Exploring Deep RL Algorithms

Team Members:
- Anuja Raghunath Katkar (50414752)
- Sagar Jitendra Thacker (50363834)

Ever since the first deep learning model was successfully implemented by Deep Mind [1] in 2013 there has been much advancement in the field of Reinforcement Learning (RL) on how to train the agent. The two main components in RL are the agent and the environment. Various improved algorithms have been introduced over the years to train an agent. Various aspects such as the number of states and actions, episodic or sequential environment, discrete or continuous environment, single or multi-agent, etc needs to be considered while choosing an algorithm. Defining a well-structured environment is important to depict real world problems.

Objective:
In this project we propose to explore different deep reinforcement learning algorithms developed over the years on environments provided in Open AI gym. We'll compare the performance of these algorithms in each of the environment to better understand how the algorithm affects the agent behaviour in those environments.

Related Work:
Atari games have been solved by Deep Mind. We will attempt to improve the performance of the agent using different algorithms (compared to the original paper [1]) along with comparisons with different parameters for the algorithms.

Compared to the initial proposal submitted, there are a few changes in the work we have done in our final project. Due to the extensive use of GPU's, lack of resources, huge training time and hyper parameter tuning it was very difficult for us to implement to complex environments i.e. Atari Breakout and Atari Pong. Hence, after having a word with the professor we decided to reduce the number of complex environment to one and added Grid World Environment to the list. We have also added one more environment to the list. Final list of environments used for the projects are:

- Grid World Environment
- CartPole-v1
- Acrobot-v1
- Atari Breakout

Environment Description:

1. Grid World:
The environment is a 5*5 Grid World that consists of 25 states. The environment is fully observable, Single-agent, Episodic, and Discrete.

s21	s22	s23	s24	s25
s16	s17	s18	s19	s20
s11	s12	s13	s14	s15
s6	s7	s8	s9	s10
s1	s2	s3	s4	s5

State set: It is the set of all possible states present in the environment.

$$S = \{s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s16, s17, s18, s19, s20, s21, s22, s23, s24, s25\}$$

State s1 is the starting state and state s25 is the goal state. States s4, s13 contains reward of +5 and +10 respectively. State s16 contains a reward of -3. States s9, s12 are dead states i.e. when an agent reaches that state the environment resets and the agent receives a reward of -50 in each state. State s25 contains a reward of +50. All the other states have a reward of -1. An episode ends if the agent reached the goal state or any one of the dead states.

Action set: It is the set of actions that an agent can take in the environment. For the grid world environment the action set is given below:

$$A = \{Up, Down, Left, Right\}$$

Rewards: Reward set: It is the set of all possible rewards an agent can receive from the environment after taking an action. For the grid world environment the reward set is as follows:

$$R = \{-1, -3, 5, 10, -50, 50\}$$

Visualization of the environment

Below are some screenshots of the environment taken at different time-steps:

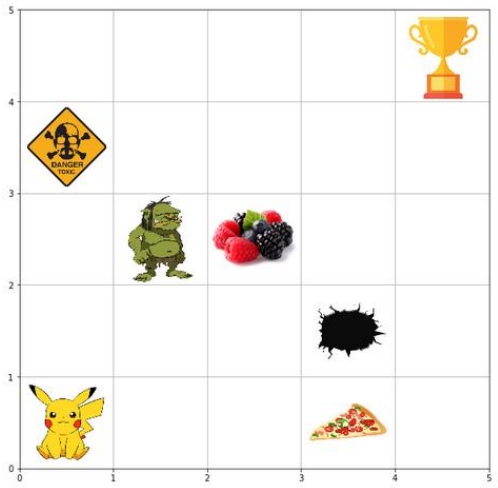

Figure 1: Initial State

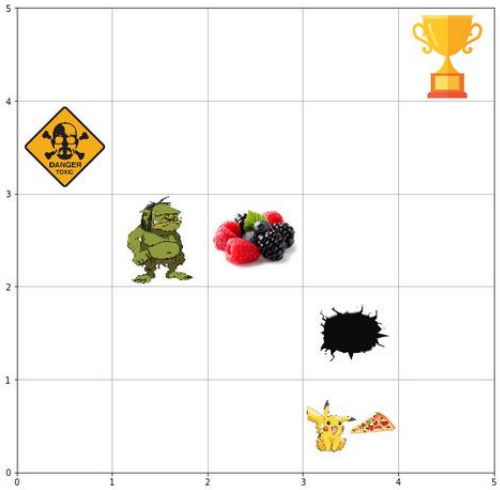

Figure 2: Agent received reward

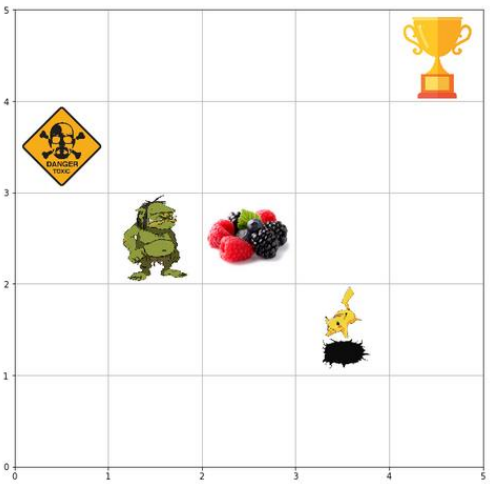

Figure 3: Agent fell in the pit

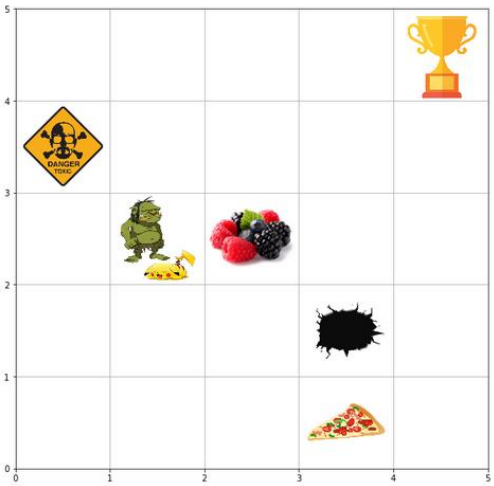

Figure 4: Agent killed by the monster

Pikachu represents the agent in the starting state (bottom left corner) and the trophy represents the goal state. Pizza and Berries represent rewards of +5 and +10 respectively. The danger sign represents a reward of -3. Monster and Pit are dead states i.e. when an agent reaches that state the environment resets and the agent receives a reward of -50 in each state. The trophy represents is our goal state and contains a reward of +50.

2. CartPole-v1:

In this environment, a pole is attached to a cart and the pole moves across the frictionless surface. Here, the goal is to balance the cartpole i.e., prevent it from falling by sliding the cart left or right.

Observation:

Number	Observation	Minimum	Maximum
0	Cart Position	-4.8	4.8
1	Cart Velocity	-∞	∞
2	Pole angle	-0.418 rad (-24°)	0.418 rad (24°)
3	Pole angular velocity	-∞	∞

Action space:
For cartpole-v1 environments, the cart can only take two actions to balance the pole above the cart.

$$A = \{Left, Right\}$$

Rewards:
The reward will be +1 for each step, including the termination step. The maximum reward cartpole can reach is 500.

$$R = \{+1\}$$

Starting state:

All observations are assigned a uniform random value in $[-0.05 \ldots 0.05]$

Episode termination:

1. Pole angle > ±12°
2. Cart position > ± 2.4 (center of the cart reaches the edge of the display)
3. Episode length > 500
4. Solved requirements:
   4.1. Average return >=475.0 over 100 consecutive trials.

3. Acrobot-v1:

In the acrobot system there are two joints and two links, where the joint between the two links is actuated. Initially the links are hanging downloads and the goal is to swing the lower part of lower link to a given height.

Observation:
The sate consists of the sin() and cos() of the two rotational joint angles and the join angular velocities:

$$[cost(theta1)\ \sin(theta1)\ \cos(theta2)\ \sin(theta2)\ thetaDot1\ thetaDot2]$$

Action Space:
The action is either applying +1, 0, or -1 torque on the joint between the two-pendulum links

$$A = \{-1,0,+1\}$$

Rewards:
For every action the agent receives a reward of -1.

Solved Requirement:
If the average reward for 100 episodes is greater than -80 then it is considered the environment is solved and the agent has learnt.

4. Atari Breakout (BreakoutDeterministic-v4):

In Atari breakout, at the top, there are eight rows of bricks and at the bottom, there is a paddle that ricochets the ball against the bricks and destroys them. Here, we use a single ball to eliminate all the bricks and the player must be able to eliminate as many bricks as possible by directly bouncing the ball at the bricks or by using the wall. If the player's paddle misses the ball's rebound, they will lose a turn.

Input:
The observation is an RGB image of the screen, which is an array of shape (210, 160, 3)

Action space:
For Atari breakout environments, the cart can only take two actions to balance the pole above the cart.

$$A = \{Left, Right, Noop, Fire\}$$

- Left: move the paddle to the left
- Right: move the paddle to the right.
- Fire: used to start the game and every time the agent loses a life.
- Noop: paddle doesn't move. (No operation)

Rewards:
The reward will be +1 for each step. The range of the reward is $(-\infty, \infty)$

$$R = \{+1\}$$

Termination Conditions:
The episode terminates after losing 5 turns.

Initial Results of our model

For each environment all the four algorithms and their results are described below:

1. Grid World

1.1. Deep Q-Network:

Below we can observe that the epsilon is decaying over the number of episodes. Total reward per episode is fluctuating in the beginning as the agent is exploring the environment and then stabilizes as the agent takes greedy actions. Agents received the maximum possible reward while taking greedy actions. This can be validated in the average reward per 50 episodes graph, in gradually decreases as the agent explores and learns more about the environment. As the agent takes greedy actions the agent reaches the goal in the optimal number of time steps. In the cumulative reward overall episodes, it's increasing.

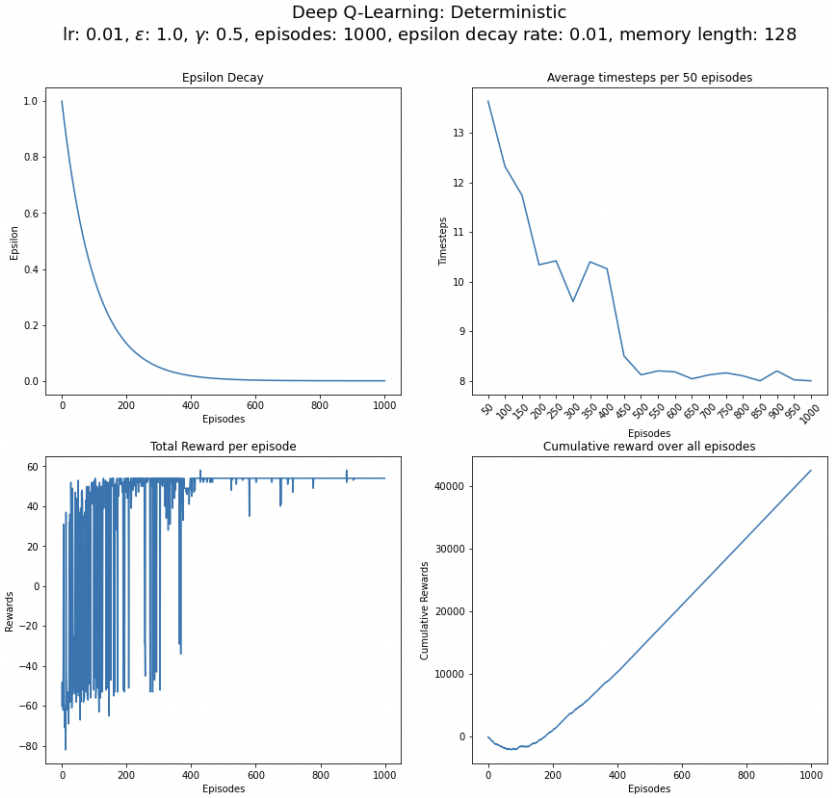

Figure 5: Training results of DQN on Grid World Environment

After training, evaluating the agent:

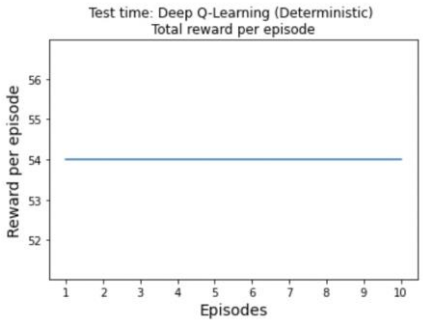

Figure 6: Agent test results of DQN on Grid World Environment

The above graph (Figure 6) showcases the total reward per episode for 10 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

1.2. Double Deep Q-Network:

In Double Deep Q-network we can see that the agent has converged after 600 odd episodes. Even though DQN converged before Double DQN, we have more stable results in Double DQN compared to DQN i.e. less variation in the graph. In total reward per episode graph it can be seen that as the agent explores there is more variation and as agent takes more greedy actions it reaches the optimal number of reward it can achieve.

This can be validated as the average timesteps per 50 episodes is decreasing and the agent takes the optimal number of steps to reach the goal. And the cumulative reward had a small dip in the beginning but started increasing as the agent started to learn more.

