

# Reinforcement learning notes

Marko Guberina

March 10, 2022

## 0.1 Plan

Let's start by reading Sutton's book to get the basic theory down. I'll definitely skip some stuff I've passed so far to make it easier to get the ball rolling. This will be supplemented by Deep Mind's lectures. Once the basic theory is introduced, we'll start going through key papers one by one, starting with deep q-learning by DeepMind and continuing until the freshest stuff.

**Update** Yeah, that did not pan out that way. Turns out Sergey Levine's Berkeley course is much more on the money for what I need right now — going straight to function approximation with neural networks and straight to policy gradients after introducing the definitions. There are also very nice homeworks to go along with that and it seems like the way to go for now. After all, I want to get up to speed with implementations immediately. Once that's covered, I'll go back to Sutton's book and learn the proper theory. It seems like those deeper theoretical insights have are more like a sauce then the meat. Of course, they are crucial if one wants to prove things, but proving things in RL is a research frontier, and not a backbone for practical usage (at least not for the deep reinforcement learning where the focus seems to be integrating other ML successes in fields like computer vision).

So, to sum up, right now I want to understand the algorithms so that I can understand their code so that I can play with their code. The focus of the "playing with the code" part is to get the algorithms to work on pixels and sticking an autoencoder in the right place. Because that requires an understanding of how the current deep reinforcement learning algorithms work, that's step 1. Implementing a few algorithms for practise (and then reading good implementations) is step 2. Only then in step 3 do I get to implement what I'm tasked with.

### 0.1.1 TODOs

1. go through berkeley lectures again, see whether you understand everything and CORRECT THE MISTAKES
2. replace enum algorithms with something nicer (some box or something) or alternatively write the algorithms as pseudocode (probably better). for the second case you can check out overleaf algorithms page
3. create index with nice links around the pdf, use overleaf hyperlinks page to get there
4. (optional for now) clean language

**Purpose** These notes serve multiple purposes: firstly, of course, is gaining the necessary theoretical knowledge to even describe what's going on. Secondly, it will enable generating hypothesis about new possible algorithms. Finally, they will serve as a reference - a reinforcement learning handbook if you will.

# Chapter 1

## Berkley AI class

### 1.1 Immitation learning

skip lel, pls do it if you go for the classes' homeworks tho

### 1.2 Formal setting

#### 1.2.1 Markov chain

- $\mathcal{M} = \{\mathcal{S}, \mathcal{T}\}$
- $\mathcal{S}$  - state space,  $s \in \mathcal{S}$  (discrete or continuous)
- $\mathcal{T}$  - transition operator — for  $p(s_{t+1}|s_t)$  let  $\mu_{t,i} = p(s_t = i)$ ,  $\mathcal{T}_{i,j} = p(s_{t+1} = j | s_t = i)$ . Then  $\vec{\mu}_t$  is a vector of probabilities and  $\vec{\mu}_{t+1} = \mathcal{T} \vec{\mu}_t$
- we have the markov property ofc

If we add actions and rewards:

#### 1.2.2 Markov decision process

- $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$
- $\mathcal{S}$  - state space,  $s \in \mathcal{S}$  (discrete or continuous)
- $\mathcal{A}$  - action space,  $a \in \mathcal{A}$  (discrete or continuous)
- $\mathcal{T}$  - transition operator is now a tensor — let  $\mu_{t,j} = p(s_t = j)$ ,  $\xi_{t,k} = p(a_t = k)$ ,  $\mathcal{T}_{i,j,k} = p(s_{t+1} = i | s_t = j, a_t = k)$  and we get  $\mu_{t+1,i} = \sum_{j,k} \mathcal{T}_{i,j,k} \mu_{t,j} \xi_{t,k}$
- so the tensor version of the operator is still linear
- $r$  - reward function ( $r(s_t, a_t)$ ),  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$

And if we don't have access to full states, but only partial observations of states:

### 1.2.3 Partially observed Markov decision process

- $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{E}, r\}$
- $\mathcal{S}$  - state space,  $s \in \mathcal{S}$  (discrete or continuous)
- $\mathcal{A}$  - action space,  $a \in \mathcal{A}$  (discrete or continuous)
- $\mathcal{P}$  - observation space,  $o \in \mathcal{O}$  (discrete or continuous)
- $\mathcal{T}$  - transition operator (like before)
- $\mathcal{E}$  - emission probability  $p(o_t|s_t)$
- $r$  - reward function  $(r(s_t, a_t))$ ,  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$

#### The goal of reinforcement learning

Let's deal with the finite horizon case for now.

$$\underbrace{p_\theta(s_1, a_1, \dots, s_T, a_T)}_{p_\theta(\tau)} = p(s_1) \prod_{t=1}^T \underbrace{\pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)}_{\text{Markov chain on } (s, a)} \quad (1.1)$$

A bit more explicitly:

$$p((s_{t+1}, a_{t+1})|(s_t, a_t)) = p(s_{t+1}|(s_t, a_t))\pi_\theta(a_{t+1}|s_{t+1}) \quad (1.2)$$

This will allow us to define the objective a bit more conveniently. We'll use marginalisation ( $p_\theta(s_t, a_t)$  is the state-action marginal) (will be useful for infinite horizon case):

$$\theta^* = \operatorname{argmax}_\theta E_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(s_t, a_t) \right] \quad (1.3)$$

$$= \operatorname{argmax}_\theta \sum_t^T E_{(s_t, a_t) \sim p_\theta(s_t, a_t)} [r(s_t, a_t)] \quad (1.4)$$

OK, let's do the infinite horizon case ( $T = \infty$ ) with a stationary distribution. One way is to do it with a discount rate  $\gamma \in (0, 1)$ . Does  $p(s_t, a_t)$  converge to a stationary distribution? It does under the ergodicity (if you can get from any state to any other state) and if the chain is aperiodic. In symbols stationarity is  $\mu = \mathcal{T}\mu$  which you get from  $(\mathcal{T} - \mathbf{I})\mu = 0$ . Here  $\mu$  is the eigenvector of  $\mathcal{T}$  with eigenvalue 1 (which always exists under some regularity conditions).

**Note** In RL we care about *expectations*. Because of this our goals are smooth and differentiable and we get to do gradient descent on them.

### 1.2.4 Value functions

Let's start with the expectation which we are trying to maximize w.r.t.  $\theta$ . We'll write it out recursively (by using the chain rule of probability), obtaining nested expectations:

$$E_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \quad (1.5)$$

$$E_{\tau \sim p_\theta(\mathbf{s}_1)} \left[ E_{\mathbf{a}_1 \sim \pi(\mathbf{a}_1 | \mathbf{s}_1)} \left[ r(\mathbf{s}_1, \mathbf{a}_1) + E_{\mathbf{s}_2 \sim p(\mathbf{s}_2 | \mathbf{s}_1, \mathbf{a}_1)} \left[ E_{\mathbf{a}_2 \sim \pi(\mathbf{a}_2 | \mathbf{s}_2)} \left[ r(\mathbf{s}_2, \mathbf{a}_2) + \dots | \mathbf{s}_2 \right] | \mathbf{s}_1, \mathbf{a}_1 \right] | \mathbf{s}_1 \right] \right] \quad (1.6)$$

