GCB6206 Homework 1: Imitation Learning

[WOOSHIK WON]
[2025961226]

1 Introduction

The goal of this assignment is to make you familiar with **(1) Yonsei AI Teaching Cluster (VESSL AI)**, **(2) Reinforcement Learning Environments**, and **(3) Imitation Learning Algorithms**, including behavioral cloning (BC) and DAgger. Here is the link to this homework report template in Overleaf.

2 (Optional) LaTeX

If you are not familiar with LaTeX, we highly recommend you to go over the Overleaf tutorial, "Learn LaTeX in 30 minutes."[1] For more detailed instructions, refer to Overleaf's documentation.[2]

For this homework assignment, you need to use Overleaf as follows:

1. Create your Overleaf account.
2. Select "Menu" on the top-left corner and click "Copy Project", so that you can change the `main.tex` file.
3. Replace the placeholder in `\author{...}` with your name and student ID (in the line 37 and line 38).
4. After making any changes, press the "Recompile" button to see updates in the PDF panel on the right.
5. Confirm that your name and student ID appear correctly under the title.
6. Read through `main.tex` and play with it!

3 (Optional) VESSL AI

If you have your own GPUs or want to use Google Colab, you can skip this section.

If you need a GPU for this assignment, we provide VESSL AI (AI-LEC-1 group), a GPU cluster for lectures in the Yonsei AI department. You can use an RTX3090 GPU for 300 hours per month. Here is the instruction about how to launch and use a workspace (i.e. a virtual machine with one GPU assigned to you) for this homework.

Warnings:

- Always stop the workspace when not in use to save the computing resources.
- The **VESSL Workspace** provides a maximum allocation of 72 hours of runtime. Exceeding this limit will automatically stop your workspace. Monitor your usage regularly to manage your available hours, and restart the workspace if it has been automatically stopped.

[1] https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes
[2] https://www.overleaf.com/learn

4 Play with Reinforcement Learning Environments

We provide a tutorial (`GymTutorial.ipynb`) inside the homework material to help you familiarize yourself to Reinforcement Learning (RL) environments. We encourage you to review and run this tutorial to understand how to interact with gym environments.

After completing the tutorial, please answer the following questions to demonstrate your understanding of the basic principles of gym environments.

4.1 Questions:

1. Describe the role of the `step()` function. What kind of information does it return?

   Answer: The `step()` function runs one timestep and returns five values: observation (the next state), reward (scalar value for this transition), terminated (true if episode ended naturally, like the robot falling), truncated (true if we hit the max timestep limit), and info (a dict with diagnostic information).

2. Describe the role of the `reset()` function in a gym environment. What is the return value of this function?

   Answer: The `reset()` function initializes the environment to a starting state for a new episode. It returns two things: observation (the initial state where the agent starts) and info (a dict with auxiliary information about the initial state—though we don't really use it in our implementation).

3. How can you figure out if some gym environment has a discrete action space or continuous action space?

   Answer: Check the type of `env.action_space`. If it is `gym.spaces.Discrete`, the environment has a discrete action space and the number of actions is `env.action_space.n`. If it is `gym.spaces.Box`, the environment has a continuous action space and the action dimension is `env.action_space.shape[0]`; actions are NumPy float arrays.

5 Behavioral Cloning

5.1 Code Structure Overview

5.1.1 Main Script: `run_hw1.py`

- Entry point for running behavioral cloning experiments.
- Parses command-line arguments and sets up experiment parameters.
- Loads the expert policy from a specified file.
- Initializes and runs the `BCTrainer` for training.

- Handles logging and experiment output storage.

5.1.2 Trainer: `bc_trainer.py`

- Defines `BCTrainer`, which manages training loops, data collection, and logging.
- Key methods:
  - `run_training_loop()`: Executes multiple training iterations.
  - `collect_training_trajectories()`: Gathers data using the current policy.
  - `train_agent()`: Updates the agent using sampled trajectories.
  - `do_relabel_with_expert()`: Modifies actions using an expert policy (for DAgger).
  - `perform_logging()`: Logs training and evaluation statistics.