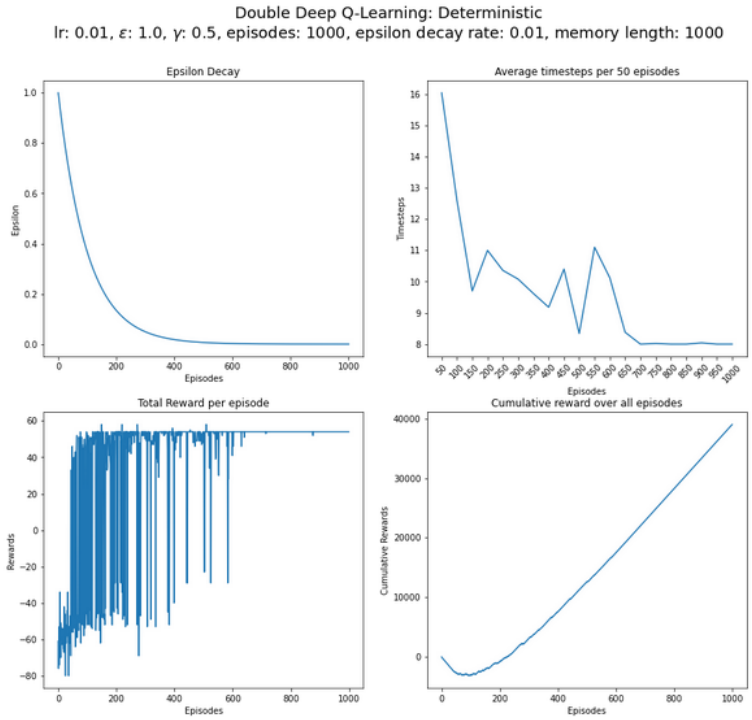

Figure 7: Double DQN results on Grid World Environment

After training, evaluating the agent:
The below graph (Figure 8) showcases the total reward per episode for 10 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

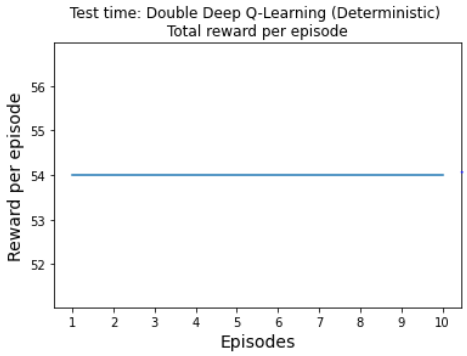

Figure 8: Test results of Double DQN on Grid World Environment

1.3. Advantage Actor-Critic (A2C):

Here, the agent can be clearly seen to take random actions till about 400 episodes, from where it starts approximating the policy well. Post about 400 episodes, it has created a working approximation of the policy and takes the best actions from about 400 episodes. Were we following an epsilon greedy policy, we would see a lot more fluctuations after the 500 episode mark, whereas in the above graphs, the convergence is fairly stable.

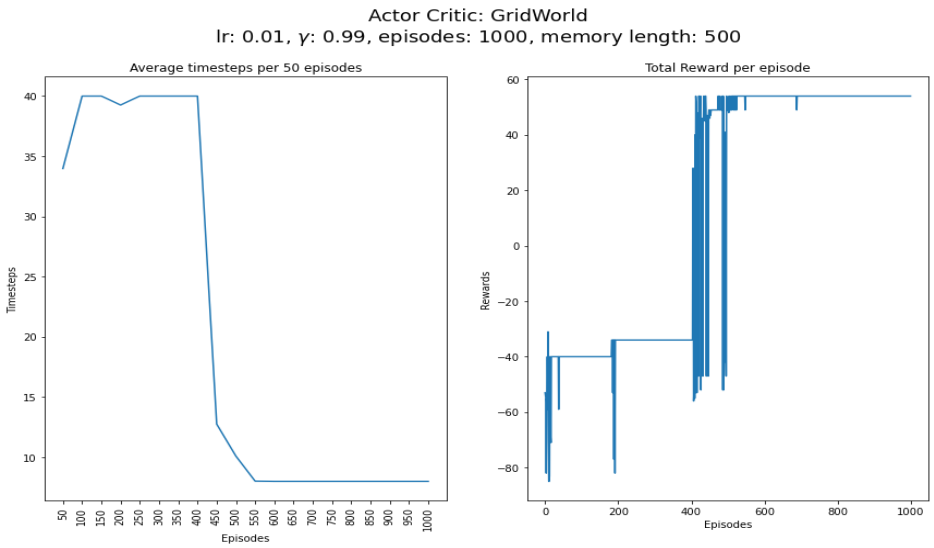

Figure 9: A2C results on Grid World

After training, evaluating the agent:
Here you can see that the returns are stable for the Grid World environment.

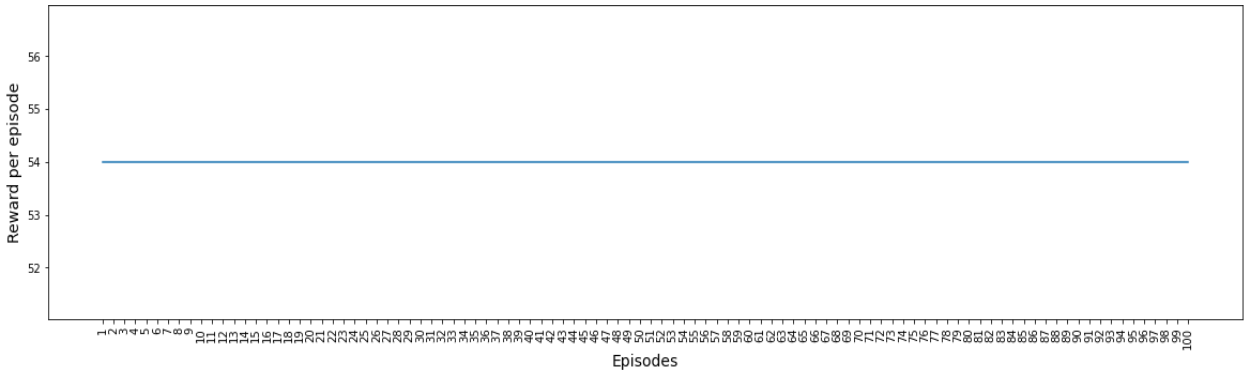

Figure 10: Test results on Grid World

1.4. Dueling DQN:

In the below graph we can see that after 400 odd episodes the agent has learnt and gets the maximum possible reward. Although there are a few bumps and a very few drops after that but after 600 episodes the agents has stabilized. After which the agent always gets the maximum reward. Similar behaviour can be seen in the average time-steps per 50 episode graph where the average is decreasing and stabilizing after 600 episodes.

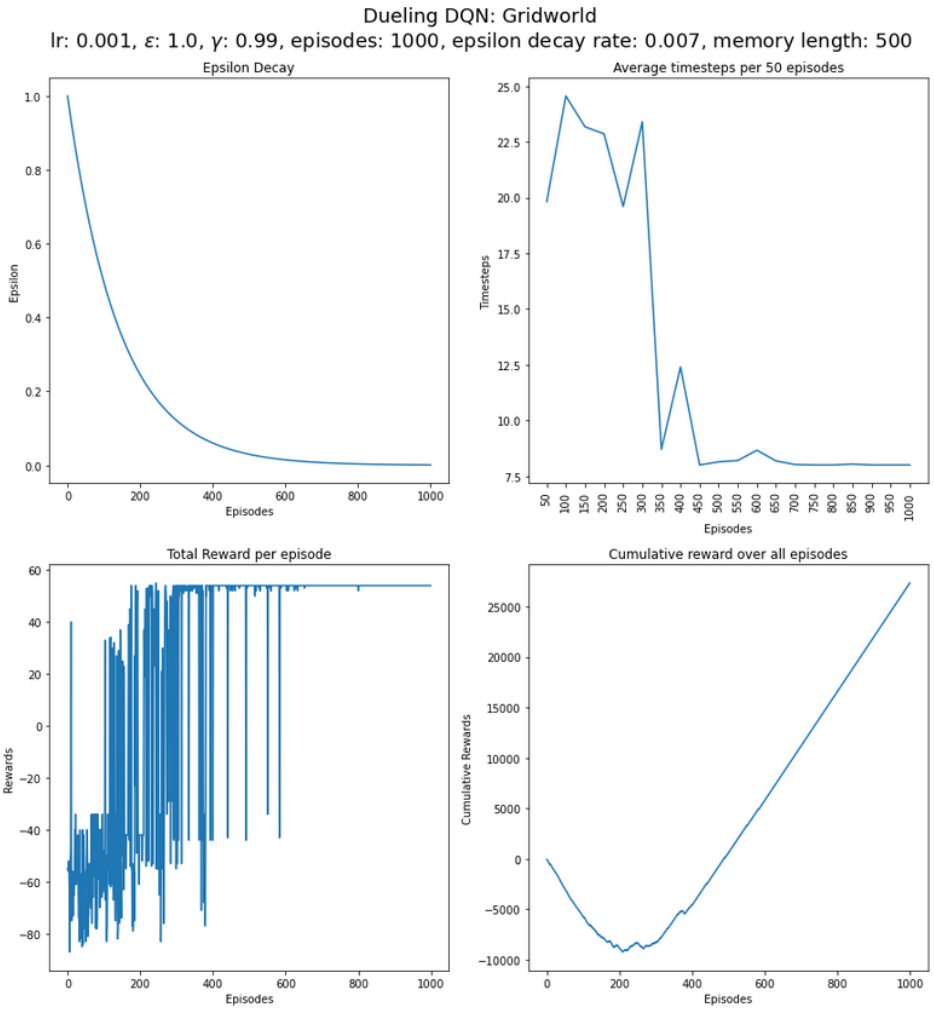

Figure 11: Dueling DQN results on Grid World

After training, evaluating the agent:
The below graph (Figure 12) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

Figure 12: Test results on Grid World

2. CartPole-v1:

2.1. Deep Q-Network:

Since CartPole is a challenging environment compared to grid world we have incorporated some techniques to improve the performance and make the agent more stable. CartPole is challenging because the agent receives a reward of +1 for each step and also then the agent falls i.e. the episode terminate the agents receives a +1 reward. The environment is challenging because the agent does not receive a negative reward to indicate that a bad action was performed.

Here we terminate the training when the average reward for the last 10 episodes is greater than 475 then we stop the training. We do this to avoid the agent learning some bad actions in the later stages. Also, rather than training the Q-

network every timestep we train the network every 10 timesteps. Similar functionality with some change in values is applied to DQN and Double DQN algorithms for CartPole and Atari Breakout.

In the graph below we can see that the total reward per episode is increasing and converging to its optimal values after 550 odd episodes. The variations in the graph can be seen due to the random actions performed by the agent while exploring and slowly converges to the optimal value as the agent takes more greedy actions.

Likewise we can see in the average timesteps per 50 episodes that the timesteps are increasing as the agent is able to balance the pole for longer time and receive more rewards. Here, we have used the default learning rate for Adam optimizer. The episode decay graph is short because the training was terminated before the value set for total episodes as 1000. Epsilon decay graph can be seen in the figure 14

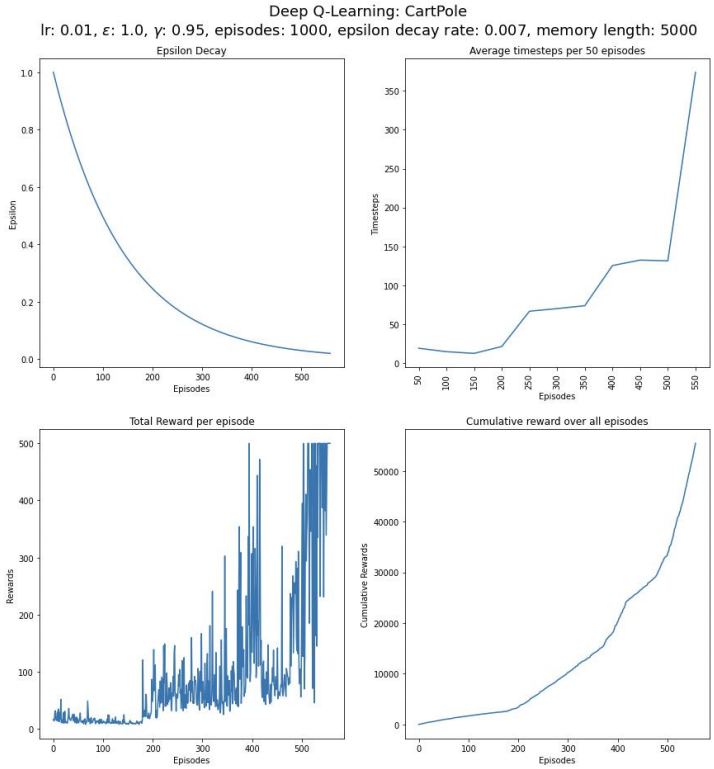

Figure 13: DQN results on Cartpole-v1

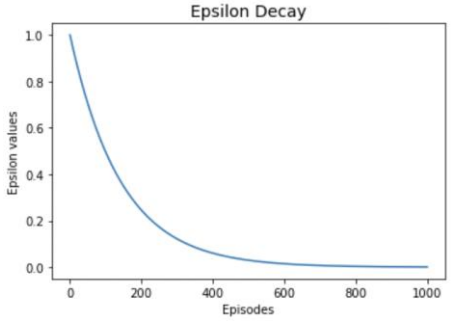

Figure 14: Epsilon decay for DQN CartPole

After training, evaluating the agent:

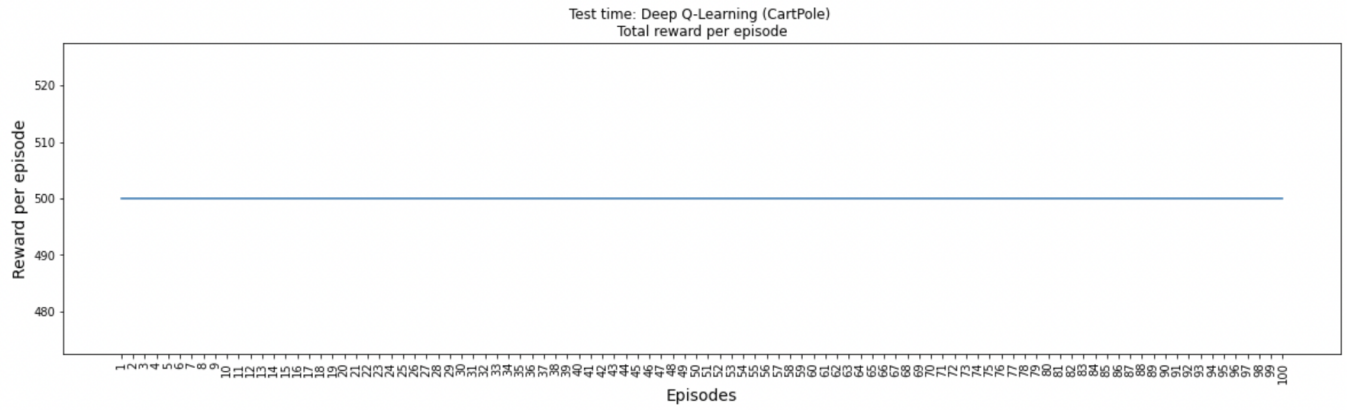

Figure 15: Test results of DQN on CartPole

The above graph (Figure 15) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

2.2. Double Deep Q-Network:

In this setting we terminate the training when the average reward for the last 10 episodes is greater than 475 and we train the agent after every 10 timesteps. Here we can see in the total rewards per episodes that compared to DQN the agent has not overestimated the Q-values since the graph is increasing slowly and reaching its optimal value.

