Homework 2: Deep-Q-Network and Actor-Critic

1 Introduction

In this homework, first we will do a brief recap of deep-Q-network (DQN) and Actor-Critic (AC) algorithms in RL. Then, you will be asked to complete the implementations for for these algorithms and then conduct a series of experiments using your implementation.

2 Review

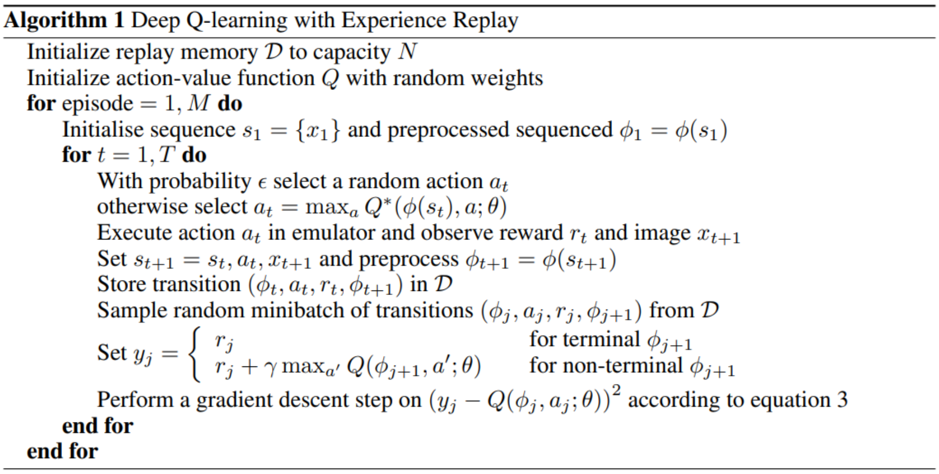
2.1 Deep-Q-Network

Recall that the Q-value $Q(s_t, a_t)$ is defined as the expected reward from state s_t after taking the action a_t. And in DQN, our goal is to learn to approximate the optimum Q-function as best as possible. In doing so, we have the following iterative equation in hand

$$Q_{i+1}(s,a) = \mathbb{E}(r_{s,a} + \gamma \max_{a'} Q_i(s', a')) \tag{1}$$

For state s, an agent considers a predefined *behavior distribution* and performs an action a, leading to a new state s'. The tuple (s, a, r, s') is stored to create an *experience replay* buffer from which we randomly sample training data.

