CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 9

This discussion focuses on reinforcement learning, covering policy gradient, actor-critic, and Q-learning algorithms.

1 Policy Gradients

Recall from the previous discussion, we formulated sequential decision making problems as **Markov Decision Problems** (MDPs). Our goal with reinforcement learning algorithms is to efficiently solve these MDPs.

We recall that each policy $\pi(a|s)$ induces a distribution over trajectories $\tau = ((s_1, a_1), \ldots, (s_T, a_T))$ in the MDP. Our objective can be formalized as finding a policy π_θ (given by some parameters θ) to maximize the expected sum of rewards over the trajectory.

$$\max_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} r(s_t, a_t)\right].$$

Now that we have an optimization problem, the natural thing to do is to solve it using stochastic gradient ascent, which gives us *policy gradient* algorithms.

1.1 Deriving Policy Gradients

We first consider a more general problem of optimizing the expectation of a random variable with respect to some parameters that control it's distribution. Suppose we have some distributions $p_\theta(X)$ over a random variable X, and we wish to compute the gradient of the expectation of a function of X

$$\nabla_\theta \mathbb{E}_{X\sim p_\theta}[f(X)].$$

We first verify the identity

$$\begin{aligned}\nabla_\theta \log g(\theta) &= \frac{\nabla_\theta g(\theta)}{g(\theta)} \\ \nabla_\theta g(\theta) &= g(\theta)\nabla_\theta \log g(\theta).\end{aligned}$$

Recalling that we can rewrite expectations as integrals and switch the order of integration and differentiation (under some regularity assumptions), we have

$$\begin{aligned}\nabla_\theta \mathbb{E}_{X\sim p_\theta(X)}[f(X)] &= \nabla_\theta \int p_\theta(x) f(x)\, dx \\ &= \int \nabla_\theta p_\theta(x) f(x)\, dx \\ &= \int p_\theta(x) \nabla_\theta \log p_\theta(x) f(x)\, dx \qquad \text{applying previous identity} \\ &= \mathbb{E}_{X\sim p_\theta(X)}[\nabla_\theta \log p_\theta(x) f(x)].\end{aligned}$$

Now that our gradient is written as an expectation, we can obtain unbiased gradient estimates using only samples from $p_\theta(X)$.

Applying this with trajectories τ as the random variable X and the total reward $R(\tau)$ as the function f, we obtain the basic policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \log \pi_\theta(\tau) R(\tau)].$$

Expanding out the log probability of a trajectory and ignoring terms that don't depend on θ (see calculations from last discussion), we have

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)\right) R(\tau)\right].$$

The policy gradient algorithm then alternates between sampling trajectories from the current policy, computing stochastic gradient estimates, and updating the policies.

1.2 Reducing Variance

While the policy gradient algorithm is very simple, it requires us to sample new trajectories for every single update, and if we don't sample many trajectories for each update, the gradient estimate can be very noisy. We will now talk about several ways we can try to reduce the variance of the policy gradient estimate.

1.2.1 Reward to Go

The first trick is to note that actions later in the trajectory cannot have any effect on earlier rewards, so instead of using the rewards across the whole trajectory $R(\tau)$, we can use the *reward-to-go* estimator $R_t(\tau) = \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$, which only includes the rewards starting from timestep t.

To show this formally, we can rewrite $J(\theta)$ as the expected sum over timesteps of rewards and simplify:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_{t'=1}^{T} \nabla_\theta \mathbb{E}_{\pi_\theta}[r_{t'}] \\
&= \sum_{t'=1}^{T} \mathbb{E}_{r_{t'}}\left[r_{t'} \sum_{t=1}^{t'} \nabla \log \pi_\theta(a_t|s_t)\right] \qquad \text{probability of } r_{t'} \text{ only depends on actions up to } t'.
\end{aligned}$$

Rearranging the summations, the new policy gradient update would then be

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r(s_{t'}, a_{t'})\right)\right].$$

1.2.2 Baselines

Another commonly used technique to reduce variance is to subtract a baseline from the return estimate. In lecture, you saw how subtracting a constant baseline (replacing $R(\tau)$ with $R(\tau) - b$ for some constant b) did

not bias the gradient estimate, with proof as follows:

