CS 234: Assignment #2

Due date: April 26, 2024 at 6:00 PM (18:00) PST

These questions require thought but do not require long answers. Please be as concise as possible.

We encourage students to discuss in groups for assignments. **However, each student must finish the problem set and programming assignment individually, and must turn in her/his assignment.** We ask that you abide by the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work is done by yourself. If you have discussed the problems with others, please include a statement saying who you discussed problems with. Failure to follow these instructions will be reported to the Office of Community Standards. We reserve the right to run a fraud-detection software on your code.

Please review any additional instructions posted on the assignment page at http://web.stanford.edu/class/cs234/assignments.html. When you are ready to submit, please follow the instructions on the course website.

1 Deep Q-Networks (DQN) (8 pts writeup)

All questions in the section pertain to DQN. The pseudocode for DQN is provided below.

Algorithm 1 Deep Q-Network (DQN)

```
1:  Initialize replay buffer $\mathcal{D}$
2:  Initialize action-value function $Q$ with random weights $\theta$
3:  Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
4:  for episode = 1, $M$ do
5:      Receive initial state $s_1$
6:      for $t = 1, T$ do
7:          With probability $\epsilon$ select a random action $a_t$
8:          otherwise select $a_t = \max_a Q(s_t, a; \theta)$
9:          Execute action $a_t$ and observe reward $r_t$ and state $s_{t+1}$
10:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
11:         Sample random minibatch $B$ of transitions from $\mathcal{D}$
12:         for each transition $(s_j, a_j, r_j, s_{j+1})$ in $B$ do
13:             if $s_{j+1}$ is terminal then
14:                 Set $y_j = r_j$
15:             else
16:                 Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
17:             end if
18:             Perform gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to network parameters $\theta$
19:         end for
20:         Every $C$ steps reset $\hat{Q} = Q$ by setting $\theta^- = \theta$
21:     end for
22: end for
```

In this pseudocode:

- D is the replay memory which stores transitions.
- θ are the weights of the Q-network, which are adjusted during training.
- θ^- are the weights of the target network, which are periodically updated to match θ.
- M is the number of episodes over which the training occurs.
- T is the maximum number of steps in each episode.
- ϵ is the exploration rate, which is typically decayed over time.
- γ is the discount factor, used to weigh future rewards.
- C is the frequency with which to update the target network's weights.

1.1 Written Questions (8 pts)

(a) (3 pts) (**written**) What are three key differences between the DQN and Q-learning algorithms?

(b) (2 pts) (**written**) When using DQN with a deep neural network, which of the above components would you hypothesize contributes most to performance gains? Justify your answer.

(c) (3 pts) (**written**) In DQN, the choice of target network update frequency is important. What might happen if the target network is updated every 10^{15} steps for an agent learning to play a simple Atari game like Pong?

2 Policy Gradient Methods (54 pts coding + 26 pts writeup)

The goal of this problem is to experiment with policy gradient and its variants, including variance reduction and off-policy methods. Your goals will be to set up policy gradient for both continuous and discrete environments, use a neural network baseline for variance reduction, and implement the off-policy Proximal Policy Optimization algorithm. The starter code has detailed instructions for each coding task and includes a README with instructions to set up your environment. Below, we provide an overview of the key steps of the algorithm.

2.1 REINFORCE

Recall the policy gradient theorem,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

REINFORCE is a Monte Carlo policy gradient algorithm, so we will be using the sampled returns G_t as unbiased estimates of $Q^{\pi_\theta}(s,a)$. The REINFORCE estimator can be expressed as the gradient of the following objective function:

$$J(\theta) = \frac{1}{\sum T_i} \sum_{i=1}^{|D|} \sum_{t=1}^{T_i} \log\big(\pi_\theta(a_t^i|s_t^i)\big) G_t^i$$

where D is the set of all trajectories collected by policy π_θ, and $\tau^i = (s_0^i, a_0^i, r_0^i, s_1^i, \ldots, s_{T_i}^i, a_{T_i}^i, r_{T_i}^i)$ is trajectory i.