Similar behaviour can be seen in the average timesteps per 50 episodes graph, it is increasing as the agent learns how to better manage the pole and receive more rewards. Likewise cumulative rewards are also increasing. Here, we have used the default learning rate for Adam optimizer. The episode decay graph is short because the training was terminated before the value set for total episodes as 1000. Epsilon decay graph can be seen in the figure 9.

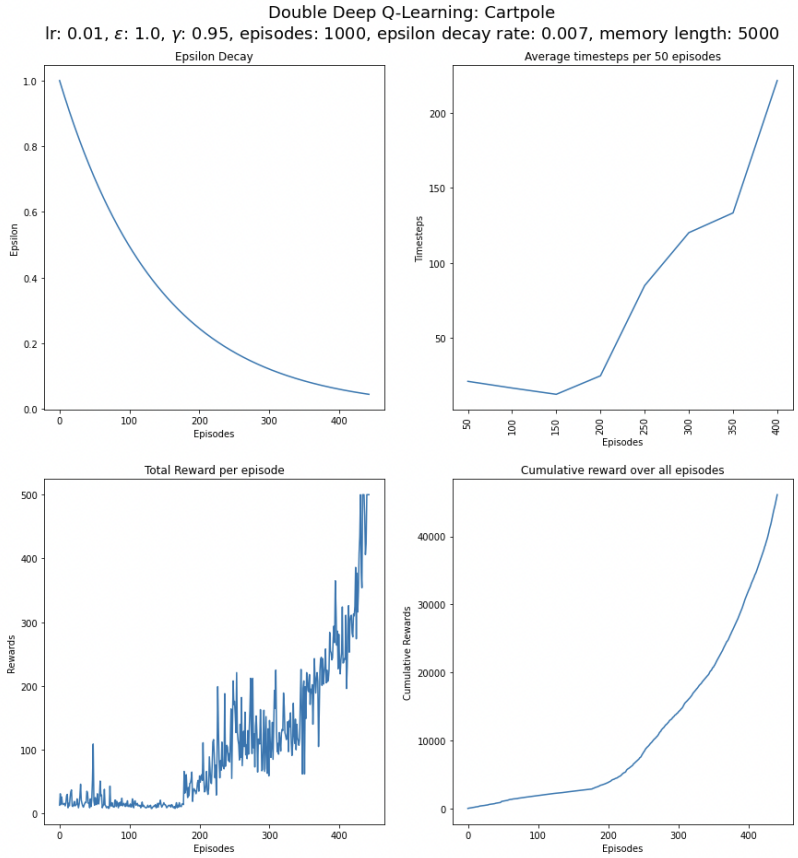

Figure 16: Results of Double DQN on CartPole

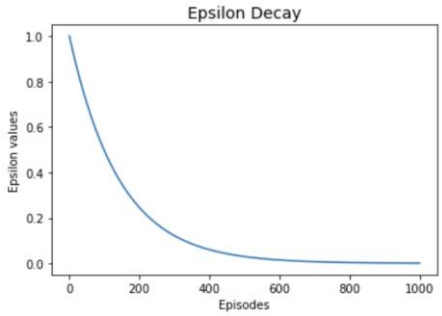

Figure 17: Epsilon Decay graph for Double DQN on CartPole

After training, evaluating the agent:

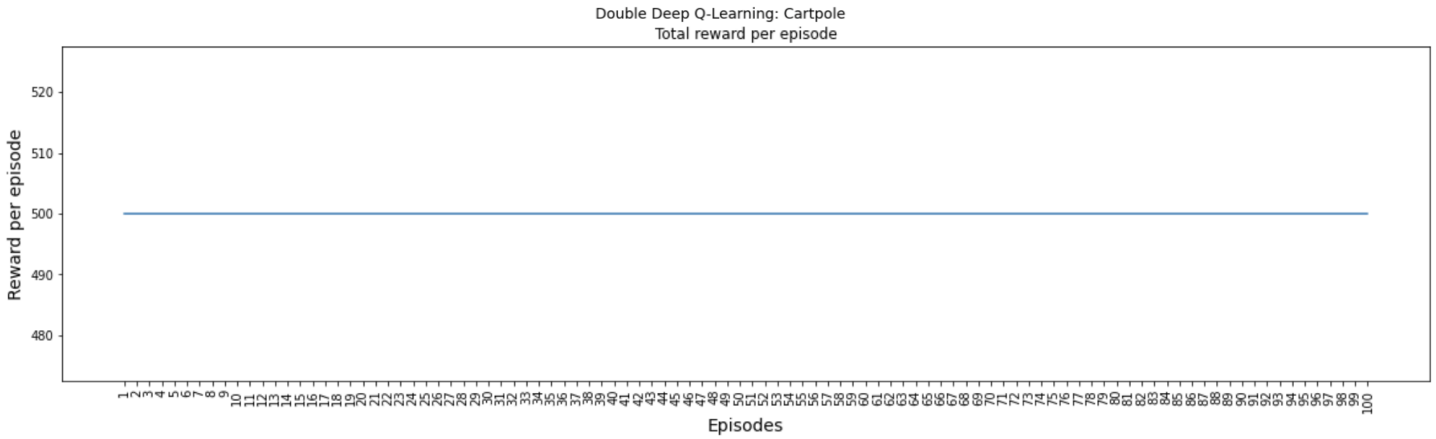

Figure 18: Test results of Double DQN on CartPole

The above graph (Figure 18) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

2.3. Advantage Actor-Critic (A2C):

In the below figure 19 we can see that till 550 odd episodes the agent was unable to learn much from the environment. But as the training went on it tried to learn the policy and optimal q-values to receive higher rewards. It was able to reach the optimal reward of more than 475 for multiple times but was unable to hold that for many episodes.

We had set criteria where if the agent receives on average for last 15 episodes more than 475 then we stop the training. This criterion was met around 1800 odd episode and the training was stopped. Here, we observe that the agent has learnt and receives maximum possible reward.

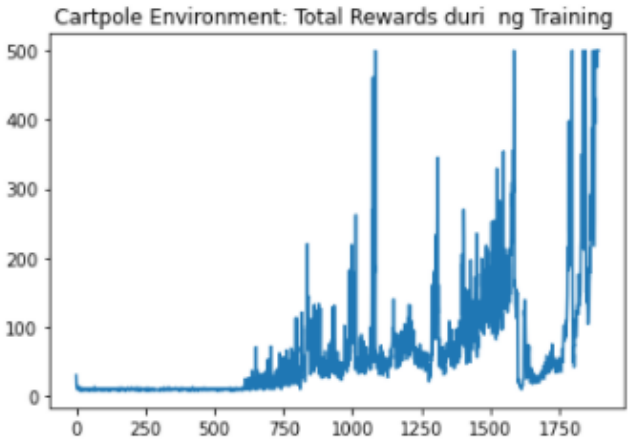

Figure 19: Training results on CartPole

After training, evaluating the agent:

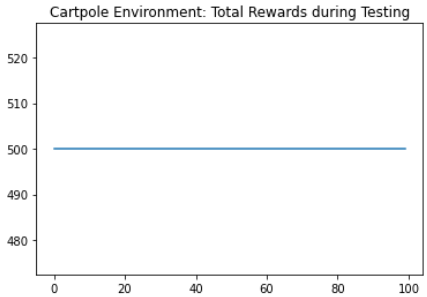

Figure 20: Testing result on CartPole

The above graph (Figure 20) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

2.4. Dueling DQN:

In the below graph we can observe that the agent can obtain the maximum reward after 150 episodes but is unable to hold it for long. Here we have set an exit criterion that the training stops when the average of last 20 episodes is greater than 475. This is achieved after 600 episodes. Here the agent finally learns and stabilizes. In the average reward per 50 episodes decreasing near the end i.e. it and achieve the maximum reward in minimum number of steps.

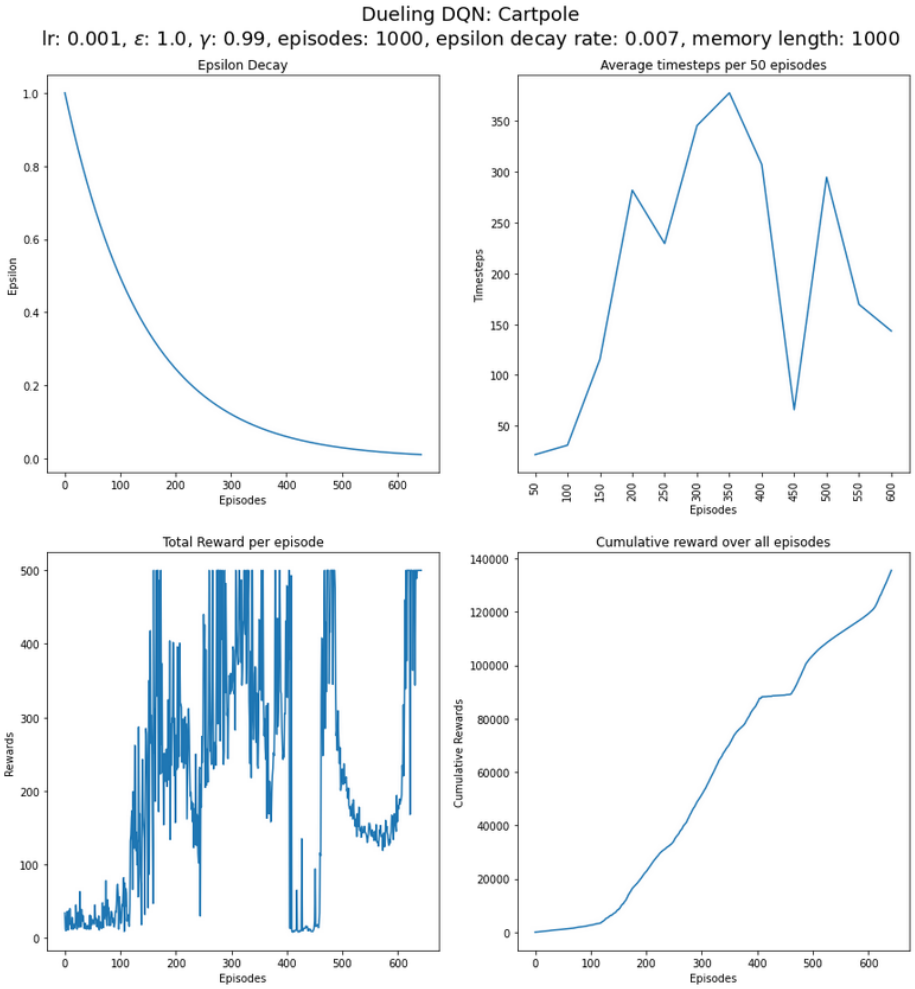

Figure 21: Dueling DQN Training results on CartPole

After training, evaluating the agent:

The above graph (Figure 22) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

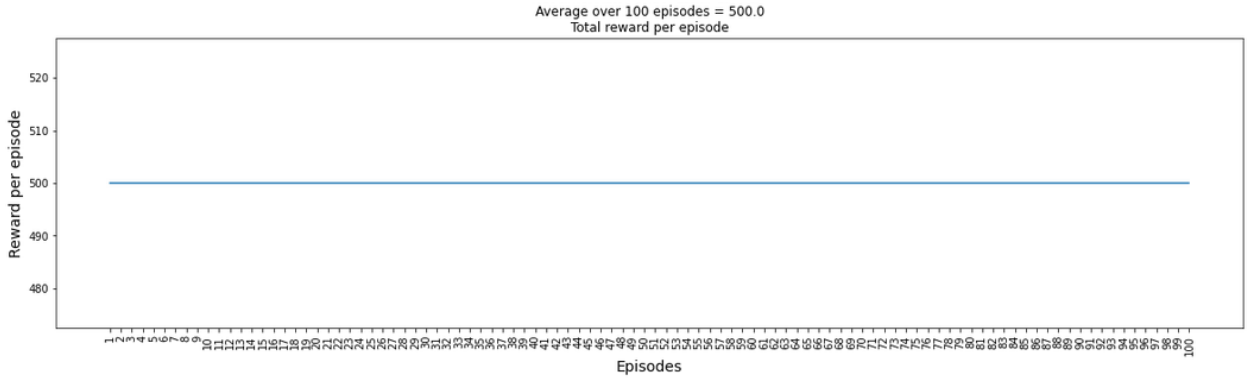

Figure 22: Testing results on CartPole

3. Acrobot-v1:

3.1.Deep Q-Network:

In total reward per episode we can observe that in the start the agent is trying to learn and explore and hence the rewards are fluctuating more. As the agent learns more it tries to converge to it optimal value. Being the environment to be complex compared to grid world and cartpole it is difficult to converge. There are few ups and down but the agent is able to learn the policy. Here we have an termination condition if the average of the last 100 episodes is greater than -75 then stop the training. The average time-steps per 50 episodes has also decreased and stabilized below 100 time-steps.