5.1.3 Behavioral Cloning Agent: `bc_agent.py`

- Implements `BCAgent`, which represents an agent trained via behavioral cloning.
- Key attributes:
  - `actor`: Uses an `MLPPolicySL` to map observations to actions.
  - `replay_buffer`: Stores past trajectories.
- Key methods:
  - `train()`: Updates the policy using supervised learning.
  - `add_to_replay_buffer()`: Stores new rollouts.
  - `sample()`: Retrieves training data from the replay buffer.

5.1.4 Policy Network: `MLP_policy.py`

- Implements `MLPPolicySL`, a feedforward neural network for behavioral cloning.
- Key methods:
  - `get_action()`: Computes an action given an observation.
  - `forward()`: Defines the network's forward pass.
  - `update()`: Trains the policy using a supervised loss function.

5.1.5 Replay Buffer: `replay_buffer.py`

- Implements a simple experience replay buffer for storing and sampling transitions.
- Key methods:

- `add_rollouts()`: Adds new trajectories to the buffer.
- `sample_random_data()`: Returns a random batch of experiences.

5.1.6 Utility Functions: `utils.py`

- Contains helper functions for environment interaction.
- Key methods:
  - `sample_trajectory()`: Collects a single episode of experience.
  - `sample_trajectories()`: Gathers multiple rollouts.
  - `sample_n_trajectories()`: Collects a fixed number of trajectories.
  - `get_trajlength()`: Computes the length of a trajectory.

5.1.7 PyTorch Helper Functions: `pytorch_util.py`

- Provides utility functions for neural network construction and tensor handling.
- Key methods:
  - `build_mlp()`: Constructs a multi-layer perceptron.
  - `from_numpy()`, `to_numpy()`: Converts between NumPy arrays and PyTorch tensors.

5.2 Implement Behavioral Cloning

Your task is to fill in sections marked with `TODO` in the code. In particular, take a look at the following files:

- `gcb6206/infrastructure/bc_trainer.py`, except for `do_relabel_with_expert` function, which is for the next section, DAgger.
- `gcb6206/policies/MLP_policy.py`
- `gcb6206/infrastructure/replay_buffer.py`
- `gcb6206/infrastructure/utils.py`
- `gcb6206/infrastructure/pytorch_util.py`

Here is a recommended order of implementation:

1. Implement `build_mlp()` in `pytorch_util.py`, as it is used in the policy network.
2. Implement the policy network in `MLP_policy.py`, as it is used by the agent.
3. Implement utility functions in `utils.py`. They provides fundamental trajectory sampling functions.
4. Implement the replay buffer in `replay_buffer.py`, as it is required for agent training.
5. Implement the behavioral cloning agent in `bc_agent.py`.

6. Implement the trainer in `bc_trainer.py`.

7. If you have completed the above implementations, the training can be launched with `run_hw1.py`, which ties everything together.

Run behavioral cloning (BC) and report results on **two tasks**: (1) the Ant environment (`Ant-v4`), where a behavioral cloning agent should achieve at least 30% of the performance of the expert, and (2) any one environment among `Walker2d-v4`, `HalfCheetah-v4`, and `Hopper-v4`, where the expert data is also provided.

The performance of the expert policy can be found in `Initial_DataCollection_AverageReturn` in the log output.

Once you implement `TODO` above, you can train a BC policy for the Ant task as follows:

```
python gcb6206/scripts/run_hw1.py \
    --expert_policy_file gcb6206/policies/experts/Ant.pkl \
    --env_name Ant-v4 --exp_name bc_ant --n_iter 1 \
    --expert_data gcb6206/expert_data/expert_data_Ant-v4.pkl \
    --video_log_freq -1
```

If your run succeeds, you will be able to find your tensorboard log data in `hw1_starter_code/data/q1_[--exp_name]_[--env_name]_[current_time]/`.

When providing results, report the **mean and standard deviation** of your policy's return **over multiple rollouts** in a table, and state which task was used. When comparing one that is working versus one that is not working, be sure to set up a fair comparison in terms of network size, amount of data, and number of training iterations. **Provide these details** (and any others you feel are appropriate) in the table caption.

Note: What "report the mean and standard deviation" means is that your `eval_batch_size` should be greater than `ep_len`, such that you're collecting multiple rollouts when evaluating the performance of your trained policy. For example, if `ep_len` is 1000 and `eval_batch_size` is 5000, then you'll be collecting approximately 5 episodes (maybe more if any of them terminate early), and the logged `Eval_AverageReturn` and `Eval_StdReturn` represents the mean/std of your policy over these 5 rollouts. Make sure you include these parameters in the table caption as well.

