HW1: Behavior Cloning in PyTorch

Atharv Kulkarni - u1322897

January 17, 2025

Part 1

In the demonstration, the car gets stuck in the valley and does not reach the top of the mountain. This is likely to happen when using Reinforcement Learning (RL) because RL depends on rewards to guide its learning, and in this problem, the agent only receives a positive reward when it successfully reaches the top of the mountain. As a result, the agent doesn't get any useful feedback for actions that move it in the right direction or make partial progress. For example, the critical action of moving backward to build momentum might not be discovered since it doesn't immediately result in a better reward.

The sparse reward structure is a major challenge, as the agent spends most of its time receiving negative rewards, such as -200 for failing to complete the task. Without intermediate rewards to reinforce beneficial actions, the RL algorithm is likely to struggle to learn the "swinging" strategy, where the agent has to move backward to gain momentum before attempting to climb the hill.

Another difficulty is the continuous state space in MountainCar, where both the position and velocity of the car are continuously variable. This means that the RL algorithm will need to explore a large range of possibilities to understand how its actions influence the car's state. This complexity makes it harder to find an effective policy, especially with simple trial-and-error methods.

Lastly, the dynamics of the environment make it possible for the agent to get stuck in local minima. For example, the agent might oscillate back and forth in the valley without gaining enough momentum to escape, making it harder for it to discover a successful strategy.

These challenges: sparse rewards, the need for an unintuitive strategy, continuous state space, and the potential for getting stuck in local minima, suggest that RL will likely struggle with the MountainCar problem unless additional techniques, such as better reward designs or advanced exploration strategies, are employed.

Part 2

In this part, I experimented with different strategies to solve the MountainCar task by manually controlling the car. The goal was to reach the flag at the top of the mountain in as few steps as possible, earning a higher score. Below are the strategies I tried, along with the corresponding total rewards achieved and their averages:

Strategy	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Avg. Reward
Right, Left, Right	-149.0	-112.0	-111.0	-115.0	-121.0	-121.6
Left, Right	-185.0	-127.0	-128.0	-117.0	-133.0	-138.0
Left, Right, Left, Right	-176.0	-172.0	-166.0	-168.0	-168.0	-170.0

Table 1: Results for Different Strategies in the MountainCar Task

Among the strategies, I preferred the approach of going right, then left, then right all the way up the hill. This method allowed me to achieve the best score of -111.0 and felt the most reliable. It provided better control over the car's momentum, reducing the need for excessive back-and-forth actions while still effectively building up the speed required to reach the flag. This strategy consistently produced favorable results compared to the others.

Among the strategies, I preferred the approach of going right, then left, then right all the way up the hill. This method allowed me to achieve the best score of -111.0 and felt the most reliable. It provided better control over the car's momentum, reducing the need for excessive back-and-forth actions while still effectively building up the speed required to reach the flag. This strategy consistently produced favorable results compared to the others.

Part 3

Using my preferred strategy of going right, then left, then right all the way up the hill, the agent successfully learns to imitate my actions and strategy. It is able to reach the top of the hill in 5 out of 6 attempts, demonstrating that it has effectively captured the behavior I demonstrated. Interestingly, the agent also manages to outperform my average human performance on one occasion. However, in one of its attempts, the agent gets stuck in the valley and fails to escape, highlighting occasional imperfections in its learned policy.

The evaluation results, including the rewards for each attempt and whether the agent successfully crossed the hill, are summarized in the table below:

Attempt	Reward	Crossed Hill?
0	-200.0	×
1	-121.0	✓
2	-156.0	✓
3	-155.0	✓
4	-160.0	✓
5	-122.0	✓

Table 2: Rewards and Success of Each Evaluation Attempt by the Agent

Additionally, the overall performance of the agent is as follows:

- **Average Reward:** -152.33
- **Minimum Reward:** -200.0
- **Maximum Reward:** -121.0

The results indicate that the agent can successfully learn the demonstrated strategy, however there might still be room for improvement in the policy refinement process.

Part 4

After providing 5 good demonstrations of the strategy (going right, then left, and then right all the way up the hill), the model showed improved performance. The agent was able to get out of the valley 5 out of 6 times, successfully mimicking the demonstrated strategy.

The average policy return is higher compared to the single demonstration case, which indicates that providing more demonstrations allows the model to better generalize the strategy. Additionally, the variation between the rewards when the agent succeeded has reduced, showing that it can now cross the hill more confidently and with sustained performance. By observing more demonstrations, the model reinforced what it had already learned and achieved better overall results.

The evaluation results, along with the average, minimum, and maximum policy returns, are summarized below:

Attempt	**Reward**	**Crossed Hill?**
0	-200.0	×
1	-124.0	✓
2	-128.0	✓
3	-136.0	✓
4	-130.0	✓
5	-124.0	✓

Table 3: Rewards and Success of Each Evaluation Attempt After 5 Demonstrations

- **Average Policy Return:** -140.33
- **Minimum Policy Return:** -200.0
- **Maximum Policy Return:** -124.0

These results demonstrate that with more demonstrations, the agent can better learn and reproduce the desired behavior. The higher average return and reduced variation in rewards when crossing the hill indicate improved confidence and consistency in the agent's performance.

Part 5

In this part, two demonstrations were provided: one bad demonstration where the right arrow key was pressed for the entire episode, and one good demonstration that involved swinging (going right, then left, and then right all the way up the hill).

The agent primarily learned to mimic the bad behavior, resulting in it repeatedly getting stuck while trying to go right without building sufficient momentum. While it did mimic the good demonstration once, most attempts reflected the poor strategy, and it failed to escape the valley. This outcome highlights the significant impact that bad demonstrations can have on the policy, as the agent struggles to differentiate between good and bad behaviors without additional guidance.

