# Using Deep Q network and Double Deep Q Network to Play Atari Game "Pong"

## INTRODUCTION

In this project, I have tried to train an agent to play Atari 2600 Game "Pong" using the concepts of Deep Reinforcement Learning. I first implemented a Naïve approach of using just one Network for choosing the optimal action and also estimating the action values. After that, I implemented Deep Q learning with experience replay and Double Deep Q Network algorithms with experience replay as the described in following 2 papers – "Human-Level Control through Deep Reinforcement Learning (2015)" [1] and "Deep Reinforcement Learning with Double Q-learning (2015)" [2]. All in all, I implemented the two aforementioned papers to effectively train an agent to Pong on Atari 2600 console and compare the results.

### Summary of Papers

**Human-Level Control through Deep Reinforcement Learning(2015)-** [1]

Reinforcement Learning (RL) has made tremendous strides in recent years. However, the traditional approaches of learning such as Q Learning, Sarsa, etc. where we can initialize a Q table to store the value of every state-action pair, are limited to environments with low dimensional state spaces. When we have a high dimensional state space environment like Classic Atari 2600 games, it becomes practically impossible to create a Q table. So how do we deal with such environments- enter Deep Q networks. The authors of the paper provide a novel artificial agent Deep Q Network, which as the name suggests uses deep neural networks instead of a Q table to approximate the optimal action values to play the Atari games better than humans.

The evolution of deep learning due to advancements in GPU and computing power has changed the dynamics of artificial intelligence. In supervised learning, neural networks have been extensively used which are nothing but layers of neurons with weights and bias arranged in a fashion such that input is fed into the network and a prediction or an approximate value is received. This approximate value is then compared with the true value and then the loss is calculated. Finally, the weights and biases in the network are adjusted by backpropagation to reduce the overall loss. A deep neural network like convolutional neural networks, recurrent neural networks, and GAN have several layers of neurons stacked in a linear fashion that provide depth to the network. The depth of the network helps in extracting more abstract representations of the data to perform complex tasks like object categorization directly from raw sensory data. The authors use one particularly successful architecture, the deep convolutional network, which uses hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields—inspired by Hubel and Wiesel's seminal work on feedforward processing in the early visual cortex18—thereby exploiting the local spatial correlations present in images, and building in robustness to natural transformations such as changes of viewpoint or scale.

In contrast to supervised learning, which focuses on comparing the true label or the ground truth with the prediction, RL is based on learning from the environment. In a typical problem, an agent interacts with an environment through a sequence of observations, actions, and rewards. The goal of the agent is to take action in a fashion that maximizes the total rewards. In terms of the bellmen equation

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s,\ a_t = a,\ \pi\right],$$

where Q*(s, a) is the maximum sum of rewards from state s, discounted by $\gamma$ at each time step t, achievable by a behavior policy pi, after making an observation (s) and taking an action (a).

In the case of the Atari emulator, at each time step, the agent selects an action from the set of legal game actions. The action is then passed to the emulator and modifies its internal state and the game score. The emulator's internal state is not observed by the agent; instead, the agent observes an image xt [Rd from the emulator, which is a vector of pixel values representing the current screen. In addition, it receives a reward rt representing the change in the game score. Because the agent only observes the current screen, the task is partially observed and many emulator states are perceptually aliased (that is, it is impossible to fully understand the current situation from only the current screen xt ). Therefore, sequences of actions and observations, st~x1, a1, x2,:::,at{1,xt, are input to the algorithm, which then learns game strategies depending upon these sequences

The deep neural network takes in the state and using its current weights ∅ , predicts an approximate value of Qapprox for all the actions possible from that state.

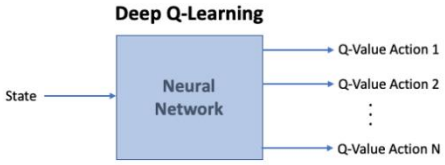


Figure 1
Source: https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af

This value is used by the agent to select the appropriate action using e-greedy policy and to move to a new state s'.
Now just like in every supervised problem, the neural networks learn by comparing the prediction with the true value to calculate the loss. In this case, instead of a predefined true value, an approximate target value is calculated using the Bellmen's equation

$$Qtarget = r + \gamma \max_a Q^*(s',a')$$

*where Q*(s', a') is the maximum action value of the next step*

Hence, the loss of the function is given by

$$Loss = (Qapprox - Qtarget)^2$$

To reduce the loss, the deep neural network performs gradient descent i.e differentiates the loss function w.r.t weights ∅ , to find the minimum value of the loss function.

