Actually the advantage of this approach lies in the fact that it helps the network to generalize learning across different actions more effectively. By decoupling the value and advantage, the network can learn which states are valuable without being influenced by the action advantages, and at the same time, it can learn which actions are beneficial without being affected by the differences in value between states.

The more prosaic explanation is that the function decomposition is always technically correct (for an MDP). Coding the network like this incorporates *known structure of the problem* into the network, which otherwise it may have to spend resources on learning. So it's a way of injecting the designer's knowledge of reinforcement learning problems into the architecture of the network.

2.5 Q-Learning with Continuous Actions

In this section, we will introduce two methods for Q-learning with continuous action space.

2.5.1 Deep Deterministic Policy Gradients

To begin with, let's see why we need DDPG for continuous actions planning. Here is a concrete example: we are trying to solve the classic Inverted Pendulum control problem. In this setting, we can take only two actions: swing left or swing right. What make this problem challenging for Q-Learning Algorithms is that actions are continuous instead of being discrete. That is, instead of using two discrete actions like -1 or +1, we have to select from infinite actions in a given range. In DQN we have the greedy policy assumption, so that in target network you have to do the 'max-trick':

$$\pi\left(\mathbf{a}_t \mid \mathbf{s}_t\right)=\begin{cases}1 \text{ if } \mathbf{a}_t=\arg\max_{\mathbf{a}_t} Q_\phi\left(\mathbf{s}_t, \mathbf{a}_t\right)\\ 0 \text{ otherwise}\end{cases} \tag{2.5.1}$$

$$y_j=r_j+\gamma \max_{\mathbf{a}_j'} Q_{\phi'}\left(\mathbf{s}_j', \mathbf{a}_j'\right) \tag{2.5.2}$$

But turning to continuous action space, it's no longer possible to perform the $\arg\max$.

There are several solutions to performing the max:

Option 1: sample and optimize

The first option is to use optimization techniques. Gradient based optimization (e.g., SGD) involves multiple steps of optimization and would be a bit slow in the inner loop. It turns out that stochastic optimization would be a better option since action space here is typically low-dimensional.

In stead of giving a close form solution of optimization on continuous action space, a deadly simple way is to sample $(\mathbf{a}_1, \ldots, \mathbf{a}_N)$ from some distribution:

$$\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) \approx \max\left\{Q\left(\mathbf{s}, \mathbf{a}_1\right), \ldots, Q\left(\mathbf{s}, \mathbf{a}_N\right)\right\} \tag{2.5.3}$$

However this does not work well in practice majorly because it is generally non-trivial to sample from an arbitrary distribution. Some more accurate solutions, like **cross-entropy method (CEM)** (simple iterative stochastic optimization) or **CMA-ES** (substantially less simple iterative stochastic optimization) works OK for up to about 40 dimensions.

Option 2: Better Q function structure

The second option is to use function class that is inherently *easy to optimize.* One example is to use **quadratic function**:

$$Q_\phi(\mathbf{s}, \mathbf{a})=-\frac{1}{2}\left(\mathbf{a}-\mu_\phi(\mathbf{s})\right)^T P_\phi(\mathbf{s})\left(\mathbf{a}-\mu_\phi(\mathbf{s})\right)+V_\phi(\mathbf{s}) \tag{2.5.4}$$

This approach is called **Normalized Advantage Functions (NAF)**. In this case the object to be maximized is given directly by the networks' output:

$$\arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})=\mu_\phi(\mathbf{s}) \quad \max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})=V_\phi(\mathbf{s}) \tag{2.5.5}$$

Using NAF does not change the algorithm itself, so that it's just as efficient as Q-learning, but it sacrifices the representational power and is less expressive.

Option 3: learn an approximate maximizer

Remember we have this equation always hold:

$$\max_{\mathbf{a}} Q_\phi(\mathbf{s},\mathbf{a}) = Q_\phi\left(\mathbf{s}, \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s},\mathbf{a})\right) \tag{2.5.6}$$

So a big idea based on this is to train a neural network $\mu_\theta(s)$ so that $\mu_\theta(\mathbf{s}) \approx \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s},\mathbf{a})$. In practice, this is equivlant to finding a set of parameters θ that maximize $Q_\phi(\mathbf{s},\mu_\theta(\mathbf{s}))$, i.e., $\theta \leftarrow \arg\max_\theta Q_\phi\left(\mathbf{s},\mu_\theta(\mathbf{s})\right)$.

Given the chain rule:

$$\frac{dQ_\phi}{d\theta} = \frac{d\mathbf{a}}{d\theta}\frac{dQ_\phi}{d\mathbf{a}} \tag{2.5.7}$$

The update rule for theta should be:

$$\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}\left(\mathbf{s}_j\right)\frac{dQ_\phi}{d\mathbf{a}}\left(\mathbf{s}_j,\mu\left(\mathbf{s}_j\right)\right) \tag{2.5.8}$$

The network $\mu_\theta(\mathbf{s})$ is sometimes referred to as *deterministic network*.

Now merging this idea to DQN, we have Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015)

Algorithm 14 DDPG algorithm

```
1: repeat
2:     take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
3:     sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
4:     compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}\left(\mathbf{s}'_j, \mu_{\theta'}(\mathbf{s}'_j)\right)$ using target network $Q_{\phi'}$ and $\mu_{\theta'}$
5:     $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)\left(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j\right)$
6:     $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(\mathbf{s}_j)\frac{dQ_\phi}{d\mathbf{a}}(\mathbf{s}_j, \mu(\mathbf{s}_j))$
7:     update $\phi'$ and $\theta'$ after every $N$ steps using $\phi$ and $\theta$ respectively
8: until Convergence
```

Note that though it has 'policy gradient' in its name, DDPG is essentially a Q-learning in disguise. And similar to DQN, DDPG is also a model-free off-policy algorithm.

2.5.2 Twin Delayed DDPG (TD3)

DDPG actually suffers from similar problems as in DQN, for example overestimation. Besides, the critices might be unstable, making convergence difficult. The critic also weirdly prefers some actions but not their neighbor actions. Twin Delayed DDPG introduced three solution to improve DDPG.