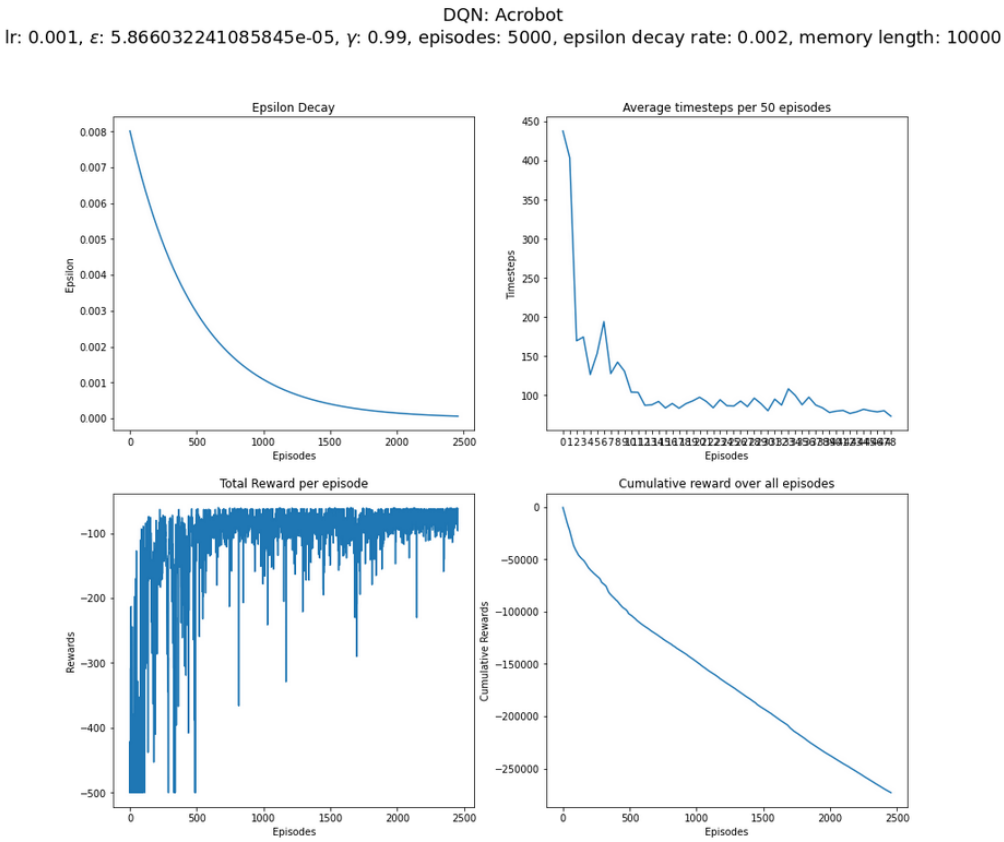

Figure 23: DQN results on Acrobot-v1

After training, evaluating the agent:
During testing we can see that the average reward for 100 episodes is -70.7 which is solved. (Referred from the mentioned link)

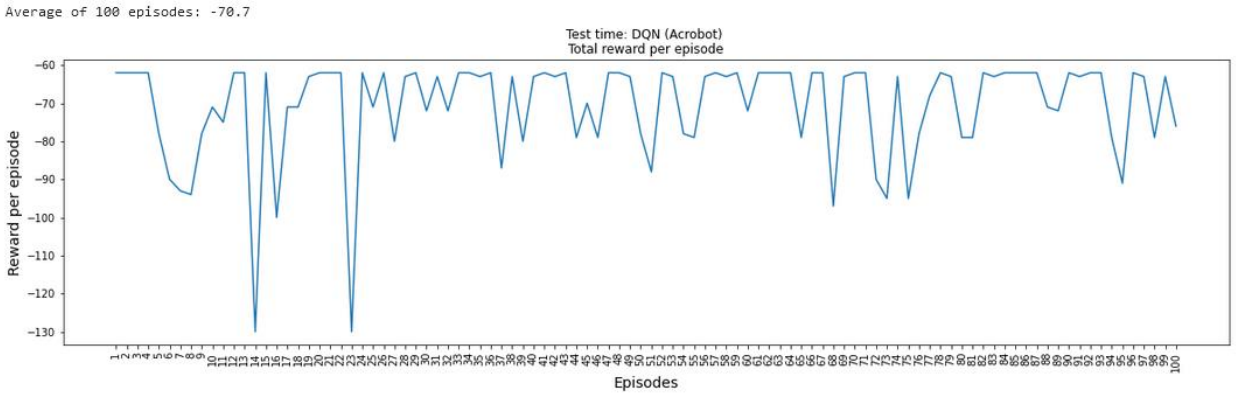

Figure 24: Testing results on Acrobot using DQN

3.2.Double Deep Q-Network:

In the total reward per episode graph we can observe that the graph is steadily increasing and converging to its optimal value. After 500 odd episodes the agent has tried to converge but we have set a terminating condition where if the average reward for the last 100 episodes is greater than -100 then stop the training. This criterion gets met around 2600 episode. Also, in the average timestep per 50 episodes we can observe a decline where the agent tried to reach the best reward in few timesteps.

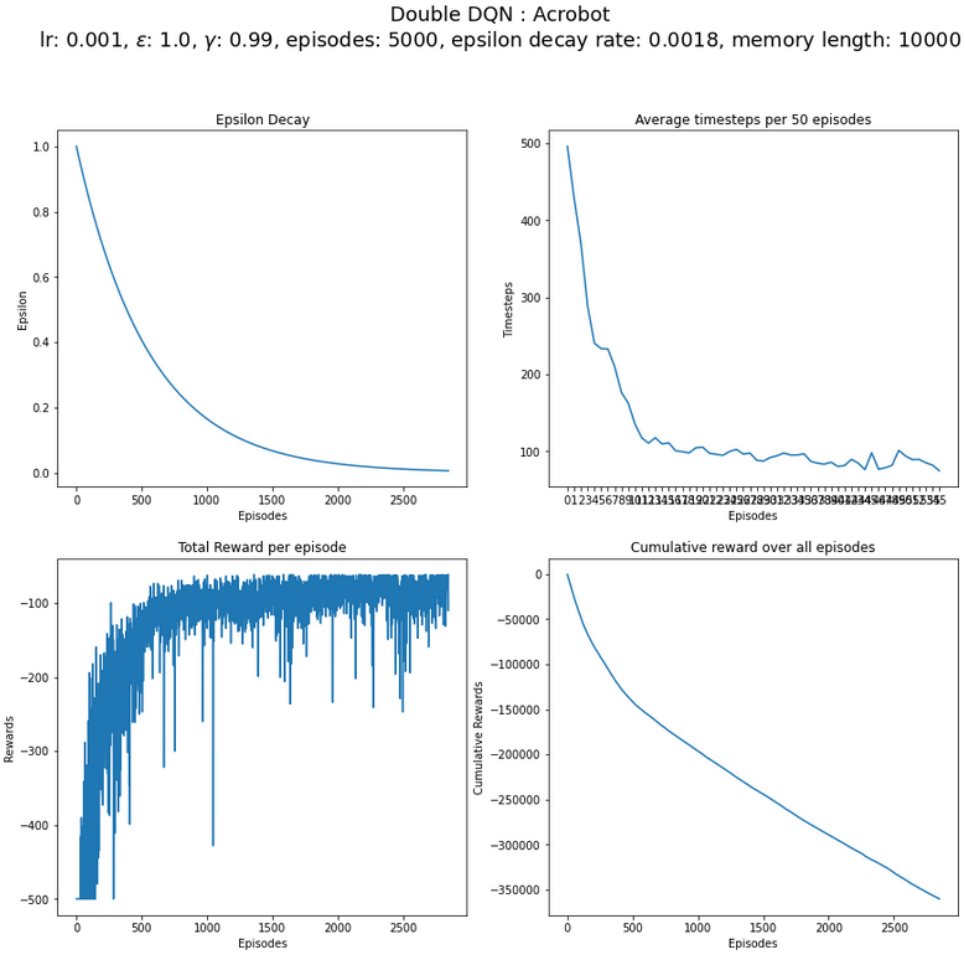

Figure 25: Double DQN result on Acrobot

After training, evaluating the agent:
During testing we can see that the average reward for 100 episodes is -78.43 which is solved. (Referred from the mentioned link)

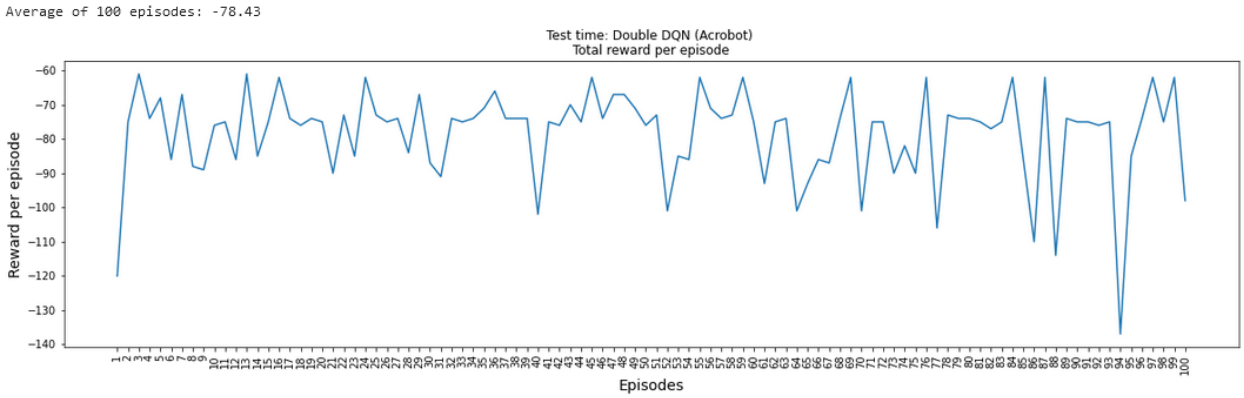

Figure 26: Testing result on Acrobot using Double DQN

3.3.A2C:

In the graph below (figure 6) we can see that after 750 odd episodes the dips in the graph have decreased and it tried to converge. Being the environment to be complex compared to grid world and cartpole it is difficult to converge. We have set an exit criterion for training that if the average of the last 100 episodes is more than -80 then stop the training and hence the training got stopped early.

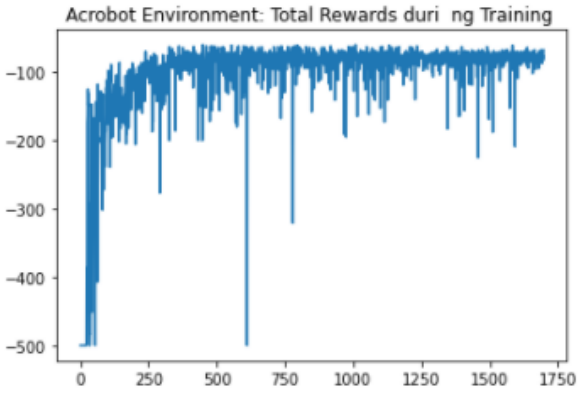

Figure 27: Training results for A2C on Acrobot-v1

After training, evaluating the agent:
During testing we can see that the average reward for 100 episodes is -80.5 which is solved. (Referred from the mentioned link)

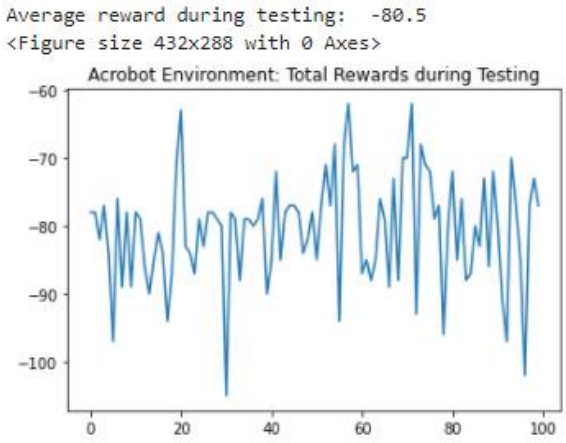

Figure 28: Testing results on Acrobot-v1

3.4. Dueling DQN:

In the below graph we can observe that the as the number of episode increases the graph is gradually increasing. Though there few ups and downs but the agent has tried and converge towards the optimal value. Being the environment to be complex compared to grid world and cartpole it is difficult to converge. In the average timesteps per 50 episodes we can observe a decline and stabilizes below 100 timesteps.

