CAS4160 Homework 1: Imitation Learning

Ingu Lee
2019126018

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, 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: Passes an action to the environment and the function returns the new observation, reward and whether the environment has terminated/truncated.

2. Describe the role of the `reset()` function in a gym environment. What is the return value of this function?
   Answer: Resets the environment to the initial state, and returns the initial observation and some extra information about it.

3. How can you figure out if some gym environment has a discrete action space or continuous action space?
   Answer: The action_space attribute of the environment object returns the type of it. For example, if env is environment object, env.action_space returns Discrete for the discrete action space, and Box(_) for the continuous action space.

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:

- `cas4160/infrastructure/bc_trainer.py`, except for `do_relabel_with_expert` function, which is for the next section, DAgger.
- `cas4160/policies/MLP_policy.py`
- `cas4160/infrastructure/replay_buffer.py`
- `cas4160/infrastructure/utils.py`
- `cas4160/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 cas4160/scripts/run_hw1.py \
    --expert_policy_file cas4160/policies/experts/Ant.pkl \
    --env_name Ant-v4 --exp_name bc_ant --n_iter 1 \
    --expert_data cas4160/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: `ep_len=1000`, `eval_batch_size=5000`, **others: default**

Environment	Performance (Mean Return ± Std)
Ant-v4	4,644.874 ± 130.682
HalfCheetah-v4	3,734.600 ± 64.945

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:

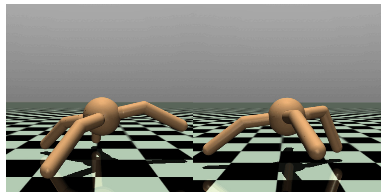

Figure 1: `ep_len=1000, eval_batch_size=5000`, **others: default**

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 cas4160/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: **The number of gradient steps for training policy per iteration.(num_agent_train_steps_per_iter)**

Values: `500, 1000(default), 1500, 2000`

Plot:

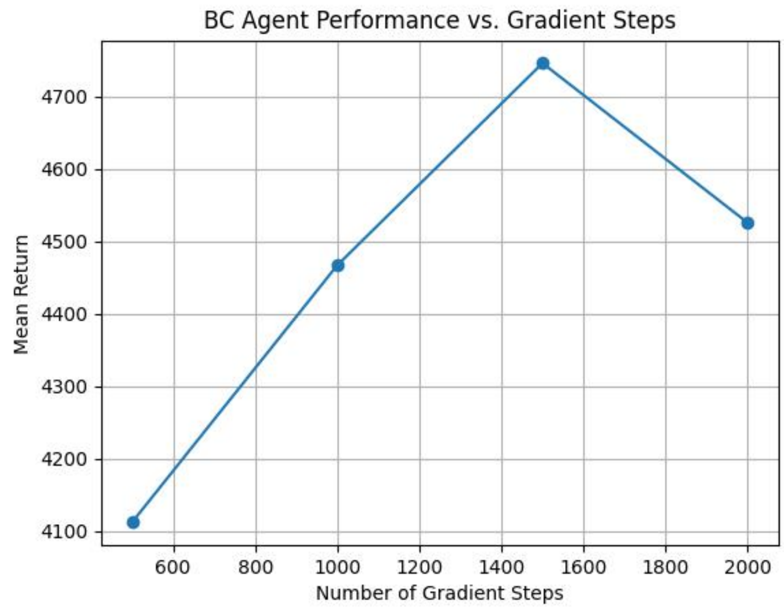

Figure 2: **Mean Return vs. Gradient Steps per Iteration**

Rationale: From this plot, we can infer that the optimal number of gradient steps per iteration is around 1500. We see that under 1500 steps, the performance degrades, likely due to underfitting, and also degrades at 2000 steps, suggesting overfitting.

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 `cas4160/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 cas4160/scripts/run_hw1.py \
  --expert_policy_file cas4160/policies/experts/Ant.pkl \
  --env_name Ant-v4 --exp_name dagger_ant --n_iter 10 \
  --do_dagger \
  --expert_data cas4160/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 (`cas4160/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 cas4160/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.

6.2.1 DAgger Result

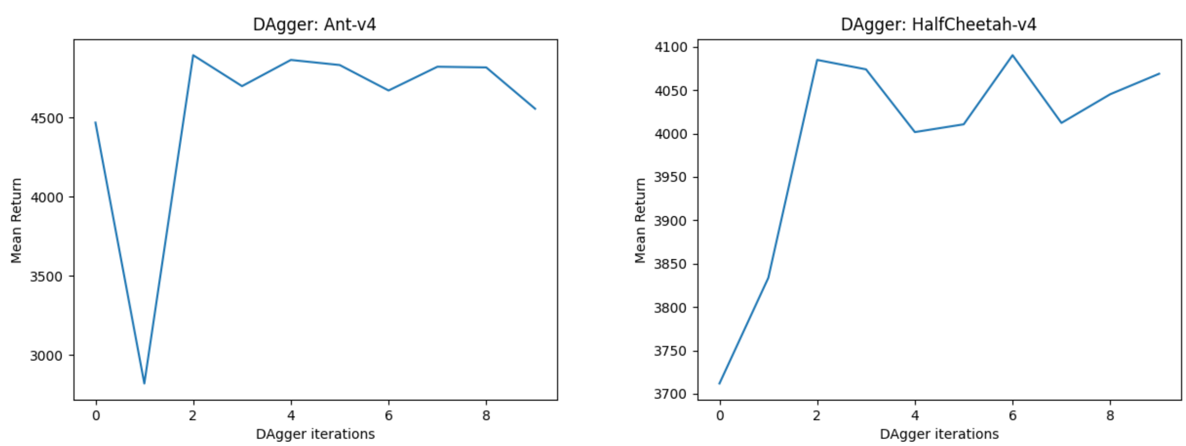

Figure 3: **Ant-v4 and HalfCheetah-v4 with DAgger Results, used default options**

6.2.2 Justification of the Result

Overall, Results with DAgger were superior to those with BC. In the Ant-v4 environment, it achieved a mean return of 4,644.874 (see 5.2.1.), while the policy with DAgger performed 4,683.944. Similarly, in the HalfCheetah-v4 environment, BC's mean return was 3,734.600, but with DAgger, it significantly improved to 4,048.863.

It is likely due to the ability of DAggerto address BC's limitation of compounding errors, by incorporating expert feedback into training, thereby outperforming BC.

7 Discussion

Please provide us a rough estimate, in hours, for each problem, how much time you spent. This will help us calibrate the difficulty for future homework.

- **Behavioral Cloning**: **1 week (approx. 24 hours)**
- **DAgger**: **1 hour**
- **Writing a report**: **12 hours**

Feel free to share your feedback here, if any: **The most time I spent was understanding the overall structure of the code I was given. The actual implementation took just a few hours. In my own opinion, I think it would have been better if we were given the TA session for explaining the parts that we should implement, rather than introduction to Vessl AI, etc.**

Plus, the Vessl AI never let me to execute the code due to the space limit, errors, etc. Setting up the environment in my local machine took almost a day, which was the second most challenging part.

8 Submission

Please submit the code, tensorboard logs, and the **"report"** in a single zip file, `hw1_2019126018.zip`. The zip file should be smaller than 20MB. The structure of the submission file should be:

```
hw1_2019126018.zip
├─ hw1_2019126018.pdf
├─ cas4160
│  ├─ agents
│  │  └─ bc_agent.py
│  └─ ...
├─ data
│  └─ ...tensorboard log folders
└─ ...
```

Note: Do NOT include the videos (.mp4 files) in your submission.