Also, note that DQN is an *off-policy* RL algorithm since it tries to learn the greedy strategy $\max_{a'} Q_i(s', a')$ while following the *behavior distribution*. In practise, we use the ϵ-*greedy* policy as our agent's *behavior distribution*, where the agent selects greedy strategy with probability $1 - \epsilon$ and a random action with probability ϵ. The Q-function can be parameterized as $Q_i(s, a; \theta_i)$ by a neural network θ to obtain the following algorithm [1] for DQN. We can call the network as *Q-net*.



```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Figure 1: DQN algorithm

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(r + \gamma \max_{a'} \underbrace{Q(s', a'; \theta_i^-)}_{\text{target-net}} - \underbrace{Q(s, a; \theta_i)}_{\text{Q-net}}\right)^2 \tag{2}$$

Equation 2 illiterates the loss function for a DQN.

$$Q(s', a'; \theta^{-}_{i+1}) \leftarrow (1 - \tau) * Q(s', a'; \theta^{-}_{i}) + \tau Q(s', a'; \theta_i) \tag{3}$$

In practice however, we keep a second network called *target-net* which is essentially a lagged-version of this *Q-net*. We use the Q-values from the *target-net* while training our *Q-net* with stochastic gradient descent and updating *target-net* with *Q-net* after every k steps or episodes. Alternatively, we can also update target network using equation 3 where τ is a very small positive real number (i.e. 5e-3). This *target-net* makes sure that our *Q-net* is not chasing a non-stationary target value over the course of training.

2.2 Actor-Critic

Recall from HW1 that estimation of loss-gradient for a vanilla policy gradient (VPG) suffers from high variance. One way to mitigate this issue is to use *reward-to-go* as illustrated in equation 4 instead of trajectory return.

$$\nabla_\theta J(\theta) \approx -\frac{1}{N} \sum_k \left[\sum_{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta(a^k_t | s^k_t) \sum_{t'=t}^{T} r^k_{t'} \right) \right] \tag{4}$$

We can further improve this approach by subtracting a value function V^π_ϕ from the reward-to-go as a state-dependant baseline by estimating the Q-value. This is shown in eq 5

$$\nabla_\theta J(\theta) \approx -\frac{1}{N} \sum_k \left[\sum_{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta(a^k_t | s^k_t) \underbrace{\left(\left[\sum_{t'=t}^{T} r^k_{t'} \right] - V^\pi_\phi(s^k_t) \right)}_{\text{estimation of advantage}} \right) \right] \tag{5}$$

However, this estimated Q-value (also known as *advantage*) also suffers from high variance and we can represent it in term of the V-values from the next state (recall the bellman equations) to yield the following eq 6.

$$\nabla_\theta J(\theta) \approx -\frac{1}{N} \sum_k \left[\sum_{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta(a^k_t | s^k_t) \underbrace{\left(\left[r^k_t + \gamma V^\pi_\phi(s^k_{t+1}) \right] - V^\pi_\phi(s^k_t) \right)}_{\text{better estimation of advantage}} \right) \right] \tag{6}$$

Here, the network V^π_ϕ is called the critic-network and we train it to minimize the MSE loss function given by eq 7

$$\mathcal{L}(\phi) \approx \sum_{k,t} \left(\underbrace{\left[r^k_t + \gamma V^\pi_\phi(s^k_{t+1}) \right]}_{\text{target}} - V^\pi_\phi(s^k_t) \right)^2 \tag{7}$$

Theoretically, we need to perform this minimization every time we update our policy. But, that would be very costly in terms of computation, so in practice, we end up performing this minimization a few times after each iteration. Also, note that as we update out critic-network, our target value will also change. So, we also need to recompute the new targets with the updated value function following these steps

1. Update target with the current value function.
2. Take a few gradient steps to update the value function using eq 7.
3. Redo step 1 and 2 several times.

3 Starter Code

This section talks about the provided starter code (available as zip file). It includes partial implementation of this assignment. There are two versions. One that you can run in your local machine from the CLI. Alternatively, you can also find a Colab notebook that you can run on Google Colab. The colab version takes the simulation parameters as a dictionary.

You should find out the tags marked as ***TODO*** and complete them accordingly. In most cases, some hints are provided for you.

Here is a list of files that you can find within the starter code folder

- main_hw2_dqn.py and main_hw2_ac.py

  This is the main file that you will use to run experiments from section 4.2 and 4.3. Running this script should generate a .pkl file as the log of the experiment. It will also create a video containing a single episode with the policy you have just trained!

- main_generate_plt.py

  This is the main file that you can use to generate visualizations for section 4. You can use other graphing methods, but you must adhere to this visualization guideline.

  After you have multiple runs (applicable for section 4.2 bonus) of one PG version, you can merge multiple .pkl files into one pandas dataframe to create a mean and standard-deviation plot. An example is given here.

- learning_algorithms.py

  This file contains most of the codes behind the PG implementation. The TODO's here are

  - Use solutions from hw1's
    - Calculate avg reward for this rollout.
    - Compute actor loss function. [**hw1's policy net**]
    - Define the actor net. [**hw1's policy net**]
    - Forward pass of actor net. [**hw1's policy net**]
  - DQN
    - Define Q-net.
    - Implement the epsilon-greedy behavior.
    - Update Q-net.
    - Implement replay buffer.
  - Actor-Critic
    - Define the critic net.
    - Forward pass of critic net.