Solution 1: Clipped Double Q Learning

Solution 2: Delayed Policy Updates

Solution 3: Target Policy smoothing

2.6 Implementation Tips

In this section we would provide some practical tips for implementing Q-learning algorithm.

- Q-learning takes some care to stabilize, so you should test on easy, reliable tasks first, to make sure your implementation is correct.
- Large replay buffers help improve stability.
- It takes time to train, so be patient because the performance might look no better than random for a while
- Start with high exploration (epsilon) and gradually reduce.
- Double Q-learning helps a lot in practice. It's really simple and basically has no downsides.

- Bellman error gradients can be big, so instead you can use clip gradients or use Huber loss $L(x)$.

$$L(x) = \begin{cases} x^2/2 & \text{if } |x| \leq \delta \\ \delta|x| - \delta^2/2 & \text{otherwise} \end{cases}$$

- N-step returns also help a lot, but have some downsides.
- Schedule exploration (high to low) and learning rates (high to low), Adam optimizer can help too.
- Run multiple random seeds, it's very inconsistent between runs.

2.7 Mathy Stuff

Theorem 2.1. Consider a state s in which all the true optimal action values are equal at $Q_*(s,a) = V_*(s)$. Suppose that the estimation errors $Q_t(s,a) - Q_*(s,a)$ are independently distributed uniformly randomly in $[-1,1]$. Then,

$$\mathbb{E}\left[\max_a Q_t(s,a) - V_*(s)\right] = \frac{m-1}{m+1} \tag{2.7.1}$$

Theorem 2.2. Consider a state s in which all the true optimal action values are equal at $Q_*(s,a) = V_*(s)$ for some $V_*(s)$. Let Q_t be arbitrary value estimates that are on the whole unbiased in the sense that

$$\sum_a (Q_t(s,a) - V_*(s)) = 0,$$

but that are not all zero, such that

$$\frac{1}{m}\sum_a (Q_t(s,a) - V_*(s))^2 = C$$

for some $C > 0$, where $m \geq 2$ is the number of actions in s.

Under these conditions,

$$\max_a Q_t(s,a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}. \tag{2.7.2}$$

This lower bound is tight. Under the same conditions, the lower bound on the absolute error of the Double Q-learning estimate is zero.

2.8 Exercises

2.8.1 Linear Approximation

This problem is adapted from Assignment 2 of CS234 (2023 Spring).

Recall that the update rule for Q-learning is

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a)\right) \tag{2.8.1}$$

Due to the scale of Atari environments, we cannot reasonably learn and store a Q value for each state-action tuple. We will instead represent our Q values as a parametric function $Q_{\mathbf{w}}(s,a)$ where $\mathbf{w} \in \mathbb{R}^p$ are the parameters of the function (typically the weights and biases of a linear function or a neural network). In this approximation setting, the update rule becomes

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\mathbf{w}}(s', a') - Q_{\mathbf{w}}(s,a)\right)\nabla_{\mathbf{w}} Q_{\mathbf{w}}(s,a) \tag{2.8.2}$$

where (s, a, r, s') is a transition from the MDP. Let $Q_{\mathbf{w}}(s, a) = \mathbf{w}^\top \delta(s, a)$ be a linear approximation for the Q function, where $\mathbf{w} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ and $\delta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ with

$$[\delta(s, a)]_{s',a'} = \begin{cases} 1 & \text{if } s' = s, a' = a \\ 0 & \text{otherwise} \end{cases}$$

In other words, δ is a function which maps state-action pairs to one-hot encoded vectors. Compute $\nabla_{\mathbf{w}} Q_{\mathbf{w}}(s, a)$ and write the update rule for $\mathbf{w}$. Show that Eq. 2.8.1 and 2.8.2 are are equal when this linear approximation is used.

2.8.2 N-step Estimators

This problem is adapted from Assignment 2 of CS234 (2020 Winter).

Add N-step Estimators

Chapter 3 Policy-based Methods

3.1 Policy Gradient

3.1.1 Direct Policy Differentiation

During previous discussions, we learned that trying to find $\theta^\star$ would help us get more reward.

$$\theta^\star = \arg\max_\theta E_{\tau\sim p_\theta(\tau)}\left[\sum_t r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] \tag{3.1.1}$$

$$p_\theta\left(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T\right) = p\left(\mathbf{s}_1\right)\prod_{t=1}\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right) \tag{3.1.2}$$

We can rewrite the objective function as:

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}\left[\sum_t r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] \approx \frac{1}{N}\sum_i\sum_t r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right) \tag{3.1.3}$$

where $\sum_i$ means the sum over samples from π_θ. If we abbreviate $\sum_t r\left(\mathbf{s}_t, \mathbf{a}_t\right)$ as $r(\tau)$, then we have:

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau)r(\tau)d\tau \tag{3.1.4}$$

To find the optimal θ, we calculate the policy differentiation:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau)r(\tau)d\tau = \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau)r(\tau)d\tau = E_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)r(\tau)\right] \tag{3.1.5}$$

The derivation from the second term to the third term of the equation might be a bit confusing. Here we use a convenient identity:

$$\underline{p_\theta(\tau)\nabla_\theta \log p_\theta(\tau)} = p_\theta(\tau)\frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \underline{\nabla_\theta p_\theta(\tau)} \tag{3.1.6}$$

Also, note that $p_\theta(\tau)$ is given in (4.2) , and take *log* on both sides, then we have:

$$\log p_\theta(\tau) = \log p\left(\mathbf{s}_1\right) + \sum_{t=1}^{T}\log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) + \log p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right) \tag{3.1.7}$$

Substituting into equation (4.5), we obtain the *direct policy differentiation.*

Theorem 3.1 (Direct policy differentiation).

$$\begin{aligned}\nabla_\theta J(\theta) &= E_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right)\left(\sum_{t=1}^{T} r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right)\right] \\ &\approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\right)\left(\sum_{t=1}^{T} r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right)\right)\end{aligned}$$

Algorithm 15 REINFORCE algorithm

```
Require: arbitrarily initialized $\pi_\theta$
1: repeat
2:     sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)$ (run the policy)
3:     $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i \mid \mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
4:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
5: until convergence
```

Note. Markov property is not actually used in policy gradient. Therefore, you can use it in partially observed MDPs without modification.

