Introduction to deep reinforcement learning: HW1

Tue Herlau
tuhe@dtu.dk

June 6, 2019

1 Objective

The goal of this assignment[1] is to experiment with imitation learning, including direct behavior cloning and the DAgger algorithm. In lieu of a human demonstrator, demonstrations will be provided via an expert policy that we have trained for you. Your goals will be to set up behavior cloning and DAgger, and compare their performance on a few different continuous control tasks from the OpenAI Gym benchmark suite. Turn in your report and code as described in Section 5.

2 Getting set up

The starter code can be found at `https://github.com/berkeleydeeprlcourse/homework/tree/master/hw1`. There are three dependencies described below. For this assignment and others in the course, you may use whatever deep learning library you wish, but your code should be in Python. We would strongly appreciate avoiding unusual dependencies (beyond Theano, TensorFlow, and other standard DL libraries). If your code does have particular additional dependencies, please be sure to document these so that we can run your code.

Tensorflow: Follow the instructions at `https://www.tensorflow.org/get_started/os_setup` to install TensorFlow. We will not require a GPU for assignments in this course, but if you have one, you may want to consider installing the GPU version along with CUDA and CuDNN.

OpenAI Gym: We will use environments in OpenAI Gym, a testbed for reinforcement learning algorithms. For installation and usage instructions see `https://gym.openai.com/docs`.

Pybullet/pybullet gym: Pybullet is a python interface for the physics engine Bullet and provides many of the virtual robots as MuJoCo, a proprietary physics engine, see `https://github.com/benelot/pybullet-gym`. Pybylletgym is an interface with openai gym.

[1]Note the text and material for this assignment is a near verbatim copy of Berkleys deep learning course, see `http://rail.eecs.berkeley.edu/deeprlcourse/`.

Baselines: To generate some of the experts, we will use the openai baselines implementations of state-of-the-art reinforcement learning algorithms, see `https://github.com/openai/baselines`. Note there is a difference between the implementation supplied in pip and on the github repo, and it is very important we use the github version. Therefore, install baselines using:

Command Line
```
$ git clone https://github.com/openai/baselines
```

after git does it's magic, include the baselines directory in your `PYTHONPATH` and check you can run `import baselines` from python

3 Behavioral Cloning

Our first task is to implement behavioral cloning. In order to clone an expert, we obviously need an expert to clone (recall an expert is just a policy, i.e. $\pi : s \mapsto a$). In this report, we will consider two different types of experts:

- Pre-trained experts supplied by pybulletgym, which we have copied to our source directory. The code that load these can be found in `agent_zoo_pybulletgym/enjoy_TF.py`. We denote these by π_e.
- Experts that you train using openai baselines and the state-of-the-art method ppo2. You do not have to understand ppo2, but it is a good idea that you familiarize yourself with how to call the baselines implementations, as it would very likely be these you would compare your own methods against. Inspect the file `baselines_train_expert.py`, which both contains code to train (and save) an expert, as well as load the expert again. Try to play around with the main function in the file to see how you can train agents and visualize the accumulated reward. For instance, this code will train an expert for 10^6 steps (of the environment) and save the experts weights to `agent_zoo_baseline/`. We denote these by π_b

```
if __name__ == "__main__":
        ev = 'InvertedPendulumPyBulletEnv-v0'
        train_expert(ev, '1e6')
```

Take care to check traceplots of the reward (see code; compare this against the pre-trained experts) to ensure your 'experts' actually learn something. Finally note this file can also allow you to obtain a baseline by simply training the expert for very few time steps.

In the following tasks, we recommend starting with the pre-trained experts.

1. Generate rollouts of your baselines experts (rollouts are collections of states and actions by the expert) using `run_expert.py`. See file for details.

2. Implement behavioral cloning (BC) and report results on two tasks – one task where a behavioral cloning agent achieves comparable performance to the expert, and one task where it does not. When providing results, report the mean and standard deviation of the return over multiple rollouts in a table, and state which task was used. Note you can use `run_expert.py` for this.

   Be sure to set up a fair comparison, in terms of network size, amount of data, and number of training iterations, and provide these details (and any others you feel are appropriate) in the table caption.

3. Experiment with one hyperparameter that affects the performance of the behavioral cloning agent, such as the number of demonstrations, the number of training epochs, 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, and state the hyperparameter and a brief rationale for why you chose it in the caption for the graph.