There are two major challenges in the naïve Q learning approach. First is the fact that the deep neural network is learning in sequential order from consecutive samples. These samples are highly correlated with each other which leads to inefficiency. The training samples tend to be skewed in the direction of the maximum next action, making the parameters stuck in a poor local minimum, or even diverge catastrophically.

*Experience Replay*
This is solved using Experience Replay. The authors use the technique of storing tuples of experiences ($s_t$, $a_t$,$r_{t+1}$,$s_{t+1}$)in dataset D, pooled after the end of every episode. These experiences are then sampled randomly to train the neural network. That is, instead of sending the current state, actions, reward, and next state of the environment we instead use random samples from the experience memory and train the network. Randomizing the samples breaks these correlations and therefore reduces the variance of the updates.

*Target Network*
Second is the use of a single network to approximate the Q values and calculate the target Q value. As we know the Bellman equation we can get Q(s, a) using the maximum value of Q(s', a'). However, Q(s', a') is calculated using the same network Q, where the weights change in the next iteration, hence we are chasing a moving target. This problem is solved by using a target network. We use a different network $Q^{\wedge}$ which is used to generate the target Q values for Q learning updates. Periodically the Q network is cloned to Q^. This makes the algorithm much more stable.

**Deep Reinforcement Learning with Double Q-learning** [2]

This paper was an extension of the previous paper highlighting the issues of overestimation in Q Learning updates. The basis of the paper is the limitation of q learning in function approximation such that the agent always chooses the optimal action based on the maximum Q value.

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-) .$$

Where Rt+1 is the reward received in St+1 by taking a  and  – are the weights of the target network used to calculate the Q value. The maximum Q values are approximated using the neural network hence don't present the full picture to the agent. In a way, the agent is choosing the non-optimal action in a state, just because it has the maximum Q value. Often, the optimal actions would have low Q values as compared to non-optimal actions in a given state, but they would be ignored and the maximum value will be chosen. This is observed in the initial stages of the training when the network has very little idea about the environment. However, the consistent selection of non-optimal action leads to an overestimation of action values for the future. This will cause a certain amount of noise in the Q values making the learning process messy.

For instance, let's say the true value for every action at a given state is zero. The estimated values will be a distribution above and below zero. Taking a maximum of these estimated values which will be above zero, to calculate the Q values will lead to overestimation.

The solution to the problem is Double Q Learning. In original double Q learning, we use 2 independent Q estimates Q^(A) and Q^(B). With a probability of 0.5, we use Q^(A) to determine the maximizing action but use it to update Q^(B). Similarly, with 0.5 probability we use Q^(B) to choose the maximum action and use it to update Q^(A).

In the paper, the author uses the basic concepts of Double Q Learning and implements them using the neural network function approximations to create an agent Double Deep Q Network(DDQN). They use two models Q and Q^, where one is used for action selection and the other is used for action evaluation.

The estimated q value Qpred is calculated using the Q network. The estimated q value for the next state Qnext is calculated using the Q^ network. Now instead of choosing the maximum action based on Qnext, we calculate the Qeval for the next state using the Q network and choose the maximum action from that. The maximum action is then used to calculate the $Q_{target}$ value using the bellman's equation

$$Qtarget = r + \gamma \max_a Q^*(s', a')$$

Hence, there is a minor change in the original DQN for the calculation of Target Q values.

DQN

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

DDQN

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)\,.$$

The overall network architecture, implementation, preprocessing and concept of replay buffer remain the same in DDQN.

**METHOD**

The code for the implementation of the two papers is taken from the repository of Phil Tabor.
https://github.com/philtabor

https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code/tree/master/DQN

https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code/tree/master/DDQN

*The implementation of pre-processing is done using the Open AI Gym wrappers.*
***https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter06/lib/wrappers.py***

The hyperparameters used for the project are mentioned in Table 1.

**Environment Details**

The environment for this project is the Pong game which is a part of 49 games from the Atari 2600 console that were used for implementation in the original paper. It is a table tennis-themed arcade video game with simple two-dimensional graphics. Just like in table tennis, in Pong, a player gets a point on winning the rally against another and the game gets over when one of the players reaches 21 points.

