Homework 2: Policy Gradients & DQN

CMU 10-703: Deep Reinforcement Learning (Fall 2022)

OUT: Wednesday, Sept. 28, 2022
DUE: Monday, Oct. 24, 2022 by 11:59pm EST

Instructions: START HERE

Note: this homework assignment requires a significant implementation effort. Please plan your time accordingly. General tips and suggestions are included in the 'Guidelines on Implementation' section at the end of this handout.

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 8 grace days for your homeworks. However, no more than 3 grace days may be applied to a single assignment. Any assignment submitted after 3 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy here for more information about grace days and late submissions: https://cmudeeprl.github.io/703website_f22/logistics/
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 2." Additionally, zip all the code folders into a directory titled `<andrew_id>.zip` and upload it the GradeScope assignment titled "Homework 2: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.
  - **Autolab:** Autolab is not used for this assignment.

This is a challenging assignment. **Please start early!**

[1] https://www.cmu.edu/policies/

Installation instructions (Linux)

For this assignment, we recommend using **Python 3.7 and above**. We've provided Python packages that you may need in `requirements.txt`. (Note: You may need to update `pytorch` to `torch` in the file if you're using pip. If you're having trouble installing packages needed to run the scripts, come to OH or post in Piazza. Generally, its fine to assume you already have the needed libraries installed). You should run with `gym<=0.21.0` for this assignment. To install these packages using pip and virtualenv, run the following commands: (for Mac, replace the first command with `brew install swig`)

```
apt-get install swig
virtualenv env
source env/bin/activate
pip install -U -r requirements.txt
```

Alternatively, install the packages in a new conda environment (e.g. named DRLhw2):

```
conda create -n DRLhw2 python=3.7
conda activate DRLhw2
conda install -c conda-forge --file requirements.txt
```

Introduction

The goal of this homework is to give you experience implementing and analysing Deep Reinforcement Learning algorithms. We'll start with one of the oldest Reinforcement Learning algorithms, REINFORCE, then build our way up to N-step Advantage Actor-Critic (A2C). We'll then compare these algorithms to Deep Q-Networks (DQN), a highly influential approach within the DRL community. Although these algorithms were proposed many years ago, they serve as the foundation of many current state of the art approaches.

Problem 0: Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

Problem 1: Policy Gradient Algorithms (48 pts)

Problem 1.1: REINFORCE (10 pts)

We begin this homework by implementing episodic REINFORCE [4], a policy-gradient RL algorithm. Please write your code in `a2c.py`; the template code should give you an idea on how you can structure your code for this problem and next two problems.

Policy gradient methods directly optimize over the policy $\pi_\theta(a|s)$ by performing a gradient-based optimisation scheme on the objective $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left\{\sum_{t=0}^{\infty}\gamma^t R_t\right\}$. The policy gradient, $\nabla_\theta J(\theta)$, can be expressed as a convenient expectation due to the Policy Gradient Theorem, and the REINFORCE algorithm is based on a simple Monte-Carlo approximation to this expectation. Refer to chapter 13 of Sutton & Barto's text for more details [3].

In [3], the authors present a version of REINFORCE that performs a single gradient update to the policy for every timestep of a collected episode. Here, we ask you to consider a slightly different version that instead performs a single policy gradient update for every *episode* of collected data, treating the entire episode as a minibatch of experience when determining the policy gradient. By performing a single gradient update per episode, we can obtain policy gradient estimates with (slightly) less variance, and we reduce the risk of significantly changing the policy if we encounter a particularly long episode of experience, which generally tends to result in learning instabilities. We also ask you to use the Adam optimizer instead of performing vanilla gradient ascent. If you're not familiar with Adam, you can think of it as a fancy version of gradient descent that introduces momentum and an adaptive learning rate for each dimension. Pseudo-code for the required version of REINFORCE is provided in Algorithm 1. Recommended hyperparameter values are included in the code template. The network settings and hyperparameters for the policy are provided to you in `a2c/net.py`. You can use the `network.summary()` and `network.get_config()` calls to inspect the network architecture.

