Reinforcement learning notes

Marko Guberina

March 11, 2022

0.1 Plan

Old plan, look at update Let's stdenomart 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 our Sergey Levine's Berkley 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 insigths 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 intergrating 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 berkley 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
5. (optional) read papers Sergey recommended at the end of lectures and put notes on those here (there's plenty, start with most relevant ones)
6. (optional) go back to skipped lectures and watch them and make notes

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 $\mu_{t,i} = p(s_t = i), \mathcal{T}_{i,j} = p(s_{t+1} = i|s_t = j)$. Then $\overrightarrow{\mu}_t$ is a vector of probabilities and $\overrightarrow{\mu}_{t+1} = \mathcal{T}\overrightarrow{\mu}_t$
- 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 ($r(s_t, a_t)$), $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

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 ($r(s_t, a_t)$), $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

The goal of reinforcement learning

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

$$\underbrace{p_\theta(\boldsymbol{s}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{s}_T, \boldsymbol{a}_T)}_{p_\theta(\tau)} = p(\boldsymbol{s}_1) \prod_{t=1}^{T} \underbrace{\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)}_{\text{Markov chain on}(\boldsymbol{s},\boldsymbol{a})} \tag{1.1}$$

A bit more explicitely:

$$p((\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1})|(\boldsymbol{s}_t, \boldsymbol{a}_t)) = p((\boldsymbol{s}_{t+1}|(\boldsymbol{s}_t, \boldsymbol{a}_t))\pi_\theta(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1}) \tag{1.2}$$

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

$$\theta^\star = \operatorname*{argmax}_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{1.3}$$

$$= \operatorname*{argmax}_\theta \sum_t^T E_{(\boldsymbol{s}_t, \boldsymbol{a}_t) \sim p_\theta(\boldsymbol{s}_t, \boldsymbol{a}_t)} \left[r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{1.4}$$

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 $p(\boldsymbol{s}_t, \boldsymbol{a}_t)$ convege 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 $(\mathcal{T} - \boldsymbol{I})\mu = 0$. Here μ is the eingenvector 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 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. θ. We'll write it out recursively (by using the chain rule of probability), obtaining nested expectations:

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

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

Enter the Q-functions:

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

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

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

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

Definition: Q-function

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

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

Definition: value function

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

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

The connection between the 2 is the following:

$$V^\pi(\boldsymbol{s}_t) = E_{\boldsymbol{a}_t \sim \pi(\boldsymbol{s}_t, \boldsymbol{a}_t)} \left[Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{1.11}$$

And we can also write the RL objective as:

$$E_{\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1)} \left[V^\pi(\boldsymbol{s}_1) \right] \tag{1.12}$$

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

- set $\pi'(\boldsymbol{a}|\boldsymbol{s}) = 1$ is $\boldsymbol{a} = \text{argmax}_{\boldsymbol{a}} Q^\pi(\boldsymbol{s}, \boldsymbol{a})$

- this policy is at least as good as π and is probabily better (easily provable)
- it does not matter what π is, this is always true

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

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:

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

The right side is the chain rule of probabilities

The objective of reinforcement learning is:

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

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):

$$\theta^\star = \operatorname*{argmax}_\theta E_{(\boldsymbol{s}, \boldsymbol{a}) \sim p_\theta(\boldsymbol{s}, \boldsymbol{a})} \left[r(\boldsymbol{s}, \boldsymbol{a}) \right] \tag{1.15}$$

Finite horizon case:

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

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

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

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):

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

Let's directly differentiate the policy. But first some more notational shorthands:

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

Now we start working on the derivative:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau)\, d\tau \tag{1.20}$$

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

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

So now:

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

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:

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

And now what's left is:

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

To evaluate the policy gradient we can sample:

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

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 $\pi_\theta(\boldsymbol{a}_t | \boldsymbol{s}_t)$(run policy)
2. $\nabla_\theta J(\theta) \approx \sum_{i=1} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t} | \boldsymbol{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \right)$
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:

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

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

Maximum likelihood:

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

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):

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

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:

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

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

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

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:

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

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

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 substracting a number will not change the gradient in expectation, but it will change its variance (so the estimator will be unbiased for any b).

$$E\left[\nabla_\theta \log p_\theta(\tau) b\right] = \int p_\theta \nabla_\theta \log p_\theta(\tau) b \, d\tau \tag{1.34}$$

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

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

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.

$$\mathrm{Var}[x] = E[x^2] - E[x]^2 \tag{1.37}$$

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

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

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

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

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 weigthed 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).

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

We need samples according to θ 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

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

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

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

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

This is all exact (in expectation).

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

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

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

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

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

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

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

$$\theta^\star = \operatorname*{argmax}_\theta J(\theta) \tag{1.51}$$

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

and we want:

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

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

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

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

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

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

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

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

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

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

If we ignore $\prod_{t''=t}^{t'} \frac{\pi_{\theta'}(\boldsymbol{a}_{t''} | \boldsymbol{s}_{t''})}{\pi_\theta(\boldsymbol{a}_{t''} | \boldsymbol{s}_{t''})}$, 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 $\prod_{t'=1}^{t} \frac{\pi_{\theta'}(\boldsymbol{a}_{t'} | \boldsymbol{s}_{t'})}{\pi_\theta(\boldsymbol{a}_{t'} | \boldsymbol{s}_{t'})}$. 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:

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

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

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

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

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

This does not in general give the correct policy gradient, but its reasonable in the sense that it gives bounded error is $\pi_{\theta'}$ is no too different form π_θ. 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 weighet maximum likelihood:

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

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

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

we can do

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

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

$$D_{KL}(\pi_{\theta'}||\pi_\theta) = E_{\pi_{\theta'}}\left[\log \pi_\theta - \log \pi_{\theta'}\right] \approx (\theta' - \theta)^T \boldsymbol{F} (\theta' - \theta) \tag{1.66}$$

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

$$\boldsymbol{F} = E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(\boldsymbol{a}|\boldsymbol{s}) \nabla_\theta \log \pi_\theta(\boldsymbol{a}|\boldsymbol{s})^T\right] \tag{1.67}$$

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

1.4 Actor-critic algorithms

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

$\hat{Q}_{i,t}$ estimates the expected reward if we take $\boldsymbol{a}_{i,t}$ in state $\boldsymbol{s}_{i,t}$. 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 $\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t})$ with $\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'})|\boldsymbol{s}_t, \boldsymbol{a}_t\right]$?

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:

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

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

$$b_t = \frac{1}{N}\sum_{i} Q(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \tag{1.71}$$

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

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

Then we can substract 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,

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

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

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

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

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.

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

But should we fit Q^π, V^π or A^π? One option:

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

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

Another option:

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

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

We like the second option because we need to learn $V^\pi(\boldsymbol{s})$ 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 dicuss those later.

