of the game, while the outputs consist of state values for each action. In combination with Experience Replay, it is used for the purpose of keeping the episode stages in memory for off-policy learning. During this kind of learning, samples are arbitrarily taken from the replay memory. In addition, the Q-Network is often optimized in the direction of a static target network, which is then updated with the most recent weights at regular intervals of k steps (where k is a hyperparameter). The latter reduces the potential for short-term oscillations during training by eliminating the influence of a moving object. The first method addresses the issue of autocorrelation that may arise from on-line learning, and the inclusion of a replay memory transforms the issue into one that is more analogous to a supervised learning issue.

The mathematical equation for DQN is:

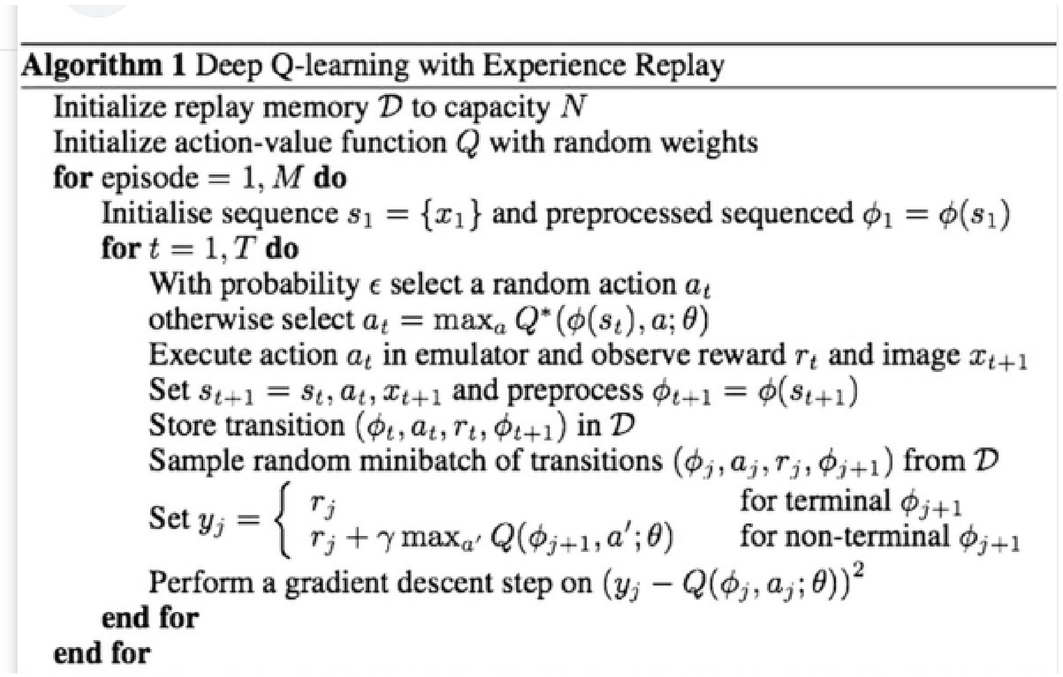
$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

Figure 7: DQN

Below we explain how DQN works:

- Experience Replay gathers a training sample by interacting with the environment and saves it as training data.
- We then take a random batch of training data from the experience Replay as to avoid overfitting and this random batch becomes input to the target network as well as the Q network.
- When training the Q Network, the Predicted Q Value, Target Q Value, and the observed reward from the data sample are all inputs into the equation used to calculate the Loss.
- The Q network is trained again for every iteration however the Target network remains fixed
- Every T Steps, we copy Q network weights to the target network.