The evaluation results, including rewards for each attempt and whether the agent successfully crossed the hill, are summarized below:

Attempt	Reward	Crossed Hill?
0	-200.0	×
1	-130.0	✓
2	-200.0	×
3	-200.0	×
4	-200.0	×
5	-200.0	×

Table 4: Rewards and Success of Each Evaluation Attempt After Good and Bad Demonstrations

- **Average Policy Return:** -188.33
- **Minimum Policy Return:** -200.0
- **Maximum Policy Return:** -130.0

Why might bad demonstrations be a problem?

Bad demonstrations are problematic because the agent does not inherently know which behaviors are correct and which are not. Behavioral cloning (BC) relies entirely on the given demonstrations, so if the dataset includes suboptimal or conflicting behaviors, the model may overfit to poor strategies. This can result in the agent learning counterproductive actions, as seen here where it mimicked the bad behavior of going right from the start.

Potential Idea for Making BC Robust to Bad Demonstrations

One potential idea to make BC robust to bad demonstrations is to weight demonstrations based on their quality. For example, assign higher weights to good demonstrations that successfully complete the task and lower weights to bad demonstrations. This could be implemented by evaluating demonstrations beforehand and introducing a quality score, or by integrating a mechanism during training to identify and prioritize successful actions.

Part 6

In this part, 5 demonstrations were provided to teach the agent to oscillate back and forth between the sides of the valley without going out and reaching the flag. After training the agent using Behavioral Cloning (BC), it successfully learned this new behavior.

The agent was able to oscillate between the sides of the hill. It performed this oscillation behavior successfully on 5 of the 6 evaluation attempts (on one of the attempts it got stuck at the bottom of the valley), demonstrating that the BC approach was effective in replicating this specific skill.

Part 7

To implement Behavioral Cloning from Observation (BCO) in `mountain_car_bc.py`, the key changes involve implementing an inverse dynamics model and fine-tuning the policy network.

1. Implement an Inverse Dynamics Model

The inverse dynamics model is essential for BCO, as it allows inferring missing actions from state transitions in human demonstrations. In BCO, the demonstrations consist of sequences of states (s and s') without explicit action labels.

The inverse dynamics model will be a neural network that predicts the action a taken between two states:

- **Input:** A pair of states (s, s'), where s represents the state before the action, and s' represents the state after the action.
- **Output:** The predicted action a, which is one of the three possible actions in the MountainCar environment: left acceleration, right acceleration, or no acceleration.

The implementation will include a class called `InvDynamicsNetwork` to define the neural network architecture. This network will have:

- An input layer that processes the concatenated states (s, s').
- 4 hidden layers with Leaky Relu as the non-linear activation function.
- An output layer that provides logits for the three possible actions.

Additionally, a function `train_inverse_dynamics_model` will be written to train the model using random interaction data collected from the environment. This function will optimize the network to minimize cross-entropy loss between the predicted and actual actions.

2. Fine-Tune the Policy

After training the policy network using the inferred actions from the inverse dynamics model, the policy will be fine-tuned for parameters like the learning rate, number of hidden layers, etc. to improve its performance.

Part 8

The implementation of BCO worked as expected, and the model successfully learned to mimic my strategy of going right, then left, and then all the way up the hill. The model climbed the hill in 5 out of 6 evaluation attempts, similar to the performance observed in Part 3 with standard Behavioral Cloning (BC). However, BCO show more variation in its results consistency as compared to standard Behavioral Cloning when run multiple times and is more sensitive to the quality of the demonstration.

To achieve this, several hyperparameters were tested and tuned to improve performance:

- **Learning Rate:** Tested values ranging from 1 to 0.0001. The best results were achieved with a learning rate of lr = 0.0009.
- **Number of Hidden Layers:** Tested architectures with 2, 3, and 4 hidden layers. A network with 4 hidden layers provided the best performance.
- **Hidden Size:** Evaluated sizes of 128, 256, and 512 neurons per layer. A hidden size of 256 was chosen for optimal performance.
- **Activation Function:** Leaky ReLU was used as the activation function to improve non-linear modeling capability.
- **Weight Initialization:** Xavier uniform distribution was applied to initialize weights, ensuring stable convergence.

The evaluation results, including rewards for each attempt and whether the agent successfully crossed the hill, are summarized in the table below:

Attempt	Reward	Crossed Hill?
0	-129.0	✓
1	-130.0	✓
2	-125.0	✓
3	-200.0	×
4	-127.0	✓
5	-129.0	✓

Table 5: Rewards and Success of Each Evaluation Attempt After Implementing BCO

- **Average Policy Return:** -140.0
- **Minimum Policy Return:** -200.0
- **Maximum Policy Return:** -125.0

The results indicate that the inverse dynamics model in BCO can effectively learn the desired behavior, producing performance comparable to standard BC. By tuning hyperparameters like learning rate, architecture, and initialization, the model achieved consistent and reliable behavior replication.

Hyperparameters

Behavioral Cloning

The following hyperparameters were used for the Behavioral Cloning (BC) implementation:

- **Learning Rate (lr):** 0.01
- **Fully Connected Hidden Layers for the Network:** 2

- **Number of neurons for each hidden layer (Hidden Size):** 128
- **Activation Function:** ReLU (Rectified Linear Unit)

Behavioral Cloning from Observation (BCO)

The following hyperparameters were used for the Behavioral Cloning from Observation (BCO) implementation:

- **Learning Rate (lr):** 0.0009
- **Fully Connected Hidden Layers for the Network:** 4
- **Number of Neurons for Each Hidden Layer (Hidden Size):** 256
- **Activation Function:** Leaky ReLU (Leaky Rectified Linear Unit)

These hyperparameters were selected based on experimentation and were found to provide reliable and consistent performance for the MountainCar task.