Reinforcement Learning Scribe: Kiran Vodrahalli
Shipra Agrawal, Columbia University 01/22/2018

1 LECTURE 1: Introduction

Reinforcement learning is a set of problems where you have an agent trying to learn from feedback in the environment in an adaptive way. For instance, a robot is trying to walk from place A to place B. It tries to move its limbs, some things work, some don't. One interesting thing about RL is that it's very old: It has been appearing and re-appearing in many disciplines. Not just in engineering, but also in neuroscience and psychology. It's inspired by a very simple model of how humans learn: people try things. It's called reward systems in neuroscience, operant conditioning in psychology. It's also shown up in optimal control theory. When you add learning to these fields, you get reinforcement learning. In control theory, it is typically assumed that you know the model, there is less uncertainty. Now, if you don't know how the environment works, and you learn that by observing how the environment works while also optimizing, that is reinforcement learning.

1.1 Characteristics of Reinforcement Learning

A reinforcement learning problem has the following properties:

- sequential/online decision making: You don't decide everything in advance. You do something, observe something, and decide the next step.
- No supervisor: This distinguishes it from supervised machine learning. There is no such supervisor telling you that if you move a limb in a particular way, you are correct. However, there are **reward signals**: If you move forward without falling, that is a good signal. It is something like a partial label.
- Delayed feedback: If I do something now, I may not see the full results of the action until further in the future. Every action affects you in a delayed manner.
- Actions affect observations: Adaptive learning. The data is not i.i.d. Here the samples really depend on what you do. You will get different observations depending on the actions you take.

Example 1.1. Examples of Reinforcement Learning.

- Automated vehicle control/robotics: unmanned helicopter learning to fly and perform stunts. This became famous when Andrew Ng got this to work around 2007.
- Game playing: backgammon, Atari breakout, Tetris, Tic Tac Toe. More recently, DeepMind has made headlines playing Go.
- Medical treatment planning: Planning a sequence of treatments based on the effect of past treatments.

- Chat bots: Agent figuring out how to make a conversation. Dialogue generation, natural language processing. There have been some royal failures in this area (racist Microsoft chatbot). Here it is very difficult to judge the rewards. What is the ultimate goal of the conversation?

2 Markov Decision Processes

Suppose you knew how the world reacts. If you had a model of how the world reacts, then it is an optimization problem. The first step is to understand that optimization problem. What can we do if we know the model? Then we can add learning the model to the problem. The basis for everything is the Markov Decision Process (MDP). You have an agent interacting with an environment, it gets feedback, and you keep going. The agent wants to make decisions based on the past feedback. The key (Markov) assumption is that the past feedback can be summarized as a state. The agent observes this state and the immediate reward it got (local increase in score, for instance). Then it decides an action to take, and so on.

In this course, I will assume discrete states, discrete time steps, etc. The extensions to continuous are sometimes a bit more hairy and difficult to explain. Usually the objective is some aggregate function of individual rewards, and is a pseudo-objective. The overall objective might be something else. Action affects not just reward, but also the state. This implements delayed feedback. Actions of course determine the next state of the reward.

Reinforcement learning is just learning added to MDP: You don't know the environment. So you have to learn what to expect from the environment, while trying to decide what action to take. Agents observe samples: rewards and state transitions. A good strategy (policy) for the MDP needs to implicitly or explicitly learn the model in addition to choosing actions. The focus of the course will be on designing an algorithm which can take actions in a sequential manner and ensure that the action not only generates "good" rewards, but also "good" information so that you have more information about the environment in the future. Sometimes, these goals will conflict. What might seem like a good reward strategy may not generate good information. This is the exploration-exploitation tradeoff. The second challenge we will face is scale: Even with just MDPs, the problem becomes complex if the state-space is large. Typically it is very large. So you need to compress the state space. If the robot is trying to walk, then every possible situation in the environment is part of the state space.

2.1 MDP Theory

An MDP is formally defined as a tuple (S, A, s_0, R, P, H). S is a state space, s_0 is the starting state, A is the action space, R is the reward function (expected reward, maybe there is a distribution on the rewards), P is the probability transition function, and H is the horizon. We write $R(s, a) = \mathbb{E}[r_t|s, a]$. We have $P(s, a, s') = \mathbb{P}\{s_{t+1} = s'|s_t = s, a_t = a\}$.

Definition 2.1. Markov property.

State and action only depend on the current state and action. This is a strong property if you are given a compact state space (where you can't stuff all prior history into the state space).

Sometimes you see reward specified depending on what state you transition to. In that case, we can use the same notation, defining $R(s, a)$ as the expected reward according to the transition model: $\sum_{s'} R(s, a, s')\mathbb{P}\{s, a, s'\}$.

The solution concept when talking about MDPs is the policy. The policy will specify what action to take given the current state. We have policy $\pi : S \to A$. If you use the same policy throughout your execution, you have a stationary policy. That is, $a_t = \pi(s_t)$ for all $t = 1, \cdots, H$. In general, the policy does not have to be stationary: You could be given a sequence of policies $(\pi_1, \cdots, \pi_T)$. Depending on the time, you take a different action for the same state. We will see that for many models, it is sufficient to consider a stationary policy, but in some cases, the policy is non-stationary. A non-stationary policy can be useful when the horizon is finite.

Also, π need not be deterministic. It could be a randomized policy, and map $S \to \Delta^A$, a distribution over actions. You will then take actions sampled from that distribution. Stationary policies are a much more convenient construct to talk about. One reason is that when using a stationary policy, in general the MDP where you can take different actions in different states actually just reduces to a Markov chain. The only difference is that there is now a reward associated as well. Thus we call it a Markov Reward Process. If you fix stationary policy π, then $\mathbb{P}\{s'|s\} = P(s, \pi(s), s')$. If this is a randomized policy, then you can write it as $\mathbb{E}_{a\sim\pi(s)}[P(s, a, s')]$. We denote this as P^π, a matrix for every s, s', what is the probability. This is the transition matrix for the Markov process. Now for every state there is a reward associated: $R^\pi(s) = \mathbb{E}_{a\sim pi(s)}[R(s, a)]$. This will be useful when we study optimization.

Definition 2.2. Stationary distribution.
The Markov chains sometimes have stationary distributions. If you run the chain for a long time, the probability of visiting a state will converge to a fixed distribution. In our case, if it exists, we call it the stationary distribution of policy π. This is $\mathbb{P}\{()\, s_t|s_1, \pi)$ as $t \to \infty$.

How do we define the goal of the MDP? What do we want to maxmize in the end? There are many goals you can use, it will be some aggregation of the rewards. Here are some options that people study.

2.1.1 Finite Horizon MDP

You are only interested in making these decisions for some fixed number of steps. Your goal is to maximize total reward over these steps. For some finite H, pre-specified in the MDP, you want to maximize $\mathbb{E}\left[\sum_{t=1}^{H} \gamma^{t-1} r_t\right]$ for some $0 < \gamma \leq 1$. This is called discounting – you may not know for how long, you may not want to use a policy that only gives you reward 100 steps later. This can express uncertainty in the future, or that you value things right now.

2.1.2 Infinite Horizon MDP

There is no horizon. This situation is much more useful than finite horizon, not because they model more problems but because you can get very useful insights from infinite horizon. You can also get much more tractable algorithms if you have infinite horizon. One intuition as to why this happens is things like stationary distribution: You can only talk about this in infinite horizon. Here there are three kinds of rewards people use.

- Total reward: $\lim_{T\to\infty} \mathbb{E}\left[\sum_{t=1}^{T} r_t\right]$. There are many examples where this does not diverge. For instance, playing a computer game where the reward is 0 until you actually win, at which point you get reward of 1. Thus, you think about as an infinite horizon with total reward which is not infinite. Another is stochastic shortest path problems.

- Total discounted reward: $\lim_{T\to\infty} \mathbb{E}\left[\sum_{t=1}^{T} \gamma^{t-1} r_t\right]$. This is the most popular objective for RL. The reason this is nice is because as long as you use any $\gamma < 1$, this always is finite and has a lot of nice properties which give fast algorithms. You can use any $\gamma < 1$ and all properties hold. There is one issue that this model does not capture.

- Average reward: $\lim_{T\to\infty} \mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} r_t\right]$. This converges under mild conditions (if you do lim sup or lim inf things are fine), but doesn't always converge. Why is this more interesting? Maybe you don't value reward right now until later. The most important reason is by its nature, discounted reward values initial rewards exponentially higher compared to later rewards. In real problems where you are just thinking about rewards, it is fine. But think about when you are learning. In the beginning, you know **nothing** about the environment, but your reward is being weighted so heavily! Your reward matters right now the most. This discounted reward will tell you to forget about information, just do what gives you reward: Exploit, don't explore. But this is bad for learning. Intuitively you should be learning more initially. This is kind of opposite to that intuition. So all the papers which give formal guarantees for actually learning something use average reward.

I haven't seen discounted rewards that do not discount exponentially. It is used mainly because of the nice structure of the discount. You have to come up with some discount scheme that gives nice structure, or wonder "what kind of discount scheme". If you had models where later rewards matter more, that should be easier from a learning perspective: Focus all your time on learning, and then do rewards later. This is one reason I have never seen papers with learning guarantees which use the discounted reward model.

We don't want to call things reward, that's something you get immediately. So, we call the objective we are trying to maximize as **gain**. We should be talking about total gain given the starting state. This is ρ^*. We can also define **gain of a stationary policy** π as $\rho^\pi(s_1) = \lim_{T\to\infty} \mathbb{E}_{a_t\sim\pi(s_t)}\left[\sum_{t=1}^{T} \gamma^{t-1} r_t | s_1\right]$. Note that this assumes that $\mathbb{E}\left[r_t | s_t, a_t\right] = R(s_t, a_t)$.

Theorem 2.3. *Infinite horizon implies stationary policies are sufficient.*
For infinite horizon, irrespective of whether we discuss average or discounted (or even total, with some extra assumptions) reward case, there exists a stationary policy that is optimal. That is, $\rho^*(s_1) = \max_{\pi \in \Pi} \rho^\pi(s_1)$ for any starting state s_1. So you only need to look over stationary policy space. In fact, in the discounted case, you only need to look at deterministic policies (don't need to look at randomized policies).

Note that you can design an RL environment many different ways for different problems. You have to design your reward signal and define what you mean by gain.