Enter the Q-functions:

$$Q(\mathbf{s}_1, \mathbf{a}_1) = r(\mathbf{s}_1, \mathbf{a}_1) + E_{\mathbf{s}_2 \sim p(\mathbf{s}_2 | \mathbf{s}_1, \mathbf{a}_1)} \left[ E_{\mathbf{a}_2 \sim \pi(\mathbf{a}_2 | \mathbf{s}_2)} \left[ r(\mathbf{s}_2, \mathbf{a}_2) + \dots | \mathbf{s}_2 \right] | \mathbf{s}_1, \mathbf{a}_1 \right] \quad (1.7)$$

If we knew  $Q(\mathbf{s}_1, \mathbf{a}_1)$ , it would be easy to modify  $\pi_\theta(\mathbf{s}_1, \mathbf{a}_1)$ :

$$E_{\tau \sim p_\theta(\tau)} \left[ \sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right] = E_{\mathbf{s}_1 \sim p_\theta(\mathbf{s}_1)} \left[ E_{\mathbf{a}_1 \sim \pi(\mathbf{a}_1 | \mathbf{s}_1)} [Q(\mathbf{s}_1, \mathbf{a}_1) | \mathbf{s}_1] \right] \quad (1.8)$$

For example we could just do  $\pi(\mathbf{s}_1, \mathbf{a}_1) = 1$  if  $\mathbf{a}_1 = \operatorname{argmax}_{\mathbf{a}_1} Q(\mathbf{s}_1, \mathbf{a}_1)$ .

**Definition: Q-function**

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t] \quad (1.9)$$

thus denoting the total reward from taking  $\mathbf{a}_t$  in  $\mathbf{s}_t$ .

**Definition: value function**

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t] \quad (1.10)$$

thus denoting the total (average/expected) reward from  $\mathbf{s}_t$ .

The connection between the 2 is the following:

$$V^\pi(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi(\mathbf{s}_t, \mathbf{a}_t)} [Q^\pi(\mathbf{s}_t, \mathbf{a}_t)] \quad (1.11)$$

And we can also write the RL objective as:

$$E_{\mathbf{s}_1 \sim p(\mathbf{s}_1)} [V^\pi(\mathbf{s}_1)] \quad (1.12)$$

How can we use Q-functions and value functions? One idea is the following: if we have  $\pi$  and we know  $Q^\pi(\mathbf{s}, \mathbf{a})$ , then we can improve  $\pi$ :

- set  $\pi'(\mathbf{a} | \mathbf{s}) = 1$  if  $\mathbf{a} = \operatorname{argmax}_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a})$

- this policy is at least as good as  $\pi$  and is probably better (easily provable)
- it does not matter what  $\pi$  is, this is always true

Another idea is to compute the gradient to increase the probability of good actions  $\mathbf{a}$ : if  $Q^\pi(\mathbf{s}, \mathbf{a}) > V^\pi(\mathbf{s})$  then  $\mathbf{a}$  is *better than average* (recall definition of  $V^\pi(\mathbf{s})$  under  $\pi(\mathbf{a}|\mathbf{s})$ ). We can then modify  $\pi(\mathbf{a}|\mathbf{s})$  to increase the probability of  $\mathbf{a}$  if  $Q^\pi(\mathbf{s}, \mathbf{a}) > V^\pi(\mathbf{s})$

### 1.3 Policy gradients

**The idea** We are going to directly formalize the concept of trial-and-error learning.

A trajectory distribution in MDP setting is:

$$\underbrace{p_\theta(\mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1) \prod_{t=1}^T \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t) \quad (1.13)$$

The right side is the chain rule of probabilities

The objective of reinforcement learning is:

$$\theta^* = \operatorname{argmax}_{\theta} E_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \quad (1.14)$$

We can push out the sum via the linearity of expectation. This can then be expanded with a marginal for the infinite horizon. Infinite case (can be achieved with value functions):

$$\theta^* = \operatorname{argmax}_{\theta} E_{(\mathbf{s}, \mathbf{a}) \sim p_\theta(\mathbf{s}, \mathbf{a})} [r(\mathbf{s}, \mathbf{a})] \quad (1.15)$$

Finite horizon case:

$$\theta^* = \operatorname{argmax}_{\theta} \sum_{t=1}^T E_{(\mathbf{s}_t, \mathbf{a}_t) \sim p_\theta(\mathbf{s}_t, \mathbf{a}_t)} [r(\mathbf{s}_t, \mathbf{a}_t)] \quad (1.16)$$

Let's talk about evaluating the reinforcement learning objective. First let's introduce a notational shorthand:

$$\theta^* = \operatorname{argmax}_{\theta} \underbrace{E_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]}_{J(\theta)} \quad (1.17)$$

We estimate  $J(\theta)$  by making rollouts from the policy (below  $i$  is the sample index and  $i, t$  is the  $t^{th}$  timestep in the  $i^{th}$  sample):

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (1.18)$$

Let's directly differentiate the policy. But first some more notational short-hands:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \underbrace{[r(\tau)]}_{\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t)} = \int p_\theta(\tau) r(\tau) d\tau \quad (1.19)$$

Now we start working on the derivative:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau \quad (1.20)$$

We'll need to use a convenient identity because we don't know  $p_\theta(\tau)$  (nor its gradient):

$$p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau) \quad (1.21)$$

So now:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = E_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p_\theta(\tau) r(\tau)] \quad (1.22)$$

THERE ARE MISTAKES BELOW, PLEASE COME BACK AND CORRECT THEM!!!!!!!

We can evaluate expectations with samples so we're on a good track. We can log  $p_\theta(\tau)$  on both sides of the equation and get a summation instead of a product. Let's see what we get from that:

$$\nabla_\theta \log p_\theta(\tau) r(\tau) = \nabla_\theta \left[ \cancel{\log p(\mathbf{s}_1)} + \sum_{t=1}^T \log \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) + \cancel{\log p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)} \right] \quad (1.23)$$

And now what's left is:

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[ \left( \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) \right) \left( \sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right] \quad (1.24)$$

To evaluate the policy gradient we can sample:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left( \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left( \sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \quad (1.25)$$

Once we have the gradient we can do a step of gradient ascent and we good to go! This is the REINFORCE algorithm:

1. sample  $\{\tau^i\}$  from  $\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)$  (run policy)
2.  $\nabla_\theta J(\theta) \approx \sum_{i=1}^N \left( \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left( \sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$
3.  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

If you implement this as-is, it won't work (well). Let's discuss the algorithm a bit more. But first, even simpler:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^T \left( \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left( \sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \quad (1.26)$$

$$\approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log \pi_{\theta}(\tau_i) r(\tau_i) \quad (1.27)$$

Maximum likelihood:

$$\nabla_{\theta} J_{ML}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log \pi_{\theta}(\tau_i) \quad (1.28)$$

In practise, we have finite samples. We also get really high variance with rewards. Thus we need some strategy to lower the variance.

### 1.3.1 Reducing variance

**Causality** policy at time  $t'$  cannot affect reward at time  $t$  when  $t < t'$ . Our algorithm thus not use this fact. Let's make it use it. First let's rewrite the policy gradient (just used distributive property):

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \left( \sum_{t'=1}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) \quad (1.29)$$

Let's change the log-probability of the action at every time step, based on whether than action led to better actions in future, present and past. But the past rewards will have to average out to 0 because they don't matter for future rewards. So just sum from  $t'$  to  $T$  and make this unbiased:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \underbrace{\left( \sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right)}_{\text{"reward to go"}} \quad (1.30)$$

"Reward to go" refers to the same estimate as the Q-function! So we can write:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \hat{Q}_{i,t} \quad (1.31)$$

This will be further discussed later.