3.1.2 Comparison to Maximum Likelihood

In policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right) \tag{3.1.8}$$

In maximum likelihood:

$$\nabla_\theta J_{\mathrm{ML}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\right) \tag{3.1.9}$$

Insight. Good stuff is made more likely; bad stuff is made less likely. What we just did in policy gradient simply formalize the notion of "through trial and error"!

3.1.3 The High Variance of Policy Gradient

One significant drawback of policy gradient is that the gradient is often noisy (i.e. the variance is high). Here are some methods of reducing variance.

Causality

Firstly, causality tells us that policy at time t' should not affect the reward at time t when $t \le t'$. The modified gradient function is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\left(\sum_{t'=t}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})\right) \tag{3.1.10}$$

The expression in brackets means "reward to go". And sometimes we also write it as $\hat{Q}_{i,t}^{\pi}$: estimate of expected reward if we take action $\mathbf{a}_{i,t}$ in state $\mathbf{s}_{i,t}$ $(\hat{Q}_{i,t}^{\pi} = \sum_{t'=1}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}))$

Baseline

The second approach is to introduce **baseline** to the reward part:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\nabla_\theta \log p_\theta(\tau)[r(\tau) - b], \quad b = \frac{1}{N}\sum_{i=1}^{N} r(\tau) \tag{3.1.11}$$

We are allowed to do that because what we really care about is *expectation* $(E[\nabla_\theta \log p_\theta(\tau) b])$, and subtracting a baseline is *unbiased in expectation*:

$$E[\nabla_\theta \log p_\theta(\tau) b] = \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) b d\tau = \int \nabla_\theta p_\theta(\tau) b d\tau = b\nabla_\theta \int p_\theta(\tau) d\tau = b\nabla_\theta 1 = 0 \tag{3.1.12}$$

Here we use the average reward as the baseline. It is not the best baseline (as to reducing variance), but it works pretty good. If you are not satisfied with it, we can try to find the best baseline:

$$\mathrm{Var}[x] = E\left[x^2\right] - E[x]^2 \tag{3.1.13}$$

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)(r(\tau) - b)\right] \tag{3.1.14}$$

$$\text{Var} = E_{\tau\sim p_\theta(\tau)}\left[(\nabla_\theta \log p_\theta(\tau)(r(\tau)-b))^2\right] - E_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)(r(\tau)-b)\right]^2 \tag{3.1.15}$$

$$\frac{d\,\text{Var}}{db} = \frac{d}{db}E\left[g(\tau)^2(r(\tau)-b)^2\right] = -2E\left[g(\tau)^2 r(\tau)\right] + 2bE\left[g(\tau)^2\right] = 0 \tag{3.1.16}$$

Solve this equation and you'll find the optimal baseline:

$$b = \frac{E\left[g(\tau)^2 r(\tau)\right]}{E\left[g(\tau)^2\right]} \tag{3.1.17}$$

Note that this is just expected reward, but weighted by gradient magnitudes!

3.1.4 More about reducing the variance

Before moving on, let's take a moment to review what it actually means to reduce the variance.

In a nutshell, in REINFORCE algorithm we are trying to optimize some function (i.e. the objective function $J(\theta)$) on a random variable (the return of some stochastic policy). What you need to do is sample a bunch of trajectories from the current policy, estimate the current expected value (remember, you are **sampling** trajectories so you don't know the expected value you can only **estimate** it), and update the parameters of the current policy in the direction that optimizes your estimate.

But you only have a finite amount of time to sample (at some point you have to update the parameters to make progress), so your estimate will be off of the true value and you will making updates based on that estimate. So let's say you are only going to sample 10 times, if the distribution of the values of that random variable has high variance, then sampling only 10 times will be a pretty poor estimate. If it has low variance, then 10 samples might be enough to approximate the value you want. Therefore we introduce the baseline to reduce the variance of the samples.

Another way of understanding *reducing the variance* is *reducing the aggressiveness of the updates.* Here is a concrete example: assuming we have a reward function that looks like

$$f(x) = \begin{cases} -x, & x < 0 \\ 0, & x \geq 0 \end{cases} \tag{3.1.18}$$

This can have a practical interpretation. For example, if you are in a fire, the only way to receive a reward of 0 is by escaping the fire in the positive x direction; otherwise, if you move in the negative x direction, you will get hurt and the reward will be negative.

In this scenario, if the baseline is not introduced, the gradient at 0 will be infinite, so one would not know if this policy would lead to some strange places. However, by introducing the baseline, a reasonable gradient will be obtained. This is why adding a baseline can also be interpreted as reducing the policy's aggressiveness.

3.2 Off-Policy Policy Gradients

If we revisit the derivation of policy differentiation, it's obvious that policy gradient is on-policy. The underlined part would be a trouble, because the neural networks change only a little bit with each gradient step, but you need to do the sampling all over again. Therefore, on-policy learning can be extremely inefficient!

$$\nabla_\theta J(\theta) = E_{\underline{\tau\sim p_\theta(\tau)}}\left[\nabla_\theta \log p_\theta(\tau) r(\tau)\right]$$

The question here is: what if we skip the sampling step? What if we don't have samples from $p_\theta(\tau)$ (we have samples from some $\bar{p}(\tau)$ instead)?

