Assignment 3 Report

Course Code: AI3000
Course Name: Reinforcement Learning

Ahmik Virani
Roll Number: ES22BTECH11001
IIT Hyderabad

31 October, 2025

Contents

Chapter 1

Problem 1 : Deep Q Learning : Programming

1.1 Part(a)

In this part, we had develop a random agent.

1.1.1 Pong-v5

- State Space Size = 210
  - obs_type="rgb" -> observation_space=Box(0, 255, (210, 160, 3), np.uint8)
  - obs_type="ram" -> observation_space=Box(0, 255, (128,), np.uint8)
  - obs_type="grayscale" -> Box(0, 255, (210, 160), np.uint8), a grayscale version of the q"rgb" type
- Action Space Size = 6
  - 0: NOOP
  - 1: FIRE
  - 2: RIGHT
  - 3: LEFT
  - 4: RIGHTFIRE
  - 5: LEFTFIRE
- Reward: You get score points for getting the ball to pass the opponent's paddle. You lose points if the ball passes your paddle.
- Termination Condition: If any player gets 21 points

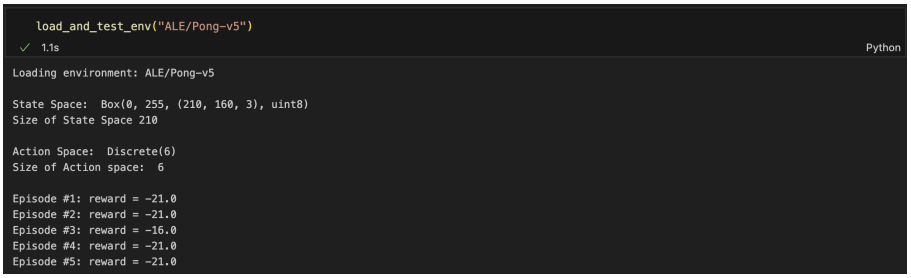

Figure 1.1: Running Pong-v5 for 5 episodes using random agent

1.1.2 MountainCar-v0

- State Space Size = 2
  - position of the car along the x-axis: range = [-1.2, 0.6]
  - velocity of the car: range = [-0.07, 0.07]
- Action Space Size = 3
  - 0: Accelerate to the left
  - 1: Don't accelerate
  - 2: Accelerate to the right
- Reward: -1 for each timestep
- Termination Condition:
  - Termination: The position of the car is greater than or equal to 0.5 (the goal position on top of the right hill)
  - Truncation: The length of the episode is 200.

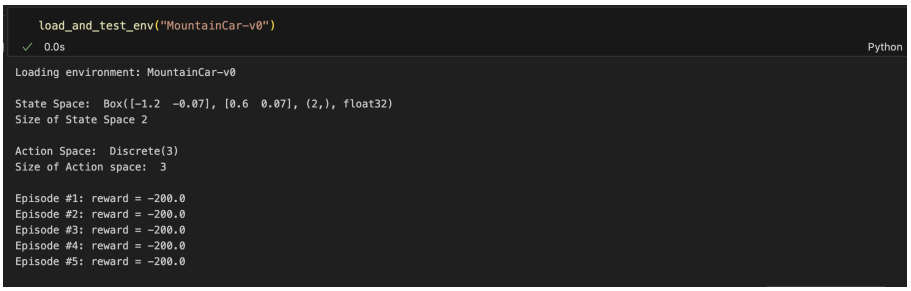

Figure 1.2: Running MountainCar-v0 for 5 episodes using random agent

1.2 Part(b)

1.2.1 Pong-v5

Network Structure

- I have implemented a replay buffer with FIFO mechanism

- For the Deep Q-Learning Network, I have used a Convnet.[1]
  - 3 Convolution Layers
  - Followed by 4 fully connected layers
  - Relu activation was used for all the intermediate layers
  - Initialized weights using Kaiming Normal for the Convolution Layers
  - Initialized weights using Xavier Normal for the fully connected layers
- Used Polyad Averaging to update the target network ($\tau = 0.99$)
- The hyperparameters I have used are:
  - Discount rate, $\gamma = 0.99$
  - Batch Size $B = 128$
  - Replay Buffer size (in units of number of transitions (s, a, a', r) it can store) $N = 500000$
  - Learning rate = 2.5e-4
  - Uniform decay of ϵ from 1 to 0.05 over 3000000 steps, then keeping ϵ constant
  - The number of training episodes: $M = 3000$
- Used Huber loss as suggested in the lecture slides
- Used RMSprop optimizer as suggested in the paper
- Used Double DQN to avoid overestimation problem of Q values.
- Preprocessing of state:
  - Converted to grayscale
  - Resized the image to 84 x 84
  - Normalized so that values between [0,1]
  - When making a state, made a stack of 4 subsequent images, each being subtracted from the previous.
- I have pre-filled the Replay buffer with 20000 states before hand

[1] I have taken inspiration from YouTube Link, and make necessary changes as per the question

