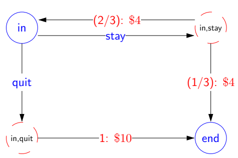
MDPs 2: Reinforcement Learning

Review: MDPs



Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

Reward(s, a, s'): reward for the transition (s, a, s')

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

- Last time, we talked about MDPs, which we can think of as graphs, where each node is either a state s or a chance node (s, a). Actions take us from states to chance nodes. This movement is something we can control. Transitions take us from chance nodes to states. This movement is random, and the various likelihoods are governed by transition probabilities.

Review: MDPs

- Following a **policy** π produces a path (**episode**)

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

- **Value** function $V_\pi(s)$: expected utility if follow π from state s

$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

- **Q-value** function $Q_\pi(s, a)$: expected utility if first take action a from state s and then follow π

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

- Given a policy π and an MDP, we can run the policy on the MDP yielding a sequence of states, action, rewards $s_0; a_1, r_1, s_1; a_2, r_2, s_2; \dots$. Formally, for each time step t, $a_t = \pi(s_{t-1})$, and s_t is sampled with probability $T(s_{t-1}, a_t, s_t)$. We call such a sequence an **episode** (a path in the MDP graph). This will be a central notion in this lecture.
- Each episode (path) is associated with a **utility**, which is the discounted sum of rewards: $u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots$. It's important to remember that the utility u_1 is a **random variable** which depends on how the transitions were sampled.
- The value of the policy (from state s_0) is $V_\pi(s_0) = \mathbb{E}[u_1]$, the expected utility. In the last lecture, we worked with the values directly without worrying about the underlying random variables (but that will soon no longer be the case). In particular, we defined recurrences relating the value $V_\pi(s)$ and Q-value $Q_\pi(s, a)$, which represents the expected utility from starting at the corresponding nodes in the MDP graph.
- Given these mathematical recurrences, we produced algorithms: policy evaluation computes the value of a policy, and value iteration computes the optimal policy.

Unknown transitions and rewards

Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

IsEnd(s): whether at end of game

$0 \le \gamma \le 1$: discount factor (default: 1)

reinforcement learning!

CS234 course: https://web.stanford.edu/class/cs234/

- In this lecture, we assume that we have an MDP where we neither know the transitions nor the reward functions. We are still trying to maximize expected utility, but we are in a much more difficult setting called **reinforcement learning**.

Mystery game

Example: mystery buttons

For each round $r = 1, 2, \dots$

- You choose A or B.
- You move to a new state and get some rewards.

Start | A | B

State: 5,0 Rewards: 0

- To put yourselves in the shoes of a reinforcement learner, try playing the game. You can either push the A button or the B button. Each of the two actions will take you to a new state and give you some reward.
- This simple game illustrates some of the challenges of reinforcement learning: we should take good actions to get rewards, but in order to know which actions are good, we need to explore and try different actions.

From MDPs to reinforcement learning

Markov decision process (offline)

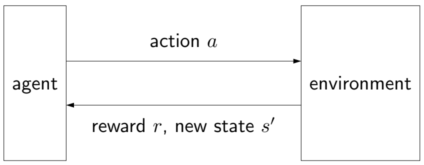
- Have mental model of how the world works.
- Find policy to collect maximum rewards.

Reinforcement learning (online)

- Don't know how the world works.
- Perform actions in the world to find out and collect rewards.

- An important distinction between solving MDPs (what we did before) and reinforcement learning (what we will do now) is that the former is **offline** and the latter is **online**.
- In the former case, you have a mental model of how the world works. You go lock yourself in a room, think really hard, and come up with a policy. Then you come out and use it to act in the real world.
- In the latter case, you don't know how the world works, but you only have one life, so you just have to go out into the real world and learn how it works from experiencing it and trying to take actions that yield high rewards.
- At some level, reinforcement learning is really the way humans work: we go through life, taking various actions, getting feedback. We get rewarded for doing well and learn along the way.

Reinforcement learning framework



Algorithm: reinforcement learning template

For $t = 1, 2, 3, \ldots$
  Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (**how?**)
  Receive reward r_t and observe new state s_t
  Update parameters (**how?**)

- To make the framework clearer, we can think of an **agent** (the reinforcement learning algorithm) that repeatedly chooses an action a_t to perform in the environment, and receives some reward r_t, and information about the new state s_t.
- There are two questions here: how to choose actions (what is π_{act}) and how to update the parameters. We will first talk about updating parameters (the learning part), and then come back to action selection later.

