Homework 2: Policy Gradient, Off-Policy Policy Gradient, and Natural Policy Gradient

CS 4789/5789: Introduction to Reinforcement Learning

(Due April 16 11:59 ET)

0 Instructions

For each question in this HW, please list all your collaborators and reference materials (beyond those specified on the website) that were used for this homework. Please add your remarks in a "Question 0".

1 The Magic of Importance Weighting *[10 points]*

We consider importance weighting trick here, particularly we will see how to use that to get some unbiased estimation. Let us consider the following simplified problem. We have k actions, $a_1, \dots, a_k$ (you can think about it as an MDP with zero states), and each action has its own reward $r_1, \dots, r_k$. Denote $\mathbf{r} = [r_1, \dots, r_k]^\top$ as the k-dim vector. We have a sampling distribution $\rho \in \Delta(k)$ over actions.

Now consider the following sampling and estimation process. We randomly sample an index of action from ρ, i.e., $i \sim \rho$, observe its reward r_i, and record the probability of the sampled index, i.e., $p_i := \rho(i)$. We form the following vector $\bar{\mathbf{r}}$ which has zero everywhere except the i-th entry containing the value r_i/p_i. In other words, we have

$$\bar{\mathbf{r}}_j = \frac{\mathbf{1}(j = i) r_j}{p_i}, \forall j$$

(this is correct, as the indicator function will just be zero for any $j \neq i$, and it will be 1 when $j = i$).

What we can show is that this estimator $\bar{\mathbf{r}}$ is indeed an unbiased estimate of the ground truth reward vector $\mathbf{r}$! This is kinda magic, because we only tried one sampled arm, but by importance weighting, we simply get an unbiased estimate of the entire rewards of all arms.

So let us prove this property. Recall that $\bar{\mathbf{r}}$ is a random quantity, i.e., it depends on the random index i that we sampled from ρ.

Q: Prove the following property:

$$\mathbb{E}\left[\bar{\mathbf{r}}\right] = \mathbf{r}.$$

Hint: What is the source of the randomness in our estimator $\bar{\mathbf{r}}$? and then prove unbiasness for each coordinate.

The unbiased estimator is used very commonly in ML, RL, causality, and some real applications such as news article recommendation.

2 Simplify REINFORCE Formulation *[30 points]*

We consider the following Finite horizon Markov Decision Process $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, H, r, P, \mu_0\}$ where μ_0 is the initial state distribution, i.e., $s_0 \sim \mu_0$.

Recall in the class we developed the PG formulation using REINFORCE:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1} \nabla \ln \pi_\theta(a_h|s_h) \cdot R(\tau)\right],$$

where $R(\tau) = \sum_{h=0}^{H-1} r_h$ with $r_h = r(s_h, a_h)$.

However, the Markov property tells us that conditioned on s_h, modifying action a_h is only going to affect the future (the past is the past, and is independent of a_h and future conditioned s_h). So let us prove the following, which actually matches to the intuition above:

Q: Prove the following equality:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1} \nabla_\theta \ln \pi_\theta(a_h|s_h) \cdot \left(\sum_{t=h}^{H-1} r_t - b\right)\right], \tag{1}$$

where $b \in \mathbb{R}^+$ is some constant.

Hint, first prove the following:

$$\mathbb{E}_{\tau\sim\rho_\theta}\left[\nabla_\theta \ln \pi_\theta(a_h|s_h) r_t\right] = 0$$

for any h, t with $t < h$. Here you might want to remind yourself how did we prove that a baseline does not affect the gradient. Also here you can use the fact that $\mathbb{E}_{x,y}[f(x)] = \mathbb{E}_x[f(x)]$ which allows you to ignore the further part of the trajectory that starts at time step $h+1$. The property about conditional expectation might be helpful too: $\mathbb{E}_{x,y}[f(x,y)] = \mathbb{E}_y[\mathbb{E}_x[f(x,y)|y]]$.

Remark Namely, we replace the sum over the entire trajectory by the sum of rewards starting from time step h (with an additional constant b as the baseline). This in practice will help on variance reduction since now for each term h, the randomness is always coming from the partial further trajectory (hence less randomness) rather than the entire trajectory.

3 Off-policy Policy Gradient *[10 points]*

In this section, we are going to study the off-policy PG formulation. We consider the following Finite horizon Markov Decision Process $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, H, r, P, \mu_0\}$ where μ_0 is the initial state distribution, i.e., $s_0 \sim \mu_0$. We are interested in computing the policy gradient of a policy $\pi : \mathcal{S} \mapsto \Delta(\mathcal{A})$ (i.e., π is a stochastic policy mapping from state to distribution over actions, $a \sim \pi(\cdot|s)$).

3.1 Background and Setting

In the lecture, we talked about how to estimate the policy gradient of π using samples/trajectories sampled from π itself, which is called on-policy estimate. There are situations where we want to perform off-policy estimate, i.e., can we use trajectories/samples from a different policy $\pi^b : \mathcal{S} \mapsto \Delta(\mathcal{A})$ to estimate the policy gradient of π? Off-policy setting is useful when we have large pre-collected offline data and we do not want to execute π in the real system due to some safety concern (think about medical application: we may have large logged data from previous patients, and we do not want to directly collect new samples by deploying some policy π, e.g., a new treatment, on new patients due to safety reason. We hope to estimate the outcome of the new policy using logged offline data).