Before delving into this issue, let's talk about some maths first: *importance sampling*

Theorem 3.2 (Importance Sampling).

$$\begin{aligned} E_{x\sim p(x)}[f(x)] &= \int p(x)f(x)dx \\ &= \int \frac{q(x)}{q(x)}p(x)f(x)dx \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\ &= E_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right] \end{aligned}$$

Its mathematical essence lies in using a *known distribution* (typically an easy-to-sample distribution) to estimate the expectation of another distribution (usually a difficult-to-sample distribution). Back to off-policy learning,

here we can turn the original objective function $J(\theta) = E_{\tau\sim p_\theta(\tau)}[r(\tau)]$, into a new objective function using importance sampling:

$$J(\theta) = E_{\tau\sim\bar{p}(\tau)}\left[\frac{p_\theta(\tau)}{\bar{p}(\tau)}r(\tau)\right] \tag{3.2.1}$$

Here we have:

$$p_\theta(\tau) = p\left(\mathbf{s}_1\right)\prod_{t=1}^{T}\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right) \tag{3.2.2}$$

$$\frac{p_\theta(\tau)}{\bar{p}(\tau)} = \frac{p\left(\mathbf{s}_1\right)\prod_{t=1}^{T}\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)p\left(\mathbf{s}_t, \mathbf{a}_t\right)}{p\left(\mathbf{s}_1\right)\prod_{t=1}^{T}\bar{\pi}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)p\left(\mathbf{s}_t, \mathbf{a}_t\right)} = \frac{\prod_{t=1}^{T}\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}{\prod_{t=1}^{T}\bar{\pi}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)} \tag{3.2.3}$$

for some new parameters θ', similarly we have:

$$\frac{p_{\theta'}(\tau)}{p_\theta(\tau)} = \frac{\prod_{t=1}^{T}\pi_{\theta'}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}{\prod_{t=1}^{T}\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)} \tag{3.2.4}$$

therefore,

$$\nabla_{\theta'}J\left(\theta'\right) = E_{\tau\sim p_\theta(\tau)}\left[\frac{\nabla_{\theta'}p_{\theta'}(\tau)}{p_\theta(\tau)}r(\tau)\right] = E_{\tau\sim p_\theta(\tau)}\left[\frac{p_{\theta'}(\tau)}{p_\theta(\tau)}\nabla_{\theta'}\log p_{\theta'}(\tau)r(\tau)\right] \tag{3.2.5}$$

Now estimate locally, at $\theta = \theta'$, we have:

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)r(\tau)\right] \tag{3.2.6}$$

When $\theta \neq \theta'$, then:

$$\begin{aligned}\nabla_{\theta'}J\left(\theta'\right) &= E_{\tau\sim p_\theta(\tau)}\left[\frac{p_{\theta'}(\tau)}{p_\theta(\tau)}\nabla_{\theta'}\log\pi_{\theta'}(\tau)r(\tau)\right]\\ &= E_{\tau\sim p_\theta(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}{\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right)\left(\sum_{t=1}^{T}r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right)\right]\end{aligned} \tag{3.2.7}$$

Also, take causality into consideration, future actions don't affect current weight:

$$\nabla_{\theta'}J\left(\theta'\right) = E_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}\left(\mathbf{a}_{t'} \mid \mathbf{s}_{t'}\right)}{\pi_\theta\left(\mathbf{a}_{t'} \mid \mathbf{s}_{t'}\right)}\right)\left(\sum_{t'=t}^{T}r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right)\underline{\left(\prod_{t''\neq t}^{t}\frac{\pi_{\theta'}\left(\mathbf{a}_{t''} \mid \mathbf{s}_{t''}\right)}{\pi_\theta\left(\mathbf{a}_{t''} \mid \mathbf{s}_{t''}\right)}\right)}\right)\right] \tag{3.2.8}$$

The underlined part in 3.2.8 is distribution mismatch between different policies. If we ignore this part (later we'll see why this is reasonable), we'll get the following:

$$\nabla_{\theta'}J\left(\theta'\right) = E_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}\left(\mathbf{a}_{t'} \mid \mathbf{s}_{t'}\right)}{\pi_\theta\left(\mathbf{a}_{t'} \mid \mathbf{s}_{t'}\right)}\right)\left(\sum_{t'=t}^{T}r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right)\right)\right] \tag{3.2.9}$$

In the next section, we'll see that 3.2.9 is equivalent to policy iteration.

3.3 (Optional) Policy Gradient as Policy Iteration

Now we give further analysis of policy gradient as policy iteration. The trick here is to express the expected value under policy π_θ in terms of the expected value with repect to the trajectory τ under policy $\pi_{\theta'}$:

$$
\begin{aligned}
J\left(\theta^{\prime}\right)-J(\theta) &= J\left(\theta^{\prime}\right)-E_{\mathbf{s}_0 \sim p\left(\mathbf{s}_0\right)}\left[V^{\pi_\theta}\left(\mathbf{s}_0\right)\right] \\
&= J\left(\theta^{\prime}\right)-E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[V^{\pi_\theta}\left(\mathbf{s}_0\right)\right] \\
&= J\left(\theta^{\prime}\right)-E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t V^{\pi_\theta}\left(\mathbf{s}_t\right)-\sum_{t=1}^{\infty} \gamma^t V^{\pi_\theta}\left(\mathbf{s}_t\right)\right] \\
&= J\left(\theta^{\prime}\right)+E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t\left(\gamma V^{\pi_\theta}\left(\mathbf{s}_{t+1}\right)-V^{\pi_\theta}\left(\mathbf{s}_t\right)\right)\right] \\
&= E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]+E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t\left(\gamma V^{\pi_\theta}\left(\mathbf{s}_{t+1}\right)-V^{\pi_\theta}\left(\mathbf{s}_t\right)\right)\right] \\
&= E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t\left(r\left(\mathbf{s}_t, \mathbf{a}_t\right)+\gamma V^{\pi_\theta}\left(\mathbf{s}_{t+1}\right)-V^{\pi_\theta}\left(\mathbf{s}_t\right)\right)\right] \\
&= E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]
\end{aligned}
\tag{3.3.1}
$$

Writing out the expectation with respect to the trajectory τ explicitly:

$$
\begin{aligned}
E_{\tau \sim p_{\theta^{\prime}}(\tau)}\left[\sum_t \gamma^t A^{\pi_\theta}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] &= \sum_t E_{\mathbf{s}_t \sim p_{\theta^{\prime}}\left(\mathbf{s}_t\right)}\left[E_{\mathbf{a}_t \sim \pi_{\theta^{\prime}}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}\left[\gamma^t A^{\pi_\theta}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]\right] \\
&= \sum_t E_{\mathbf{s}_t \sim p_{\theta^{\prime}}\left(\mathbf{s}_t\right)}\left[E_{\mathbf{a}_t \sim \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}\left[\frac{\pi_{\theta^{\prime}}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}{\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)} \gamma^t A^{\pi_\theta}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]\right]
\end{aligned}
\tag{3.3.2}
$$

where the second equation is obtained by using the importance sampling formula 3.2. Now the remaining question is: can we use p_θ' instead of p_θ in the expectation? The answer is yes, because we can actually bound the distribution gap between p_θ and p_θ' if the two policies π_θ and $\pi_{\theta'}$ are close enough.