Volcano crossing



Run (or press ctrl-enter)

		-50	20
		-50	
2			

Utility: 2

a	r	s
		(2,1)
W	0	(2,1)
W	0	(2,1)
N	0	(1,1)
W	0	(1,1)
N	0	(1,1)
E	0	(1,2)
S	0	(2,2)
W	0	(2,1)
N	0	(2,2)
N	0	(3,2)
S	0	(3,2)
W	2	(3,1)

- Recall the volcano crossing example from the previous lecture. Each square is a state. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. The starting state is (2,1), and the end states are the four marked with red or green rewards. Transitions from (s, a) lead where you expect with probability `1-slipProb` and to a random adjacent square with probability `slipProb`.
- If we solve the MDP using value iteration (by setting `numIters` to 10), we will find the best policy (which is to head for the 20). Of course, we can't solve the MDP if we don't know the transitions or rewards.
- If you set `numIters` to zero, we start off with a random policy. Try pressing the Run button to generate fresh episodes. How can we learn from this data and improve our policy?

Outline

- Next, we will discuss with model-based methods for learning Markov Decision processes (MDPs).

Model-Based Value Iteration

Data: $s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$

Key idea: model-based learning

Estimate the MDP: $T(s, a, s')$ and $\text{Reward}(s, a, s')$

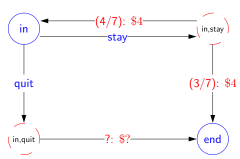
Transitions:

$$\hat{T}(s, a, s') = \frac{\text{\# times } (s, a, s') \text{ occurs}}{\text{\# times } (s, a) \text{ occurs}}$$

Rewards:

$$\widehat{\text{Reward}}(s, a, s') = r \text{ in } (s, a, r, s')$$

Model-Based Value Iteration



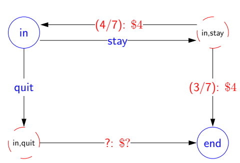
Data (following policy $\pi(s) = $ stay):

[in; stay, 4, end]

- Estimates converge to true values (under certain conditions)
- With estimated MDP $(\hat{T}, \widehat{\text{Reward}})$, compute policy using value iteration

- The first idea is called **model-based** value iteration, where we try to estimate the model (transitions and rewards) using Monte Carlo simulation.
- Monte Carlo is a standard way to estimate the expectation of a random variable by taking an average over samples of that random variable.
- Here, the data used to estimate the model is the sequence of states, actions, and rewards in the episode. Note that the samples being averaged are not independent (because they come from the same episode), but they do come from a Markov chain, so it can be shown that these estimates converge to the expectations by the ergodic theorem (a generalization of the law of large numbers for Markov chains).
- But there is one important caveat...

Problem



Problem: won't even see (s, a) if $a \neq \pi(s)$ ($a = $ quit)

Key idea: exploration

To do reinforcement learning, need to explore the state space.

Solution: need π to **explore** explicitly (more on this later)

- So far, our policies have been deterministic, mapping s always to $\pi(s)$. However, if we use such a policy to generate our data, there are certain (s, a) pairs that we will never see and therefore never be able to estimate their Q-value and never know what the effect of those actions are.
- This problem points at the most important characteristic of reinforcement learning, which is the need for **exploration**. This distinguishes reinforcement learning from supervised learning, because now we actually have to act to get data, rather than just having data poured over us.
- To close off this point, we remark that if π is a non-deterministic policy which allows us to explore each state and action infinitely often (possibly over multiple episodes), then the estimates of the transitions and rewards will converge.
- Once we get an estimate for the transitions and rewards, we can simply plug them into our MDP and solve it using standard value or policy iteration to produce a policy.
- Notation: we put hats on quantities that are estimated from data $(\hat{Q}^*, \hat{T})$ to distinguish from the true quantities (Q^*, T).

Outline

- Next, we will discuss model-free methods for learning Markov Decision processes (MDPs).

From model-based to model-free

$$\hat{Q}_{\text{opt}}(s,a) = \sum_{s'} \hat{T}(s,a,s')[\widehat{\text{Reward}}(s,a,s') + \gamma \hat{V}_{\text{opt}}(s')]$$

All that matters for prediction is (estimate of) $Q_{\text{opt}}(s,a)$.

Key idea: model-free learning

Try to estimate $Q_{\text{opt}}(s,a)$ directly.

- Taking a step back, if our goal is to just find good policies, all we need is to get a good estimate of $\hat{Q}_{\text{opt}}$. From that perspective, estimating the model (transitions and rewards) was just a means towards an end. Why not just cut to the chase and estimate $\hat{Q}_{\text{opt}}$ directly? This is called **model-free** learning, where we don't explicitly estimate the transitions and rewards.