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 8, 41, 41]             520
            Conv2d-2          [-1, 16, 19, 19]           2,064
            Conv2d-3            [-1, 32, 8, 8]           8,224
            Linear-4                 [-1, 256]         524,544
            Linear-5                 [-1, 256]          65,792
            Linear-6                 [-1, 256]          65,792
            Linear-7                   [-1, 6]           1,542
================================================================
Total params: 668,478
Trainable params: 668,478
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.17
Params size (MB): 2.55
Estimated Total Size (MB): 2.83
----------------------------------------------------------------
```

Figure 1.3: Pong Network

Explaining the Network Structure

- A low learning rate along with a large number of episodes ensures that we converge well.
- The decay of ϵ is inspired from the V. Minh paper.
- Reason for a large replay buffer size:
  - As seen in class, it helps break correlations and avoid subsequent transitions from being selected
  - Since we can store large amount of actions, the agent can learn from the same action several times. This improves data efficiency.

Experimental Results

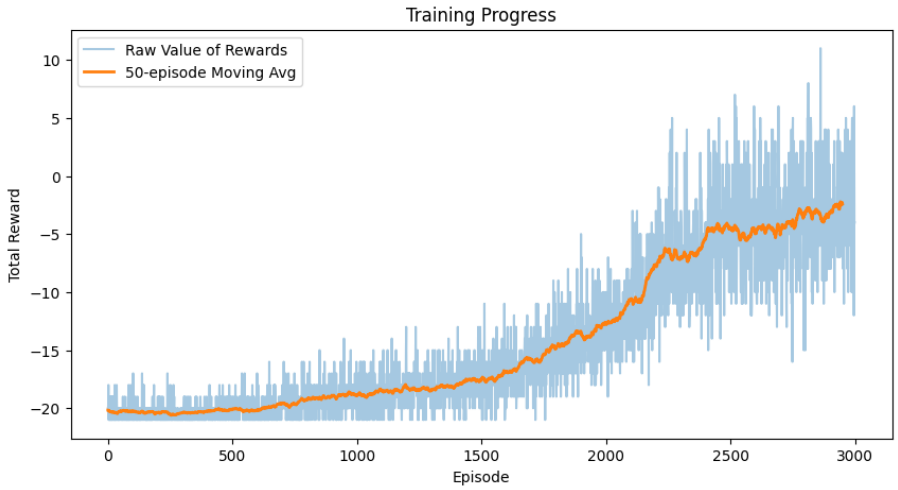

Figure 1.4: The reward earned over each episode of training

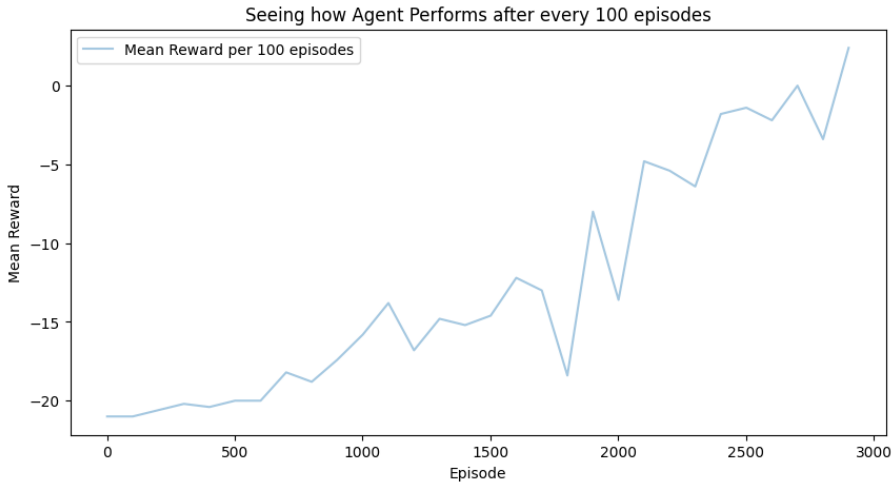

Figure 1.5: Average reward of 100 test episodes ran after every 100 episodes of training

- The first figure shows the reward earned per episode during the training phase. It also has a moving average curve to better see the trend.
- The second figure shows the average reward over 100 episodes I have run, after training on 100 episodes. These are basically tested set, different from the training set.
- Maximum Reward: 11.0

1.2.2 MountainCar-v0

Network Structure

- I have implemented a replay buffer with FIFO mechanism

- For the Deep Q-Learning Network, I have made a simple feedforward neural network [2]:
  - 2 hidden layers with Relu activation
  - Initialized weights using Xavier Normal for the hidden layers
  - Initialized weights uniformly, close to zero, for the output layer
- Used Polyak Averaging to update the target network ($\tau = 0.99$)
- The hyperparameters I have used are:
  - Discount rate, $\gamma = 0.99$
  - Batch Size $B = 128$
  - Replay Buffer size (in units of number of transitions (s, a, a', r) it can store) $N = 50000$
  - Learning rate = 5e-4
  - Uniform decay of ϵ from 1 to 0.05 over 600000 steps, then keeping ϵ constant
  - The number of training episodes : $M = 4000$
- Used Huber loss as suggested in the lecture slides
- Used RMSprop optimizer as suggested in the paper
- Used Double DQN to avoid overestimation problem of Q values.
- I have also done data normalization before giving input to the neural network
- I have pre-filled the Replay buffer with 1000 states before hand
- I have modified the reward to take into consideration the position of the car (More explanation later).

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 64]             192
            Linear-2                   [-1, 64]           4,160
            Linear-3                    [-1, 3]             195
================================================================
Total params: 4,547
Trainable params: 4,547
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.02
Estimated Total Size (MB): 0.02
----------------------------------------------------------------
```

Figure 1.6: MountainCar Network

[2]I have taken inspiration from YouTube Link, and make necessary changes as per the question. This is the same youtube link as I used for pong, adapted it to this environment

Explainationg of Network Structure

- A low learning rate along with a large number of episodes ensures that we converge well.
- The decay of ϵ is inspired from the V. Minh paper.
- Reason for a large replay buffer size:
  - As seen in class, it helps break correlations and avoid subsequent transitions from being selected
  - Since we can store large amount of actions, the agent can learn from the same action several times. This improves data efficieny.
- Reward Modification:
  - I gave a large reward (+100) if the car reaches the goal
  - I gave an additional reward for the absolute position of the car. I did not consider the direction because it would then it could get stuck at the local optimum and never reach the goal.