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

1.4.1 Policy evaluation

$$V^\pi(\boldsymbol{s}_t) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\boldsymbol{s}_t, \boldsymbol{a}_t)\boldsymbol{s}_t\right] \tag{1.82}$$

$$J(\theta) = E_{\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1}\left[V^\pi(\boldsymbol{s}_1)\right] \tag{1.83}$$

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

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

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

Unfortunatelly, 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 inital state)). Fortunatelly, 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 $V^{\pi}(\boldsymbol{s}_t) \approx \sum_{t'=t}^{T} r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'})$ will still be pretty good.

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

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

We then do supervised regresion:

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_{i} ||\hat{V}^{\pi}_{\phi}(\boldsymbol{s}_i) - y_i||^2 \tag{1.87}$$

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

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

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

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

We then again do supervised regresion:

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_{i} ||\hat{V}^{\pi}_{\phi}(\boldsymbol{s}_i) - y_i||^2 \tag{1.90}$$

So again we have lower variance and higher bias (because $\hat{V}^{\pi}_{\phi}$ 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 $\{\boldsymbol{s}_i, \boldsymbol{a}_i\}$ from $\pi_\theta(\boldsymbol{a}|\boldsymbol{s})$ (run policy)
2. fit $\hat{V}^\pi_\theta(\boldsymbol{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\boldsymbol{s}_i, \boldsymbol{a}_i) = r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \hat{V}^\pi_\theta(\boldsymbol{s}'_i) - \hat{V}^\pi_\theta(\boldsymbol{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\boldsymbol{a}_i|\boldsymbol{s}_i)\hat{A}^\pi(\boldsymbol{s}_i, \boldsymbol{a}_i)$
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 $\gamma, \gamma \in [0, 1]$,

$$y_{i,t} \approx r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) + \gamma\hat{V}^\pi_\theta(\boldsymbol{s}_{i,t+1}) \tag{1.91}$$

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

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

$$\text{option 2: } \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t})\right)\left(\sum_{t=1}^{T}\gamma^{t-1}r(\boldsymbol{s}_{i,t'}, \boldsymbol{a}_{i,t'})\right) \tag{1.93}$$

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

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

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

1.4.4 Actor-critic design choises

We can do a two network design: one for the value function $\boldsymbol{s} \to \hat{V}_\theta^\pi(\boldsymbol{s})$ and one for the policy $\boldsymbol{s} \to \pi_\theta(\boldsymbol{a}|\boldsymbol{s})$. 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 desing (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 works, 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 $\boldsymbol{a} \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s})$, get $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}, r)$, store in $\mathcal{R}$ (replay buffer)
2. sample a batch $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$ from buffer $\mathcal{R}$
3. update $\hat{V}_\theta^\pi$ using target $y_i = r_i + \gamma \hat{V}_\theta^\pi(\boldsymbol{s}'_i)$
4. evaluate $\hat{A}^\pi(\boldsymbol{s}_i, \boldsymbol{a}_i) = r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \gamma \hat{V}_\theta^\pi(\boldsymbol{s}'_i) - \hat{V}_\theta^\pi(\boldsymbol{s}_i)$
5. $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\boldsymbol{a}_i|\boldsymbol{s}_i) \hat{A}^\pi(\boldsymbol{s}_i, \boldsymbol{a}_i)$
6. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

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

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

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

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

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

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

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 $\boldsymbol{a}_i$ instead of $\boldsymbol{a}_i'$ Thus we'll sample $\boldsymbol{a}_i^\pi \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s}_i)$ and get the following gradient:

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

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

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

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, $\boldsymbol{s}_i$ didn't come from $p_\theta(\boldsymbol{s})$. Unfortunatelly, we can't do anything about this. Fortunatelly, 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 $\boldsymbol{a} \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s})$, get $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}, r)$, store in $\mathcal{R}$ (replay buffer)
2. sample a batch $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$ from buffer $\mathcal{R}$
3. update $\hat{Q}^\pi_\theta$ using target $y_i = r_i + \gamma \hat{Q}^\pi_\theta(\boldsymbol{s}'_i, \boldsymbol{a}'_i) \forall \boldsymbol{s}_i, \boldsymbol{a}_i$
4. $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\boldsymbol{a}^\pi_i|\boldsymbol{s}_i) \hat{Q}^\pi(\boldsymbol{s}_i, \boldsymbol{a}^\pi_i)$, where $\boldsymbol{a}^\pi_i \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s}_i)$
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:

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

and the policy gradient:

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

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 ubiased policy gradient and still use the critic to reduce the variance? The way to do this is to use a state-dependent baseline, namely:

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

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 variane as much as the actor-critic, but it's certainly substantially better than the vannila 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 refered to as "controlled variance" in the literature. So let's create the following advantige function estimate:

$$\hat{A}^\pi(\boldsymbol{s}_i, \boldsymbol{a}_i) = \sum_{t=t}^{\infty} \gamma^{t'-t} r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) - V^\pi_\theta(\boldsymbol{s}_t) \tag{1.104}$$

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

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

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:

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

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

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

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

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

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 sence 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:

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

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

Let's do one more trick:

Generalied advantage estimation (GAE)

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

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

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:

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

where $\delta_{t'} = r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) + \gamma \hat{V}^{\pi}_{\theta}(\boldsymbol{s}_{t'+1}) - \hat{V}^{\pi}_{\theta}(\boldsymbol{s}_{t'})$ Here larger λ 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 $A^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t)$. It tells us how much better $\boldsymbol{a}_t$ is than the average action according to π and it is at least as good as any $\boldsymbol{a}_t \sim \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$ So let's just use $\text{argmax}_{\boldsymbol{a}_t} A^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t)$, which gives the best action from $\boldsymbol{s}_t$ if we then follow π:

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

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

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

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:

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

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

Policy iteration with dynamic programming

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

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

Even simpler dynamic programming

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

$$A^\pi(\boldsymbol{s}, \boldsymbol{a}) = r(\boldsymbol{s}, \boldsymbol{a}) + \gamma E\left[V^\pi(\boldsymbol{s}')\right] - V^\pi(\boldsymbol{s}) \tag{1.115}$$

$$\underset{\boldsymbol{a}_t}{\operatorname{argmax}} A^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) = \underset{\boldsymbol{a}_t}{\operatorname{argmax}} Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{1.116}$$

$$Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = r(\boldsymbol{s}, \boldsymbol{a}) + \gamma E\left[V^\pi(\boldsymbol{s}')\right] \tag{1.117}$$

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

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

You can even plug step 2 into step 1 lel. Again, this is simpler because we don't have to recover the indeces — 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 $\operatorname{argmax}_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a})$). Then the fitted value iteration algorithm is:

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

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^π with Q^π in policy evaluation, getting:

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

Now we fit $Q^\pi(\boldsymbol{s}, \boldsymbol{a})$ by sampling $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')$ 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 $\boldsymbol{y}_i \leftarrow r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \gamma E\left[V_\phi(\boldsymbol{s}_i')\right]$
2. set $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i ||Q_\phi(\boldsymbol{s}_i, \boldsymbol{a}_i) - \boldsymbol{y}_i||^2$

where we'll approximate the expectation $E[V(\boldsymbol{s}_i')] \approx \max_{\boldsymbol{a}'} Q_\phi(\boldsymbol{s}_i, \boldsymbol{a}_i)$. 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). Unfortunatelly, there are no convegence guarantees for non-linear function approximation (lmao).