Model-free Monte Carlo

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

Recall:

$Q_\pi(s,a)$ is expected utility starting at s, first taking action a, and then following policy π

Utility:

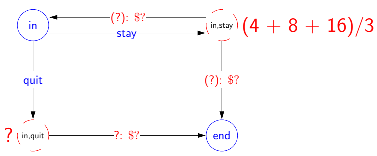
$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots$

Estimate:

$\hat{Q}_\pi(s,a) =$ average of u_t where $s_{t-1} = s, a_t = a$

(and s, a doesn't occur in $s_0, \cdots, s_{t-2}$)

Model-free Monte Carlo



Data (following policy $\pi(s) = \text{stay}$):

[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]

Note: we are estimating Q_π now, not Q_{opt}

Definition: on-policy versus off-policy

On-policy: estimate the value of data-generating policy

Off-policy: estimate the value of another policy

- Recall that $Q_\pi(s, a)$ is the expected utility starting at s, taking action a, and the following π.
- In terms of the data, define u_t to be the discounted sum of rewards starting with r_t.
- Observe that $Q_\pi(s_{t-1}, a_t) = \mathbb{E}[u_t]$; that is, if we're at state s_{t-1} and take action a_t, the average value of u_t is $Q_\pi(s_{t-1}, a_t)$.
- But that particular state and action pair (s, a) will probably show up many times. If we take the average of u_t over all the times that $s_{t-1} = s$ and $a_t = a$, then we obtain our Monte Carlo estimate $\hat{Q}_\pi(s, a)$. Note that nowhere do we need to talk about transitions or immediate rewards; the only thing that matters is total rewards resulting from (s, a) pairs.
- One technical note is that for simplicity, we only consider $s_{t-1} = s, a_t = a$ for which the (s, a) doesn't show up earlier. This is not necessary for the algorithm to work, but it is easier to analyze and think about.
- Model-free Monte Carlo depends strongly on the policy π that is followed; after all it's computing Q_π. Because the value being computed is dependent on the policy used to generate the data, we call this an **on-policy** algorithm. In contrast, model-based Monte Carlo is **off-policy**, because the model we estimated did not depend on the exact policy (as long as it was able to explore all (s, a) pairs).

Model-free Monte Carlo (equivalences)

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

Original formulation

$\hat{Q}_\pi(s, a) =$ average of u_t where $s_{t-1} = s, a_t = a$

Equivalent formulation (convex combination)

On each (s, a, u):

$$\eta = \frac{1}{1 + (\#\text{ updates to } (s, a))}$$

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

- Over the next few slides, we will interpret model-free Monte Carlo in several ways. This is the same algorithm, just viewed from different perspectives. This will give us some more intuition and allow us to develop other algorithms later.
- The first interpretation is thinking in terms of **interpolation**. Instead of thinking of averaging as a batch operation that takes a list of numbers (realizations of u_t) and computes the mean, we can view it as an iterative procedure for building the mean as new numbers are coming in.
- In particular, it's easy to work out for a small example that averaging is equivalent to just interpolating between the old value $\hat{Q}_\pi(s, a)$ (current estimate) and the new value u (data). The interpolation ratio η is set carefully so that u contributes exactly the right amount to the average.
- Advanced: in practice, we would constantly improve the policy π constantly over time. In this case, we might want to set η to something that doesn't decay as quickly (for example, $\eta = 1/\sqrt{\#\text{ updates to } (s, a)}$). This rate implies that a new example contributes more than an old example, which is desirable so that $\hat{Q}_\pi(s, a)$ reflects the more recent policy rather than the old policy.

Model-free Monte Carlo (equivalences)

Equivalent formulation (convex combination)

On each (s, a, u):

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

Equivalent formulation (stochastic gradient)

On each (s, a, u):

$$\hat{Q}_\pi(s, a) \leftarrow \hat{Q}_\pi(s, a) - \eta[\underbrace{\hat{Q}_\pi(s, a)}_{\text{prediction}} - \underbrace{u}_{\text{target}}]$$

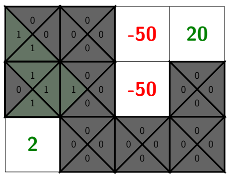
Implied objective: least squares regression

$$(\hat{Q}_\pi(s, a) - u)^2$$

- The second equivalent formulation is making the update look like a stochastic gradient update. Indeed, if we think about each (s, a, u) triple as an example (where (s, a) is the input and u is the output), then the model-free Monte Carlo is just performing stochastic gradient descent on a least squares regression problem, where the weight vector is $\hat{Q}_\pi$ (which has dimensionality SA) and there is one feature template "(s, a) equals ___".
- The stochastic gradient descent view will become particularly relevant when we use non-trivial features on (s, a).

