CS285 (Deep Reinforcement Learning) Notes [Fall 2020]

Patrick Yin

Updated July 20, 2021

Contents

1 Note **3**
1.1 Note . . . 3

2 Imitation Learning **4**
2.1 Imitation Learning . . . 4
2.2 Goal-Conditioned Behavorial Cloning . . . 5

3 Reinforcement Learning **6**
3.1 Definitions . . . 6
3.2 RL Algorithm Anatomy . . . 7
3.3 Value Functions . . . 7
3.4 Types of Algorithms . . . 8
3.5 Tradeoffs Between Algorithms . . . 9

4 Policy Gradient **11**
4.1 Direct Policy Differentiation . . . 11
4.2 Understanding Policy Gradients . . . 12
4.3 Reducing Variance . . . 13
4.4 Off-Policy Policy Gradients . . . 15
4.5 Covariant/Natural Policy Gradient . . . 16

5 Actor-Critic Algorithms **17**
5.1 Policy Evaluation . . . 17
5.2 Actor-Critic . . . 19
5.3 Actor-Critic Design Decisions . . . 20
5.4 Critics as Baselines . . . 21

6 Value Function Methods **23**
6.1 Policy Iteration . . . 23
6.2 Fitted Value Iteration & Q-Iteration . . . 24
6.3 Q-Learning . . . 25
6.4 Value Functions in Theory . . . 25

7 Deep RL with Q-Functions **27**
7.1 Replay Buffers 27
7.2 Target Networks 27
7.3 Improving Q-Learning 28
7.4 Implementation Tips 30

Chapter 1

Note

1.1 Note

These course notes are my notes from CS 285 : Deep Reinforcement Learning taught by Professor Sergey Levine. The course is linked here. I did not formally take this course, but self-studied the material via the lectures posted published on YouTube. These notes are currently in progress.

Chapter 2

Imitation Learning

2.1 Imitation Learning

Imitation learning has the issue of distributional drift. We can solve this in two ways.

The first way is just to mimic the expert so accurately so that it doesn't drift. Failing to fit the expert accurately could be due to non-markovian and/or multimodal behavior. For the former problem, we can consider history with an RNN. For the later problem, we can output a MoG, use latent variable models, or use autoregressive discretization.

The second way is to generate more data so that the training distribution matches the policy trajectory distribution. DAgger, Dataset Aggregation, does this by collecting data from $p_{\pi_\theta}(o_t)$ instead of $p_{data}(o_t)$.

Algorithm 1 DAgger

1: train $\pi_\theta(a_t|o_t)$ from human data $\mathcal{D} = \{o_1, a_1, ..., o_N, a_N\}$
2: run $\pi_\theta(a_t|o_t)$ to get dataset $\mathcal{D}_\pi = \{o_1, ..., o_M\}$
3: ask human to label $\mathcal{D}_\pi$ with actions a_t
4: aggregate: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_\pi$

Let us prove theoretically why the error with DAgger is an order of magnitude lower than that of traditional behaviorial cloning. Define

$$c(s,a) = \begin{cases} 0 & a = \pi^*(s) \\ 1 & otherwise \end{cases}$$

Assuming that $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ for all s in $\mathcal{D}_{train}$,

$$\mathbb{E}[\sum_t c(s_t, a_t)] \leq \epsilon T + (1-\epsilon)\epsilon T + (1-\epsilon)^2 \epsilon T + ... = \mathcal{O}(\epsilon T^2)$$

More generally, let us instead assume that $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ for all s $\sim p_{train}(s)$. With DAgger, since $p_{train}(s) \to p_\theta(s)$,

$$\mathbb{E}[\sum_t c(s_t, a_t)] \leq \epsilon T$$

With behavorial cloning, we can compute the probability distribution of the a state under the current policy in terms of the probability distribution of training data as such:

$$p_\theta(s_t) = (1-\epsilon)^t p_{train}(s_t) + (1-(1-\epsilon)^t) p_{mistake}(s_t)$$

so,

$$|p_\theta(s_t) - p_{train}(s_t)| = (1-(1-\epsilon)^t)|p_{mistake}(s_t) - p_{train}(s_t)| \leq 2(1-(1-\epsilon)^t) \leq 2\epsilon t$$

Then, we know that

$$\begin{aligned}\sum_t \mathbb{E}_{p_\theta(s_t)}[c_t] = \sum_t \sum_{s_t} p_\theta(t) c_t(s_t) &\leq \sum_t \sum_{s_t} p_{train}(s_t) c_t(s_t) + |p_\theta(s_t) - p_{train}(s_t)| c_{max} \\ &\leq \sum_t \epsilon + 2\epsilon t = \mathcal{O}(\epsilon T^2)\end{aligned}$$

It turns out we can get the same bounds with the looser assumption that $\mathbb{E}_{p_{train}(s)}[\pi_\theta(a \neq \pi^*(s)|s)] \leq \epsilon$, but we won't prove this here.

2.2 Goal-Conditioned Behavorial Cloning

For a policy to reach any goal p, which may not be in the training dataset, we can condition our policy on p. In other words, we collect data and train a goal conditioned policy with the last state being the goal state. In "Learning to Reach Goals via Iterated Supervised Learning", the authors start with a random policy, collect data with random goals, treat this data as demonstrations for the goals that were reached, used this to improve the policy, and repeated.