    - Update target values.
    - Estimate advantage.
    - Compute critic loss function.

- utils.py

  - Compute discounted_rtg_reward (as a list) from raw_reward. [**from hw1**]

- requirements.txt

  A text file that has a list of required packages to run this assignment.

- hw2_dqn.ipynb and hw2_ac.ipynb

  Notebooks which you can upload into your Google drive to run the experiments there. This can save you from creating an environment locally. :)
  Don't forget to upload your scripts with TODO's as well. One possible solution is discussed here.

4 Tasks for You

4.1 Part 1 - Completion of TODOs

Complete the TODOs in the started code.

4.2 Part 2 - Experiment I (CartPole with DQN)

In this section, you will run couple of experiments with your completed implementation of DQN and the environment CartPole-v1. The goal of this experiment is to train the agent on CartPole environment using the given hyper-parameters and answer couple of questions based on the observed results.

Target Network Update Rate

Run the following trials and create a plot where reward graph for all three of trials are shown and labeled. Report the observed optimum value as τ_{opt}.

Trials

- **T0**: python main_hw2_dqn.py -e CartPole-v1 -tau 0.5 -xn CartPole_v1_t0
- **T1**: python main_hw2_dqn.py -e CartPole-v1 -tau 0.05 -xn CartPole_v1_t1
- **T2**: python main_hw2_dqn.py -e CartPole-v1 -tau 0.005 -xn CartPole_v1_t2

Exploration vs Exploitation

Run the following trials and create a plot where reward graph for all three of trials are shown and labeled. Use the optimum value for τ_{opt} that have observed from the above experiment. Report the observed optimum value as ϵ_{opt}^{init}, ϵ_{opt}^{min}.

Trials

- **T3**: python main_hw2_dqn.py -e CartPole-v1 -tau τ_{opt} -init_eps 0.0 min_eps 0.00 -xn CartPole_v1_t3
- **T4**: python main_hw2_dqn.py -e CartPole-v1 -tau τ_{opt} -init_eps 0.1 min_eps 0.05 -xn CartPole_v1_t4

Deliverables for report

1. Create a graph that compares the learning curve from the four trials above. Label the curves as t0, t1, t2. Report optimum value as τ_{opt}
2. Create a graph that compares the learning curve from the four trials above. Label the curves as t0, t1, t2. Report observed optimum values as ϵ_{opt}^{init}, ϵ_{opt}^{min}.
3. Answer the following questions
   (a) How does changing target network update rate affect the learning curve? Can you justify your observation?
   (b) How does changing range for ϵ affect the learning curve? Can you justify your observation?

What to expect

- While doing experiment with varying τ, the best configuration should converge around **110**.
- Reward from the best configuration for exploration vs exploitation should converge around **500**

4.3 Part 3 - Experiment II (LunarLander with AC)

In this section, you will run couple of experiments with your completed implementation for Actor-Critic and use that to solve LunarLander-v2 environment.

Trials

You will try out different combinations for updating the critic-network and find the best values.

- **T0**: python main_hw1.py -e LunarLander-v2 -nci 1 -nce 1 -xn LunarLander_v2_t0
- **T1**: python main_hw1.py -e LunarLander-v2 -nci 10 -nce 10 -xn LunarLander_v2_t1
- **T3**: python main_hw1.py -e LunarLander-v2 -nci 20 -nce 20 -xn LunarLander_v2_t2

Deliverables for report

1. Create a graph that compares the learning curve from the three trials above. Label the curves as t0, t1, t2.
2. Answer the following questions
   (a) How does changing the critic network update parameters (number of iterations and number of epochs) affect learning performance? How can you justify this relationship?

What to expect

- The average-trajectory-reward for the best configuration (among the three listed above) should converge around **180**. Expected results from the Colab version was tested without any hardware acceleration.

5 Submission Guideline

Please read the following very carefully. Unable to meet any of the requirements will cause you to lose points-

1. You are **highly encouraged** to create a GitHub repo and perform frequent commit. Attach the repo link within your report. This will serve as a history of your activity for this assignment.
2. Ideally you should not require to create any other additional functions or classes except the ones that are already provided. You are only expected to fill-in the TODOs.
3. Submit as a zipped file named as cse6369_a1__StudentID__StudentName.zip. A student named "John Doe" with ID 0123456789 should have a zipped file name cse6369_a1__0123456789__john_doe.zip.

   Make sure this zip file contains the following folder/files.

   - report.pdf
   - main_hw1.py
   - main_generate_plot.py

- learning_algorithms.py
- utils.py
- requirements.txt
- (optional) Two .mp4 files from best configuration of each environment.

4. All the submitted code and the report should be your own work. If you have used any reference, make sure to use the source as a reference. If any form of plagiarism is found, you won't receive any credit for this assignment and it will reflect negatively on your final grade as well.

5. Your implementations must be written in **Python 3.8** or later versions.

6. You are not allowed to import any additional libraries or packages except the ones that are already imported. Please consult with the TA if you think you need any other libraries apart from those.

7. Make sure to fill in student name and ID in the Colab file.

8. Do not submit figures as stand-alone files, please embed them into your report.

9. Make sure to include your name and student ID in the report.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).