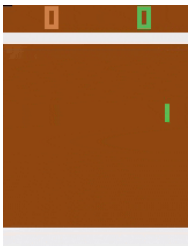

*Figure 2*

*Source:https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af*

Action Space: The action space consists of 6 discrete actions as mentioned below. However, 3 of the actions are redundant Noop is the same as fire, Right is the same as right fire, and Left is the same as Leftfire.

**['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']**

State Space: **(210, 160, 3)**
The state space is defined by the current image of the screen with a size of 210*160*3, where 210 is the height, 160 is the width and 3 corresponds to the three channels, RGB in this case. However, several pre-processing steps and modifications are done to get more precise information from the screen images. That will be covered in the pre-processing section.

Reward function: The reward function is straightforward. The agent gets +1 for every point missed by the opponent, -1 for every point lost by itself, and 0 otherwise.

Episode: The entire game of 21 points, that is whenever the agent or its opponent reaches 21 points, the game is won and the episode terminates

**Pre-processing**

There are several challenges when it comes to using the original state space of the environment. I will be handling these issues using the pre-processing steps mentioned in the paper.

Flickering- First, to encode a single frame we take the maximum value for each pixel color value over the frame being encoded and the previous frame. This was necessary to remove flickering that is present in games where some objects appear only in even frames while other objects appear only in odd frames, an artifact caused by the limited number of sprites Atari 2600 can display at once.

High Dimensions- The state space is defined as a vector of size (210,160,3). Using an image of size 210*160 pixels, directly with neural would be complex. It would require a complex neural network with many layers and high computing power and GPU. To reduce the complexity, we use dimensionality reduction and convert it into grayscale and then scale it down to size 84*84. Now the vector looks like this (84,84,1).

Single Frame-Another issue is that we only observe the fixed image of the screen at a particular time, which doesn't give the full picture. The network won't understand whether the ball is moving upwards or downwards. Hence, it is essential to present a series of images that would present the real dynamics of the game. This is achieved using stacking n consecutive frames together, in our case n=4 and then that entire sequence is treated as one state.

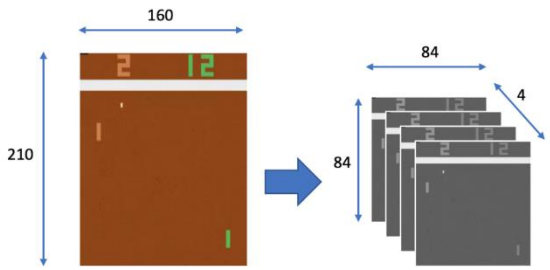


*Figure 3*

*Source: https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af*

**Network Architecture**

When it comes to networks, the same network mentioned in the paper will be used for all the 3 algorithms. The network consists of 3 convolutional layers and 2 fully connected layers. The input is first processed by 3 convolutional layers which help in understanding the spatial relationships between different pixels of the images. Since the input to the layer is (84,84,4) with four images stacked together, the Convolutional Layers also understand the temporal properties across the frames.

The first conv layer has 32 output filters and a convolutional kernel of size 8*8 with a stride of 4. The second layer consists of 64 input filters, 64 output filters, with a convolutional kernel of size 4*4 with a stride of 2. The third layer consists of 64 input filters, 64 output filters, kernel size of 3*3, and stride of 1. The convolutional layers are all activated with a relu function (Rectified Linear Unit). The output will have dimensions as (64, H, W) where H and W are the height and width of the convolved image after the third layer.

The first fully connected layer receives input of size which is calculated by flattening the output from the third convolutional layer(64*H*W). The output filters for the first fully connected layer are 512. The second fully connected layer has 512 input features and several actions, 6 in our case as the output filters. The fully connected layers are activated by the linear function.

```
self.conv1 = nn.Conv2d(input_dims[0], 32, 8, stride=4)

self.conv2 = nn.Conv2d(32, 64, 4, stride=2)
self.conv3 = nn.Conv2d(64, 64, 3, stride=1)

fc_input_dims = self.calculate_conv_output_dims(input_dims)

self.fc1 = nn.Linear(fc_input_dims, 512)
self.fc2 = nn.Linear(512, n_actions)
```

As mentioned in the paper, the optimizer and loss chosen for the training are RMS prop and Mean Squared Error loss respectively.

```
self.optimizer = optim.RMSprop(self.parameters(), lr=lr)

self.loss = nn.MSELoss()
```

**Replay Buffer**

The concept of experience replays as mentioned above was implemented using a replay buffer. The experience tuples ($s_t$, $a_t$,$r_{t+1}$,$s_{t+1}$) are stored in this buffer and they are sampled randomly after a period to train the network. The idea was to use a queue-like data structure with a limited size, that would store the new experience tuples in the front and would keep popping out the old experiences from the back. We also define a batch size, such that the network is fed many samples at once instead of just one sample and hence expediting the training process. I chose a batch size of 32 to train the network. Hence the input dimensions for the network would be 32*84*84*4.