Chapter 3

Reinforcement Learning

3.1 Definitions

MDP is defined by $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$, and POMDP is defined by $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{E}, r\}$ where $\mathcal{E}$ gives the emission probability $p(o_t|s_t)$. Let us define

$$p_\theta(\tau) := p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

Then the optimal RL policy is

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

For convenient, let us rewrite this expression in terms of state-action marginals:

$$\theta^* = \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)}[r(s_t, a_t)]$$

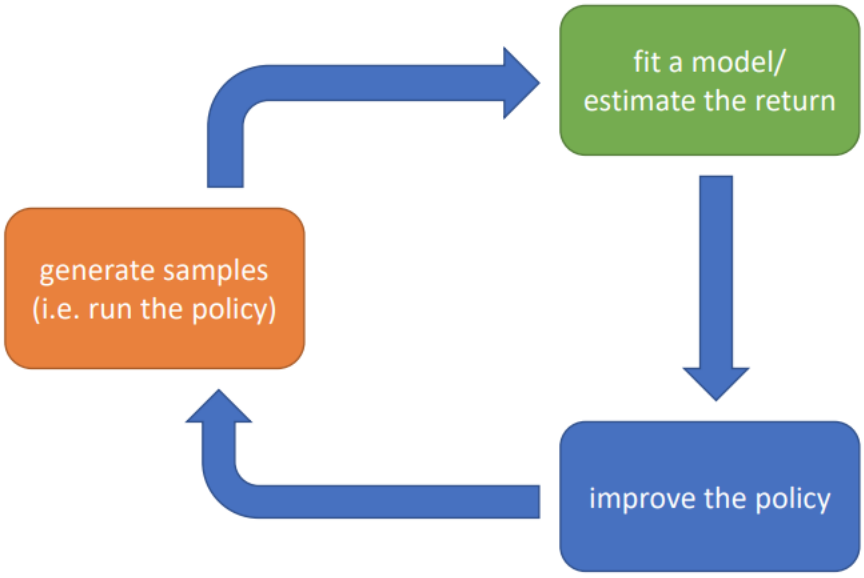
where $p_\theta(s_t, a_t)$ is our state-action marginal. Our "new" MC now has the transition probability $p((s_{t+1}, a_{t+1})|(s_t, a_t)) = p(s_{t+1}|s_t, a_t)\pi_\theta(a_{t+1}|s_{t+1})$.

In the infinite horizon case (i.e. when $T = \infty$), $p(s_t, a_t)$ converges to a stationary distribution if it is ergodic. If so, then the stationary distribution μ satisfies $\mu = \mathcal{T}\mu$, so $\mu := p_\theta(s, a)$ is the eigenvector of $\mathcal{T}$ with eigenvalue 1. If rewards are positive, the reward sum can go to infinity, so we also divide by T. Thus, in the infinite horizon case, our optimal RL policy is

$$\theta^* = \arg\max_\theta \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)}[r(s_t, a_t)] \to \mathbb{E}_{(s,a) \sim p_\theta(s,a)}[r(s, a)]$$

Also note that we care about expectations over the reward accrued because it is smooth in θ.

3.2 RL Algorithm Anatomy



3.3 Value Functions

Remember that our objective function is

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

This can rewritten as

$$\mathbb{E}_{s_1 \sim p(s_1)} \left[\mathbb{E}_{a_1 \sim \pi(a_1|s_1)} \left[r(s_1, a_1) + \mathbb{E}_{s_1 \sim p(s_2|s_1, a_1)} \left[\mathbb{E}_{a_2 \sim \pi(a_2|s_1)} \left[r(s_2, a_2) + ... | s_2 \right] | s_1, a_1 \right] | s_1 \right] \right]$$

If we define

$$Q(s_1, a_1) = r(s_1, a_1) + \mathbb{E}_{s_1 \sim p(s_2|s_1, a_1)} \left[\mathbb{E}_{a_2 \sim \pi(a_2|s_1)} \left[r(s_2, a_2) + ... | s_2 \right] | s_1, a_1 \right]$$

then

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right] = \mathbb{E}_{s_1 \sim p(s_1)} \left[\mathbb{E}_{a_1 \sim \pi(a_1|s_1)} \left[Q(s_1, a_1) | s_1 \right] \right]$$

This "Q-function" is very useful as if we know it, we can easily improve the policy. So, we will define a Q-function as such

$$Q^\pi(s_t, a_t) := \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta} \left[r(s_{t'}, a_{t'}) | s_t, a_t \right]$$

and a value function as such

$$V^{\pi}(s_t) := \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\left[r(s_{t'}, a_{t'})|s_t\right] = \mathbb{E}_{a_t \sim \pi(a_t|s_t)}\left[Q^{\pi}(s_t, a_t)\right]$$

Note that $\mathbb{E}_{s_1 \sim p(s_1)}\left[V^{\pi}(s_1)\right]$ is the RL objective.
Using Q-functions and value functions, we have two high-level ideas of how we can improve our policy π over time:

1. Set $\pi'(a|s) = 1$ if $a = \arg\max_a Q^{\pi}(s, a)$. This is at least as good as π.
2. If $Q^{\pi}(s, a) > V^{\pi}(s)$, then a is better than average. Modify $\pi(a|s)$ to increase probability of a if $Q^{\pi}(s, a) > V^{\pi}(s)$.

3.4 Types of Algorithms

RL Algorithm Types			
RL Type	Description	Fit model and/or estimate return	Improve policy
Policy gradient	directly differentiate the above objective	evaluate returns $R_\tau = \sum_t r(s_t, a_t)$	$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}[\sum_t r(s_t, a_t)]$
Value-based	estimate value function or Q-function of the optimal policy (no explicit policy)	fit $V(s)$ or $Q(s, a)$	set $\pi(s) = \arg\max_a Q(s, a)$
Actor-critic	estimate value function or Q-function of the current policy, use it to improve policy	fit $V(s)$ or $Q(s, a)$	$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}[Q(s_t, a_t)]$
Model-based	estimate transition model and use it for planning, to improve a policy, etc.	learn $p(s_{t+1}\|s_t)$	A few ways: 1. Just use the model to plan (no policy), such as trajectory optimization/optimal control (e.g. backprop in continuous space or Monte Carlo tree search in discrete space) 2. Backprop gradients into policy 3. Use model to learn a value function with DP or by generating simulated experience for model-free learner

