

10707

Deep Learning: Spring 2020

Andrej Risteski

Machine Learning Department

Guest Lecturer:
Elan Rosenfeld

Lecture 22:

Reinforcement Learning

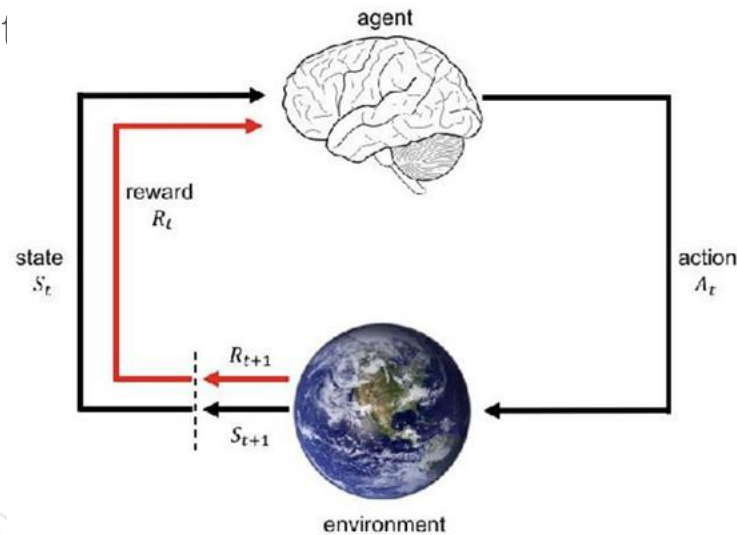
(some material shamelessly stolen from <http://rail.eecs.berkeley.edu/deeprlcourse>)

A (very) brief introduction to RL

In most of the machine learning we've done in this class, we've begun by assuming the data is drawn i.i.d. from some unknown distribution.

This of course is not how the world (nor our intelligence) works. In reality, we take in data in a continuous stream, and act based on it.

The analog in machine learning is **Reinforcement Learning**, in which we train an algorithm ¹ experience.



A (very) brief introduction to RL

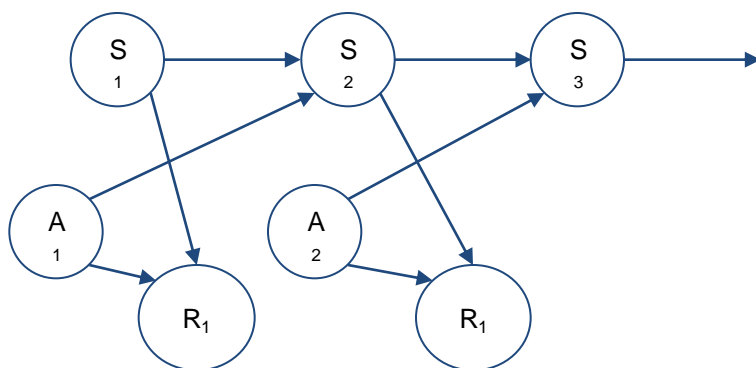
The formal model of this interaction process is called a **Markov Decision Process (MDP)**. This consists of:

A set of (potentially uncountably infinite) possible states $S = \{s_1, s_2, \dots\}$

A set of (potentially uncountably infinite) possible actions $A = \{a_1, a_2, \dots\}$

A matrix of transition probabilities $T: S \times A \rightarrow P(S)$

Agent interacts with its environment (modeled as this system) as follows:



Note: We're assuming here that the state is *fully observed*. RL frequently deals with only *partially observed* states. This is called a POMDP. We won't cover POMDPs in this lecture.

...and so on. We also model an agent's **reward**. This is a measure of how successfully the agent is achieving its goal. The reward is a function $r: S \times A \rightarrow \mathbb{R}$.

A (very) brief introduction to RL

A key assumption of an MDP is the *Markov property*:

The only thing that affects the next state is the current state and chosen action.

The system is *memoryless*; all salient information is encoded in the current state.

Given this system, our goal is to take the actions which maximize our total reward.

The function we use to decide which action to take is the **policy** π .

The policy is a (possibly stochastic) function from state to action.

Given a policy π_θ parameterized by θ , we write $\pi_\theta(a/s)$ to denote the probability of taking action a from state s .