Note: To generate videos of the policy rollouts, remove the flag "`--video_log_freq -1`". However, this is slower, and so you probably want to keep this flag on while debugging.

5.2.1 BC Results

Table 1: Behavioral Cloning performance on two continuous control tasks. Network: 2 hidden layers with 64 units each (tanh activation). Training: 1000 gradient steps, batch size 100, learning rate 5e-3. Evaluation: 5000 timesteps ($\sim$5 episodes of length 1000).

Environment	**Performance (Mean Return $\pm$ Std)**
`Ant-v4`	4004.17 ± 1297.84
`HalfCheetah-v4`	3801.78 ± 116.14

5.3 Rendering and Evaluating Rollouts

To familiarize yourself with rendering environments, you are required to include a screenshot of an evaluation rollout of your trained policy. If you are using tensorboard, you can check the *eval_rollouts* section. To analyze how the training is going, it is important to render and watch videos rather than just looking at numbers!

5.3.1 Environment Rendering

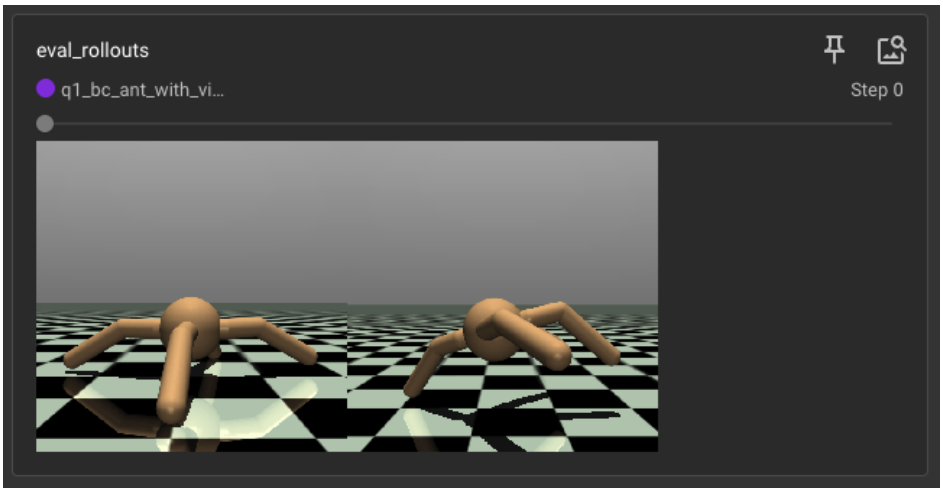

Figure 1: Trained BC policy rollout on Ant-v4. The quadruped robot successfully walks forward after 1000 gradient steps of training.

5.4 Hyperparameter Tuning of Behavioral Cloning

Experiment with **one set of hyperparameters** that affects the performance of the behavioral cloning agent, such as the amount of training steps, the amount of expert data provided, or something that you come up with yourself. For one of the tasks used in the previous question, show a graph of how the BC agent's performance varies with the value of this hyperparameter. State the hyperparameter and a brief rationale for why you chose it.

You should include at least **4 different** settings for the hyperparameter you have chosen, including the default setting you used in the previous part.

Note: There are some default hyperparameters you can specify using the command line arguments. You may want to choose one of the hyperparameters listed below:

- Number of gradient steps for training policy (`--num_agent_train_steps_per_iter`, default: 1000)
- The amount of training data (`--batch_size`, default: 1000)
- Training batch size (`--train_batch_size`, default: 100)
- Depth of the policy neural net (`--n_layers`, default: 2)
- Width of the policy neural net (`--size`, default: 64)

- Learning rate for supervised learning (`--learning_rate`, default: $5e-3$)

You can specify the hyperparameter in the command line when you execute the script. For example, if you run the command like this, you can train the policy for 500 gradient steps:

```
python gcb6206/scripts/run_hw1.py \
    --num_agent_train_steps_per_iter 500 \
    --some other arguments...,
```

Note: Use `matplotlib` for drawing the plots. If you are not familiar with `matplotlib`, you can refer to its official tutorial.