3.4 (Optional) Bounding the Distribution Gap

3.5 (Optional) Advanced Policy Gradient Methods

In this section, we'll introduce the ideas of two advanced policy gradient methods. Feel free to skip this section if you are not interested in mathy details.

3.5.1 Trust Region Policy Optimization (TRPO)

Add TRPO algorithm

3.5.2 Proximal Policy Optimization (PPO)

Add PPO algorithm

3.6 Implementation Tips

There are some tips in implementing policy gradient:

1. Remember that the gradient has high variance. This isn't the same as supervised learning, actually the gradients would be really noisy.
2. Consider using much larger batches.
3. Tweaking learning rates is very hard.(We'll learn about policy gradient-specific learning rate adjustment methods later)

Chapter 4 Actor-Critic Methods

4.1 Introducing the Actor-Critic Methods

4.1.1 Recap

We mention Q function and V function here again as a recap.

$$Q^{\pi}\left(\mathbf{s}_t, \mathbf{a}_t\right)=\sum_{t'=t}^{T} E_{\pi_\theta}\left[r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right) \mid \mathbf{s}_t, \mathbf{a}_t\right]: \text{ total reward from taking } \mathbf{a}_t \text{ in } \mathbf{s}_t$$

$$V^{\pi}\left(\mathbf{s}_t\right)=E_{\mathbf{a}_t \sim \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}\left[Q^{\pi}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]: \text{ total reward from } \mathbf{s}_t$$

$$A^{\pi}\left(\mathbf{s}_t, \mathbf{a}_t\right)=Q^{\pi}\left(\mathbf{s}_t, \mathbf{a}_t\right)-V^{\pi}\left(\mathbf{s}_t\right) \ : \text{how much better } \mathbf{a}_t \text{ is than average}$$

Policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right) Q\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right) \tag{4.1.1}$$

Adding a baseline:

$$\begin{aligned}\nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(Q\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right)-V\left(\mathbf{s}_{i,t}\right)\right) \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right) A^{\pi}\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right)\end{aligned} \tag{4.1.2}$$

4.1.2 Policy Evaluation

Let's go back to three basics steps of RL: **generate samples** (i.e. run the policy), **fit a model to estimate return**, and **improve the policy**. In policy gradient, we figure out how to calculate $\nabla_\theta J(\theta)$, which tells us how to improve the policy. But we still need to fit the model, which quantatively tell us how good the policy is. We have three possible quantities, Q^π, V^π or A^π , so the question is: *what* should we fit to *what*? Since the current reward of taking $\mathbf{a}_t$ at $\mathbf{s}_t$ is fixed, so we can rewrite $Q^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)$ and $A^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)$:

$$\begin{aligned}Q^{\pi}\left(\mathbf{s}_t, \mathbf{a}_t\right) &= r\left(\mathbf{s}_t, \mathbf{a}_t\right)+\sum_{t'=t+1}^{T} E_{\pi_\theta}\left[r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right) \mid \mathbf{s}_t, \mathbf{a}_t\right] \\ &\approx r\left(\mathbf{s}_t, \mathbf{a}_t\right)+V^{\pi}\left(\mathbf{s}_{t+1}\right)\end{aligned} \tag{4.1.3}$$

$$A^{\pi}\left(\mathbf{s}_t, \mathbf{a}_t\right) \approx r\left(\mathbf{s}_t, \mathbf{a}_t\right)+V^{\pi}\left(\mathbf{s}_{t+1}\right)-V^{\pi}\left(\mathbf{s}_t\right) \tag{4.1.4}$$

Therefore V^π would be a nice choice, since Q^π and A^π depend on both actions and states, while V^π depends on solely states. Of course this is not the only option in actor-critic algorithm. We'll talk about that later.

Actually calculating the value function is essentially calculating the expectation of reward under a given policy. That's why the process of fitting value fuction is also called policy evaluation.

Theorem 4.1 (Monte Carlo policy evaluation). Monte Carlo policy evaluation estimates the value of state-action pairs based on random sampling of experiences. It involves averaging the returns observed from multiple episodes to approximate the true value function for a given policy.

$$V^{\pi}\left(\mathbf{s}_t\right) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right)$$

Note that in traditional Monte Carlo policy evaluation you need to generate samples from the same initial state, which means constantly resetting the simulator. Another way of doing that is training a neural network to estimate value function:

$$\text{training data: } \left\{\left(\mathbf{s}_{i,t}, \sum_{t'=t}^{T} r\left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right)\right)\right\}$$

$$\text{supervised regression: } \mathcal{L}(\phi) = \frac{1}{2}\sum_i \left\|\hat{V}_\phi^\pi\left(\mathbf{s}_i\right) - y_i\right\|^2$$

In fact, the value function is very intuitive. For example, when training an agent to play a chess-like game, if we set the probability of winning the game to 1 and the probability of losing to 0, the value function typically indicates the probability of eventually winning the game in the current state.

4.1.3 From Evaluation to Actor Critic

Now that we have learned policy evaluation and policy gradients, now we are able to put the two pieces together. And that makes an actor-critic algorithm:

Algorithm 16 Batch actor-critic algorithm

```
1: repeat
2:     sample $\left\{\tau^i\right\}$ from $\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)$ (run the policy)
3:     fit $\hat{V}_\phi^\pi(\mathbf{s})$ to sampled reward sums
4:     evaluate $\hat{A}^\pi\left(\mathbf{s}_i, \mathbf{a}_i\right) = r\left(\mathbf{s}_i, \mathbf{a}_i\right) + \hat{V}_\phi^\pi\left(\mathbf{s}_i'\right) - \hat{V}_\phi^\pi\left(\mathbf{s}_i\right)$
5:     $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta\left(\mathbf{a}_i \mid \mathbf{s}_i\right) \hat{A}^\pi\left(\mathbf{s}_i, \mathbf{a}_i\right)$
6:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
7: until convergence
```

Note that while fitting the value function, we are fitting the sum of reward in a given episode length T. What if T is ∞ ? In many cases $\hat{V}_\phi^\pi$ can get infinitely large. A simple trick here is to all discount factor γ, which would tell the policy that its better to get rewards sooner than later.