Evaluate your implementation of Algorithm 1 on the `CartPole-v0` environment by running

Algorithm 1 REINFORCE

procedure REINFORCE
    Start with policy network π_θ
    repeat for E **training episodes:**
        Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ *following* $\pi_\theta(\cdot)$
        for $t = 0, 1, \ldots, T-1$:
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} R_{k+1}$
        $L(\theta) = -\frac{1}{T} \sum_{t=0}^{T-1} G_t \ln \pi_\theta(A_t|S_t)$
        Update π_θ *using* `Adam` $(\nabla_\theta L(\theta))$
end procedure

5 IID trials[2] with $E = 3,500$. During each trial, freeze the current policy every 100 training episodes and run 20 independent test episodes. **When running your test episodes, you should continue to sample actions with respect to the learned policy**. Record the mean **un**discounted return obtained over these 20 test episodes for each trial and store them in a matrix $D \in \mathbb{R}^{\text{number of trials} \times \text{number of frozen policies per trial}} = \mathbb{R}^{5 \times (3,500/100)} = \mathbb{R}^{5 \times 35}$. Note that each entry in D is an average of 20 values, and any particular column of D gives us a "snapshot" of your implementation at some point in time during training on `CartPole-v0`.

In one figure, plot the **mean of the mean undiscounted return** on the y-axis against **number of training episodes** on the x-axis. In other words, plot the average of the entries in columns of D on the y-axis. On the same figure, also plot a shaded region showing the maximum and minimum mean undiscounted return against number of training episodes, where the max and min are performed across the trials (i.e. show the the max and min of each column in D). The plotting format is given to you in `a2c/run.py`.

As a brief aside, in RL, the term "expected return" refers to $\mathbb{E}_{\tau \sim \pi_\theta} \{\sum_{t=0}^{\infty} \gamma^t R_t\}$, where the expectation is with respect to the randomess in the MDP under a particular policy (the initial state, any randomness in the state transitions, any randomness in the action selection, and any randomness in the reward generation). When we ask you to plot the mean of the mean cumulative undiscounted reward, the first mean refers to the randomness in the DRL algorithm (e.g. random weight initialisation), and the second mean refers to the randomness in the underlying MDP itself. When we look at a particular column of D, the variability amongst these 5 values is attributable to the DRL algorithm itself, not the underlying MDP. For the purposes of comparing DRL algorithms, this variability is of critical importance.

Runtime Estimation: To help you plan your time, note that our implementation of Algorithm 1 takes 9 minutes to complete a single trial with $E = 3,500$ on a 2020 MacBook Pro.

[2] We ask you to run multiple trials as DRL algorithms tend to exhibit a significant deal of random variation both across implementations and across random seeds for a given implementation. This is well documented in the literature [1] and we encourage you all to reflect on the various sources of randomness that make DRL algorithms hard to compare.

Problem 1.2: REINFORCE with Baseline (10 pts)

An important early development to REINFORCE was the introduction of a baseline. In this part of the homework, we ask you to make a small change to your implementation of REINFORCE to examine the impact of this on the same `CartPole-v0` test environment.

Again, we ask you to consider a slightly modified version of the algorithm presented in Sutton & Barto's text. The pseducode is provided in Algorithm 2. For the baseline, we recommend making changes to the output layer of the network provided in `a2c/net.py`. Generate the same plot as for REINFORCE (i.e. run 5 IID trials and plot the mean, max and min of the mean undiscounted returns across trials). Take $E = 3,500$.

Runtime Estimation: Note that our implementation of Algorithm 2 takes 11 minutes to complete a single trial with $E = 3,500$ running on a 2020 MacBook Pro.

Algorithm 2 REINFORCE with Baseline

procedure REINFORCE WITH BASELINE
    Start with policy network π_θ *and baseline network* b_ω
    repeat for E **training episodes:**
        Generate an episode $S_0, A_0, R_0, \ldots, S_{T-1}, A_{T-1}, R_{T-1}$ *following* $\pi_\theta(\cdot)$
        for $t = 0, 1, \ldots, T-1$:
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} R_{k+1}$
        $L(\theta) = -\frac{1}{T} \sum_{t=0}^{T-1} (G_t - b_\omega(S_t)) \ln \pi_\theta(A_t|S_t)$
        $L(\omega) = \frac{1}{T} \sum_{t=0}^{T-1} (G_t - b_\omega(S_t))^2$
        Update π_θ *using* `Adam` $(\nabla_\theta L(\theta))$
        Update b_ω *using* `Adam` $(\nabla_\omega L(\omega))$
end procedure

Problem 1.3: N-step Advantage Actor Critic (20 pts)

Another important REINFORCE development was in choosing the baseline to be an approximate state value function, and to use this baseline to bootstrap estimates of the return. We call such a baseline a *critic*, and it carves out a family of algorithms depending on the degree of bootstrapping performed. See chapter 13 of the Sutton & Barto text for more details on this. In this part of the homework, we ask you to implement this algorithm for a variety of different bootstrapping strategies. In particular, we ask you to implement the version of N-step A2C given in Algorithm 3.