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

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

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 $\boldsymbol{s}$ 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:

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

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

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

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

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 what that while learning because it is deterministic and we'll forever be stuck using bad actions (bad exploration). One common choise is the classic **epsilon-greedy** policy:

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

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

$$\pi(\boldsymbol{a}_t|\boldsymbol{s}_t) \propto \exp\left(Q_\phi(\boldsymbol{s}_t, \boldsymbol{a}_t)\right) \tag{1.121}$$

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

1.5.3 Value function in theory

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

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

Let's define the Bellman operator:

$$\mathcal{B} : \mathcal{B}V = \max_{\boldsymbol{a}} r_{\boldsymbol{a}} + \gamma \mathcal{T}_{\boldsymbol{a}} V \tag{1.122}$$

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

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

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

It's possible to show that $V^\star$ 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:

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

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

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

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

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

where Ω 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 Ω. Let's introduce an operator for this projection:

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

So the fitter value iteration algorithm is:

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

and here $\mathcal{B}$ is a contraction (w.r.t. ∞-norm ("max" norm)), Π 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:

$$\mathcal{B} : \mathcal{B}Q = r + \gamma\mathcal{T}\max_{\boldsymbol{a}} Q \tag{1.127}$$

the operator:

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

turn the algorithm into

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

and get that $\mathcal{B}$ and Π 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:

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

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.

Fortunatelly, 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 $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)$ from $\mathcal{B}$
2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\boldsymbol{s}_i, \boldsymbol{a}_i)\left(Q_\phi(\boldsymbol{s}_i, \boldsymbol{a}_i) - [r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \gamma \max_{\boldsymbol{a}'} Q_\phi(\boldsymbol{s}'_i, \boldsymbol{a}'_i)]\right)$

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

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. Also training to convergence on a moving target is not really what we want anyway ('cos that leads to local overfitting). Let's do Q-learning with a replay buffer and a target network:

1. save target network parameters: $\phi' \leftarrow \phi$
2. collect dataset $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
3. sample a batch $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)$ in i.i.d. fashion from $\mathcal{B}$
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\boldsymbol{s}_i, \boldsymbol{a}_i)\left(Q_\phi(\boldsymbol{s}_i, \boldsymbol{a}_i) - [r(\boldsymbol{s}_i, \boldsymbol{a}_i) + \gamma \max_{\boldsymbol{a}'} Q_{\phi'}(\boldsymbol{s}'_i, \boldsymbol{a}'_i)]\right)$ and go back to previous step K times. after that go N times back to step 2. finally return to step 1.

Thus targets don't change in the inner loop. This makes steps 2.-4. into supervised regression. Some example back-of-the-envelope numbers are $K = 4$ and $N = 10000$. This algorithm is the "classic" deep Q-learning algorithm (DQN). It's really the above, but with $K = 1$. Let's write it out again, a bit clearer and with more ML-ly language:

1. take some action $\boldsymbol{a}_i$, observe $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)$ and add it to $\mathcal{B}$
2. sample a mini-batch $\left(\boldsymbol{s}_j, \boldsymbol{a}_j, \boldsymbol{s}'_j, r_j\right)$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\boldsymbol{a}'_j} Q_{\phi'}(\boldsymbol{s}'_j, \boldsymbol{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\boldsymbol{s}_j, \boldsymbol{a}_j)\left(Q_\phi(\boldsymbol{s}_i, \boldsymbol{a}_i) - y_j\right)$
5. update ϕ': copy ϕ every N steps

It is worth to experiment with alternative target networks. When we update $\phi' \leftarrow \phi$, we get the moving target problem again. It's not too bad because N is usually large, but it might make more sense to have the same lag all the time. The popular alternative is a variant of Polyak averaging:

$$\text{update } \phi' : \phi' \leftarrow \tau\phi' + (1 - \tau)\phi \tag{1.130}$$

where $\tau = 0.999$ works well. This feels bad because we're linearly interpolating neural network parameters (which are nonlinear function). It works because ϕ' and ϕ are similar and there are *some* theoretical justifications for this.

1.6.2 A general view of Q-learning

It's important to note that process in step 1 and process in step 3 are quite separate - you can run them in parallel and they don't really need to care about each other (but of course they shouldn't be too divergent because then Q-learning won't really work).

1.7 Improving Q-learning

Are Q-values accurate?

Q-values help us select a good policy, but they are also a prediction of future rewards. So do they predict Q-values accurately? [nice graphs on average returns and corresponding average Qs on Atari games where you can see that the Q values increase almost monotonically, but the average rewards are much noisier. However, both Q and average returns increase with training time.] But why are Q-values overestimating?

$$\text{target value } y_j = r_j + \gamma \underbrace{\max_{\boldsymbol{a}'_j} Q_{\phi'}(\boldsymbol{s}'_j, \boldsymbol{a}'_j)}_{\text{herein lies the problem}} \tag{1.131}$$

Let's explain this in simple terms. Image we have 2 random variables X_1 and X_2 and let's say they represent a true value plus some noise. Proveably,

$$E\left[\max(X_1, X_2)\right] \geq \max\left(E[X_1], E[X_2]\right) \tag{1.132}$$

The relation to Q-learning is the following. If we imagine that $Q_{\phi'}(\boldsymbol{s}', \boldsymbol{a}')$ is not perfect because it has added noise, we get exactly the situation in the inequality — the max over the actions and the expectation over it will lead to systematic overestimation. Thus $\max_{\boldsymbol{a}'} Q_{\phi'}(\boldsymbol{s}', \boldsymbol{a}')$ *overestimates* the next value. Note that $\max_{\boldsymbol{a}'} Q_{\phi'}(\boldsymbol{s}', \boldsymbol{a}') = Q_{\phi'}(\boldsymbol{s}', \text{argmax}_{\boldsymbol{a}'} Q_{\phi'}(\boldsymbol{s}', \boldsymbol{a}'))$. If we can somehow decorrelate the noise in the action selection mechanism and the noise in the value evaluation mechanism, this problem will go away (so let's not get both actions and values from $Q_{\phi'}$). This is done in:

1.7.1 Double Q-learning

Double Q-learning uses two networks:

$$Q_{\phi_A}(\boldsymbol{s}, \boldsymbol{a}) \leftarrow r + \gamma Q_{\phi_B}\left(\boldsymbol{s}', \underset{\boldsymbol{a}'}{\text{argmax}}\, Q_{\phi_A}(\boldsymbol{s}', \boldsymbol{a}')\right) \tag{1.133}$$

$$Q_{\phi_B}(\boldsymbol{s}, \boldsymbol{a}) \leftarrow r + \gamma Q_{\phi_A}\left(\boldsymbol{s}', \underset{\boldsymbol{a}'}{\text{argmax}}\, Q_{\phi_B}(\boldsymbol{s}', \boldsymbol{a}')\right) \tag{1.134}$$

if we assume that Q_{ϕ_A} and Q_{ϕ_B} are decorrelated, the noise will be different and we won't overestimate.