5.4.1 Hyperparameter Tuning Results

Hyperparameter: Number of Gradient Steps (`num_agent_train_steps_per_iter`)

Tested values: [500, 1000, 2000, 3000]

Plot:

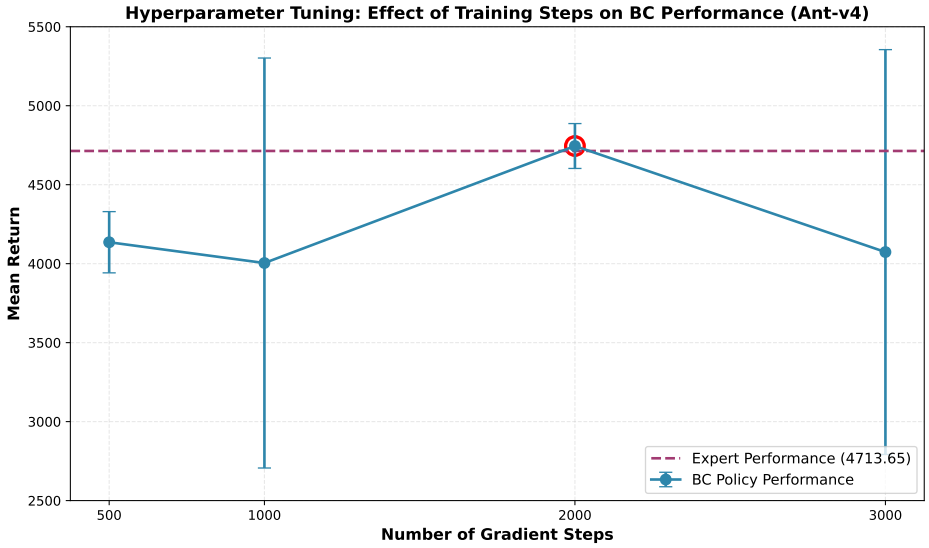

Figure 2: BC performance in `Ant-v4` across different training steps. The policy peaks at 2000 gradient steps (4744.97 ± 142.09), actually beating the expert slightly (4713.65). Training setup: 2 hidden layers with 64 units, batch size 100, learning rate 5e-3. Error bars show standard deviation over 5 evaluation episodes.

Rationale: I varied the number of gradient steps since it's one of the key hyperparameters in supervised learning—it lets me see the trade-off between under-fitting early on and potentially over-fitting with too much training.

Performance peaks at 2000 steps, where the policy surpasses expert level (100.7%). But pushing to 3000 steps actually hurts performance, dropping to 86.4% of expert. This could be over-fitting on the limited expert dataset, though some of it might just be evaluation noise.

What's also notable is that the standard deviation is much lower at 2000 steps (142.09) compared to 1000 and 3000 steps (both ¿1200), suggesting the policy is more stable there. Based on this, 2000 gradient steps seems like the sweet spot for this task.

6 DAgger

6.1 Implement DAgger

Now your task is to implement the DAgger algorithm. If you implemented the BC part correctly, you can just implement the `TODO` in the following file.

- `do_relabel_with_expert` function in `gcb6206/infrastructure/bc_trainer.py`

Once you have filled in all of the instructions specified with `TODO` comments in the code, you should be able to train DAgger with the following command:

```
python gcb6206/scripts/run_hw1.py \
--expert_policy_file gcb6206/policies/experts/Ant.pkl \
--env_name Ant-v4 --exp_name dagger_ant --n_iter 10 \
--do_dagger \
--expert_data gcb6206/expert_data/expert_data_Ant-v4.pkl \
--video_log_freq -1
```

6.2 Compare BC and DAgger

Run DAgger and report results on the two tasks you tested previously with BC (i.e., Ant + another environment). Report your results in the form of a learning curve, plotting the number of DAgger iterations vs. the policy's mean return. In the caption, state which task you used, and any details regarding network architecture, amount of data, etc. (as in the previous section).