Experimental Results

First, for every 100 episodes, I was running my learnt deep Q-Learning Agent on 100 Episodes to see how much it is learning.

Figure 1.7: Average reward of 100 test episodes ran after every 100 episodes of training

From the above figure, we can see how the learning is improving overtime.

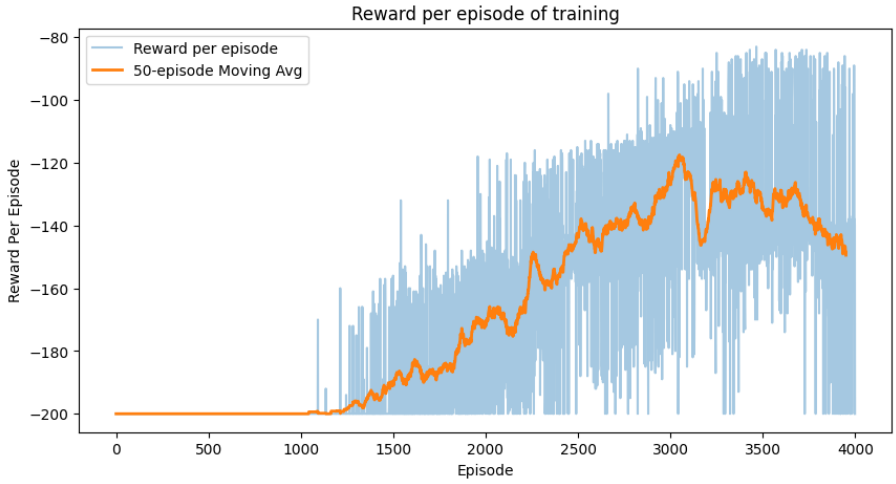

Figure 1.8: The reward earned over each episode of training

This graph shows the reward it has been earning over each episode during training, along with the moving average.

We see that the maximum reward is -83.

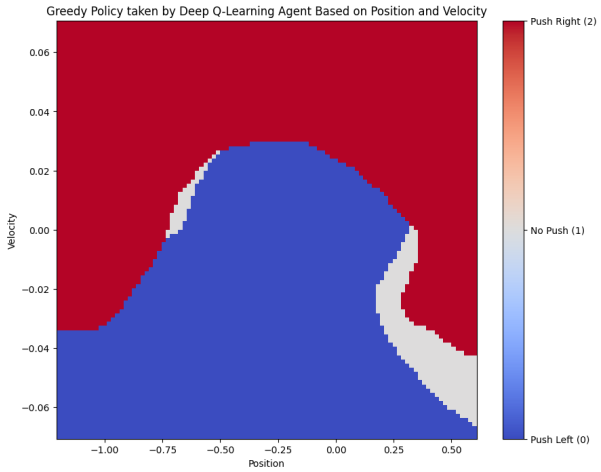

Figure 1.9: Greedy Action taken based on position and veloctity

This shows what action my learn Agent is taking depending on the position and velocity.

- Agent pushes right when it has large rightward velocity or position is right of the center. This signifies that Agent knows giving more push will increase momentum towards right
- Agent pushes left when it is already moving left and is left from the center. This will allow it to go further up left and gain momentum while coming back down so it can reach further to the right
- When the agent is closer to center and has less velocity, it is doing nothing. This may be due to the fact that this position may not really be encountered much and the agent did learn what to do in this scenario.

- Even the scenario when the position is very far right but the velocity is negative, it does nothing. Perhaps this state is not observed very often.

1.3 Part(c)

Please note that I have done this part only on MountainCar because the run time for Pong is very high exceeding 16 hrs as shown in figure 1.10.

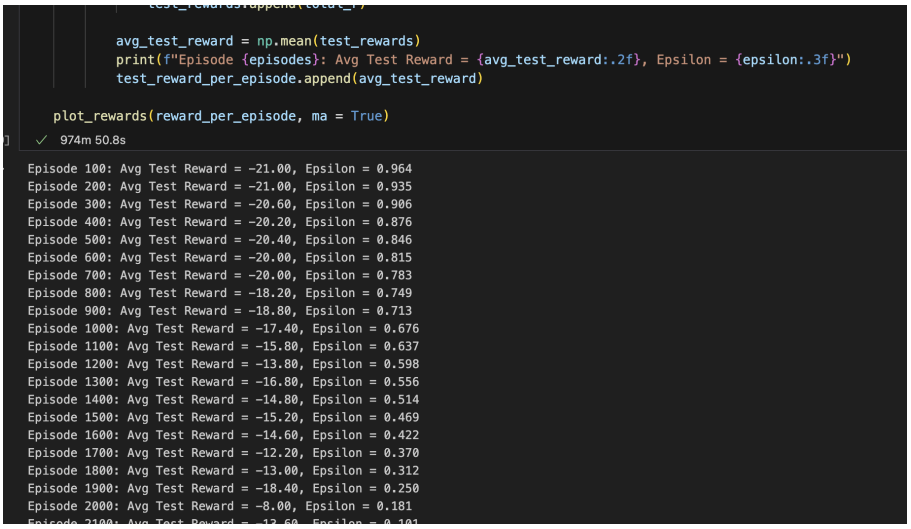

Figure 1.10: Long runtime of pong

Here I have compared how changing the neural network structure can affect the performance of the DQN Agent. I have compared hidden layer sizes of 32, 64 and 128.