Without discount factors, the target function and loss function of the neural network is:

$$\begin{aligned} & y_{i,t} \approx r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right) + \hat{V}_\phi^\pi\left(\mathbf{s}_{i,t+1}\right) \\ & \mathcal{L}(\phi) = \tfrac{1}{2}\sum_i \left\|\hat{V}_\phi^\pi\left(\mathbf{s}_i\right) - y_i\right\|^2 \end{aligned} \tag{4.1.5}$$

Now adding the discount factors, we'll have:

$$y_{i,t} \approx r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right) + \gamma \hat{V}_\phi^\pi\left(\mathbf{s}_{i,t+1}\right), \quad \gamma \in [0,1] (0.99 \text{ works well }) \tag{4.1.6}$$

Insight. One way of understanding discount factors is that we change the MDP by adding an extra state of death. Once we enter the death state we never leave, and the reward is zero. In each state the agent has the probability of $1-\gamma$ of falling into the death state.

4.1.4 Aside: the Discount Factor

Then how do we introduce discount factors to (Monte Carlo) policy gradients? Actually there seems to be two ways of doing this.

$$\begin{aligned} \text{option 1:} \quad & \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(\sum_{t'=t}^{T} \gamma^{t'-t} r\left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right)\right) \\ \text{option 2:} \quad & \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\right)\left(\sum_{t=1}^{T} \gamma^{t-1} r\left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right)\right) \end{aligned} \tag{4.1.7}$$

The only difference is whether we introduce the discount factors before or after we use the causality rules.

To showcase the differences more effectively, we can rewrite the expression of option 2 in the form of option 1.

$$
\begin{aligned}
\nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right) \right) \left(\sum_{t=1}^{T} \gamma^{t-1} r \left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right) \right) \\
&= \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right) \left(\sum_{t'=t}^{T} \gamma^{t'} \left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right) \right) \\
&= \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \gamma^{t-1} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r \left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right) \right)
\end{aligned}
\tag{4.1.8}
$$

Note that γ^{t-1} comes before $\nabla_\theta \log \pi_\theta$. Then it becomes much clearer. Option 2 implies that we not only care less about the rewards in the future, we also care less about the decisions in the future. As a result you are discounting the gradients. In other word, making right decision in the first time step is more important than making right decision in future time steps. If you are solving a real discount problem, this is exactly what we are trying to do. Because remember the modified MDP settings, later steps don't matter when you are dead. **But**, in reality, this is often not quite what we want. The version that we actually use is **option 1**.

Take some time to review why we introduced the discount factor. For tasks with particularly long time episodes, we introduced the discount factor to artificially reduce the rewards for future actions in order to compute the value function. However, in practical tasks, we do not want to do this. For example, if we want a robot to run steadily forward, we actually want it to keep running. We want to learn a policy that can do the right thing for a long time, rather than just at nearby timesteps. Therefore, option 1 becomes more reasonable.

Algorithm 17 Batch actor-critic algorithm (with discount)

```
1: repeat
2:     sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)$ (run the policy)
3:     fit $\hat{V}_\phi^\pi(\mathbf{s})$ to sampled reward sums
4:     evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}_\phi^\pi(\mathbf{s}_i') - \hat{V}_\phi^\pi(\mathbf{s}_i)$
5:     $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i \mid \mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
6:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
7: until convergence
```

The only difference that in step 3 we introduce the discount factor. In previous discussions we have been using policy gradients in episodic batch mode settings. If we modify a little bit, we'll get the online actor-critic algorithm:

Algorithm 18 Online actor-critic algorithm (with discount)

```
1: repeat
2:     take action $\mathbf{a} \sim \pi_\theta(\mathbf{a} \mid \mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
3:     update $\hat{V}_\phi^\pi(\mathbf{s})$ using target $r + \gamma \hat{V}_\phi^\pi(\mathbf{s}')$
4:     evaluate $\hat{A}^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \hat{V}_\phi^\pi(\mathbf{s}') - \hat{V}_\phi^\pi(\mathbf{s})$
5:     $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a} \mid \mathbf{s}) \hat{A}^\pi(\mathbf{s}, \mathbf{a})$
6:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
7: until convergence
```

Insight. In actor-critic algorithm, **actor** is the policy, and **critic** is the value function. It can be viewed as a improved version of policy gradient, with reduced variance.

4.2 Further Analysis

4.2.1 Critics as Baselines

Let's make a comparison here between the actor-critic algorithm and the policy gradient algorithm:

$$\begin{aligned}&\text{Actor-critic:}\quad \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right)+\gamma \hat{V}_\phi^\pi\left(\mathbf{s}_{i,t+1}\right)-\hat{V}_\phi^\pi\left(\mathbf{s}_{i,t}\right)\right)\\&\text{Policy gradient:}\quad \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(\left(\sum_{t'=t}^{T}\gamma^{t'-t} r\left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right)\right)-b\right)\end{aligned} \tag{4.2.1}$$

In actor-critic, we have lower variance since we have the critic, but it is not unbiased if the critic is not perfect. While in policy gradient, it is unbiased but the variance would be high due to the single-sample estimate.

So intuitively a nature step we can take is to use $\hat{V}_\phi^\pi$ while still keep the estimater unbiased:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(\left(\sum_{t'=t}^{T}\gamma^{t'-t} r\left(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}\right)\right)-\hat{V}_\phi^\pi\left(\mathbf{s}_{i,t}\right)\right) \tag{4.2.2}$$

Up to now we are all using the state-dependent functions as baselines. So can we use functions that involves both action and state as baselines? Here we come to control variates, which means baselines that are action-dependent. Here we are going to talk about that.

