Report: Assignment 3

Aditya Varun V

1 Introduction

The following is a report on the different sub-questions in assignment 3 of Reinforcement Learning.

2 Question 1: Deep Q-Network (DQN)

2.1 1 (a)

2.1.1 Environment Overview

Mountain Car:

- **State Space:** The state space is continuous and consists of two variables:
  - `position` - the horizontal position of the car (ranging from -1.2 to 0.6).
  - `velocity` - the car's velocity

  This results in a 2-dimensional continuous state space, with each state represented by a tuple `(position, velocity)`.
- **Action Space:** The action space is discrete and consists of three actions:
  - `0` - push the car to the left.
  - `1` - do nothing.
  - `2` - push the car to the right.
- **Reward:**
  - The agent receives a reward of -1 for each timestep until it reaches the goal, encouraging the agent to reach the goal as quickly as possible, as observed from the random agent
  - **Modified Reward Function Used:** The modified reward function refines the agent's incentives in the Mountain Car environment by providing context-sensitive rewards based on position and velocity, encouraging specific goal-oriented behaviors. Here are the key modifications and their benefits:

* **Interpolated Velocity**: The velocity is scaled to a range of $[-0.5, 0.5]$ to control its impact on reward, allowing smoother feedback based on movement dynamics.
* **Position-Based Rewards**: The reward function provides increasing bonuses as the car approaches the goal ($x \approx 1.0$), with significant rewards near the top. This encourages the agent to continue pushing toward the goal.
* **Step Penalty**: A small penalty (-0.5) per step discourages taking unnecessary actions and promotes efficient strategies.
* **Momentum-Based Bonus**: An additional reward is given if the car is moving from left to right with substantial velocity, encouraging the agent to build up the momentum needed to overcome the hill.

Without this modified reward function, the default reward is simply -1 per timestep until the goal is reached, which provides minimal feedback on proximity or momentum, leading to slower convergence. By rewarding both proximity to the goal and momentum, this function effectively guides the agent toward an optimal trajectory, accelerating convergence. Referenced the following for the above reward function: GitHub Link.

- **Objective:** The goal is to drive the car up a steep hill on the right. However, due to the car's low power, the agent must use momentum and go back and forth to build enough speed to reach the top.

Pong:

- **State Space:**
  - The state is represented as a 2D pixel array capturing the game screen. Each frame is an image of the game, which is typically preprocessed (e.g., downsampled and converted to grayscale) to reduce complexity.
  - For most implementations, the agent works with stacked frames (e.g., the last 4 frames) to capture motion information for effective decision-making.
- **Action Space:** The action space is discrete with six actions:
  - `0` - no action.
  - `1` - move up.
  - `2` - move down.
  - Actions `3`, `4`, and `5` represent other paddle movements but are rarely used as `1` and `2` are sufficient for effective gameplay.
- **Reward:**

  - The agent receives a reward of +1 for scoring a point and a reward of -1 when the opponent scores. No reward is given during other frames, as observed from the random agent.

- **Objective:** The goal for the agent is to maximize its score by successfully hitting the ball past the opponent's paddle while minimizing the points it loses to the opponent.

2.2 1 (b): General Code Overview

Overview of the DQN Implementation:

- **Device Setup**: Sets the computation device to GPU if available; otherwise, uses the CPU.
- **Hyperparameters**: Defines key hyperparameters: the discount factor γ, exploration rate ϵ, learning rate, and memory buffer size.
- **Replay Memory**: Uses a deque to store transitions, enabling the agent to learn from past experiences by sampling batches.
- **Neural Network Models**: Initializes the policy and target networks for Q-value prediction. Only the policy network is updated at each step, while the target network is updated periodically (every C steps) for stability.
- **Training Loop**: For each episode, the agent selects actions using an ϵ-greedy policy, stores transitions in replay memory, and updates the policy network by sampling batches when memory is sufficient.
- **Target Network Update**: Periodically copies the weights from the policy network to the target network.
- **Neural Network Parameter Update**: The parameters of the neural network are updated by using the *Bellman update* to calculate target Q-values through the target network. The policy network predicts Q-values for the current states, and the difference between the predicted and target Q-values is used as the loss function. This loss guides the parameter updates for the policy network, minimizing prediction errors over time.
- **Rewards Tracking**: Records total rewards per episode, calculates the mean reward over the last n episodes, and updates the best mean reward to monitor learning progress.