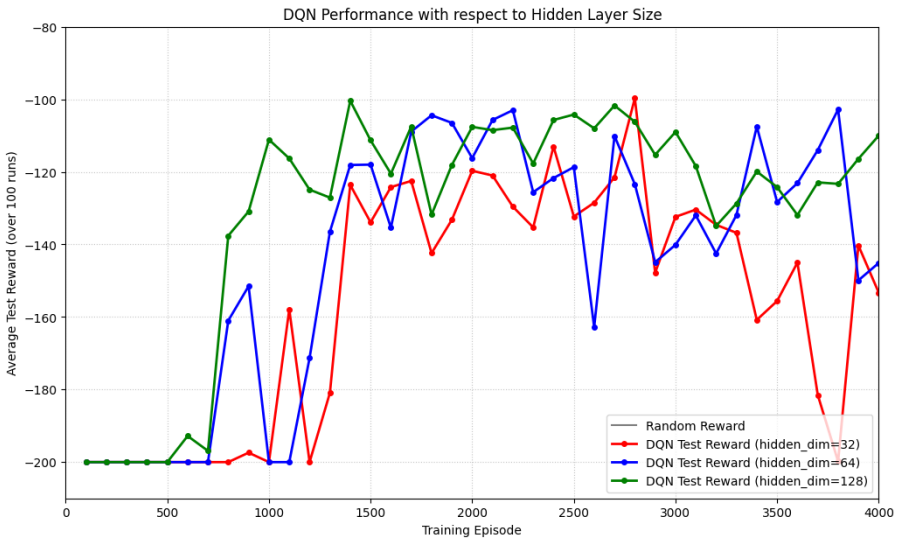

Figure 1.11: Comparing how hidden layer size affects DQN Agent performance, reward averaged over 100 episodes after every 100 episodes

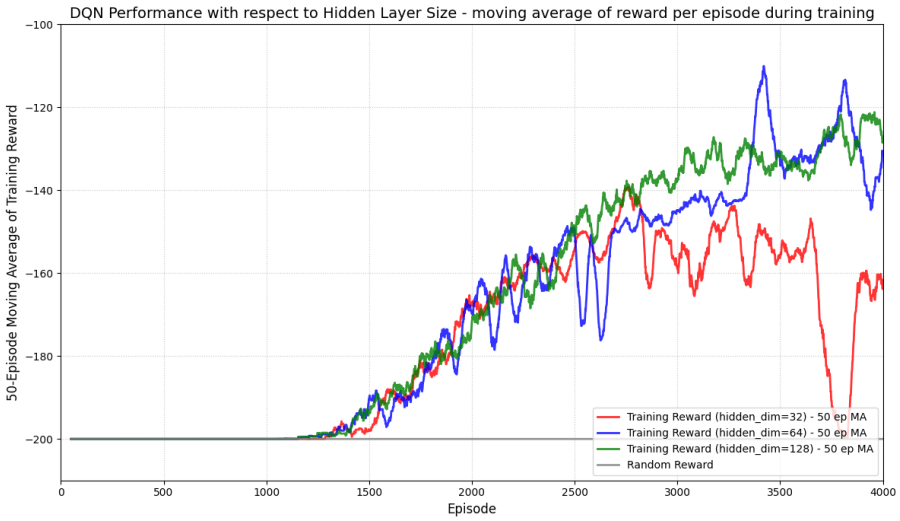

Figure 1.12: Comparing how hidden layer size affects DQN Agent performance, Moving Average of reward while training is going on

- Looking at this figure, we can see that higher hidden dimension is able to perform well and generalize, whereas when the exploration rate drop and exploitation phase takes over, the Agent with the smallest hidden layer actually performs very bad. This shows that more hidden neurons in this case is able to generalize well. But maybe when we increase the neural network size too much, it may overfit and not give very good results in testing.
- Also, it is very evident that the random agent is performing the worst and terminating after 200 episodes and never finishing the task.
- Conclusion, the more the hidden layer size upto 128, the better the agent performs.

Chapter 2

Problem 3 : Policy Gradient

2.1 Part(a)

In this part, we had develop a random agent.

2.1.1 Cartpole-v0

- State Space Size = 4
  - Cart Position: range = [-4.8, 4.8]
  - Cart Velocity: range = $[-\infty, \infty]$
  - Pole Angle: range = [-24°, 24°]
  - Pole Angular Velocity: range = $[-\infty, \infty]$
- Action Space Size = 2
  - 0: Push cart to the left
  - 1: Push cart to the right
- Reward: +1 is given for every step taken, including the termination step
- Termination Condition:
  - Termination: Pole Angle is greater than ±12°
  - Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
  - Truncation: Episode length is greater than 200