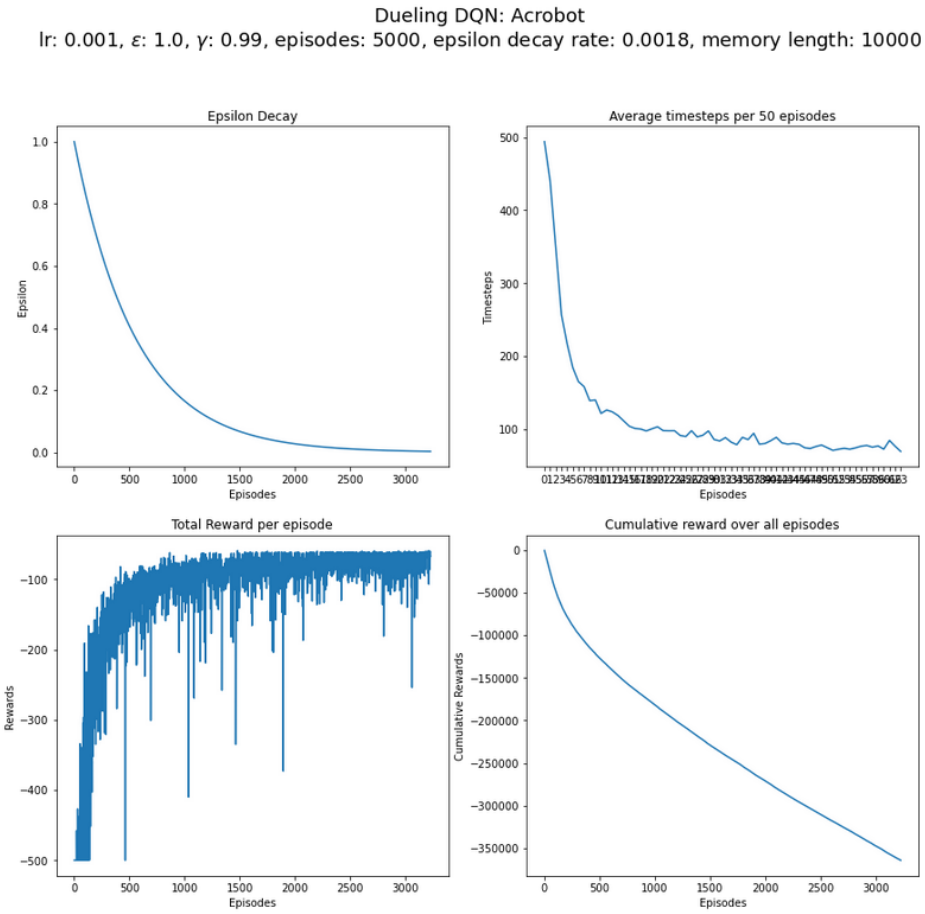

Figure 29: Dueling DQN results on Acrobot

After training, evaluating the agent:
During testing we can see that the average reward for 100 episodes is -66.95 which is solved. (Referred from the mentioned link)

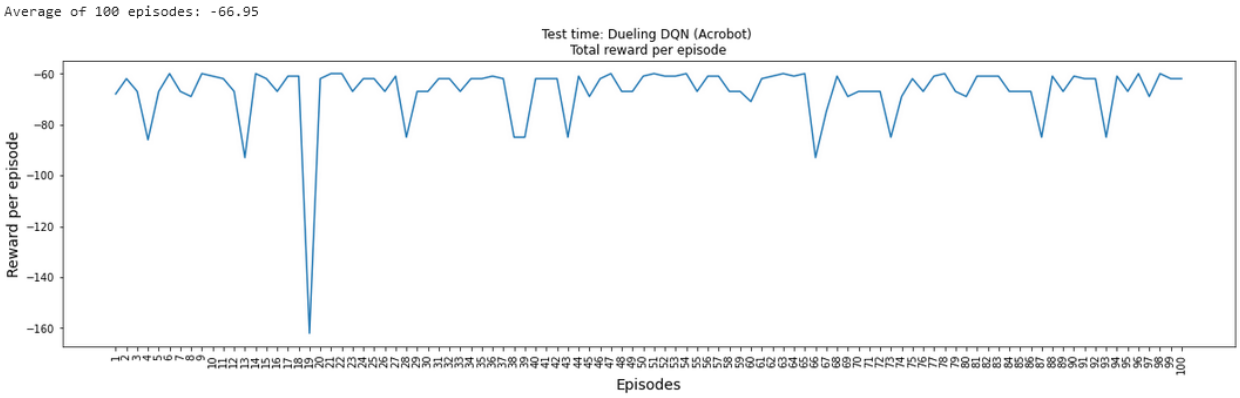

Figure 30: Testing results on Acrobot using Dueling DQN

4. Atari Breakout:

4.1. Deep Q-Network:

Atari Breakout is even more complex environment compared to CartPole-v1. Here each state is a combination of 4 frames. We use CNN to extract features and solve the environment. In the below figure 31, total reward per episode graph we can see that initially the scores are really low as the agent is trying to explore and learn. As the number of episodes increases the agents learns and score increases. As you can see in the graph it is difficult for the agent to learn. There is a lot of variation even during the end when agent is taking greedy actions. This is because of the environment is complex to learn. But in the average timesteps per 50 episodes we can see that the initially the timesteps were increasing, which slowly started to decrease i.e. the agent could get high scores by performing less actions and reach the optimal score faster.

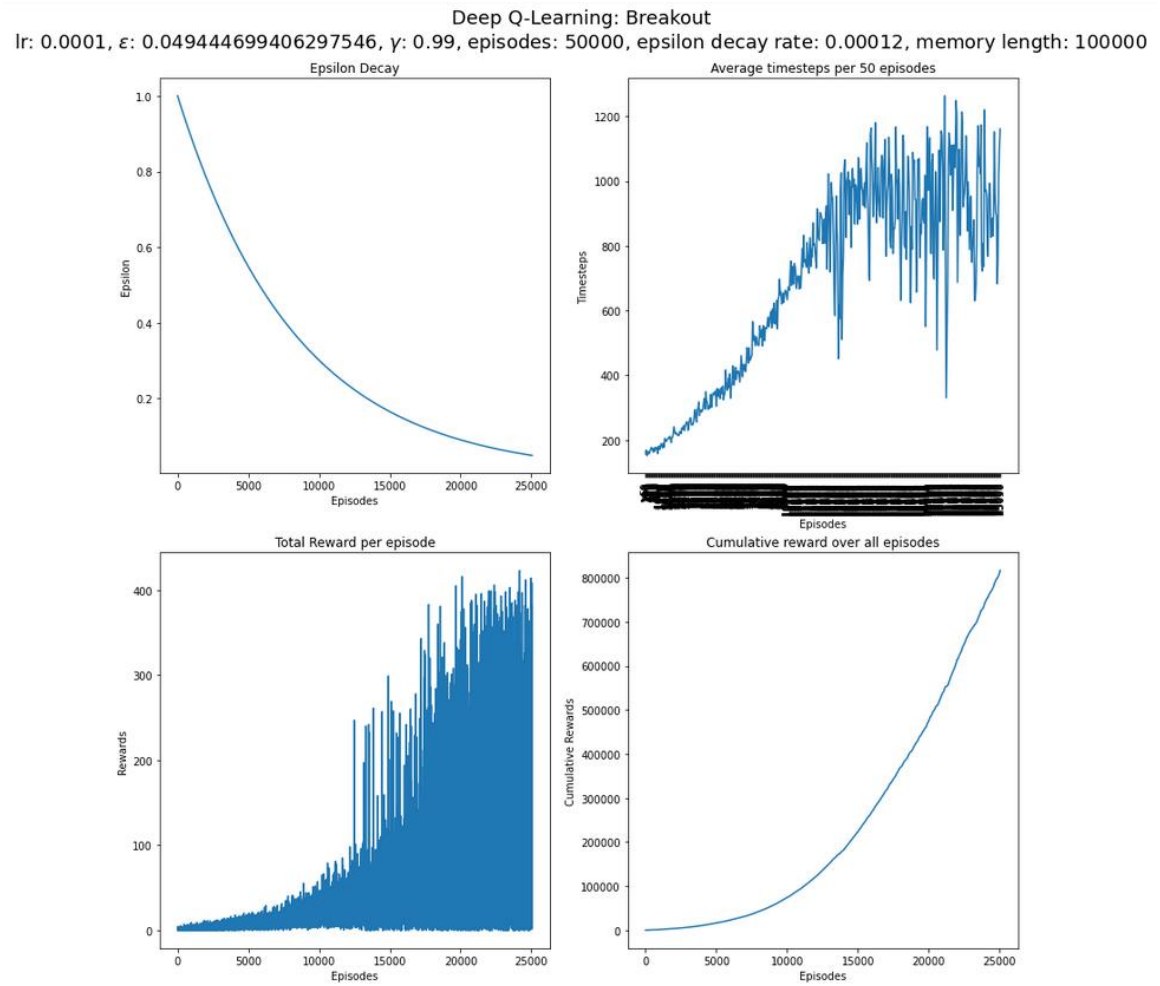

Figure 31: DQN result on Atari Breakout

After training, evaluating the agent:
The above graph (Figure 32) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

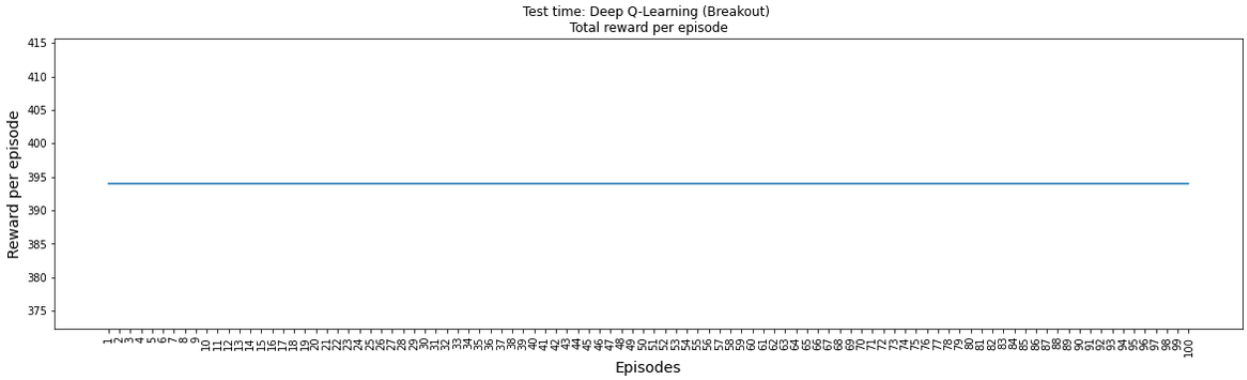

Figure 32: Test results DQN Breakout

The episode decay graph is short because the training was terminated before the value set for total episodes as 50000. Epsilon decay graph can be seen in the figure 33.

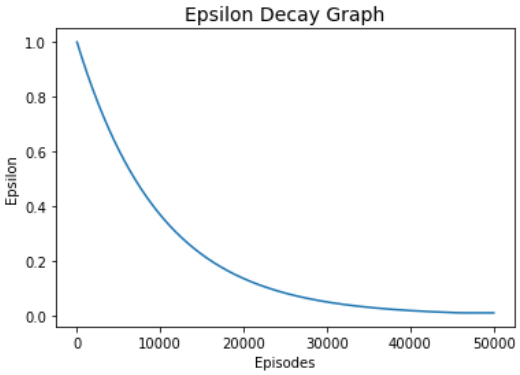

Figure 33: Breakout DQN - Epsilon Decay Graph

4.2. Double Deep Q-Network:

In Double DQN we can see similar results as DQN but there are comparatively stable than DQN. The rewards are low when exploring but learns quite well as the actions get greedy. Though there are a lot of variations but after 25000 episodes the score are more constant at 221. There can be an upper bound on the score the agent is receiving.

Similarly the average timesteps per 50 episodes is increasing and the slowly decreases and increases a bit to stabilize between 400 and 500 timesteps. Cumulative rewards are increasing as the number of episodes is increasing.

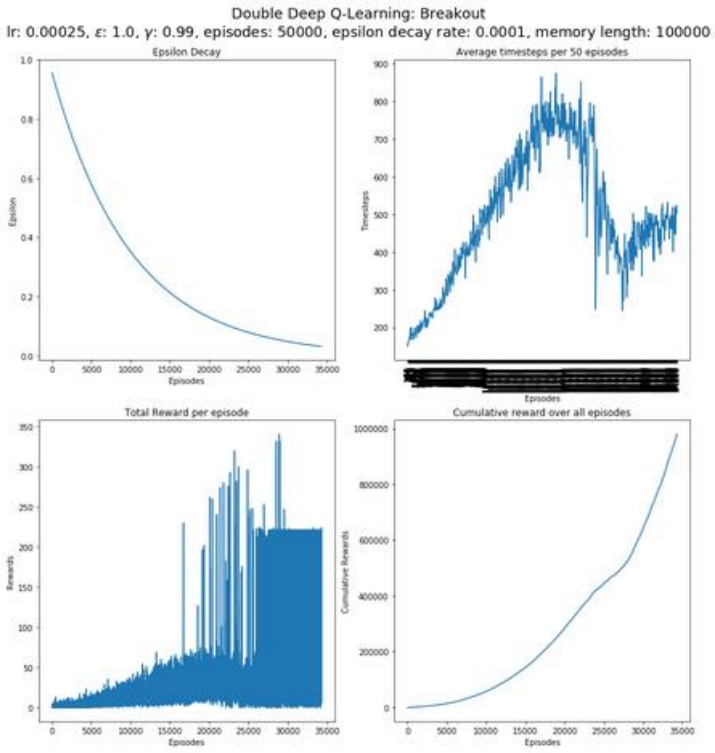

Figure 34: Double DQN Breakout Results

After training, evaluating the agent:
The above graph (Figure 35) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

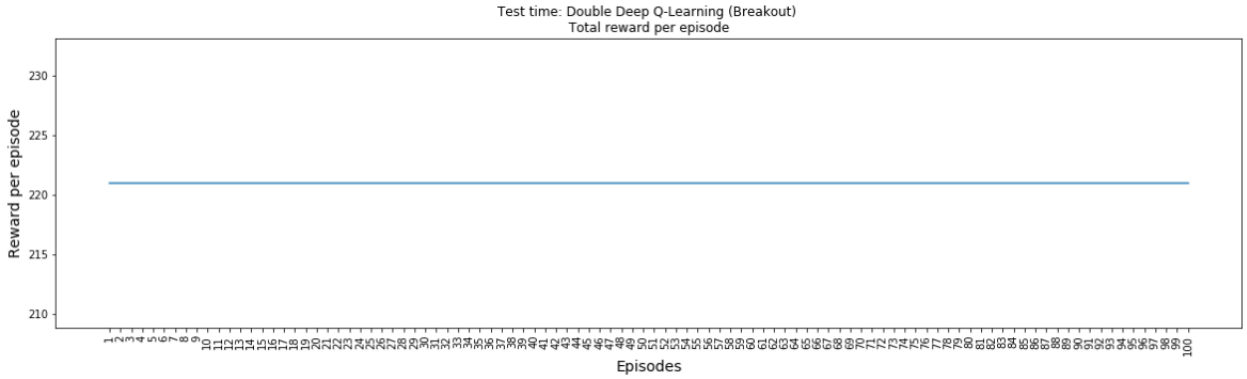

Figure 35: Double DQN Breakout Test results

The episode decay graph is short because the training was terminated before the value set for total episodes as 50000. Epsilon decay graph can be seen in the figure 36.