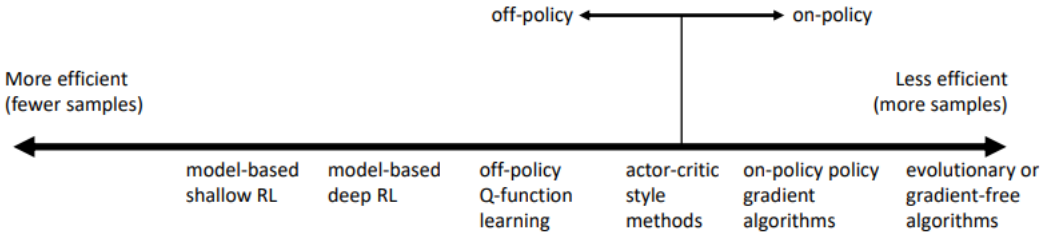
3.5 Tradeoffs Between Algorithms

There are many RL algorithms because there are

- Different tradeoffs: sample efficiency, stability and ease of use
- Different assumptions: stochastic or deterministic, continuous or discrete, episodic or infinite horizon
- DIfferent things are easy or hard in different settings: easier to represent the policy, easier to represent the model

3.5.1 Sample Efficiency

Sample efficiency spectrum:



Off-policy means being able to improve the policy without generating new samples from the policy. On-policy means that each time the policy is changed, even a little bit, we need to generate new samples. Note that wall clock time is not the same as sample efficiency.

3.5.2 Stability and Ease of Use

Unlike supervised learning, RL often does not use gradient descent so there are not as much convergence guarantees:

- Value function fitting: At best, Bellman error (i.e. error of fit) is minimized. At worst, nothing is optimized (many not guaranteed to converge to anything in nonlinear case)
- Model-based RL: Model minimizes error of fit. But no guarantee better model = better policy.
- Policy gradient: The only one that actually performs gradient descent (or ascent) on the true objective.

3.5.3 Common Assumptions

1. Full observability
    (a) Generally assumed by value function fitting methods
    (b) Can be mitigated by adding recurrence
2. Episodic learning
    (a) Often assumed by pure policy gradient methods
    (b) Assumed by some model-based RL methods
3. Continuity/Smoothness
    (a) Assumed by some continuous value function learning methods
    (b) Often assumed by some model-based RL methods

Chapter 4

Policy Gradient

4.1 Direct Policy Differentiation

Remember that in RL

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t r(s_t, a_t)\right]$$

For simplicity, define

$$J(\theta) := \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t r(s_t, a_t)\right] \approx \frac{1}{N}\sum_i \sum_t r(s_{i,t}, a_{i,t})$$

and define

$$r(\tau) = \sum_{t=1}^{T} r(s_t, a_t)$$

so, then

$$J(\theta) := \mathbb{E}_{\tau\sim p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau) r(\tau) d\tau$$

A convenient identity we will use quite a bit for the future is that

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

Using this identity, we find that

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau = \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = \mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

We know that

$$p_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$\log p_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)$$

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Now, we have a nicer expression for $\nabla_\theta J(\theta)$:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right] \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right)\end{aligned}$$

Now if we simply do gradient ascent using this gradient, we have our most basic policy gradient algorithm REINFORCE:

Algorithm 2 REINFORCE

1: **loop**
2: sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (run the policy)
3: $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right)$
4: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
5: **end loop**

4.2 Understanding Policy Gradients

In policy gradients, the gradient is

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right)$$

In MLE, the gradient is

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right)$$

Intuitively, REINFORCE is then just trial and error learning where good stuff is made more likely and bad stuff is made less likely.

Also note that if we follow the same derivation with partial observability, we get

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|o_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right)$$

so we can use policy gradient on POMDPs without modification.

4.3 Reducing Variance

The current policy gradient won't work in practice because the variance of the gradient is too big. We can implement some tricks to reduce the variance of the gradient while keeping the estimate unbiased.

4.3.1 Causality

Through some derivation, we can show that

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=1}^{t} r(s'_t, a'_t) \right) \right] = 0$$

Intuitively, this makes sense since the policy at time t' cannot affect reward at time t when $t < t'$. So,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) \right)$$

is the same as our original approximation in expectation and now has smaller variance. We will define the right-hand expression as our "reward to go" $\hat{Q}_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$.

4.3.2 Baselines

Average Reward

It turns out that

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

is also a valid approximation where

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau)$$

This is because

$$\begin{aligned}\mathbb{E}[\nabla_\theta \log p_\theta(\tau) b] &= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) b d\tau \\ &= \int \nabla_\theta p_\theta(\tau) b d\tau \\ &= b \nabla_\theta \int p_\theta(\tau) d\tau \\ &= b \nabla_\theta 1 = 0\end{aligned}$$

In other words, subtracting a baseline is also unbiased in expectation. Note that average reward is not the best baseline, but it's still pretty good.

Optimal Baseline

The best baseline minimizes the variance of the gradient, which is

$$\text{Var} = \mathbb{E}_{\tau \sim p_\theta(\tau)}[(\nabla_\theta \log p_\theta(\tau)(r(\tau) - b))^2] - \mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau)(r(\tau) - b)]^2$$

For convenience, let

$$g(\tau) := \nabla_\theta \log p_\theta(\tau)$$

To minimize this variance, we take a derivative of the variance and set it to zero. Note that the latter part of the variance expression is just $\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$ because baselines are unbiased in expectation, so its derivative in terms of b is just zero.

$$\begin{aligned}\frac{\partial \text{Var}}{\partial b} &= \frac{\partial}{\partial b} \mathbb{E}[g(\tau)^2 (r(\tau) - b)^2] \\ &= \frac{\partial}{\partial b} (\mathbb{E}[g(\tau)^2 r(\tau)^2] - 2\mathbb{E}[g(\tau)^2 r(\tau) b] + b^2 \mathbb{E}[g(\tau)^2]) \\ &= -2\mathbb{E}[g(\tau)^2 r(\tau)] + 2b\mathbb{E}[g(\tau)^2] = 0 \\ b &= \frac{\mathbb{E}[g(\tau)^2 r(\tau)]}{\mathbb{E}[g(\tau)^2]}\end{aligned}$$

So the optimal baseline is just the expected reward, but weighted by gradient magnitudes.

