Stanford RL Pset 3

Andrew Koulogeorge

February 2024

1 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, using a neural network baseline for variance reduction, and implement the off-policy Proximal Policy Optimization algorithm. The programming has detailed instructions for each implementation task, including a README for instructions to setup an environment, but an overview of key steps in the algorithm is provided here.

1.1 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.

1.2 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:

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

1.3 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.

1.4 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 $r_t(\theta)$, which measures the probability ratio between a current policy π_θ (the "actor") and an old policy π_θ^{old}:

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

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

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

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 exactly 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 $[\theta, \phi]$, we update the old policy π_θ^{old} to equal π_θ.

1.5 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`

Solution: See code!

1.6 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.

1.7 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.

Solution: Recall the naive computation for all the values in a given trajectory: For each time step t_i from $i \to n$, we compute the sum of the following $n - i + 1$ elements weighted by the discount factor. The number of operations is equal to $1 + 2 + 3 + ... + n = \frac{n(n-1)}{2} = O(n^2)$. Note that this solution requires a constant amount of space.

For the linear time solution, we can use an array to store previously computed values and take advantage of the recurrent structure of the rewards in a given trajectory. The reward at time step t can be expressed as:

$$G_t = r_t + \gamma G_{t+1}$$

Following the base case where $G_T = r_T$, we can loop from the end of the trajectory to the beginning and use the following recurrent relationship to compute all rewards. Since we only consider each time step once and use an array to store all intermediate reward values, this solution is $O(n)$ time and space □

(b) (3 pts) Consider the cases in which the PPO update results in the gradient equalling zero. Express these cases mathematically and explain why PPO behaves in this manner.
Solution: Recall the loss function for PPO:

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

where D is the number of trajectories considered in this batch, T_i is the number of steps in a given trajectory, r_θ is the ratio of probabilities between the new policy and the old policy, and $\hat{A}_t^i$ is the difference between the reward we observed and our critic evaluation of the state. Its important to note that we are computing the gradient of our clipped loss with respect to the parameters of our policy θ.

Consider the following cases:

- $r_\theta > 1+\epsilon$: When the probability of taking this action has increased a lot from the old policy to the new policy, the ratio of policy probabilities, r_θ, will also be large. In this case, the clip function will reduce the second argument in the min function to be $(1+\epsilon)\hat{A}_t^i$. When the advantage function is positive, the empirical reward observed is larger than our baselines prediction. If $r_\theta > 1+\epsilon$ and $\hat{A}_t^i > 0 \implies \hat{A}_t^i r_\theta > (1+\epsilon)\hat{A}_t^i$ and the min function will return $(1+\epsilon)\hat{A}_t^i$. Since the output is not a function of θ, the gradient of the clip loss is zero.
- $r_\theta < 1-\epsilon$: When the probability of taking this action has decreased a lot from the old policy to the new policy, the ratio of policy probabilities, r_θ, will be small. When the advantage function is negative, the empirical reward observed is smaller than our baselines prediction. If $r_\theta < 1-\epsilon$ and $\hat{A}_t^i < 0 \implies \hat{A}_t^i r_\theta > (1-\epsilon)\hat{A}_t^i$ and the min function will return $(1-\epsilon)\hat{A}_t^i$. Note that this is because multiplying an inequality by a negative number flips the sign. Since the output is not a function of θ, the gradient of the clip loss is zero.

Intuitively, when our new policy is more more likely than our old policy to take an action which yielded higher empirical reward than our baseline, we dont want this large change in policy and large empirical reward to swing our behavior. The same logic applies when our new policy has changed its action probability to occur significantly less often and we get an empirical reward much lower than the baseline. Its important to see that for the PPO updates, we only compute the gradient of the policy when our ratio of new and old policies is within the range and when the ratio and the observed advantage are opposite signs. □

(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 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?

Solution: The loss function we optimize for REINFORCE depends only on the log probability of the policy at the current state and action and the advantage at that time step. PPO, however, computes the ratio of probabilities between the new policy and the old policy in order to ensure the updates to the policy are not too large. Computing the log-probability information during the roll-out of the state-action-rewards enables us to compute the denominator of the ratio during steps of PPO. Note that for a given batch of trajectories in PPO, we update the parameters of the policy $update_freq$ number of times where as in REINFORCE we do a single update per batch. Each time we do an update step in PPO, we are updating the policy parameters but the probabilities of the actions taken during the roll-out have been caches and are fixed. If we hadn't collected the probability of the actions during the roll-out, we could store a copy of the old model parameters and compute the old probabilities on the fly (less efficient!).