Baselines that use both actions and states are also called *control variates.* The true advantage function is:

$$\begin{aligned}A^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right) &= Q^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)-V^\pi\left(\mathbf{s}_t\right)\\&=\sum_{t'=t}^{T} E_{\pi_\theta}\left[r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right) \mid \mathbf{s}_t, \mathbf{a}_t\right]-E_{\mathbf{a}_t \sim \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)}\left[Q^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]\end{aligned} \tag{4.2.3}$$

While the approximate advantage function we use in policy gradient is:

$$\hat{A}^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)=\sum_{t'=t}^{\infty}\gamma^{t'-t} r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right)-V_\phi^\pi\left(\mathbf{s}_t\right) \tag{4.2.4}$$

This is unbiased and has a lower variance (but still a higer variance than the actor-critic because of the single sample estimate). But we can make the variance even lower if we substract the Q value.

$$\hat{A}^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)=\sum_{t'=t}^{\infty}\gamma^{t'-t} r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right)-Q_\phi^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right) \tag{4.2.5}$$

This version has a nice property that it goes to zero in expectation if critic is correct. Unfortunately, it does not work if you simply plug it into the policy gradient, since there is an error term you have to compensate for. Now taking that error term into accounts, we have:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(\hat{Q}_{i,t}-Q_\phi^\pi\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right)\right)+\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta E_{\mathbf{a}\sim\pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_{i,t}\right)}\left[Q_\phi^\pi\left(\mathbf{s}_{i,t}, \mathbf{a}_t\right)\right] \tag{4.2.6}$$

The second term represents the gradient of expectation value under the policy of the baseline. Note that this equation is valid even when the baseline depends merely on the state functions, but in that case the second term equals to zero. This kind of trick can provide for a very low variance policy gradient.

We can also take a look at the two different versions of advantage function:

$$\begin{aligned}&\hat{A}_{\mathrm{C}}^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)=r\left(\mathbf{s}_t, \mathbf{a}_t\right)+\gamma \hat{V}_\phi^\pi\left(\mathbf{s}_{t+1}\right)-\hat{V}_\phi^\pi\left(\mathbf{s}_t\right) && \text{+ lower variance}\\& && \text{- higher bias if value is wrong (it always is)}\\&\hat{A}_{\mathrm{MC}}^\pi\left(\mathbf{s}_t, \mathbf{a}_t\right)=\textstyle\sum_{t'=t}^{\infty}\gamma^{t'-t} r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right)-\hat{V}_\phi^\pi\left(\mathbf{s}_t\right) && \text{+ no bias}\\& && \text{- higher variance (because single-sample estimate)}\end{aligned} \tag{4.2.7}$$

So can we find something in the middle, combine these two, to control bias/variance tradeoff? The trick here is that instead of unsing an infinite time horizon, you cut it off before the variance goes too high, given that the variance is generally small in the near future and goes higher in the far future. Here is what an n-step return estimater does:

$$\hat{A}_n^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) - \hat{V}_\phi^\pi(\mathbf{s}_t) + \gamma^n \hat{V}_\phi^\pi(\mathbf{s}_{t+n}) \tag{4.2.8}$$

Generally the larger n you choose, the bias goes smaller and the variance goes larger. The first and the third term contributes to variance, and the second term contributes to bias. The n here we use in n-step return estimater is fixed. Actually we do not have to choose a single n. Instead, we can take all possible n at one time, which is the generalized advantage estimation.

$$\hat{A}_{\mathrm{GAE}}^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{n=1}^{\infty} w_n \hat{A}_n^\pi(\mathbf{s}_t, \mathbf{a}_t) \tag{4.2.9}$$

The generalized advantage estimation is actually a weighted combination of all n-step returns. In terms of determining the weights, we mostly prefer cutting earlier because it brings less variance. Therefore, a descent choice is to use exponential falloff ($w_n \propto \lambda^{n-1}$).

$$\hat{A}_{\mathrm{GAE}}^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'} \quad \delta_{t'} = r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) + \gamma \hat{V}_\phi^\pi(\mathbf{s}_{t'+1}) - \hat{V}_\phi^\pi(\mathbf{s}_{t'}) \tag{4.2.10}$$

4.3 (Optional) Advanced Actor-Critic Methods

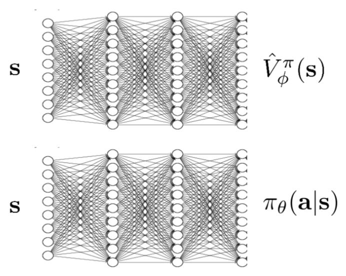
4.3.1 Soft Actor-Critic (SAC)

Add SAC

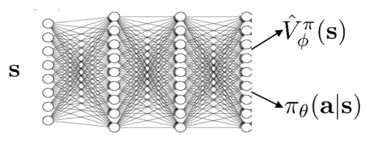
4.4 Implementations Tips

4.4.1 Architecture Design

There are two options of the neural network architecture design: two network design and shared network design.


s
$\hat{V}_\phi^\pi(\mathbf{s})$
s
$\pi_\theta(\mathbf{a}|\mathbf{s})$

(a) two network design


s
$\hat{V}_\phi^\pi(\mathbf{s})$
$\pi_\theta(\mathbf{a}|\mathbf{s})$

(b) shared network design

The two-network design is simple and stable, but lacks efficient feature sharing between the actor and critic. In contrast, the shared network design allows for potential feature sharing efficiency, but it has more hyperparameters and can be more complex and less stable during training.

4.4.2 Parallel Actor-Critic

In a typical online actor-critic algorithm, step2 and step4 work best with a batch (e.g., parallel workers). There are also two types of parallel actor-critic design: **sychronized parallel actor-critic** and **asynchronous parallel actor-critic**.

Sychronized parallel actor-critic: multiple agents learn simultaneously, but they must synchronously update parameters at each step to ensure consistent behavior across all agents. However different agents would use different random seeds so their actions would be a little bit different.