Volcanic model-free Monte Carlo

Run (or press ctrl-enter)



a	r	s
		(2,1)
N	0	(1,1)
W	0	(1,1)
S	0	(2,1)
E	0	(2,2)
W	0	(2,1)
S	2	(3,1)

Utility: 2

- Let's run model-free Monte Carlo on the volcano crossing example. `slipProb` is zero to make things simpler. We are showing the Q-values: for each state, we have four values, one for each action.
- Here, our exploration policy is one that chooses an action uniformly at random.
- Try pressing "Run" multiple times to understand how the Q-values are set.
- Then try increasing `numEpisodes`, and seeing how the Q-values of this policy become more accurate.
- You will notice that a random policy has a very hard time reaching the 20.

Outline

- Next, we will discuss SARSA learning Markov Decision processes (MDPs).

Using the utility

Data (following policy $\pi(s) =$ stay):

[in; stay, 4, end]	$u = 4$
[in; stay, 4, in; stay, 4, end]	$u = 8$
[in; stay, 4, in; stay, 4, in; stay, 4, end]	$u = 12$
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	$u = 16$

Algorithm: model-free Monte Carlo

On each (s, a, u):

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta \underbrace{u}_{\text{data}}$$

Using the reward + Q-value

Current estimate: $\hat{Q}_\pi(s, \text{stay}) = 11$

Data (following policy $\pi(s) = \text{stay}$):

[in; stay, 4, end]	$4 + 0$
[in; stay, 4, in; stay, 4, end]	$4 + 11$
[in; stay, 4, in; stay, 4, in; stay, 4, end]	$4 + 11$
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	$4 + 11$

Algorithm: SARSA

On each (s, a, r, s', a'):

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta[\underbrace{r}_{\text{data}} + \gamma \underbrace{\hat{Q}_\pi(s',a')}_{\text{estimate}}]$$

- Broadly speaking, reinforcement learning algorithms interpolate between new data (which specifies the **target** value) and the old estimate of the value (the **prediction**).
- Model-free Monte Carlo's target was u, the discounted sum of rewards after taking an action. However, u itself is just an estimate of $Q_\pi(s,a)$. If the episode is long, u will be a pretty lousy estimate. This is because u only corresponds to one episode out of a mind-blowing exponential (in the episode length) number of possible episodes, so as the epsiode lengthens, it becomes an increasingly less representative sample of what could happen. Can we produce a better estimate of $Q_\pi(s,a)$?
- An alternative to model-free Monte Carlo is SARSA, whose target is $r + \gamma\hat{Q}_\pi(s', a')$. Importantly, SARSA's target is a combination of the data (the first step) and the estimate (for the rest of the steps). In contrast, model-free Monte Carlo's u is taken purely from the data.

Model-free Monte Carlo versus SARSA

Key idea: bootstrapping

SARSA uses estimate $\hat{Q}_\pi(s,a)$ instead of just raw data u.

u	$r + \hat{Q}_\pi(s', a')$
based on one path	based on estimate
unbiased	biased
large variance	small variance
wait until end to update	can update immediately

- The main advantage that SARSA offers over model-free Monte Carlo is that we don't have to wait until the end of the episode to update the Q-value.
- If the estimates are already pretty good, then SARSA will be more reliable since u is based on only one path whereas $\hat{Q}_\pi(s', a')$ is based on all the ones that the learner has seen before.
- Advanced: We can actually interpolate between model-free Monte Carlo (all rewards) and SARSA (one reward). For example, we could update towards $r_t + \gamma r_{t+1} + \gamma^2\hat{Q}_\pi(s_{t+1}, a_{t+2})$ (two rewards). We can even combine all of these updates, which results in an algorithm called SARSA(λ), where λ determines the relative weighting of these targets. See the Sutton/Barto reinforcement learning book (chapter 7) for an excellent introduction.
- Advanced: There is also a version of these algorithms that estimates the value function V_π instead of Q_π. Value functions aren't enough to choose actions unless you actually know the transitions and rewards. Nonetheless, these are useful in game playing where we actually know the transition and rewards, but the state space is just too large to compute the value function exactly.

think and share

Question

Which of the following algorithms allows you to estimate $Q^*(s,a)$ (select all that apply)?

(a) model-based value iteration

(b) model-free Monte Carlo

(c) SARSA

- Model-based value iteration estimates the transitions and rewards, which fully specifies the MDP. With the MDP, you can estimate anything you want, including computing $Q^*(s,a)$
- Model-free Monte Carlo and SARSA are on-policy algorithms, so they only give you $\hat{Q}_\pi(s,a)$, which is specific to a policy π. These will not provide direct estimates of $Q^*(s,a)$.

Outline

- Next, we will discuss Q-learning Markov Decision processes (MDPs).