Example 2.4. Robot.
A robot can be upright or fallen down. The state is location and posture. Let us say its state is along a line. Thus state space is given by $\{U, F\} \times \{1, 2, 3, 4, 5\}$. We let the action space have two possible actions, a slow movement and a fast movement. Thus action space consists of $\{\text{slow}, \text{fast}\}$. If you do a fast movement, there's a 20% chance you've fallen down. Once you have fallen you cannot do anything else. Let us define the reward as 1 if you move one step, and 2 if you move two steps (the fast action). Let us say you want to maximize total expected reward: That is the number of steps you need in total. In that case, what is the optimal policy? In this case, you don't value moving faster compared to slower. Here you should just walk slowly. Now suppose you define the goal as discounted reward. Then what should you do? Then it matters what you do initially versus later. The intial steps are way more important than the later steps. Maybe you should take some risk and move faster. The other way to define a reward could be to define a target location, irrespective of what you do. Now we don't care about number of steps, maybe all the rewards are 0 except the reward 1 at the end. Now if you look at discounted reward function, then at some time τ when you actually reach the target location, which will be different depending on the strategy, you will be trying to maximize $\mathbb{E}[\gamma^\tau]$. This is equivalently minimizing the time to reach the target location. Discounted reward tells you to minimize time to reach the target location. Then, you might sometimes want to take a risk and move faster, even at the risk of falling.

An example policy is as follows: In every standing state, you're deciding to move slow. Another one is in every location which is more than two steps away, you move slow, otherwise, you move fast. This is a stationary policy, and also deterministic. It only depends on the state. You could easily make randomized policies as well.

Example 2.5. Inventory control problem.
This is a very modern problem in dynamic programming literature. It has not gotten that much attention in RL, but is definitely a problem where RL could be used. Suppose you have a timeline and you discretize it. At every time t, you get to see the current inventory, how much product you have so far, then you get to order something, the ordered stuff arrives. Then you get to see the demand over a period of time which you can satisfy if you have a product. At this point you have fulfilled all the orders. Then you have the remaining inventory left over after. Formally you can think of s_t as the inventory level. The Markov property is satisfied, since once you know how much inventory you have, that's all that

matters. History does not matter. The only control you have here is ordering. Thus, action at time t, a_t, is the order amount. Here we assume that everything you order arrives, but not how much the customers want. We have $s_t, a_t \to_{D_t} s_{t+1}$. Then $s_{t+1} = [s_t + a_t - D_t]_+$. Here you are not explicitly given the transition function: It depends implicitly on the demand distribution. If you know the demand distribution you know the model, even if you don't have a specific form for it; it's still defined. If the demand has a nice distribution, maybe you can compute that specific formula. Now how to define the reward? You have a cost and a profit. This is an example of a case where the reward is given in terms of the triple (s, a, s'). We have $R(s_t, a_t, s_{t+1}) = f(s_t + a_t - s_{t+1}) - O(a_t) - h(s_t + a_t)$. The first term f is a function of the total amount you have sold. Here O is the order cost and h is the holding cost. You can of course have a lot of variations of this problem.

Example 2.6. Toy numerical example for MDPs.
Now I'll simplify the robot example. I'll eliminate locations and have three postures for the robots: upright, moving, and fallen ($\{U, M, F\}$). There will still be two actions: Slow and fast movement. In standing state if you do slow movement, with probability 1 you transition to moving state, and you get a reward. In the moving state, if you do a slow action, you get 1 reward and go back to the moving state with probability 1. If you do a fast action, you could stay in moving state and get reward 2 with 80% probability, or you could transition to Fallen state with probability 20%. From standing state, if you take a fast movement, you fall with 40% probability and get reward -1 and transition to moving state with 60% probability and get reward $+2$. You can now get up from fallen state: With slow action, you transition to standing state with probability 40% and stay in fallen state with probability 60%. Fast action is a dummy action here, which changes nothing.

Here, R is a matrix where columns are actions and rows are states. Here, it's 3×2 matrix. $R(s, a)$ specifies the reward. But here, we really are talking about $R(s, a, s')$ since we need to know if we get to the moving state to get reward. Thus, the matrix is

$$R = \begin{bmatrix} x & \text{slow} & \text{fast} \\ F & -0.2 & 0 \\ U & 1 & 0.8 \\ M & 1 & 2 * .8 + -1 * .2 \end{bmatrix}$$

We can also write the transition matrix for each action, whose rows sum to 1.

$$P_{slow} = \begin{bmatrix} x & F & U & M \\ F & 0.6 & 0.4 & 0 \\ U & 0 & 0 & 1 \\ M & 0 & 0 & 1 \end{bmatrix}$$

$$P_{fast} = \begin{bmatrix} x & F & U & M \\ F & 1 & 0 & 0 \\ U & 0.4 & 0 & 0.6 \\ M & 0.2 & 0 & 0.8 \end{bmatrix}$$

Now depending on the reward function you are using, you'll get different policies. The next thing we will learn are the Bellman dynamic programming equations. Then, the key thing that comes in Bellman equations is not finite horizon dynamic programming, but rather the structure that exists in infinite horizon which allows us to find fixed points.

3 LECTURE 2: Algorithms for MDPs and RL

In particular, we will talk about algorithms for solving MDPs, and then how to solve MDPs if you do not know the model: Dynamic programming, iterative algorithms (value iteration $\to$ Q-learning) and (policy iteration $\to$ TD learning).

3.1 Solving MDPs

If you recall, and MDP is a tuple $M = (S, A, s_0, R, P, H)$: state space, action space, initial state, reward function, probability transition function, and a horizon which may or may not be there. R is a $S \times A$ matrix, and P is a $S \times A \times S$ tensor. Let's start with solving finite-horizon MDPs.

3.1.1 Finite Horizon MDPs

You want to solve

$$\max \mathbb{E}\left[\sum_{t=1}^{H} \gamma^{t-1} r_t | s_0 = s\right]$$

under the MDP assumptions:

- $\mathbb{E}\left[r_t | s_t, a_t\right] = R(s_t, a_t)$
- the probability $\mathbb{P}\left\{s_{t+1} | s_t, a_t\right\} = P(s_t, a_t, s_{t+1})$

Let us try to solve this by enumeration, brute force: Start in state s_0, and try to look at all possible instances. We can take $|A|$ actions. Recall the model from last time: a robot is walking and can be in falling, standing, and moving settings. There were two cases: one can move fast or slow. If you fall you get reward -1, if you achieve standing from falling, you get $+1$. From standing, slow action gets you to moving with probability 1 with reward $+1$, fast action gets you to moving with probability 0.6 and gives you $+2$, but with probability 0.4 gives you -1, etc. (See the state matrices given in the previous lecture).

Now, there is a state tree that you will end up traversing if you want to see all the effects. Often, you want to precalculate and store what you should do at various states in order to compress the tree: This is called using the "optimal substructure" of the tree. This allows you to avoid repeating calculation. This is memoization, from dynamic programming, allowing us to compress the tree. We store the solutions to larger problems and use them to compute the answer to large problems. These will give us the famous Bellman equations for finite horizon problems.

Theorem 3.1. *Bellman equation*
$V_k^*(s)$ is the optimal value of total discounted award if you start from state s and go for k steps. I will express this in terms of $V_{k-1}^*(s)$ and reduce to a smaller problem. The answer is

$$V_k^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s,a,s') V_{k-1}^*(s')$$

Proof. Let us consider a policy $\pi = (\pi_1, \cdots, \pi_H)$ (it may not in general be stationary, the action you may want to take can depend on which time step you are at). We have

$$\begin{aligned}
V_k^*(s) &= \max_\pi \mathbb{E}\left[\sum_{t=1}^k \gamma^{t-1} r_t | s_1 = s\right] \\
&= \max_\pi \mathbb{E}\left[r_1 | s_1 = s\right] + \mathbb{E}\left[\mathbb{E}\left[\sum_{t=2}^k \gamma^{t-1} r_t | s_1 = s, s_2 = s'\right]\right] \\
&\le \max_{\pi_1} \mathbb{E}\left[r_1 | s_1 = s\right] + \max_{\pi_2, \cdots, \pi_k} \mathbb{E}\left[\mathbb{E}\left[\sum_{t=2}^k \gamma^{t-1} r_t | s_1 = s, s_2 = s'\right]\right] \\
&= \max_a R(s_1, a) + \max_{\pi_2, \cdots, \pi_k} \mathbb{E}\left[\mathbb{E}\left[\sum_{t=2}^k \gamma^{t-1} r_t | s_1 = s, a_1 = a\right]\right] \\
&\le \max_a R(s_1, a) + \gamma \max_{\pi_1, \cdots, \pi_{k-1}} \mathbb{E}\left[\mathbb{E}\left[\sum_{t=1}^{k-1} \gamma^{t-1} r_t | s_1 = s, a_1 = a\right]\right] \\
&= \max_a \left[R(s_1, a) + \gamma \max_{\pi_1, \cdots, \pi_{k-1}} \sum_{s'} \mathbb{E}\left[\sum_{t=1}^{k-1} \gamma^{t-1} r_t | s_1 = s'\right] \mathbb{P}\left\{s' | s, a\right\}\right] \\
&= \max_a \left[R(s_1, a) + \gamma \sum_{s'} \left(\max_{\pi_1, \cdots, \pi_{k-1}} \mathbb{E}\left[\sum_{t=1}^{k-1} \gamma^{t-1} r_t | s_1 = s'\right]\right) \mathbb{P}\left\{s' | s, a\right\}\right] \\
&= \max_a \left[R(s, a) + \gamma \sum_{s'} V_{k-1}^*(s') P(s, a, s')\right]
\end{aligned} \tag{1}$$

where we used the Markov property to assert that it doesn't matter what timestep you start from. □

3.1.2 Infinite horizon and discounted reward

The infinite horizon case where $\gamma < 1$ is very different from the finite horizon case where $\gamma = 1$. The goal here is

$$\max \lim_{T \to \infty} \mathbb{E}\left[\sum_{t=1}^T \gamma^{t-1} r_t\right]$$

If you make simple assumptions about the state space (countable and so on), this limit exists. We also saw last time that the optimal policy in this case happens to be stationary (same policy over different time steps, $\pi_t = \pi_{t+1}$). Taking into account the fact that we are actually looking for stationary policies. For any given policy, the value of a policy π in state s (start from state s and use π again and again) is $V^\pi(s) = \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=1}^T \gamma^{t-1} r_t | s_1 = s\right]$. I will now give the Bellman equation to simply evaluate the value of a policy, and also the Bellman equation for the optimal value.

Theorem 3.2. *Bellman equations (2).*
Assuming we take actions according to policy π, we get a fixed point equation

$$V^\pi = R^\pi + \gamma P^\pi V^\pi$$

where these values are all matrices. If you can solve the fixed point equation, you can get the value function. For a optimal policies,

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s,a,s')V^*(s')$$

and

$$\pi^*(s) = \text{argmax}_a R(s,a) + \gamma \sum_{s'} P(s,a,s')V^*(s')$$

The problem is we don't know how to calculate this. So a lot of algorithmic design will be about computing the optimal value function.

Proof.