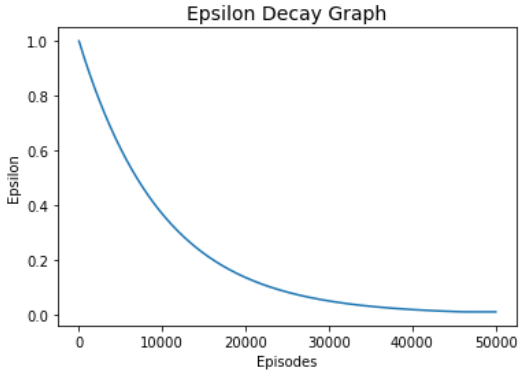

Figure 36: Breakout Double DQN - Epsilon Decay Graph

4.3. Dueling DQN:

In the below graph we can see that the graph for total reward per episode does oscillate a lot but it does reach the maximum reward multiple times. Here we have set an exit criterion that if the average reward for the last 20 episodes is greater than 200 then stop the training. This is happens near 35000 odd episodes. The batch size we have used is 32. In the average time-steps per 50 episodes we can see there is a rise when the again is trying to learn and then decreases gradually and stabilizes below 600 steps where it achieves the maximum rewards multiple times.

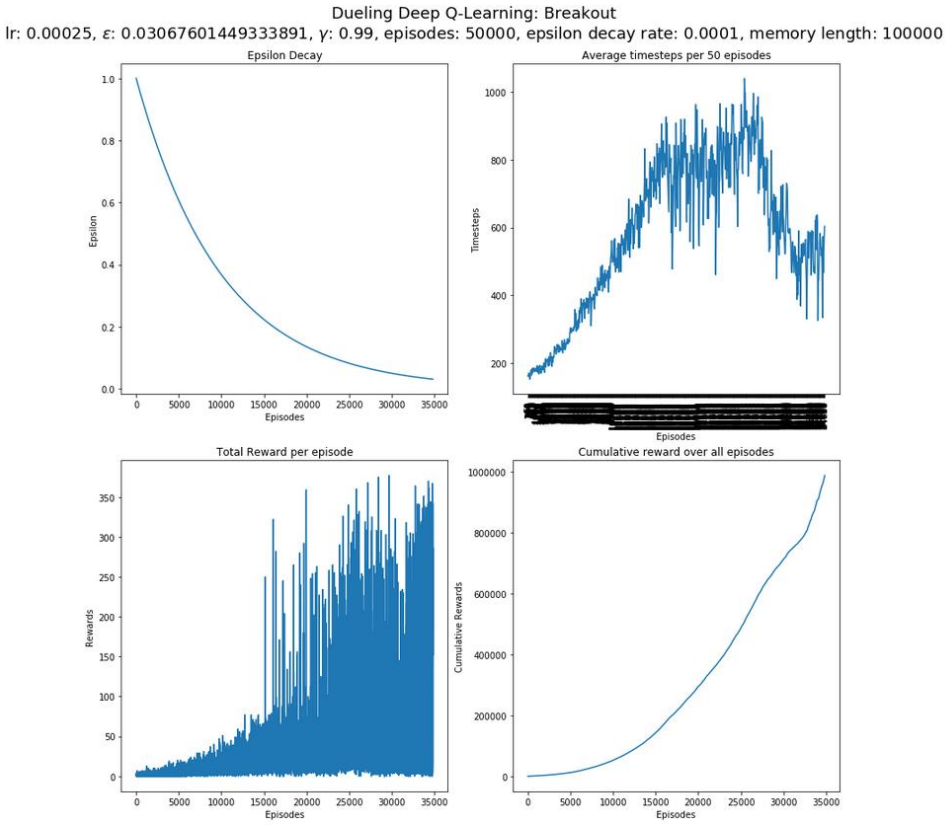

Figure 37: Dueling DQN results on Breakout

After training, evaluating the agent:
The above graph (Figure 38) showcases the total reward per episode for 100 episodes. The agent learns the optimal policy and follows the path to obtain the maximum reward.

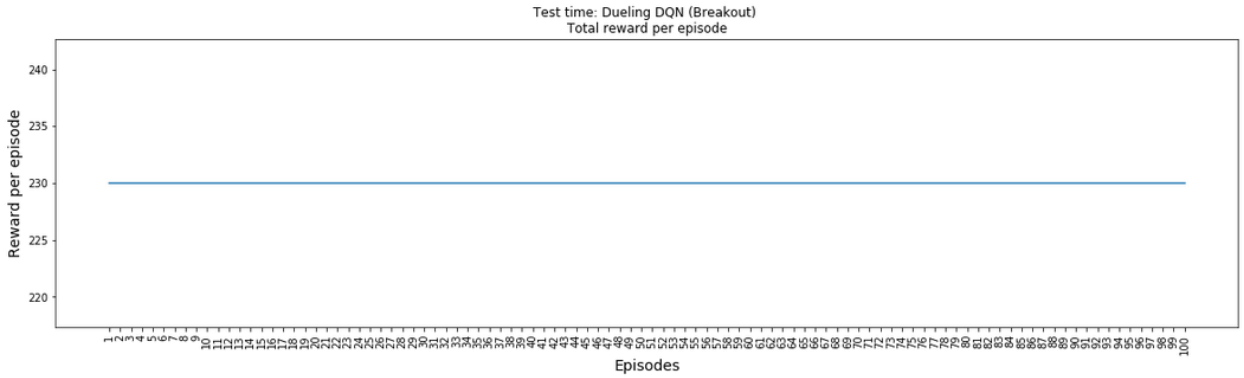

Figure 38: Testing results on Breakout using Dueling DQN

5. Comparison of different algorithms:

5.1. Grid World Environment:

In the below graphs we can compare the results of all the four algorithms on grid world environment. Grid World being the simplest environment all the algorithms are able to converge faster without need of hyper parameter tuning or excessive time to train. DQN has converged the quickest compared to all. Double DQN and Dueling DQN have similar results; both converge after 600 odd episodes. Whereas in A2C there was a plateau at –40 for some time and upon further exploring the agent learnt to converge towards the maximum reward. All the four algorithms were able to produce stable results and solve the environment. Upon testing the agent as we have seen in the graphs above, the agent was able to receive the maximum possible reward.

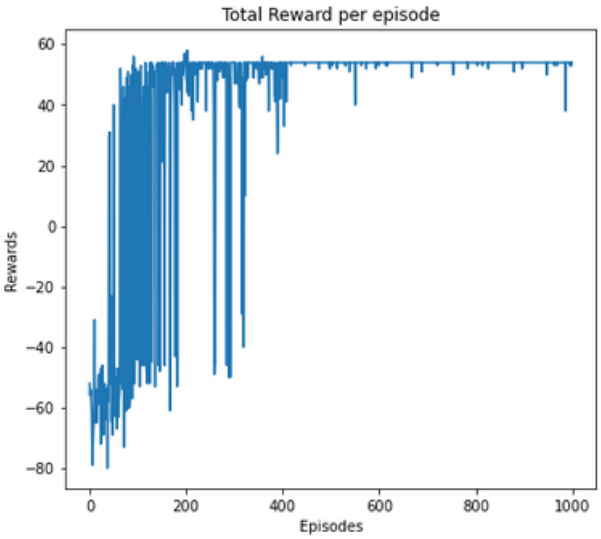

Figure 39: DQN

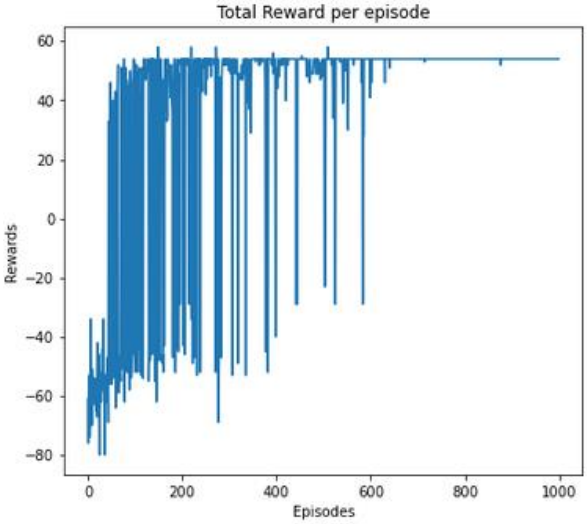

Figure 40: Double DQN

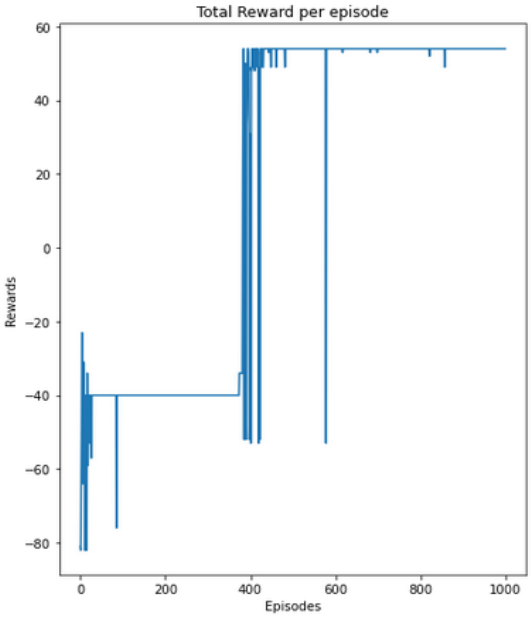

Figure 41: A2C

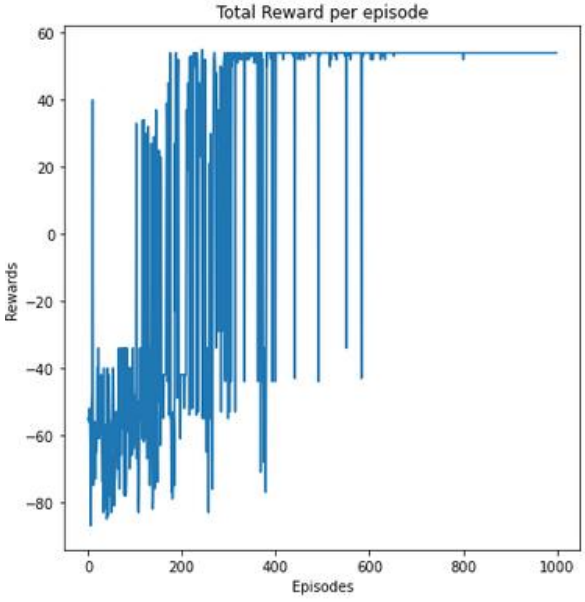

Figure 42: Dueling DQN

5.2. CartPole-v1:

In the below graphs we can observe that agent was able to receive the maximum reward using each algorithms. Compared to DQN, Double DQN produced gradual and stable increase in rewards and solved the environment. But in A2C and Dueling DQN we can observe that the agent could reach the maximum reward multiple times but could not maintain the same. This can be due to the fact that these algorithms being more complex or due to bad samples in the batch while training or in general the algorithms take more time to converge and find the optimal policy. Nonetheless all the four algorithms are able to converge and find the optimal policy. Likewise during testing as shown above all the four algorithms are above to receive the maximum reward.