**Algorithms Used**

1) **Naïve Algorithm**

Initialize action-value function Q with random weights
For episode =1, N do
    Initialize sequence $s_1=\{x_1\}$ and pre-processed sequence $\omega = \omega(s_1)$
    For t=1, T, do
        with probability E, select a random action $a_t$
        otherwise select at= argmaxQ($\omega(s_1)$, a)
        set $s_{t+1}=s_t,a_t$, $x_{t+1}$ and preprocess $\omega = \omega(s_{t+1})$
        set target value
            $Q_{tar}=r_t+1+\gamma\, Q_{max}(\omega_{t+1})$
        perform gradient descent on $(Q_{tar} - Q(\omega(s_t),a_t))^2$ with respect to network parameters
    End for
End For

In the naïve implementation of the Deep Q network, we initialize the neural networks with random weights. For every N episode, we first select a random action for epsilon probability, or else we select the maximum action value using the current estimates of the Q. The action is passed into the environment and the next state and reward are received. The Q target is now calculated using the reward, next state, and gamma values. Then, the loss is calculated by mean squaring the difference between Q approx. and Qtarget. Finally, gradient descent is performed on the loss with respect to the weights.

2) **Deep Q Learning with Experience Replay**

Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t$ = 1,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t),a;\theta)$
    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 $D$
    Sample random minibatch of transitions $(\phi_j,a_j,r_j,\phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1},a';\theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j,a_j;\theta)\right)^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**

In DQN with the Experience Replay algorithm, we first initialize replay memory of size D. In our case it is 10000. Next, we initialize two networks Qeval and Qnext with random initial weights. For N episodes, 250 in our case, we first preprocess a sequence of 4 images grayscaled, downscaled, and stacked together to form the initial state. We also initialize a score variable to 0, to which we add our rewards for every time step. Now we start a loop and iterate until we are done. We first select action as per the epsilon greedy and observe the next state, reward, and the done information. Now we store the tuple (state, action, reward, next state) in the memory buffer. We also append the score with the reward. Also, note that we initialize a memory counter to track the number of experiences in the memory buffer. Once the memory counter has reached the batch size i.e 32, then only we start the

learning process. During the learning process, we sample an experience from the memory buffer and use it to train the network. From the experience tuple, we take state s and get the estimate of q_pred through the Qeval network for the selected action $a_t$. We also use next state $s_{t+1}$ to get q_next using the target network Qnext. Now we calculate the q_target using the formula putting gamma and reward $r_{t+1}$ values.

$$q\_target = r_{t+1} + \gamma \, max\, q_{next}$$

Finally, we perform the gradient descent on the loss $(q\_target - q\_pred)^2$ to make the network Qeval learn. We also decrease the epsilon after this step. Periodically, in our case every 1000 steps, we also update the target network with the weights of the evaluation network. After the learning process, we update the current state as the next state that was observed. We then move on to a new iteration where we select an action for the current state with the epsilon greedy policy. The same process is repeated till we get "done" as true from the game.
Once an episode is finished, we append the episode score to a list. We also print the average score, best score, and the epsilon value at the end of every episode. Whenever the average score becomes higher than the previous best average score, we save the model.

### 3) Double Deep Q Learning with Experience Replay

Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
**for** each iteration **do**
  **for** each environment step **do**
    Observe state $s_t$ and select $a_t \sim \pi(a_t, s_t)$
    Execute $a_t$ and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
    Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
  **for** each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma \, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

The implementation of the DDQN algorithm in terms of pre-processing, initializing the weights, sampling memory buffer, and selecting the action is the same as DQN. The only difference is in the learning process. IN DDQN also we have two networks Qeval and Qnext. When we sample an experience tuple(state, action, reward, next state), we take the state to calculate the q_pred using the Qeval. We also take the next state to calculate the q_next. However, instead of selecting the maximum action from q_next, we find q_eval bypassing the next state to the Qeval network. Based on q_eval we select the maximum action and use that action to calculate the q_target.

$$a = argmax(q\_eval)$$
$$q\_target = r_{t+1} + \gamma \, max\, q_{next(a)}$$

The rest of the algorithm remains the same.