2.2.1 Mountain Car

Neural Network Structure and Hyperparameters

- **Neural Network Structure**: The implemented Deep Q-Network (DQN) consists of a feedforward neural network with three fully connected layers:

  - The first layer (`fc1`) is a fully connected layer with 15 neurons, followed by a ReLU activation function to introduce non-linearity.
  - The second layer (`fc2`) is another fully connected layer with 6 neurons, also followed by a ReLU activation function.
  - The output layer (`fc3`) has a size equal to the action space dimension (`output_dim`), representing the Q-values for each possible action. No activation function is applied here, as this layer outputs raw Q-values.

- **Hyperparameter Values Used**:
  - **Learning Rate** (α): `0.001`
  - **Discount Factor** (γ): `0.99` — Determines the weight of future rewards versus immediate rewards in the Q-value calculation.
  - **Initial Exploration Rate, Decay** (ϵ): `0.999, 0.997` — Governs the balance between exploration (trying new actions) and exploitation (choosing actions based on learned Q-values).
  - **Batch Size**: `64` — Number of experiences sampled from the replay buffer in each training step.
  - **Replay Buffer Size**: `10000` — Maximum number of experiences stored in the buffer for sampling.
  - **Target Network Update Frequency**: `10` — Number of training steps between updates of the target network with the online network's weights.
  - **Optimizer**: `Adam`
  - **Loss Function**: `MSELoss` — Measures the difference between predicted Q-values and target Q-values.

Analysis of Results

Description: Figure 1 shows the average rewards of the 10 previous episodes during training.

Trends:

- **Early Stages of Training**: There isn't much reward improvement. This is due to the trainer exploring the different actions for higher values of epsilon, and not finding the optimal actions yet
- **Later Stages of Training**: There is a spike in the value of the rewards at around the 800 episode mark, with the best mean reward reaching a peak of around +150 (positively rewarded for reaching the top)

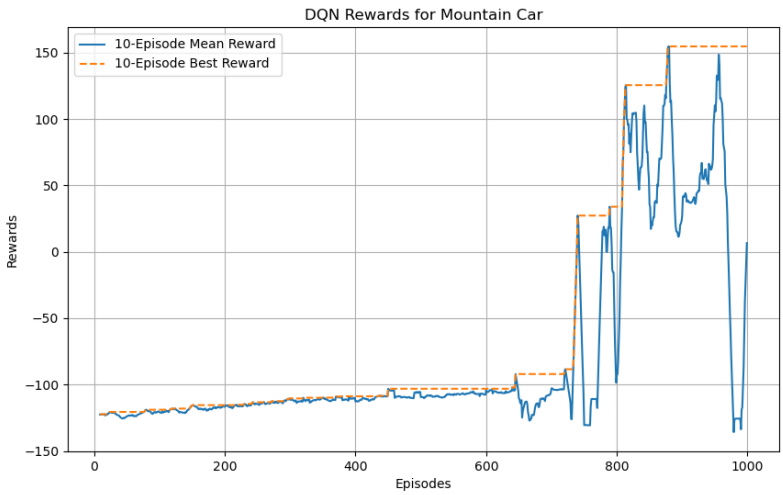

Figure 1: Learning curve for Mountain Car environment.

Performance:

- **Final Performance**: As seen in Figure 1, the agent learns to choose optimal actions at around the 800th episode. The positive rewards and the high best mean reward show that the agent learns to successfully traverse the hill in a short amount of time. This can also be observed from testing the agent without epsilon-greedy
- **Stability and Consistency**: There is a large variance towards the end of the training. This is perhaps due to overestimation in the Q-values and the maximization bias

Action Plot Analysis

Description: Figure 2 shows the different actions taken by the agent for the different states, identified by the position and velocity. The actions at the different states are colour coded.