Q-learning

Problem: model-free Monte Carlo and SARSA only estimate Q_π, but want Q_{opt} to act optimally

Output	**MDP**	**reinforcement learning**
Q_π	policy evaluation	model-free Monte Carlo, SARSA
Q_{opt}	value iteration	**Q-learning**

- Recall our goal is to get an optimal policy, which means estimating Q_{opt}.
- The situation is as follows: Our two methods (model-free Monte Carlo and SARSA) are model-free, but only produce estimates Q_π. We have one algorithm, model-based value iteration, which can be used to produce estimates of Q_{opt}, but is model-based. Can we get an estimate of Q_{opt} in a model-free manner?
- The answer is yes, and Q-learning is an algorithm that accomplishes this.
- One can draw an analogy between reinforcement learning algorithms and the classic MDP algorithms. MDP algorithms are offline, RL algorithms are online. In both cases, algorithms either output the Q-values for a fixed policy or the optimal Q-values.

Q-learning

Bellman optimality equation:

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')]$$

Algorithm: Q-learning [Watkins/Dayan, 1992]

On each (s,a,r,s'):

$$\hat{Q}_{\text{opt}}(s,a) \leftarrow (1-\eta)\underbrace{\hat{Q}_{\text{opt}}(s,a)}_{\text{prediction}} + \eta\underbrace{(r + \gamma\hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

Recall: $\hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s',a')$

- To derive Q-learning, it is instructive to look back at the Bellman optimality equation for Q_{opt}. There are several changes that take us from this recurrence to Q-learning. First, we don't have an expectation over s', but only have one sample s'.
- Second, because of this, we don't want to just replace $\hat{Q}_{\text{opt}}(s,a)$ with the target value, but want to interpolate between the old value (prediction) and the new value (target).
- Third, we replace the actual reward $\text{Reward}(s,a,s')$ with the observed reward r (when the reward function is deterministic, the two are the same).
- Finally, we replace $V_{\text{opt}}(s')$ with our current estimate $\hat{V}_{\text{opt}}(s')$.
- Importantly, the estimated optimal value $\hat{V}_{\text{opt}}(s')$ involves a maximum over actions rather than taking the action of the policy. This max over a' rather than taking the a' based on the current policy is the principle difference between Q-learning and SARSA.

SARSA versus Q-learning

Algorithm: SARSA

On each (s, a, r, s', a'):

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta(r + \gamma\hat{Q}_\pi(s',a'))$$

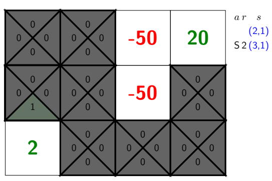
Algorithm: Q-learning [Watkins/Dayan, 1992]

On each (s, a, r, s'):

$$\hat{Q}_{\text{opt}}(s,a) \leftarrow (1-\eta)\hat{Q}_{\text{opt}}(s,a) + \eta(r + \gamma \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s',a'))]$$

Volcanic SARSA and Q-learning

Run (or press ctrl-enter)



Utility: 2

- Let us try SARSA and Q-learning on the volcanic example.
- If you increase `numEpisodes` to 1000, SARSA will behave very much like model-free Monte Carlo, computing the value of the random policy.
- However, note that Q-learning is computing an estimate of $Q_{\text{opt}}(s,a)$, so the resulting Q-values will be very different. The average utility will not change since we are still following and being evaluated on the same random policy. This is an important point for **off-policy** methods: the online performance (average utility) is generally a lot worse and not representative of what the model has learned, which is captured in the estimated Q-values.

Off-Policy versus On-Policy

Definition: on-policy versus off-policy

On-policy: evaluate or improve the data-generating policy

Off-policy: evaluate or learn using data from another policy

	on-policy	**off-policy**
policy evaluation	Monte Carlo SARSA	
policy optimization		Q-learning

- What do we mean by off-policy?
- Model-free Monte Carlo depends strongly on the policy π that is followed; after all it's computing Q_π. Because the value being computed is dependent on the policy used to generate the data, we call this an **on-policy** algorithm. In contrast, model-based value iteration is **off-policy**, because the model we estimated did not depend on the exact policy (as long as it was able to explore all (s,a) pairs).
- Further, model-free Q-learning is also **off-policy**, since it can learn the optimal policy using data from other policies.

Reinforcement Learning Algorithms

Algorithm	Estimating	Based on
Model-Based Monte Carlo	$\hat{T}, \hat{R}$	$s_0, a_1, r_1, s_1, \ldots$
Model-Free Monte Carlo	$\hat{Q}_\pi$	u
SARSA	$\hat{Q}_\pi$	$r + \hat{Q}_\pi$
Q-Learning	$\hat{Q}_{\text{opt}}$	$r + \hat{Q}_{\text{opt}}$

Outline

MDPs: model-based methods

MDPs: model-free methods

MDPs: SARSA

MDPs: Q-learning

MDPs: epsilon-greedy

MDPs: function approximation

MDPs: recap and extensions

- Next, we will discuss epsilon-greedy exploration for learning Markov Decision processes (MDPs).