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}\left[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots | s_1 = s\right] \\
&= \mathbb{E}\left[r_1 | s_1 = s, a_1 = \pi(s)\right] + \gamma \mathbb{E}\left[r_2 + \gamma r_3 + \gamma^2 r_4 + \cdots | s_1 = s\right] \\
&= \mathbb{E}\left[r_1 | s_1 = s\right] + \gamma \sum_{s'} \mathbb{E}\left[r_2 + \gamma r_3 + \gamma^2 r_4 + \cdots | s_1 = s, s_2 = s'\right] \mathbb{P}\left\{s' | s, a = \pi(s)\right\} \\
&= R(s,\pi(s)) + \gamma \sum_{s'} \mathbb{E}\left[r_2 + \gamma r_3 + \gamma^2 r_4 + \cdots | s_1 = s, s_2 = s'\right] \mathbb{P}\left\{s' | a = \pi(s)\right\} \qquad (2) \\
&= R(s,\pi(s)) + \gamma \sum_{s'} V^\pi(s')P(s,\pi(s),s') \\
V^\pi &= R^\pi + \gamma P^\pi V^\pi
\end{aligned}$$

□

Note that for the calculation of the value of an arbitrary policy, if $\gamma < 1$, you can take inverses: $(I - \gamma P^\pi)V^\pi = R^\pi$, so $V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$. Thus you have a formula that can tell you the value of a policy in every state. But what is more useful is the whole equation, we will see iterative algorithms which apply the equation.

3.1.3 Solving for Value Function with LPs

Theorem 3.3. *LP for Value Functions.*
We can write for non-negative weights w_s the following linear program:

$$\max_v \sum_s v_s w_s \text{ subject to } v_s \leq \max_a R(s,a) + \gamma P(s,a)^T v$$

Then, you will find a vector satisfying the equation for the value function.

Proof. Let us start by assuming there is an optimal stationary policy. If there is such a stationary policy, and π^* is deterministic (can be modified to show that it works even if π^* is not deterministic), then the resulting v must hold for all s, a. It must satisfy the constraint $v_s \leq R(s, \pi^*(s)) + \gamma P(s, \pi^*(s))^T v$. Therefore,

$$(I - \gamma P^{\pi^*})v \leq R^{\pi^*}$$

Now we'd like to do the same trick as before and move the inverse. However, there's an inequality. What we will show that the matrix is nice and the inverse is actually all positive numbers. We show

$$(I - \gamma P^\pi)^{-1} = I + \gamma P^\pi + \gamma^2 (P^\pi)^2 + \cdots \geq 0$$

and thus all the entries in the inverse are non-negative since the entries of P are all positive (it is a transition matrix). (This identity is just Taylor expansion). Thus, you can indeed multiply by inverse on both sides and the inequality will remain intact. Now what is $(I - \gamma P^{\pi^*})^{-1} R^{\pi^*}$? This is just V^{π^*}. Therefore, you can just write down the linear program above. This is efficient as long as the number of constraints is small (over all s, a) (note that this can be very large if you don't have a small MDP!!). □

3.1.4 Iterative Algorithms

The first algorithm which is very popular is value iteration. In the value iteration algorithm, what you do is take the Bellman equation and do the right-hand side again and again. Then you plug in the output back again.

Definition 3.4. Value Iteration.
Start with initialization v^0. Then repeat for $k = 1, 2, \cdots$ until you have converged ($\|v^k - v^{k-1}\|_\infty \leq \epsilon \left(\frac{1-\gamma}{2\gamma}\right)$:

(a) Set $v^k(s) := \max_{a \in A} R(s,a) + \gamma \sum_{s'} P(s,a,s') V^{k-1}(s')$.

(b) Compute $\pi^k(s) = \text{argmax}_a R(s,a) + \gamma \sum_{s'} P(s,a,s') v^{k-1}(s')$.

We will show this algorithm converges exponentially fast. First, we give some notion which will simplify the discussion.

Definition 3.5. Bellman Operator.
Let $LV : \mathbb{R} \to \mathbb{R}$. Then the *optimal Bellman operator* is

$$LV(s) := \max_{a \in A} R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')$$

The Bellman operator for policy π is

$$L^{\pi}V(s) := R(s,\pi(s)) + \gamma \sum_{s'} P(s,\pi(s),s')V(s')$$

Essentially, we will demonstrate the contraction property of the Bellman operator, which will establish the convergence rate of value iteration.

Lemma 3.6. *Contraction of the Bellman Operator.*
For any two vectors v, u:

$$\|L(v-u)\|_\infty \leq \gamma \|v-u\|_\infty$$
$$\|L^{\pi}(v-u)\|_\infty \leq \gamma \|v-u\|_\infty$$

Proof. Fix state s. First wlog assume the s^{th} component has $(Lv)_s \geq (Lu)_s$. Let $a_s^* = \text{argmax}_a R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')$. Then

$$0 \leq (L(v-u))_s \leq R(s,a_s^*) + \gamma \sum_{s'} P(s,a_s^*,s')v(s') - R(s,a_s^*) - \gamma \sum_{s'} P(s,a_s^*,s')u(s')$$

Then, this term can be at most $\gamma\|v-u\|_\infty$ since P sums to 1 (as it is a probability matrix), proving the contraction property.

□

How can we use this contraction property? We'll use this to prove convergence.

Theorem 3.7. *Exponential convergence for value iteration.*
For v^k, the k^{th} iteration vector, we have

$$\|v^k - V^*\|_\infty \leq \frac{\gamma^k}{1-\gamma}\|v^1 - v^0\|_\infty$$

Furthermore, this kind of result applies to the value vector for the policy.

$$\|V^{\pi_k} - V^*\|_\infty \leq \frac{2\gamma^k}{1-\gamma}\|v^1 - v^0\|_\infty$$

So, you need $\log(1/\gamma)$ steps to converge for some constant ϵ. For every iteration, you just do the transformation LV, which is a linear transformation.

Proof. First, we know that $V^* = LV^*$ (by fixed point). Thus,

$$\begin{aligned}
\|V^* - v^k\|_\infty &= \|LV^* - v^k\|_\infty \\
&= \|Lv^* - Lv^k\|_\infty + \|Lv^k - Lv^{k-1}\|_\infty \\
&\leq \gamma\|v^* - v^k\|_\infty + \gamma\|v^k - v^{k-1}\|_\infty \\
&\leq \gamma\|v^* - v^k\|_\infty + \gamma^k\|v^1 - v^0\|_\infty \\
\|v^* - v^k\|_\infty(1-\gamma) \leq \gamma^k\|v^1 - v^0\|_\infty
\end{aligned} \tag{3}$$

Thus the size reduces exponentially fast as long as $\gamma < 1$.

Now, we show that the value vector for the policy is also close to the optimal value vector. First, observe the relation between L and L^π. For vector v^k, L (optimal) and L^π (specific policy) are the same in the sense that L^π is

$$L^\pi v^k(s) = R(s,\pi(s)) + \sum_{s'} P(s,\pi(s),s')v^k = \max_a R(s,a) + \sum_{s'} P(s,a,s')v^k = Lv^k$$

because we chose policy π as the maximizer for v^k (the way π was constructed). Then we want to show that π is a good policy. Since we have already shown that $\|v^k - V^*\|_\infty$ is bounded, we will just show the distance between $\|v^k - V^{\pi_k}\|_\infty$ and can then conclude by triangle inequality.

$$\begin{aligned}
\|V^\pi - v^k\|_\infty &\leq \|L^\pi V^\pi - Lv^k\|_\infty \\
&\leq \|L^\pi V^\pi - L^\pi v^k\|_\infty + \|L^\pi v^k - v^k\|_\infty \\
&\leq \gamma\|V^\pi - v^k\|_\infty + \|Lv^k - v^k\|_\infty \\
&\leq \gamma\|V^\pi - v^k\|_\infty + \gamma^k\|v^1 - v^0\|_\infty \\
\|V^\pi - v^k\|_\infty &\leq \frac{\gamma^k}{1-\gamma}\|v^1 - v^0\|_\infty
\end{aligned} \tag{4}$$

by contraction applied iteratively and the fact that L and L^π applied to v^k have the same result. To finish, just triangle inequality, then you get

$$\|V^{\pi_k} - v^*\|_\infty \leq \|v^k - V^*\|_\infty + \|V^\pi - v^k\|_\infty \leq \frac{2\gamma^k}{1-\gamma}\|v^1 - v^0\|_\infty$$

□

3.1.5 Q-Value Iteration

In every step above, we were computing

$$v^k(s) = \max_s R(s,a) + \sum_{s'} P(s,a,s')V^{k-1}(s')$$

Now, we instead learn $Q(s,a)$: We want to know how much value we get if we start at state s, and our first action is a. Q-iteration can be derived from Bellman equations: We rename things and kind of write the same equation. Thus we write

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') \left(\max_{a'} Q^*(s',a') \right)$$

You are basically specifying one more step. Thus, the same Bellman equations we had earlier can be written in Q-function form. Only difference is that you also need to choose action a' in the next state. Of course,

$$V^*(s) = \max_a Q^*(s,a)$$

So now you can write a similar Q-iteration algorithm with a matrix q for every state-action paper. In learning situations, we care about learning about state-action pairs rather than just learning about state, so it will be useful there. Basically, we make the replacements for the estimates of the value function as well.

$$V^k(s) \to \max_{a'} Q^k(s,a')$$

Now, the proof for convergence is the same since you're doing the exact same things, you are just maintaining more terms.

3.1.6 Policy Iteration

This is another version of the basic algorithm, but it is a more direct method. What we have been doing above is first calculate a good value function vector. Then, once it conveges, you get a policy. There is no intermediate policy here.

In this algorithm, you will initialize a policy and at each step you will improve the policy.

Definition 3.8. Policy iteration.
Start with policy π^0, arbitrary. Now it uses computation of the value function as a subroutine. For every iteration k,

(a) Compute the value of policy π_k and call it v^{k-1}. This is a simpler value iteration: Just computing the value of the policy, which we also have a formula for from before: $V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$. Now it is not usually done this way, people only compute the value of the policy approximately (not to convergence).

(b) Now you improve the policy which takes the action good according to this:

$$\pi^k(s) := \text{argmax}_a R(s,a) + \sum_{s'} P(s,a,s') v^{k-1}$$

So, you calculate a new policy using the value based on the previous policy. If you keep doing this for a long time, your policy will not change after a while. Once the policy stops changing, then you have your function.

The difference between these algorithms is subtle but important. The first algorithm is about computing the optimal value vector, while the second algorithm is about computing optimal policy. Usually this method is used if computing the value of a given policy is easier than finding the value of the optimal policy. For instance, when computing the inverse is easier.