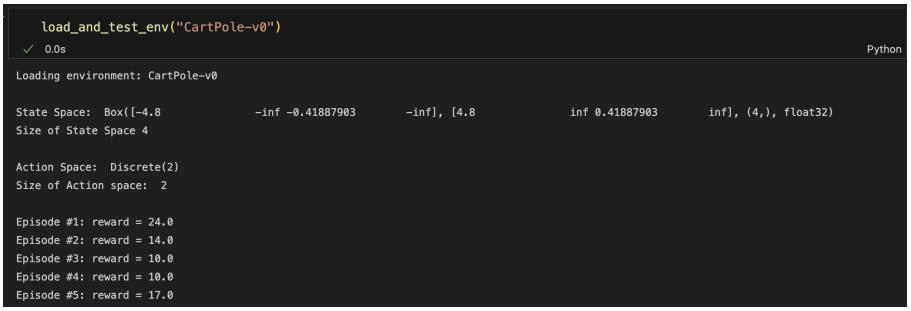

Figure 2.1: Running Cartpole-v0 for 5 episodes using random agent

2.1.2 LunarLander-v3

- State Space Size = 8
  - Lander Coordinates in x-axis: range = [-2.5, 2.5]
  - Lander Coordinates in y-axis: range = [-2.5, 2.5]
  - Lander Linear Velocity in x-axis: range = [-10, 10]
  - Lander Linear Velocity in y-axis: range = [-10, 10]
  - Angle: range = [-360°, 360°]
  - Angular Velocity: range = [-10, 10]
  - Leg 1 in Contact with Ground: range = True, False
  - Leg 2 in Contact with Ground: range = True, False
- Action Space Size = 4
  - 0: do nothing
  - 1: fire left orientation engine
  - 2: fire main engine
  - 3: fire right orientation engine
- Reward:
  - is increased/decreased the closer/further the lander is to the landing pad.
  - is increased/decreased the slower/faster the lander is moving.
  - is decreased the more the lander is tilted (angle not horizontal).
  - is increased by 10 points for each leg that is in contact with the ground.
  - is decreased by 0.03 points each frame a side engine is firing.
  - is decreased by 0.3 points each frame the main engine is firing.
  - The episode receive an additional reward of -100 or +100 points for crashing or landing safely respectively.
- Termination Condition:
  - the lander crashes (the lander body gets in contact with the moon);
  - the lander gets outside of the viewport (x coordinate is greater than 1);
  - the lander is not awake.

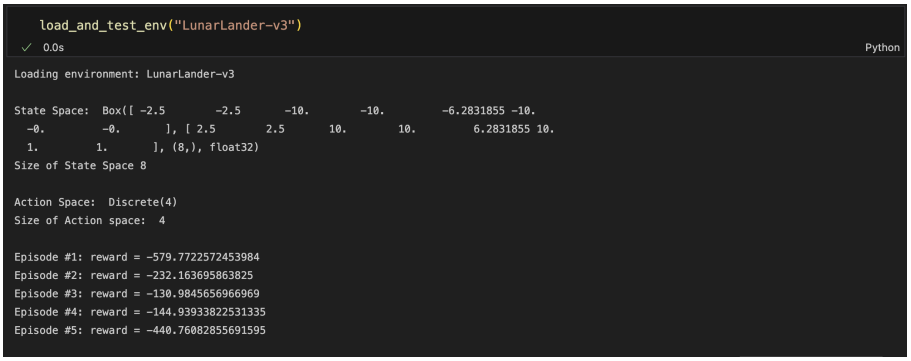

Figure 2.2: Running LunarLander-v3 for 5 episodes using random agent

2.2 Part(b)

The goal of this part was to implement policy gradient algorithm. [1]

- The policy Network used is a simple feedforward neural network with one hidden state (with 128 dimensions).
- The actiavtion functions used are Relu and Softmax
- For the advantage normalization part, the baseline function used is the average cumulative reward of the epsiodes. This ensures that no additional bias is added to the model.
- Since these are indefinite horizon settings but the termination/truncation is guaranteed to occur, I have kept discount factor $\gamma = 0.99$, which is close to 1, at the same time, if the episode goes very long, ensures that the reward does not explore and cause problems in back propagation.
- I have trained for 100 episodes
- The batch size I have used is 32 (The effect of changing batch size is studied in a later part of this question)
- The learning rate used was 0.01

2.2.1 CartPole-v0

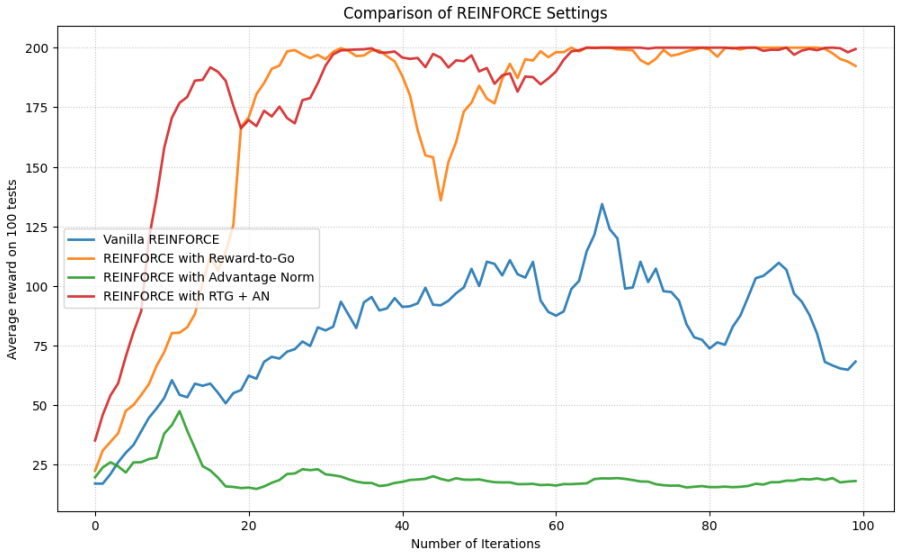

Figure 2.3: Comparing different settings of the REINFORCE Algorithm

From the figure we can see that

- Without Reward-to-go and without Advantage normalization, we are not reaching the optimum reward but still learning to do good enough to some extent

[1] This is inspired from this Youtube Link and modified as per the need of this question

- With only Reward-to-go, we do eventually reach a maximum reward and the curve seems to converge. This is likely due to the fact that there is lower variance in reward to go due to the fact that we are actually taking into consideration reward from each step seperately than the same reward for the entire episode. This allows better understanding of which reward at which step is actually better.
- With only advantage normalization, it is actually performing very poorly. The reason for this is in this setting, I have chosen baseline as the mean reward. And without reward to go, all the steps in the episode have the same reward, so effectively, subtracting the bias, the advantage term is always zero, and causes no gradients to flow.
- When both the advantage normalization and reward to go is enforced, we actually see the best result. This does not face the problem described earlier because cumulative reward starting from each step will be different. So the advantage term will not be zero. And with normalization, the neural network works better. So we get the benefits of both while taking away the drawback seen earlier.