Exploration

Algorithm: reinforcement learning template

For $t = 1, 2, 3, \ldots$
  Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (**how?**)
  Receive reward r_t and observe new state s_t
  Update parameters (**how?**)

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

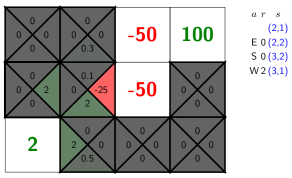
Which **exploration policy** π_{act} to use?

- We have so far given many algorithms for updating parameters (i.e., $\hat{Q}_\pi(s,a)$ or $\hat{Q}_{\text{opt}}(s,a)$). If we were doing supervised learning, we'd be done, but in reinforcement learning, we need to actually determine our **exploration policy** π_{act} to collect data for learning. Recall that we need to somehow make sure we get information about each (s,a).
- We will discuss two complementary ways to get this information: (i) explicitly explore (s,a) or (ii) explore (s,a) implicitly by actually exploring (s',a') with similar features and generalizing.
- These two ideas apply to many RL algorithms, but let us specialize to Q-learning.

No exploration, all exploitation

Attempt 1: Set $\pi_{\text{act}}(s) = \arg\max_{a \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s,a)$

Run (or press ctrl-enter)



Average (lifetime) utility: 1.95

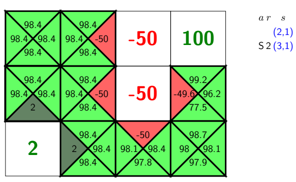
Problem: $\hat{Q}_{\text{opt}}(s,a)$ estimates are inaccurate, **too greedy**!

- The naive solution is to explore using the optimal policy according to the estimated Q-value $\hat{Q}_{\text{opt}}(s,a)$.
- But this fails horribly. In the example, once the agent discovers that there is a reward of 2 to be gotten by going south that becomes its optimal policy and it will not try any other action. The problem is that the agent is being too greedy.
- In the demo, if multiple actions have the same maximum Q-value, we choose randomly. Try clicking "Run" a few times, and you'll end up with minor variations.
- Even if you increase `numEpisodes` to 10000, nothing new gets learned.

No exploitation, all exploration

Attempt 2: Set $\pi_{\text{act}}(s) =$ random from Actions(s)

Run (or press ctrl-enter)



Average (lifetime) utility: -19.15

Problem: average utility is low because exploration is **not guided**

- We can go to the other extreme and use an exploration policy that always chooses a random action. It will do a much better job of exploration, but it doesn't exploit what it learns and ends up with a very low utility.
- It is interesting to note that the value (average over utilities across all the episodes) can be quite small and yet the Q-values can be quite accurate. Recall that this is possible because Q-learning is an off-policy algorithm.

Exploration/exploitation tradeoff

Key idea: balance

Need to balance **exploration** and **exploitation**.

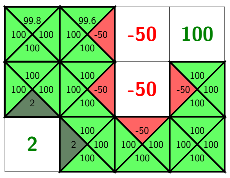
Examples from life: restaurants, routes, research

Epsilon-greedy

Algorithm: epsilon-greedy policy

$$\pi_{\text{act}}(s) = \begin{cases} \arg\max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s,a) & \text{probability } 1-\epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

Run (or press ctrl-enter)



a	r	s
		(2,1)
W	0	(2,1)
N	0	(1,1)
S	0	(2,1)
W	0	(2,1)
W	0	(2,1)
E	0	(2,2)
S	0	(3,2)
E	0	(3,3)
E	0	(3,4)
N	0	(2,4)
N	100	(1,4)

Average (lifetime) utility: 30.71

- The natural thing to do when you have two extremes is to interpolate between the two. The result is the **epsilon-greedy** algorithm which explores with probability ϵ and exploits with probability $1-\epsilon$.
- It is natural to let ϵ decrease over time. When you're young, you want to explore a lot ($\epsilon = 1$). After a certain point, when you feel like you've seen all there is to see, then you start exploiting ($\epsilon = 0$).
- For example, we let $\epsilon = 1$ for the first third of the episodes, $\epsilon = 0.5$ for the second third, and $\epsilon = 0$ for the final third. This is not the optimal schedule. Try playing around with other schedules to see if you can do better.