Denote $\tau^1, \ldots \tau^N$ as N trajectories generated by the behavior policy π^b, i.e., $\tau^i = \{s_h^i, a_h^i, \pi^b(a_h^i|s_h^i), r_h^i\}_{h=0}^{H-1}$, where $s_0^i \sim \mu_0$, $a_h^i \sim \pi^b(\cdot|s_h^i), s_{h+1}^i \sim P(\cdot|s_h^i, a_h^i), r_h^i = r(s_h^i, a_h^i)$. These N trajectories is our offline dataset. We are going to assume that π^b is diverse enough to capture π, i.e.,

$$\forall s, a, \frac{\pi(a|s)}{\pi^b(a|s)} < C, \text{ where } C \in \mathbb{R}^+.$$

3.2 Unbiased Estimator of Off-policy PG

Below we show that how we can use the offline dataset to compute an unbiased estimate of the gradient of π, without collecting any samples/trajectories from π itself. For notation simplicity, we drop the parameterization on π, and simply denote $\nabla \ln \pi(a|s)$ as the gradient of the log-likelihood of π with respect to its internal parameters (i.e., we use $\nabla \ln \pi(a|s)$ in short of $\nabla_\theta \ln \pi_\theta(a|s)$).

Q: Prove that the following estimator is an unbiased estimate of the original PG of π:

$$\frac{1}{N} \sum_{i=1}^{N} \left(\prod_{h=0}^{H-1} \frac{\pi(a_h^i|s_h^i)}{\pi^b(a_h^i|s_h^i)} \right) \sum_{h=0}^{H-1} \nabla \ln \pi(a_h^i|s_h^i) \left(R(\tau^i) \right), \tag{2}$$

where $R(\tau^i) = \sum_{h=0}^{H-1} r_h^i$, i.e., the total reward of trajectory τ^i (In other words, you just need to derive the formulation with the right expectation corresponding to off-policy).

With the above unbiased estimator, we can simply run stochastic gradient ascent algorithm to optimize policy in a completely offline manner, without ever interacting with the system to get online samples. Of course you can apply the variance reduction tricks here as well.

4 Natural Policy Gradient (Programming) *[50 points]*

In this section, we will walk through Natural Policy Gradient (NPG) and also implement it for CarPole Simulation. You can find the starter code on https://github.com/cs4789-s21/PA2

4.1 Background and Setting

We will consider the default CartPole simulator in OpenAI gym where we have two discrete actions $\mathcal{A} = \{0, 1\}$ (here 0 and 1 are the index of actions, and physically, 0 means applying a left push to the cart, and 1 means applying a right push to the cart. You just need to compute a stochastic policy which samples 0 or 1, and feed it to the step function which will do the rest of the job for you). Before defining the policy parameterization, let us first featuerize our state-action pair. Specifically, we will use random fourier feature (RFF) $\phi(s, a) \in \mathbb{R}^d$, where d is the dimension of RFF feature. RFF is a randomized algorithm that takes the concatenation of (s, a) as input, outputs a vector $\phi(s, a) \in \mathbb{R}^d$, such that it approximates the RBF kernel, i.e., for any $(s, a), (s', a')$ pair, we have:

$$\lim_{d\to\infty} \langle \phi(s, a), \phi(s', a') \rangle = k\left([s, a], [s', a']\right),$$

where k is the RBF kernel on the concatenation of state-action (we denote $[s, a]$ as the vector $[s^\top, a]^\top$). In summary, RFF feature approximates RBF kernel, but allows us to operate in the primal space rather than the dual space where we need to compute and invert the Gram matrix (recall Kernel trick and Kernel methods from the ML introduction course) which is very computationally expensive and does not scale well to large datasets.

We parameterization our policy as follows:

$$\pi_\theta(a|s) \propto \exp\left(\theta^\top \phi(s, a)\right),$$

where the parameters $\theta \in \mathbb{R}^d$. Our goal is of course to find the best θ such that the resulting π_θ achieves large expected total reward.

4.2 Gradient of Policy's Log-likelihood

Q Given (s, a), first write down the expression $\nabla_\theta \ln \pi_\theta(a|s)$ below. Now go to `utils.py` to implement the computation of $\nabla_\theta \ln \pi_\theta(a|s)$ with `compute_log_softmax_grad`. You can also implement `compute_softmax` and `compute_action_distribution` first to use them to calculate the gradient.