A (very) brief introduction to RL

The policy, combined with the MDP, induces a probability distribution on **trajectories** τ , sequences of state/action pairs.

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

We will sometimes also write $\pi_{\theta}(\tau)$,
these mean the same thing.

Our objective is thus to find the parameters θ which will maximize our expected reward, where the expectation is with respect to the distribution over trajectories.

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

Note that the distribution above assumes a *finite-horizon* ($T < \infty$). We can also handle an *infinite-horizon*. We typically add a **decay parameter** γ . This diminishes future reward in a geometric fashion.



Model based vs. model free

There are two different ways of approaching this problem: **Model-Based** and **Model-Free** algorithms.

A *model-based* algorithm fits a model to the environment. The agent attempts to learn the system dynamics (the transition matrix T). This approximate (or exact) knowledge allows for efficient planning.

A *model-free* algorithm makes no attempt to learn the dynamics. Instead, it hopes to learn a set of rules for maximizing reward. This can be much less efficient, but is more flexible, and doesn't fail as much under model misspecification.

Each works towards improving the policy in a different way.



Model based vs. model free

Model-based techniques have been studied extensively in control theory for decades.

Much of model-based deep RL borrows ideas from control theory, but works to improve inference over states via modern deep learning techniques.

A common approach is latent variable inference to convert a POMDP into an MDP.

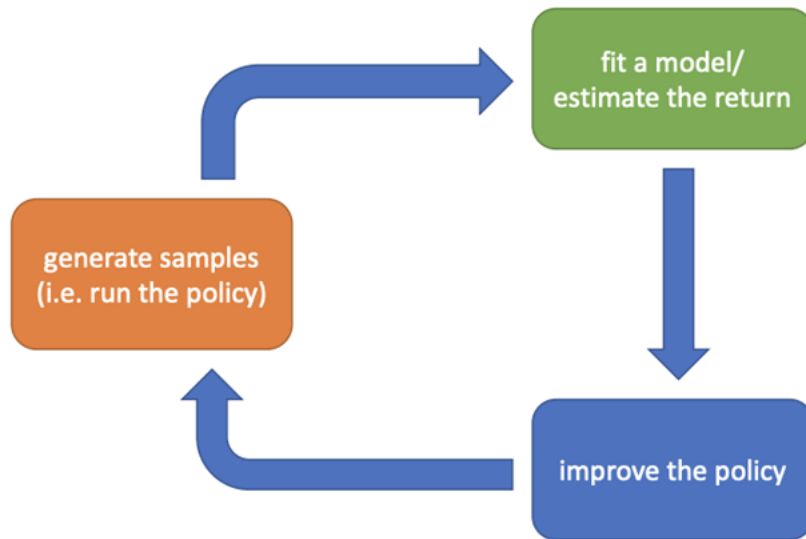
Model-free control is much more recent, and is a larger focus of the deep RL community.

Most of what we will be discussing in this lecture is model-free.



The anatomy of a reinforcement learning algorithm

Almost every RL algorithm follows the same process:



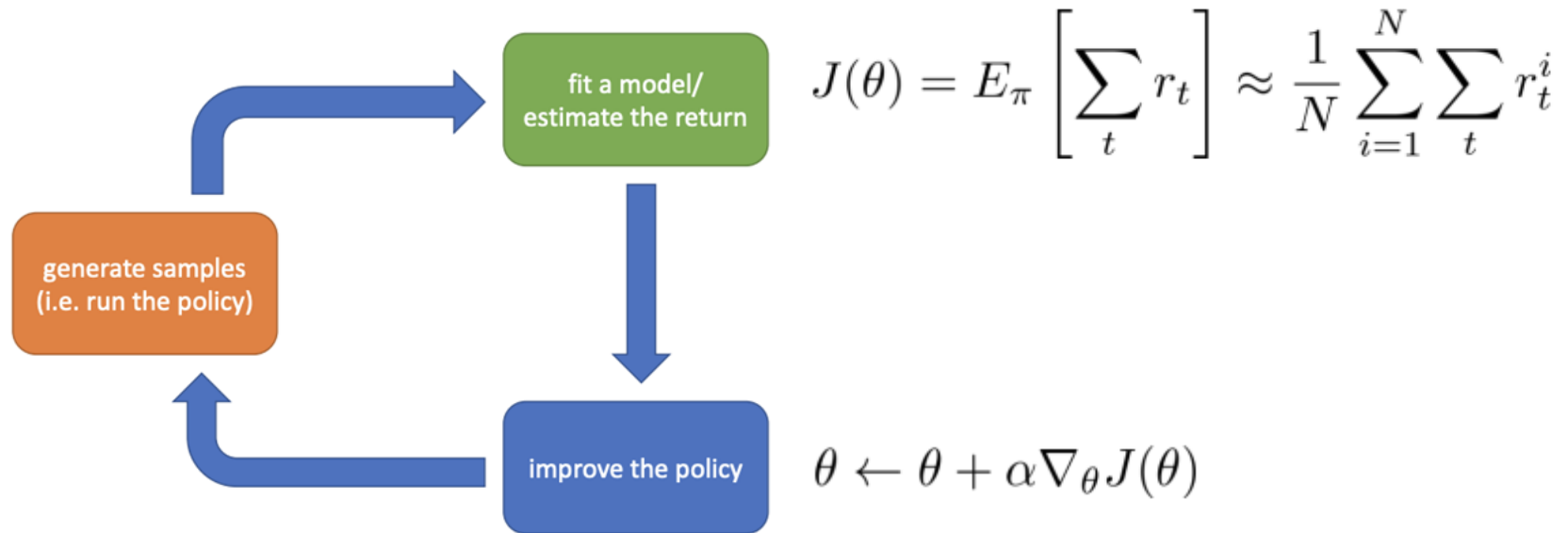
They differ in how they approach each step.

This lecture will *very briefly* touch upon some common approaches.

If you want to learn more, check out 10-703!

The anatomy of a reinforcement learning algorithm

A straightforward example:



We'll get into the details of how this works later.

That was a lot

Any questions?



Part I: Q-learning



Let's start with something simple

One of the most intuitively simple RL algorithms is **Q-learning**.

[Watkins, 1989]

For an infinite horizon MDP, the Q function $Q(s, a)$ approximates the expected cumulative reward for taking action a in state s and then continuing to choose actions based on the Q function.

Question: *is Q-learning model-based or model-free?*

Q-learning is model-free. We make no attempt to learn how our action will influence future states. We just estimate our expected reward for taking a given action in a given state.



Q-learning

So how do we learn this function?

We can treat this as a standard regression problem!

In state s_i , take action a_i , receive reward $r(s_i, a_i)$.

Then regress $Q(s_i, a_i)$ on observed reward + all future rewards.

Wait...

We don't *know* all future rewards! Are we supposed to wait until the end of the trajectory to update our Q function??

We can estimate it again with the Q function!

In other words, we instead regress Q on $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$

This is known as *Bellman Backup*.



Q-learning

Historically, the Q function would just be a lookup table. This of course does not work for continuous state/action spaces. We could use a simple parametric function, but this might not be flexible enough.

Enter **Deep Q-learning**.

It's exactly what it sounds like. Use a deep network to approximate the Q function.

Given a network Q parameterized by ϕ , our objective is to find

$$\phi^* = \arg \min_{\phi} \sum_i \|Q_{\phi}(s_i, a_i) - y_i\|_2^2$$



Using the Q function

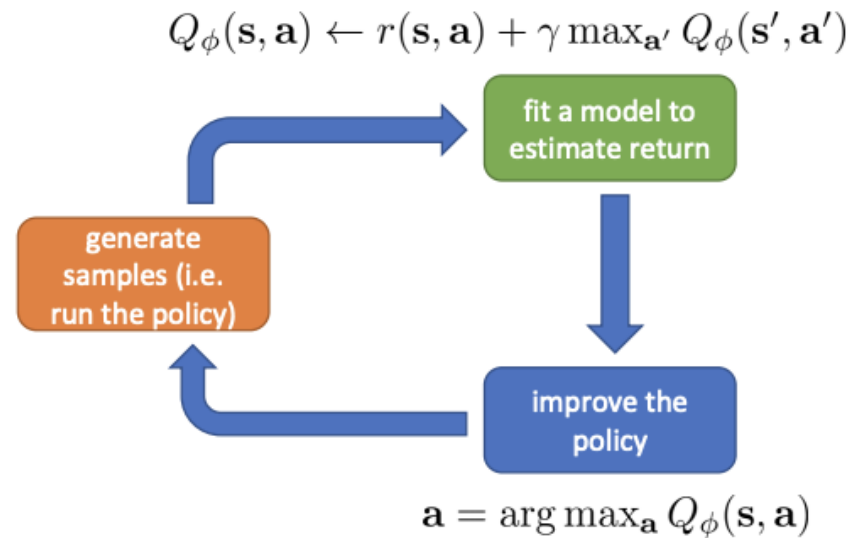
Given a Q function, our policy is very easy:

Take whatever action maximizes our expected future reward.