Double Q-learning in practise

We already have 2 networks, Q_ϕ and $Q_{\phi'}$! So in standard Q-learning we do:

$$y = r + \gamma Q_{\phi'}\left(\boldsymbol{s}', \underset{\boldsymbol{a}'}{\text{argmax}}\, Q_{\phi'}(\boldsymbol{s}', \boldsymbol{a}')\right) \tag{1.135}$$

and in double Q-learning we do:

$$y = r + \gamma Q_{\phi'}\left(\boldsymbol{s}', \underset{\boldsymbol{a}'}{\text{argmax}}\, Q_{\phi}(\boldsymbol{s}', \boldsymbol{a}')\right) \tag{1.136}$$

so we just use the current network (not the target network) to evaluate action and we still use the target network to evaluate value. Of course, Q_ϕ and $Q_{\phi'}$ are periodically set to be the same (and are not too different to begin with), so this is far from the perfect solution, but it works well in practise nonetheless.

Multi-step returns

The Q-learning target is:

$$y_{j,t} = r_{j,t} + \gamma Q_{\phi'}\left(\boldsymbol{s}', \arg\max_{\boldsymbol{a}_{j,t+1}} Q_{\phi'}(\boldsymbol{s}'_{j,t+1}, \boldsymbol{a}'_{j,t+1})\right) \tag{1.137}$$

Where does the signal come from? In the beginning, $Q_{\phi'}$ is bad so most signal comes from $r_{j,t}$ ($Q_{\phi'}$ is just additional noise). Later, it's mostly $Q_{\phi'}$ tho. Could we construct multi-step target like in actor critic (the Monte Carlo estimate)? Yes,

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{j,t'} + \gamma^N \max_{\boldsymbol{a}_{j,t+N}} Q_{\phi'}(\boldsymbol{s}_{j,t+N}, \boldsymbol{a}_{j,t+N}) \tag{1.138}$$

This is sometimes called the n-step return estimator and the tradeoff is the same as in actor-critic (you get lower bias and higher variance).

Q-learning with N-step returns

Less bias target values when Q-values are inaccurate, typically faster learning (especially early on), but only actually correct when learning on-policy (because you use action your new policy might not have taken).

How do we fix the issue? Ignore it (often works well (lmao ofc)), cut the trace (dynamically choose N to get only on-policy data), do importance sampling and the mystery solution where Q is conditioned on something else which Sergey says is homework.

1.7.2 Q-learning with continuous actions

How do we select continuous actions when we have to do the argmax to select the action? We also have to do the max to calculate the target values. Well, we can do optimization (e.g., SGD), or do stochastic optimization.

Option 1 Simple solution

$$\max_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a}) \approx \max \{Q(\boldsymbol{s}, \boldsymbol{a}_1), \ldots, Q(\boldsymbol{s}, \boldsymbol{a}_N)\} \tag{1.139}$$

where $(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_N)$ are sampled from some distribution (e.g. uniform). This is simple, efficiently parallelizable, but not very accurate. We can do the more accurate cross-entropy method (CEM) which is simple iterative stochastic optimization (it refines the distribution and then re-samples) This can also be fast. Or do CMA-ES which is substantially less simple iterative stochastic optimization but is more accurate. Anyhow CEM works OK for up to 40 dimensions of the actions space.

Option 2 Use a function class that is easy to optimize:

$$Q_\phi(\boldsymbol{s}, \boldsymbol{a}) = -\frac{1}{2}(\boldsymbol{a} - \mu_\phi(\boldsymbol{s}))^T P_\phi(\boldsymbol{s})(\boldsymbol{a} - \mu_\phi(\boldsymbol{s})) + V_\phi(\boldsymbol{s}) \tag{1.140}$$

Because for a given state the fuction is quadratic, we have a normalized advantage function (NAF):

$$\operatorname{argmax}_{\boldsymbol{a}} Q_\phi(\boldsymbol{s}, \boldsymbol{a}) = \mu_\phi(\boldsymbol{s}) \tag{1.141}$$

$$\max_{\boldsymbol{a}} Q_\phi(\boldsymbol{s}, \boldsymbol{a}) = V_\phi(\boldsymbol{s}) \tag{1.142}$$

With this there are no changes to the algorithm and it's just as efficient as Q-learning, but it loses reparametrizational power.

Option 3 We can also learn an approximate maximizer, i.e. learn another network to estimate the (arg)max. This method can be interpreted as a "deterministic" actor-critic, or as approximate Q-learning.

$$\max_{\boldsymbol{a}} = Q_\phi(\boldsymbol{s}, \boldsymbol{a}) = Q_\phi(\boldsymbol{s}, \operatorname*{argmax}_{\boldsymbol{a}} Q_\phi(\boldsymbol{s}, \boldsymbol{a})) \tag{1.143}$$

So train another network $\mu_\theta(\boldsymbol{s})$ such that $\mu_\theta(\boldsymbol{s}) \approx \operatorname{argmax}_{\boldsymbol{a}} Q_\phi(\boldsymbol{s}, \boldsymbol{a})$ How? Just solve $\theta \leftarrow \operatorname{argmax}_\theta Q_\theta(\boldsymbol{s}, \mu_\theta(\boldsymbol{s}))$. Then $\frac{dQ_\phi}{d\theta} = \frac{d\boldsymbol{a}}{d\theta}\frac{dQ_\phi}{d\boldsymbol{a}}$. The new target is then

$$y_j = r_j + \gamma Q_{\phi'}(\boldsymbol{s}'_j, \mu_\theta(\boldsymbol{s}'_j)) \approx r_j + \gamma Q_{\phi'}(\boldsymbol{s}'_j, \operatorname*{argmax}_{\boldsymbol{a}'} Q_{\phi'}(\boldsymbol{s}'_j, \boldsymbol{a}'_j)) \tag{1.144}$$

If we do this, we get DDPG

DDPG

1. take action $\boldsymbol{a}_i$ and observe $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample a mini-batch $\left(\boldsymbol{s}_j, \boldsymbol{a}_j, \boldsymbol{s}'_j, r_j\right)$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma Q_{\phi'}(\boldsymbol{s}'_j, \mu_\theta(\boldsymbol{s}'_j))$ using *target* networks $Q_{\phi'}$ and $\mu_{\theta'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\boldsymbol{s}_j, \boldsymbol{a}_j)\left(Q_\phi(\boldsymbol{s}_j, \boldsymbol{a}_j) - y_j\right)$
5. $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(\boldsymbol{s}_j, \mu(\boldsymbol{s}_j))$
6. update ϕ' and θ' (e.g. Polyak averaging)

1.7.3 Implementation tips and examples

Q-learning takes some care to stabilize — test on easy and reliable tasks first — you want to get through debugging first and then do the hyperparameter tuning. Also, Q-learning much differently on different tasks. Namely, there's a huge difference between stability. It can even happen that some runs works fine and other fail completely.