## RESULTS

### Naïve Implementation

The results for Naïve implementation were as expected. We plot the graph for an average score with a moving average of 100 episodes, against the total number of steps in the training. Over 250 episodes and 200000 steps, the network hardly ever learns the game. The maximum average score it achieves is around -20. The reason for this type of behavior is that we are using the same network to approximate

the Q values and the same network to calculate the target values as well. If we train the model for a larger number of episodes let's say 10000, we might see some learning, however, that would be computationally expensive.

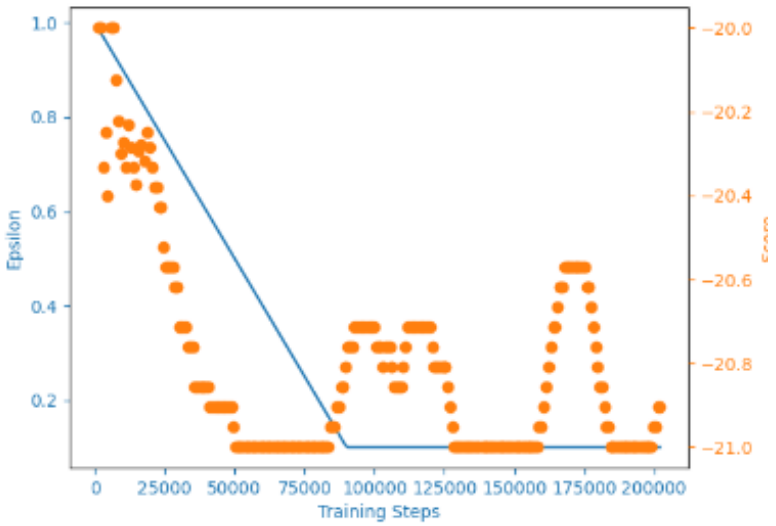


*Figure 4: Plot of average score and epsilon against the time steps for naïve implementation*

```
episode:  236 score:  -20.0  average score -20.5 best score -20.25 epsilon 0.78 steps 216910
episode:  237 score:  -20.0  average score -20.4 best score -20.25 epsilon 0.78 steps 217753
episode:  238 score:  -20.0  average score -20.4 best score -20.25 epsilon 0.78 steps 218619
episode:  239 score:  -20.0  average score -20.4 best score -20.25 epsilon 0.78 steps 219485
episode:  240 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.78 steps 220277
episode:  241 score:  -20.0  average score -20.4 best score -20.25 epsilon 0.78 steps 221119
episode:  242 score:  -21.0  average score -20.4 best score -20.25 epsilon 0.78 steps 221999
episode:  243 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.78 steps 222851
episode:  244 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.78 steps 223738
episode:  245 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.78 steps 224590
episode:  246 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.77 steps 225520
episode:  247 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.77 steps 226406
episode:  248 score:  -20.0  average score -20.5 best score -20.25 epsilon 0.77 steps 227244
episode:  249 score:  -21.0  average score -20.5 best score -20.25 epsilon 0.77 steps 228027
... saving checkpoint ...
PS C:\Users\risha\Desktop\RL>
```

*Figure 5: Snapshot of Episode, average score, best score, epsilon, and timesteps for the final training episode of Naïve Implementation*

**Deep Q Network**

Using the DQN agent, we can see solid evidence of learning. The agent performs pretty well and learns the game in 250 episodes and 500000 steps. We can see from the graph, that there is an upward trend towards positive high scores during the initial stages of learning and it tends to stabilize at the end. One interesting thing to notice is that, during the initial stages when epsilon is slightly higher, the learning is quite slow and the average score is still below zero. This is because initially the random weights approximate the Q values and these values are far from real estimates of Q. There is also a lot

of exploration in the initial stages. Since we are using a moving average of 100 episodes, a lot of negative values are included in the initial average score. With time, the values start to take better shape as well as we start selecting greedy actions more often with a decrease in epsilon.

The human benchmark when the paper was released was +9.3. We were certainly able to cross that benchmark and got an average score of +11.9 at the end. There were even some episodes in which the agent scored around +15. As compared to the original author's findings they got an average score of +18.0. The difference in results is due to several factors. Firstly, the authors trained the network for a week for 1000000 episodes, while I only trained for 250 episodes to show evidence of learning. Secondly, certain parameters such as memory buffer and update frequency were quite different. They use a memory buffer of 1000000 while I used 10000.