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\nabla \log \pi_\theta(\tau) b\right] &= b\mathbb{E}_{\pi_\theta}\left[\nabla \log \pi_\theta(\tau)\right] && \text{pull constant outside of expectation} \\
&= b \int \pi_\theta(\tau) \nabla \log \pi_\theta(\tau)\, d\tau \\
&= b \int \pi_\theta(\tau) \frac{\nabla \pi_\theta(\tau)}{\pi_\theta(\tau)}\, d\tau \\
&= b \int \nabla \pi_\theta(\tau)\, d\tau \\
&= b \nabla \int \pi_\theta(\tau)\, d\tau \\
&= b\nabla 1 && \pi_\theta(\tau) \text{ integrates to 1 since it is a probability distribution} \\
&= 0. && \text{derivative of a constant function is 0}
\end{aligned}$$

In lecture, we mentioned that setting $b = \mathbb{E}_{\pi_\theta}[R(\tau)]$ to be the average return of the policy was a reasonable choice to reduce variance. We can provide some intuition why here.

Let us compute the variance of

$$\text{Var}(\nabla \log \pi_\theta(\tau)(R(\tau) - b)) = \mathbb{E}[\nabla \log \pi_\theta(\tau)^2 (R(\tau) - b)^2] - \mathbb{E}[\log \pi_\theta(\tau)(R(\tau) - b)]^2.$$

We note the second term is simply the expected policy gradient squared. Since we know constant baselines can't affect that term (since they leave the gradient estimator unbiased). We will then try to pick the baseline b to minimize the first term, under the simplifying assumption (though not true) that $\nabla \log \pi_\theta(\tau)$ and $R(\tau)$ are independent. Using this independence to separate the expectations, we want to find the baseline b that satisfies

$$\begin{aligned}
\arg\min_b \mathbb{E}[\nabla \log \pi_\theta(\tau)^2 (R(\tau) - b)^2] &= \mathbb{E}[\nabla \log \pi_\theta(\tau)^2] \mathbb{E}[(R(\tau) - b)^2] \\
&= \arg\min_b \mathbb{E}[(R(\tau) - b)^2].
\end{aligned}$$

The minimizing constant b is simply the expectation $\mathbb{E}[R(\tau)]$, which motivates why we subtract the average return as a simple constant baseline. In general, due to the untrue independence assumption we made, there are no guarantees that this baseline minimizes variance (or even reduces variance at all), but it tends to work reasonably well in practice.

Problem 1: State-dependent baselines

Show that a subtracting state-dependent baseline $b(s_t)$ also doe not bias the policy gradient estimator. Hint: break up the summation over the timestep t and consider each timestep separately.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) - b(s_t)\right)\right)\right].$$

Solution 1: State-dependent baselines

Consider a single timestep t, and we will show that the state dependent baseline term has expectation 0. We then have

$$\begin{aligned}
\mathbb{E}_{s_t, a_t \sim \pi_\theta}[\nabla \log \pi_\theta(a_t|s_t) b(s_t)] &= \mathbb{E}_{s_t}\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\nabla \log \pi_\theta(a_t|s_t) b(s_t)\right] | s_t\right] \\
&= \mathbb{E}_{s_t}\left[b(s_t)\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\nabla \log \pi_\theta(a_t|s_t)\right]\right] \\
&= \mathbb{E}_{s_t}[b(s_t) \cdot 0] \\
&= 0.
\end{aligned}$$

Similar in spirit to subtracting out the average return as a baseline, we can thus use the expected future return from state s_t (denoted $V(s_t)$ in the next section) as a baseline to reduce variance.

2 Value functions and Q-learning

For a given policy π, we can define the time-dependent state value function $V_t^\pi(s)$ and the state-action value function $Q_t^\pi(s,a)$ as expectations of future rewards conditioned on being at state s at time t (and taking action a for Q_t^π.).

$$V_t^\pi(s) = \mathbb{E}_\pi\left[\sum_{t'=t}^{T} r_t | s_t = s\right]$$
$$Q_t^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t'=t}^{T} r_{t'} | s_t = s, a_t = a\right].$$

2.1 Discounting

In the infinite horizon setting ($T = \infty$), future rewards no longer depend on the absolute timestep t, so we can then simply use non-time dependent value functions Q^π and V^π. However, infinite horizons can lead to unbounded future returns, so we introduce a *discount factor* $\gamma \in (0,1)$, which tells to prioritize more immediate rewards. The discounted value functions are then

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$$
$$Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right].$$

We note that these two functions are also related by $V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}[Q^\pi(s,a)]$ by the tower property of conditional expectations. In practice, we'll often use discounted returns even if there is a finite horizon T in order to reduce variance and make things easier to estimate, both in value function learning and directly in policy gradient methods.