In summary, in this setting the advantage noralization alone performs the worst. Then comes the vanilla implementation. Next is reward-to-go alone. The best is combining both reward-to-go and advantage normalization.

2.2.2 LunarLander-v3

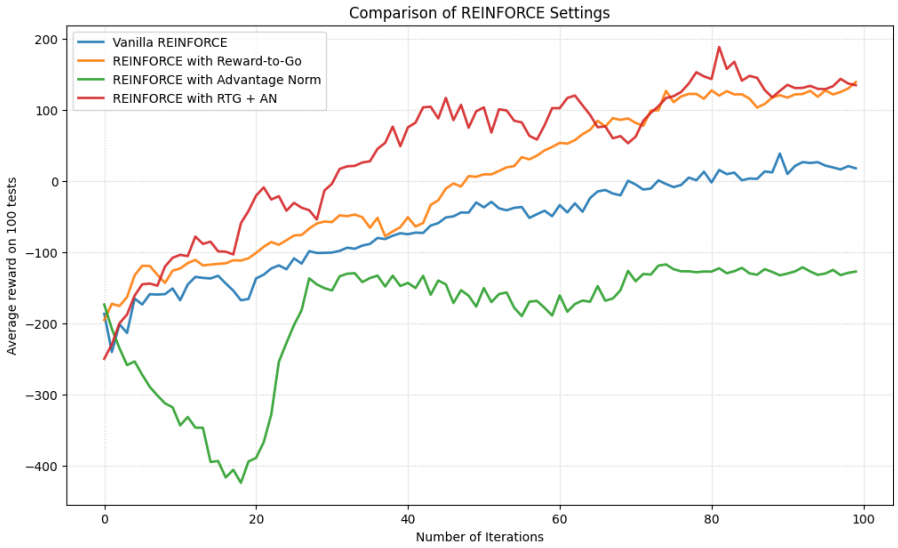

Figure 2.4: Comparing different settings of the REINFORCE Algorithm

Here, the results are very similar to the previous case, and the same explanation would hold.

2.3 Part(c)

2.3.1 CartPole-v0

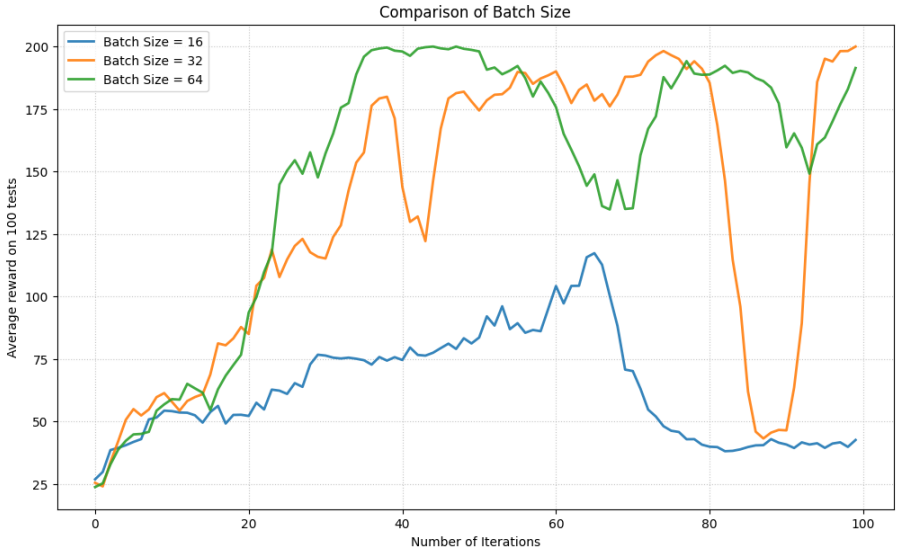

Figure 2.5: Comparing Effect of batch size on Vanilla REINFORCE Algorithm

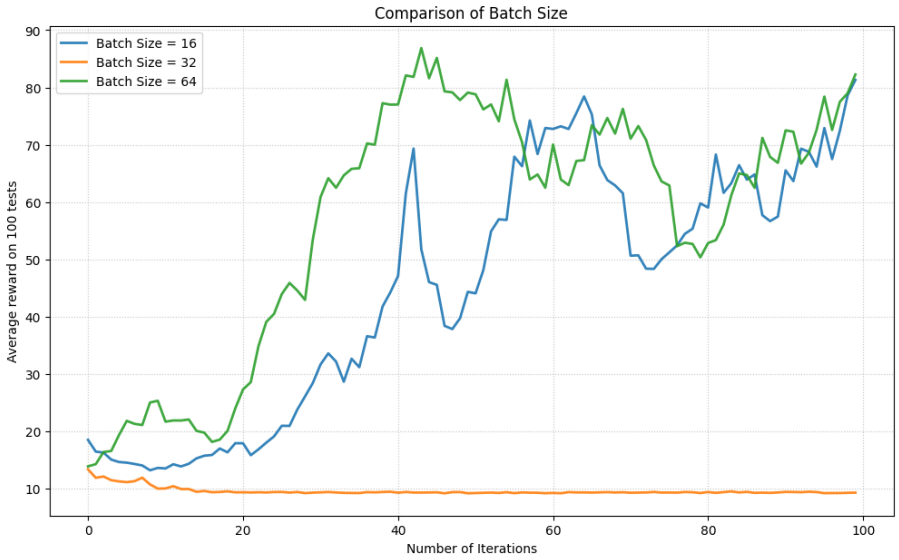

Figure 2.6: Comparing Effect of batch size on REINFORCE with Advantage Normalization