4.4 Off-Policy Policy Gradients

In off-policy policy gradients, we want to improve our policy with data generated from a different policy. We can do this with importance sampling:

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

In our case, we want to improve policy θ' with data generated from policy θ. So,

$$\begin{aligned}
\nabla_{\theta'}J(\theta') &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\frac{p_{\theta'}(\tau)}{p_\theta(\tau)}\nabla_{\theta'}\log\pi_{\theta'}(\tau)r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\right)\left(\sum_{t=1}^{T}r(s_t,a_t)\right)\right] \\
&= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_\theta(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T}r(s_{t'},a_{t'})\left(\prod_{t''=t}^{t'}\frac{\pi_{\theta'}(a_{t''}|s_{t''})}{\pi_\theta(a_{t''}|s_{t''})}\right)\right)\right]
\end{aligned}$$

If we ignore the importance weight ratios over t'', we get a policy iteration algorithm (more on this later), so we can simply the gradient to

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

Note that the importance weight ratios are exponential in T, so its variance will grow exponentially in T. Instead, let us write the objective in terms of state-action marginals, so

$$\begin{aligned}
\nabla_{\theta'}J(\theta') &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t},a_{i,t})}{\pi_\theta(s_{i,t},a_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t} \\
&\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t})}{\pi_\theta(s_{i,t})}\frac{\pi_{\theta'}(a_{i,t}|s_{i,t})}{\pi_\theta(a_{i,t}|s_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t}
\end{aligned}$$

If we ignore the ratio of the state priors, we have

$$\nabla_{\theta'}J(\theta') \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(a_{i,t}|s_{i,t})}{\pi_\theta(a_{i,t}|s_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t}$$

This is no longer unbiased, but its error can be bounded in terms of the difference between θ and θ' (more on this later). Under this approximation, we get rid of the exponentially growing variance.

4.5 Covariant/Natural Policy Gradient

One issue with policy gradients is that most likely than not the gradients dominate in terms of some parameters, leading it to be hard to converge to optimal parameters. So instead, we need to rescale the gradient so that this doesn't happen.

Recall that with vanilla gradient ascent, we have

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\theta_k)$$

This is equivalent to

$$\theta_{k+1} \leftarrow \arg\max_{\theta} \alpha \nabla_{\theta_k} J(\theta_k)^T \theta - \frac{1}{2}\|\theta - \theta_k\|^2$$

This makes sense since the above expression is just maximizing the linear approximation of $J(\theta)$ at θ_k with a quadratic regularization. Using Lagrangian form, we can rewrite the above expression as

$$\theta' \leftarrow \arg\max_{\theta'} (\theta' - \theta)^T \nabla_\theta J(\theta) \text{ s.t. } \|\theta' - \theta\| \leq \epsilon$$

Currently the constraint is in parameter-space. As a result, it doesn't account for the fact that some parameters influence the policy more than others. So we would like a constraint in policy-space. One parameterization-independent divergence measure is KL-divergence: $\mathcal{D}_{KL}(\pi_{\theta'} \| \pi_\theta) = \mathbb{E}_{\pi_{\theta'}}[\log \pi_\theta - \log \pi_{\theta'}]$.

$$\theta' \leftarrow \arg\max_{\theta'} (\theta' - \theta)^T \nabla_\theta J(\theta) \text{ s.t. } \mathcal{D}_{KL}(\pi_{\theta'} \| \pi_\theta) \leq \epsilon$$

We can approximate the KL-divergence with its second-order Taylor approximation around $\theta' = \theta$,

$$\mathcal{D}_{KL}(\pi_{\theta'} \| \pi_\theta) \approx (\theta' - \theta)^T \mathbf{F} (\theta' - \theta)$$

$\mathbf{F}$ is the Fischer-information matrix, where

$$\mathbf{F} = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s})^{\mathbf{T}}]$$

which can be estimated with samples taken form π_θ.

After writing out the Lagrangian and solving for the optimal solution, we find that the update rule is

$$\theta \leftarrow \theta + \alpha \mathbf{F^{-1}} \nabla_\theta \mathbf{J}(\theta)$$

The natural policy gradient selects α. Trust region policy optimization (TRPO) selects ϵ and then derives α using conjugate gradient.

Chapter 5

Actor-Critic Algorithms

5.1 Policy Evaluation

Recall that in policy gradient, we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \hat{Q}^\pi_{i,t}$$

$\hat{Q}_{i,t}$ is an estimate of expected reward if we take action $a_{i,t}$ in state $s_{i,t}$. So far, we use a single-sample estimate with $\hat{Q}_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$. If we can instead approximate the true expected reward-to-go, $Q(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t]$, then we will have a lower variance estimate. So can could instead approximate the Q-function and calculate the gradient as

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) Q(s_{i,t}, a_{i,t})$$

We can also add in our baseline to reduce variance, so

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) (Q(s_{i,t}, a_{i,t}) - b)$$

One idea for the baseline is to average Q-values: $b_t = \frac{1}{N} \sum_i Q(s_{i,t}, a_{i,t})$. We can reduce variance even more by averaging over actions for a specific state; this is exactly the value function: $V(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q(s_t, a_t)]$. Our gradient then becomes

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) (Q(s_{i,t}, a_{i,t}) - V(s_{i,t}))$$

In terms of notation, we denote the Q-function, or the total reward from taking a_t in s_t, as

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t]$$

We denote value function, or the total reward from s_t, as

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q^\pi(s_t, a_t)]$$

We define the advantage function, or how much better a_t is as

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

With this notation we can write our gradient as

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) A^\pi(s_{i,t}, a_{i,t})$$

Now the question is whether we fit Q^π and/or V^π and/or A^π. We know that

$$\begin{aligned} Q^\pi(s_t, a_t) &= r(s_t, a_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] \\ &= r(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)}[V^\pi(s_{t+1})] \\ &\approx r(s_t, a_t) + V^\pi(s_{t+1}) \end{aligned}$$

In the last line, we approximate the Q-function with a single-sample estimate from the transition dynamic $p(s_{t+1}|s_t, a_t)$ at the cost of increasing the variance of our Q-value approximation. With this approximation, we now have

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t)$$

Thus, we can just fit the value function and use it to calculate the advantage.