One way to show a policy has converged is to show the policy vector has converged. First you could assume for simplicity that the policy value calculation is not approximate, then you just check the difference of values in the policy. You can in fact show that this will converge even if there are errors in the computation of the value of the policy. You cannot say arbitrary error, some kind of bounded error is necessary (this is a relatively new result).

Note that we are using nothing but the L operator, which is essentially the model: It has all the calculation. When we do learning, we will have to learn the model L.

3.1.7 Average Reward Case

The focus of our class will be to study the discounted case. But the infinite horizon average reward case is still very important. In the discounted case, you are exponentially decreasing the weight of the later steps. This is a drawback if you do'nt know the model, and you are spending time learning the model. If your weight is on the initial steps, then the tradeoff between learning and optimisation is difficult to manage: If you're not interested in the long term, then why are you spending much time learning? This is what average reward fixes: You give equal weight to rewards now or later. It becomes a bit difficult to define the Bellman equations in the finite horizon case. In particular, you need your MDP to satisfy a certain property: You cannot get stuck in any region of the MDP.

So, we will deal with infinite horizon average reward. We would like to maximize average reward:

$$\lim_{T\to\infty} \mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} r_t\right]$$

Definition 3.9. Communicating MDP.
Here we make the assumption of a *communicating MDP*: You cannot get stuck in a portion of the MDP, there is always a way to come out in finite time. Formally, for all s, s', there is some policy π such that $\mathbb{E}\left[\#\text{ steps to reach } s' \text{ from } s\right] < \infty$. The policy is allowed to be different for different pairs of states (s, s'). The expectation quantity is known as the *diameter* of the MDP.

One nice thing that communicating MDP gives you is that the optimal gain for an MDP is the same for every state: it does not matter where you start.

Lemma 3.10. *For communicating MDPs, the optimal infinite horizon average reward (the optimal gain $\rho^*(s)$)) is the same for all states s.*

Proof. This is somewhat surprising, but it happens because we are thinking about infinite reward, and moreover you can't get stuck in a state. Suppose that this was not the case:

$\rho^*(s_1) < \rho^*(s_2)$. We can show a contradiction. We can go in finite time ffrom s_1 to s_2. I'll ditch what you say the optimal policy says, and I'll pick the policy which gets me from s_1 to s_2. Then I'll reach s_2 in finite time, then I can reach the gain in s_2. Since I'm talking about infinite time average, the average of what happens in the finite time is not relevant. So I'll be fine. Thus you cannot get more gain in one state than another. □

Definition 3.11. Gain.
This is the average infinite horizon reward you will get starting from state S and keep applying policy π.

$$\lim_{T\to\infty} \mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} r_t | s_1 = s, a_t = \pi(s_t)\right]$$

We need one more concept to get to the Bellman equations. What do we need for infinite horizon still?

Definition 3.12. Bias.
Define $V^*(s)$ as the difference between the total reward starting from state s compared to the optimal reward.

$$V^\pi(s) = \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=1}^{T}(r_t - \rho^\pi)|s_1\right]$$

This is the bias of a policy π. The gap in the initial step is the bias, before converence to the optimal policy.

Now this will appear in the Bellman equation.

Definition 3.13. Biased Bellman equation.

$$V^\pi(s) + \rho^\pi = R(s, \pi(s)) + P^\pi V^\pi$$

When you apply value iteration, you will not converge to V^π, but a translation of it: That constant will keep increasing due to the bias. There is no discount to make it converge! It will only increase by a constant vector, so you can always subtract some constant vector. But if you take the limit as $T \to \infty$, it will go to zero.

The Bellman Optimality equations are given by

$$V^\pi(s) + \rho^* = \max_{Q} R(s,a) + \sum_{s'} P(s,a,s')V^*(s')$$

which is equivalently

$$V^* + \rho e = LV^*$$

In this case, you will never get $v^k - v^{k-1}$ is small in value iteration. Instead, you check whether the span (max minus min) of $v^k - v^{k-1}$ is small.

We now discuss value iteration for the average case. Define

$$\gamma = \max_{s,s',a,a'} \left[\gamma - \sum_{j \in S} \min\{P(s,a,j)\} \right] < 1$$

This is a weird definition, but it makes it work to prove contraction in the average case.

4 LECTURE 3: RL Algorithms

We know that we can apply something like dynamic programming in the setting where we know the model. We have a direct way to compute optimal values, but all this requires us to know the L operator. Now, reinforcement learning is the problem setting where you don't know L. You know the state space and the action space, but not L: the reward vector and the transition probability. However, you can observe: You can go in a state, observe the action for that state-action. You can also observe a tradition. That allows you to try explicitly or implicitly learn the model L. We will look at two kinds of methods: indirect methods and direct methods.

Indirect methods are model based: You explictly learn the model and calculate an optimal policy (R-MAX).

On the other hand, direct methods are directly approximating the procedures we say before: You try to directly approximate the update, the L operator. Apply the L operator approximately. These are called "approximate-DP-based methods". Examples are Q-learning, TD-learning.

More recently, people comfortable with supervised learning work on "direct policy search" methods. Here you think of the optimal policy as a prediction, and everything you observe as samples of that prediction. You try to directly fit the policy to your samples. You try to see what is the best policy given the samples. A popular example is policy gradient. In the purest form, you ignore the dynamic programming structure and think of everything as an observation. And of course, you see people who claim neither of these methods are good alone, so people say to combine them: these are the "actor-critic methods". You have DP-based methods to approximate gradients in policy gradient.

Suppose you have a simulator of a game. You can't just say, "let's go to a certain state of a game" - you don't know the policy to get a sample from a certain state with a certain action. You have some sampling power, but you can't generate any sample you want! Going to a state to generating a sample may not be reasonable. Maybe you can only start in 5 states, and then you have to get to different states to general samples, even if you have a virtual simulator. We will see that TD-learning was the first idea that handled this problem, that you can't go to arbitrary state and generate arbitrary samples. This was one of the first ideas in reinforcement learning.

4.1 TD Learning

This was introduced by Rich Sutton in 1988. Let us say we have a policy π and we want to evaluate $V^\pi(S)$ for all S. We want to know the asymptotic reward. For the rest of the discussion, we will assume the discounted reward case. This is the most popular regime in the RL literature. We have a stationary policy π. We want to simply evaluate the value of the policy. This was easy if we knew the model, we even had a formula:

$$V^\pi = R^\pi + \gamma P^\pi V$$

where $R^\pi(s) = R(s, a(s))$ and $P^\pi(s, s') = P(s, \pi(s), s')$ is a transition matrix. Then you can get it by simply $(I - \gamma P^\pi)^{-1} R^\pi$. If you don't want to use inverse, you can use value iteration by applying $v^k = L^\pi v^{k-1}$ if you knew L^π operator. Now the question is, if you didn't know R^π and L^π, how will you evaluate? One way to do this is to directly use samples. For every policy in state s, you can play the policy and see what happens. Do sample average approximations, this is called Monte Carlo – this is by execution. The problem with this approach is a) this is very slow, high variance. You'll get one sample by playing one policy for a long time, and doing it again and again you'll get $V^\pi(S)$ for one state. And you have to do this again and again. You're not re-using your previous calculations at all. Intuitively, dynamic programming should help here also. Also, going to every possible state whenever you want, even in a simulator, is not necessarily possible. These problems you want to make more practical in the following setting. You can always reset your simulator into some distribution of starting states. You want to evaluate the policy from starting state $s_1 \sim D$. Eventually we want to evaluate for all S. The structure I want to the algorithm to work in is called "episodic structure". In episodes we reset our game to the starting state. In each episode, we take action and observe. Eventually we terminate and reset back to the starting state. We want any estimation algorithm to work in this framework. All algorithms we will try to fill in the structure.

Definition 4.1. TD Learning Algorithm (Sutton 1988).
Given starting state distribution $s_1 \sim D$, repeat:

(a) Episode $e = e + 1$. Reset $s_1 \sim D$.

(b) Repeat $t = 1, 2, 3, \cdots$: Observe s_t, take action a_t, observe s_{t+1}, reward r_t.

(c) Update $\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha_t s_t$.

for some number of episodes.

How do we use the state and reward to estimate $V^\pi(s)$ for all s? We want to approximate these equations:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s, \pi(s), s') V^\pi(s')$$

We get samples, and maintain $\hat{V}(s)$. First observe $r_t+\gamma\hat{V}(s_{t+1})$. This quantity is an unbiased estimate of the RHS of the equation above. That is,

$$\mathbb{E}\left[r_t+\gamma\hat{V}(s_{t+1})|s_t,a_t,\hat{V}\right]=R(s_t,a_t)+\gamma\sum_{s'}P(s_t,a_t(s_t),s')\hat{V}(s')$$

Now, think of $z_t=r_t+\gamma\hat{V}(s_{t+1})$ as a sample, and we want this to match $\hat{V}(s_t)$. We can try a least squares estimate to minimize the square loss:

$$\frac{1}{2}\left(\hat{V}(s_t)-\left(r_t+\gamma\hat{V}(s_{t+1})\right)\right)^2$$

How do we use the ideas of supervised learning here? So we can look at $\sum_t\frac{1}{2}\left(\hat{V}(s_t)-z_t\right)^2$. Thinking of SGD, we can calcluate the gradient with respect to $\hat{V}$. We have

$$\frac{\partial\text{loss}}{\partial\hat{V}}=\hat{V}(s_t)-z_t$$

and thus our update is

$$\hat{V}(s_t)\leftarrow\hat{V}(s_t)-\left(\hat{V}(s_t)-z_t\right)a_t=\hat{V}(s_t)(1-\alpha_t)+\alpha_t(r_t+\gamma\hat{V}(s_{t+1}))$$

We can think of this as an estimate of the next state value using "one lookahead". You move towards the target estimate using a single look ahead. Why doesn't this fit the bill of supervised learning? Well, z_t is not an observation or label: It itself depends on $\hat{V}$. This is called bootstrapping: You are bootstrapping your prediction, you're creating the prediction. If your prediction is bad, the "label" itself is bad.

The name "temporal-difference" learning comes from writing the equation in another way: We can update

$$\hat{V}(s_t)+\alpha_t\delta_t$$

where δ_t is the *temporal difference*: difference between current and one lookahead:

$$\delta_t=r_t+\gamma\hat{V}(s_{t+1})-\hat{V}(s_t)$$

This version of TD learning uses one lookahead, but there are other variants of TD learning which use more lookaheads. This may look arbitrary now, and it should, and it is just one idea to approximate the updates. We will see why it converges soon.

What else can we do? You are using one lookahead estimate. You are constructing your target using just one lookahead. Why just one? You have only one true observation.

One issue is for rarely visited states, it will have high-variance since it did not get many updates. This is where the issue of exploration will come. How do you ensure that all states will get enough samples.