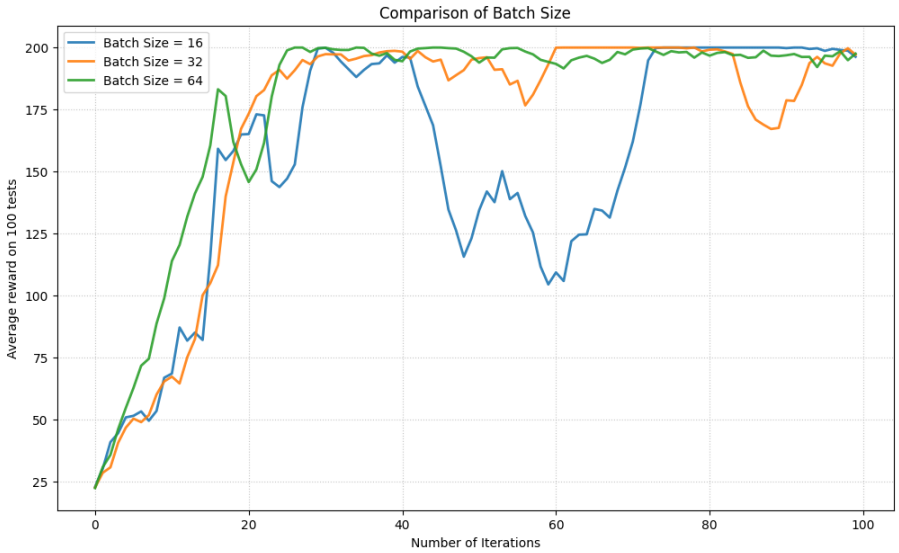

Figure 2.7: Comparing Effect of batch size on REINFORCE with Reward-to-go

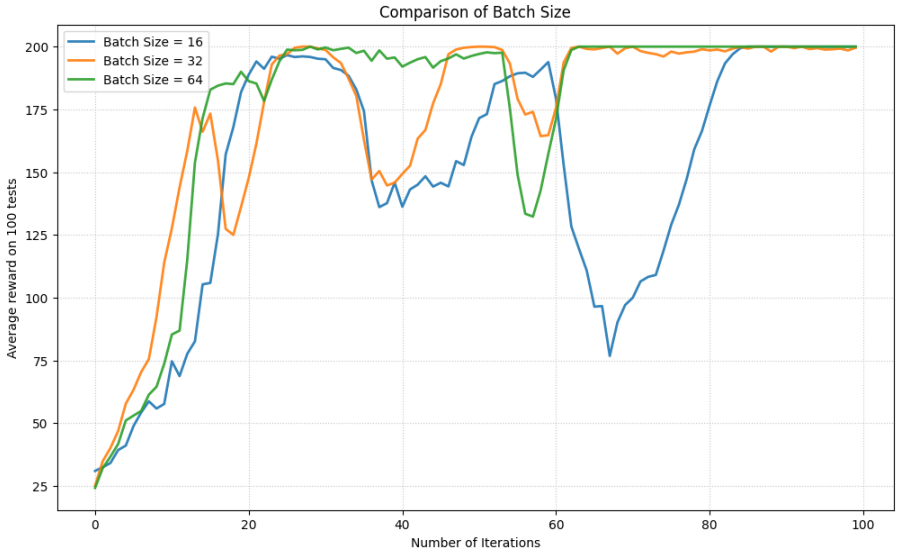

Figure 2.8: Comparing Effect of batch size on REINFORCE with both advantage normalization and reward-to-go

- Generally looking at the above curves, increase in batch size is leading to faster convergence.
- One reason for this is because as batch size increases, we use more samples to learn and back propogate. Thus, we could learn both from more samples, and potentially learn from same sample multiple times.
- However, all the batch sizes are eventually converging to the same optimum
- The one anomaly is Advantage Normalization case, and the potential reason for the same was discussed earlier.

