Assignment 3: Q-Learning and Actor-Critic Algorithms

Due October 31, 11:59 pm

1 Part 1: Q-Learning

1.1 File overview

The starter code for this assignment can be found at

`https://github.com/LeCAR-Lab/16831-F24-HW.git`

We will be building on the code that we have implemented in the first two assignments. All files needed to run your code are in the `hw3` folder.

In order to implement deep Q-learning, you will be writing new code in the following files:

- In `agents/dqn_agent.py`, you will need to implement `step_env`. This function essentially combines taking an environment step and adding a transition to the memory-optimized replay buffer. You will also need to implement `train` to update the Q network and target network.
- In `critics/dqn_critic.py`, you will need to implement the `update` function to update the Q network, given a batch of data.
- In `policies/argmax_policy.py`, you will need to implement `get_action` to take the action that maximizes $Q(s, a)$ for the given observation s.

1.2 Implementation

The first phase of the assignment is to implement a working version of Q-learning. The default code will run `LunarLander-v3` with reasonable hyperparameter settings. Our reference solution with the default hyperparameters achieves around 125 reward after 300k timesteps, but there is considerable variation between runs and without the double-Q trick the average return often decreases after reaching 125.

1.3 Evaluation

Question 1: DQN and DDQN. Include a learning curve plot showing the performance of your implementation on `LunarLander-v3`. The x-axis should correspond to number of time steps (consider using scientific notation) and the y-axis should show the average per-epoch reward as well as the best mean reward so far. These quantities are already computed and printed in the starter code. They are also logged to the `data` folder, and can be visualized using Tensorboard as in previous assignments. Be sure to label the y-axis, since we need to verify that your implementation achieves similar reward as ours. You should not need to modify the default hyperparameters in order to obtain good performance, but if you modify any of the parameters, list them in the caption of the figure. Compare the performance of DDQN to vanilla DQN. Since there is considerable variance between runs, you must run at least three random seeds for both DQN and DDQN. (To be clear, there should be one line for DQN and one line for DDQN on the same plot, each with error bars). The final results should use the following experiment names:

```
python rob831/scripts/run_hw3_dqn.py --env_name LunarLander-v3 \
  --exp_name q1_dqn_1 --seed 1 --no_gpu
python rob831/scripts/run_hw3_dqn.py --env_name LunarLander-v3 \
  --exp_name q1_dqn_2 --seed 2 --no_gpu
python rob831/scripts/run_hw3_dqn.py --env_name LunarLander-v3 \
  --exp_name q1_dqn_3 --seed 3 --no_gpu
```

```
python rob831/scripts/run_hw3_dqn.py --env_name LunarLander-v3 \
  --exp_name q1_doubledqn_1 --double_q --seed 1 --no_gpu
python rob831/scripts/run_hw3_dqn.py --env_name LunarLander-v3 \
```

```
  --exp_name q1_doubledqn_2 --double_q --seed 2 --no_gpu
python rob831/scripts/run_hw3_dqn.py --env_name LunarLander-v3 \
  --exp_name q1_doubledqn_3 --double_q --seed 3 --no_gpu
```

Submit the run logs (in `rob831/data`) for all of the experiments above. In your report, make a single graph that averages the performance across three runs for both DQN and double DQN. See `scripts/read_results.py` for an example of how to read the evaluation returns from Tensorboard logs.

2 Part 2: Actor-Critic

2.1 Introduction

Part 2 of this assignment requires you to modify policy gradients (from hw2) to an actor-critic formulation. Note that evaluation may take longer for actor-critic than policy gradient (on half-cheetah) due to the significantly larger number of training steps for the value function.

Recall the policy gradient from hw2:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{it}|s_{it})\left(\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{it'},a_{it'})\right) - V_\phi^\pi\left(s_{it}\right)\right).$$

In this formulation, we estimate the Q function by taking the sum of rewards to go over each trajectory, and we subtract the value function baseline to obtain the advantage

$$A^\pi(s_t,a_t) \approx \left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{t'},a_{t'})\right) - V_\phi^\pi\left(s_t\right)$$

In practice, the estimated advantage value suffers from high variance. Actor-critic addresses this issue by using a *critic network* to estimate the sum of rewards to go. The most common type of critic network used is a value function, in which case our estimated advantage becomes

$$A^\pi(s_t,a_t) \approx r(s_t,a_t) + \gamma V_\phi^\pi\left(s_{t+1}\right) - V_\phi^\pi\left(s_t\right)$$

In this assignment we will use the same value function network from hw2 as the basis for our critic network. One additional consideration in actor-critic is updating the critic network itself. While we can use Monte Carlo rollouts to estimate the sum of rewards to go for updating the value function network, in practice we fit our value function to the following *target values*:

$$y_t = r(s_t,a_t) + \gamma V^\pi(s_{t+1})$$

we then regress onto these target values via the following regression objective which we can optimize with gradient descent:

$$\min_\phi \sum_{i,t}(V_\phi^\pi(s_{it}) - y_{it})^2$$

In theory, we need to perform this minimization every time we update our policy, so that our value function matches the behavior of the new policy. In practice however, this operation can be costly, so we may instead just take a few gradient steps at each iteration. Also note that since our target values are based on the old value function, we may need to recompute the targets with the updated value function, in the following fashion:

1. Update targets with current value function
2. Regress onto targets to update value function by taking a few gradient steps
3. Redo steps 1 and 2 several times

In all, the process of fitting the value function critic is an iterative process in which we go back and forth between computing target values and updating the value function to match the target values. Through experimentation, you will see that this iterative process is crucial for training the critic network.