2.2 Baseline

One difficulty of training with the REINFORCE algorithm is that the Monte Carlo sampled return(s) G_t can have high variance. To reduce variance, we subtract a baseline $b_\phi(s)$ from the estimated returns when computing the policy gradient. A good baseline is the state value function, $V^{\pi_\theta}(s)$, which requires a training update to ϕ to minimize the following mean-squared error loss:

$$L_{\text{MSE}}(\phi) = \frac{1}{\sum T_i} \sum_{i=1}^{|D|} \sum_{t=1}^{T_i} (b_\phi(s_t^i) - G_t^i)^2$$

2.3 Advantage Normalization

After subtracting the baseline, we get the following new objective function:

$$J(\theta) = \frac{1}{\sum T_i} \sum_{i=1}^{|D|} \sum_{t=1}^{T_i} \log\big(\pi_\theta(a_t^i|s_t^i)\big) \hat{A}_t^i$$

where

$$\hat{A}_t^i = G_t^i - b_\phi(s_t^i)$$

A second variance reduction technique is to normalize the computed advantages, $\hat{A}_t^i$, so that they have mean 0 and standard deviation 1. From a theoretical perspective, we can consider centering the advantages to be simply adjusting the advantages by a constant baseline, which does not change the policy gradient. Likewise, rescaling the advantages effectively changes the learning rate by a factor of $1/\sigma$, where σ is the standard deviation of the empirical advantages.

2.4 Proximal Policy Optimization

One might notice that the REINFORCE algorithm above (with or without a baseline function) is an on-policy algorithm; that is, we collect some number of trajectories under the current policy network parameters, use that data to perform a single batched policy gradient update, and then proceed to discard that data and repeat the same steps using the newly updated policy parameters. This is in stark contrast to an algorithm like DQN which stores all experiences collected over several past episodes. One might imagine that it could be useful to have a policy gradient algorithm "squeeze" a little more information out of each batch of trajectories sampled from the environment. Unfortunately, while the Q-learning update immediately allows for this, our derived REINFORCE estimator does not in its standard form.

Ideally, an off-policy policy gradient algorithm will allow us to do multiple parameter updates on the same batch of trajectory data. To get a suitable objective function that allows for this, we need to correct for the mismatch between the policy under which the data was collected and the policy being optimized with that data. Proximal Policy Optimization (PPO) restricts the magnitude of each update to the policy (i.e., through gradient descent) by ensuring the ratio of the current and former policies on the current batch is not too different. In doing so, PPO tries to prevent updates that are "too large" due to the off-policy data, which may lead to performance degradation. This technique is related to the idea of importance sampling which we will examine in detail later in the course. Consider the following ratio z_θ, which measures the probability ratio between a current policy π_θ (the "actor") and an old policy π_θ^{old}:

$$z_\theta(s_t^i, a_t^i) = \frac{\pi_\theta(a_t^i \mid s_t^i)}{\pi_{\theta_{\text{old}}}(a_t^i \mid s_t^i)}$$

To do so, we introduce the clipped PPO loss function, shown below, where $\text{clip}(x, a, b)$ outputs x if $a \le x \le b$, a if $x < a$, and b if $x > b$:

$$J_{\text{clip}}(\theta) = \frac{1}{\sum T_i} \sum_{i=1}^{|D|} \sum_{t=1}^{T_i} \min(z_\theta(s_t^i, a_t^i)\hat{A}_t^i, \text{clip}(z_\theta(s_t^i, a_t^i), 1-\epsilon, 1+\epsilon)\hat{A}_t^i)$$

where $\hat{A}_t^i = G_t^i - V_\phi(s_t^i)$. Note that in this context, we will refer to $V_\phi(s_t^i)$ as a "critic"; we will train this like the baseline network described above.

To train the policy, we collect data in the environment using π_θ^{old} and apply gradient ascent on $J_{\text{clip}}(\theta)$ for each update. After every K updates to parameters $[\pi, \phi]$, we update the old policy π_θ^{old} to equal π_θ.