Outline

MDPs: model-based methods

MDPs: model-free methods

MDPs: SARSA

MDPs: Q-learning

MDPs: epsilon-greedy

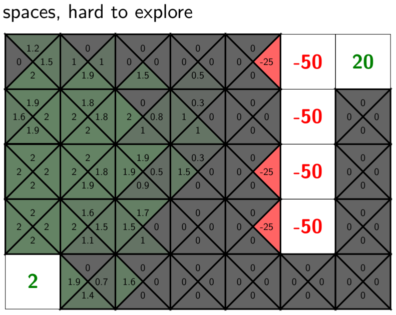
MDPs: function approximation

MDPs: recap and extensions

- Next, we will discuss function approximation methods for learning Markov Decision processes (MDPs).

Generalization

Problem: large state spaces, hard to explore



a	r	s
		(3,1)
S	0	(4,1)
S	2	(5,1)

Average (lifetime) utility: 0.44

- Now we turn to another problem with vanilla Q-learning.
- In real applications, there can be millions of states, in which there's no hope for epsilon-greedy to explore everything in a reasonable amount of time.

Q-learning

Stochastic gradient update:

$$\hat{Q}_{\text{opt}}(s,a) \leftarrow \hat{Q}_{\text{opt}}(s,a) - \eta[\underbrace{\hat{Q}_{\text{opt}}(s,a)}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}]$$

This is **rote learning**: every $\hat{Q}_{\text{opt}}(s,a)$ has a different value

Problem: doesn't generalize to unseen states/actions

- If we revisit the Q-learning algorithm, and think about it through the lens of machine learning, you'll find that we've just been memorizing Q-values for each (s,a), treating each pair independently.
- In other words, we haven't been generalizing, which is actually one of the most important aspects of learning!

Function approximation

Key idea: linear regression model

Define **features** $\phi(s,a)$ and **weights** $\mathbf{w}$:

$$\hat{Q}_{\text{opt}}(s,a;\mathbf{w}) = \mathbf{w} \cdot \phi(s,a)$$

Example: features for volcano crossing

$\phi_1(s,a) = \mathbf{1}[a = \mathsf{W}]$	$\phi_7(s,a) = \mathbf{1}[s = (5,*)]$
$\phi_2(s,a) = \mathbf{1}[a = \mathsf{E}]$	$\phi_8(s,a) = \mathbf{1}[s = (*,6)]$
...	...

- **Function approximation** fixes this by parameterizing $\hat{Q}_{\text{opt}}$ by a weight vector and a feature vector, as we did in linear regression.
- Recall that features are supposed to be properties of the state-action (s,a) pair that are indicative of the quality of taking action a in state s.
- The ramification is that all the states that have similar features will have similar Q-values. For example, suppose ϕ included the feature $\mathbf{1}[s = (*,4)]$. If we were in state $(1,4)$, took action E, and managed to get high rewards, then Q-learning with function approximation will propagate this positive signal to all positions in column 4 taking any action.
- In our example, we defined features on actions (to capture that moving east is generally good) and features on states (to capture the fact that the 6th column is best avoided, and the 5th row is generally a good place to travel to).

Function approximation

Algorithm: Q-learning with function approximation

On each (s,a,r,s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s,a;\mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}]\phi(s,a)$$

Implied objective function:

$$(\underbrace{\hat{Q}_{\text{opt}}(s,a;\mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}})^2$$

- We now turn our linear regression into an algorithm. Here, it is useful to adopt the stochastic gradient view of RL algorithms, which we developed a while back.
- We just have to write down the least squares objective and then compute the gradient with respect to $\mathbf{w}$ now instead of $\hat{Q}_{\text{opt}}$. The chain rule takes care of the rest.

Outline

- Finally, let's recap our discussion on methods for learning Markov Decision processes (MDPs).

Summary of MDPs

- **Markov decision processes** (MDPs) cope with uncertainty
- Solutions are **policies** rather than paths
- **Policy evaluation** computes policy value (expected utility)
- **Value iteration** computes optimal value (maximum expected utility) and optimal policy
- Main technique: write recurrences $\rightarrow$ algorithm

Summary of Reinforcement Learning

- Online setting: learn and take actions in the real world!
- Monte Carlo: estimate transitions, rewards, Q-values from data
- Bootstrapping: update towards target that depends on estimate rather than just raw data

- This concludes the technical part of reinforcement learning.
- The first part is to understand the setup: we are taking good actions in the world both to get rewards under our current model, but also to collect information about the world so we can learn a better model. This exposes the fundamental exploration/exploitation tradeoff, which is the hallmark of reinforcement learning.
- We looked at several algorithms: model-based value iteration, model-free Monte Carlo, SARSA, and Q-learning. There were two complementary ideas here: using Monte Carlo approximation (approximating an expectation with a sample) and bootstrapping (using the model predictions to update itself).