The testing results were quite different from what was expected. While testing the trained agent on the game, we found that it wasn't playing as good it should. The testing round was done for 30 games where the trained agent selected the greedy action at every state. The agent failed to win a single game. The average score was around -20. The reasons for this could be the same as mentioned above for the difference between my average score and that of the authors. Another reason could be that of overestimations. This provides a great opportunity to implement Double Q learning and compare the results

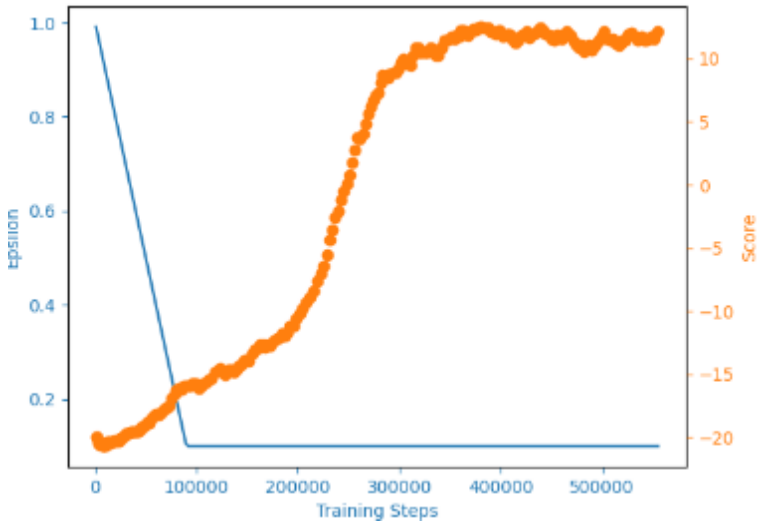


*Figure 6: Plot of average score and epsilon against the time steps for DQN implementation*

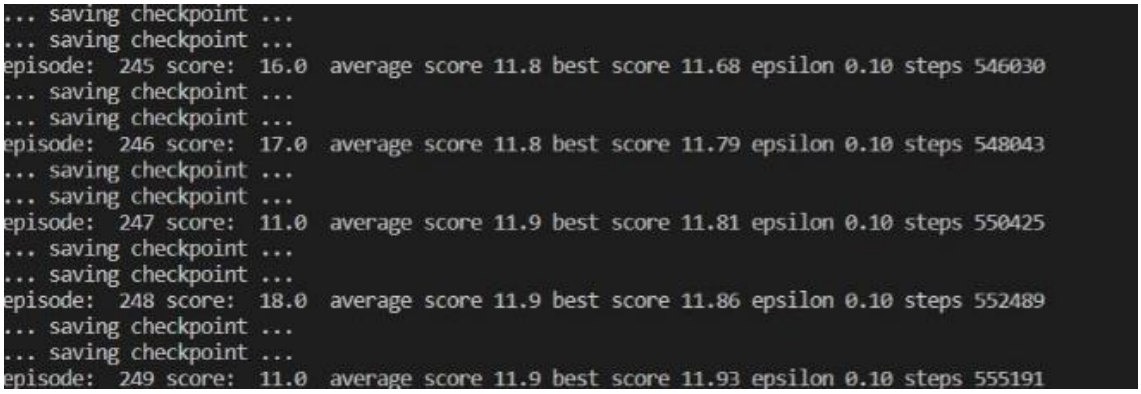


*Figure 7: Snapshot of Episode, average score, best score, epsilon, and timesteps for final training episodes of DQN Implementation*