2.5 Coding Questions (50 pts)

The functions that you need to implement in `network_utils.py`, `policy.py`, `policy_gradient.py`, and `baseline_network.py` are enumerated here. Detailed instructions for each function can be found in the comments in each of these files.

Note: The "batch size" for all the arguments is $\sum T_i$ since we already flattened out all the episode observations, actions, and rewards for you.

In `network_utils.py`, you need to implement:

- `build_mlp`

In `policy.py`, you need to implement:

- `BasePolicy.act`
- `CategoricalPolicy.action_distribution`
- `GaussianPolicy.__init__`
- `GaussianPolicy.std`

- `GaussianPolicy.action_distribution`

In `policy_gradient.py`, you need to implement:

- `PolicyGradient.init_policy`
- `PolicyGradient.get_returns`
- `PolicyGradient.normalize_advantage`
- `PolicyGradient.update_policy`

In `baseline_network.py`, you need to implement:

- `BaselineNetwork.__init__`
- `BaselineNetwork.forward`
- `BaselineNetwork.calculate_advantage`
- `BaselineNetwork.update_baseline`

In `ppo.py`, you need to implement:

- `PPO.update_policy`

2.6 Debugging

To help debug and verify that your implementation is correct, we provide a set of sanity checks below that pass with a correct implementation. Note that these are not exhaustive (i.e., they do not verify that your implementation is correct) and that you may notice oscillation of the average reward across training.
Across most seeds:

- Policy gradient (without baseline) on Pendulum should achieve around an average reward of 100 by iteration 10.
- Policy gradient (with baseline) on Pendulum should achieve around an average reward of 700 by iteration 20.
- PPO on Pendulum should achieve an average reward of 200 by iteration 20.
- All methods should reach an average reward of 200 on Cartpole, 1000 on Pendulum, and 200 on Cheetah at some point.

2.7 Writeup Questions (26 pts)

(a) (3 pts) To compute the REINFORCE estimator, you will need to calculate the values $\{G_t\}_{t=1}^{T}$ (we drop the trajectory index i for simplicity), where

$$G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

Naively, computing all these values takes $O(T^2)$ time. Describe how to compute them in $O(T)$ time.

(b) (3 pts) Consider the cases in the gradient of the clipped PPO loss function equals 0. Express these cases mathematically and explain why PPO behaves in this manner.

(c) (3 pts) Notice that the method which samples actions from the policy also returns the log-probability with which the sampled action was taken. Why does REINFORCE not need to cache this information while PPO does? Suppose this log-probability information had not been collected during the rollout. How would that affect the implementation (that is, change the code you would write) of the PPO update?

(d) (12 pts) The general form for running your policy gradient implementation is as follows:

```
python main.py --env-name ENV --seed SEED --METHOD
```

`ENV` should be `cartpole`, `pendulum`, or `cheetah`, `METHOD` should be either `baseline`, `no-baseline`, or `ppo`, and `SEED` should be a positive integer.

For the `cartpole` and `pendulum` environments, we will consider 3 seeds (`seed = 1, 2, 3`). For `cheetah`, we will only require one seed (`seed = 1`) since it's more computationally expensive, but we strongly encourage you to run multiple seeds if you are able to. Run each of the algorithms we implemented (PPO, PG with baseline, PG without baseline) across each seed and environment. In total, you should end up with at least 21 runs.

