Lecture 16: Stochastic Actors

1 Introduction

1.1 Logistics

- For reproducibility projects, expectation is that you not use the paper's code itself.

1.2 Review: DDPG and TD3

DDPG actor loss:

$$\max_{\phi} \mathbb{E}_s[Q(s, a = \pi_\phi(s))]. \tag{1}$$

DDPG critic loss:

$$\max_{\theta} \mathbb{E}_{\substack{(s,a,r,s')\sim\mathcal{B}, \\ a'\sim\pi_{\text{target}}(a'|s')}} \left[\left(Q_\theta(s,a) - (r + \gamma Q_{\text{target}}(s', a')) \right)^2 \right]. \tag{2}$$

Tricks from TD3:

1. additive clipped noise on actions
2. double critics and actors
3. delayed actors update

1.3 Plan for today

- Why use stochastic actors?
- Optimizing forward reward + randomness

2 Stochastic Actors

So far in this course, we've primarily been looking at *deterministic* policies, policies that output a single action for a given state. These policies are sometimes denoted $\pi(s) \to a$. For example, when doing Q-learning, select the single action that maximizes the Q function. Indeed, one can show that every MDP has a deterministic policy as a solution.[1] Intuitively, the space of all deterministic policies is smaller than the set of stochastic policies.

However, optimizing over deterministic policies has a few limitations. The biggest one is about exploration – if we want to learn by trial and error, collecting *diverse* data is important. So, when using a method like DDPG, we have to add some noise to the actions taken in the environment. This means that the actual policy used for data collection is different from the one that we're optimizing. Mathematically, this is fine, because DDPG is an off-policy algorithm. But aesthetically, it would be nice if we could optimize over the same policy that we're using for data collection. Using stochastic policies also has the potential to make optimization easier (intuition: In the TD target, because we're sampling actions now, we are averaging the Q function w.r.t. the action dimension) and avoid local optima.

Broadly, our aim will be

- If there are many ways to solve the task, find all of them!

[1] When there are multiple reward maximizing policies, it's possible that there are also stochastic policies that are optimal.

- If there are many paths to a goal, try all possible paths, but more frequently use short paths.

In class on last week, we looked at the deep *deterministic* policy gradient (DDPG) algorithm, which learns a deterministic actor. One thing that was a bit weird with this method is that you were learning a noiseless policy, but using a noisy policy for exploration. You can imagine scenarios where this would get you into trouble. E.g., if you're crossing a narrow bridge, the noiseless policy can successfully cross this bridge but a noisy policy cannot. So, even though the policy you're learning can successfully cross the bridge, you wouldn't see these successes during training. In many settings, it's convenient to directly optimize the policy that is being used for exploration.

One method for doing that is *stochastic value gradients* (SVG) [1]. This paper introduces a family of methods, and the method that we're referring to here is SVG(0). I'll explain the other members of this family at the end, if time permits. The only difference between SVG and DDPG is that SVG learns a stochastic policy. The critic loss is exactly the same. Intuitively, the actor loss for SVG can be written as

$$\max_{\psi} Q(s, a \sim \pi_\phi(a \mid s)). \tag{3}$$

That is, we optimize the actor so that it samples actions with high Q-value. To write this objective formally, we use an expectation:

$$\max_{\psi} \mathbb{E}_{a\sim\pi_\phi(a|s)}[Q(s,a)]. \tag{4}$$

History of SVG. The SVG paper was released just one month after the DDPG paper, so presumably much of the research happened in parallel. Today, the DDPG paper is cited 26× more than the SVG paper, despite the algorithms being very similar and (when compared on a level playing field) achieving very similar performance (SVG is often slightly better). Perhaps the difference is that DDPG demonstrated more results on high-dimensional tasks (a car racing game, which used pixels)?

2.1 Gradients.

What is the gradient of this objective? How does this compare to the gradient of the actor loss in DDPG?

$$\nabla_a Q(s,a) \nabla_\phi \pi_\phi(s) \tag{DDPG}$$

$$\mathbb{E}_{a\sim\pi_\phi(a|s)}[Q(s,a)\nabla_\phi \log \pi_\phi(a \mid s)]. \tag{SVG(0)}$$

Note that the DDPG actor loss makes use of the gradients of the Q function, while the SVG loss doesn't. Intuitively, gradient information is really helpful – it tells you the direction you should head it. It's like standing on a mountain and trying to find your way down; you could take a step in random directions and see which directions take you up/down, or you could directly feel the slope of the hillside under your feet. Said in other words, the gradient of the Q function is giving us a "clue" for how to update the policy, a clue that DDPG seems to exploit but the SVG gradient above doesn't exploit. *Is there a way to compute the gradients for the actor in SVG that makes use of the gradient of Q?*

