HW1: Behavior Cloning in PyTorch

CS 6955: Adv Artificial Intelligence
University of Utah

Name: Seongil Heo
UID: u1527760
Date: January 14, 2026

Part 1:

Run the following code again

```
python test_gym.py
```

What do you notice happening? MountainCar is a classic RL problem known for being a difficult exploration problem. Why do you think a trial and error method like RL would struggle with MountainCar? Hint: RL uses the reward as a signal for what is good and bad so why might that be a problem in MountainCar?

Answer

The agent moves around the bottom of the valley and fails to reach the goal.

Reinforcement learning performs poorly in MountainCar because the constant -1 reward provides no guidance before the goal is reached. The agent must first take actions that seem bad in the short term to succeed in the long term, which is difficult for trial-and-error learning to discover. Therefore, MountainCar is a hard exploration problem due to sparse and uninformative rewards and the need for long-term planning.

Part 2:

You will now learn how to solve the MountainCar task by driving the car yourself. Run

```
python mountain_car_play.py
```

Use the arrow keys to control the car by accelerating left and right. Note that if you run out of time you get a reward of -200.0 and the car will reset to the bottom of the hill. If you get to the flag in less than 200 steps you can get a higher score. Once you reach the flag the environment will restart at the bottom of the hill.

Keep practicing until you can reliably get out of the valley. You can see your score for each epsiode output on the command line. Experiment with different strategies: e.g.

1. Going right, then left, then right all the way up the hill
2. Going left, then right all the way up the hill.
3. Left, right, left, right up the hill.

Which strategy do you like best? What is the best score you can get as a human demonstrator?

Answer

The car must first reach a certain height on the left slope and then accelerate toward the goal. If the initial state is on the right slope, it is possible to reach a sufficient height directly. However, when starting near the center or the left side, the car should first move slightly up the right

slope, then go left to build enough momentum, and finally accelerate to the right toward the goal.

When the initial state starts on the right slope, accelerating to the right after reaching about half of the left slope results in the highest reward, -84. **(Going left, then right all the way up the hill.)**

Part 3:

Now you will code up a simple behavioral cloning (BC) agent to drive itself out of the valley.

First take a look at `mountain_car_bc.py` and try to get a basic understanding of what is happening. By default this code will collect a single demonstration, then parse the states and actions into tensors for easy input to a PyTorch neural network. You then need to add code to trains a policy to imitate the demonstrations and evaluate the policy by testing it on different initial states. The commmand line will output the average, min, and max total rewards.

Play with the learning rate and number of iterations and network architecture a bit if it doesn't work initially, but don't spend too much time finetuning. If it roughly imitates you that is great. Move on. You've got better things to do with your life than fine-tuning hyperparams for a simple assignment :). Note that from 1 good demo the car should get out of the valley at least some of the time.

Test it out by running

```
python mountain_car_bc.py
```

Try to give a good demonstration. Then watch what the agent has learned. Does it do a good job imitating? Does it ever get stuck? What is the average, min, and max?

Answer

The results vary depending on the quality of the demonstration. Overall, the agent imitates my behavior well. When I provide a good demonstration, the policy consistently alternates between moving left and right to build momentum and reaches the goal in all evaluation episodes. However, with poor demonstrations, the agent sometimes gets stuck in the valley and fails to reach the goal.

In my best run, the evaluation returns were:

- Average policy return: -126.33
- Min policy return: -136.0
- Max policy return: -123.0

Part 4:

Let's give more than one demonstration. Run the following to give 5 good demonstrations. If you mess up during one demo, feel free to restart until you give 5 good demos and try to keep to a consistent strategy.

```
python mountain_car_bc.py --num_demos 5
```

Report the average min and max returns. Did it perform any better? Why or why not? Does the agent copy the strategy you used?

Answer

Overall, the performance did not improve and instead became worse. However, in some runs, the results were occasionally better than with one demonstration. This is because the optimal demonstration depends on the initial state, and my demonstrations are not always identical. As a result, the demonstrations contained slightly different strategies, which introduced inconsistency in the dataset. This led to degraded average performance.

Nevertheless, the agent still generally imitates my behavior well and is able to escape the valley in most cases.

The evaluation results were:

- Average policy return: -149.83
- Min policy return: -165.0
- Max policy return: -124.0

Part 5:

What do you think will happen if we give good and bad demonstrations? You will now give two demonstrations. For the first one, just press the right arrow key for the entire episode until it restarts. Then for the second demo, give a good demonstration that quickly gets out of the valley.

```
python mountain_car_bc.py --num_demos 2
```

What does the policy learn? Why might bad demonstrations be a problem? Briefly describe one potential idea for making BC robust to bad demonstrations.

Answer

The policy learns to do almost nothing. This happens because behavior cloning treats all demonstrations equally and simply imitates the most frequent actions in the dataset. Therefore, when the bad demonstration consists only of moving left, the learned policy imitates this behavior and keeps moving left.

One potential way to make behavior cloning more robust to bad demonstrations is to remove low-reward demonstrations and train only on high-quality demonstrations.

Part 6:

Let's teach the agent to do something else. Give 5 demonstrations that drive back and forth going up and down the sides of the valley, but without going out and reaching the flag. Run BC and see if the agent learns this new skill:

```
python mountain_car_bc.py --num_demos 5
```

Were you able to teach the agent to oscillate without ending the episode early?

Answer

Yes. The agent learns to oscillate without terminating the episode early. Because behavior cloning does not use reward signals or goal information, it has no incentive to reach the goal.

Part 7:

Describe what changes you would need to make to the code in `mountain_car_bc.py` to implement Behavior Cloning from Observation (BCO). Answer this question before starting Part 8 and before looking at `mountain_car_bco.py`.

Answer

To implement BCO, an inverse dynamics model is added. Then, interaction data is collected and the inverse dynamics model is trained to predict actions from state transitions. After that, the inverse dynamics model is used to infer actions from state-only demonstrations, and finally a policy is trained using standard behavior cloning on the reconstructed state–action dataset.

Part 8:

Implement and test BCO(0) (see paper above for details) by training an inverse dynamics model. You can find starter code in `mountain_car_bco.py`. Report how well it works and what you tried to get it to work. Note that `mountain_car_bco.py` already imports and calls the bc functions — the only thing you have to do is code the inverse dynamics model and training code. You may need to play around a little with hyperparamters to adjust the amount of random exploration data, learning rate, network architecture, etc. But don't spend too much time perfecting this. If it's able to get out of the valley sometimes, that's great and sufficient for this project.

Answer

BCO(0) is able to escape the valley reliably. The evaluation results were:

- Average policy return: -123.17
- Min policy return: -181.0
- Max policy return: -87.0

Following the paper, I used an inverse dynamics model with two hidden layers of 8 units each and Leaky ReLU activations, trained using Adam. I collected 2000 random interactions to train the inverse dynamics model (200 timesteps per episode and 10 iterations).

I experimented with hyperparameters. The inverse dynamics model performed best when trained for 2000 gradient steps, while fewer than 1000 steps often failed to produce a successful policy. Training for more than 4000 gradient steps tended to degrade performance, likely due to overfitting. Additionally, collecting random samples for fewer than 7 iterations often failed to learn a successful policy, whereas using at least 7 iterations made successful learning possible, with 10 iterations producing the best results.