Trends and Performance: We see that the agent learns to accelerate along the direction of the velocity, and accelerates left for lower values of velocity. This shows that the agent learns to increase the momentum, hence providing the energy to reach the top.

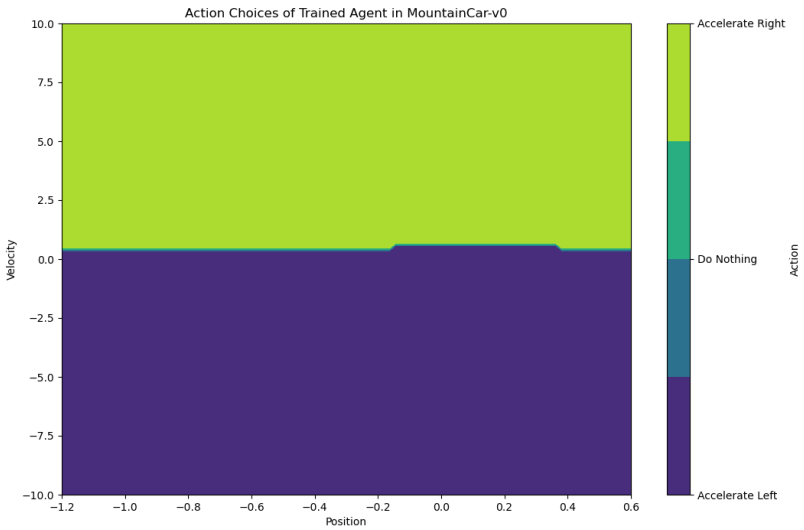

Figure 2: Actions Taken by Agent at Different States

2.2.2 Pong

Neural Network Structure, Image Preprocessing, and Hyperparameters

- **Neural Network Structure:** The neural network for the Pong-v0 environment is a CNN with three convolutional layers and two fully connected layers:
  - **Convolutional Layers:**
    - Layer 1: 32 filters, kernel size 8, stride 4, with ReLU.
    - Layer 2: 64 filters, kernel size 4, stride 2, with ReLU.
    - Layer 3: 64 filters, kernel size 3, stride 1, with ReLU.
  - **Fully Connected Layers:**
    - First layer: 512 units with ReLU.
    - Output layer: Q-values for actions, no activation.
- **Image Preprocessing:**
  - Convert each RGB frame to grayscale.
  - Downsample to 84×84 resolution.

- Use frame subtraction (difference between consecutive frames) to highlight movement.

- **Hyperparameter values used:**
  - **Learning Rate** (α): `0.001`
  - **Discount Factor** (γ): `0.99`
  - **Initial Exploration Rate, Decay** (ϵ): `0.999, 0.997`
  - **Batch Size**: `32`
  - **Replay Buffer Size**: `10000`
  - **Target Network Update Frequency**: `10`
  - **Optimizer**: `Adam`
  - **Loss Function**: `MSELoss`

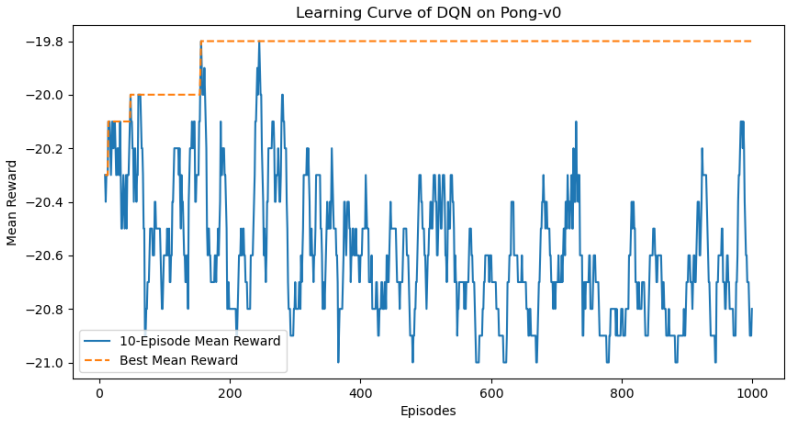

Figure 3: Learning curve for Mountain Car environment.