In other words, $\mathbf{a} = \arg \max_{\mathbf{a}} Q_{\phi}(\mathbf{s}, \mathbf{a})$

Note that this is a *deterministic* policy!
Typically this will be implemented in an ϵ -greedy fashion.

Our algorithm looks like this:



Online vs. Offline

We can perform this algorithm *online* as our agent explores, or *offline* on a collection of data.

Offline is very straightforward:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
 2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. set $\phi \leftarrow \arg \min_{\phi} \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$
- Under some regularity conditions, this converges.


What's wrong with this?

We frequently care about learning from observations quickly.

If something changes drastically during an episode, the policy won't be updated until after the episode ends, which could be catastrophic.

Online vs. Offline

Online Q-learning is a little more complicated:

- 
1. take some action \mathbf{a}_i and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
 2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
 3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$
- These are correlated!
- This is not actually gradient descent!


What's more complicated? It looks basically the same.

There are several techniques for addressing each of these. We'll cover just one for each.



Replay Buffers

So how do we deal with correlated observations?

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


We can keep a **Replay Buffer** of all previous (s, a, s', r) tuples.
When we observe a new tuple, add it to the buffer.
When updating the Q function, sample from the buffer.

Sampling uniformly is not always the best idea. Not all observations are equally useful/representative of the current environment.
We can *prioritize* sampling particular observations.



Target Networks

Ok, what about the other problem?

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

← This is not actually gradient descent!

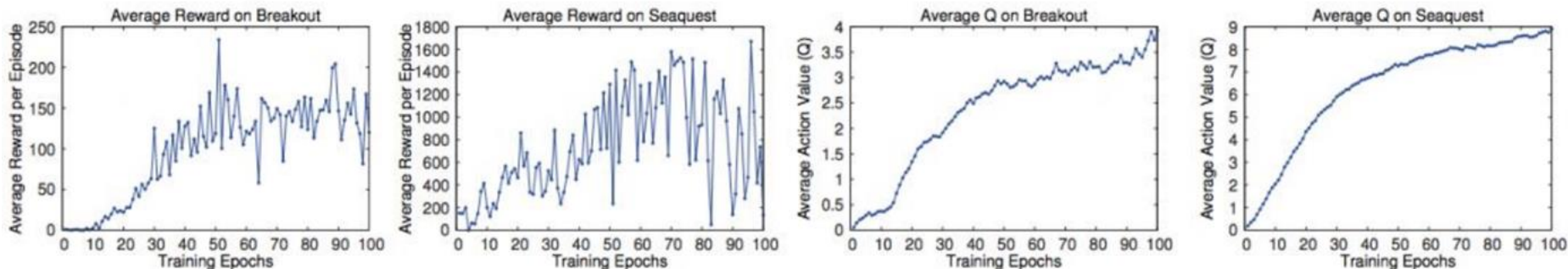
This iterative process results in a *moving target* for our Q function. Instead, we can do a sort of “batch offline optimization” inner loop.

Copy and freeze network parameters ϕ' . This also means the targets y_i don't change (so now it's correct not to propagate the gradient). Regress Q_ϕ on observations, but use $Q_{\phi'}$ to estimate future rewards. $Q_{\phi'}$ is a stable **Target Network**. After N steps, update $\phi' = \phi$, repeat.



Double Q-learning

How accurate are Q-values?



They're correlated with observed reward. But they overestimate it!
Why?

We use the *same* Q function to **choose** the action to take, and to **estimate** the value of said action.

Recall: $\mathbf{a} = \arg \max_{\mathbf{a}} Q_{\phi}(\mathbf{s}, \mathbf{a})$

This means we are biased upwards. We will necessarily have an optimistic estimate of the expected reward.