The advantage of using DQN is that the performance is relatively stable when a bad gradient is estimated. But then DQN fails if the Q function aka reward function is too complex to be learnt



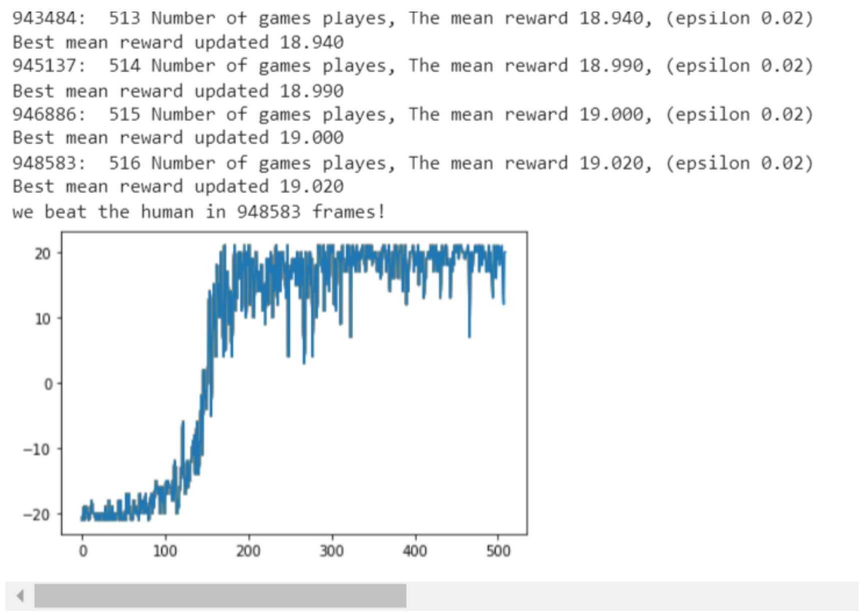
Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in 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 transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

Figure 8: Algorithm for DQN

Observation:

From the Graph below we see that the performance of DQN steadily increases and by the episode 516 the reward is 21 which means that the agent wins against the AI 21-0 times



```
943484:  513 Number of games playes, The mean reward 18.940, (epsilon 0.02)
Best mean reward updated 18.940
945137:  514 Number of games playes, The mean reward 18.990, (epsilon 0.02)
Best mean reward updated 18.990
946886:  515 Number of games playes, The mean reward 19.000, (epsilon 0.02)
Best mean reward updated 19.000
948583:  516 Number of games playes, The mean reward 19.020, (epsilon 0.02)
Best mean reward updated 19.020
we beat the human in 948583 frames!
```

Figure 9 : DQN on Pong

<u>Double Deep Q-Network:</u>

Double DQN takes its cue from Double Q-Learning which employs two distinct Deep Neural Networks, Deep Q Network (DQN) and the Target Network. Since it will be applied during the optimization stage of updating the Deep Q Network parameters, it should be noted that there is no learning rate when updating the Q-values. After the main neural network selects the optimal next action from all the alternative actions, the target neural network examines this action to determine its Q-value. This straightforward technique has been demonstrated to lower overestimations, which improves the final policy.

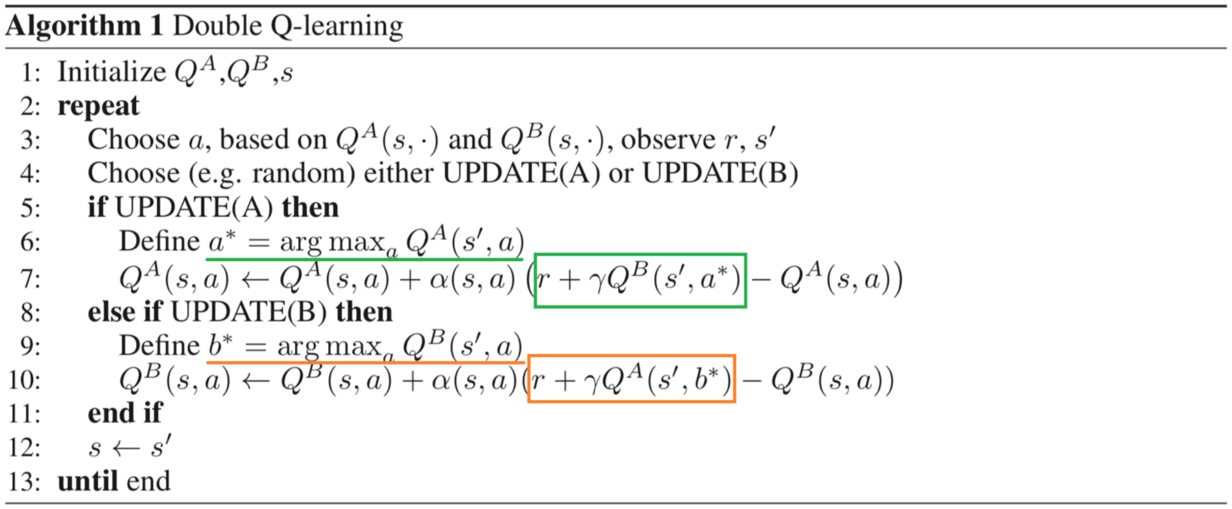
$$Q_{qnet}(s_t, a_t) \leftarrow Q_{qnet}(s_t, a_t) + \alpha(R_{t+1} + \gamma \underbrace{\boxed{Q_{tnet}(s_{t+1}, \overset{\text{Choose action by DQN network}}{\boxed{a}})}}_{\text{Calculate estimated Q-value with action from QNET by using Target Network}} - Q_{qnet}(s_t, a_t))$$