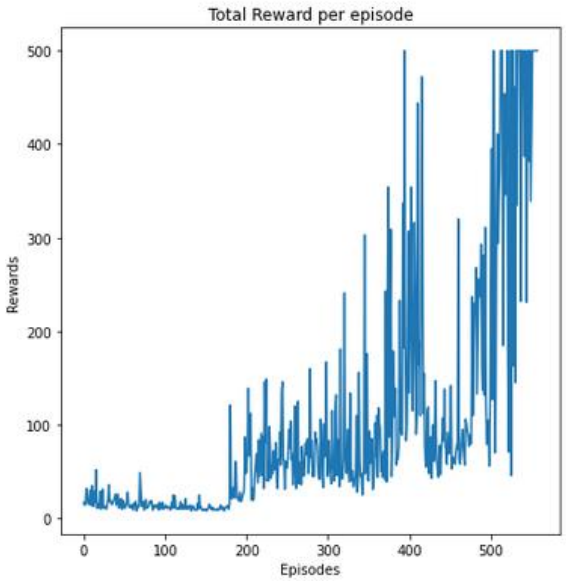

Figure 43: DQN

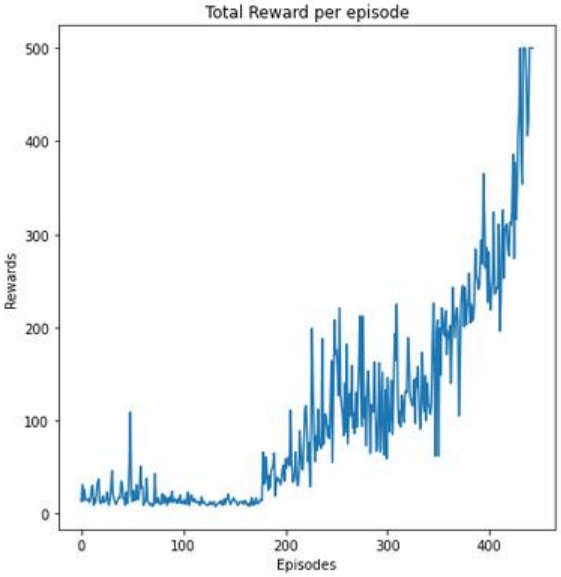

Figure 44: Double DQN

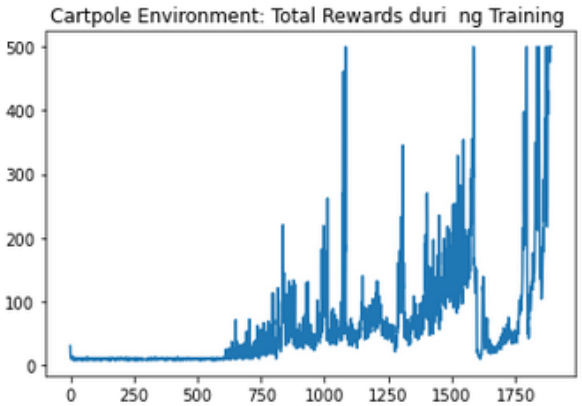

Figure 45: A2C

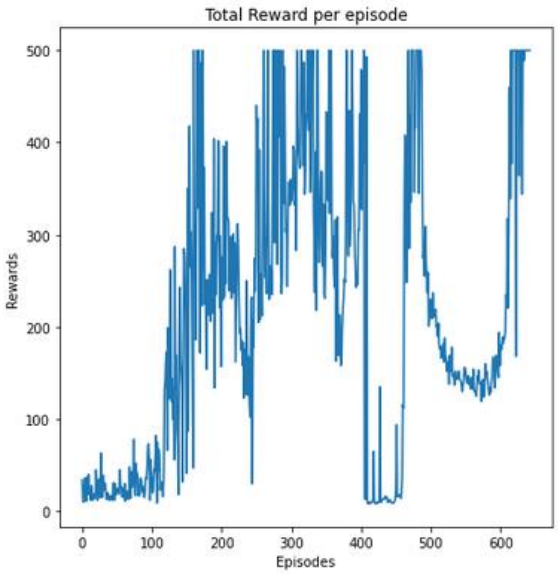

Figure 46: Dueling DQN

5.3.Acrobot-v1:

Out of all the algorithms we can observe that A2C has converged better than any other algorithm with fewer fluctuations. This could be due to the fact that DQN family algorithms which are value based and use maximization operation to find the best action may be the problem. The maximization bias issue, weights update in a bad fashion may cause this issue as well. A2C learns the policy and the value function both, which might help to learn the optimal policy in a better way than only calculating the value function.

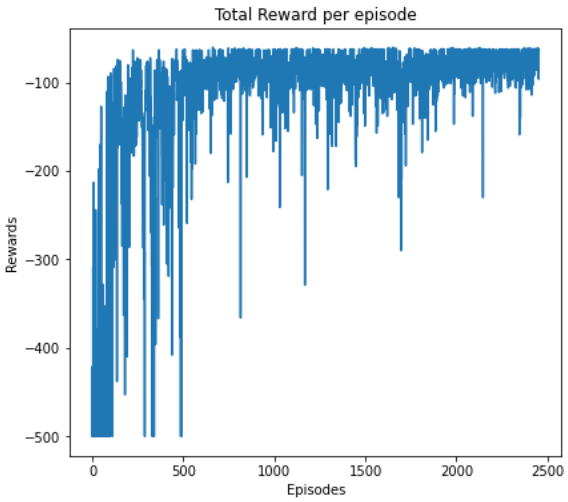

Figure 47: DQN

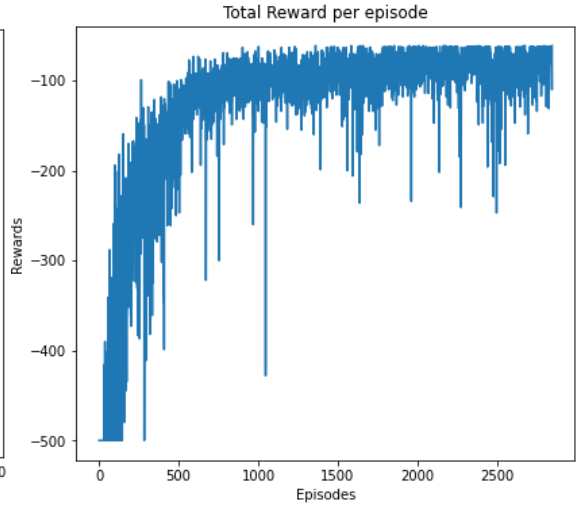

Figure 48: Double DQN

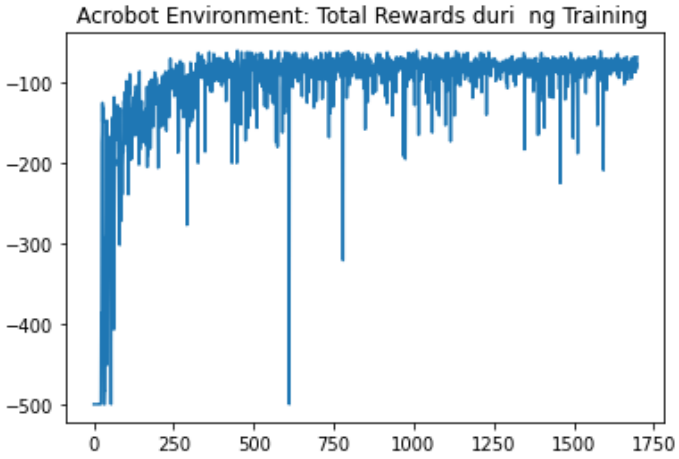

Figure 49: A2C

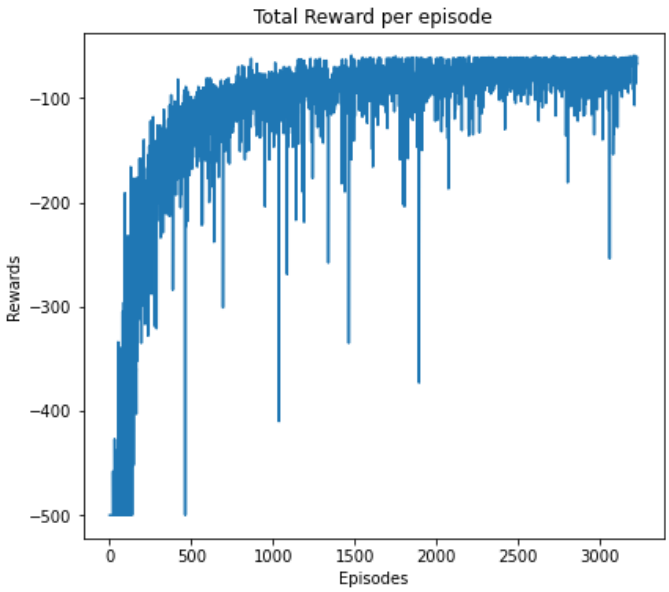

Figure 50: Dueling DQN

5.4.Breakout:

In breakout the story is slight different. Here we can see that the agent is able to receive the reward at which it is considered the environment to be solved but it is unable to maintain the same. This is primarily due to the fact of lower batch size. We have used batch size 32 because anything above that we repeatedly got *"Cuda Out of Memory"* error. Empirically it has been shown that with higher batch size the result stabilize but due to the hardware limitation we weren't able to do the same.

Compared to all the algorithms Double DQN does stabilize to a degree but often drop down. The graph for all gradually increases towards the reward we would like i.e. above 200. Breakout is a more complex environment with infinite number of permutations ranging from convolutional neural network size, weights initializations, and number of nodes in each layer, strides, padding; further more batch size, number of episodes, learning rate, optimizer, and many more. It is very difficult to try so many permutations but upon trial of tuning various parameters expect the batch size we tried to tune others and make the agent learn. The agent was able to learn and receive the optimal reward. Hence, hereby we conclude the agent would perform better and stabilize with trial higher batch size since these same algorithms are able to converge on other environments as well.

For A2C results refer to the technical difficulties section for detail results.

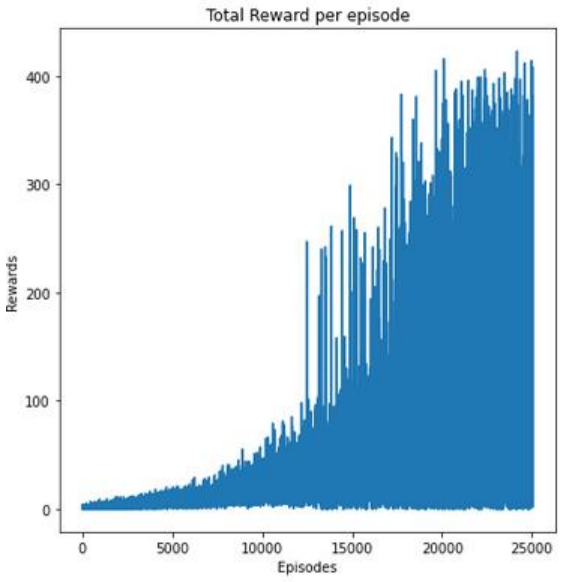

Figure 51: DQN

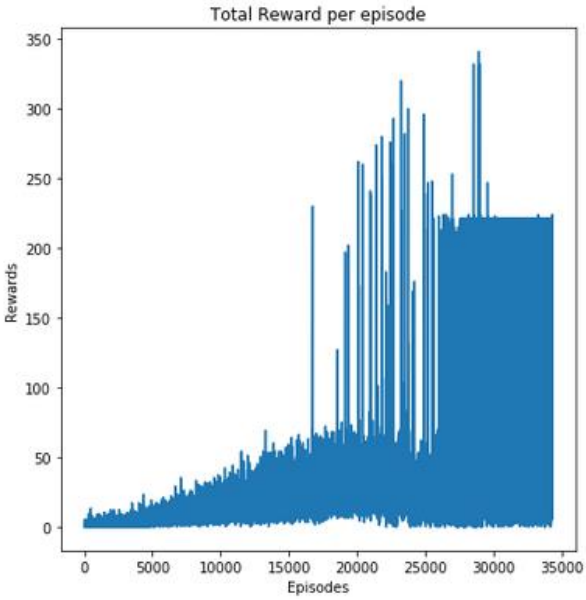

Figure 52: Double DQN

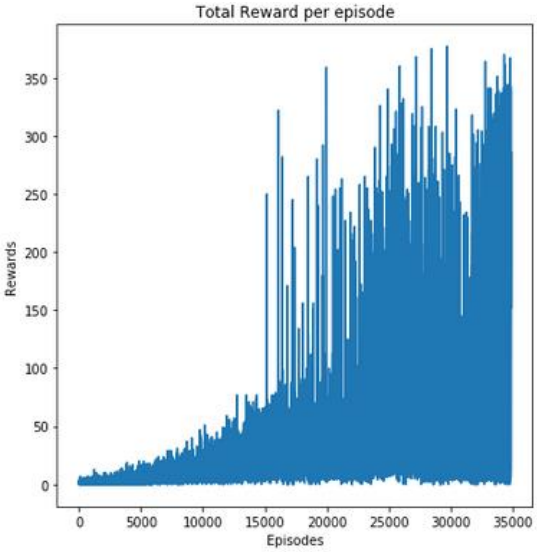

Figure 53: Dueling DQN

Technical Difficulties (A2C on Breakout):

1. Batch size + GPU:
    a. We have tried to use multiple batch size and use GPU while training but we always got "*Cuda out of Memory error*".
    b. The batch size used are: 32, 16, 8, 4, 2
    c. Please find the screenshots for the same

Figure 54: Batch size 32

Figure 45: Batch size 16

```
  0%|          | 0/50000 [00:00<?, ?it/s]
Batch size: 8
  1%|          | 441/50000 [05:34<9:32:10,  1.44it/s]
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-10-dbebf849690e> in <module>
     45
     46         if timesteps % train_timestep == 0:
---> 47             agent.train_model()
     48
     49         total_reward += reward

<ipython-input-7-bba454562e02> in train_model(self)
    114
    115             self.critic_network.optimizer.zero_grad()
--> 116             critic_loss.backward(retain_graph=True)
    117             self.critic_network.optimizer.step()
    118

/projects/academic/courses/cse546f21/anujarag/anaconda3/lib/python3.7/site-packages/torch/_tensor.py in backward(self, gradien
t, retain_graph, create_graph, inputs)
    305                 create_graph=create_graph,
    306                 inputs=inputs)
--> 307         torch.autograd.backward(self, gradient, retain_graph, create_graph, inputs=inputs)
    308
    309     def register_hook(self, hook):

/projects/academic/courses/cse546f21/anujarag/anaconda3/lib/python3.7/site-packages/torch/autograd/__init__.py in backward(tens
ors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    154     Variable._execution_engine.run_backward(
    155         tensors, grad_tensors_, retain_graph, create_graph, inputs,
--> 156         allow_unreachable=True, accumulate_grad=True)  # allow_unreachable flag
    157
    158

RuntimeError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0; 15.78 GiB total capacity; 14.02 GiB already allocated; 1
1.50 MiB free; 14.40 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb
to avoid fragmentation.  See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Figure 55: Batch size 8

```
  0%|          | 0/50000 [00:00<?, ?it/s]
Batch size: 4
  1%|          | 440/50000 [05:26<9:40:05,  1.42it/s]
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-10-dbebf849690e> in <module>
     45
     46         if timesteps % train_timestep == 0:
---> 47             agent.train_model()
     48
     49         total_reward += reward

<ipython-input-7-bba454562e02> in train_model(self)
    114
    115             self.critic_network.optimizer.zero_grad()
--> 116             critic_loss.backward(retain_graph=True)
    117             self.critic_network.optimizer.step()
    118