Double Q-learning

[Hasselt, 2010]: To address this, introduce a *second* Q function. Call the parameters ϕ_A and ϕ_B , so the functions are Q_{ϕ_A} and Q_{ϕ_B} .

When we update the functions, each estimates the value of the other's maximizing action.

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

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

The noise in the two Q functions' approximations are uncorrelated, so they don't work together to bias the estimate.

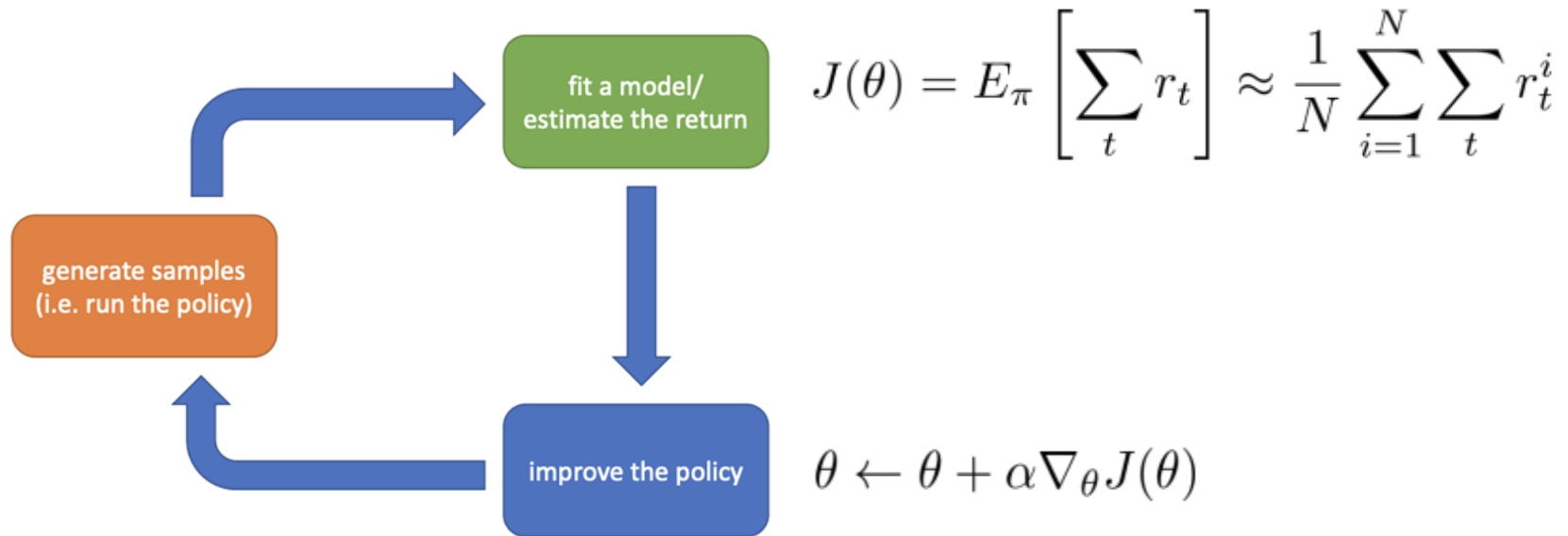


Part II: Policy Gradients



Policy Gradients

Remember that original example we saw for learning a policy?