With Monte Carlo policy evaluation, we can estimate the value function as

$$V^\pi(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$$

However, we can't do this in most model-free settings since we can't reset the simulator at any state. So instead, we will create a function approximation of the value function with a network. This isn't as good as Monte Carlo, but still pretty good. In a function approximation setting, our training data for the value network will be $\{(s_{i,t}, y_{i,t})\}$ where $y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$. We then train supervised regression with the loss $\mathcal{L}(\phi) = \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi(s_i) - y_i\|^2$. By using a network, similar states will map to similar actions.

We can further reduce the variance with bootstrapping. Recall that our ideal target is

$$y_{i,t} = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{i,t}] \approx r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}) \approx r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$$

In the first approximation, we increase the variance of our estimate by using a single-sample estimate of the expectation under $p(s_{t+1}|s_t, a_t)$ (as we did before). In the second approximation, we increase the variance of our estimate by using bootstrapped estimate, where we directly use our previous fitted value function. With bootstrapping, our training data now is $\{(s_{i,t}, y_{i,t})\}$ where $y_{i,t} = r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$, and we train supervised regression on the value network with loss function $\mathcal{L}(\phi) = \frac{1}{2}\sum_i \|\hat{V}^\pi_\phi(s_i) - y_i\|^2$.

5.2 Actor-Critic

Algorithm 3 Batch Actor-Critic Algorithm (Without Discounts)

1: **loop**
2: sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$
3: fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums
4: evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$
5: $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)\hat{A}^\pi(s_i, a_i)$
6: $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
7: **end loop**

We can fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums with the loss $\mathcal{L}(\phi) = \frac{1}{2}\sum_i \|\hat{V}^\pi_\phi(s_i) - y_i\|^2$ where $y_{i,t} = r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$.

If the episode length, T, is ∞, $\hat{V}^\pi_\phi$ can get infinitely large in many cases. So we will add a discount factor $\gamma \in [0, 1]$ (0.99 works well). Adding γ changes the MDP. There is a new state we can call the death state and at each state, there is a $1-\gamma$ probability of transitioning to the death state. The original transition dynamics now get multiplied by a factor of γ (i.e. $\gamma p(s'|s, a)$). Our new target becomes $y_{i,t} = r(s_{i,t}, a_{i,t}) + \gamma\hat{V}^\pi_\phi(s_{i,t+1})$. With actor-critic, our gradient is

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(r(s_{i,t}, a_{i,t}) + \gamma\hat{V}^\pi_\phi(s_{i,t+1}) - \hat{V}^\pi_\phi(s_{i,t}))$$

With Monte Carlo policy gradients, we have two options:

$$\text{Option 1: } \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{i,t'},a_{i,t'}))$$

$$\begin{aligned}\text{Option 2: } \nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=1}^{T}\gamma^{t'-1}r(s_{i,t'},a_{i,t'}))\\ &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=t}^{T}\gamma^{t'-1}r(s_{i,t'},a_{i,t'}))\\ &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\gamma^{t-1}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{i,t'},a_{i,t'}))\end{aligned}$$

We use option 1 over option 2 because we want a policy that does well at every timestep, not just earlier timesteps.

Algorithm 4 Batch Actor-Critic Algorithm

loop
  sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$
  fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums
  evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma\hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$
  $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)\hat{A}^\pi(s_i, a_i)$
  $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
end loop

We can also create a fully online version of actor-critic.

Algorithm 5 Online Actor-Critic Algorithm

loop
  take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
  update $\hat{V}^\pi_\phi$ using target $r + \gamma\hat{V}^\pi_\phi(s')$
  evaluate $\hat{A}^\pi(s, a) = r(s, a) + \gamma\hat{V}^\pi_\phi(s') - \hat{V}^\pi_\phi(s)$
  $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s)\hat{A}^\pi(s, a)$
  $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
end loop

5.3 Actor-Critic Design Decisions

We can either use a separate network for both $\hat{V}^\pi_\phi(s)$ and $\pi_\theta(a|s)$ or just a shared network design. Single-sample backpropagation updates in online actor-critic is typically not stable due to the high variance of policy gradients. We could instead take 8-16 steps in the environment and update our network on that

batch. However, these data points are highly correlated. Instead, we can use parallel workers. In synchronized parallel actor-critic, each worker is initialized in a separate seed, and, at each timestep, all the workers take a step in the environment and return (s, a, s', r), which are batched up and used to update θ. In asynchronous parallel actor-critic, the workers collect data at whatever rate they are able to, and once they collect a transition, they send it to a central parameter server. The parameter server can then make an update and send the updated parameters back to the workers. In the meantime, the workers aren't waiting but are collecting more samples. One might note that workers might be taking steps with an older version of the policy even though the parameter server might have a newer version that hasn't deployed yet. This tends to be ok because the old and new parameters are similar enough.

5.4 Critics as Baselines

5.4.1 Critics as State-Dependent Baselines

In actor-critic,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) - \hat{V}_\phi^\pi(s_{i,t}))$$

This has lower variance (due to critic), but is not unbiased if critic is not perfect. On the other hand, in policy gradient,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})((\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t}, a_{i,t})) - b$$

This has no bias, but has higher variance because it is a single-sample estimate. One way to balance this tradeoff is to use

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})((\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t}, a_{i,t})) - \hat{V}_\phi^\pi(s_{i,t}))$$

This has no bias and has lower variance than the policy gradient approach since the baseline is closer to rewards.