Tips Large replay buffers help improve stability (as it looks more like fitted Q-iteration). It takes time — it might be no better than random for a while. To remedy this somewhat, start with high exploration and gradually reduce it. Bellman error gradients can be big so clip gradients or use Huber loss:

$$L(x) = \begin{cases} \frac{x^2}{2} & \text{if } |x| \leq \delta \\ \delta|x| - \frac{\delta^2}{2} & \text{otherwise} \end{cases} \tag{1.145}$$

Double Q-learning helps a lot in practise and has no downsides. N-step returns help a lot (particularly in the beginning), but introduce bias. Reducing learning rates over time also help, Adam optimizer can help too. Also, it's very important to run different random seeds as the algorithm is quite inconsistent between runs.

1.8 Even more advanced policy gradients (PPO and TRPO)

Skipping this as it's not the highest priority at the moment, but will come back to it eventually (maybe). This is the entire lecture 9.

1.9 Optimal control and planning

I'll be mostly skipping this as it is mostly re-doing what I've written already.

The objective Can be expressed as a optimization problem:

$$\min_{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_T} \sum_{t=1}^{T} c(\boldsymbol{s}_t, \boldsymbol{a}_t) \text{ s.t. } \boldsymbol{s} = f(\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}) \tag{1.146}$$

Equivalently, in terms of rewards we get:

$$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T = \arg\max_{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_T} \sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t) \text{ s.t. } \boldsymbol{s}_{t+1} = f(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{1.147}$$

All good in the deterministic case, but what about the stochastic?

$$p_\theta\left(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_T | \boldsymbol{a}_1, \ldots, \boldsymbol{a}_T\right) = p(\boldsymbol{s}_1) \prod_{t=1}^{T} p(\boldsymbol{s}_{t+1} | (\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{1.148}$$

Now we do:

$$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T = \arg\max_{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_T} E\left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t) | \boldsymbol{a}_1, \ldots, \boldsymbol{a}_T\right] \tag{1.149}$$

However, this can be very suboptimal. Namely, open-loop planning in stochastic settings is horrible. Reinforcement learning typically solves things in a closed-loop fashion (it tells the agent what to do at every possible state, and it continuously observes states and acts on them as they come).

Stochastic optimization

Let's abstract away optimal control/planning (the optimization problem is a black box):

$$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T = \arg\max_{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_T} \underbrace{J(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T)}_{\text{don't care what this is}} \tag{1.150}$$

also let $\boldsymbol{A} = \boldsymbol{a}_1, \ldots, \boldsymbol{a}_T = \text{argmax}_{\boldsymbol{A}} = J(\boldsymbol{A})$. The simplest method is to guess and check:

1. pick $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_N$ from some distribution (e.g. uniform)

2. choose $\boldsymbol{A}_i$ based on $\text{argmax}_i\, J(\boldsymbol{A}_i)$

This is also called "random shooting method". In practise this can work well for small problems. The main benefit is that it is super simple. It is also quite fast to evaluate on modern hardware. The disadvantage is that you might not pick good actions (it's luck based after all).

A better way to do black-box optimization is

Cross-entropy method (CEM)

1. pick $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_N$ from some distribution (e.g. uniform)
2. choose $\boldsymbol{A}_i$ based on $\text{argmax}_i\, J(\boldsymbol{A}_i)$

In cross-entropy method, we'll be a bit smarter about picking the distribution. We'll do an iterative process of progressively refining the probability distribution from which we pick actions which we evaluate. So we'll generate some samples from a broad distribution, use the results they provide to create a new narower distribution which is centered around the best-performing samples from the previous step and then draw new samples from this distribution. We then repeat this process. With continuous action this would be:

1. sample $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_T$ from $p(\boldsymbol{A})$
2. evaluate $J(\boldsymbol{A}_1), \ldots, J(\boldsymbol{A}_N)$
3. pick the *elites* $\boldsymbol{A}_{i_1}, \ldots, \boldsymbol{A}_{i_M}$ with the highest value, where $M < N$
4. refit $p(\boldsymbol{A})$ to the elites $\boldsymbol{A}_{i_1}, \ldots, \boldsymbol{A}_{i_M}$ and go back to 1.

This method has a number of nice properties: it guarantees to find the optimum (provided enough samples of course) and it is also relatively fast. Typically the Gaussian distribution is used. For a fancier version check out CMA-ES (which is sort of like CEM with momentum) whose benefit is better results with smaller populations.

In total, the benefits are that these methods are fast if parallelized and they're super simple, and the drawbacks are that they suffer from a very harsh dimensionality limit (top limit 30-60 depending on the problem) and are available only for open-loop planning. Generally 10 dimensions and 15 timesteps is what you can expect from this.

Discrete case: Monte Carlo tree search (MCTS)

(Can actually be used for continuous problems, but eh). This shined in Go and poker. Anyhow, tree search blows up exponentially. But could we approximate a value of some node without expanding it? We could get that approximation by following some baseline policy (even a random policy) from that state onward and using the obtained return as the approximate value. In practise, this

algorithm is quite good for many problems and of course it gets better the more you expand.

Here's a generic sketch of MCTS:

1. find a leaf s_l using TreePolicy(s_1)
2. evalute the leaf using DefaultPolicy(s_l)
3. update all value in the tree between s_1 and s_l. then go back to 1.

Finally take best action from s_1.

A common choise for the TreePolicy is the UCT TreePolicy(s_t) which goes as follows. If s_t is not fully expanded, choose new a_t. Else choose child with best Score(s_{t+1}). The score in UCT is (the choise of score is non-trivial, this is just one option):

$$\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C\sqrt{\frac{2\ln N(s_{t-1}}{N(s_t)}} \tag{1.151}$$

So we give a bonus for rarely visited nodes (states in tree terminology I assume). Here N is the number of times a state has been visited and Q is the return obtained. Of course, you can do MCTS with RL and use value functions to do the estimated values for leaf nodes (ex. AlphaGo).

1.9.1 Trajectory optimization with derivatives

Skipping this as it's mostly about LQR which I won't be using soon (because right now I'm working with pixels). This is lecture 10, parts 3, 4 and 5.

1.10 Model-based reinforcement learning

We'll mostly be working with deterministic model of form $f(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{s}_{t+1}$, because they're easier to deal with and many results go over to the stochastic case $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ as well. When needed, the distinction will be made explicit.

Let's learn $f(\boldsymbol{s}_t, \boldsymbol{a}_t)$ from data, and *plan* through it. Model-based reinforcement learning version 0.5:

1. run base policy $\pi_0(\boldsymbol{a}_t|\boldsymbol{s}_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')_i\}$
2. learn dynamics model $f(\boldsymbol{s}, \boldsymbol{a})$ to minimize $\sum_i ||f(\boldsymbol{s}_i, \boldsymbol{a}_i) - \boldsymbol{s}'_i||^2$ (for discrete states use ex. cross-entropy loss, if continous ex. square-error loss, most generally you'd use negative log-likelihood loss)
3. plan through $f(\boldsymbol{s}, \boldsymbol{a})$ to choose actions

Does this basic recipe work? Yes... This is how system identification works in classical robotics, i.e. this is the problem of using data to fit unknown parameters in a model (most likely a physics model). So it's system identification,

not system learning. Here some care should be taken to design a good base policy. This approach is particurly effective if you can hand-engineer a dynamics representation using the knowledge of physics and just fit a few parameters.