Analysis of Results

Description: Figure 3 shows the average rewards of the 10 previous episodes during training.

Trends:

- **Early Stages of Training**: There isn't much reward improvement. This is due to the trainer exploring the different actions for higher values of epsilon, and not finding the optimal actions yet
- **Later Stages of Training**: The trainer doesn't improve the rewards for the environment even after further episodes.

Performance and Future Improvements:

- **Final Performance**: The agent doesn't learn to choose optimal actions, since there isn't a sufficient improvement in the rewards after a considerable number of episodes
- **Potential Areas of Improvement**: For better performance, the following areas will be explored:
  - **Hyperparameter Tuning**: Using different learning rates, a slower exploration decay, and different neural network architectures, could help with the performance. Since the exploration rate decays too quickly in comparison to the large state space, allowing for more time for exploration would help with learning optimal strategies
  - **Better Computation Facilities**: Using better processing units will allow a larger number of episodes to train on in a shorter time, hence allowing a larger potential for improvement. Computation was a constraint when performing the above tests.
  - **Modified Reward Function**: Using a modified reward function could reward or punish certain actions, providing additional reinforcement to the learner.

2.3 1 (c): Hyperparameter Considerations and Observations

- **Hyperparameters Chosen and Justification**: The initial exploration rate (ϵ) was varied among 0.999, 0.9, and 0.8 for the mountain car environment. The exploration rate is crucial in determining how much the agent explores versus exploits its current knowledge. A higher initial exploration rate encourages the agent to try different actions across various states, which helps prevent it from prematurely converging on a suboptimal policy. This allows the agent to gather diverse experiences that can improve learning.
- **Observations from Results and Learning Curves**: From the learning curves, it is observed that having a higher initial exploration rate (e.g., 0.999) is beneficial in the long run. The agent explores more extensively, which allows it to discover effective action strategies across different states. As a result, higher exploration rates lead to improvements in both the mean and best rewards achieved over time, indicating more robust learning and better overall performance.

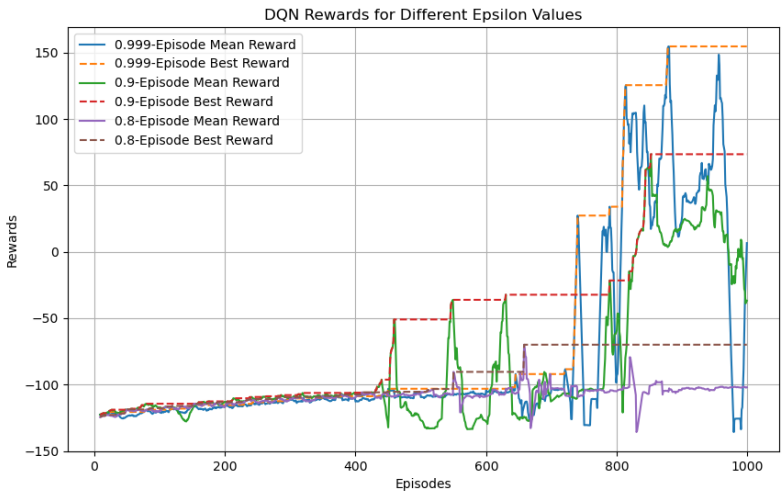

Figure 4: Learning curve showing the effect of varying initial exploration rates on mean and best rewards for the mountain car environment.

3 Question 2: Policy Gradient

3.1 2 (a): Actions, States, and Rewards

3.1.1 Environment Overview

CartPole-v0:

- **State Space:**
  - The state is represented by a 4-dimensional vector containing the cart position, cart velocity, pole angle, and pole angular velocity.
- **Action Space:** The action space is discrete with two actions:
  - `0` - push cart to the left.
  - `1` - push cart to the right.
- **Reward:**
  - The agent receives a reward of +1 for every time step that the pole remains upright, allowing the episode to continue.
- **Objective:** The goal is to balance the pole on the cart by applying left or right forces, keeping the pole upright for as long as possible without it tipping over or the cart moving out of bounds.