(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:

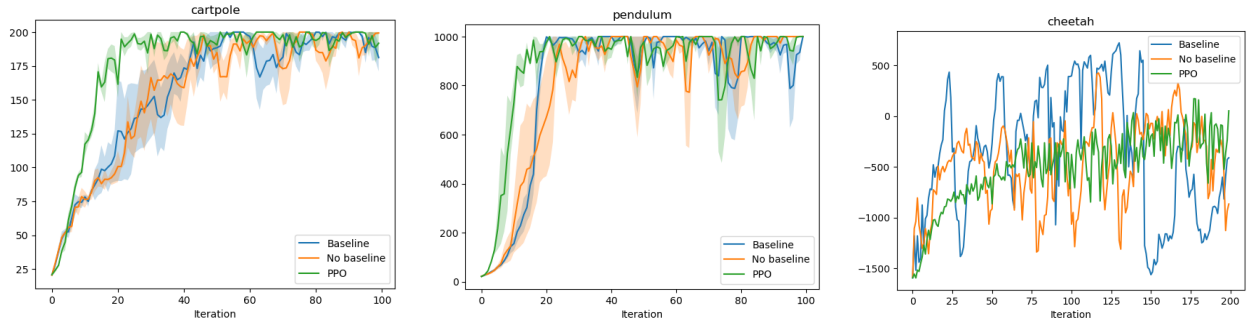
```
python plot.py --env-name ENV --seeds SEEDS
```

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)

Solution: AS we can see from the below plots, all of the solutions achieve the max reward desired. Cartpole and Pendulum are both easier, less stochastic environments where as cheetah is much more noisy. Its good to see that for the very noisy environments that the PPO method makes the training in the cheetah environments have less variance. This is due to PPO not taking gradient steps when our policy has changed to much compared to the previous step and the advantage function is aligned with the highly perturbed policy- ie, when the sample advantage is positive and our policy has changed a lot towards the action or when the sample advantage is negative and our policy has changed a lot against the action. These update steps would cause large changes in our policy gradient step and would cause more stochasticity .



2 Best Arm Identification in Multi-armed Bandit (35pts)

In many experimental settings we are interested in quickly identifying the "best" of a set of potential interventions, such as finding the best of a set of experimental drugs at treating cancer, or the website design that maximizes user subscriptions. Here we may be interested in efficient pure exploration, seeking to quickly identify the best arm for future use.

In this problem, we bound how many samples may be needed to find the best or near-optimal intervention. We frame this as a multi-armed bandit with rewards bounded in $[0, 1]$. Recall a bandit problem can be considered as a finite-horizon MDP with just one state ($|\mathcal{S}| = 1$) and horizon 1: each episode consists of taking a single action and observing a reward. In the bandit setting – unlike in standard RL – the action (or "arm") taken does not affect the distribution of future states. We assume a simple multi-armed bandit, meaning that $1 < |\mathcal{A}| < \infty$. Since there is only one state, a policy is simply a distribution over actions. There are exactly $|\mathcal{A}|$ different deterministic policies. Your goal is to design a simple algorithm to identify a near-optimal arm with high probability.

We recall Hoeffding's inequality: if $X_1, \ldots, X_n$ are i.i.d. random variables satisfying $0 \leq X_i \leq 1$ with probability 1 for all i, $\overline{X} = \mathbb{E}[X_1] = \cdots = \mathbb{E}[X_n]$ is the expected value of the random variables, and $\widehat{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean, then for any $\delta > 0$ we have

$$\Pr\left(|\widehat{X} - \overline{X}| > \sqrt{\frac{\log(2/\delta)}{2n}}\right) < \delta. \tag{1}$$

Assuming that the rewards are bounded in $[0, 1]$, we propose this simple strategy: pull each arm n_e times, and return the action with the highest average payout $\widehat{r}_a$. The purpose of this exercise is to study the number of samples required to output an arm that is at least ϵ-optimal with high probability. Intuitively, as n_e increases the empirical average of the payout $\widehat{r}_a$ converges to its expected value $\overline{r}_a$ for every action a, and so choosing the arm with the highest empirical payout $\widehat{r}_a$ corresponds to approximately choosing the arm with the highest expected payout $\overline{r}_a$.

(a) (15 pts) We start by bounding the probability of the "bad event" in which the empirical mean of some arm differs significantly from its expected return. Starting from Hoeffding's inequality with n_e samples allocated to every action, show that:

$$\Pr\left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right) < |\mathcal{A}|\delta. \tag{2}$$

Solution: Let the number of actions $|\mathcal{A}| = m$. We can apply Hoeffding's inequality to each of the arms in the bandits framework where each of the m arms has a different mean population $\overline{r}_a$ and we compute n_e number of samples from each of the arms resulting in m sampled means $\widehat{r}_a$. The Hoeffding's inequality lets us bound the probability of the sample mean deviating more than a constant from the population mean. Consider the set of m events where each event in this set is defined by:

$$A_k = \left(|\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right)$$

We know that the $\Pr(A_k) < \delta \;\forall\; k \in [m]$. We want to bound the probability of any of these events occurring after we have sampled from all of the arms. Recall from basic probability theory that if we have two events A and B and we wish to compute the probability of A or B occurring, we have that $\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B] \leq \Pr[A] + \Pr[B]$. This formula has a general form where we can consider an set of k events.