In general this approach doesn't work with high capacity models like neural networks. To show why, imagine you're learning to walk on a mountain and you want to reach the top. You learn that some direction gets you higher and then you fall of a cliff on the top becuase you've only learned to follow that direction. More concretely, your planning algorithm works only within the model. But your model is incomplete. Thus you experience distributional shift:

$$p_{\pi_f}(\boldsymbol{s}_t) \neq p_{\pi_0}(\boldsymbol{s}_t) \tag{1.152}$$

The distribution mismatch problem because exacerbated as you use more expressive model classes. It's really hard to hard-overfit 3 numbers, but it's different for millions of numbers. Can we do better? Can we make

$$p_{\pi_0}(\boldsymbol{s}_t) = p_{\pi_f}(\boldsymbol{s}_t) \tag{1.153}$$

Now the model-based reinforcement learning algorithm version 1.0 is:

1. run base policy $\pi_0(\boldsymbol{a}_t|\boldsymbol{s}_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')_i\}$
2. learn dynamics model $f(\boldsymbol{s}, \boldsymbol{a})$ to minimize $\sum_i ||f(\boldsymbol{s}_i, \boldsymbol{a}_i) - \boldsymbol{s}'_i||^2$
3. plan through $f(\boldsymbol{s}, \boldsymbol{a})$ to choose actions
4. execute those actions and add the resulting data $\left\{(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')_j\right\}$ to $\mathcal{D}$ and go back to 2.

This is like DAgger for models.

What if we make a mistake? Asymptotically, the model will get update and the issue will be solved? But can we fix the mistake immediately? Enter model-based reinforcement learning algorithm version 1.5:

1. run base policy $\pi_0(\boldsymbol{a}_t|\boldsymbol{s}_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')_i\}$
2. learn dynamics model $f(\boldsymbol{s}, \boldsymbol{a})$ to minimize $\sum_i ||f(\boldsymbol{s}_i, \boldsymbol{a}_i) - \boldsymbol{s}'_i||^2$
3. plan through $f(\boldsymbol{s}, \boldsymbol{a})$ to choose actions
4. execute the first planned action, observe resulting $\boldsymbol{s}'$ (MPC)
5. append $\{(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')\}$ to $\mathcal{D}$ and go back to 3 (and replan). every N steps go back to 2.

This works better, but it is much more computationally expensive. Essentially, the more you replan, the less perfect each individual plan needs to be. We use shorter horizons here. And even random sampling can work well here.

1.10.1 Uncertainty in model-based RL

There is a performance gap in model-based RL (when compared to model-free RL). The problem is that model is overfitting when it has little data and it needs not to to get good results later (they get stuck). If the model is broken somehow, the planner will exploit that. Uncertainty estimation can help. Just estimating the confidence interval around the reward expectation can be used in avoiding uncertain areas. To crete an uncertainty-aware RL model-based algorithm, we just need to change step 3 so that only the actions which are deemed to be high reward in expectation are taken. This avoids "exploiting" the model. The model will then adapt and get better. There are a few caveats tho. We need to explore to get better. Thus too much caution could lead to never exploring the high reward regions. Furthermore, expected value is not the same as pessimistic value nor the optimistic value, it's just a good start.

Uncertainty-aware neural network models

Idea 1: use output entropy. This is a bad idea. ex:

$$(\boldsymbol{s}_t, \boldsymbol{a}_t) \rightarrow \text{network} \rightarrow p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{1.154}$$

Why is this not enough? Because the optimizer can exploit errors to optimize against our model. Then it will be finding out of distribution actions which lead to out of distinction states which means that our model will have to make predictions on states it was not trained on. The out-of-distribution predictions will result in both wrong means and variances. That's because the uncertainty of the neural network output is the wrong kind of uncertainty (not neural network specific). This is because this measure of entropy is not trying to predict the uncertainty about the model, it's trying to predict how noisy the environment dynamics are.

There are 2 types of uncertainty:

1. *aleatoric* or *statistical* uncertainty (will not go down over time if the environment is random)
2. *epistemic* or *model* uncertainty (should go down over time because you don't know what the model is)

Idea 2: estimate model uncertainty — " the model is certain about the data, but we are not certain about the model."

$$(\boldsymbol{s}_t, \boldsymbol{a}_t) \rightarrow \text{network} \rightarrow p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t), \text{parameters } \theta \tag{1.155}$$

Usually we estimate

$$\underset{\theta}{\text{argmax}} \log p(\theta|\mathcal{D}) = \underset{\theta}{\text{argmax}} \log(\mathcal{D}|\theta) \tag{1.156}$$

but can we instead estimate $p(\theta|\mathcal{D})$? The entropy of this tells us the model uncertainty. Then we'd predict according to

$$\int p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t, \theta)p(\theta|\mathcal{D})d\theta \tag{1.157}$$

Quick overview of Bayesian neural networks

Just the high-level idea here, we'll get back to this. In Bayesian neural networks, there's a distribution over every weight. If you want a prediction, you can sample over every weight (thus sampling a network from a distribution of neural networks) and ask it for its prediction. You can also get a posterior distribution by samplin a net, then predicting, sampling a net again and predicting and repeating this until you get enough samples to form an idea of the posterior. Of course, neural nets are highly dimensional so this is expensive. Thus some common approximations are introduced:

$$p(\theta|\mathcal{D}) = \prod_i p(\theta_i|\mathcal{D}) \tag{1.158}$$

this is not particularly good because it's so crude, but it is very simple and tracktable approximation so its used often. Another option is

$$p(\theta_i|\mathcal{D}) = \mathcal{N}(\mu_i, \sigma_i) \tag{1.159}$$

Thus introducing 2 numbers for each weight which gives you the uncertainty of each weight. Check papers if you want to know more about this (some will be covered in the variational inference lecture too). Let's talk about a simpler method.

Bootstrap ensembles

What instead of training a Bayesian neural net, we instead train many different nets and diversify them. Ideally they'd do similar and accurate things on the training data, but they'd all make different mistakes outside of training data. The dispersion of their votes would then give us an estimate of their uncertainty. Formally:

$$p(\theta|\mathcal{D}) \approx \frac{1}{N}\sum_i \delta(\theta_i) \tag{1.160}$$

So this is a mixture of Dirac δ functions, where each δ function is centered at a parameter vector in the corresponding network ensemble. So train multiple models and see whether they agree. Formally you average over the models:

$$\int p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t, \theta)p(\theta|\mathcal{D})d\theta \approx \frac{1}{N}\sum_i p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t, \theta_i) \tag{1.161}$$

We're mixing the probabilites, not the means (so treat this as a mixture of Gaussians). How do we train this? Need to generate "independent" dataset

to get "independent" models, of course sample from the same dataset. If we have a large amount of data, we can just split the dataset into N datasets and train on that. Of course that's data-inefficient. Instead we could train θ_i on $\mathcal{D}_i$, sampled *with replacement* form $\mathcal{D}$. This means that each slot in $\mathcal{D}_i$ is obtained by randomly picking an element from $\mathcal{D}$. In practise it's even easier. We do < 10 models as they're expensive to trian. Also, just by training with stochastic gradient descent is often enough diversity and you don't even need to resample with replacement (even though that's still important for theoretical results).