2.2 Learning Value Functions via the Bellman Equations

To learn value functions, one straightforward approach would be to simply to take Monte Carlo estimates and regress onto the observed returns from sampled trajectories. Similar to our previous policy gradient algorithms, it will suffer due to the returns in trajectories having high variance. In practice, we will often learn value functions with a *dynamic programming* approach based on a set of consistency equations the value functions satisfy, known as the **Bellman equations**.

Separating out the reward at the first timestep from the future rewards, we have

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_0] + \mathbb{E}_\pi\left[\sum_{t=1}^{\infty} \gamma^t r_t | s_0 = s\right] \\ &= \mathbb{E}_\pi[r_0] + \mathbb{E}_{s_1}\left[\mathbb{E}_\pi\left[\sum_{t=1}^{\infty} \gamma^t r_t | s_1 = s_1\right] | s_0 = s\right] \\ &= \mathbb{E}_\pi[r_0] + \gamma \mathbb{E}_{s_1}\left[V(s_1) | s_0 = s\right]. \end{aligned}$$

This tells us the value at the current state is simply the expected immediate reward plus the discount times the expected value of the next state.

Similarly for Q-functions, we can derive

$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s_1}\left[\mathbb{E}_{a_1\sim\pi(\cdot|s_1)}[Q^\pi(s_1,a_1)]\right].$$

If we consider a tabular setting (where we compute exact values for every state), we can find the value function of the policy by satisfying the Bellman equations. Concretely, we can take transitions (s, a, r, s') from the policy's trajectories and update $V(s)$ towards the target value $r + \gamma V(s')$ (with a step size α):

$$V(s) \leftarrow V(s) - \alpha(r(s,a) + \gamma V(s') - V(s)).$$

Note that learning the state value function requires *on-policy* samples from the policy being evaluated, due to the expectation over the immediate action that leads to the immediate reward.

Alternatively, we can learn the Q-values instead with

$$Q(s,a) \leftarrow Q(s,a) - \alpha(r(s,a) + \gamma \mathbb{E}_{a' \sim \pi(a'|s')} Q(s', a') - Q(s,a)),$$

and note that this can be learned with *off-policy* data, as only the future values $Q(s', a')$ depend on the policy being evaluated.

2.2.1 Learning with Function Approximation

We see that the tabular updates above resemble taking a gradient step on the squared error between the current value $V(s)$ and a regression target $r(s,a) + \gamma V(s'))^2$ (similarly for the Q-values). We can straighforwardly extend these updates to a setting where our value functions are parameterized by some parameters ϕ (for example a deep neural network), and take a gradient step on the objective (with some loss function L such as squared error)

$$\mathbb{E}_{s,a,s' \sim \pi} \left[L(V_\phi^\pi(s), r(s,a) + \gamma V_{\bar{\phi}}(s')) \right].$$

Here, $\bar{\phi}$ is an identical copy of the current parameters ϕ, to indicate that we do not pass gradients through the target value $V_{\bar{\phi}}(s')$. Due to this moving target, doing these updates is **not** doing gradient descent on any fixed objective, since the regression targets are constantly changing as we optimize.

In practice, these updates are often unstable due to these moving target values, so we often slow down how the target parameters $\bar{\phi}$ change. One way is to simply keep $\bar{\phi}$ fixed for multiple iterations of updates to ϕ, before copying the current parameters to the target. Another way to slow down the changing target values is to update $\bar{\phi}$ as an exponential moving average of ϕ.

