XCS224R Assignment 1

Due Sunday, February 22 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs231n-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the environment setup for the coding problems, please refer to the `README.md` file in the assignment directory.
4. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. For SCPD classes, it is also important that students avoid opening pull requests containing their solution code on the shared assignment repositories. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START CODE HERE ###` and `### END CODE HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are NOT visible locally. These hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned. These tests will evaluate elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None   <-- This error caused the test to fail.
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0   <-- In this case, start your debugging in line 23 of grader.py.
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

Remote Execution

Basic and hidden tests are treated the same by the remote autograder, however the output of hidden tests will only appear once you upload your code to GradeScope. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

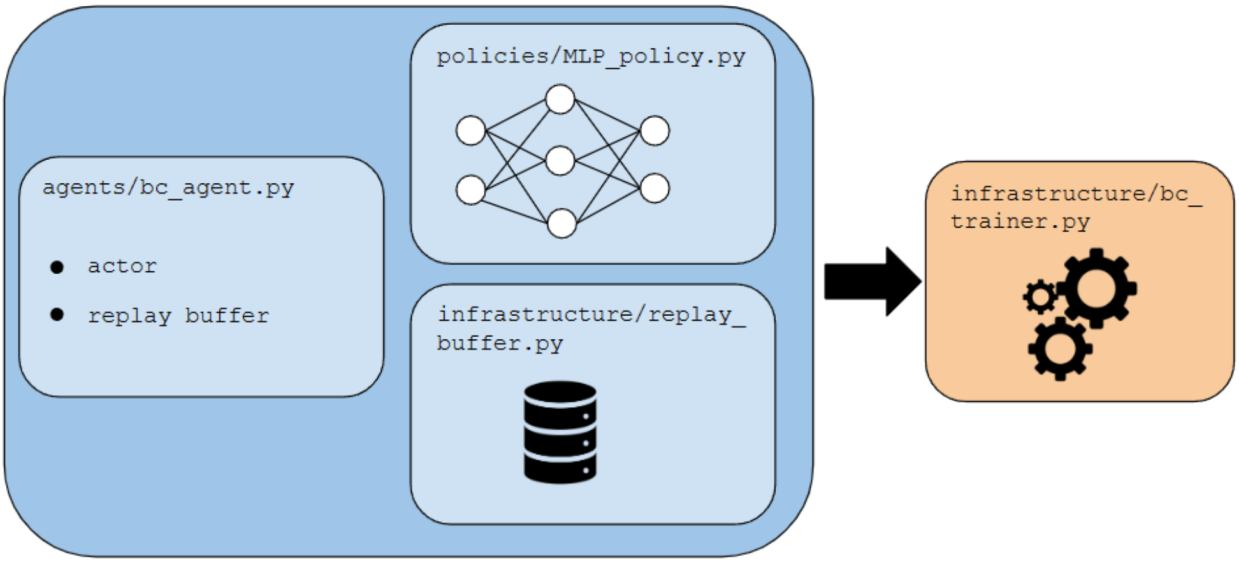
1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

0 Goals and Codebase

In this assignment you will experiment with two common methods of imitation learning: behavior cloning and DAgger. You will then have the opportunity to train and tune your agents in a variety of environments in the MuJoCo physics simulator (https://mujoco.org/). In lieu of a human demonstrator, we will provide demonstrations from a pre-trained expert policy.

1. Implement and train an imitation learning agent that can learn from expert data.
2. Analyze the shortcomings of behavior cloning and how DAgger can address some of these issues.
3. Experiment with hyperparameters and explore how they affect performance.



We have provided you with starter code for this homework. A simplistic diagram of its structure is shown above. In the `agents` directory, you will define your main behavior cloning agent. This agent will contain contain references to its policy (or actor) and replay buffer. This agent will be fed into the BC trainer, which will contain its training loop.

We recommend that you read the files from the `submission` folder in the following order. Your task is to fill in the blanks labeled `TODO`.

- `run_hw1.py (read-only file)`
- `xcs224r/infrastructure/bc_trainer.py`
- `xcs224r/agents/bc_agent.py (read-only file)`
- `xcs224r/policies/MLP_policy.py`
- `xcs224r/infrastructure/replay_buffer.py`
- `xcs224r/infrastructure/utils.py`
- `xcs224r/infrastructure/pytorch_util.py`

Use of GPT/Codex/Copilot/Any AI code assistant/agent: For the sake of deeper understanding on implementing imitation learning methods, assistance from generative models to write code for this homework is prohibited.