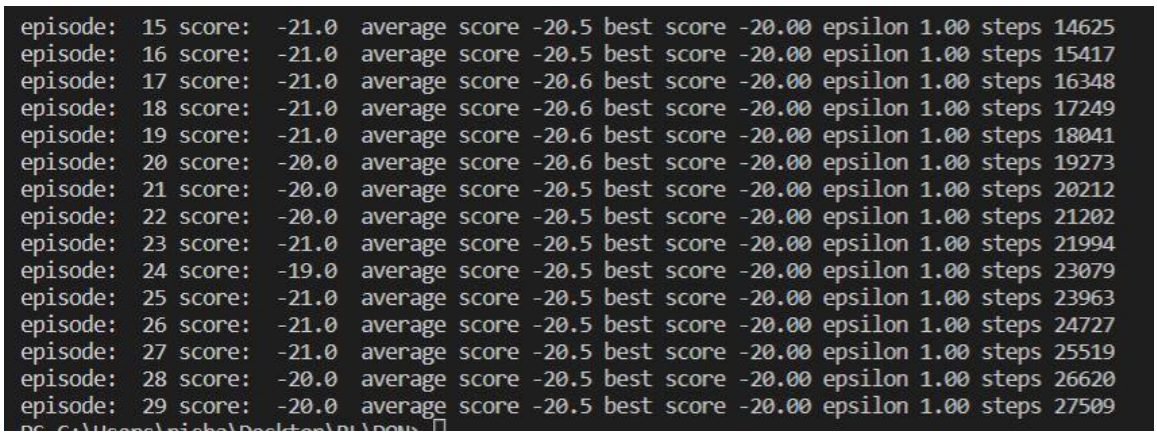


*Figure 8: Snapshot of Episode, average score, best score, epsilon, and timesteps for the testing episodes of DQN Implementation*

**Double Deep Q Network(DDQN)**

The training results obtained for DDQN were quite similar to DQN for 250 episodes and 500000+ steps. The learning curve seems to be similar to the DQN, where it starts slow during higher epsilon but then shoots up to around 200000 steps and then stabilizes at the end. However, the average score obtained here was slightly better. The agent obtains a score of +12 at the end of 250 episodes. The agent even wins some of the games in the process.

The main difference was however observed in the testing. The trained agent performed extremely well by winning all the 30 games against the emulator with a score of +20 in every game. As discussed in previous sections, the DQN algorithm suffers from overestimations of Q values. These overestimations are resolved in DDQN, which might be the reason why DDQN not only learns the environment in a better way but also performs extremely well compared to DQN

Another thing that was observed in the testing of DDQN, was the agent learned a strategy to hit the ball in the lower wall, such that it bounces off at a particular angle which makes it difficult for the opponent to return. The interesting thing was that the agent doesn't try to do that with the top wall. This might be because it might have received a lot of points during the initial stages for hitting the ball

downward, making the Q value for that action value higher. As the epsilon decreases, the agent starts selecting the greedy action more often, and hence the option to hit the ball using the top wall might not have been explored. In testing, we only select the greedy action and hence the agent keeps attempting to hit the ball in the downward direction.

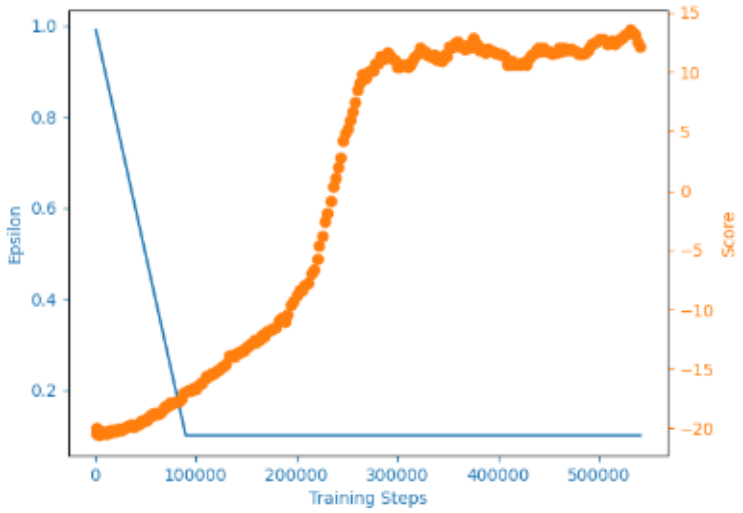


*Figure 9: Plot of average score and epsilon against the time steps for DDQN implementation*

```
episode:  241 score:  15.0  average score 11.8 best score 11.90 epsilon 0.10 steps 522810
episode:  242 score:  18.0  average score 11.8 best score 11.90 epsilon 0.10 steps 524603
episode:  243 score:  16.0  average score 11.9 best score 11.90 epsilon 0.10 steps 526553
episode:  244 score:  13.0  average score 11.9 best score 11.90 epsilon 0.10 steps 529153
episode:  245 score:  16.0  average score 11.9 best score 11.90 epsilon 0.10 steps 531068
... saving checkpoint ...
... saving checkpoint ...
episode:  246 score:  9.0  average score 11.9 best score 11.92 epsilon 0.10 steps 533440
episode:  247 score:  13.0  average score 12.0 best score 11.92 epsilon 0.10 steps 535605
... saving checkpoint ...
... saving checkpoint ...
episode:  248 score:  3.0  average score 11.8 best score 11.98 epsilon 0.10 steps 538770
episode:  249 score:  12.0  average score 12.0 best score 11.98 epsilon 0.10 steps 541293
PS C:\Users\risha\Desktop\RL\DDQN>
```