We again emphasize that state-action values, unlike the state values, can be learned directly from off-policy data, leading to taking updates in the direction given by

$$\mathbb{E}_{s,a,s' \sim D} \left[\nabla_\phi L(Q_\phi^\pi(s,a), r(s,a) + \gamma \mathbb{E}_{a' \sim \pi(a'|s} [Q_{\bar{\phi}}(s', a')]) \right].$$

Here, the dataset D can include transitions from other policies, for example the trajectories from previous policies (known as *experience replay*). By reusing experiences from past policies to learn the Q-function of the current policy, learned Q-values can potentially make reinforcement learning much more sample-efficient.

2.3 Using Value Functions in Actor-Critic Algorithms

Assuming we had the exact state values $V(s)$ or state-action values $Q(s,a)$, we can go back to our policy gradient algorithm and verify that the following gradient estimators are all unbiased.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) Q_t^\pi(s_t, a_t))\right)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)(r(s_t, a_t) + \gamma V_{t+1}^\pi(s_{t+1}))\right)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)(r(s_t, a_t) + \gamma V_{t+1}^\pi(s_{t+1}) - \underbrace{V_t^\pi(s_t)}_{\text{state dependent baseline}})\right)\right].
\end{aligned}
$$

Note that if we did have the exact value functions, these would lead to lower variance estimates for our policy gradient, since we have replaced noisy estimates of future returns with their exact expectations. Using learned value functions V_ϕ^π and Q_ϕ^π in the above updates leads to the broad class of *actor-critic* algorithms, where we update our policy (the actor) using learned value functions (the critic). Since we use learned values instead of the true conditional expectations, this will typically bias our policy gradient estimates, but often improves policy learning due to the much decreased variance in the updates.

3 Q-Learning

In the previous section, we discussed value functions and how they could be combined with policy gradient updates to solve RL problems. In Q-learning, we will explore how to discuss how we can learn to solve RL problems without explicitly keeping track of a policy, but instead only keeping track of a value function.

While we previously learned value functions for the current policy π, we can instead try to directly learn the values associated with the *optimal policy* denoted π^*. Let Q^* denote the state-action value of the optimal policy. Then, from our previous section, we know it satisfies a Bellman equation

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[\mathbb{E}_{a' \sim \pi^*(\cdot|s')}[Q^*(s',a')]\right].$$

Now we note that the optimal policy π^* must also take actions that maximize future values. For simplicity, if π^* is a determnistic policy mapping states to actions, then we must have $\pi^*(s) = \arg\max_a Q^*(s,a)$. Substituting this in, we have

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[\max_{a'} Q^*(s',a')\right],$$

which will form the basis of Q-learning algorithms. Q-learning can loosely be seen as always trying to learn the value of a deterministic policy that always take the action that maximizes the current Q^*-values.

Note that Q-learning requires us to be able to compute the optimal action over the Q-values, which typically limits us discrete action spaces (though we can get around this with either approximations to the max or assuming additional structure over the Q-functions).

The classic Q-learning algorithm will then alternate between sampling a new transitions using a policy derived from the Q-function and updating the Q-function on that new transition. In deep Q-learning algorithms, we'll often use experience replay, where we sample batches of previously seen experiences to update along

$$\mathbb{E}_{s,a,s' \sim D}\left[\nabla_\phi L(Q^*_\phi(s,a), r(s,a) + \gamma \max_{a'}[Q^*_{\bar{\phi}}(s',a')])\right].$$

As mentioned previously, this can help make Q-learning much more sample efficient as well as helping stabilize the updates.

3.1 Exploration in Q-Learning

As Q-learning implicitly gives us a deterministic policy, exploration can be very limited if we directly use that policy for exploration. Intuitively, we can fall into a situation where we think we know the optimal behavior, and if we always act optimally according to our beliefs, we never experience anything new that might lead us to better returns.

One simple approach to get around this is the ϵ-greedy strategy. Instead of always taking the optimal action, we will instead take a completely random action with probability ϵ and act optimally otherwise. This way, the random actions can eventually lead us to discover better behaviors.

Another approach is known as Boltzmann exploration, where we explore according to

$$\pi(a|s) \propto \exp\Big(Q^*_\phi(s,a)\Big),$$

which is a random policy that weights each action according to how well we think the action will do.

We again note that unlike policy gradient algorithms, we can trivially explore using other policies since our goal is to learn Q-values directly, which can be done with off-policy data.