How to plan with uncertainty

Before:

$$J(\boldsymbol{a}_1, \dots, \boldsymbol{a}_H) = \sum_{t=1}^{H} r(\boldsymbol{s}_t, \boldsymbol{a}_t), \text{ where } \boldsymbol{s}_{t+1} = f(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{1.162}$$

and now:

$$J(\boldsymbol{a}_1, \dots, \boldsymbol{a}_H) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{H} r(\boldsymbol{s}_{t,i}, \boldsymbol{a}_{t,i}), \text{ where } \boldsymbol{s}_{t+1,i} = \underbrace{f_i(\boldsymbol{s}_{t,i}, \boldsymbol{a}_{t,i})}_{\text{distribution over deterministic models}} \tag{1.163}$$

In general, for candidate action sequence $\boldsymbol{a}_1, \dots, \boldsymbol{a}_H$:

1. sample $\theta \sim p(\theta|\mathcal{D})$
2. at each time step t, sample $\boldsymbol{s}_{t+1} \sim p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t, \theta)$
3. calculate $R = \sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t)$
4. repeat steps 1 and 3 to accumulate the average reward

Other options: moment matching, more complex posterior estimation with BNNs, etc. NOTE: there is a gazillion of papers to read here and you should do it if you're serious about all this.

1.10.2 Model-based reinforcement learning with images

We had $f(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{s}_{t+1}$. But this is particularly hard or images because:

- they're high dimensional
- a lot of information is redundand (think of Pong)
- there's partial observability

We'd like to learn the transition dynamics in the state space (images are observations of course), but we don't even know what the state space is . How about separately learning $p(\boldsymbol{o}_t|\boldsymbol{s}_t)$ (high-demensional but not dynamic) and $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ (low dimensional but dynamic)? We'll discuss this and learning dynamics straight from the images. Let's start with state space (latent space models) models.

State space (latent space models)

Notation:

- $p(\boldsymbol{o}_t|\boldsymbol{s}_t)$ - observation model
- $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ - dynamics model
- $p(r_t|\boldsymbol{s}_t, \boldsymbol{a}_t)$ - reward model

How do we train this? If we had the standard (fully observed) model, we'd train it with maximum likelihood:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(\boldsymbol{s}_{t+1,1}|\boldsymbol{s}_{t,i}, \boldsymbol{a}_{t,i}) \tag{1.164}$$

In a latent space model we do (we need want to add the reward model in there if we want one):

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} E_{(\boldsymbol{s}_t, \boldsymbol{s}_{t+1}) \sim p(\boldsymbol{s}_t, \boldsymbol{s}_{t+1}|\boldsymbol{o}_{1:T}, \boldsymbol{a}_{1:T})} \left[\log p_\phi(\boldsymbol{s}_{t+1,1}|\boldsymbol{s}_{t,i}, \boldsymbol{a}_{t,i}) + \log p_\phi(\boldsymbol{o}_{t,i}|\boldsymbol{s}_{t,i})\right] \tag{1.165}$$

The problem is that we don't know what $\boldsymbol{s}$ is so we need to do the expected log likelihood objective, where the expectation is taken over the distinction over the unknown states in our training trajectories. How to do this then? Learn *approximate* posterior $q_\psi(\boldsymbol{s}_t|\boldsymbol{o}_{1:T}, \boldsymbol{a}_{1:T})$ which is called the "encoder". There's many other choises for appropriate posterior, like a neural net which gives you

$$q_\psi(\boldsymbol{s}_t, \boldsymbol{s}_{t+1}|\boldsymbol{o}_{1:T}, \boldsymbol{a}_{1:T}) \tag{1.166}$$

This called the full smoothing posterior and it's the best, most complex and refined thing you can do, but it's also the most difficult to train. On the other end you could ask for just

$$q_\psi(\boldsymbol{s}_t|\boldsymbol{p}_t) \tag{1.167}$$

This is called the single-step encoder, it's the easiest posterior to train, but also the worst in the sence that using it is the furtherst from what you really want. Training this requires understanding of variational inference.

Anyway, let's talk about the single step encoder. A simple special case: $q(\boldsymbol{s}_t|\boldsymbol{o}_t)$ is *deterministic*. Then

$$q_\psi(\boldsymbol{s}_t|\boldsymbol{o}_t) = \delta(\boldsymbol{s}_t = g_\psi(\boldsymbol{o}_t)) \implies \boldsymbol{s}_t = g_\psi(\boldsymbol{o}_t) \tag{1.168}$$

With this we can remove the expectation from the loss, getting:

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(g_\psi(\boldsymbol{o}_{t+1,i})|g_\psi(\boldsymbol{o}_{t,i}), \boldsymbol{a}_{t,i}) + \log p_\phi(\boldsymbol{o}_{t,i}|g_\psi(\boldsymbol{o}_{t,i})) \tag{1.169}$$

The full version with the reward model looks like:

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \underbrace{\log p_\phi(g_\psi(\boldsymbol{o}_{t+1,i})|g_\psi(\boldsymbol{o}_{t,i}), \boldsymbol{a}_{t,i})}_{\text{latent space dynamics}} + \underbrace{\log p_\phi(\boldsymbol{o}_{t,i}|g_\psi(\boldsymbol{o}_{t,i}))}_{\text{image reconstruction}} + \underbrace{\log p_\phi(r_{t,i}|g_\psi(\boldsymbol{o}_{t,i}))}_{\text{reward model}} \tag{1.170}$$

Let's write out a model-based RL algorithm which uses this:

1. run base policy $\pi_0(\boldsymbol{a}_t|\boldsymbol{o}_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(\boldsymbol{o}, \boldsymbol{a}, \boldsymbol{o}')_i\}$
2. learn $p_\phi\left(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t\right), p_\phi(r_t|\boldsymbol{s}_t), p(\boldsymbol{o}_t|\boldsymbol{s}_t), g_\psi(\boldsymbol{o}_t)$
3. plan through the model to choose actions
4. execute the first planned action, observe resulting $\boldsymbol{o}'$ (MPC)
5. append $\{(\boldsymbol{o}, \boldsymbol{a}, \boldsymbol{o}')\}$ to $\mathcal{D}$ and go back to 3 (and replan). every N steps go back to 2.