Now let's try to improve this, this is wasteful, why should we just look at one lookahead? One lookahead is not accurate. Let's instead go all the way and use z_t as $r_t+\gamma r_{t+1}+\gamma^2r_{t+2}+$

$\cdots$. Basically you get to a certain state, then you have a simulator where you can execute a policy ahead at that state for some time and compute its reward. You can do this for a very long time even. You could do it until γ becomes very small – this is like integrating the Monte Carlo method with this approach. The Monte Carlo method uses effectively infinite lookahead to compute z_t – there, you are not bootstrapping at all. In the TD0 method you are not simulating, and thus have more inaccurate "labels". So which is better? You will see a mix of these two, which is called TD(λ), where you do an intermediate amount of lookahead. Precisely,

$$TD(\lambda) : z_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t_n} + \gamma^n \hat{V}(s_{t+n})$$

Bootstrapping is not just because you want to avoid unrolling the estimate of z_t, and only do one look-ahead. Bootstrapping can also help: You are *re-using* what you have already computed. Once your $\hat{V}$ becomes good, it helps to use it because you can use it to improve your estimate for other states. This is basically using the "dynamic programming" part of the structure of the problem. So sometimes it might be better to do Monte Carlo and sometimes it might be better to do value. People tend to initially use longer lookahead, but later on when you are more confident, you can do more bootstrapping. Initially you want to use true rewards as much as possible, but later on, you can do dynamic stuff – you can change how much you lookahead with time as well, and then you get a lot of papers.

So thus far we have just estimated the value of a policy, how about finding a good one? We can do some policy iteration. This gives us the policy valuation step. Now we need a way to improve the policy. We need a Q function to improve the policy, since we need both a state and an action (so that we can modify the action). We can replace the $\hat{V}$-update step in the TD Learning algorithm instead with $\hat{Q}(s, a)$. We can still use one policy: thus we look at $\hat{Q}^\pi(s, a)$. This is equal to $R(s, a) + \sum_{s'} P(s, a, s') V^\pi(s')$. We will update

$$\hat{Q}^\pi(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha_t \delta_t$$

where $\delta_t = r_t + \gamma \hat{Q}(s_{t+1}, \pi(s_{t+1})) - \hat{Q}(s_t, \pi(s_t))$. The obvious problem with this approach is that these updates will only happen for the actions you play with the policy you're using. So if you want to make sure other actions also get played beyond the policy you're using, you have to do exploration. If you can get these Q values, then you can replace with the aforementioned update. After an episode, you'll get to improve a policy.

First, you'll evaluate the policy using the $\hat{V}$ approximation. Then you can improve the policy:

$$\pi^{k+1}(s) = \text{argmax}_a Q^{\pi^*}(s, a)$$

So your initial policy should be diverse so that you have good estimates for all actions. You should also add some exploration to the policy, not just take the argmax:

$$\pi^{k+1}(s) = (1 - \epsilon)\text{argmax}_a Q^{\pi^*}(s, a) + \epsilon \text{ random action}$$

where ϵ is a small probability. This is an ϵ-greedy choice. This way you don't affect the Q-value a lot, but it lets you update the Q-values for those other actions.

We will later see a weak kind of convergence result: If your policy visits everything in finite time, then you will converge. This basically says you are using some kind of exploration; how you are doing it is a different matter.

4.2 Q Learning

Q Learning is very close to what we just saw. We will basically combine the two steps. Q-learning will combine the two steps and directly try to learn the Q-values on optimal policies, analagously to value iteration combining the two steps to calculate optimal values.

We have

$$V^*(S) = \max_a R(s,a) + \sum_{s'} P(s,a,s') \max_{a'} Q^*(s',a')$$

We can call the inside of the $\max_a$ as $Q^*(s,a)$. Thus,

$$V^*(S) = \max_a Q^*(s,a)$$

From this let's make things approximate. We have as our update:

Definition 4.2. Q-learning Update.

$$\hat{Q}(s_t,a_t) \leftarrow (1-\alpha_t)\hat{Q}(s_t,a_t) + \alpha_t \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1,a'}) \right)$$

This is basically Q-learning. Then once you have your Q values, you can decide optimal actions. Your policy is just going to be $\text{argmax}_a \hat{Q}(s,a)$. So you can choose the greedy choice, the ϵ-greedy choice (randomly pick a random action with small probability), or some other choice.

5 LECTURE 4: Convergence results and function approximation

To recap: we finished learning about tabular Q-learning and TD learning. Meaning that we considered the case where we can make a table and learn all the Q-values and so on. This implicits assumes the number of states are not so large so that we can make the tables, and that we can explore enough to cover all the actions.

Today, we will seem some convergence results – why do these methods make sense? They will not be completely satisfactory – they will only hold under the assumption that there is enough exploration going on. Then we will introduce a scalable version of Q-learning: Namely, Q-learning with function approximation. You don't want to explicitly separately learn a Q-value for every state. You want to do some transfer learning between one state and another, and say something like similar states have similar Q-values. In particular, this will use deep learning as a subroutine. We can use deep learning to define these similarities. This is not a deep learning course, but I'll tell you enough to be able to use it.

5.1 Convergence Theorems

Let us remember what the methods look like.

Definition 5.1. Tabular Q-learning method.
At every step t, we picked an action a_t on the current state and observe reward r_t and next state s_{t+1}. Then we updated the Q values: $\hat{Q}(s_t, a_t)$. We updated it by taking the previous estimate and stepping in the direction of the temporal difference $\alpha_t\delta_t$, where $\delta_t = r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a')$. This is essentially like a gradient step for minimizing the loss with stepsize α_t. The convergence result will be a condition on the stepsize and how you pick the actions. Basically, you should pick actions in such a way that all states and actions are explored. Then we will see that $\hat{Q}$ will converge to true Q-values with probability 1 for all states and actions.

In general, we may not need for every state-action Q value to converge – bad states/actions are not that useful. We also may not need to explore everywhere – these assumptions in the results should be subject to scrutiny. Often very strong results are proved by making these very strong assumptions (converge everywhere, if explore everywhere infinitely often).

Theorem 5.2. *Watkins and Dayan* 1992*.*
If you have bounded rewards $|r_t| \leq k$ *and learning rates* $0 \leq \alpha_t \leq 1$ *and*

$$\sum_{i=1}^{\infty} \alpha_{n^i(s,a)} = \infty$$

$$\sum_{i=1}^{\infty} (\alpha_{n^i(s,a)})^2 < \infty$$

for all s, a *then*

$$\hat{Q}_t(s, a) \to Q(s, a)$$

as $t \to \infty$ *with probability* 1*.*

Q-learning turns out to be an instance of a larger class of problems called stochastic approximation algorithms. TD-learning is another example. So it's useful to look at how you would prove a stochastic approximation algorithm converges since it'll prove for many things you'll see in literature. Let's look at stochastic approximation method.

Definition 5.3. Stochastic Approximation Method.
This is a general form of approximation algorithm where you have some equation like $h(\theta) = 0$ and you are trying to find θ. What you have available is some noisy observation of $h(\theta)$: $Z_1, \cdots, Z_n, \cdots$ where $Z_n = h(\theta_n)$+noise where the noise is zero-mean (unbiased estimate).

Now does there exist a method such that we can find a point θ? There is a general method to do this.

The method is as follows, due to Robbins and Monro 1951:

$$\theta_{n+1} = \theta_n + \alpha_n Z_n$$

θ_n here can be a vector. We can modify the setting so that you only get to see one component of Z_n at a time: $Z_{n,i}$. The asynchronous version just updates the i^{th} component:

$$\theta_{n+1}[i] = \theta_n[i] + \alpha_n Z_n[i]$$

Example 5.4. Estimating averages.
Suppose you want to estimate the mean of a random variable: $h(\theta) = \theta - \mu$. Suppose you observe i.i.d. samples of Y_n such that $\mathbb{E}[Y_n] = \mu$. Let $Z_n = Y_n - \theta_n$. Then $\mathbb{E}[Z_n|\theta_n] = \mu - \theta_n$. Then stochastic approximation method says

$$\theta_{n+1} = \theta_n + \alpha_n(Y_n - \theta_n) = (1 - \alpha_n)\theta_n + \alpha_n Y_n$$

So this is nothing but a running average if we choose $\alpha_n = 1/(n+1)$.

Example 5.5. TD-learning as stochastic approximation.

Let us look at TD learning for policy evaluation. We had

$$\hat{V}_{n+1}(s) \leftarrow \hat{V}_n(s) + \alpha_n(r_t + \gamma\hat{V}_n(s_{t+1}) - \hat{V}_n(s_t))$$

We have $\theta_n[s] = \hat{V}_n(s)$. We are trying to find a fixed point of the Bellman equation. Let us set $h(\theta)$ so that it is a fixed point of the equation. To do this, we set

$$h(\theta) = r_\pi - \gamma P^\pi\theta - \theta$$

Thus

$$Z_n[s] = r_n + \gamma\theta_n(s') - \theta_n(s)$$

In TD-learning, we have $\theta \in \mathbb{R}^{|S|}$.

Example 5.6. Q-learning as stochastic approximation.
Here, we have the same setup except $\theta \in \mathbb{R}^{|S|\times|A|}$.

$$h(\theta) = r(s,a) + \gamma\sum_{s'} P(s,a,s')\max_{a'}\theta(s',a') - \theta(s,a)$$

Basically if you plug in $\theta = Q^*$, you'll get back the Bellman equations. This is just written in a form so that we get the form $h(\theta) = 0$. So if we look back to our Q-learning method, we'll see δ_t plays the role of Z_n. To get an unbiased estimate of θ,

$$Z_n = r_n + \gamma\max_{a'}\theta[s',a'] - \theta(s,a)$$

and its expected value will be exactly the RHS of the $h(\theta)$ definition:

$$\mathbb{E}[Z_n|\theta_n] = r_n + \sum_{s'} P(s,a,s')\max_{a}\theta_n(s',a') - \theta_n(s,a)$$

So you can think of all these approaches as stochastic approximation methods. Updates are stepsize times estimate of $h(\theta)$. For TD learning, θ is just the V function, and for Q learning, θ is just the Q function. So we just need to analyze the overarching algorithm to show these methods work.

Theorem 5.7. *Stochastic Optimization Convergence. (Proposition 4.5 of Bertsekas and Tsitskilis (Neurodynamic Programming))*
Assumptions: First assumption is on noise: We said $Z_n = h(\theta_n) + \omega_n$. We want the noise to be mean zero. $\mathbb{E}[\omega_n | F_{n-1}] = 0$, where F_{n-1} is the filtration defined by everything you have done so far ($\{\theta_1, \alpha_1, \omega_1, \cdots, \theta_n, \alpha_n\}$), but before you have seen the noise ω_n. We also make the assumption that the noise is subgaussian. Additionally, assume that $h(\theta) = L\theta - \theta$, where L is a contraction mapping ($\|L\theta - L\theta'\|_\infty \leq \gamma\|\theta - \theta'\|_\infty$). Then, $\theta_n \to \theta^*$ as $n \to \infty$ with probability 1. You can extend to asynchronous setting with all these proofs as long as you assume that all components of θ_n are updated for infinitely long. There are many versions of this theorem.