Note: You can use the example helper script (`gcb6206/scripts/parse_tensorboard.py`) to parse the data from the tensorboard logs and plot the figure. Here's an example usage that saves the figure as `output_plot.png`:

```
python gcb6206/scripts/parse_tensorboard.py \
--input_log_files
data/[replace_here_with_the_name_of_log_folder] \
--data_key "Eval_AverageReturn" \
--title "DAgger: Ant-v4" \
--x_label_name "DAgger iterations" \
--y_label_name "Mean Return" \
--output_file "output_plot.png"
```

You may also want to plot the performances of BC and expert policy as a horizontal line and plot the standard deviations as the error bars. Feel free to modify the example parsing script as you want.

analysis on the result

6.2.1 DAgger Result

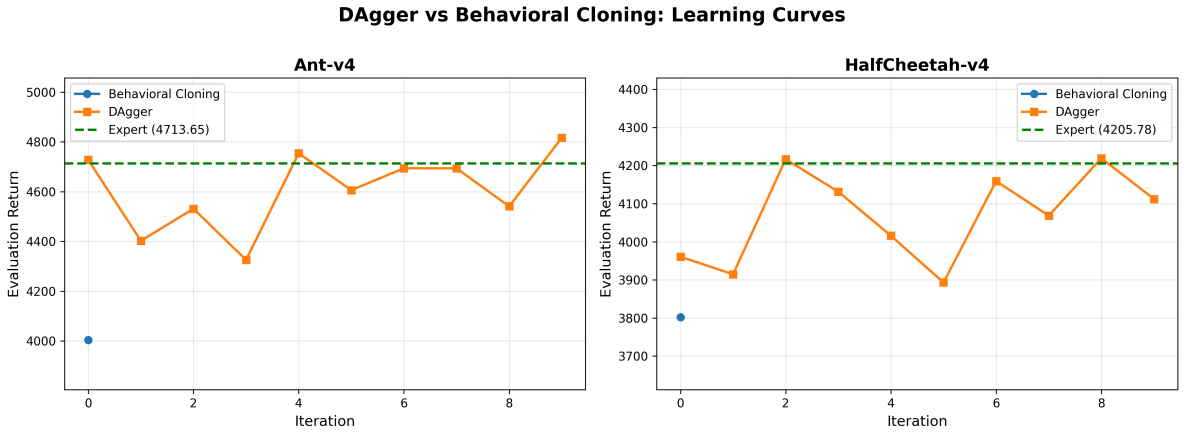

Figure 3: Learning curves for DAgger (orange) and BC (blue) across 10 iterations on `Ant-v4` (left) and `HalfCheetah-v4` (right). Both use the same network architecture—2 hidden layers with 64 units and tanh activation. BC trains once on 5000 expert timesteps, while DAgger adds 1000 new timesteps per iteration with expert relabeling. The green dashed line shows expert performance. You can see DAgger's advantage over BC pretty clearly: it hits 102.2% of expert level on Ant (4816.39 vs 4713.65 expert) and 97.8% on HalfCheetah (4112.30 vs 4205.78), while BC only reaches 84.9% and 90.4% respectively.

6.2.2 Justification of the Result

DAgger beats BC on both tasks by solving the distributional shift problem. Standard BC fails when the learned policy deviates from what it saw in the training data—small mistakes snowball because the policy never learned how to recover from these states.

Environment	BC	DAgger	Expert
`Ant-v4`	4004.17 (84.9%)	**4816.39 (102.2%)**	4713.65
`HalfCheetah-v4`	3801.78 (90.4%)	**4112.30 (97.8%)**	4205.78

The key difference is how I collected training data. BC trains once on the expert demonstrations and that's it. DAgger uses iterative aggregation—for iteration 0, I trained on the original expert data (same as BC). Then for iterations 1-9, I rolled out the current policy to see where it actually goes (including mistakes), and queried the expert for the right actions in those states. Each iteration adds more training examples from states the policy really encounters, including recovery behaviors.

This way, the policy learns to correct errors instead of just copying the expert's ideal trajectory.

Looking at the numbers, DAgger hit 4816.39 on Ant-v4—that's 102.2% of expert and 20.3% better than BC's 4004.17. I think this happened because the iterative collection lets the agent find and handle scenarios that weren't in the original expert demos. The neural network seems to generalize these patterns pretty well across different states.

HalfCheetah-v4 was different. DAgger reached 97.8% of expert (8.2% improvement over BC's 90.4%). Didn't beat the expert here, but got a lot closer. The policy picked up recovery strategies for off-distribution states.

The learning curves are pretty noisy—DAgger bounces around between iterations because the rollouts sample different states each time. But by iteration 10, both environments stabilized near or above expert level, so the approach works for this task.

7 Submission

Please submit the **"report"** `hw1_[YourStudentID].pdf`.