You will submit the entire `submission` directory. To do so run the command below from the `src` directory

```
bash collect_submission.sh
```

or you can choose to zip the folder up manually.

1 Behavior Cloning

Behavior Cloning is the simplest form of Imitation Learning, where the agent learns a policy by treating the problem as a supervised learning task rather than a trial-and-error reinforcement challenge. In this approach, an expert provides a dataset of demonstrations consisting of specific state-action pairs `(s,a)`, and the agent is trained to learn a mapping function (policy) that mimics the expert's actions given the same states.

Unlike standard Reinforcement Learning, Behavior Cloning does not require a complex reward function or active exploration of the environment during training. Instead, the agent simply minimizes a loss function to replicate the expert's decisions.

(a) **[22 points (Coding)] Behavior Cloning Coding**

Implement the following functions within the below mentioned files to finish the implementation:

- `infrastructure/pytorch_util.py`
  - `build_mlp`
- `policies/MLP_policy.py`
  - `MLPPolicySL::forward`
  - `MLPPolicySL::get_action`
  - `MLPPolicySL::update`
  - `MLPPolicySL::update`
- `infrastructure/utils.py`
  - `sample_trajectory`
  - `sample_trajectories`
  - `sample_n_trajectories`
- `infrastructure/replay_buffer.py`
  - `ReplayBuffer::sample_random_data`
- `infrastructure/bc_trainer.py`
  - `BCTrainer::collect_training_trajectories`
  - `BCTrainer::train_agent`

(b) **[5 points (Written)] Run Behavior Cloning**

Run behavioral cloning (BC) and report results on two tasks: the Ant environment, where a behavioral cloning agent should achieve at least 30% of the performance of the expert, and one environment of your choosing where it does not. For your reference, the other environments with expert data include Hopper, Walker2d, and HalfCheetah. A policy that achieves greater than 30% of the expert on the Ant task will receive full credit on the autograder. Here is how you can run the Ant task:

```
python run_hw1.py \
    --expert_policy_file xcs224r/policies/experts/Ant.pkl \
    --env_name Ant-v4 --exp_name bc_ant --n_iter 1 \
    --expert_data xcs224r/expert_data/expert_data_Ant-v4.pkl \
    --video_log_freq -1
```

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.

Note: To report the mean and standard deviation, your eval batch size should be greater than `ep_len`, so 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 trajectories (maybe more if any of them terminate early), and the logged `Eval_AverageReturn` and `Eval_StdReturn` represents the mean and standard deviation of your policy over these 5 rollouts.

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

(c) **[5 points (Written)] Behavior Cloning Experiment**

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 one or two sentence rationale for why you chose it. The plot should contain clearly labeled axes.

Include your chosen hyperparameter, plot, and rationale here.

2 DAgger (Dataset Aggregation)

In standard Behavior Cloning, the agent is only trained on states the expert visited, drifting into unfamiliar territory and failing. DAgger changes this by letting the agent interact with the world, collecting the states the agent actually visits, including its mistakes.

(a) **[8 points (Coding)] DAgger Coding**

Implement DAgger by filling the `do_relabel_with_expert` function within `infrastructure/bc_trainer.py` file

(b) **[10 points (Written)] Run DAgger**

```
python run_hw1.py \
  --expert_policy_file xcs224r/policies/experts/Ant.pkl \
  --env_name Ant-v4 --exp_name dagger_ant --n_iter 10 \
  --do_dagger \
  --expert_data xcs224r/expert_data/expert_data_Ant-v4.pkl \
  --video_log_freq -1
```

Run DAgger and report results on the two tasks you tested previously with behavioral cloning (i.e., Ant + another environment). Report your results in the form of a learning curve, plotting the number of DAgger iterations on the x-axis versus the policy's mean return on the y-axis, with error bars to show the standard deviation. Include the performance of the expert policy and your behavioral cloning agent on the same plot. State which task you used, and any details regarding network architecture, amount of data, etc. (as in the previous section).