We will prove an easier version. The proof becomes much easier if we assume that L is a contraction mapping except in the $\|\cdot\|_2$ norm instead of $\|\cdot\|_\infty$ (it is continuous).

Theorem 5.8. *Assume $h(\theta) = L\theta - \theta$ where L is a contraction mapping w.r.t. $\|\cdot\|_2$. Then, $\theta_n \to \theta^*$ as $n \to \infty$ with probability 1.*

Proof. We get that additional conditions are true directly from the fact that we are using the $\|\cdot\|_2$ norm:

$$h(\theta_n)^T(\theta_n - \theta^*) \leq -c\|\theta_n - \theta^*\|_2^2$$

and

$$\|h(\theta_n)\|_2 \leq (1+\gamma)\|\theta_n - \theta^*\|_2$$

We will use these conditions to prove the theorem (In $\|\cdot\|_\infty$ case, we would also end up showing these conditions hold and use them in the proof – it is more involved to show this for $\|\cdot\|_\infty$).

To show the first condition holds, we have

$$(L\theta_n - \theta_n)^T(\theta_n - \theta^*) = (L\theta_n - \theta^*)^T(\theta_n - \theta^*) + (\theta^* - \theta_n)^T(\theta_n - \theta^*) \leq \gamma(\theta_n - \theta^*)^T(\theta_n - \theta^*) - \|\theta_n - \theta^*\|_2^2 = -(1-\gamma)\|\theta_n$$

since $L\theta^* = \theta^*$ and applying contraction. To show the second condition holds, we have

$$\|h(\theta_n)\|_2 = \|L\theta_n - \theta_n\| = \|L\theta_n - \theta_n - (L\theta^* - \theta^*)\|_2 \leq \|L\theta_n - L\theta^*\|_2 + \|\theta_n - \theta^*\|_2 \leq \gamma\|\theta_n - \theta^*\| + \|\theta_n - \theta^*\|_2 = (1+\gamma$$

Now we are going to relate the gap $(\theta_{n+1} - \theta^*)^2$ with $(\theta_n - \theta^*)^2$, and show that the gap decreases as n increases. We have

$$\mathbb{E}\left[(\theta_{n+1} - \theta^*)^2 | F_{n-1}\right] = \mathbb{E}\left[\|\theta_{n+1} - \theta_n + \theta_n - \theta^*\|_2^2 | F_{n-1}\right] = \mathbb{E}\left[\|\theta_{n+1} - \theta_n\|^2 + 2(\theta_{n+1} - \theta_n)(\theta_n - \theta^v) + (\theta_n$$

Call $b_n = (\theta_n - \theta^*)^2$. Then,

$$b_{n+1} - b_n = \mathbb{E}\left[\alpha_n^2\|Z_n\|_2^2 | F_{n-1}\right] + 2\mathbb{E}\left[\alpha_n(h(\theta_n) + \omega_n)(\theta_n - \theta^*) | F_{n-1}\right] = \mathbb{E}\left[\alpha_n^2\|h(\theta_n)\|_2^2 + \alpha_n^2\omega_n^2 + \alpha_n h(\theta_n)^T(\theta_n$$

Now we can apply the extra conditions we got to bound the cross terms and the additional terms. The above is

$$\leq (1+\gamma)^2\alpha_n^2\|\theta_n - \theta^*\|_2^2 + \alpha_n^2\omega_n^2 - c\alpha_n\|\theta_n - \theta^*\|_2^2$$

So the intuition is you have these α_n^2, which goes down fast even though α_n doesn't go down fast. Since the α_n^2 sum to a constant, the individual terms must go to zero. So at some point, $(1+\gamma)^2\alpha_n^2\|\theta_n - \theta^*\|_2^2$ will become smaller than $c\alpha_n\|\theta_n - \theta^*\|_2^2$. So $(1+\gamma)^2\alpha_n^2\|\theta_n - \theta^*\|_2^2 - c\alpha_n\|\theta_n - \theta^*\|_2^2$ summed up at some point will become negative, and the only thing left over is the sum over $\alpha_n^2\omega_n^2$. Thus

$$b_{n+1} - b_n \leq \sum_n \alpha_n^2\omega_n^2 < \infty$$

So we are showing that the difference is small.

□

5.2 Function Approximation for Q-functions

Instead of learning Q-function explicitly, we represent states and actions via some featurization, and learn some parametrized Q function $Q_\theta(s,a)$, where θ are the parameters of the model, and (s,a) are featurized according to the domain. For instance, we could have

$$Q_\theta(s,a) = \theta_0 + \theta_1 f_1(s,a) + \cdots + \theta_n f_n(s,a)$$

So the problem is transformed into learning a good θ over time. How would we apply the Q-learning methods we have seen to do that? Recall the derivation of Q-learning. The original reason for devising Q-learning is we wanted to learn an approximation method for Q-value iteration. We want to approximate the Bellman equation:

$$Q^*(s,a) = R(s,a) + \gamma\sum_{s'} P(s,a,s')\max_{a'} Q^*(s,a')$$

If we have $\hat{Q}$ right now, what values of $\hat{Q}$ will minimize the difference in terms of squared loss? Since we don't directly have these things, we will use our current sample as a prediction of this – we'll play a state and action, and see the subsequent state. We have

$$\hat{Q}(s,a) - \left(r_t + \gamma\max_{a'}\hat{Q}(s_{t+1},a')\right)$$

where the second term is Z_t. We want to minimize this difference (assuming ℓ_2 norm loss). Thus, we can update with respect to this gradient: $\hat{Q}(s,a) - \alpha_t(\hat{Q}(s,a) - Z_t)$. Using our current value of θ, we need to construct a target label, and construct an update so that the loss is minimized. Thus, we have

$$\ell_\theta(s,a,s') = \left(Q_\theta(s,a) - \text{target}(s')\right)^2$$

Unlike supervised learning, the target isn't a true label – we are just taking it to be what it should be. We are pretending it is a label (this is a constant, incidentally). Then we take the gradient with respect to θ, and do a gradient descent step on θ. For instance, we could imagine that Q_θ is linear: Let f be the featurization of s, a, then perhaps we have $Q_\theta(s, a) = \theta^T(f_{s,a})$. This is where RL gets connected with supervised learning methods. For instance, instead of a linear function class, Q_θ could be the function class of deep networks. So we will try to construct a deep network so that whenever you get an s, a, it will try to output $Q_\theta(s, a)$. We can then apply the supervised learning approaches from deep learning by pretending the target is a label.

However, the main thing, to re-iterate, is that the target is NOT a true label – we are bootstrapping to create it. A lot of the empirical papers out there are focusing on stablizing the estimation of that target.

6 LECTURE 5: (TO FILL IN FROM ONLINE NOTES)

7 LECTURE 6: (TO FILL IN FROM ONLINE NOTES)

8 LECTURE 7:

We have been talking about policy gradients. Here, we have $\pi_\theta(s, a)$, and we would like to get the gradient of the gain $\nabla_\theta \rho(\pi_\theta)$. One can derive a formula for infinite horizon, either discounted or average reward case. The formula is

$$\sum_s d^\pi(s) \sum_a Q^{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(s, a)$$

where d is the distribution of states for the stationary distribution. Then,

$$d^{\pi_\theta}(s) = \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{P}\{s_t = s\}$$

This is not exactly a distribution, it sums to $(1-\gamma)d^\pi(s)$, so we have to normalize it. To avoid writing $1-\gamma$ everywhere, we will use somewhat bad notation and pretend that it's a distribution and write $s \sim d^\pi(s)$, referring to the unnormalized distribution.

We saw **actor-only** method, where we estimate this only using the rewards. YOu can directly estimate the Q-value using rewards we saw.

We will now look at the actor-critic method, where we use a function approximation for both the policy and the Q-values. So the policy iteration method that we are aiming for, using these theorems, is as follows:

(a) Have current policy π.

(b) Do policy evaluation (approximately, with $Q^{\pi_{\theta_k}}(s, a) \leftarrow f_w(s, a)$).

(c) Use this to do policy improvement: $Q_{k+1} \leftarrow Q_k + \alpha_k \nabla_\theta f(\pi_\theta)|_{\theta=\theta_k}$.

If both of our function approximations are extremely expressive, there is no issue. However, if the first function approximation is not good enough, there might be issues.

We have two conditions :

(a) We want w to be $\min_w \mathbb{E}_{s\sim d^\pi(s), a\sim\pi_\theta(s)}\left[(Q^{\pi_\theta}(s,a) - f_w(s,a))^2\right]$, or

$$\mathbb{E}_{s\sim d^\pi(s), a\sim\pi_\theta(s)}\left[(Q^{\pi_\theta}(s,a) - f_w(s,a))\nabla_w f_w(s,a)\right] = 0$$

(b) The second condition is just that

$$\nabla_w f_w(s,a) = \nabla_\theta \log(\pi_\theta(s,a))$$

for all s, a. That is, we can relate ws to θs. (The gradient estimate will be unbiased).

The theorem due to Sutton is as follows:

Theorem 8.1. *(Sutton et. al., 1999)*

$$\nabla_\theta f(\pi_\theta) = \mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta(s)}\left[f_w(s,a)\nabla \log \pi_\theta(s,a)\right]$$

This actually provides a very strong restriction on the function classes which satisfy this: They need to be linear. Otherwise, you have no idea how much error you are introducing.

Proof. Given the assumptions, this proof will almost follow directly.

FIX THIS: SOMETHING WRONG (first equality wrong)

$$\begin{aligned}\nabla_\theta f(\pi_\theta) &= \mathbb{E}_{s\sim d^\pi(s), a\sim\pi_\theta(s)}\left[(Q^{\pi_\theta}(s,a) - f_w(s,a))\nabla_w f_w(s,a)\right] \\ &= \mathbb{E}_{s\sim d^\pi(s), a\sim\pi_\theta(s)}\left[(Q^{\pi_\theta}(s,a) - f_w(s,a))\nabla_\theta \log(\pi_\theta(s,a))\right]\end{aligned} \tag{5}$$

Suppose $\pi_\theta(s,a) = \frac{e^{\theta\cdot\phi_{s,a}}}{\sum_{a'} e^{\theta\cdot\phi_{s,a'}}}$. Then $\nabla_\theta \log \pi_\theta(s,a) = (\phi_{s,a} - \sum_{a'} \phi_{s,a'}\pi_\theta(s,a)) = \nabla_w f_w(s,a)$. This is a linear policy setting $f_w(s,a) = \phi_{s,a}^T\theta$.