5.4.2 Control Variates: Action-Dependent Baselines

Currently, our advantage is

$$\hat{A}^\pi(s_t, a_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r(s_t, a_t) - \hat{V}_\phi^\pi(s_t)$$

This has no bias but has higher variance due to it being a single-sample estimate. Instead, if we use

$$\hat{A}^\pi(s_t, a_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r(s_t, a_t) - \hat{Q}^\pi_\phi(s_t, a_t)$$

This goes to zero in expectation if the critic is correct, but is not exactly correct. To be unbiased, we need to add an error term

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\hat{Q}_{i,t} - Q^\pi_\phi(s_{i,t}, a_{i,t})) + \nabla_\theta \mathbb{E}_{a \sim \pi_\theta(a_t|s_t)}[Q^\pi_\phi(s_{i,t}, a_t)] \right)$$

Provided that the second term can be evaluated (which it can in this case), we have an unbiased estimate.

5.4.3 Eligibility Traces & N-Step Returns

A bootstrapped estimate has lower variance but higher bias if the estimate is wrong (it always is). A Monte Carlo estimate has no bias but has higher variance because it is a single-sample estimate. We can control the bias/variance tradeoff by switching from Monte Carlo to bootstrapping at some timestep $n > 1$.

$$\hat{A}^\pi_n(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}^\pi_\phi(s_t) + \gamma^n \hat{V}^\pi_\phi(s_{t+n})$$

Later steps along Monte Carlo has bigger variance, so we using bootstrapping for later on.

5.4.4 Generalized Advantage Estimation

GAE is a weighted combination of n-step returns.

$$\hat{A}^\pi_{GAE}(s_t, a_t) = \sum_{n=1}^{\infty} w_n \hat{A}^\pi_n(s_t, a_t)$$

Since we prefer cutting earlier since there is less variance early, we can use an exponential falloff. Define $\delta_{t'} = r(s_{t'}, a_{t'}) + \gamma \hat{V}^\pi_\phi(s_{t'+1}) - \hat{V}^\pi_\phi(s_{t'})$.

$$\begin{aligned}
\hat{A}^\pi_{GAE}(s_t, a_t) &= (1-\lambda)(\hat{A}^\pi_1(s_t, a_t) + \lambda \hat{A}^\pi_2(s_t, a_t) + \lambda^2 \hat{A}^\pi_3(s_t, a_t) + ...) \\
&= (1-\lambda)(\delta_t(1 + \lambda + \lambda^2 + ...) + \gamma \delta_{t+1}(\lambda + \lambda^2 + ...) + ...) \\
&= \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'}
\end{aligned}$$

This has a very similar effect to discounts, also implying that discounts have a role in the bias-variance tradeoff.

Chapter 6

Value Function Methods

6.1 Policy Iteration

Instead of using policy gradient to train a policy, we could define

$$\pi'(a_t|s_t) = \begin{cases} 1 & a_t = \arg\max_{a_t} A^\pi(s_t, a_t) \\ 0 & otherwise \end{cases}$$

Since $A^\pi(s,a) = r(s,a) + \gamma \mathbb{E}[V^\pi(s')] - V^\pi(s)$, we can just evaluate $V^\pi(s)$. For now, assume we know $p(s'|s,a)$, and s and a are both discrete (and small). Since our policy is deterministic, our bootstrapped update is

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma \mathbb{E}_{s' \sim p(s'|s,\pi(s))}[V^\pi(s')]$$

With our policy and policy evaluation procedure defined, we now have a policy iteration algorithm using dynamic programming:

Algorithm 6 Policy Iteration

1: **loop**
2: Evaluate $V^\pi(s)$ with bootstrapped estimate
3: set $\pi \leftarrow \pi'$ where π' is defined up above
4: **end loop**

Note that $\arg\max_{a_t} A^\pi(s_t, a_t) = \arg\max_{a_t} Q^\pi(s_t, a_t)$. Since $\arg\max_a Q(s,a)$ is our policy, we can compute Q-values instead of a policy.

Algorithm 7 Value Iteration

1: **loop**
2: set $Q(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}[V(s')]$
3: set $V(s) \leftarrow \max_a Q(s,a)$
4: **end loop**

6.2 Fitted Value Iteration & Q-Iteration

Representing states in a big table for DP becomes unrealistic for larger state spaces such as images. Instead, we should do regression on the value function

$$\mathcal{L}(\phi) = \frac{1}{2}\|V_\phi(s) - \max_a Q^\pi(s,a)\|^2$$

With this loss, we can do value iteration with a regression value network.

Algorithm 8 Fitted Value Iteration

loop
  set $y_i \leftarrow \max_{a_i}(r(s_i,a_i) + \gamma \mathbb{E}[V_\phi(s_i')])$
  set $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|V_\phi(s_i) - y_i\|^2$
end loop

However, in order to do this, we need to know outcomes for different actions. Instead, let us iterate on the Q-function instead of the value function. For policy evaluation, instead of

$$V^\pi(s) \leftarrow r(s,\pi(s)) + \gamma \mathbb{E}_{s'\sim p(s'|s,\pi(s))} V^\pi(s')$$

we iterate on the Q-values

$$Q^\pi(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)}[Q^\pi(s',\pi(s'))]$$

This way, we don't have to take a max over a_i when computing target values.

Algorithm 9 Fitted Q-Iteration

loop
  set $y_i \leftarrow r(s_i,a_i) + \gamma \mathbb{E}[V_\phi(s_i')]$
  $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|Q_\phi(s_i,a_i) - y_i\|^2$
end loop

We can now write out the full fitted Q-iteration algorithm:

Algorithm 10 Full Fitted Q-Iteration

loop
  collect dataset $\{(s_i,a_i,s_i',r_i)\}$ using some policy
  for K iterations **do**
    set $y_i \leftarrow r(s_i,a_i) + \gamma \max_{a_i'} Q_\phi(s_i',a_i')$
    $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|Q_\phi(s_i,a_i) - y_i\|^2$
  end for
end loop

This works for off-policy samples, uses only one network, and has no high-variance policy gradient. However, there are no convergence guarantees for

non-linear function approximation (more on this later). Now, we have a Q-network with takes in state and action and outputs a number. In the discrete action case as we have right now, we can also structure the network to just take in state and output a head for each action.