Covering the unknown

Epsilon-greedy: balance the exploration/exploitation tradeoff

Function approximation: can generalize to unseen states

Challenges in reinforcement learning

Binary classification (sentiment classification, SVMs):

- Stateless, full supervision

Reinforcement learning (flying helicopters, Q-learning):

- Stateful, partial supervision

Key idea: partial supervision

Reward feedback, but not given the solution directly.

Key idea: state

Rewards depend on previous actions $\Rightarrow$ can have delayed rewards.

States and information

	stateless	**state**
full supervision	supervised learning (binary classification)	supervised imitation learning (structured prediction)
partial supervision	multi-armed bandits	reinforcement learning

- If we compare simple supervised learning (e.g., binary classification) and reinforcement learning, we see that there are two main differences that make learning harder.
- First, reinforcement learning requires the modeling of state. State means that the rewards across time steps are related. This results in **delayed rewards**, where we take an action and don't see the ramifications of it until much later.
- Second, reinforcement learning requires dealing with partial supervision (rewards). This means that we have to actively explore to acquire the necessary supervision.
- There are two problems that move towards reinforcement learning, each on a different axis. Structured prediction introduces the notion of state, but the problem is made easier by the fact that we have full supervision, which means that for every situation, we know which action sequence is the correct one; there is no need for exploration; we just have to update our weights to favor that correct path.
- Multi-armed bandits require dealing with partial supervision, but do not have the complexities of state. One can think of a multi-armed bandit problem as an MDP with unknown random rewards and one state. Exploration is necessary, but there is no temporal depth to the problem.

Deep reinforcement learning

just use a neural network for $\hat{Q}_{\text{opt}}(s,a)$, π_{opt}, T, etc

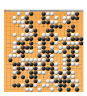
Playing Atari [Google DeepMind, 2013]:

- last 4 frames (images) $\Rightarrow$ 3-layer NN $\Rightarrow$ keystroke
- ϵ-greedy, train over 10M frames with 1M replay memory
- Human-level performance on some games (breakout), less good on others (space invaders)

- Recently, there has been a surge of interest in reinforcement learning due to the success of neural networks. If one is performing reinforcement learning in a simulator, one can actually generate tons of data, which is suitable for rich functions such as neural networks.
- A recent success story is DeepMind, who successfully trained a neural network to represent the $\hat{Q}_{\text{opt}}$ function for playing Atari games. The impressive part was the lack of prior knowledge involved: the neural network simply took as input the raw image and outputted keystrokes.

Deep reinforcement learning

- Policy gradient: train a policy $\pi(a \mid s)$ (say, a neural network) to directly maximize expected reward
- Google DeepMind's AlphaGo (2016), AlphaZero (2017)



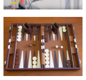
- Stanford CS224R course:

https://cs224r.stanford.edu/

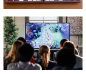
- One other major class of algorithms we will not cover in this class is **policy gradient**. Whereas Q-learning attempts to estimate the value of the optimal policy, policy gradient methods optimize the policy to maximize expected reward, which is what we care about. Recall that when we went from model-based methods (which estimated the transition and reward functions) to model-free methods (which estimated the Q function), we moved closer to the thing that we want. Policy gradient methods take this farther and just focus on the only object that really matters at the end of the day, which is the policy that an agent follows.
- Policy gradient methods have been quite successful. For example, it was one of the components of AlphaGo, Google DeepMind's program that beat the world champion at Go. One can also combine value-based methods with policy-based methods in actor-critic methods to get the best of both worlds.
- There is a lot more to say about deep reinforcement learning. If you wish to learn more, see the Stanford CS224R course website.

Applications

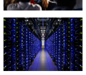
Robotics Applications: learning dexterous manipulation, control helicopter to do maneuvers in the air



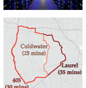
Backgammon: TD-Gammon plays 1-2 million games against itself, human-level performance



Games: DQN solving Atari Games, openAI Five playing Dota.



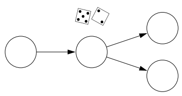
Managing datacenters; actions: bring up and shut down machine to minimize time/cost



Routing Autonomous Cars: bring down the total latency of vehicles on the road

- There are many other applications of RL, which range from robotics to game playing to other infrastructural tasks. One could say that RL is so general that anything can be cast as an RL problem.
- For a while, RL only worked for small toy problems or settings where there were a lot of prior knowledge / constraints. Deep RL — the use of powerful neural networks with increased compute — has vastly expanded the realm of problems which are solvable by RL.

Markov decision processes: against nature (e.g., Blackjack)



Next time...

Adversarial games: against opponent (e.g., chess)