Figure 10: Double DQN Mathematical Representation

Working Procedure:

- Deep Q Network chooses the best course of action with the highest Q-value for the upcoming state.
- Target Network uses the above selected action a, computing the estimated Q-value.
- According to the Target Network's projected Q-value, update the Deep Q Network's Q-value.

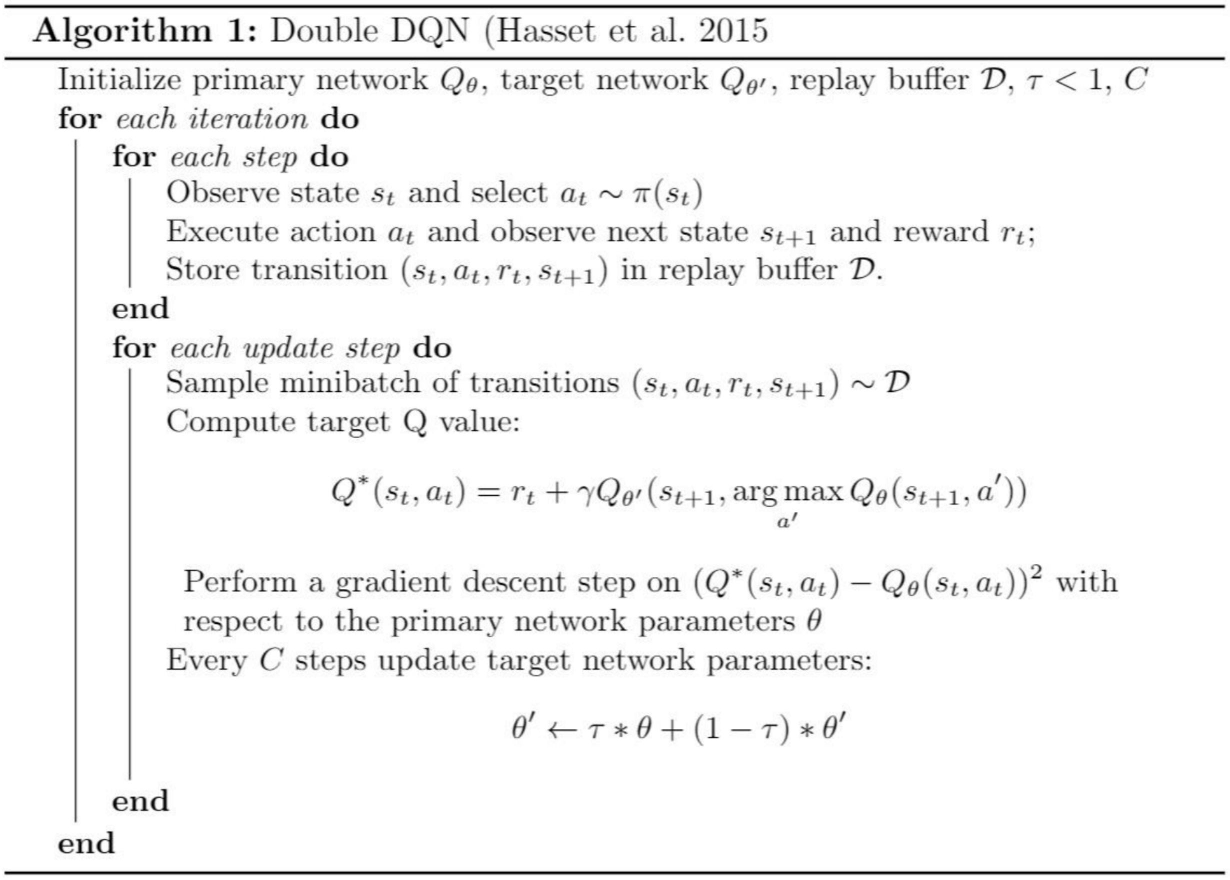
- Update the Target Network's parameters based on the Deep Q Network's parameters throughout numerous iterations.
- Deep Q Network settings should be updated based on Adam optimizer.



