CS294-112 Deep Reinforcement Learning HW3: Q-Learning and Actor-Critic

Due October 10th, 11:59 pm

1 Part 1: Q-Learning

1.1 Introduction

Part 1 of this assignment requires you to implement and evaluate Q-learning with convolutional neural networks for playing Atari games. The Q-learning algorithm was covered in lecture, and you will be provided with starter code. You may modify the code to use any automatic differentiation package you want, though the default code uses TensorFlow, and you may find it easier to use that. You may run the code either on GPU or CPU. A GPU machine will be faster, but you should be able to get good results with about 20 hours of compute on a modern CPU.

Please start early! The questions will require you to perform multiple runs of Q-learning, each of which can take quite a long time. Furthermore, depending on your implementation, you may find it necessary to tweak some of the parameters, such as learning rates or exploration schedules, which can also be very time consuming. The actual coding for this assignment will involve about 50 lines of code, but the evaluation may take a very long time.

1.2 Installation

Obtain the code from `https://github.com/berkeleydeeprlcourse/homework/tree/master/hw3`. To run the code, go into the `hw3` directory and simply execute `python run_dqn_atari.py`. It will not work however until you finish implementing the algorithm in `dqn.py`.

You will also need to install the dependencies, which are OpenAI Gym, TensorFlow, and OpenCV (which is used to resize the images). Remember to also follow the instructions for installing the Atari environments for OpenAI gym, which can be found on the Github page for OpenAI Gym. To install OpenCV,

run `pip install opencv-python`. If you have trouble with `ffmpeg` you can install it via `homebrew` or `apt-get` depending on your system.

There are also some slight differences between different versions of TensorFlow in regard to initialization. If you get an error inside `dqn_utils.py` related to variable initialization, check the comment inside `initialize_interdependent_variables`, it explains how to modify the code to be compatible with older versions of TensorFlow.

You may want to look at `run_dqn_atari.py` before starting the implementation. This file defines the convolutional network you will be using for image-based Atari playing, defines which Atari game will be used (Pong is the default), and specifies the hyperparameters.

1.3 Implementation

The first phase of the assignment is to implement a working version of Q-learning. The default code will run the Pong game with reasonable hyperparameter settings. The starter code already provides you with a working replay buffer, all you have to do is fill in parts of `dqn.py`, by searching for `YOUR CODE HERE`. The comments in the code describe what should be implemented in each section. You may find it useful to look inside `dqn_utils.py` to understand how the replay buffer works, but you should not need to modify it. You may also look inside `run_dqn_atari.py` to change the hyperparameters or the particular choice of Atari game. 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 editing `run_dqn_atari.py`.

To determine if your implementation of Q-learning is performing well, you should run it with the default hyperparameters on the Pong game. Our reference solution gets a reward of around -20 to -15 after 500k steps, -15 to -10 after 1m steps, -10 to -5 after 1.5m steps, and around +10 after 2m steps on Pong. The maximum score of around +20 is reached after about 4-5m steps. However, there is considerable variation between runs.

To accelerate debugging, you may also check out `run_dqn_ram.py`, which runs the game Pong but using the state of the emulator RAM instead of images as observations. This version will run faster, especially if you're not using a GPU, but takes more iterations and probably won't converge to as good of a solution. You may use this version for debugging your algorithm, though you are not required to report results for it. Our reference solutions with the default hyperparameters gets around -19.9 after 500k steps, -19.2 after 1m steps, -15.0 after 1.5m steps, and -13.2 after 2m steps.

Another debugging option is provided in `run_dqn_lander.py`, 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 400k timesteps. We recommend using Lunar Lander to check the correctness of your code before running longer experiments with `run_dqn_ram.py` and `run_dqn_atari.py`.

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. For all the questions below, it is your choice how long to run for. Although running for 2-4m steps is ideal for a solid evaluation, especially when running on CPU, this may be difficult. We strongly recommend running at least 1m steps, and including at least one run of 4m steps for Question 1. If you have severe computational constraints and find that you are unable to run image-based Atari fast enough to complete Q2 or Q3, you may use Lunar Lander or the RAM version of Pong for Q2 or Q3 only, but Q1 results *must* use images. If you use Lunar Lander or the RAM version of Pong for Q2 or Q3, please specify this in the caption.