LunarLander-v3:

- **State Space:**
  - The state is represented by an 8-dimensional vector containing the lander's position (x and y coordinates), velocity (x and y), angle, angular velocity, and two Boolean values indicating whether the left or right leg is in contact with the ground.
- **Action Space:** The action space is discrete with four actions:
  - `0` - do nothing.
  - `1` - fire left orientation engine.
  - `2` - fire main engine.
  - `3` - fire right orientation engine.
- **Reward:**
  - The agent receives a reward based on how close it lands to the target and whether it lands safely. A reward of +100 to +140 is given for successful landing, while a crash results in a reward of -100. Each leg contacting the ground gives +10 points, and firing the main engine costs -0.3 points per frame.
- **Objective:** The goal is to land the spacecraft smoothly at a designated landing pad by controlling its engines effectively to avoid crashing while minimizing fuel consumption.

3.2 2 (b): General Code Overview

Overview of the Policy Gradient Implementation:

- **Device Setup**: Sets the computation device to GPU if available, otherwise defaults to CPU.
- **Environment and Policy Network Initialization**: Initializes the environment specified by `env_name`, and sets up the policy network with the state and action dimensions of the environment. An Adam optimizer is initialized for updating the policy network parameters.
- **Baseline and Advantage Calculation**: Defines helper functions for computing the baseline (mean reward) and calculating the advantages. If advantage normalization is enabled, the advantages are standardized to improve learning stability.
- **Training Loop**: For each epoch, the agent plays several episodes (determined by `batch_size`). During each episode:
  - Actions are sampled from the policy distribution, and transitions (states, actions, rewards) are recorded.

  - Cumulative rewards are calculated with discounting, with an option to use "reward-to-go" or total episode reward for each state.
  - Advantages are calculated by subtracting the baseline from the returns. The advantages are normalized if the `advantage_norm` is set to `true`
  - Computes the loss for the episode using the log probabilities of actions weighted by advantages. This is accumulated over all the episodes.

- **Policy Update**: Perform backpropagation using the accumulated loss value and updates the network parameters.
- **Gradient Tracking and Logging**: Calculates the gradient magnitudes of policy network parameters, logs the average reward per epoch, and periodically prints the average reward.
- **Model Saving and Data Export**: Saves the trained policy network, and exports the average rewards and gradient magnitudes to files for further analysis or comparison.

3.2.1 Cartpole

Neural Network Structure and Hyperparameters

- **Neural Network Structure**: The implemented Policy Network for the Policy Gradient (PG) algorithm consists of a feedforward neural network with two fully connected layers:
  - The first layer (`fc1`) is a fully connected layer with 128 neurons, followed by a ReLU activation function to introduce non-linearity.
  - The second layer (`fc2`) has a size equal to the action space dimension (`action_dim`), representing the probability distribution over actions. A softmax activation function is applied to output action probabilities.
- **Hyperparameter Values Used**:
  - **Learning Rate** (α): `0.01` — Controls the step size for updating network weights.
  - **Discount Factor** (γ): `0.99` — Determines the weight of future rewards in calculating the return for each action taken.
  - **Batch Size**: `32` — Number of episodes sampled for each training update.
  - **Reward-to-Go**: `True or False` — Indicates if the cumulative future reward is calculated from each time step, rather than from the episode's start, to stabilize training.
  - **Advantage Normalization**: `True or False` — If true, normalizes the advantages for each batch to improve training stability.