6.3 Q-Learning

Fitted Q-iteration is off-policy because given s and a, the transition is independent of π. The bellman error

$$\mathcal{E} = \frac{1}{2}\mathbb{E}_{(s,a)\sim\beta}\left[\left(Q_\phi(s,a) - [r(s,a) + \gamma \max_{a'} Q_\phi(s',a')]\right)^2\right]$$

is approximated by $\sum_i \|Q_\phi(s_i,a_i) - y_i\|^2$ at each step of fitted Q-iteration. If $\mathcal{E} = 0$, then $Q_\phi(r,a) = r(s,a) + \gamma \max_{a'} Q_\phi(s',a')$. This is an optimal Q-function, corresponding to the optimal policy π'. However, most guarantees are lost when we leave the tabular case.

We can convert fitted Q-iteration into an online analogue, which we call Q-learning.

Algorithm 11 Online Q-Iteration

1: **loop**
2: take some action a_t and observe (s_i, a_i, s_i', r_i)
3: $y_i = r(s_i,a_i) + \gamma \max_{a'} Q_\phi(s_i', a_i')$
4: $\phi \leftarrow \phi - \alpha \frac{\partial Q_\phi}{\partial \phi}(s_i,a_i)(Q_\phi(s_i,a_i) - y_i)$
5: **end loop**

We call $(Q_\phi(s_i,a_i) - y_i)$ the temporal difference (TD) error. In Q-learning, if we explore just based on just the argmax policy, we may be stuck in a subset of the environment and miss out on states and actions that give larger rewards. Some exploration policies are epsilon-greedy

$$\pi(a_t|s_t) = \begin{cases} 1-\epsilon & a_t = \arg\max_{a_t} Q_\phi(s_t,a_t) \\ \epsilon/(|\mathcal{A}|-1) & otherwise \end{cases}$$

and Boltzmann exploration

$$\pi(a_t|s_t) \propto \exp\left(Q_\phi(s_t,a_t)\right)$$

We'll discuss exploration in detail later.

6.4 Value Functions in Theory

We will look at convergence guarantees for these algorithms.

6.4.1 Tabular Value Iteration

Tabular value iteration can be concisely described with the bellman backup operator $\mathcal{B} : \mathcal{B}V = \max_{\mathbf{a}} r_{\mathbf{a}} + \gamma \mathcal{T}_{\mathbf{a}} V$ where $\mathcal{T}_{\mathbf{a},i,j} = p(\mathbf{s}' = i | \mathbf{s} = j, \mathbf{a})$, $r_{\mathbf{a}}$ is a stacked vector of rewards for all states for action $\mathbf{a}$, and we are doing an element-wise max. Note that V^* is a fixed point of $\mathcal{B}$ because $V^* = \mathcal{B}V^*$, so it is unique and always corresponds to the optimal policy. Value iteration reaches V^* because $\mathcal{B}$ is a contraction: for any V and $\bar{V}$, we have $\|\mathcal{B}V - \mathcal{B}\bar{V}\|_\infty \leq \gamma \|V - \bar{V}\|_\infty$ (not proved here), so $\|\mathcal{B}V - V^*\|_\infty \leq \gamma \|V - V^*\|_\infty$. Here, we have shown that we converge on the optimal policy with tabular value iteration.

6.4.2 Non-Tabular Value Iteration

Let's introduce a new operator $\Pi : \Pi V = \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(s) - V(s)\|^2$, which is a projection onto ω in terms of ℓ_2 norm. Fitted value iteration can be described as $V \leftarrow \Pi \mathcal{B} V$. $\mathcal{B}$ is a contraction w.r.t ∞-norm and Π is a contraction w.r.t ℓ_2-norm, but $\Pi\mathcal{B}$ is not a contraction of any kind, so it does not converge in general and often not in practice.

6.4.3 Fitted Q-Iteration

Similarly, if we define $\Pi : \Pi Q = \arg\min_{Q' \in \Omega} \frac{1}{2} \sum \|Q'(s,a) - Q(s,a)\|^2$, we can describe fitted Q-iteration as $Q \leftarrow \Pi \mathcal{B} Q$. Again, we see that $\Pi\mathcal{B}$ is not a contraction of any kind, so it does not converge in general and often not in practice. This also applies to online Q-learning. Note that Q-learning is not gradient descent because there is no gradient through the target value.

6.4.4 Actor-Critic

In actor-critic, we also have $\mathcal{B}$ without the max and Π in fitting the value function, so fitted bootstrapped policy evaluation also does not converge for the same reason.

Chapter 7

Deep RL with Q-Functions

7.1 Replay Buffers

In online Q-learning, sequential states are strongly correlated and the target value is always changing. Since sequential states are strongly correlated, it is possible for our Q-network to overfit to different chunks along a training trajectory. We could use synchronized parallel Q-learning or asynchronous parallel Q-learning as we did with actor-critic. Another solution is to use replay buffers, which stores a dataset of the agent's most recent trajectories (old trajectories are thrown away when the replay buffer hits a threshold limit).

Algorithm 12 Full Q-Learning with Replay Buffer

1: **while** some stop condition is not satisfied **do**
2: collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
3: **for** K iterations **do**
4: sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$
5: $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)])$
6: **end for**
7: **end while**

7.2 Target Networks

We have one more issue in that our Q network changes every gradient step, so our target changes every gradient step. As a result, it is possible that this type of "gradient descent" won't converge, since our network is sort of "chasing its own tail". The solution to this is to save an old version of the model for gradient descent and take multiple gradient steps before updating a newer version of the model for gradient descent. We call this old version of the model to be used in the loss function for gradient descent the target network.

