Assignment 3: Q-Learning and Actor-Critic Algorithms

Due October 18, 11:59 pm

1 Part 1: Q-Learning

1.1 Introduction

Part 1 of this assignment requires you to implement and evaluate Q-learning for playing Atari games. The Q-learning algorithm was covered in lecture, and you will be provided with starter code. This assignment will be faster to run on a GPU, though it is possible to complete on a CPU as well. Note that we use convolutional neural network architectures in this assignment. Therefore, we recommend using the Colab option if you do not have a GPU available to you. Please start early!

1.2 File overview

The starter code for this assignment can be found at

`https://github.com/berkeleydeeprlcourse/homework_fall2022/tree/master/hw3`

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, but there will be some blanks you will fill with your solutions from homework 1. These locations are marked with `# TODO: get this from hw1 or hw2` and are found in the following files:

- `infrastructure/rl_trainer.py`
- `infrastructure/utils.py`
- `policies/MLP_policy.py`

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

- `agents/dqn_agent.py`
- `critics/dqn_critic.py`
- `policies/argmax_policy.py`
- `policies/sac_agent.py`
- `policies/sac_critic.py`
- `policies/sac_policy.py`

There are two new package requirements (`gym[atari]` and `pip install gym[accept-rom-license]`) beyond what was used in the first two assignments; make sure to install these with `pip install -r requirements.txt` if you are running the assignment locally.

1.3 Implementation

The first phase of the assignment is to implement a working version of Q-learning. The default code will run the `Ms. Pac-Man` game with reasonable hyperparameter settings. Look for the `# TODO` markers in the files listed above for detailed implementation instructions. You may want to look inside `infrastructure/dqn_utils.py` to understand how the (memory-optimized) replay buffer works, but you will not need to modify it.

Once you implement Q-learning, answering some of the questions may require changing hyperparameters, neural network architectures, and the game, which should be done by changing the command line arguments passed to `run_hw3_dqn.py` or by modifying the parameters of the `Args` class from within the Colab notebook.

To determine if your implementation of Q-learning is correct, you should run it with the default hyperparameters on the `Ms. Pac-Man` game for 1 million steps using the command below. Our reference solution gets a

return of 1500 in this timeframe. On Colab, this will take roughly 3 GPU hours. If it takes much longer than that, there may be a bug in your implementation.

To accelerate debugging, you may also test on `LunarLander-v3`, which trains your agent to play Lunar Lander, a 1979 arcade game (also made by Atari) that has been implemented in OpenAI Gym. Our reference solution with the default hyperparameters achieves around 150 reward after 350k timesteps, but there is considerable variation between runs and without the double-Q trick the average return often decreases after reaching 150. We recommend using `LunarLander-v3` to check the correctness of your code before running longer experiments with `MsPacman-v0`.

1.4 Evaluation

Once you have a working implementation of Q-learning, you should prepare a report. The report should consist of one figure for each question below. You should turn in the report as one PDF and a zip file with your code. If your code requires special instructions or dependencies to run, please include these in a file called `README` inside the zip file.

Question 1: basic Q-learning performance (DQN). Include a learning curve plot showing the performance of your implementation on `Ms. Pac-Man`. 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. The final results should use the following experiment name:

```
python cs285/scripts/run_hw3_dqn.py --env_name MsPacman-v0 --exp_name q1
```

Question 2: double Q-learning (DDQN). Use the double estimator to improve the accuracy of your learned Q values. This amounts to using the online Q network (instead of the target Q network) to select the best action when computing target values. 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. You may use `LunarLander-v3` for this question. The final results should use the following experiment names:

```
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q2_dqn_1 --seed 1
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q2_dqn_2 --seed 2
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q2_dqn_3 --seed 3
```

```
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q2_doubledqn_1 --
    double_q --seed 1
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q2_doubledqn_2 --
    double_q --seed 2
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q2_doubledqn_3 --
    double_q --seed 3
```

Submit the run logs (in `cs285/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.

Question 3: experimenting with hyperparameters. Now let's analyze the sensitivity of Q-learning to hyperparameters. Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter, in addition to the one used in Question 1, and plot all four values on the same graph. Your choice what you experiment with, but you should explain why you chose this hyperparameter in the caption. Examples include: (1) learning rates; (2) neural network architecture for the Q network, e.g., number of layers,

hidden layer size, etc; (3) exploration schedule or exploration rule (e.g. you may implement an alternative to ϵ-greedy and set different values of hyperparameters), etc. Discuss the effect of this hyperparameter on performance in the caption. You should find a hyperparameter that makes a nontrivial difference on performance. Note: you might consider performing a hyperparameter sweep for getting good results in Question 1, in which case it's fine to just include the results of this sweep for Question 3 as well, while plotting only the best hyperparameter setting in Question 1. The final results should use the following experiment name:

```
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q3_hparam1
```

```
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q3_hparam2
```

```
python cs285/scripts/run_hw3_dqn.py --env_name LunarLander-v3 --exp_name q3_hparam3
```

You can replace `LunarLander-v3` with `PongNoFrameskip-v4` or `MsPacman-v0` if you would like to test on a different environment.