Plot the results using:

```
python plot.py --env-name ENV --seeds SEEDS
```

where `SEEDS` should be a comma-separated list of seeds which you want to plot (e.g. `--seeds 1,2,3`). **Please include the plots (one for each environment) in your writeup, and comment on the performance of each method.**

We have the following expectations about performance to receive full credit:

- cartpole: Should reach the max reward of 200 (although it may not stay there)
- pendulum: Should reach the max reward of 1000 (although it may not stay there)
- cheetah: Should reach at least 200 (could be as large as 900)

3 Distributions induced by a policy (13 pts)

Suppose we have a single MDP and two policies for that MDP, π and π'. Naturally, we are often interested in the performance of policies obtained in the MDP, quantified by V^π and $V^{\pi'}$, respectively. If the reward function and transition dynamics of the underlying MDP are known to us, we can use standard methods for policy evaluation. There are many scenarios, however, where the underlying MDP model is not known and we must try to infer something about the performance of policy π' solely based on data obtained through executing policy π within the environment. In this problem, we will explore a classic result for quantifying the gap in performance between two policies that only requires access to data sampled from one of the policies.

Consider an infinite-horizon MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$ and stochastic policies of the form $\pi : \mathcal{S} \to \Delta(\mathcal{A})$[1]. Specifically, $\pi(a|s)$ refers to the probability of taking action a in state s, and $\sum_a \pi(a|s) = 1,\ \forall s$. For simplicity, we'll assume that this decision process has a single, fixed starting state $s_0 \in \mathcal{S}$.

(a) (3 pts) (**written**) Consider a fixed stochastic policy and imagine running several rollouts of this policy within the environment. Naturally, depending on the stochasticity of the MDP $\mathcal{M}$ and the policy itself, some trajectories are more likely than others. Write down an expression for $\rho^\pi(\tau)$, the probability of sampling a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$ from running π in $\mathcal{M}$. To put this distribution in context, recall that $V^\pi(s_0) = \mathbb{E}_{\tau\sim\rho^\pi}\left[\sum\limits_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \,|\, s_0\right]$.

(b) (1 pt) (**written**) What is $p^\pi(s_t = s)$, where $p^\pi(s_t = s)$ denotes the probability of being in state s at timestep t while following policy π? (Provide an equation)

(c) (5 pts) (**written**) Just as ρ^π captures the distribution over trajectories induced by π, we can also examine the distribution over states induced by π. In particular, define the *discounted, stationary state distribution* of a policy π as

$$d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty} \gamma^t p^\pi(s_t = s),$$

where $p^\pi(s_t = s)$ denotes the probability of being in state s at timestep t while following policy π; your answer to the previous part should help you reason about how you might compute this value.

The value function of a policy π can be expressed using this distribution $d^\pi(s,a) = d^\pi(s)\pi(a \mid s)$ over states and actions, which will shortly be quite useful.

Consider an arbitrary function $f : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Prove the following identity:

$$\mathbb{E}_{\tau\sim\rho^\pi}\left[\sum_{t=0}^{\infty} \gamma^t f(s_t, a_t)\right] = \frac{1}{(1-\gamma)}\mathbb{E}_{s\sim d^\pi}\left[\mathbb{E}_{a\sim\pi(s)}\left[f(s,a)\right]\right].$$

Hint: You may find it helpful to first consider how things work out for $f(s,a) = 1, \forall (s,a) \in \mathcal{S} \times \mathcal{A}$.

(d) (5 pts) (**written**) For any policy π, we define the following function

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

$A^\pi(s,a)$ is known as the advantage function and shows up in a lot of policy gradient based RL algorithms, which we shall see later in the class. Intuitively, it is the additional benefit one gets from first following action a and then following π, instead of always following π. Prove that the following statement holds for all policies π, π':

$$V^\pi(s_0) - V^{\pi'}(s_0) = \frac{1}{(1-\gamma)}\mathbb{E}_{s\sim d^\pi}\left[\mathbb{E}_{a\sim\pi(s)}\left[A^{\pi'}(s,a)\right]\right].$$

[1]For a finite set $\mathcal{X}$, $\Delta(\mathcal{X})$ refers to the set of categorical distributions with support on $\mathcal{X}$ or, equivalently, the $\Delta^{|\mathcal{X}|-1}$ probability simplex.