Algorithm 13 Q-Learning with Replay Buffer and Target Network

1: **while** some stop condition is not satisfied **do**
2: save target network parameters: $\phi' \leftarrow \phi$
3: **for** N iterations **do**
4: collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
5: **for** K iterations **do**
6: sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$
7: $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_{\phi'}(s'_i, a'_i)])$
8: **end for**
9: **end for**
10: **end while**

The classic DQN is essentially Q-Learning with Replay Buffer and Target Network with $K = 1$:

Algorithm 14 Classic Deep Q-Learning (DQN)

1: **while** some stop condition is not satisfied **do**
2: take some action a_i and observe (s_i, a_i, s'_i, r_i), add it to $\mathcal{B}$
3: sample mini-batch (s_j, a_j, s'_j, r_j) from $\mathcal{B}$ uniformly
4: compute $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using target network $Q_{\phi'}$
5: $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_j)$
6: update ϕ': copy ϕ every N steps
7: **end while**

A popular alternative target network is using Polyak averaging, where in line 6 of DQN we instead set $\phi' : \phi' \leftarrow \tau\phi' + (1-\tau)\phi$. $\tau = 0.999$ works well in practice. The intuition here is to avoid the maximal lag that takes place for update $N-1$ (mod N) compared to to update 0 (mod N) where there is no lag.

7.3 Improving Q-Learning

7.3.1 Double Q-Learning

Recall that in Q-learning, our target value is $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$. However, $Q_{\phi'}(s'_j, a'_j)$ is noisy and thus $\max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ overestimates the next value because for two random variables X_1 and X_2, $\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$. Note that $\max_{a'} Q_{\phi'}(s', a') = Q_{\phi'}(s', \arg\max_{a'} Q_{\phi'}(s', a'))$. If the value calculated and the best action selected were decorrelated, then this problem goes away. This is where double Q-learning comes in, which involves two networks:

$$Q_{\phi_A}(s, a) \leftarrow r + \gamma Q_{\phi_B}(s', \arg\max_{a'} Q_{\phi_A}(s', a'))$$

$$Q_{\phi_B}(s, a) \leftarrow r + \gamma Q_{\phi_A}(s', \arg\max_{a'} Q_{\phi_B}(s', a'))$$

By using a different function approximator for selecting the best action and calculating the value, it is unlikely that the action will be overestimated. In practice, we can use our current network to find the best action and our target network to evaluate its value.

7.3.2 Multi-Step Returns

Again, recall that in Q-learning, our target value is

$$y_{j,t} = r_{j,t} + \gamma \max_{a_{j,t+1}} Q_{\phi'}(s_{j,t+1}, a_{j,t+1})$$

Like in actor-critic, we can instead use N-step return estimators:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{j,t'} + \gamma^N \max_{a_{j,t+N}} Q_{\phi'}(s_{j,t+N}, a_{j,t+N})$$

This estimator has less biased target values when Q-values are inaccurate (i.e. in the beginning of training) and typically learn faster early on. However, it is only actually correct when learning on-policy because we need transitions $s_{j,t'}, a_{j,t'}, s_{j,t'+1}$ to come from π for $t' - t < N - 1$ when $N > 1$. To fix this, we could ignore the problem (often works well in practice), cut the trace by dynamically choosing N to get only on-policy data (works well when data is mostly on-policy and action space is small), or do importance sampling.

7.3.3 Q-Learning with Continuous Actions

For continuous actions, we have trouble finding $\max_{a_{j,t+1}} Q_{\phi'}(s_{j,t+1}, a_{j,t+1})$. We have three options:

1. Optimization: we could do gradient based optimization such as SGD. However, this is a bit slow in the inner loop. Instead, we can do stochastic optimization. A simple and parallelizable solution is to sample actions from some distribution (e.g. uniform) and take the max over sampled actions. This is not very accurate however. More accurate solutions are cross-entropy method (CEM), a simple iterative stochastic optimization, and CMA-ES, which is less simple.

2. Maximizable Q-functions: we could use a function class that is easy to optimize such as Normalized Advantage Functions (NAF):

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

   where $\arg\max_a Q_\phi(s, a) = \mu_\phi(s)$ and $\max_a Q_\phi(s, a) = V_\phi(s)$. This efficient option requires no change to the algorithm, but representation power is lost.

3. Approximate Maximizer: we could train another network such that $\mu_\theta(s) \approx \arg\max_a Q_\phi(s,a)$ with backpropagation: $\frac{dQ_\phi}{d\theta} = \frac{da}{d\theta}\frac{dQ_\phi}{da}$. Then our new target becomes $y_j = r_j + \gamma Q_{\phi'}(s'_j, \mu_\theta(s'_j))$. The Deep Deterministic Policy Gradient (DDPG) algorithm is as follows

Algorithm 15 DDPG

while some stop condition is not satisfied **do**
  collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
  sample mini-batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$ uniformly
  compute $y_j = r_j + \gamma Q_{\phi'}(s'_j, \mu_{\theta'}(s'_j))$ using target nets $Q_{\phi'}$ and $\mu_{\theta'}$
  $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$
  $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(s_j)\frac{dQ_\phi}{da}(s_j, \mu(s_j))$
  update ϕ' and θ' (e.g. Polyak averaging)
end while

7.4 Implementation Tips

- Q-learning takes some care to stabilize, so test on easy, reliable tasks first to make sure your implementation is correct
- Large replay buffers help improve stability
- It takes time
- Start with high exploration (epsilon) and gradually reduce
- Bellman error gradients can be big; clip gradients or user Huber loss
- Double Q-learning helps a lot in practice and has no downsides
- N-step returns help a lot, but have some downsides
- Schedule exploration and learning rates high to low, Adam optimizer can help too
- Run multiple random seeds, it's very inconsistent between runs