### Baselines

If the good actions yield positive rewards and the bad actions yield negative rewards, the policy gradient will decrease the probability of bad actions and



increase the probability of good actions. But what if all the rewards are positive? Then all actions' probabilities will be increased, only by different amounts. And that's not really what we want — we want to increase only the probability of good actions, and decrease the probability of bad actions. How do we do that if the rewards are all positive? The below is what we'd like:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log p_{\theta}(\tau) [r(\tau) - b] \quad (1.32)$$

$$b = \frac{1}{N} \sum_{i=1}^N r(\tau) \quad (1.33)$$

Here  $b$  is the average reward and thus we'd increase the probability of actions which are better than average. But are we allowed to do that? Well, one can show that subtracting a number will not change the gradient in expectation, but it will change its variance (so the estimator will be unbiased for any  $b$ ).

$$E [\nabla_{\theta} \log p_{\theta}(\tau) b] = \int p_{\theta} \nabla_{\theta} \log p_{\theta}(\tau) b d\tau \quad (1.34)$$

$$= \int \nabla_{\theta} p_{\theta}(\tau) b d\tau \quad (1.35)$$

$$= b \nabla_{\theta} \int p_{\theta}(\tau) b d\tau = b \nabla_{\theta} 1 = 0 \quad (1.36)$$

For a finite number of samples, it won't be 0 so it will alter the variance! Also, this is not a perfect baseline (it's good tho) . We will derive the perfect baseline for the knowledge gains, even though it's rarely used in practise.

$$\text{Var}[x] = E[x^2] - E[x]^2 \quad (1.37)$$

$$\nabla_{\theta} J(\theta) = E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) (r(\tau) - b)^2] - E_{\tau \sim p_{\theta}(\tau)} \underbrace{[\nabla_{\theta} \log p_{\theta}(\tau) r(\tau) - b]^2}_{\text{is just } E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) r(\tau)]} \quad (1.38)$$

$$\frac{d\text{Var}}{db} = \frac{d}{db} E[g(\tau)^2 (r(\tau) - b)^2] = \frac{d}{db} (E[g(\tau)^2 r(\tau)^2] - 2E[g(\tau)^2 r(\tau) b] + b^2 E[g(\tau)^2]) \quad (1.39)$$

$$= -2E[g(\tau)^2 r(\tau) b] + b^2 E[g(\tau)^2] = 0 \quad (1.40)$$

$$b = \frac{E[g(\tau)^2 r(\tau) b]}{E[g(\tau)^2]} \quad (1.41)$$

So this is the optimal  $b$  (the baseline which minimizes the variance). You'll have a different baseline for every parameter as this is just the expected reward, by weighted by gradient magnitudes.

### 1.3.2 Off-policy gradients

Let's first discuss why policy gradients are an on-policy method (the classic one in fact).

$$\nabla_{\theta} J(\theta) = \underbrace{E_{\tau \sim p_{\theta}(\tau)}}_{\text{this is the trouble!}} [\nabla_{\theta} p_{\theta}(\tau) r(\tau)] \quad (1.42)$$

We need samples according to  $\theta$  and hence we can't retain data from other policies, or even the previous versions of our own policy (we can't skip step 1 in the REINFORCE algorithm). Neural networks require small gradients ('cos they are nonlinear). So if generating samples is expensive, this will be bad (on the other hand, if they're not, this will be nice).

What if we don't have samples from  $p_{\theta}(\tau)$ , but we have let's say  $\bar{p}(\tau)$ . Well, we can use importance sampling.

#### Importance sampling

$$E_{x \sim p(x)}[f(x)] = \int p(x) f(x) dx \quad (1.43)$$

$$= \int \frac{q(x)}{q(x)} p(x) f(x) dx \quad (1.44)$$

$$= \int q(x) \frac{p(x)}{q(x)} f(x) dx \quad (1.45)$$

$$= E_{x \sim p(x)} \left[ \frac{p(x)}{q(x)} f(x) \right] \quad (1.46)$$

This is all exact (in expectation).

The importance-sampled version of the RL objective is then:

$$J(\theta) = E_{\tau \sim \bar{p}(\tau)} \left[ \frac{p_{\theta}(\tau)}{\bar{p}(\tau)} r(\tau) \right] \quad (1.47)$$

Let's write out the trajectory probability distribution and see what we get:

$$p_{\theta}(\tau) = p(\mathbf{s}_1) \prod_{t=1}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t) \quad (1.48)$$

$$\frac{p_{\theta}(\tau)}{\bar{p}(\tau)} = \frac{\cancel{p(\mathbf{s}_1)} \prod_{t=1}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \cancel{p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)}}{\cancel{p(\mathbf{s}_1)} \prod_{t=1}^T \bar{\pi}_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \cancel{p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)}} \quad (1.49)$$

$$= \frac{\prod_{t=1}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)}{\prod_{t=1}^T \bar{\pi}_{\theta}(\mathbf{a}_t | \mathbf{s}_t)} \quad (1.50)$$

Now we will derive the policy gradient with importance sampling. Let's do a quick recap of where we're at:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} J(\theta) \quad (1.51)$$

$$J(\theta) = E_{\tau \sim p_{\theta}(\tau)} [r(\tau)] \quad (1.52)$$

and we want:

$$J(\theta') = E_{\tau \sim p_\theta(\tau)} \left[ \frac{p_{\theta'}(\tau)}{p_\theta} r(\tau) \right] \quad (1.53)$$

$$\nabla_{\theta'} J(\theta') = E_{\tau \sim p_\theta(\tau)} \left[ \frac{\nabla_{\theta'} p_{\theta'}(\tau)}{p_\theta} r(\tau) \right] \quad (1.54)$$

$$= E_{\tau \sim p_\theta(\tau)} \left[ \frac{p_{\theta'}(\tau)}{p_\theta(\tau)} \nabla_{\theta'} \log p_{\theta'}(\tau) r(\tau) \right] \quad (1.55)$$

If you estimate locally, at  $\theta = \theta'$ :

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p_\theta(\tau) r(\tau)] \quad (1.56)$$

thus getting the same gradient. But if they're not the same:

$$\nabla_\theta J(\theta') = E_{\tau \sim p_\theta(\tau)} \left[ \frac{p_{\theta'}(\tau)}{p_\theta} \nabla_{\theta'}(\tau) r(\tau) \right] \text{ when } \theta \neq \theta' \quad (1.57)$$

$$= E_{\tau \sim p_\theta(\tau)} \left[ \left( \prod_{t=1}^T \frac{\pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t)}{\pi_\theta(\mathbf{a}_t | \mathbf{s}_t)} \right) \left( \sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t) \right) \left( \sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right] \quad (1.58)$$

$$= E_{\tau \sim p_\theta(\tau)} \left[ \sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{a}_t | \mathbf{s}_t) \left( \prod_{t'=1}^t \frac{\pi_{\theta'}(\mathbf{a}_{t'} | \mathbf{s}_{t'})}{\pi_\theta(\mathbf{a}_{t'} | \mathbf{s}_{t'})} \right) \left( \sum_{t'=1}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \left( \prod_{t''=t'}^t \frac{\pi_{\theta'}(\mathbf{a}_{t''} | \mathbf{s}_{t''})}{\pi_\theta(\mathbf{a}_{t''} | \mathbf{s}_{t''})} \right) \right) \right] \quad (1.59)$$

where for the last equality we used the fact that future actions don't affect the current weight.