Algorithm 1 Double Q-learning

```
1:  Initialize $Q^A$,$Q^B$,$s$
2:  repeat
3:    Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s'$
4:    Choose (e.g. random) either UPDATE(A) or UPDATE(B)
5:    if UPDATE(A) then
6:      Define $\underline{a^* = \arg\max_a Q^A(s',a)}$
7:      $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\big(\boxed{r + \gamma Q^B(s',a^*)} - Q^A(s,a)\big)$
8:    else if UPDATE(B) then
9:      Define $\underline{b^* = \arg\max_a Q^B(s',a)}$
10:     $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)\big(\boxed{r + \gamma Q^A(s',b^*)} - Q^B(s,a)\big)$
11:   end if
12:   $s \leftarrow s'$
13: until end
```

Figure 11: Algorithm for Double DQN



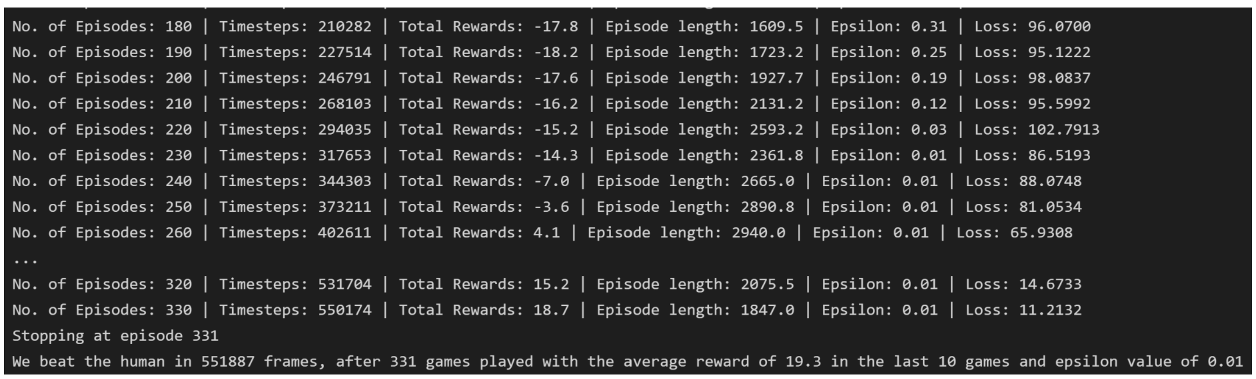
Algorithm 1: Double DQN (Hasset et al. 2015

```
Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau < 1$, $C$
for each iteration do
    for each step do
        Observe state $s_t$ and select $a_t \sim \pi(s_t)$
        Execute action $a_t$ and observe next state $s_{t+1}$ and reward $r_t$;
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$.
    end
    for each update step do
        Sample minibatch of transitions $(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
        Compute target Q value:
            $Q^*(s_t,a_t) = r_t + \gamma Q_{\theta'}(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1},a'))$
        Perform a gradient descent step on $(Q^*(s_t,a_t) - Q_\theta(s_t,a_t))^2$ with
        respect to the primary network parameters $\theta$
        Every $C$ steps update target network parameters:
            $\theta' \leftarrow \tau * \theta + (1-\tau) * \theta'$
    end
end
```