Yes, for most parametrizations of the policy. For simplicity, we'll consider a policy that outputs a Gaussian distribution over actions, and assume that the standard deviation is fixed at 1:

$$\pi_\phi(a \mid s) = \mathcal{N}(a; \mu = \mu_\phi(s), \sigma = 1). \tag{5}$$

The same trick can readily be applied to other parametrizations. The key idea, known as the *reparametrization trick*[2], is to think about how you implement sampling from the policy. The standard way of doing this for a Gaussian distribution with mean μ and standard deviation σ is

$$x = \mu + \sigma z, \qquad \text{where} \quad z \sim \mathcal{N}(0,1). \tag{6}$$

Applying this trick to the policy, we get

$$a = \mu_\phi(s) + z. \tag{7}$$

We can use this reparametrization to rewrite the SVG actor objective:

$$\max_\phi \mathbb{E}_{z\sim\mathcal{N}(0,1)}[Q(s, a = \mu_\phi(s) + z)]. \tag{8}$$

The key difference from the original SVG actor objective is that the sampling distribution no longer depends on the policy parameters. Remember back to when we were deriving the policy gradient: the reason it was challenging was because the sampling distribution depended on our policy parameters. But with this reparametrization trick, we can write this actor objective with a sampling distribution that doesn't depend on the policy parameters.

Because the sampling distribution doesn't depend on the policy parameters, we'll be able to directly push the gradient operator inside the expectation:

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{z\sim\mathcal{N}(0,1)}[Q(s, a = \mu_\phi(s) + z)] &= \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\nabla_\phi Q(s, a = \mu_\phi(s) + z)] &(9)\\
&= \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\nabla_\phi Q(s, a = \mu_\phi(s) + z)\nabla_\phi(\mu_\phi(s) + z)] &(10)\\
&= \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\nabla_\phi Q(s, a = \mu_\phi(s) + z)\nabla_\phi \mu_\phi(s)]. &(11)
\end{aligned}$$

Exercise: why can't we apply the reparametrization trick to the policy gradient?

Implementation. Note that implementing the parametrization trick is very easy: just parametrize your neural network to output $\mu_\phi(s), \sigma_\phi(s)$, and then feed $\mu + \sigma \cdot z$ into the Q function. Backprop will take care of the rest!

Transition: How can we move beyond have a fixed degree of randomness at each state?

3 A First Attempt [3]

"The use of entropy maximization is designed to help keep the search alive by preventing convergence to a single choice of output, especially when several choices all lead to roughly the same reinforcement value." – Williams and Peng [3]

To start, we need a measure of stochasticity. We'll use perhaps the most natural measure: entropy. In particular, we'll look at the entropy of the distribution over actions, given the current state:

$$\mathcal{H}_\pi[a \mid s] \triangleq \mathbb{E}_{\pi(a|s)}[-\log \pi(a \mid s)]. \tag{12}$$

Note the negative sign, which means that maximizing entropy corresponds to trying to sample actions with low likelihood.

One straightforward way of learning policies that maximize entropy is to add the above entropy maximization term into the actor loss of an actor-critic method. The quote above is from one early paper that does this. We will use a weight of α on the entropy term

$$\max_\pi \mathbb{E}_\pi[Q(s,a) - \alpha \log \pi] \qquad \text{(actor loss)}$$

$$\min_Q \mathbb{E}_{p(s,a,s')}\left[\left(Q(s,a) - (r(s,a) + \gamma Q(s', a'))\right)^2\right]. \qquad \text{(critic loss)}$$

Intuitively, this idea make sense – we're optimizing the policy to take actions that have high entropy *now* and yield high returns *in the future*. But, we can do better. In particular, if we want to maximize entropy across an entire trajectory, then t may be worthwhile to take low-entropy actions now so that we can take higher entropy actions in the future. For example, if the agent is walking along the top of a cliff, initially taking a few (deterministic) actions away from the cliff can enable it to act more randomly in the future. This idea of taking actions now that put the agent in a good position for obtaining good outcomes (either high rewards or the ability to act randomly) looks exactly the standard RL problem.

References

[1] Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., and Tassa, Y. (2015). Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28.

[2] Kingma, D. P., Welling, M., et al. (2013). Auto-encoding variational bayes. *International Conference of Learning Representations*.

[3] Williams, R. J. and Peng, J. (1991). Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268.