Hint 1: Try adding and subtracting a term that will let you bring $A^{\pi'}(s,a)$ into the equation. What happens on adding and subtracting $\sum_{t=0}^{\infty} \gamma^{t+1} V^{\pi'}(s_{t+1})$ on the LHS?
Hint 2: Recall the tower property of expectation which says that $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$.

After proving this result, you might already begin to appreciate why this represents a useful theoretical contribution. We are often interested in being able to control the gap between two value functions and this result provides a new mechanism for doing exactly that, when the value functions in question belong to two particular policies of the MDP.
Additionally, to see how this result is of practical importance as well, suppose the data-generating policy in the above identity π is some current policy we have in hand and π' represents some next policy we would like to optimize for; concretely, this scenario happens quite often when π is a neural network and π' denotes the same network with updated parameters. As is often the case with function approximation, there are sources of instability and, sometimes, even small parameter updates can lead to drastic changes in policy performance, potentially degrading (instead of improving) the performance of the current policy π. These realities of deep learning motivate a desire to occasionally be conservative in our updates and attempt to reach a new policy π' that provides only a modest improvement over π. Practical approaches can leverage the above identity to strike the right balance between making progress and maintaining stability.

4 Ethical concerns with Policy Gradients (5 pts)

In this assignment, we focus on policy gradients, an extremely popular and useful model-free technique for RL. However, policy gradients collect data from the environment with a potentially suboptimal policy during the learning process. While this is acceptable in simulators like Mujoco or Atari, such exploration in real world settings such as healthcare and education presents challenges.
Consider a case study of a Stanford CS course considering introducing a RL-based chat bot for office hours. For each assignment, some students will be given 100% human CA office hours; others 100% chatbot; others a mix of both. The reward signal is the student grades on each assignment. Since the AI chatbot will learn through experience, at any given point in the quarter, the help given by the chatbot might be better or worse than the help given by a randomly selected human CA.
If each time students are randomly assigned to each condition, some students will be assigned more chatbot hours and others fewer. In addition, some students will be assigned more chatbot hours at the beginning of the term (when the chatbot has had fewer interactions and may have lower effectiveness) and fewer at the end, and vice versa. All students will be graded according to the same standards, regardless of which type of help they have received.
Researchers who experiment on human subjects are morally responsible for ensuring their well being and protecting them from being harmed by the study. A foundational document in research ethics, the Belmont Report, identifies three core principles of responsible research:

1. **Respect for persons:** individuals are capable of making choices about their own lives on the basis of their personal goals. Research participants should be informed about the study they are considering undergoing, asked for their consent, and not coerced into giving it. Individuals who are less capable of giving informed consent, such as young children, should be protected in other ways.

2. **Beneficence:** the principle of beneficence describes an obligation to ensure the well-being of subjects. It has been summarized as "do not harm" or "maximize possible benefits and minimize possible harms."

3. **Justice:** the principle of justice requires treating all people equally and distributing benefits and harms to them equitably.

(a) (4 pts) In 4-6 sentences, describe **two** experimental design or research choices that researchers planning the above experiment ought to make in order to respect these principles. Justify the importance of these choices using one of the three ethical principles above and indicating which principle you have chosen. For example, "Researchers ought to ensure that students advised by the chatbot are able to revise their assignments after submission with the benefit of human advice if needed. If they did not take this precaution, the principle of justice would be violated because the risk of harm from poor advice from the AI chatbot would be distributed unevenly."

At universities, research experiments that involve human subjects are subject by federal law to Institutional Review Board (IRB) approval. The purpose of IRB is to protect human subjects of research: to "assure, both in advance and by periodic review, that appropriate steps are taken to protect the rights and welfare of humans participating as subjects in the research" (reference). The IRB process was established in response to abuses of human subjects in the name of medical research performed during WWII (reference). The IRB is primarily intended to address the responsibilities of the researcher towards the subjects. Familiarize yourself with Stanford's IRB Research Compliance process at this link.

(b) (1 pt) If you were conducting the above experiment, what process would you need to follow at Stanford (who would you email/ where would you upload a research protocol) to get clearance?