4. Train a baselines agents on your selected environments using `baselines_train_expert.py`. Plot the learning curves of the agents to ensure they obtain a non-trivial performance.

5. Re-run behavioral cloning on the agent you just trained (use the included load function) on the same two environments and compute the mean/std of it's performance.

6. Also use the baselines interface in `baselines_train_expert.py`, but with a very low number of training iterations, to obtain a baseline performance on the two environments.

4 DAgger

1. Implement DAgger. See the code provided in `run_expert.py` to see how to query the expert policy and perform roll-outs in the environment.

2. Run DAgger and report results on one task in which DAgger can learn a better policy than behavioral cloning. Report your results in the form of a learning curve, plotting the number of DAgger iterations vs. the policy's mean return, with error bars to show the standard deviation (you can use `run_expert.py:evaluate_policy` for this). Include the performance of the expert policy and the behavioral cloning agent on the same plot. In the caption, state which task you used, and any details regarding network architecture, amount of data, etc. (as in the previous section).

5 Turning it in

1. Your report should be a PDF with 1 or 2 pages, only consisting of figures, tables and captions. You should not write anything else, i.e. no text outside captions. We have included a template of the report in the following section, and we recommend you download the .tex file from gitlab and re-use it.

2. Since your grade is individualized, you should use the attribution table to specify what parts of the report each student was responsible for. Note for each item, one student has to be the main responsible in % (with some margin). The table will only be used to individualize grades in the rare case one students performance is so significantly different than the others we can make a clear determination.

3. Due to the high variance of reinforcement learning algorithms it is important to guard against seed hacking by resisting the temptation to re-do bad experiments and by including the standard deviation of the obtained results. The standard deviation of some random quantity X from which we have samples $x_1, \dots, x_N \sim X$ is computed as:

$$\hat{\sigma} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mathbb{E}[X])^2}.$$

   Where $\mathbb{E}[X] = \frac{1}{N}\sum_{i=1}^{N} x_i$ is the mean. Normally one would then report numbers in either of the forms: $\mathbb{E}[X] \pm \hat{\sigma}$ or $\underset{(\pm\hat{\sigma})}{\mathbb{E}[X]}$ as in the table template.

   See the handout at `http://rail.eecs.berkeley.edu/deeprlcourse/static/misc/viz.pdf` for notes on how to generate plots.

Attribution table

Part	Responsible
Algorithm A	Donald (30%), Mickey (70%)
Plot X	Goofy
Plot Y	Donald (70%), Goofy (30%)

Figures and tables

Task	Reward π_e	BC reward π_e	Reward π_b	BC reward π_b	Baseline reward
foo-env-v0	0.91 (± 1.93)	0.91 (± 1.93)	$\dots$		
bar-env-v0	0.91 (± 1.93)	0.91 (± 1.93)	$\dots$		
$\dots$	$\dots$	$\dots$	$\dots$		

Table 1: [Behavioral cloning results from section 3]. Training iterations: ... etc. etc. We conclude ...

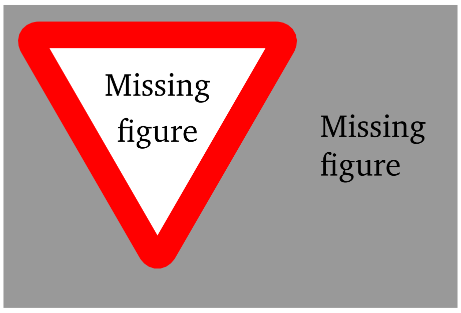

Figure 1: [Figure answering the question about hyper-parameters in section 3]. Plot of ...

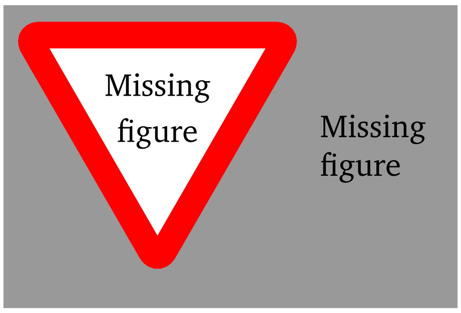

Figure 2: [Learning curve as a function of training iterations using DAgger from section 4] Plot of ...