Asynchronous parallel actor-critic: different agents independently update parameters during learning without the need for synchronous updates, leading to improved learning efficiency. However, each agent independently update parameters at their own pace, therefore the parameters update may be inconsistent.

4.4.3 Off-Policy Actor-Critic Algorithm

The problem of sample effeciency gets us thinking about another problem: can we remove the on-policy assumption entirely? A primitive way is to use a replay buffer where we store the transitions we saw in prior time steps. Based on this we have an immature off policy actor-critic algorithm:

Algorithm 19 (Fake) Off-policy actor-critic algorithm

```
1: repeat
2:     take action $\mathbf{a} \sim \pi_\theta(\mathbf{a} \mid \mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$, store in $\mathcal{R}$
3:     sample a batch $\{\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_i'\}$ from buffer $\mathcal{R}$
4:     update $\hat{V}_\phi^\pi(\mathbf{s})$ using target $\underline{y_i = r_i + \gamma \hat{V}_\phi^\pi(\mathbf{s}_i')}$ for each $\mathbf{s_i}$, $\mathcal{L}(\phi) = \frac{1}{N}\sum_i \left\| \hat{V}_\phi^\pi(\mathbf{s}_i) - y_i \right\|^2$
5:     evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}_\phi^\pi(\mathbf{s}_i') - \hat{V}_\phi^\pi(\mathbf{s}_i)$
6:     $\nabla_\theta J(\theta) \approx \underline{\sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i \mid \mathbf{s}_i)} \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
7:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
8: until convergence
```

There are two fallacies lying in this immature algorithm. The first problem comes from the traget value. When you are loading transitions from the replay buffer, you are actually loading actions from the older version of policy, not the latest one. Therefore $\mathbf{s}'$ comes from the older actions, which is not we actually want.

The second issue comes from the same reason. Because the action $\mathbf{a_i}$ does not come from the latest policy, you cannot calculate policy gradient.

To fix the first problem, the method we take here is to substitute V-function with Q-function. Note the difference between these two functions:

$$\begin{aligned} V^\pi(\mathrm{s}_t) &= \textstyle\sum_{t'=t}^T E_{\pi_\theta}\left[r(\mathrm{s}_{t'}, \mathbf{a}_{t'}) \mid \mathbf{s}_t\right] \\ Q^\pi(\mathbf{s}_t, \mathbf{a}_t) &= \textstyle\sum_{t'=t}^T E_{\pi_\theta}\left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \mid \mathbf{s}_t, \mathbf{a}_t\right] \end{aligned} \tag{4.4.1}$$

Q-function cares about the total reward from taking $\mathbf{a}_t$ in $\mathbf{s}_t$, and then following the policy π_θ. However we do not restrict that action $\mathbf{a}_t$ must come from π_θ. It is a valid function for any action.

So here in the thrid step in Algorithm 6, instead of using $\hat{V}_\phi^\pi(\mathbf{s})$, we update $\hat{Q}_\phi^\pi(\mathbf{s})$ using target y_i. This time we have:

$$\begin{aligned} y_i &= r_i + \gamma \hat{V}_\phi^\pi(\mathbf{s}_i') \\ &= r_i + \gamma \hat{Q}_\phi^\pi(\mathbf{s}_i', \mathbf{a}_i') \end{aligned} \tag{4.4.2}$$

The trick here is to use:

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T E_{\pi_\theta}\left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \mid \mathbf{s}_t\right] = E_{\mathbf{a} \sim \pi(\mathbf{a}_t \mid \mathbf{s}_t)}\left[Q(\mathbf{s}_t, \mathbf{a}_t)\right] \tag{4.4.3}$$

Note that action $\mathbf{a}_i'$ in (5.10) does not come from the replay buffer $\mathcal{R}$. Instead, it is the action an agent would have taken following policy π_θ under state $\mathbf{s}_i'$. This procedure does not require state $\mathbf{s}_i'$ to actually happens in a simulator, because all you need is to plug state $\mathbf{s}_i'$ into the neural network and get the output action.

We can use a similar approach to resolve the policy gradient issue. Instead of using $\mathbf{a_i}$ which comes from the replay buffer, we use $\mathbf{a}_i^\pi \sim \pi_\theta(\mathbf{a} \mid \mathbf{s}_i)$ that comes from the latest policy.

So now we have step 5 in Algorithm 6 as:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i^\pi \mid \mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i^\pi) \tag{4.4.4}$$

One more thing to say. In practice, we don't actually use the advantage function in policy gradient. We simply use the Q-function here:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta \left(\mathbf{a}_i^\pi \mid \mathbf{s}_i\right) \hat{Q}^\pi \left(\mathbf{s}_i, \mathbf{a}_i^\pi\right) \tag{4.4.5}$$

This would increase the variance because it does not involve a baseline. However, a high variance here is OK since we don't need to interact with the simulators to sample actions. So it's easy to lower the variance by generating more samples of actions $\mathbf{a}_i^\pi$, without generating states $\mathbf{s}_i$

So we have the fixed version of off-policy actor-critic algorithm here:

Algorithm 20 Off-policy actor-critic algorithm

```
1: repeat
2:     take action $\mathbf{a} \sim \pi_\theta(\mathbf{a} \mid \mathbf{s})$, get $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$, store in $\mathcal{R}$
3:     sample a batch $\{\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_i'\}$ from buffer $\mathcal{R}$
4:     update $\hat{Q}_\phi^\pi(\mathbf{s})$ using target $y_i = r_i + \gamma \hat{Q}_\phi^\pi(\mathbf{s}_i', \mathbf{a}_i')$ for each $\mathbf{s_i}$, $\mathcal{L}(\phi) = \frac{1}{N} \sum_i \left\| \hat{Q}_\phi^\pi(\mathbf{s}_i) - y_i \right\|^2$
5:     evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}_\phi^\pi(\mathbf{s}_i') - \hat{V}_\phi^\pi(\mathbf{s}_i)$
6:     $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i^\pi \mid \mathbf{s}_i) \hat{Q}^\pi(\mathbf{s}_i, \mathbf{a}_i^\pi)$
7:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
8: until convergence
```