Now suppose we have a Gaussian policy: $\pi(s) = \mathcal{N}(\phi_S^T\mu, \sigma^2)$. Then $\nabla_\theta \log(\pi_\theta(s)) = \frac{(\theta-\phi_S^T\mu)}{\sigma^2}\cdot\phi_S$. Again it'll be linear in θ. There is a paper which shows that linear is the only thing that works. □

This is the only pure theorem we know, where we don't introduce any errors with this approach. Basically take this theorem with a grain of salt, since you're not going to satisfy condition 1. That will already introduce some error. And then you will not have the second point as well. So lots of errors get introduced here.

So what happens if you approximate these things? What happens if you don't use the exact gradient? If you use the exact gradient, you do converge. There's a theorem which says that if you use policy iteration method, with this kind of update, then

Theorem 8.2. *Policy iteration convergence guarantees.*
If you have $\sum \alpha_k^2 < \infty$ for step sizes α_k, and you have a mild condition where you compute w as a stationary point, and you make sure your function approximation is linear (satisifes the second point), and then you update, then you can guarantee that you have convergence as long as you have a mild condition on the second derivative of the policy (this is not actually such a big deal. It's a smoothness constraint.): $\frac{\partial}{\partial\theta_i\partial\theta_j}\pi_\theta(s,a) < \infty$, then

$$\lim_{k\to\infty} \nabla_\theta \rho(\pi_k) = 0$$

This is basically stochastic approximation method, and you satisfy the constraints you need for it to converge, so you do.

Now we'll ask two questions: Is it ok to just converge to this point? How good will the actual policy be? How long will it take to converge? Do we have an idea of how much improvement happens in every step? What can we expect?

We'll discuss two papers today. One is by Sham Kakade and John Langford from ICML 2002: "Approximately optimal approximate RL". If you have approximations everywhere, what happens? Does the method still converge with reasonable speed? This is a really good paper in that it doesn't talk about sample complexity and $1-\delta$ type guarantees; it's really studying the stability problem of what your gradients become. Even if your estimation is very very correct, the process is very unstable. The gradients can become completely uninformative in certain situations. They will not be informative for a long time. They give very conceptual ways of resolving this problem, and give an algorithm called Trust-Region Policy Optimization (TRPO (Schulman, Abbeel, etc. 2016)). This is a more practical version of the algorithm they introduce. If you have to pick a paper to read, pick a paper by Sham Kakade – they are all wonderful papers, very conceptual.

8.1 Approximately optimal approximate RL

It's possible to construct a simple example such that you'll have a 0 gradient most of the time by estimation, when the true gradient is $1/2^n$– and the kicker is that this is a good estimate! Yet it is still non-informative here. You're looking at the distribution of states by the current policy. The difference of gains between two policies depends on the distribution of states of the target policy! So there is a measure mismatch. You have policy π, and the policy you want to go to. You are estimating $\nabla\rho(\pi)$, which depends on π, but the difference between $\rho(\pi)$ and $\rho(\pi')$ depends on π'! So this is a measure mismatch. If you started with a policy that is not optimal, and a state distribution that is not favorable, things could be very bad – things will get a lot worse before getting better in perhaps an exponential amount of time. Exploration is not the main issue here – recall we had a uniform distribution at the start! The issue is that we are looking at a local approximation with the gradient. We are only looking at the current distribution of states. So the approximation is only correct very locally, and is not enough to improve the policy by a large amount. So let us formalize this measure mismatch issue.

We are trying to approximate $\rho(\tilde{\pi}) - \rho(\pi)$ by gradient, and we were trying to do this locally (recall again that this is just a fixed point we are trying to get to). What is the true gap? It's given by the following theorem:

Theorem 8.3 (Kakade and Langford 2002)**.** .

$$\rho(\tilde{\pi}) - \rho(\pi) = \sum_s d^{\tilde{\pi}}(s) \sum_a \tilde{\pi}(s,a) A^{\pi}(s,a)$$

where $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$. This is known as ***advantage****. The main difference from our policy gradient approximation from before is that we are evaluating on the stationary distribution of the new policy, $d^{\tilde{\pi}}$ and the advantage of the old policy.*

Proof. So we need to look for a new policy, not by local gradient improvement, but in a region around the correct policy to search for. This will give us a quantification of improvement more than anything else. We have

$$\rho(\tilde{\pi}) = V_{\tilde{\pi}}(s_1) = \mathbb{E}_{a_t \sim \tilde{\pi}(s_t)}\left[\sum_{t=1}^{\infty} \gamma^{t-1} R(s_t, a_t) + V^{\pi}(s_t) - V^{\pi}(s_t)\right]$$

$$= \mathbb{E}_{a_t \sim \tilde{\pi}(s_t)}\left[\sum_{t=1}^{\infty} \gamma^{t-1}\left[R(s_t, a_t) + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right] + V^{\pi}(s_1)\right]$$

$$= \mathbb{E}_{a_t \sim \tilde{\pi}(s_t)}\left[\sum_{t=1}^{\infty} \gamma^{t-1}\left[Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)\right] + V^{\pi}(s_1)\right]$$

$$= \mathbb{E}_{a_t \sim \tilde{\pi}(s_t)}\left[\sum_{t=1}^{\infty} \gamma^{t-1} A^{\pi}(s_t, a_t)\right] + \rho(\pi)$$

□

So, how might we want to do policy improvement? In this framework, we might think of a conservative policy update of the form $\pi_{\text{new}}(a,s) = (1-\alpha)\pi(a,s) + \alpha\pi'(a,s)$. We will first discuss how to select α. The second idea is how to pick this new policy. So what happens if we change π to π_{new}. How much will the policies improve compared to π, depending on α? Let us define

$$\nabla_{\alpha}\rho(\pi_{\text{new}}(\alpha)) = A_{\pi}(\pi') = \sum_s d^{\pi}(s) \sum_a \pi'(s,a) A^{\pi}(s,a)$$

We still haven't moved to the new distribution completely, but we are using advantage of π to define this improvement. First let's show this. The improvement in gain depends on this term for π', so we should pick some policy π' which has large value of the above. That's what the ϵ-greedy approach will be. Now you're thinking about all possible policies π'. That opens up the scope to think more generally about arbitrary policies. It allows us to think more broadly instead of just locally optimal policies. We will prove

$$\nabla_\alpha \rho(\pi_{\text{new}}(\alpha))|_{\alpha=0} = \sum_s d^{\pi_{\text{new}}}(s) \sum_a Q^{\pi_{\text{new}}(\alpha)}(s,a) \nabla_\alpha \pi_{\text{new}}(\alpha)(s,a)$$

where we can just use the theorem of Sutton to compute the gradient, where we are taking α in place of θ (in the statement of the theorem at the start of this section) to compute the gradient. This equals

$$\begin{aligned} &\sum_s d^\pi(s) \sum_a Q^\pi(s,a) \left(\pi'(s,a) - \pi(s,a)\right) \\ &= \sum_s d^\pi(s) \sum_a Q^\pi(s,a)\pi'(s,a) - \sum_a Q^\pi(s,a)\pi(s,a) \\ &= \sum_s d^\pi(s) \sum_a Q^\pi(s,a)\pi'(s,a) - \sum_a Q^\pi(s,a)\pi(s,a) \end{aligned}$$

So we want to get π' which has a large advantage. A greedy choice of π' is just $\text{argmax}_{\pi'} A_\pi(\pi')$. In fact you can use on-policy examples to do this; you're just re-weighting the action distribution to do this.

Theorem 8.4.

$$\rho(\pi_{\text{new}}) - \rho(\pi) \geq \alpha A^\pi(\pi') - \alpha^2 2\gamma\epsilon$$

where $\epsilon = \max_s \mathbb{E}_{a \sim \pi'(a,s)} \left[A_\pi(s,a)\right]$.

This is true for any choice of π', and you'll get roughly A^2/R improvement. If you want to get a lot of improvement, you'll want to choose the π' that has the best improvement possible, which is the greedy choice. This theorem is the key thing in showing TRPO works.

Proof. This lets you choose your region to find your good π'. It opens up the possibility for lots of algorithms beyond simple gradient descent for choosing your π'. We have from the previous theorem

$$\rho(\tilde{\pi}) - \rho(\pi) = \sum_s d^{\tilde{\pi}}(s) \sum_a \tilde{\pi}(s,a) A^\pi(s,a)$$

Now, we have $\tilde{\pi} = (1-\alpha)\pi + \alpha\pi'$. Let's open this up:

$$= \mathbb{E}_{s_1,\cdots,s_T \sim \tilde{\pi}} \left[\sum_{t=1}^\infty \gamma^{t-1} \sum_a \tilde{\pi}(s_t,a) A^\pi(s_t,a)\right]$$

Then we consider coupling the two trajectories: what would have happened if we used the new policy versus the old policy? How does the new policy differ from the old policy? With probability $(1-\alpha)^t$, there is no difference between the two policies. Let us consider up to step t, you behave as if you were using old policy, in the other situation, you don't. Let's divide the expectation into two cases:

$$\sum_t \mathbb{E}\left[\cdot|\eta_t = 0\right] \mathbb{P}\left\{\eta_t = 0\right\} + \mathbb{E}\left[\cdot|\eta_t = 1\right] \mathbb{P}\left\{\eta_t = 1\right\}$$

where η represents the Bernoulli(p_t) coin tosses at each step, where $p_t = 1 - (1-\alpha)^t$. Thus, $\mathbb{P}\left\{\eta_t = 0\right\} = (1-\alpha)^t$, $\mathbb{P}\left\{\eta_t = 1\right\} = 1-(1-\alpha)^t$. Let's just focus on the first part, where you are perfectly coupled ($\eta_t = 0$): Then,

$$\sum_a (1-\alpha)\pi(s_t, a)A^{\pi(s_t,a)} + \alpha\pi(s_t, a)A^{\pi(s_t,a)}$$

We claim the first term is 0. Advantage is the difference between Q-values and value function. So if you take advantage and weight it by the same probability π, this becomes 0. So, we can always replace this by $\alpha\pi'$: We don't need to consider the $(1-\alpha)\pi$ part.