- **Baseline Function**: `Mean of Returns` — A baseline to subtract from rewards when calculating advantages, reducing variance in policy updates. 0 if the advantage normalization is set to false.
- **Optimizer**: `Adam`

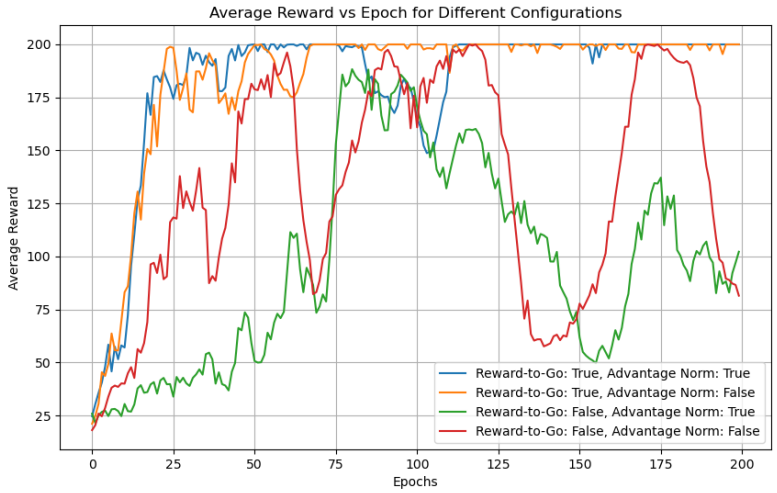

Figure 5: Learning curve showing average reward vs. epoch for CartPole with different settings of `advantage_norm` and `reward_to_go`.

Analysis of Results

- **Graph Analysis:** The learning curve for CartPole shows the average reward over epochs for the four different combinations of the boolean hyperparameters: `advantage_norm` and `reward_to_go`. The plot demonstrates the impact of each setting on the stability and noise in the rewards achieved.
  - When `reward_to_go` is set to `false`, the rewards exhibit significantly more noise. This is likely because the total reward has much more potential to have noise, compared to the reward-to-go function
  - Additionally, when `advantage_norm` is set to `false`, the reward curve appears somewhat noisier compared to when both `advantage_norm` and `reward_to_go` are set to `true`. Advantage normalization appears to help reduce variance, making the reward curve smoother and helping the agent converge more consistently.

3.2.2 Lunar Landing

Neural Network Structure and Hyperparameters

- **Neural Network Structure**: The implemented Policy Network for the Policy Gradient (PG) algorithm consists of a feedforward neural network with two fully connected layers:
  - The first layer (`fc1`) is a fully connected layer with 128 neurons, followed by a ReLU activation function to introduce non-linearity.
  - The second layer (`fc2`) has a size equal to the action space dimension (`action_dim`), representing the probability distribution over actions. A softmax activation function is applied to output action probabilities.
- **Hyperparameter Values Used**:
  - **Learning Rate** (α): `0.01` — Controls the step size for updating network weights.
  - **Discount Factor** (γ): `0.99` — Determines the weight of future rewards in calculating the return for each action taken.
  - **Batch Size**: `32` — Number of episodes sampled for each training update.
  - **Reward-to-Go**: `True or False` — Indicates if the cumulative future reward is calculated from each time step, rather than from the episode's start, to stabilize training.
  - **Advantage Normalization**: `True or False` — If true, normalizes the advantages for each batch to improve training stability.
  - **Baseline Function**: `Mean of Returns` — A baseline to subtract from rewards when calculating advantages, reducing variance in policy updates. 0 if the advantage normalization is set to false.
  - **Optimizer**: `Adam`

Analysis of Results

- **Graph Analysis:** The learning curve for Lunar Landing shows the average reward over epochs for the four different combinations of the boolean hyperparameters: `advantage_norm` and `reward_to_go`. The plot demonstrates the impact of each setting on the stability and noise in the rewards achieved.
  - When `reward_to_go` is set to `true`, we observe that the agent exhibits a much better performance.
  - When `reward_to_go` is set to `false`, the rewards exhibit significantly more noise. This is likely because the total reward has much more potential to have noise, compared to the reward-to-go function