Figure 12: Pseudocode for Double DQN

Observation:

From the below output, we can see that the performance of Double-DQN steadily increases after certain episodes and the mean reward has finally reached 19. The agent took 551887 timesteps to reach the mean reward requirement after playing 331 games.



```
No. of Episodes: 180 | Timesteps: 210282 | Total Rewards: -17.8 | Episode length: 1609.5 | Epsilon: 0.31 | Loss: 96.0700
No. of Episodes: 190 | Timesteps: 227514 | Total Rewards: -18.2 | Episode length: 1723.2 | Epsilon: 0.25 | Loss: 95.1222
No. of Episodes: 200 | Timesteps: 246791 | Total Rewards: -17.6 | Episode length: 1927.7 | Epsilon: 0.19 | Loss: 98.0837
No. of Episodes: 210 | Timesteps: 268103 | Total Rewards: -16.2 | Episode length: 2131.2 | Epsilon: 0.12 | Loss: 95.5992
No. of Episodes: 220 | Timesteps: 294035 | Total Rewards: -15.2 | Episode length: 2593.2 | Epsilon: 0.03 | Loss: 102.7913
No. of Episodes: 230 | Timesteps: 317653 | Total Rewards: -14.3 | Episode length: 2361.8 | Epsilon: 0.01 | Loss: 86.5193
No. of Episodes: 240 | Timesteps: 344303 | Total Rewards: -7.0 | Episode length: 2665.0 | Epsilon: 0.01 | Loss: 88.0748
No. of Episodes: 250 | Timesteps: 373211 | Total Rewards: -3.6 | Episode length: 2890.8 | Epsilon: 0.01 | Loss: 81.0534
No. of Episodes: 260 | Timesteps: 402611 | Total Rewards: 4.1 | Episode length: 2940.0 | Epsilon: 0.01 | Loss: 65.9308
...
No. of Episodes: 320 | Timesteps: 531704 | Total Rewards: 15.2 | Episode length: 2075.5 | Epsilon: 0.01 | Loss: 14.6733
No. of Episodes: 330 | Timesteps: 550174 | Total Rewards: 18.7 | Episode length: 1847.0 | Epsilon: 0.01 | Loss: 11.2132
Stopping at episode 331
We beat the human in 551887 frames, after 331 games played with the average reward of 19.3 in the last 10 games and epsilon value of 0.01
```

Figure 13: Output for Double DQN on Pong

The below graph is plotted for the number of episodes vs the length of episodes. The length of an episode is declared by the number of timesteps taken between consecutive episodes (range 10).