OK. What about learning directly in observation spaces, i.e. directly learning $p(\boldsymbol{o}_{t+1}|\boldsymbol{o}_t, \boldsymbol{s}_t)$

1.11 Model-based policy learning

While our MBRL algorithm ver 1.5 makes open-loop predictions and then replans on every timestep, it's still open-loop overall because it can't plan to make other decisions in the future in responce to other informaton that will be revealed in the future. It's really doing (this is recap but still):

$$p_\theta(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_T \boldsymbol{a}_1, \ldots, \boldsymbol{a}_T) = p(\boldsymbol{s}_1) \prod_{t=1}^{T} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{1.171}$$

$$\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T = \arg\max_{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T} E\left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t)|\boldsymbol{a}_1, \ldots, \boldsymbol{a}_T)\right] \tag{1.172}$$

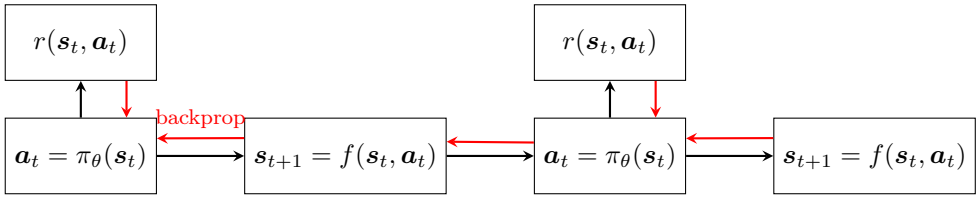
This is suboptimal when compared to learning a policy, which is a closed-loop mechanism (because it knows what respone it will make for anything that can happen, which effectively makes it reactive to random events). Thus a closed-loop looks like:

$$p(\boldsymbol{s}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{s}_T, \boldsymbol{a}_T) = p(\boldsymbol{s}_1) \prod_{t=1}^{T} \pi(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t \tag{1.173}$$

$$\pi = \operatorname{argmax}_\pi E_{\tau \sim p(\tau)}\left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right] \tag{1.174}$$

And this is the big difference between open-loop and closed-loop control. In deep RL π is a neural net which is global in scope. You could also have a local, time-varying linear controller like LQR: $\boldsymbol{K}_t \boldsymbol{s}_t + \boldsymbol{k}_t$.

One thing we can if we want to train a policy using a learned dynamics model is the obvious thing: write down the total reward you get from the policy and the dynamics and do backprop to optimize it. If we assume everything is deterministic (both model and policy), we can set up a computational graph which represents the total reward of the policy. So we can just computer the



derivative of the total reward w.r.t. policy parameters, run backpropagation to find that derivative and just do gradient ascent on that derivative (in the diagram above the red arrows are backprop arrows). This can be easily done in Pytorch and TensorFlow. It's easy to do for deterministic policies, but it's also possible to implement for stochastic policies. But it is problematic. That's because you'll get big gradients on the actions in the first timesteps and the actions on the last timestep towards the end, due to the fact that earlier actions lead to bigger difference in the trajectories later on. Thus the first-order methods are poorly conditioned. There are similar parameter sensitivity probles as shooting methods. Also no dynamic programming is possible because the policy parameters couple all time steps. Overall the problems are similar to those when training RNNs with very long time sequences (vanishing and exploding gradients). However, unlike LSTMs, we can't just "choose" a simple dynamics to control the gradients — nature chose our dynamics. So all in all this is bad for model-based reinforcement learning.

What's the solution?

First class of solutions Use derivative-free (model-free) RL algorithms with using the model not in the real world, but on the model to generate synthetic samples. Altough it seems strange to use the model to train model-free algorithms, it works well in practise. It's essentially model-based acceleration for model-free RL.

Second class of solutions Use simpler policies than neural networks, ex. LQR with learned models (LQR-FLM (fitted local models)). Then use those models to train local policies to solve simple(r) tasks and the combine the local policies into a global policy via some supervised learning procedure.

1.11.1 Model-free learning with a model

Basically use the model-free algorithm to make use of the virtually infinite synthetic data you can get from your model. This can be really good if, for example, you use policy gradients because the fact that you have many samples will lead to lower variance. The policy gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \hat{Q}^\pi_{i,t} \tag{1.175}$$

is also the gradient of the total reward with respect to policy parameters. So it's also computing the derivative through the dynamics, it just doesn't require knowing the functional form of the dynamics log probability for this. Backprop (pathwise) gradient:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{t=1}^{T} \frac{dr_t}{d\boldsymbol{s}_t} \prod_{t'=2}^{t} \frac{d\boldsymbol{s}_{t'}}{d\boldsymbol{a}_{t'-1}} \frac{d\boldsymbol{a}_{t'-1}}{d\boldsymbol{s}_{t'-1}} \tag{1.176}$$

Thus there 2 gradients represent the same quantity. So policy gradient might be more stable (if enough samples are used) because it does not require multiplying many Jacobians.

Dyna

Let's first cover the original method from Sutton's paper. It's an iterative online procedure.

1. given state s, pick action a using exploration policy
2. observe s' and r to get transition (s, a, s', r)
3. update model $\hat{p}(s'|s, a)$ and $\hat{(}s, a)$ using (s, a, s')
4. Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s',r} \left[r + \max_{a'} Q(s', a') - Q(s, a) \right]$
5. repeat K times:
6. sample $(s, a) \sim \mathcal{B}$ from buffer of past states and actions
7. Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s',r} \left[r + \max_{a'} Q(s', a') - Q(s, a) \right]$. after K repetitions go collect more samples

The original version suggests you take an action from the dataset, but it makes more sense take actions from a new policy. And this is how modern versions do this.

General "Dyna-style" model-based RL recipe

1. collect some data, constisting of transitions (s, a, s', r)
2. use that data to learn a model $\hat{p}(s'|s, a)$ (using whatever supervise learning technique), and also learn the reward model $\hat{r}(s, a)$ if you don't know it
3. repeat K times:
4. sample $s \sim \mathcal{B}$ from buffer
5. choose action a (from $\mathcal{B}$ (bad part: closer to the dataset distribution), from π (bad part: incurs distributional shift in model) , or random (bad 'cos it's random)
6. simulate $s' \sim \hat{p}(s'|s, a)$ (and r $= \hat{r}(s, a)$ if needed)
7. train on (s, a, s', r) with model-free RL (everything but MC policy gradients works well)
8. (optional) take N more model-based steps

The advantages are: that only short rollouts are required (as few as one step). This is good because distributional shift gets higher the longer the rollout is. You get to see diverse states. There are 3 algorithms which use this (sorted by age):

- model-based acceleration (MBA)
- model-based value expansion (MVE)
- model-based policy optimiation (MBPO)

Here's the model ML-y version of the algorithm:

1. take some action $\boldsymbol{a}_i$ and observe $(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)$ and add it to $\mathcal{B}$
2. sample mini-batch $\left\{\boldsymbol{s}_j, \boldsymbol{a}_j, \boldsymbol{s}'_j, r_j\right\}$ uniformly
3. use $\left\{\boldsymbol{s}_j, \boldsymbol{a}_j, \boldsymbol{s}'_j\right\}$ to update model $\hat{p}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$
4. sample $\{\boldsymbol{s}_j\}$ from $\mathcal{B}$
5. for each $\boldsymbol{s}_j$, perform model-based rollout with $\boldsymbol{a} = \pi(\boldsymbol{s})$
6. use all transtitions $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', r)$ along rollout to update Q-function

Local models

Super interesting actually, but will skip atm