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

```
python cs285/scripts/run_hw3_actor_critic.py --env_name CartPole-v0 -n 100 -b 1000 --
    exp_name q4_ac_1_1 -ntu 1 -ngsptu 1
```

In the example above, we alternate between performing one target update and one gradient update step for the critic. As you will see, this probably doesn't work, and you need to increase both the number of target updates and number of gradient updates. Compare the results for the following settings and report which worked best. Do this by plotting all the runs on a single plot and writing your takeaway in the caption.

```
python cs285/scripts/run_hw3_actor_critic.py --env_name CartPole-v0 -n 100 -b 1000 --
    exp_name q4_100_1 -ntu 100 -ngsptu 1
```

```
python cs285/scripts/run_hw3_actor_critic.py --env_name CartPole-v0 -n 100 -b 1000 --
    exp_name q4_1_100 -ntu 1 -ngsptu 100
```

```
python cs285/scripts/run_hw3_actor_critic.py --env_name CartPole-v0 -n 100 -b 1000 --
    exp_name q4_10_10 -ntu 10 -ngsptu 10
```

At the end, the best setting from above should match the policy gradient results from Cartpole in hw2 (200).

Question 5: Run soft actor-critic with more difficult tasks Use the best setting from the previous question to run InvertedPendulum and HalfCheetah:

```
python cs285/scripts/run_hw3_actor_critic.py --env_name InvertedPendulum-v2 --ep_len 1000
    --discount 0.95 -n 100 -l 2 -s 64 -b 5000 -lr 0.01 --exp_name q5_<ntu>_<ngsptu> -ntu <>
     -ngsptu <>
```

where `<ntu>_<ngsptu>` is replaced with the parameters you chose.

```
python cs285/scripts/run_hw3_actor_critic.py --env_name HalfCheetah-v2 --ep_len 150 --
    discount 0.90 --scalar_log_freq 1 -n 150 -l 2 -s 32 -b 30000 -eb 1500 -lr 0.02 --
    exp_name q5_<ntu>_<ngsptu> -ntu <> -ngsptu <>
```

Your results should roughly match those of policy gradient. After 150 iterations, your HalfCheetah return should be around 150. After 100 iterations, your InvertedPendulum return should be around 1000. Your deliverables for this section are plots with the eval returns for both enviornments.

As a debugging tip, the returns should start going up immediately. For example, after 20 iterations, your HalfCheetah return should be above -40 and your InvertedPendulum return should near or above 100. However, there is some variance between runs, so the 150-iteration (for HalfCheetah) and 100-iteration (for InvertedPendulum) results are the numbers we use to grade.

3 Part 3: Soft Actor-Critic

Part 3 of this assignment requires you to implement a "soft" actor critic (SAC), an algorithm that optimizes the stochastic policy in an off-policy way. Its central feature is entropy regularization, where the agent gets a bonus reward at each time step proportional to the entropy of the policy at that time step. Other tricks that it incorporates include the clipped double-Q trick and target policy smoothing.

Entropy denotes how random a random variable is. If we denote X as a random variable with probability mass or density function p, the entropy H of X is computed from its distribution p according to:

$$H(X) = \underset{x\sim p}{E}[-\log p(x)].$$

We will fit our value function to the following *target values* incorporating entropy regularization. Note that we use the double Q-trick to get the minimum of the two Q value estimates (d stands for terminal):

$$y_t = r(s_t, a_t) + \gamma(1-d)(\min_{j=1,2} Q_{\text{target}}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}, s_{t+1}))$$

.

Feel free to read more in the SAC paper: `https://arxiv.org/abs/1801.01290`

3.1 Implementation

You will need to fill in the TODOS for the following parts of the code.

- In `agents/sac_agent.py`, implement the `update_critic` and `train` functions for the class `SACAgent`. You will need to compute target Q and target V values with entropy here.

- In `critics/sac_critic.py`, implement the `forward` function for the class.
- In `policies/sac_policy.py`, implement the `get_action` and `update` functions for class `MLPPolicySAC`. You will also need to update your regularizer (alpha) values.
- In `policies/rl_trainer.py`, implement `train_agent`.

3.2 Evaluation

Question 6: Run soft actor-critic more difficult tasks. Use the best setting from the previous question to run InvertedPendulum and HalfCheetah. You may use InvertedPendulum as a debugging environment, as it is much faster to train.

```
python cs285/scripts/run_hw3_sac.py \
    --env_name InvertedPendulum-v4 --ep_len 1000 \
    --discount 0.99 --scalar_log_freq 1000 \
    -n 100000 -l 2 -s 256 -b 1000 -eb 2000 \
    -lr 0.0003 --init_temperature 0.1 ----exp_name q6a_sac_InvertedPendulum_<parameters> \
    --seed 1
```

where `<parameters>` is replaced with the parameters you chose.

```
python cs285/scripts/run_hw3_sac.py \
    --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.99 --scalar_log_freq 1500 \
    -n 2000000 -l 2 -s 256 -b 1500 -eb 1500 \
    -lr 0.0003 --init_temperature 0.1 --exp_name q6b_sac_HalfCheetah_<parameters> \
    --seed 1
```

Here the number of iterations stands for the number of environment steps taken. Your results should roughly match and exceed those of policy gradient. After 50000 steps, your HalfCheetah return should be around 200. After 20000 steps, your InvertedPendulum return should reach 1000. Your deliverables for this section are plots with the eval returns for both enviornments.

As a debugging tip, the returns should start going up immediately. For example, after 10000 steps, your HalfCheetah return should be above -40 (trending toward positive) and your InvertedPendulum return should be near or above 100. However, there is some variance between seeds, so the 1000 eval average return under 100000 steps (for InvertedPendulum) and 300 average return under 200000 steps (for HalfCheetah) results are the numbers we use to grade.

4 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 `cs285` 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.**

If you are a Mac user, **do not use the default "Compress" option to create the zip**. It creates artifacts that the autograder does not like. 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**.