Question 1: basic Q-learning performance. Include a learning curve plot showing the performance of your implementation on the game Pong. The x-axis should correspond to number of time steps (consider using scientific notation) and the y-axis should show the mean 100-episode reward as well as the best mean reward. These quantities are already computed and printed in the starter code. Be sure to label the y-axis, since we need to verify that your implementation achieves similar reward as ours. If you needed to modify the default hyperparameters to obtain good performance, include the hyperparameters in the caption. You only need to list hyperparameters that were modified from the defaults.

Question 2: double Q-learning. Use the double estimator [1] 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 double DQN to vanilla DQN.

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: learning rates, neural network architecture, exploration schedule or exploration rule (e.g. you

may implement an alternative to ϵ-greedy), 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.

Note: for Questions 2 and 3, you may run on other games or multiple games, but please submit results for Question 1 using only the game Pong. Running on multiple games may require considerable time or computing power, so it is not required, but encouraged. If you have any other interesting experiments you wish to report on, you may include those after your answer to Question 2. Interesting additional experiments or extensions will be considered for bonus points.

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. Part 2 is relatively shorter than part 1. The actual coding for this assignment will involve less than 20 lines of code. Note however that evaluation may take longer for actor-critic than policy gradient 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 reward to go by taking the sum of rewards to go over each trajectory to estimate the Q function, and subtracting 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:

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

In theory, we need to perform this minimization everytime 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 Installation

Obtain the code from `https://github.com/berkeleydeeprlcourse/homework/tree/master/hw3`. To run the code, go into the `hw3` directory and simply execute `python train_ac_f18.py`.

You should have already installed all the required dependencies in hw2. Refer to that assignment for installation instructions if you have issues.

2.3 Implementation

We have taken the `python train_pg_f18.py` starter code from hw2 and modified it slightly to fit the framework of actor-critic. Core functions, such as `Agent.build_mlp`, `Agent.define_placeholders`, `Agent.policy_forward_pass`,

and `Agent.get_log_prob` remain unchanged from last time. This assignment requires that you use your solution code from hw2. Before you begin, go through `python train_ac_f18.py` and in all places marked `YOUR HW2 CODE HERE`, paste in your corresponding hw2 solution code.

In order to accommodate actor-critic, the following functions have been modified or added:

- `Agent.build_computation_graph`: we now have `actor_update_op` for updating the actor network, and `critic_update_op` for updating the critic network.
- `Agent.sample_trajectory`: in addition to logging the observations, actions, and rewards, we now need to log the next observation and terminal values in order to compute the advantage function and update the critic network. Please implement these features.
- `Agent.estimate_advantage`: 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, in which case we have

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

- `Agent.update_critic`: 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.

Go through the code and note the changes from policy gradient in detail. Then implement all requested features, which we have marked with `YOUR CODE HERE`.

2.4 Evaluation

Once you have a working implementation of actor-critic, 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 (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 1: Sanity check with Cartpole Now that you have implemented actor-critic, check that your solution works by running Cartpole-v0. Using the same parameters as we did in hw2, run the following:

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
↪  --exp_name 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. Provide a short explanation for your results.

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
↪  --exp_name 100_1 -ntu 100 -ngsptu 1
```

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
↪  --exp_name 1_100 -ntu 1 -ngsptu 100
```

```
python train_ac_f18.py CartPole-v0 -n 100 -b 1000 -e 3
↪  --exp_name 10_10 -ntu 10 -ngsptu 10
```

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

Question 2: Run actor-critic with more difficult tasks Use the best setting from the previous question to run InvertedPendulum and HalfCheetah:

```
python train_ac_f18.py InvertedPendulum-v2 -ep 1000 --discount
↪  0.95 -n 100 -e 3 -l 2 -s 64 -b 5000 -lr 0.01 --exp_name
↪  <>_<> -ntu <> -ngsptu <>
```

```
python train_ac_f18.py HalfCheetah-v2 -ep 150 --discount 0.90 -n
↪  100 -e 3 -l 2 -s 32 -b 30000 -lr 0.02 --exp_name <>_<> -ntu
↪  <> -ngsptu <>
```

Your results should roughly match those of policy gradient, perhaps a little bit worse in performance.

Bonus: You may have noticed that actor-critic does not perform as well as policy gradient in some instances. This is because the critic network may need to be more expressive and have a different learning rate than the one used to train the actor network. Try implementing a more expressive value function

network (more hidden units, more layers) and adopting a separate learning rate for training the critic network than the actor network. Also experiment with increasing the number of target updates or gradient steps for updating the critic network.

3 Submission

Turn in both parts of the assignment on Gradescope as one submission. Upload the zip file with your code to **HW3 Code**, and upload the PDF of your report to **HW3**.

References

[1] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.