2.2 Implementation

Your code will build off your solutions from homework 2. You will need to fill in the TODOS for the following parts of the code.

- In `policies/MLP_policy.py`, implement the `update` function for the class `MLPPolicyAC`. You should note that the AC policy class is in fact the same as the policy class you implemented in the policy gradient homework (except we no longer have a nn_baseline).
- In `agents/ac_agent.py`, finish the `train` function. This function should implement the necessary critic updates, estimate the advantage, and then update the policy. Log the final losses at the end so you can monitor it during training.
- In `agents/ac_agent.py`, finish the `estimate_advantage` function: this function uses the critic network to estimate the advantage values. The advantage values are computed according to

  $$A^\pi(s_t, a_t) \approx r(s_t, a_t) + \gamma V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t)$$

  Note: for terminal timesteps, you must make sure to cut off the reward to go (i.e., set it to zero), in which case we have

  $$A^\pi(s_t, a_t) \approx r(s_t, a_t) - V^\pi_\phi(s_t)$$

- `critics/bootstrapped_continuous_critic.py` complete the TODOS in `update`. In `update`, perform the critic update according to process outlined in the introduction. You must perform

  `self.num_grad_steps_per_target_update * self.num_target_updates`

  number of updates, and recompute the target values every `self.num_grad_steps_per_target_update` number of steps.

2.3 Evaluation

Once you have a working implementation of actor-critic, you should prepare a report. The report should consist of figures for the question below. You should turn in the report as one PDF (same PDF as part 1) and a zip file with your code (same zip file as part 1). If your code requires special instructions or dependencies to run, please include these in a file called `README` inside the zip file.

Question 2: Sanity check with Cartpole Now that you have implemented actor-critic, check that your solution works by running Cartpole-v0.

```
python rob831/scripts/run_hw3_actor_critic.py --env_name CartPole-v0 \
  -n 100 -b 1000 --exp_name q2_10_10 -ntu 10 -ngsptu 10 --no_gpu
```

The results should match the policy gradient results from Cartpole in hw2 (200). Your deliverable for this section is a plot with the eval returns. You may take use a TensorBoard screenshot with smoothing = 0.

Question 3: Run actor-critic with InvertedPendulum Run InvertedPendulum:

```
python rob831/scripts/run_hw3_actor_critic.py --env_name InvertedPendulum-v4 \
  --ep_len 1000 --discount 0.95 -n 100 -l 2 -s 64 -b 5000 -lr 0.01 \
  --exp_name q3_10_10 -ntu 10 -ngsptu 10 --no_gpu
```

Your results should roughly match those of policy gradient. After 100 iterations, your InvertedPendulum return should be around 1000. The policy performance may fluctuate around 1000; this is fine. Your deliverable for this section is a plot with the eval returns. You may use a Tensorboard screenshot with smoothing = 0.

As a debugging tip, the returns should start going up immediately. For example, after 20 iterations, your InvertedPendulum return should be near or above 100.

3 Submitting the code and experiment runs

In order to turn in your code and experiment logs, create a folder that contains the following:

- A folder named `data` with all the experiment runs from this assignment. **Do not change the names originally assigned to the folders, as specified by `exp_name` in the instructions. Video logging is disabled by default in the code, but if you turned it on for debugging, you will need to run those again with `--video_log_freq -1`, or else the file size will be too large for submission.**
- The `rob831` folder with all the `.py` files, with the same names and directory structure as the original homework repository (excluding the `data` folder). Also include any special instructions we need to run in order to produce each of your figures or tables (e.g. "run python myassignment.py -sec2q1" to generate the result for Section 2 Question 1) in the form of a README file.

As an example, the unzipped version of your submission should result in the following file structure. **Make sure that the submit.zip file is below 15MB and that they include the prefix `q1_`, `q2_`, `q3_`, etc.**

```
submit.zip
├─ data
│  ├─ q1_dqn_...
│  │  └─ events.out.tfevents.1567529456.e3a096ac8ff4
│  ├─ q2_ac_...
│  │  └─ events.out.tfevents.1567529456.e3a096ac8ff4
│  └─ ...
├─ rob831
│  ├─ agents
│  │  ├─ ac_agent.py
│  │  └─ ...
│  ├─ policies
│  │  └─ ...
│  └─ ...
├─ README.md
└─ ...
```

If you are a Mac user, **do not use the default "Compress" option to create the zip**. You may use `zip -vr submit.zip submit -x "*.DS_Store"` from your terminal.

Turn in your assignment on Gradescope. Upload the zip file with your code and log files to **HW3 Code**, and upload the PDF of your report to **HW3**.