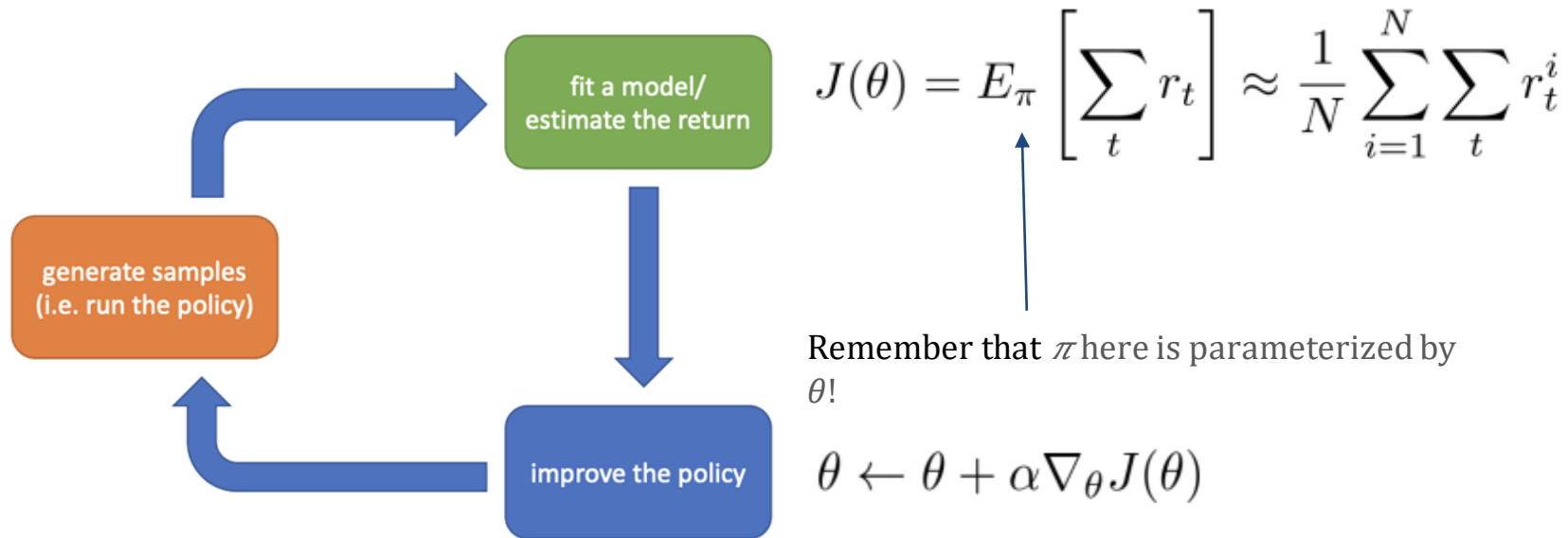
Seems pretty straightforward. We follow the gradient of the policy parameters to increase the expected reward.

There's a catch!



Policy Gradients

Remember that original example we saw for learning a policy?



This means we need to take the gradient of the parameters over an expectation *which depends on the parameters*.

Recall this same problem occurs when optimizing a VAE. But here, there's no simple method for reparameterization.

The REINFORCE algorithm

[Williams, 1992]

Recall:

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

Rearranging,

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

Our objective:

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

Pushing through the gradient,

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



The REINFORCE algorithm

As usual, working with log-likelihood makes everything simpler.

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)} [\nabla_{\theta} \log \pi_{\theta}(\tau) r(\tau)] \\ &= \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)} \left[\nabla_{\theta} \left[\cancel{\log p(\mathbf{s}_1)} + \sum_{t=1}^T \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) + \log p(\cancel{\mathbf{s}_{t+1}} | \mathbf{s}_t, \mathbf{a}_t) \right] \right] \\ \nabla_{\theta} J(\theta) &= E_{\tau \sim \pi_{\theta}(\tau)} \left[\left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \right) \left(\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right]\end{aligned}$$

In practice, this is approximated with the empirical estimate.


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



The REINFORCE algorithm

Putting it all together:

REINFORCE algorithm:

- 
1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run it on the robot)
 2. $\nabla_\theta J(\theta) \approx \sum_i (\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)) (\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i))$
 3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

So what does this accomplish?

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

maximum likelihood:
$$\nabla_\theta J_{\text{ML}}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right)$$

Looks like MLE, but weighted by the reward of the trajectory!
Makes good trajectories more likely, and bad trajectories less likely.
We've effectively formalized “trial and error”.



Variance reduction for PG methods

Unfortunately, this estimator has very high variance.

This is *not* the same as gradient descent on a standard supervised learning problem.

How can we address this?

One common approach is to use a *control variate*.

This is a “baseline” which we subtract from the reward at each step.

The most basic choice is just the average reward.

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

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



Variance reduction for PG methods

Can we just subtract the baseline like that?

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

This term has expectation 0, so subtracting it is ok!

Average reward is not the best control variate. We can *learn* a better one from data.



Part III: Actor-Critic Methods



On-policy vs. Off-policy

The policy gradient method we just saw had an agent acting according to its policy and then improving the policy based on observations.

This is known as **On-policy Reinforcement Learning**. The observations we use for learning come directly from the policy we are improving.