/projects/academic/courses/cse546f21/anujarag/anaconda3/lib/python3.7/site-packages/torch/_tensor.py in backward(self, gradien
t, retain_graph, create_graph, inputs)
    305                 create_graph=create_graph,
    306                 inputs=inputs)
--> 307         torch.autograd.backward(self, gradient, retain_graph, create_graph, inputs=inputs)
    308
    309     def register_hook(self, hook):

/projects/academic/courses/cse546f21/anujarag/anaconda3/lib/python3.7/site-packages/torch/autograd/__init__.py in backward(tens
ors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    154     Variable._execution_engine.run_backward(
    155         tensors, grad_tensors_, retain_graph, create_graph, inputs,
--> 156         allow_unreachable=True, accumulate_grad=True)  # allow_unreachable flag
    157
    158

RuntimeError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0; 15.78 GiB total capacity; 14.01 GiB already allocated; 1
9.50 MiB free; 14.39 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb
to avoid fragmentation.  See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Figure 56: Batch size 8

```
  0%|          | 0/50000 [00:00<?, ?it/s]
Batch size: 2
  1%|          | 446/50000 [05:15<10:26:12,  1.32it/s]
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-20-dbebf849690e> in <module>
     45
     46         if timesteps % train_timestep == 0:
---> 47             agent.train_model()
     48
     49         total_reward += reward

<ipython-input-15-bba454562e02> in train_model(self)
    114
    115             self.critic_network.optimizer.zero_grad()
--> 116             critic_loss.backward(retain_graph=True)
    117             self.critic_network.optimizer.step()
    118

/projects/academic/courses/cse546f21/anujarag/anaconda3/lib/python3.7/site-packages/torch/_tensor.py in backward(self, gradien
t, retain_graph, create_graph, inputs)
    305                 create_graph=create_graph,
    306                 inputs=inputs)
--> 307         torch.autograd.backward(self, gradient, retain_graph, create_graph, inputs=inputs)
    308
    309     def register_hook(self, hook):

/projects/academic/courses/cse546f21/anujarag/anaconda3/lib/python3.7/site-packages/torch/autograd/__init__.py in backward(tens
ors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    154     Variable._execution_engine.run_backward(
    155         tensors, grad_tensors_, retain_graph, create_graph, inputs,
--> 156         allow_unreachable=True, accumulate_grad=True)  # allow_unreachable flag
    157
    158

RuntimeError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0; 15.78 GiB total capacity; 14.01 GiB already allocated; 1
7.50 MiB free; 14.39 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb
to avoid fragmentation.  See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Figure 57: Batch size 2

2. CPU:
    a. We tried running the same model keeping everything the same on CPU
    b. The agent was not able to learn even after long time of training
    c. Below are the results for the same:

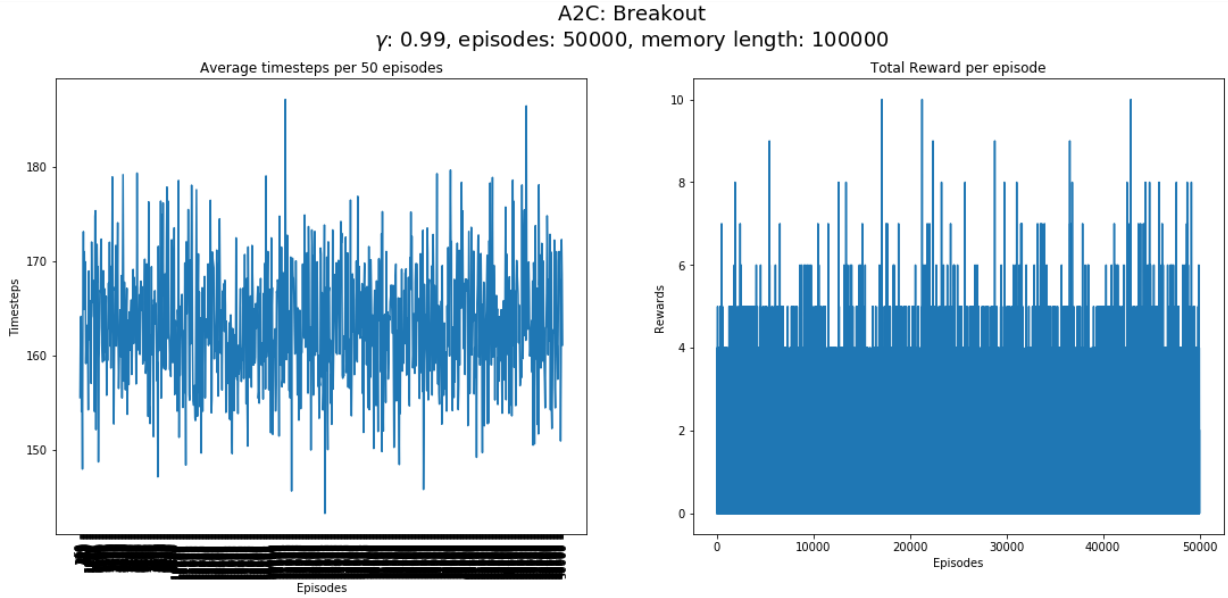

Figure 58: A2C Results on breakout using CPU

    d. Above we can see that the agent hasn't learnt.
    e. Further we tried to tweak the network parameter to make the agent learn
3. Network Size/Parameters + GPU:
    a. Next we tried to reduce the parameters in the network and train the agent so not to get the error
    b. The input image size is 84x84 which we reduce it 20x20 then 9x9 and finally to 7x7 which then is connected to fully connected layers

c. Reducing the images sizes further didn't help the learning since image is already very small to begin with
d. Hence we tried reducing the number of kernels to find the features
e. Figure 59 shows the original network we used for DQN, Double DQN and Dueling DQN

```
class DQNnet(nn.Module):
    '''
    DQNnet is a class that defines the Q-net
    '''
    def __init__(self, input_size, output_size, lr):
        super(DQNnet, self).__init__()
        self.linear_model = nn.Sequential(OrderedDict([
            ('conv1', nn.Conv2d(4, 32, kernel_size=8, stride=4)),
            ('relu1', nn.ReLU()),
            ('conv2', nn.Conv2d(32, 64, 4, 2)),
            ('relu2', nn.ReLU()),
            ('conv3', nn.Conv2d(64, 64, 3, 1)),
            ('relu3', nn.ReLU()),
            ('flatten1', nn.Flatten()),
            ('dense1', nn.Linear(7*7*64, 512)),
            ('relu4', nn.ReLU()),
            ('dense2', nn.Linear(512, output_size))]
        ))
        self.optimizer = optim.AdamW(self.parameters(),lr=lr)

    def forward(self, x):
        '''
        Perform forward pass in the network
        '''
        x = self.linear_model(x)
        return x
```

Figure 59: Original network structured of network used in DQN, Double DQN and Dueling DQN

f. Please find below the observations we got for the same:
    i. Here we have reduce the number of kernels by half:

```
class Critic_Network(nn.Module):
                .
    def __init__(self, input_size, output_size, learning_rate, path = 'models/'):
        super(Critic_Network, self).__init__()
        self.path = path

        self.linear_model = nn.Sequential(OrderedDict([
            ('conv1', nn.Conv2d(4, 16, kernel_size=8, stride=4)),
            ('relu1', nn.ReLU()),
            ('conv2', nn.Conv2d(16, 32, 4, 2)),
            ('relu2', nn.ReLU()),
            ('conv3', nn.Conv2d(32, 32, 3, 1)),
            ('relu3', nn.ReLU()),
            ('flatten1', nn.Flatten()),
            ('dense1', nn.Linear(7*7*32, 256)),
            ('relu4', nn.ReLU()),
            ('dense2', nn.Linear(256, output_size))]
        ))
        self.optimizer = optim.AdamW(self.parameters(),lr=lr)
```

Figure 60: Network structure where number of kernels is halved

    ii. Below the screenshot where we get the error:

```
  1%|          | 595/50000 [07:51<10:52:24,  1.26it/s]
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-9-fbdea138307e> in <module>()
     44
     45         if timesteps % train_timestep == 0:
---> 46             agent.train_model()
     47
     48         total_reward += reward

<ipython-input-6-5fcc7af193f1> in train_model(self)
     84                 actions.append(action)
     85                 rewards.append(reward)
---> 86                 next_states = torch.cat((next_states, next_state)).float().to(self.device)
     87                 dones.append(done_boolean)
     88                 log_probability.append(log_prob)

RuntimeError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0; 15.90 GiB total capacity; 14.84 GiB already allocated; 39.75 MiB free; 14.85 GiB reserved in tot
al by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid fragmentation.  See documentation for Memory Management and PYTORCH_CU
DA_ALLOC_CONF
```

Figure 61: Error for A2C on Breakout with number of kernels halved

g. Now we reduced the kernel size by one forth:
   i. Here the agent does run without errors but doesn't learn anything

```python
class Critic_Network(nn.Module):

    def __init__(self, input_size, output_size, learning_rate, path = 'models/'):
        super(Critic_Network, self).__init__()
        self.path = path

        self.linear_model = nn.Sequential(OrderedDict([
            ('conv1', nn.Conv2d(4, 8, kernel_size=8, stride=4)),
            ('relu1', nn.ReLU()),
            ('conv2', nn.Conv2d(8, 16, 4, 2)),
            ('relu2', nn.ReLU()),
            ('conv3', nn.Conv2d(16, 16, 3, 1)),
            ('relu3', nn.ReLU()),
            ('flatten1', nn.Flatten()),
            ('dense1', nn.Linear(7*7*16, 128)),
            ('relu4', nn.ReLU()),
            ('dense2', nn.Linear(128, output_size))]
        ))
        self.optimizer = optim.AdamW(self.parameters(),lr=lr)

    def forward(self, x):
        x = self.linear_model(x)
        return x
```

Figure 62: Network with number of kernels reduced by one forth

   ii. Below are the results for the same:

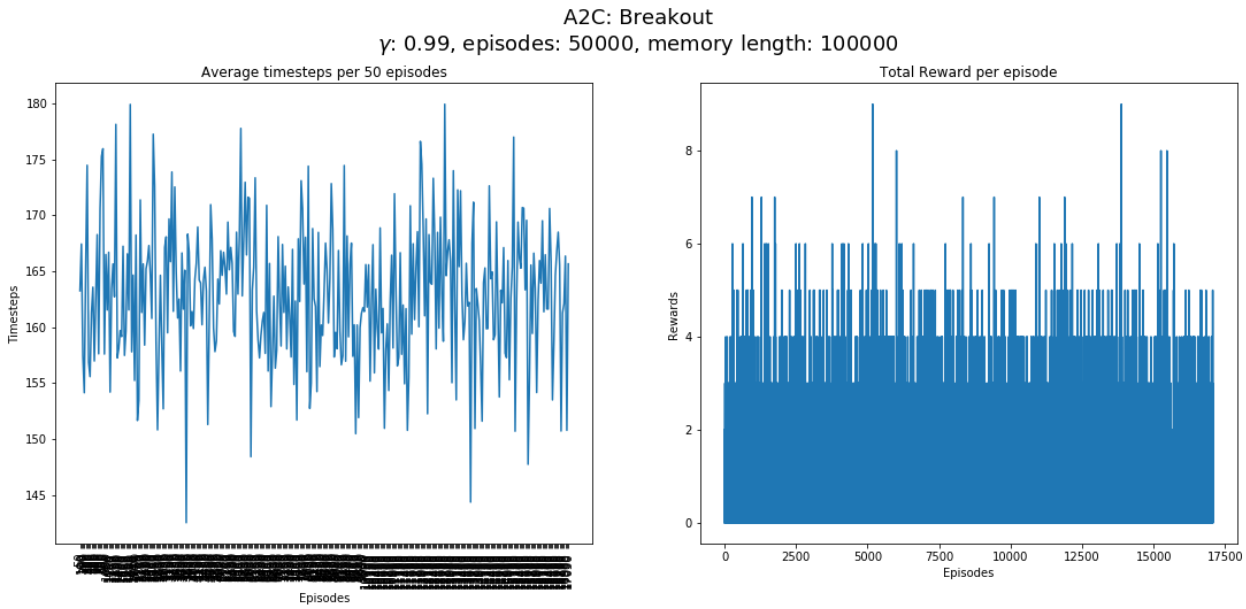

Figure 63: Results of A2C with number of kernel reduced by one forth

h. Based on our experiments below are the observations we have found:
   i. The network structure we used for other algorithms has more number of kernels which are required by the network to learn the features to make the agent learn.
   ii. Even though reducing the number of kernels doesn't give us any error but it also doesn't help the agent learn.
   iii. While on GPU it is difficult to train since for any batch size we get "*Cuda out of memory error*".
   iv. The same A2C algorithms successfully run on other environments and produces fruitful results while on breakout it is not learning

i. Hence, based on this, we observe the algorithm is proper but due to lack of memory resources we were not able to train our agent on breakout using A2C algorithm

Conclusion:

All the four algorithms can be used to solve all the four environments listed above but the performance of both the algorithms different not only on the same environment but even in between the four environments.

Normally we can see that Double DQN performance better than DQN on any of the four environments and produces much stable results. A2C and Dueling DQN are at par with the other two but take little more time and resources to train. On complex environment like breakout Dueling DQN starts to reach the optimal reward quicker than DQN and Double DQN.

Also, considering different parameters to tune for each algorithm greatly differs for each environment. As the environment gets more complex the tuning of the parameters play a crucial role on the performance of the agent. Also, as the environment gets complex the time for the agent to learn also differs for each algorithm.

Hence, algorithms performance differs not only on different environment but even in the same environment. Finding better algorithms can greatly improve the performance of the agent and exploring them is our next goal for our project.

Contribution Summary

Team Member	Assignment part	Contribution (%)
Sagar Jitendra Thacker	DQN and A2C on all four environments and report + presentation	50%
Anuja Raghunath Katkar	Double DQN and Dueling DQN on all four environments and report + presentation	50%

Citations:

1. Playing Atari with Deep Reinforcement Learning