Once again, please use the provided network architectures and hyperparameters.

Please provide 3 separate figures similar to the previous problems for $N = 1, 10, 100$. For each value of N please run 5 IID trials as in the previous parts of the homework problem. Take $E = 3,500$ for each value of N.

Runtime Estimation: Note that our implementation of Algorithm 3 takes 11 minutes to complete a single trial with $E = 3,500, N = 1, 10, 100$ running on a 2020 MacBook Pro.

Algorithm 3 N-step Advantage Actor-Critic

procedure N-STEP ADVANTAGE ACTOR-CRITIC
  Start with actor network π_θ *and critic network* V_ω
  repeat:
    Generate an episode $S_0, A_0, R_0, \ldots, S_{T-1}, A_{T-1}, R_{T-1}$ *following* $\pi_\theta(\cdot)$
    for $t = 0, 1, \ldots, T-1$:
      $$V_{\text{end}} = \begin{cases} V_\omega(S_{t+N}) & \text{if } t+N < T \\ 0 & \text{otherwise} \end{cases}$$
      $$G_t = \left(\sum_{k=t}^{\min(t+N-1,T-1)} \gamma^{k-t} R_k\right) + \gamma^N V_{\text{end}}$$
    $L(\theta) = -\frac{1}{T}\sum_{t=0}^{T-1}(G_t - V_\omega(S_t))\ln\pi_\theta(A_t|S_t)$
    $L(\omega) = \frac{1}{T}\sum_{t=0}^{T-1}(G_t - V_\omega(S_t))^2$
    Update π_θ *using* `Adam` $(\nabla_\theta L(\theta))$
    Update V_ω *using* `Adam` $(\nabla_\omega L(\omega))$
end procedure

1.4 N-step A2C & REINFORCE with Baseline (4 pts)

How does N-step A2C relate to REINFORCE and REINFORCE with Baseline? (i.e. under what conditions do these algorithms become equivalent)

1.5 REINFORCE with & without Baseline (4 pts)

Does adding a baseline improve the performance? Please briefly explain why you think this happens.

Problem 2: DQN (32 pts)

In this problem you will implement a version of Q-learning with function approximation, DQN, following the work of Mnih et al. [2]. Instead of leveraging policy gradients, DQN takes inspiration from generalised policy iteration algorithms developped in the tabular RL literature. Some of the key contributions of [2] include the introduction of several tricks to address instability issues when function approximation is incorporated into Q-learning.

Problem 2.1: Temporal Difference & Monte Carlo (6 pts)

Answer the true/false questions below, providing one or two sentences for **explanation**.

1. (3 pts) TD methods can't learn in an online manner since they require full trajectories.
2. (3 pts) MC rollouts as described in class can be applied with non-terminating episodes.

Problem 2.2: DQN Implementation (15 pts)

You will implement DQN and compare it against your policy gradient implementations on `CartPole-v0`. Please write your code in the `dqn/dqn.py` file. This code template includes recommended hyperparameter values for your implementation of DQN. Additional hyperparameter suggestions are included in the 'Guidelines on Implementation' section at the end of this handout.

Implement a deep Q-network with experience replay. While the DQN paper [2] uses a convolutional architecture, a neural network with 3 fully-connected layers should suffice for the low-dimensional environments that we are working with. For the deep Q-network, use the provided `FullyConnectedModel` class in `dqn.py`. You will have to implement the following:

- Create an instance of the Q Network class.
- Create a function that constructs a greedy policy and an exploration policy (ϵ-greedy) from the Q values predicted by the Q Network.
- Create a function to train the Q Network, by interacting with the environment.
- Create a function to test the Q Network's performance on the environment and generate a D matrix similar to the algorithms in Question 1. Please reference section 1.1 for further description.

Recall that the key tricks of DQN paper are experience replay from a constant-capacity buffer, and the use of a target Q-network in addition to the usual Q-network. In your implementation, use a replay buffer with capacity of 50,000. Use a hard update scheme for the target Q-network with frequency of 50 timesteps. That is, every 50 timesteps, copy the parameters of the Q-network into the target Q-network and then keep the target Q-network fixed until another 50 timesteps have passed.

Although the original Nature DQN paper uses RMSprop as an optimizer, we recommend using Adam with a learning rate of 5×10^{-4} (you can keep the default values for the other Adam hyperparameters).