As we saw, this can lead to very high variance because of the large variance in trajectories and rewards.

What if we “smooth out” the observed reward by replacing it with the *expected reward*?

In other words, separate the *policy improvement* and the *policy evaluation* into two roles.



Actor-Critic Methods

In **Actor-Critic Methods**, the actor collects observations and improves the policy (e.g., by following the policy gradient). But the rewards used for weighting the trajectories come from the critic, who learns a function to estimate the *expected* reward. This smooths out the high variance observed by the actor.

This is known as **Off-policy reinforcement learning**. The observations used for improving the policy come from a *different source* than the policy being improved upon.

What's a good choice for a critic who wants to estimate the expected reward of taking a particular action from a particular state?



Q Actor-Critic

... Remember Q-learning?

We can estimate our new policy gradient as:

$$\nabla_{\theta} J(\theta) \approx \sum_i \nabla_{\theta} \log \pi_{\theta}(a_i | s_i) \hat{Q}(s_i, a_i)$$

At the same time as our actor is following the policy gradient using Q-values from the critic, the critic is updating its Q function using observations from the actor!

Q functions are not the only possible function to use for the critic.



Advantage Actor-Critic

Similar to the Q function, we can define a *value function* $V : \mathcal{S} \rightarrow \mathbb{R}$. This function is an estimate of the expected *value* of being in a particular state, and then following the policy from that state. We can learn it similarly to how we learn the Q function.

Why is the value function useful?

It turns out the value function is an excellent control variate for the Q function. We can define the *advantage* of action a_t from state s_t as:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

Intuitively, this represents how much better it is to take the specific action a_t compared to the average action from state s_t .



Advantage Actor-Critic

So we have learn both a Q function *and* a V function?

No. Remember Bellman Backup? It can be shown that at optimality, the following equation holds:

$$Q(s_t, a_t) = \mathbb{E}[r_t + \gamma V(s_{t+1})]$$

This means that we can rewrite the advantage as

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$$

So we can just learn the value function!

Our new estimate of the policy gradient becomes

$$\begin{aligned}\nabla_{\theta} J(\theta) &\sim \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) (r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)) \\ &= \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A(s_t, a_t)\end{aligned}$$



Part IV: Analyzing Performance



Minimizing regret

So far, we've only considered algorithms for maximizing expected reward.

But what if we want a sense of how well we're doing?

It doesn't make sense to measure this by reward alone. We often don't have a good sense of what is a "good" reward.

Instead of analyzing expected reward, we can consider the difference between our performance and some baseline policy.

This difference between (expected) rewards is **(expected) regret**.

Clearly, in a situation with only bad choices, we can't expect our agent to make a good choice. Instead, we want it to have the *least regret* compared to the optimal policy.



Minimizing regret

Recall our original objective: we define a family of parameterized policies π_θ and our goal is to find the parameters θ which maximize the expected reward:

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{\tau \sim \pi_\theta} [r(\tau)]$$

It is natural to compare our performance to the best choice we *could* have made in hindsight.

The *optimal policy* $\pi_{\theta^*}(T)$ is the policy parameterized by the θ^* which maximizes the expected reward as of T steps.

The **expected regret at timestep T** can now be formalized:

$$\mathcal{R}_T(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_{\theta^*}(T)} [r(\tau)] - \mathbb{E}_{\tau \sim \pi_\theta} [r(\tau)]$$

Observe that by definition, regret is always non-negative.

The optimal policy doesn't *have* to be parameterized!



Minimizing regret

What constitutes a “good” regret upper bound?

Intuitively, our goal should be that as our agent explores, its expected future reward approaches that of the optimal policy.

Formally, we want it to be the case that the expected regret *per timestep* is decreasing. That is,

$$\lim_{T \rightarrow \infty} \frac{\mathcal{R}(T)}{T} \rightarrow 0$$

This is called *sublinear regret*, if the numerator grows sub-linearly, our agent’s expected regret per timestep converges to 0 in the limit. We can also prove *lower bounds* on regret. This quantifies the minimum regret we suffer for not knowing the optimal policy ahead of time.

These bounds typically also have terms that grow with the complexity of the MDP.



Check out 10-703

This is a very broad field with tons of interesting stuff going on!