2.3.2 LunarLander-v3

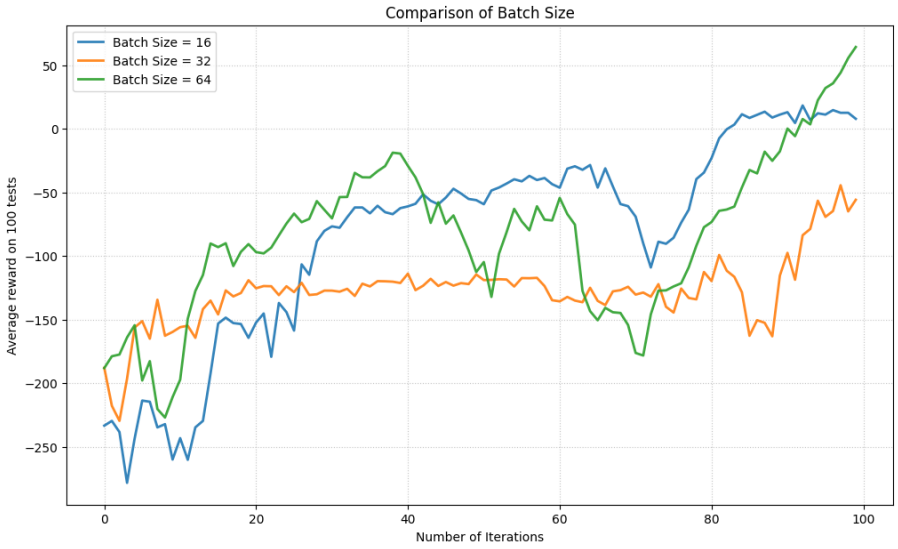

Figure 2.9: Comparing Effect of batch size on Vanilla REINFORCE Algorithm

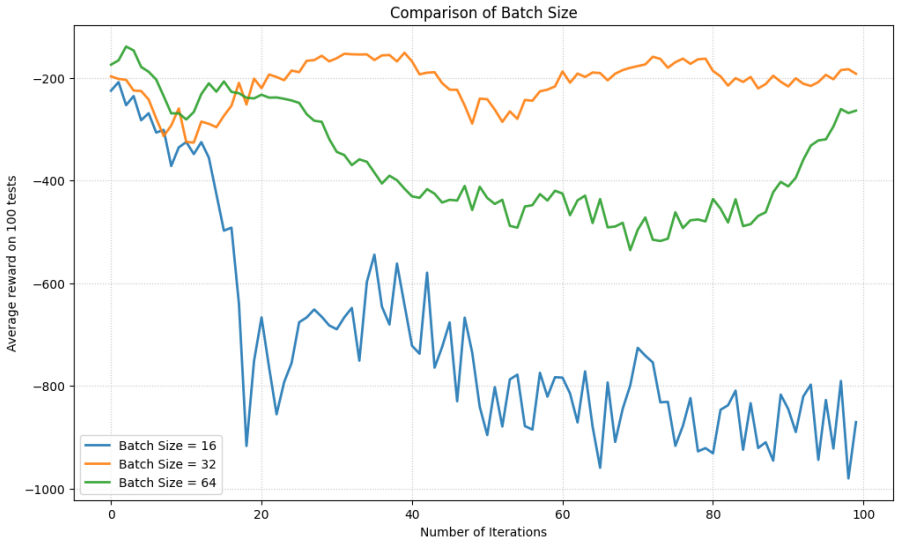

Figure 2.10: Comparing Effect of batch size on REINFORCE with Advantage Normalization

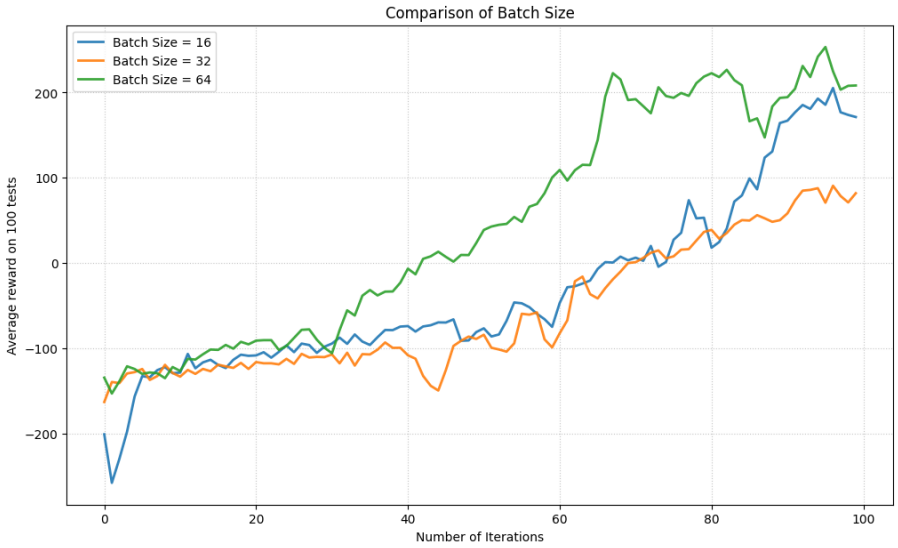

Figure 2.11: Comparing Effect of batch size on REINFORCE with Reward-to-go

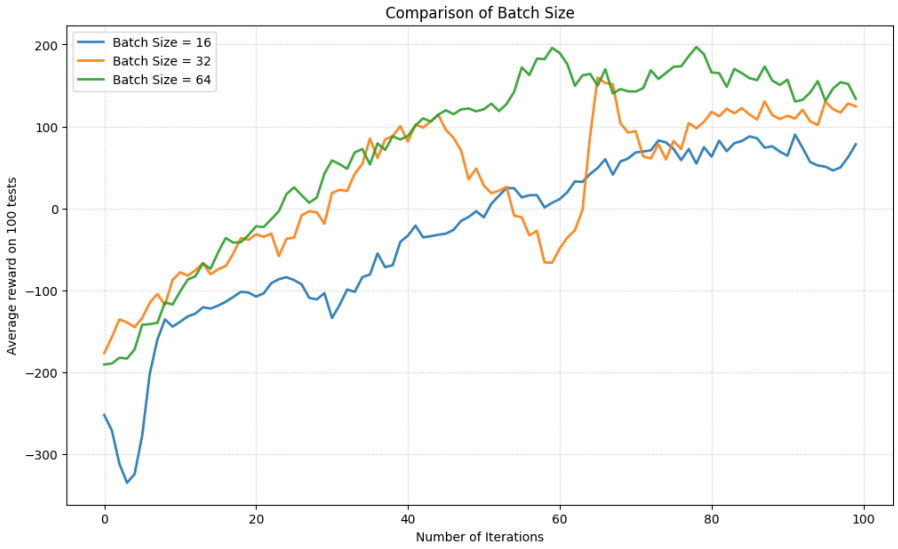

Figure 2.12: Comparing Effect of batch size on REINFORCE with both advantage normalization and reward-to-go

The trend is very similar and a similar explanation to the CartPole could be used.