For exploration, define an ϵ-greedy policy on the current Q-network (not the target Q-network) with $\epsilon = 0.05$. Starting from the `Replay_Memory` class, implement the following functions:

- Append a new transition from the memory.
- Uniformly sample (with replacement) a batch of transitions from the memory to train your network. Use a batch size of 32.
- Initialize the replay buffer with 10,000 timesteps of experience in the environemnt following a uniform random policy before starting DQN training.
- Modify your training function of your network to learn from experience sampled *from the memory*, rather than learning online from the agent.

Generate a similar plot as for the policy gradient algorithms illustrating the min, max and mean performance of your implementation across 5 IID trials. Take $E = 200$ for each trial, and instead of freezing the policy every 100 training episodes to perform 20 test episodes, freeze the policy every 10 training episodes. In this case, at test time, **you should select actions using the greedy policy**.

Runtime Estimation: Note that our solution DQN implementation takes 3 minutes to complete a single trial with $E = 200$ running on a 2020 MacBook Pro.

2.3 DQN vs Policy Gradient Algorithms (5 pts)

Briefly explain why DQN outperforms the policy gradient algorithms that you implemented in Problem 1 on `CartPole-v0`.

2.4 Pros and Cons of Policy gradient methods (6 pts)

Briefly describe a setting where policy gradient methods like N-step A2C would be preferable to DQN and vice versa.

Feedback

Feedback: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing?

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade.

Guidelines on Implementation

This homework requires a significant implementation effort. It is hard to read through the papers once and know immediately what you will need to implement. We suggest you to think about the different components (e.g., network definition, network updater, policy runner, ...) that you will need to implement for each of the different methods that we ask you about, and then read through the papers having these components in mind. By this we mean that you should try to divide and implement small components with well-defined functionalities rather than try to implement everything at once. Much of the code and experimental setup is shared between the different methods so identifying well-defined reusable components will save you trouble.

A couple of points which may make life easier:

- **Episode generation**: In keras, model.predict() is considerably slower than `__call__` for single batch execution. Make sure you use the latter for episode generation. (https://www.tensorflow.org/api_docs/python/tf/keras/Model?hl=en#predict)
- (optional) Training progress logging: An easy way to keep track of training progress is to use TensorBoard. TensorBoard can be used with Tensorflow, keras as well as PyTorch. This tutorial https://www.tensorflow.org/tensorboard/scalars_and_keras is on using TensorBoard with keras.
- Using a debugger like `pudb` or built in debuggers in IDEs are **extremely** useful as we start to consider more sophisticated implementations.
- Consider dumping your data (the matrix D) after every trial instead of generating the required plots in the same script.

Some hyperparameter and implementation tips and tricks:

- For efficiency, you should try to vectorize your code as much as possible and use **as few loops as you can** in your code. For example, in lines 5 and 6 of Algorithm 1 (REINFORCE) you should not use two nested loops. How can you formulate a single loop to calculate the cumulative discounted rewards? Hint: Think backwards!
- Feel free to experiment with different policy architectures. Increasing the number of hidden units in earlier layers may improve performance. If you change anything from the provided networks, please describe your changes in your writeup.
- We recommend using a discount factor of $\gamma = 0.99$.

Algorithm 4 DQN

procedure DQN
  Initialize network Q_ω *and* Q_{target} *as a clone of* Q_ω
  Initialize replay buffer R *and burn in with trajectories followed by random policy*
  Initialize $c = 0$
  repeat for E **training episodes:**
    Initialize S_0
    for $t = 0, 1, \ldots, T-1$:
      $$a_t = \begin{cases} \arg\max_a Q_\omega(s_t, a) & \text{with probability } 1-\epsilon \\ \text{Random action} & \text{otherwise} \end{cases}$$
      Take a_t *and observe* r_t, s_{t+1}
      Store (s_t, a_t, r_t, s_{t+1}) *in* R
      Sample minibatch of (s_i, a_i, r_i, s_{i+1})*with size* N *from* R
      $$y_i = \begin{cases} r_i & s_{i+1} \text{ is terminal} \\ r_i + \gamma \max_a Q_{\text{target}}(s_{i+1}, a) & \text{otherwise} \end{cases}$$
      $L(\omega) = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - Q_\omega(s_i, a_i))^2$
      Update Q_ω *using* $\texttt{Adam}\,(\nabla_\omega L(\omega))$
      $c = c + 1$
      Replace Q_{target} *with current* Q_ω *if* $c \,\%\, 50 = 0$
end procedure

References

[1] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.