Now let's come back to the imperfectly coupled case. So we are replacing $d^{\tilde{\pi}}(s)$ with a mixture. We'll replace it with something expensive: The negative of the maximum possible advantage, which is $-\alpha\epsilon$, where ϵ is the maximum possible advantage. Therefore, we get

$$\alpha A_\pi(\pi') - \alpha A_\pi(\pi) - \alpha\epsilon p_t \geq \alpha A_\pi(\pi') - 2\alpha\epsilon p_t$$

Thus you get

$$\begin{aligned} &\alpha A_\pi(\pi') - 2\alpha\epsilon \sum_{t=1}^{\infty} \gamma^{t-1}(1-(1-\alpha)^t) \\ &= \alpha A_\pi(\pi') - 2\alpha\epsilon\left(\frac{1}{1-\gamma} - \frac{1}{1-\gamma(1-\alpha)}\right) \\ &= \alpha A_\pi(\pi') - \frac{2\alpha^2\epsilon\gamma}{(1-\gamma)(1-\gamma(1-\alpha))} \end{aligned}$$

□

Corollary 8.5. *So you want to choose α so that everything is non-negative. ϵ is the max possible advantage. You can think $\epsilon \leq R$. Then one choice of α which ensures your improvements will always be positive (in general, it doesn't have to be) is*

$$\alpha = \frac{A^\pi(\pi')(1-\gamma)}{4R}$$

Then your improvement is given by

$$\text{improvement} \geq \frac{A^2}{4R}(1-\gamma) - \frac{A^2}{|6R^2|}\frac{2\gamma R}{(1-\gamma)^2}(1-\gamma)^2 = \frac{A^2}{8R}$$

so you can get an improvement in a finite number of steps, as long as you have some advantage where A is advantage. So if you are 2ϵ away from optimal policy, that's enough.

If you are more than 2 or 3 ϵ away from optimal, there will be a policy with some advantage, and you'll get some improvement. So you can bound number of samples and so on in terms of ϵ. This is the intuition behind getting a definitive bound on policy improvement, since that tells you how much leeway you have in terms of approximation in terms of your policy.

Now suppose you had an ϵ-greedy policy chooser: You want to find $A_\pi(\pi') \geq \max_\pi(A_\pi(\pi')) - \epsilon/3$. This is the highlight of the TRPO paper. The algorithm is as follows:

(a) Given current policy π, find an $\epsilon/3$-greedy policy π'.

(b) Estimate $\hat{A}$ as $A_\pi(\pi')$ within $\epsilon/3$.

(c) If $\hat{A} \leq 2\epsilon/3$ stop. Otherwise, update $\pi_{\text{new}} = (1-\alpha)\pi + \alpha\pi'$.

So the thing about these mixture of policies is that you have to store all the policies you have. You'll be maintaining a mixture of policies that you keep appending to. This is one of the main complaints with which the TRPO paper starts: The mixture policies are difficult to maintain and implement; you want a parametric approximation. That's the nice thing about gradient descent. So can we instead get back to that parametric approximation instead of needing to maintain this mixture of policies. They show that you don't need to update policies in that way, and you can still update one policy: You just have to find the best policy in a certain reason.

I'll finish today's lecture by showing what this algorithm converges to. This algorithm will tell you you cannot get more than ϵ-advantage. You'll end with the case where your advantage is $< \epsilon$. No policy can improve your policy in terms of having advantage more than ϵ. So what does that mean in terms of gain?

Theorem 8.6. *Given policy π such that $\max_{\pi'} A_\pi(\pi') \leq \epsilon$,*

$$\rho_{\tilde{\mu}}(\phi^*) - \rho_{\tilde{\mu}}(\pi) \leq \frac{\epsilon}{1-\gamma} \|\frac{d_{\pi_1^*,\tilde{\mu}}}{d_{\pi,\mu}}\|_\infty$$

if you are allowed to use any distribution you want to start.

We know that $\mu(s) \leq d_{\pi,\mu}(s) = (I - \gamma P)^{-1}\mu(s) = \mu(s) + \gamma P\mu(s) + \gamma^2 P^2 \mu(s) + \cdots$, by Taylor expansion.

Two ideas will improve your approximations: Instead of looking at nearby policies, you look at arbitrary policies and measure them by looking at the advantage, which is more of a global measure. The key piece is to characterize how much improvement you get so that you can select a step size. The other problem is stopping: In gradient descent, you stop when gradient is really small, but you have no idea how bad you are when compared to optimal. If you stop, you can tell exactly how far you are – it's in terms of the distance between stationary distributions of the policies. But if you're allowed to use different starting state distributions, start with a uniform distribution over states, you're allowing some exploration and that will allow you to bound the ℓ_∞ term really well. In typical applications, that's not an option: But it does suggest you should randomize as much as possible.

9 Lecture 8: Guest Lecture (Krzysztof Choromanski, Learning compressed RL policies via gradient sensing)

This talk is about applications of structured pseudorandom matrices in reinforcement learning. It's along the evolution set of methods which are simple to implement and do as well as policy gradients. Let's start. We are interested in learning good quality policies $\pi : S \to A$. We will use a neural net to represent this map. The input layer is the state vector, the output layer is the action vector. We are interested in the blackbox optimization setting. The object of interest is the blackbox function which I will define in a moment. Optimizing this function will correspond to finding an optimal policy. It's the function $F : \mathbb{R}^d \to R$, which takes in policy parameters and outputs total reward. Our focus is on the case where we assume we have a good simulator for the agent (a robot).

If we are talking about blackbox optimization, we say we have a function which we can query (could be randomized output with noise); given the answers to the queries, we would like to get the optimal answer. Each query will correspond to a rollout of the simulator. We run this policy in the environment and this is the output of the function F. If you want to optimize some very complicated functions, who knows how it looks: First you might try to approximate the gradient. If it's smooth enough, then you might do some optimization procedure to do this gradient. The ancient way to do this which is inefficient is finite-difference method. This basically tells you how to estimate the gradient: perturb your policy along the canonical basis. You then run the simulator and collect the words, and then by doing standard finite difference you can approximate every element of the gradient vector. It's inefficient: At any given point of the iteration, you need $n + 1$ iterations of the rollout ($n =$dimension of policy). It's not robust to noise either.

A much more efficient version essentially conducts finite-difference calculations, but not in the canonical basis (look at random rotation; you still get good quality gradients; you also get good policies). This is related to Salimans et al (evolution strategies) from OpenAI, where they propose to replace directions of basis by Gaussian vectors. How does their work differ from our work? You can think of rotating the canonical basis, also rescale it: The **only difference** is that we condition on the directions to be pairwise orthogonal exactly. Exact orthogonality is key to getting better results. By using orthogonal vectors, you can get a better quality estimator of the gradient. In practice, you can do Gram-Schmidt on Gaussian matrix and re-normalize the lengths to get exactly othogonal basis.

What if the function isn't differentiable? We'll approximate the gradient of the Gaussian smoothing of a function. You average the values of the function in a neighborhood looking at random Gaussian directions. There are other possible smoothings, but this is pretty standard. This is also helpful for getting guarantees with gradient techniques.

If we take this definition of Gaussian smoothing, then the gradient of the Gaussian smoothing gives a derivative over the expectation. We can use Monte-Carlo approximation to get the gradient of Gaussian smoothing. We have several unbiased estimators: Evolutionary strategies style method:

$$\hat{\nabla}_N J(\theta) := \frac{1}{N\sigma}\sum_{i=1}^{N} F(\theta + \sigma\epsilon_i)\epsilon_i$$

Gradient estimator with antithetic pairs

$$\hat{\nabla}_N J(\theta) := \frac{1}{2N\sigma}\sum_{i=1}^{N} F(\theta + \sigma\epsilon_i)\epsilon_i - F(\theta - \sigma\epsilon_i)\epsilon_i)$$

Finite-difference-style gradient estimator:

$$\hat{\nabla}_N J(\theta) = \frac{1}{N\sigma}\sum_{i=1}^{N}(F(\theta + \sigma\epsilon_i)\epsilon_i - F(\theta)\epsilon_i)$$

You'd like samples to be independent; hard to prove things about it. If the samples are dependent, then the estimators have stronger concentration results (?). Structured (orthogonal) random estimators are good - reducing variance of estimator by doing a quasi-Monte Carlo method.

Theorem 9.1. *The orthogonal gradient estimator is unbiased and yields lower MSE than the unstructured estimator, namely*

$$MSE(\hat{\nabla}_N^{ort} J(\theta)) = MSE(\hat{\nabla}_N J(\theta) - \frac{N-1}{N}\|\nabla J(\theta)\|_2^2)$$

where MSE is of the Gaussian smoothed version

What I want to tell you now is even if this estimator based on orthogonal trick is more accurate, it still doesn't solve the problem of scalability. Why do we use the Monte Carlo techniques? They can be run in parallel.

The problem with compressed sensing methods applied to this is that it's not easy to distribute. It then becomes more like hyper-parameter tuning. If you have policies with thousands of parameters we don't know how to apply techniques in an undistributed way. At every point of the gradient algorithm, you need to do orthogonalization – this is expensive. This is cubic in dimension of parameters. So there are two tricks we applied here to make it feasible.

The idea is that there is a lot of redundance in neural nets. You often don't need to train full nets. Thi sis along these lines. One approach is compact policies. You can also use random Hadamard matrices rather than Gaussians. You want pairwise orthogonal, and have a nice isotropic property of Gaussians. You can replace Gaussian orthogonal matrix with a product of random Hadamard matrices. So you can write $\prod_{i=1}^{k} SD_i$, where D_i is diagonal. You'll get orthogonal rows and columns. The elements will be $\pm 1 \to \{\}$. These are Kronecker product based matrices. They are orthogonal and have lots of nice properties. You can make it random by multiplying by a diagonal. You can replace the rows of the

Gaussian with the rows of the product. What is nice about this? You don't need to do any orthogonalization, since it's embedded in the structure of the Hadamard matrices. In practice, you only need a a few copies. In our experiments, we just used one Hadamard matrix.

Toeplitz matrices have nice structure for computation, it's defined by first row and first column (log displacement matrices?). So you can do fast matrix-vector products ($n \log n$): they implement one-dimensional linear convolutions and arise naturally in time-series and dynamical systems. Hankel matrix is related: antidiagonals are constant. The goal of using structured matrices might be fast inference – but actually, it doesn't help much (operations are already optimized, the change in time complexity doesn't help that much). What we care more about is reducing the number of parameters. This allows us to get much faster algorithms to learn policies. Why do they have good quality? It's a bit mysterious for us, we do not have good theoretical results here. In most cases, the policies we learn are better than the unstructured ones. It seems like it's easier for the algorithm to train. We tried linear models, they weren't good enough though on the OpenAI gym tasks.

Ideas for projects: learn the estimator as you go along (different estimators (see the three above) were good for different projects); it would also be good to learn the right set of directions based on the history, take into account second order terms. gradient estimation is also very related to compressed sensing. how should you construct the matrices that you use to sense such that you can use compressed sensing techniques. How can you make it work in the distributed scenario (very large policies)? What would be efficient ways to do that? It feels like that should be more accurate than this Monte Carlo estimation. Monte carlo based techniques are great because easier to distribute, but maybe can come up with more techniques for better gradient estimation which still can be distributed nicely. Also another thing that's not clear is how gradient estimates interact with the Gaussian smoothing.