If we ignore  $\prod_{t''=t}^{t'} \frac{\pi_{\theta'}(\mathbf{a}_{t''} | \mathbf{s}_{t''})}{\pi_\theta(\mathbf{a}_{t''} | \mathbf{s}_{t''})}$ , we get a policy iteration algorithm (will be covered later). Then we won't have gradient, but we'll still improve our policy.

The problem lies in  $\prod_{t'=1}^t \frac{\pi_{\theta'}(\mathbf{a}_{t'} | \mathbf{s}_{t'})}{\pi_\theta(\mathbf{a}_{t'} | \mathbf{s}_{t'})}$ . The reason is that it is exponential in  $T$ . Let's say that the importance weights are all less than 1 (totally plausible). Then their product will go to 0 exponentially fast and that's bad for numerical reasons. So let's write the objective a bit differently. The on-policy policy gradient is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t} \hat{Q}_{i,t}) \quad (1.60)$$

where  $(\mathbf{s}_{t,i}, \mathbf{a}_{t,i}) \sim \pi_\theta(\mathbf{s}_{t,i}, \mathbf{a}_{t,i})$  The a different Off-policy policy gradient would be:

$$\nabla_{\theta'} J(\theta') \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \frac{\pi_{\theta'}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})}{\pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})} \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \hat{Q}_{i,t} \quad (1.61)$$

Not useful 'cos you can't calculate probabilities of the marginals. But we can split it via chain rule and ignore the state marginals:

$$\nabla_{\theta'} J(\theta') \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \frac{\pi_{\theta'}(\mathbf{s}_{i,t})}{\pi_{\theta}(\mathbf{s}_{i,t})} \frac{\pi_{\theta'}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})}{\pi_{\theta}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})} \nabla_{\theta'} \log \pi_{\theta'}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \hat{Q}_{i,t} \quad (1.62)$$

This does not in general give the correct policy gradient, but its reasonable in the sense that it gives bounded error if  $\pi_{\theta'}$  is not too different from  $\pi_{\theta}$ . But that will be discussed later.

### Policy gradient with automatic differentiation

We don't want to calculate the grad for every state-action pair 'cos neural nets have a lot of parameters. Typically we want to use the backpropagation algorithm. Thus we need to set our computational graph so that its gradient is the policy gradient. So we'll implement a "pseudo-loss" as a weighted maximum likelihood:

$$\tilde{J}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \hat{Q}_{i,t} \quad (1.63)$$

This equation means nothing, but it will give us the gradient that we want (lol).

### Policy gradients in practice

- gradient has high variance, so use very large batch sizes (in the thousands)
- tweaking learning rates is very hard (ADAM can be OK-ish), we'll do specific stuff on this later.

### 1.3.3 Advanced policy gradients

(There will be more on this later (even more advanced policy gradients (lol))). We have the following problem: some parameters change probabilities a lot more than others! We'd like to increase the changes made by parameters that make small changes, and decrease the effect of the parameters which make the larger changes. To see why this is necessary, imagine a vector field which does not point directly to the goal because a certain direction is too dominant. This problem is also similar to that of poor-performing gradient descent — the one which goes zig-zag instead of going straight to the goal. In short, we're dealing with a common problem in optimization.

The idea is to rescale the gradient so that that doesn't happen. So instead of doing

$$\theta' \leftarrow \operatorname{argmax}_{\theta'} (\theta' - \theta)^T \nabla_{\theta} J(\theta) \text{ s.t. } \|\theta' - \theta\|^2 \leq \epsilon \quad (1.64)$$

we can do

$$\theta' \leftarrow \operatorname{argmax}_{\theta'} (\theta' - \theta)^T \nabla_{\theta} J(\theta) \text{ s.t. } D(\pi_{\theta'}, \pi_{\theta}) \leq \epsilon \quad (1.65)$$

where  $D(\pi_{\theta'}, \pi_{\theta})$  is the parametrization-independent divergence measure. usually the KL-divergence:

$$D_{KL}(\pi_{\theta'} || \pi_{\theta}) = E_{\pi_{\theta'}} [\log \pi_{\theta} - \log \pi_{\theta'}] \approx (\theta' - \theta)^T \mathbf{F} (\theta' - \theta) \quad (1.66)$$

where  $\mathbf{F}$  is the Fisher-information matrix which can be estimated with samples:

$$\mathbf{F} = E_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(\mathbf{a}|\mathbf{s}) \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}|\mathbf{s})^T] \quad (1.67)$$

So for the natural gradient pick  $\alpha$ . For trust region policy optimization pick  $\epsilon$ . Then solve for optimal  $\alpha$  while solving  $\mathbf{F}^{-1} \nabla_{\theta} J(\theta)$ . Here conjugate gradient works well.

## 1.4 Actor-critic algorithms

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \underbrace{\left( \sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right)}_{\substack{\text{"reward to go"} \\ \hat{Q}_{i,t}}} \quad (1.68)$$

$\hat{Q}_{i,t}$  estimates the expected reward if we take  $\mathbf{a}_{i,t}$  in state  $\mathbf{s}_{i,t}$ . Can we get a better estimate? This is just a single-run Monte-Carlo estimate. Could we get the full expectation? In math, can we replace  $\hat{Q}_{i,t} \approx \sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})$  with  $\hat{Q}_{i,t} \approx \sum_{t'=t}^T E_{\pi_{\theta}} [r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) | \mathbf{s}_{i,t}, \mathbf{a}_{i,t}]$ ?

Having the correct full expectation (the correct Q-function), we'd have much lower variance policy gradient. We can also apply a baseline to this:

$$Q(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_{\theta}} [r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) | \mathbf{s}_t, \mathbf{a}_t] \text{ true expected reward-to-go} \quad (1.69)$$

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) (Q(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) - b) \quad (1.70)$$

$$b_t = \frac{1}{N} \sum_i Q(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (1.71)$$

If we make the baseline depend on the action, that will lead to bias. But it can depend on the state. So we can use

$$V(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi_{\theta}(\mathbf{s}_t, \mathbf{a}_t)} [Q(\mathbf{s}_t, \mathbf{a}_t)] \quad (1.72)$$

Then we can subtract the value function from the Q-value and we get an estimate of how much an action is better than the average. This difference is so

important that we call it the **advantage function**. So,

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t] \text{ total reward from } \mathbf{a}_t \text{ in } \mathbf{s}_t \quad (1.73)$$

$$V^\pi(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t | \mathbf{s}_t)} [Q^\pi(\mathbf{s}_t, \mathbf{a}_t)] \text{ total reward from } \mathbf{s}_t \quad (1.74)$$

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) = Q^\pi(\mathbf{s}_t, \mathbf{a}_t) - V^\pi(\mathbf{s}_t) \text{ how much better } \mathbf{a}_t \text{ is} \quad (1.75)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) A^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (1.76)$$

The better the estimate of the advantage, the lower the variance will be. However, since it is only approximate, it will introduce a bias. But we're OK with this tradeoff. To repeat, the below is the unbiased, but high variance single-sample estimate.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( \sum_{t'=1}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) - b \right) \quad (1.77)$$

But should we fit  $Q^\pi$ ,  $V^\pi$  or  $A^\pi$ ? One option:

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \underbrace{\sum_{t'=t+1}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t]}_{V^\pi(\mathbf{s}_{t+1})} \quad (1.78)$$

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + E_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)} [V^\pi(\mathbf{s}_{t+1})] \quad (1.79)$$

Another option:

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1}) \quad (1.80)$$

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1}) - V^\pi(\mathbf{s}_t) \quad (1.81)$$

We like the second option because we need to learn  $V^\pi(\mathbf{s})$  because it depends only on the state. Since there are less states than state-actions, it should be easier to learn. There are methods which go for option 1, but we'll discuss those later.