$$\Pr\left[\bigcup_{i=1}^{m} A_i\right] \leq \sum_{i=1}^{m} \Pr[A_i]$$

$$\Pr\left[\bigcup_{i=1}^{m} A_i\right] < \sum_{i=1}^{m} \delta = m\delta$$

(b) (20 pts) After pulling each arm (action) n_e times our algorithm returns the arm with the highest empirical mean:

$$a^\dagger = \arg\max_a \widehat{r}_a \tag{3}$$

Notice that $a^\dagger$ is a random variable. Let $a^\star = \arg\max_a \overline{r}_a$ be the true optimal arm. Suppose that we want our algorithm to return at least an ϵ-optimal arm with probability at least $1 - \delta'$, as follows:

$$\Pr\left(\overline{r}_{a^\dagger} \geq \overline{r}_{a^\star} - \epsilon\right) \geq 1 - \delta'. \tag{4}$$

How accurately do we need to estimate each arm in order to pick an arm that is ϵ-optimal? Then derive how many total samples we need total (across all arms) to return an ϵ-optimal arm with prob at least 1- δ' (that satisfies Equation 4). Express your result as a function of the number of actions, the required precision ϵ and the failure probability δ'.

Solution: I claim that in order for our algorithm to choose an ϵ-optimal arm, each of our sample means for each arm, $\widehat{r}_a$, need to be within $\frac{\epsilon}{2}$ away from the population mean $\overline{r}_a$ for each arm. If this constraint is met, then our algorithm will output an action $a^\dagger$ such that the population mean of the reward corresponding to $a^\dagger$ will be ϵ close to the population mean of the optimal action, $a^\star$.

Assume $d(\widehat{r}_{a_k}, \overline{r}_{a_k}) < \frac{\epsilon}{2}\ \forall\ k\ \in [m]$. Since there is a finite number of arms the agent can choose from, we know there is an arm with the largest population reward. Denote this action as $a^\star$. Denote the action with the second largest population mean as a. Consider the following two cases:

- $d(\overline{r}_{a^\star}, \overline{r}_a) > \epsilon$: If the population means for the best and second best arm are farther than ϵ apart and we know that the region in which our sample mean can be drawn from is an $\frac{\epsilon}{2}$ ball around each population mean, this implies that our algorithm will always return $r_{a^\star}$ since the largest sample mean will always be drawn from the arm with the largest population mean
- $d(\overline{r}_{a^\star}, \overline{r}_a) \leq \epsilon$: If the population means for the best and second best arm are less than ϵ apart, then its possible that we get a larger sample mean from the sampling from the second best arm than we do from sampling the best arm. In this case, our algorithm will pick action a. Since, however, we know that the population means are within ϵ of each other, our solution is still epsilon optimal!

Thus, we have shown that if each of the sample means we draw from each arm are within $\frac{\epsilon}{2}$ of their population mean, this implies that our algorithm will be ϵ optimal. Now, all that's left to argue is what type of bound we can place on the probability of this event (all samples being close to their population mean) occurring. Recall our result from part (a) which placed a bound on the probability of at least one of the sample means falling outside the population mean:

$$\Pr\left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right) < |\mathcal{A}|\delta.$$

Setting the constant $\sqrt{\frac{\log(2/\delta)}{2n_e}} = \frac{\epsilon}{2}$, we can solve for the number of samples needed to bound the probability that at least one of our sample rewards will deviate more than $\frac{\epsilon}{2}$ from the population mean:

$$\frac{-\exp\left\{\frac{n_e\epsilon^2}{2}\right\}}{2} = \delta$$

Now we can rewrite the above bound as:

$$\Pr\left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\widehat{r}_a - \overline{r}_a| > \frac{\epsilon}{2}\right) < \delta'$$

where $|\mathcal{A}|\frac{-\exp\left\{\frac{n_e\epsilon^2}{2}\right\}}{2} = \delta'$. Since we need to draw n_e samples from each arm, the total number of samples N required is equal to:

$$N = m \times n_e = m \times \frac{2\log(2/\delta)}{\epsilon^2}$$

By taking the compliment of our re-expressed union bound, we arrive at the desired result for our randomized algorithm:

$$\Pr\left(\forall a \in \mathcal{A} \quad \text{s.t.} \quad |\widehat{r}_a - \overline{r}_a| \leq \frac{\epsilon}{2}\right) \geq 1 - \delta'$$

Since we already proved that this constraint implies that our algorithm returns an ϵ optimal arm, this is equivalent to:

$$\Pr\left(\overline{r}_{a^\dagger} \geq \overline{r}_{a^\star} - \epsilon\right) \geq 1 - \delta'.$$

□