*Figure 11: Snapshot of Episode, average score, best score, epsilon, and timesteps for final training episodes of DDQN Implementation*

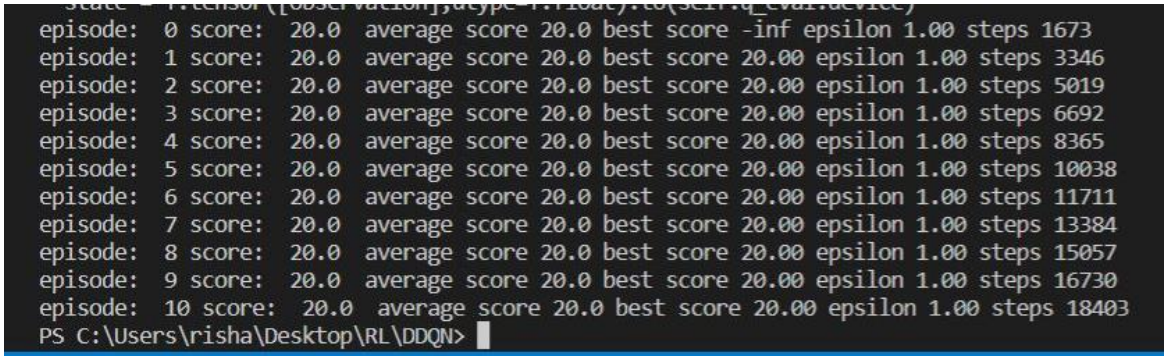


*Figure 12: Snapshot of Episode, average score, best score, epsilon, and timesteps for the testing episodes of DDQN Implementation*

**Challenges**

1) The biggest challenge for me in this project was to understand the Deep Q Networks and function approximation concepts. Most of the learning I did in the course was through the assignments. I understood the working of Neural Networks and function approximation through assignment 5, where I implemented the actor-critic algorithm. However, it was already the first week of April and I did not grasp deep Q learning in that detail. This impacted my timeline as I was spending additional time understanding the algorithms from different youtube videos and articles.

2) Implementing the code directly from the paper was another challenge I faced. Since I did not implement any other NN algorithm for RL, I had to start from the basic implementation. The details mentioned in the paper were great, however, for a beginner, it becomes tough to translate them into code. The pre-processing part was another tough task. Finally, I resorted to taking code for the implementation from a Github repository.

3) The training time was unprecedented. Initially, I started training for 500 episodes. However, it took around 10 hours for the network to train. There was a mistake in the path where plots and models were supposed to be saved, and hence it did not save anything at all. Then I changed the episodes to 250 and I completed the training in 6 hours. Another instance was when I had finished training and testing the model and I was running the code the debug some parts. While running the code, I realized that it has overwritten my previous trained models in the same path. Then again I spent 6 hours retraining the model.

4) I felt, that doing the project all alone was something that I did not enjoy. Although it was a great learning experience, I feel projects flourish when a team works on them. Every person brings a different set of skills to the table and it can be exploited in a way such that the outcome is great. For instance, somebody who is great at writing can team up with somebody great at implementing the code, and together they can learn from each other and make a great project. The solo project was just a lone battle to fight and sometimes it was too tough to move forward.

**Hyperparameters**

| | | |
|---|---|---|
| Learning rate | 0.0001 | Learning rate used by RMSProp |
| Gamma | 0.99 | Discount Factor used in the Bellman equation |
| Target Replace Frequency | 1000 | The frequency with which the target network is updated |
| Number of Episodes | 250 | Total Number of episodes |
| Action Repeat | 4 | Repeat each action selected by the agent this many times |
| Update Frequency | 4 | The number of actions selected by the agent between every update |
| Initial Epsilon | 1.0 | Initial of epsilon for exploration |
| Minimum Epsilon= 0.1 | 0.1 | Minimum value of epsilon for exploration |
| Memory Buffer Size | 10000 | Updates are sampled from this memory |

*Table 1*

REFERENCES


[1] "Human-level control through deep reinforcement," 2015.- Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

[2] "Deep Reinforcement Learning with Double Q-learning," 2015. - Hado van Hasselt and Arthur Guez and David Silver