OK, how do we learn  $V^\pi(\mathbf{s})$  (it will be a neural net ofc). We need to evaluate the policy.

#### 1.4.1 Policy evaluation

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t] \quad (1.82)$$

$$J(\theta) = E_{\mathbf{s}_1 \sim p(\mathbf{s}_1)} [V^\pi(\mathbf{s}_1)] \quad (1.83)$$

How can we perform policy evaluation? Use Monte Carlo policy evaluation (this is what policy gradient does), i.e.

$$V^\pi(\mathbf{s}_t) \approx \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \quad (1.84)$$

$$V^\pi(\mathbf{s}_t) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \quad (1.85)$$

Unfortunately, we can't do the second thing in general (as you'd need to reset the simulator and obtain another trajectory from that state (and in general we can only reset to the initial state)). Fortunately, if we use a neural network to fit the value function, the network will generalize between similar states — similar states will have similar values. This is especially cool when we're working in continuous settings. So  $V^\pi(\mathbf{s}_t) \approx \sum_{t'=t}^T r(\mathbf{s}_{t'}, \mathbf{a}_{t'})$  will still be pretty good.

Thus we do the following: we run the policy and get the training data:

$$\left\{ \left( \mathbf{s}_{i,t}, \underbrace{\sum_{t'=t}^T r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})}_{y_{i,t}} \right) \right\} \quad (1.86)$$

We then do supervised regression:

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi(\mathbf{s}_i) - y_i\|^2 \quad (1.87)$$

But can we do even better (here we substitute the reward-to-go from the  $\mathbf{s}_{t+1}$  with the appropriate value function):

$$\text{ideal target } y_{i,t} = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_{i,t}] + V^\pi(\mathbf{s}_{i,t+1}) \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1}) \quad (1.88)$$

Thus we get a bootstrapped estimate. Our training data becomes:

$$\left\{ \left( \mathbf{s}_{i,t}, \underbrace{r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}_\phi^\pi(\mathbf{s}_{i,t+1})}_{y_{i,t}} \right) \right\} \quad (1.89)$$

We then again do supervised regression:

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi(\mathbf{s}_i) - y_i\|^2 \quad (1.90)$$

So again we have lower variance and higher bias (because  $\hat{V}_\phi^\pi$  can (will) be incorrect).

The value functions are very intuitive. For example, in board games, it tells you how likely you are to win in a given board state. Also, in this particular example it is very easy to restart from a given board state and get better estimates for the value function in that state.

### 1.4.2 From evaluation to actor-critic

Basic example actor-critic algorithm:

1. sample  $\{\mathbf{s}_i, \mathbf{a}_i\}$  from  $\pi_\theta(\mathbf{a}|\mathbf{s})$  (run policy)
2. fit  $\hat{V}_\theta^\pi(\mathbf{s})$  to sampled reward sums
3. evaluate  $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}_\theta^\pi(\mathbf{s}'_i) - \hat{V}_\theta^\pi(\mathbf{s}_i)$
4.  $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5.  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

### 1.4.3 Aside: discount factors

In infinite-episode length the value-function can get infinitely large. So we'll discount the reward from states with  $\gamma, \gamma \in [0, 1]$ ,

$$y_{i,t} \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \gamma \hat{V}_\theta^\pi(\mathbf{s}_{i,t+1}) \quad (1.91)$$

Can we do the same for (Monte Carlo) policy gradients?:

$$\text{option 1: } \nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( \sum_{t'=t}^T \gamma^{t'-t} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) \quad (1.92)$$

$$\text{option 2: } \nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left( \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \left( \sum_{t=1}^T \gamma^{t-1} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \quad (1.93)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( \sum_{t'=t}^T \gamma^{t'-1} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) \quad (1.94)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \gamma^{t-1} \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( \sum_{t'=t}^T \gamma^{t'-t} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) \quad (1.95)$$

So the second option also discounts the importance of a decision in later steps (i.e. it discounts future gradients as well), which makes it more correct if we want to do discounts. But do we want the later steps to matter less? In practise we use option 1 more often because we don't really want the discounted problem,



we just want to use the discount to get finite values for our value functions. That also makes our variance smaller. We actually want the average reward, but that's impractical and that's why we use the discount factor.

Let's now create an online actor-critic algorithm:

1. take action  $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$ , get  $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
2. update  $\hat{V}_\theta^\pi$  using target  $r + \gamma \hat{V}_\theta^\pi(\mathbf{s}')$
3. evaluate  $\hat{A}^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \hat{V}_\theta^\pi(\mathbf{s}') - \hat{V}_\theta^\pi(\mathbf{s})$
4.  $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \hat{A}^\pi(\mathbf{s}, \mathbf{a})$
5.  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

#### 1.4.4 Actor-critic design choices

We can do a two network design: one for the value function  $\mathbf{s} \rightarrow \hat{V}_\theta^\pi(\mathbf{s})$  and one for the policy  $\mathbf{s} \rightarrow \pi_\theta(\mathbf{a}|\mathbf{s})$ . The good thing about this is simple and stable. The bad thing is that it has no shared features between the actor and the critic. Alternatively, you can go for the shared network design (have a single network for both). It will probably need more hyperparameter tuning, but it is in principle more efficient.

#### 1.4.5 Online actor-critic in practise

In practice (due to the properties of neural networks) we want to update with batches and not do a single sample gradient. One way to get a batch is to use multiple workers, i.e. do the synchronized parallel actor-critic. This way you'll get n\_workers-sized batches. The alternative is to do the asynchronous parallel actor-critic. In general you'll get samples from different actors with approach (there is some lag in different threads). This makes it mathematically incorrect, but in practise this leads to overall performance benefits (because the actors are not that different, because the lag is not so large (all workers are running the same program after all (if they don't hang up that is lol))).

Cool. But it could be even better to use an off-policy actor-critic. However, to do so we need to modify the algorithm. We'd do this:

1. take action  $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$ , get  $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$ , store in  $\mathcal{R}$  (replay buffer)
2. sample a batch  $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$  from buffer  $\mathcal{R}$
3. update  $\hat{V}_\theta^\pi$  using target  $y_i = r_i + \gamma \hat{V}_\theta^\pi(\mathbf{s}'_i)$
4. evaluate  $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}_\theta^\pi(\mathbf{s}'_i) - \hat{V}_\theta^\pi(\mathbf{s}_i)$
5.  $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
6.  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

where  $\mathcal{L}(\phi) = \frac{1}{N} \sum_i \|\hat{V}_\theta^\pi(\mathbf{s}_i) - y_i\|^2$ .