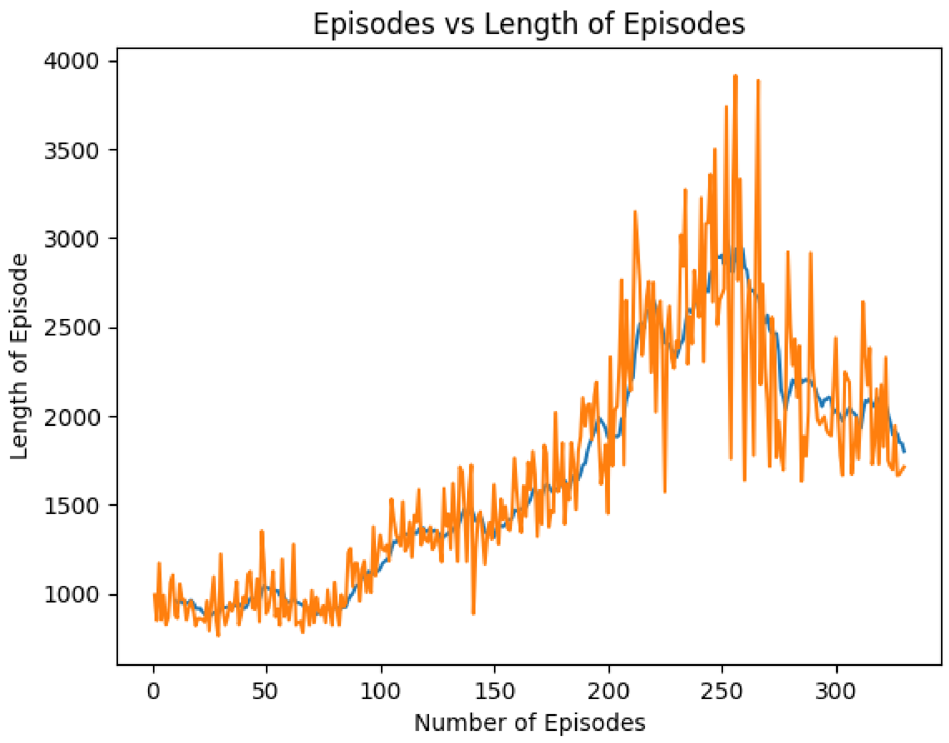


Figure 14: Number of Episodes vs Length of Episodes

The below graph is plotted for the number of episodes vs the rewards per episode. We train the agent till we get a mean reward of 19 for at least 10 episodes.

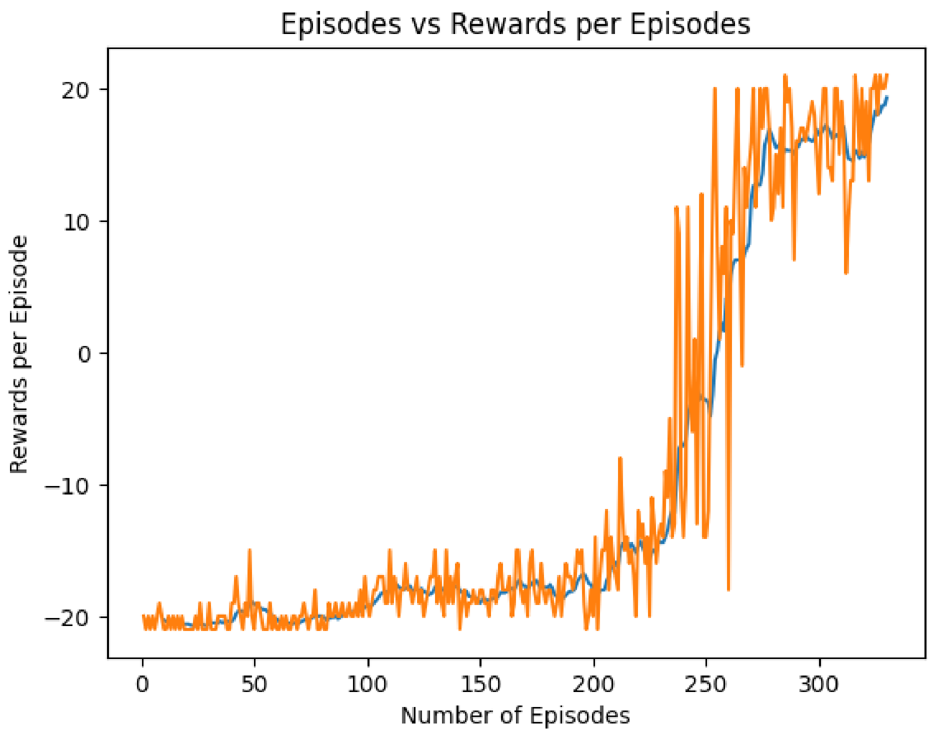


Figure 15: Number of Episodes vs Rewards per Episodes

Contribution:

1	**Faizuddin Mohammed**	**Proximal Policy Optimization Actor Critic Report**	**40%**
2	**Sai Charan Reddy Duvvuru**	**Double Deep Q network Report**	**30%**
3	**Ayesha Humaera**	**DQN Report**	**30%**

References:

[1] A Brief Introduction to Proximal Policy Optimization - GeeksforGeeks

[2] Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.13.0+cu117 documentation

[3] DQN Explained | Papers With Code

[4] udacity-deep-reinforcement-learning/pong-PPO.ipynb at master · handol-park/udacity-deep-reinforcement-learning (github.com)