4.3 Fisher Information Matrix and Policy Gradient

We consider finite horizon MDP. Let us now computer the fisher information matrix. Let us consider a policy π_θ. Recall the definition of Fisher information matrix:

$$F_\theta = \mathbb{E}_{s,a\sim d^{\pi_\theta}_{\mu_0}} \left[\nabla_\theta \ln \pi_\theta(a|s) \left(\nabla_\theta \ln \pi_\theta(a|s)\right)^\top\right] \in \mathbb{R}^{d\times d}.$$

We approximate F_θ using trajectories sampled from π_θ. We first sample N trajectories $\tau^1, \ldots, \tau^N$ from π_θ, where $\tau^i = \{s^i_h, a^i_h, r^i_h\}_{h=0}^{H-1}$ with $s^i_0 \sim \mu_0$. We approximate F_θ using all (s, a) pairs:

$$\widehat{F}_\theta = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{H}\sum_{h=0}^{H-1} \nabla_\theta \ln \pi_\theta(a^i_h|s^i_h) \nabla_\theta \ln \pi_\theta(a^i_h|s^i_h)^\top\right] + \lambda I,$$

where $\lambda \in \mathbb{R}^+$ is the regularization for forcing positive definiteness.

Remark Note the way we estimate the fisher information. Instead of doing the roll-in procedure we discussed in the class to get $s, a \sim d^{\pi_\theta}_{\mu_0}$ (this is the correct way to ensure the samples are i.i.d), we simply sample N trajectories, and then average over all state-action (s_h, a_h) pairs from all N trajectories. This way, we lose the i.i.d property (these state-action pairs are dependent), but we gain sample efficiency by using all data.

Q First, go to `train.py` and implement the `sample` function to sample trajectories using the current policy. Then go to file `utils.py` and implement $\widehat{F}_\theta$ in `compute_fisher_matrix`. Note that in OpenAI Gym cartpole, there is a termination criteria when the par or the cart is too far away from the goal (i.e., during execution, if the termination criteria is met or it hits the last time step H, the simulator will return *done = True* in step function. During generating a trajectory, it is possible that we will just terminate the trajectory early since we might meet the termination criteria before getting to the last time step H). Hence, we will see that when we collect trajectories, each trajectory might have different lengths. Thus, for estimating F_θ, we need to properly average over the trajectory length.

Q Denote V^θ as the objective function $V^\theta = \mathbb{E}\left[\sum_{h=0}^{H-1} r(s_h, a_h) | s_0 \sim \mu_0, a_h \sim \pi_\theta(\cdot|s_h)\right]$. Similarly, let us implement the policy gradient (Eq. 1), i.e.,

$$\widehat{\nabla} V^\theta = \frac{1}{N} \sum_{i=1}^{N} \left[\frac{1}{H} \sum_{h=0}^{H-1} \nabla_\theta \ln \pi_\theta(a^i_h | s^i_h) \left[\sum_{t=h}^{H-1} r^i_t - b \right] \right],$$

where b is a constant baseline $b = \sum_{i=1}^{N} R(\tau^i)/N$, i.e., the average total reward over a trajectory. Again be mindful that each trajectory might have different lengths due to early termination. Go to `utils.py` to implement this PG estimator in `compute_value_gradient`.

4.4 Implement the step size

With $\widehat{F}_\theta$ and $\widehat{\nabla}_\theta V^\theta$, recall that NPG has the following form:

$$\theta' := \theta + \eta \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta.$$

We need to specify the step size η here. Recall the trust region interpretation of NPG. We perform incremental update such that the KL divergence between the trajectory distributions of the two successive policies are not that big. Recall that the KL divergence $KL(\rho^{\pi_\theta} | \rho^{\pi_{\theta'}})$ can be approximate by Fisher information matrix as follows (ignoring constant factors):

$$KL(\rho^{\pi_\theta} | \rho^{\pi_{\theta'}}) \approx (\theta - \theta')^\top F_\theta (\theta - \theta').$$

As we explained in the lecture, instead of setting learning rate as the hyper-parameter, we set the trust region (which has a more transparent interpretation) size as a hyper parameter, i.e., we set δ such that:

$$KL(\rho^{\pi_\theta} | \rho^{\pi_{\theta'}}) \approx (\theta - \theta')^\top \widehat{F}_\theta (\theta - \theta') \leq \delta.$$

Since $\theta' - \theta = \eta \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta$, we have:

$$\eta^2 (\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta \leq \delta.$$

Solving for η we get $\eta \leq \sqrt{\frac{\delta}{(\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta}}$. We will just set $\eta = \sqrt{\frac{\delta}{(\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta}}$, i.e., be aggressive on setting learning rate while subject to the trust region constraint. To ensure numerical stability when the denominator is close to zero, we add $\epsilon = 1e - 6$ to the denominator so the expression we will use is

$$\eta = \sqrt{\frac{\delta}{(\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta + \epsilon}} \tag{3}$$

Q Now go to `utils.py` and implement these step size computation in `compute_eta`.

4.5 Putting Everything Together

Now we can start putting all pieces together. Go to `train.py` to implement the main framework in `train`. We will iterate the NPG process to $T = 20$ iterations, report the performance of each π_{θ_t} for $t = 0$ to 19, and plot the performance curve.
Hint: your algorithm should achieve average reward of over 190 in about 15 steps if implemented correctly.