Unfortunately, this algorithm is broken! Firstly,  $y_i = r_i + \gamma \hat{V}_\theta^\pi(\mathbf{s}'_i)$  will not give you the target value of the current actor, but a past actor:  $\mathbf{a}_i$  did not come from  $\pi_\theta$  and therefore  $\mathbf{s}'_i$  didn't either. Likewise, the policy gradient  $\nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)$  is also wrong for the same reason. To solve this, we could use importance sampling (or something else (soon...)). Let's first fix the value function. Well, the value function tells us the expected reward if we start in state  $\mathbf{s}_t$  and then follow the policy  $\pi$  onward, the Q-function tells you the expected reward if you start in state  $\mathbf{s}_t$  and take action  $\mathbf{a}_t$  and then follow the policy  $\pi$ . Notice that in the Q-function it doesn't matter if  $\mathbf{a}_t$  was taken from policy  $\pi$ . Thus it is valid for any action, it's just that in all subsequent steps  $\pi$  needs to be followed. So to solve the problem, we'll learn  $Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$  instead of  $V^\pi(\mathbf{s}_t)$ . We do this by updating  $\hat{Q}_\phi^\pi$  using the targets  $y_i = r_i + \gamma \hat{V}_\theta^\pi(\mathbf{s}') \forall \mathbf{s}_i, \mathbf{a}_i$ . We still need  $\hat{V}$  for the target values however. But we can use:

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T E_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t] = E_{\mathbf{a} \sim \pi(\mathbf{a}_t | \mathbf{s}_t)} [Q(\mathbf{s}_t, \mathbf{a}_t)] \quad (1.96)$$

Now we can update  $\hat{Q}_\phi^\pi$  using

$$y_i = r_i + \gamma \hat{V}_\theta^\pi(\mathbf{s}') \forall \mathbf{s}_i, \mathbf{a}_i \quad (1.97)$$

$$= r_i + \gamma \hat{Q}_\phi^\pi(\mathbf{s}'_i, \underbrace{\mathbf{a}'_i}_{\substack{\text{not from replay buffer } \mathcal{R} \\ \mathbf{a}'_i \sim \pi_\theta(\mathbf{a}'_i | \mathbf{s}'_i)}}) \quad (1.98)$$

This works because you don't need to interact with the simulator to ask which action your current network would have taken if it found itself in this (old) state (even though it never got there itself).

Now we'll deal with the policy gradient and we'll do the same trick, but for  $\mathbf{a}_i$  instead of  $\mathbf{a}'_i$ . Thus we'll sample  $\mathbf{a}_i^\pi \sim \pi_\theta(\mathbf{a}|\mathbf{s}_i)$  and get the following gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i^\pi | \mathbf{s}_i) \hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i^\pi) \quad (1.99)$$

where  $\mathbf{a}_i^\pi$  is not from the replay buffer  $\mathcal{B}$ . But in practice we don't actually use advantages:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i^\pi | \mathbf{s}_i) \hat{Q}^\pi(\mathbf{s}_i, \mathbf{a}_i^\pi) \quad (1.100)$$

This will lead to higher variance, but we don't really care because we don't need to interact the simulator and we can thus lower the variance by generating more samples (just run the network a few more times, no need for more state).

There are still problems with the current version of our off-policy actor-critic algorithm. Namely,  $\mathbf{s}_i$  didn't come from  $p_\theta(\mathbf{s})$ . Unfortunately, we can't do anything about this. Fortunately, we'll get an optimal policy on a broader distribution. Yes, it will be more work due to the higher variance, but the final result will be better. So in total we're left with:

1. take action  $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$ , get  $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$ , store in  $\mathcal{R}$  (replay buffer)
2. sample a batch  $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$  from buffer  $\mathcal{R}$
3. update  $\hat{Q}_\theta^\pi$  using target  $y_i = r_i + \gamma \hat{Q}_\theta^\pi(\mathbf{s}'_i, \mathbf{a}'_i) \forall \mathbf{s}_i, \mathbf{a}_i$
4.  $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i^\pi | \mathbf{s}_i) \hat{Q}^\pi(\mathbf{s}_i, \mathbf{a}_i^\pi)$ , where  $\mathbf{a}_i^\pi \sim \pi_\theta(\mathbf{a}|\mathbf{s}_i)$
5.  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

In practise, people use the reparametrization trick in the gradient estimate and get a better estimate with it. Furthermore, there are a lot of fancier ways to fit Q-functions (for example soft actor-critic (SAC)).

### 1.4.6 Critics as state-dependent baselines

Let's first restate the actor-critic policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \gamma \hat{V}_\theta^\pi(\mathbf{s}_{i,t+1}) - \hat{V}_\theta^\pi(\mathbf{s}_{i,t}) \right) \quad (1.101)$$

and the policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( \left( \sum_{t'=t}^T \gamma^{t'-t} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) - b \right) \quad (1.102)$$

More recap: the actor-critic policy gradient has much lower variance (due to the critic), but it is biased (if the critic is not perfect). On the other hand, the policy gradient has no bias, but it has high variance (because it uses a single-sample estimate). Now a question: can we have an unbiased policy gradient and still use the critic to reduce the variance? The way to do this is to use a state-dependent baseline, namely:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \left( \left( \sum_{t'=t}^T \gamma^{t'-t} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) - \hat{V}_\theta^\pi(\mathbf{s}_{i,t}) \right) \quad (1.103)$$

Exercise: use a previous proof to derive this (will do such things when I circle back to this when I do Sutton's book). Anyway, this does not lower the variance as much as the actor-critic, but it's certainly substantially better than the vanilla policy gradient with a constant baseline. Next question: can we make the baseline depend on not just the state, but the action as well? Would that lead to even lower variance? Yes, but it is complicating life. State and action dependent baselines are sometimes referred to as "controlled variance" in the literature. So let's create the following advantage function estimate:

$$\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = \sum_{t=t}^{\infty} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) - V_\theta^\pi(\mathbf{s}_t) \quad (1.104)$$

This has no bias and higher variance due to the single-sample estimate. We could try:

$$\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = \sum_{t=t}^{\infty} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) - Q_\theta^\pi(\mathbf{s}_t, \mathbf{a}_t) \quad (1.105)$$

This goes to 0 in expectation if the critic is correct, but the critic is not correct. If we incorporate both the state and action dependency and also account for the error we get:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \left( \hat{Q}_{i,t} - Q_\phi^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) + \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta E_{\mathbf{a} \sim \pi_\theta(\mathbf{a}_t | \mathbf{s}_{i,t})} [Q_\phi^\pi(\mathbf{s}_{i,t}, \mathbf{a}_t)] \quad (1.106)$$

This is a valid estimate for the policy gradient. It is much better in some cases, providing you can evaluate the second term in the expression.

Let's cook up some more options with different tradeoffs.

### Eligibility traces and n-step returns

Thus far we've had

$$\hat{A}_C^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \hat{V}_\theta^\pi(\mathbf{s}_{t+1}) - \hat{V}_\theta^\pi(\mathbf{s}_t) \quad (1.107)$$

which had lower variance and higher bias, and we've had the Monte Carlo advantage estimate:

$$\hat{A}_{MC}^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) - \hat{V}_\theta^\pi(\mathbf{s}_t) \quad (1.108)$$

which had no bias and higher variance.

So we've used the information about the next step only  $\hat{A}_C^\pi$  and information about the whole trajectory  $\hat{A}_{MC}^\pi$ . Can we do something in between (like 5 timesteps)? Here note that the variance between nearby timesteps will be smaller than those which are far away. Thus it makes sense to cut off with the value estimate after some n number of timesteps after the current state. This is called the n-step return estimator:

$$\hat{A}_n^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) - \hat{V}_\theta^\pi(\mathbf{s}_t) + \gamma^n \hat{V}_\theta^\pi(\mathbf{s}_{t+n}) \quad (1.109)$$

Using  $n > 1$  often works better! Actually, in most cases the sweet spot is somewhere between 1 and  $\infty$ .