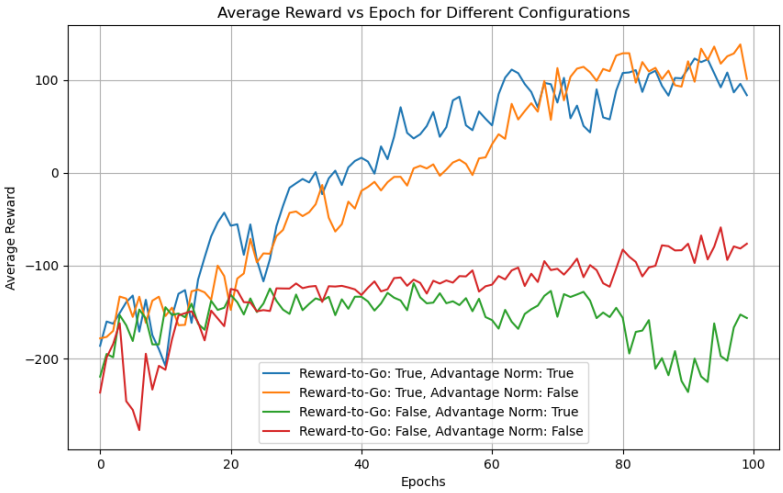

Figure 6: Learning curve showing average reward vs. epoch for Lunar Landing with different settings of `advantage_norm` and `reward_to_go`.

3.3 2 (c): Gradient Magnitude Analysis

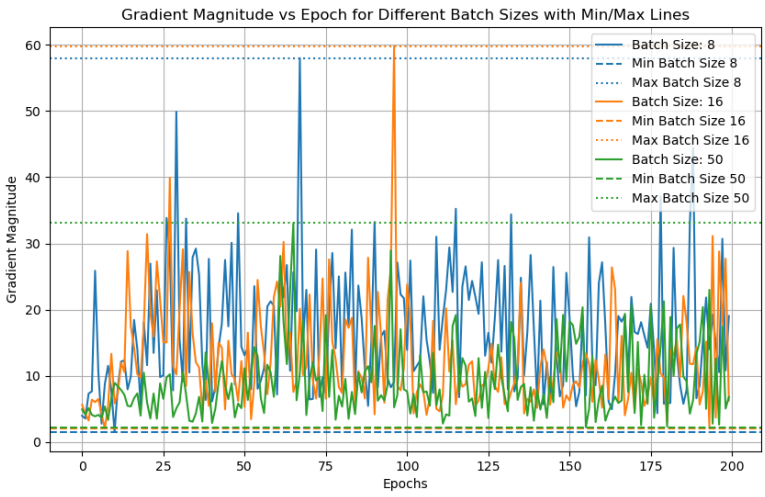

Figure 7: Magnitude of the Gradient vs epoch for Cartpole

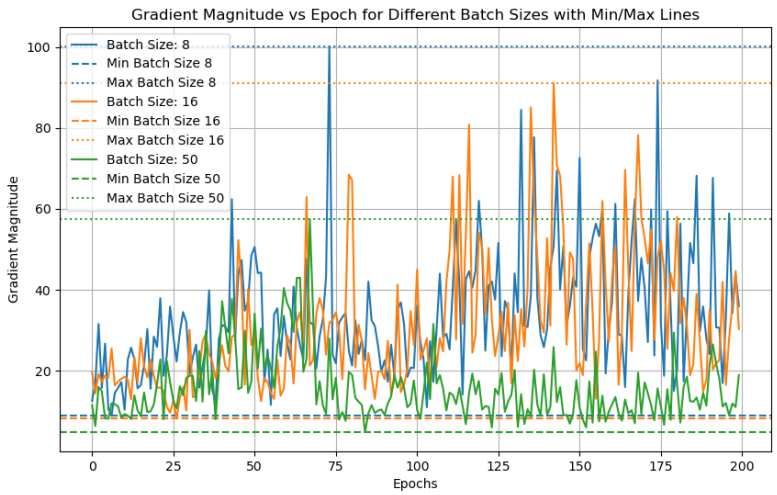

Figure 8: Magnitude of the Gradient vs epoch for the Lunar Lander

From the magnitude of the gradient vs epoch plots, it is clear that as we increase the batch size, there is a clear reduction in the range of the magnitude of the gradient. This shows that there is lesser variance in the gradient estimation as we increase the batch size.