Let's do one more trick:

### Generalized advantage estimation (GAE)

How about we construct all possible n-step return estimators and average them together?:

$$\hat{A}_{GAE}^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{n=1}^{\infty} w_n \hat{A}_n^\pi(\mathbf{s}_t, \mathbf{a}_t) \quad (1.110)$$

where  $w_n \propto \lambda^{n-1}$  is the exponential falloff. Here i'm skipping writing the above eq out (one boring eq) and will just provide the reduced form:

$$\hat{A}_{GAE}^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{n=1}^{\infty} (\gamma \lambda)^{t'-t} \delta_{t'} \quad (1.111)$$

where  $\delta_{t'} = r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) + \gamma \hat{V}_\theta^\pi(\mathbf{s}_{t'+1}) - \hat{V}_\theta^\pi(\mathbf{s}_{t'})$  Here larger  $\lambda$  looks further in the future and vice-versa. This has a similar effect as a discount.

## 1.5 Value function methods

### Can we omit policy gradient completely?

We have  $A^\pi(\mathbf{s}_t, \mathbf{a}_t)$ . It tells us how much better  $\mathbf{a}_t$  is than the average action according to  $\pi$  and it is at least as good as any  $\mathbf{a}_t \sim \pi(\mathbf{a}_t | \mathbf{s}_t)$  So let's just use  $\operatorname{argmax}_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t)$ , which gives the best action from  $\mathbf{s}_t$  if we then follow  $\pi$ :

$$\pi'(\mathbf{s}_t | \mathbf{a}_t) = \begin{cases} 1 & \text{if } \mathbf{a}_t = \operatorname{argmax}_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 & \text{otherwise} \end{cases} \quad (1.112)$$

So the policy is this implicit argmax policy (does not require a neural net to generate actions) and we know how to improve it. This is the idea behind:

### 1.5.1 Policy iteration

On a high level the policy iteration algorithm is:

1. evaluate  $A^\pi(\mathbf{s}, \mathbf{a})$
2. set  $\pi \leftarrow \pi'$

Now we need to figure out how to evaluate  $A^\pi(\mathbf{s}, \mathbf{a})$  (and whether we'll fit  $Q^\pi$  or  $V^\pi$ ).

### Dynamic programming

Skipping explaining this from a single Sergey slide, Sutton did it better. Plus even then I can only pretend to know the full depth. So let's just get to how we use it for policy iteration. We're in the tabular setting. For now the only point is that it gives the bootstrapped update:

$$V^\pi(\mathbf{s}) \leftarrow E_{\mathbf{a} \sim \pi(\mathbf{a} | \mathbf{s})} [r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}' | \mathbf{a}, \mathbf{s})} [V^\pi(\mathbf{s}')] ] \quad (1.113)$$

which we can then use to calculate the advantage  $A^\pi(\mathbf{s}_t, \mathbf{a}_t)$  and update the policy.

### Policy iteration with dynamic programming

We evaluate  $V^\pi(\mathbf{s})$  by doing

$$V^\pi(\mathbf{s}) \leftarrow r(\mathbf{s}, \pi(\mathbf{s})) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}' | \mathbf{s}, \pi(\mathbf{s}))} [V^\pi(\mathbf{s}')] \quad (1.114)$$

### Even simpler dynamic programming

Looking at the argmax of the advantage function (specifically looking at what's relevant in the argmax):

$$A^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma E[V^\pi(\mathbf{s}')] - V^\pi(\mathbf{s}) \quad (1.115)$$

$$\operatorname{argmax}_{\mathbf{a}_t} A^\pi(\mathbf{s}_t, \mathbf{a}_t) = \operatorname{argmax}_{\mathbf{a}_t} Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \quad (1.116)$$

$$Q^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma E[V^\pi(\mathbf{s}')] \quad (1.117)$$

So we can skip the policy and compute the values directly! With this we get the value iteration algorithm:

1. set  $Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E[V(\mathbf{s}')]$
2. set  $V(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$

You can even plug step 2 into step 1 lel. Again, this is simpler because we don't have to recover the indices — no need to do the whole table lookup, just do the max.

### Fitted value iteration and Q-iteration

Now we're using function approximators instead of tables to map states to values. This is done to combat the curse of dimensionality. We'll do least-squares regression on the target values (which are  $\operatorname{argmax}_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$ ). Then the fitted value iteration algorithm is:

1. set  $\mathbf{y}_i \leftarrow \max_{\mathbf{a}_i} (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)])$
2. set  $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i \|V_\phi(\mathbf{s}_i) - \mathbf{y}_i\|^2$

The problem is that we're required to know the transition dynamics: in step 1, we need to evaluate the expectation, but also be able to try out different actions in a state, which we can't do in general.

Let's replace  $V^\pi$  with  $Q^\pi$  in policy evaluation, getting:

$$Q^\pi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s}, \mathbf{a})} [Q^\pi(\mathbf{s}', \pi(\mathbf{s}'))] \quad (1.118)$$

Now we fit  $Q^\pi(\mathbf{s}, \mathbf{a})$  by sampling  $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$  and we don't need to know the transition dynamics. But now we need to simplify policy iteration to value iteration again (via the "max" trick).

Our current Q iteration algorithm looks like this:

1. set  $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)]$
2. set  $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

where we'll approximate the expectation  $E[V(\mathbf{s}'_i)] \approx \max_{\mathbf{a}'} Q_\phi(\mathbf{s}_i, \mathbf{a}_i)$ . This doesn't require simulation of actions, only the acquired samples. It works even for off-policy samples (unlike actor-critic). There's only one network (the Q-function estimator). Unfortunately, there are no convergence guarantees for non-linear function approximation (lmao).

We're now able to give the full fitted Q-iteration algorithm:

1. collect dataset  $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$  using policy
2. set  $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set  $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

The simplest way to design a Q network is to input both states and actions and to output a single scalar value. A common design for Q networks in discrete spaces is to input the state  $\mathbf{s}$  and output Q values for every possible action. The parameters here are the dataset size  $N$ , the collection policy, the number of iterations  $K$  (how much you go from step 3 back to step 2) the number of gradient steps  $S$ .

### 1.5.2 From Q-iteration to Q-learning

#### Why is this algorithm off-policy

The one place where the policy is used is when using the Q-function (in step 2 in the algorithm under the max). The Q-function functions as kind of a model which tells us which actions will do what (in terms of reward). Really you have a dataset of transitions and you're fitting your Q-function on it. Let's write out the error in step 3:

$$\mathcal{E} = \frac{1}{2} E_{(\mathbf{s}, \mathbf{a}) \sim \beta} \left[ \left( Q_\phi(\mathbf{s}, \mathbf{a}) - \left[ r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}') \right] \right)^2 \right] \quad (1.119)$$

if  $\mathcal{E} = 0$ , then  $Q_\phi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$ . This is an *optimal* Q-function, corresponding to optimal policy  $\pi'$ .

Let's write out a basic online on-policy Q-iteration algorithm:

1. take some action  $\mathbf{a}_i$  and observe  $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2.  $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3.  $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i) (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

where in step 3 we applied the chain rule in the arg. What policy to use here? In the end we'll just do the greedy policy. We don't want that while learning because it is deterministic and we'll forever be stuck using bad actions (bad exploration). One common choice is the classic **epsilon-greedy** policy:

$$\pi(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 - \epsilon & \text{if } \mathbf{a}_t = \operatorname{argmax}_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ \frac{\epsilon}{|\mathcal{A}| - 1} & \text{otherwise} \end{cases} \quad (1.120)$$

You can reduce  $\epsilon$  over time, thus getting more exploration early on, and nailing the best actions later. Another exploration rule is the **Boltzmann exploration** rule

$$\pi(\mathbf{a}_t | \mathbf{s}_t) \propto \exp(Q_\phi(\mathbf{s}_t, \mathbf{a}_t)) \quad (1.121)$$

Here there's a roughly same probability to take actions which are roughly equally good

### 1.5.3 Value function in theory

Let's discuss why there are no convergence guarantees. The value iteration (tabular) algorithm is:

1. set  $Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E[V(\mathbf{s}')] ]$
2. set  $V(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$

Let's define the Bellman operator:

$$\mathcal{B} : \mathcal{B}V = \max_{\mathbf{a}} r_{\mathbf{a}} + \gamma \mathcal{T}_{\mathbf{a}} V \quad (1.122)$$

where  $r_{\mathbf{a}}$  is the stacked vector of rewards at all states for action  $\mathbf{a}$ , and  $\mathcal{T}_{\mathbf{a}, i, j} = p(\mathbf{s}' = i | \mathbf{s} = j, \mathbf{a})$  is the matrix of transitions for the corresponding action  $\mathbf{a}$ . With this we've written the Bellman backup so that it looks like value iteration.

Now  $V^*$  is a *fixed point* of  $\mathcal{B}$ , meaning that if we recover it we get the optimal policy:

$$V^*(\mathbf{s}) = \max_{\mathbf{a}} r(\mathbf{s}, \mathbf{a}) + \gamma E[V^*(\mathbf{s}')], \text{ so } V^* = \mathcal{B}V^* \quad (1.123)$$

It's possible to show that  $V^*$  always exists, is unique and corresponds to the optimal policy. Will we reach it? (Yes) We can prove that  $\mathcal{B}$  is a *contraction* which means that for any  $V, \bar{V}$  we have:

$$\|\mathcal{B}V - \mathcal{B}\bar{V}\|_{\infty} \leq \underbrace{\gamma}_{\text{gap always gets smaller by } \gamma \text{ w.r.t. } \infty\text{-norm}} \|V - \bar{V}\|_{\infty} \quad (1.124)$$

Let's now check the fitted value iteration algorithm. To recap, it's

1. set  $\mathbf{y}_i \leftarrow \max_{\mathbf{a}_i} (r(\mathbf{s}_i, \mathbf{a}_i) + \gamma E[V_\phi(\mathbf{s}'_i)])$
2. set  $\phi \leftarrow \operatorname{argmin}_{\phi} \frac{1}{2} \sum_i \|V_\phi(\mathbf{s}_i) - \mathbf{y}_i\|^2$

Step 1. is just the definition of  $\mathcal{B}V$ . What does 2. do?

$$V' \leftarrow \arg \min_{V' \in \Omega} \frac{1}{2} \sum \|V'(\mathbf{s}) - (\mathcal{B}V)(\mathbf{s})\|^2 \quad (1.125)$$

where  $\Omega$  is the hypothesis space (in this case the space of all weights of our neural network architecture)  $V'$  will be a projection of  $\mathcal{B}V$  back to  $\Omega$ . Let's introduce an operator for this projection:

$$\Pi : \Pi V = \arg \min_{V' \in \Omega} \frac{1}{2} \sum \|V'(\mathbf{s}) - V(\mathbf{s})\|^2 \quad (1.126)$$

So the fitted value iteration algorithm is:



1.  $V \leftarrow \Pi \mathcal{B}V$

and here  $\mathcal{B}$  is a contraction (w.r.t.  $\infty$ -norm (“max” norm)),  $\Pi$  is a contraction w.r.t.  $l_2$ -norm (Euclidean distance), but  $\Pi \mathcal{B}$  is not a contraction of any kind!

Thus the sad conclusion is that fitted value iteration does not converge in general and it often does not converge in practise. In fitter Q-iteration, we get the same thing: define:

$$\mathcal{B} : \mathcal{B}Q = r + \gamma \mathcal{T} \max_{\mathbf{a}} Q \quad (1.127)$$

the operator:

$$\Pi : \Pi Q = \arg \min_{Q' \in \Omega} \frac{1}{2} \sum ||Q'(s, \mathbf{a}) - Q(s, \mathbf{a})||^2 \quad (1.128)$$

turn the algorithm into

1.  $Q \leftarrow \Pi \mathcal{B}Q$

and get that  $\mathcal{B}$  and  $\Pi$  are contractions (in the same spaces) and that  $\Pi \mathcal{B}$  is not a contraction of any kind. Of course, this also applies to Q-learning.

This is weird given how similar Q-learning is to gradient descent. But Q-learning is not gradient descent! That’s because:

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, \mathbf{a}_i) \left( Q_\phi(s_i, \mathbf{a}_i) - \underbrace{\left[ r(s_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(s'_i, \mathbf{a}'_i) \right]}_{\text{no gradient through target value}} \right) \quad (1.129)$$

the target Q-values themselves depend of Q-values. Now we could turn this algorithm into a gradient descent algorithm, but the resulting “residual algorithm” has very bad numerical properties and performs very poorly in practise.

### A sad corollary

The batch actor-critic algorithm is also not guaranteed to converge under function approximation :(

The reasons for this are the same.

Fortunately, we can actually make these algorithms work very well in practise (ML amirite). And now we’ll do that:

## 1.6 Deep RL with Q-functions

To recap look at 1.5.1 and 1.5.2 (NOTE: these links are bad as they don’t link to the enumerated algorithms, solving that is a TODO for later).

There’s another problem with the online Q-learning algorithm. The sequential states we observe are strongly correlated. Thus we are likely to overfit to

local transitions. This is made worse by the fact that the target value is always changing. So the algorithm is designed to overfit to what it has seen last and it doesn't really learn properly accross the entire state-action trajectory as it should. One practical way to mitigate this is to use multiple workers (running multiple simulators with our agent at the same time). This can be done in both the synchronized and the asynchronous fashion. But there is another solution: using replay buffers.

### Replay buffers

Q-learning with a replay buffer:

1. sample a batch  $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$  from  $\mathcal{B}$
2.  $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i) (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

The benefits: the samples are no longer correlated and there are multiple samples in the batch (low-variance gradient). How do we fill the replay buffer? We should be refilling it with new transitions because the initial batch of them are probably bad because they were collected with a bad policy (ex. epsilon-greedy on a freshly initialized Q-network). OK, now the full Q-learning with a replay buffer looks like:

1. collect dataset  $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$  using some policy, add it to  $\mathcal{B}$
2. sample a batch  $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$  in i.i.d. fashion from  $\mathcal{B}$
3.  $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i) (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

where we repeat going from 3. to 2. K times.

#### 1.6.1 Target networks

There is another problem we haven't tackled yet, in particular the fact that Q-learning is not gradient descent and it has a moving target which makes it very hard to converge. So what does Q-learning really have to do with regression?