• Last time, we talked about MDPs, which we can think of as graphs, where each node is either a state or a chance node. Actions take us from states to chance nodes (which we choose), and transitions take us from chance nodes to states (which nature chooses according to the transition probabilities).

In the previous lecture, you probably had some model of the world (how far Mountain View is, how long biking, driving, and Caltraining each take). But now, you should have no clue what’s going on. This is the setting of reinforcement learning. Now, you just have to try things and learn from your experience - that’s life!

• In the previous lecture, you probably had some model of the world (how far Mountain View is, how long biking, driving, and Caltraining each take). But now, you should have no clue what’s going on. This is the setting of reinforcement learning. Now, you just have to try things and learn from your experience - that’s life!

If you wanted to go from Orbisonia to Rockhill, how would you get there?
ride bus 1
ride bus 17
ride the magic tram

States: the set of states
S_{start} \subset S: starting state
A(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 ≤ γ ≤ 1: discount factor (default: 1)

Following a policy \( \pi \) produces a path (episode)

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

Value function \( V_\pi(s) \): expected utility if follow \( \pi \) 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 \( \pi \)

\[
Q_\pi(s, a) = \sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_\pi(s')]
\]
• Given a policy \( \pi \) and an MDP, we can run the policy on the MDP yielding a sequence of states, action, rewards \( s_0, a_1, r_1, s_1, a_2, r_2, s_2, \ldots \). 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: \( u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots \). 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 \( V_\pi(s_0) = E[u_1] \), 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 \( V_\gamma(s) \) and Q-value \( Q_\gamma(s,a) \), 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
- **States**: starting state \( s_{\text{start}} \) ∈ States
- **Actions\( (s) \)**: possible actions from state \( s \)
- **IsEnd\( (s) \)**: whether at end of game
- **0 ≤ γ ≤ 1**: discount factor (default: 1)

**reinforcement learning**

---

**Mystery game**

**Example: mystery buttons**

For each round \( r = 1, 2, \ldots \):

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

**Roadmap**

**Reinforcement learning**

- Monte Carlo methods
- Bootstrapping methods
- Covering the unknown

**Summary**

---

• 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**.

• 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 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.

Reinforcement learning framework

Algorithm: reinforcement learning template
For $t = 1, 2, 3, \ldots$
Choose action $a_t = \pi(s_{t-1})$ (how?)
Receive reward $r_t$ and observe new state $s_t$
Update parameters (how?)

Volcano crossing

(2,1) W 0 N 0 E 0 S 0
(2,1) W 0 N 0 E 0 S 0
(1,1) W 0 N 0 E 0 S 0
(1,1) W 0 N 0 E 0 S 0
(1,2) W 0 N 0 E 0 S 0
(2,2) W 0 N 0 E 0 S 0
(3,2) W 0 N 0 E 0 S 0
(3,1) W 0 N 0 E 0 S 0

Run (or press ctrl-enter)

-50 20
-50
2
Utility: 2

- 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, 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.

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 $\pi(s_t)$) and how to update the parameters. We will first talk about updating parameters (the learning part), and then come back to action selection later.

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 $1 - \text{slipProb}$ and to a random adjacent square with probability $\text{slipProb}$.

If we solve the MDP using value iteration (by setting $\text{numIters}$ 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 $\text{numIters}$ 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?
Model-based Monte Carlo

Data (following policy $\pi$):
$S_1; A, 3, S_2; B, 0, S_1; A, 5, S_1; A, 7, S_1$

Estimates:
$\hat{T}(S_1, A, S_1) = \frac{2}{3}$
$\hat{T}(S_1, A, S_2) = \frac{1}{3}$
$\hat{\text{Reward}}(S_1, A, S_1) = \frac{1}{2}(5 + 7) = 6$
$\hat{\text{Reward}}(S_1, A, S_2) = 3$

Estimates converge to true values (under certain conditions)

Problem

Data (following policy $\pi$):
$S_1; A, 3, S_2; B, 0, S_1; A, 5, S_1; A, 7, S_1$

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

Key idea: exploration
To do reinforcement learning, need to explore the state space.

Solution: need $\pi$ to explore explicitly (more on this later)

Model-based Monte Carlo

Example: model-based Monte Carlo

Data (following policy $\pi$):
$S_1; A, 3, S_1; A, 2, r_2, S_2; A_3, r_3, S_3; \ldots; A_n, r_n, S_n$

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:
$\hat{\text{Reward}}(s, a, s') = \text{average of } r \text{ in } (s, a, r, s')$

- The first idea is called model-based Monte Carlo, where we try to estimate the model (transitions and rewards) using Monte Carlo simulation. Note: if the rewards are deterministic, then every time we see a $(s, a, s')$ triple, we get the same number so taking the average just returns that number; allowing averages allows us to handle non-deterministic rewards, in which case $\hat{\text{Reward}}(s, a, s')$ would converge to its expectation.
- 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 if they 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...

- 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 $\pi$ 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 ($\hat{Q}_{opt}, T$) to distinguish from the true quantities ($Q_{opt}, T$).
From model-based to model-free

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

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.

Model-free Monte Carlo

Data (following policy \( \pi \)):

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

Recall:

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

Utility:

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

Estimate:

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

Model-free Monte Carlo (equivalences)

Data (following policy \( \pi \)):

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

Original formulation

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

Equivalent formulation (convex combination)

On each \((s, a, u)\):

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

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

[whiteboard: \( u_1, u_2, u_3 \)]

Caveat: converges, but still need to follow \( \pi \) that explores

Note: we are estimating \( Q_\pi \) now, not \( Q_{\text{opt}} \)
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 $Q_\pi(s, a)$ (current estimate) and the new value $u$ (data). The interpolation ratio $\eta$ is set carefully so that $u$ contributes exactly the right amount to the average.

But we could use a different choice of $u$. In practice, it is useful to set $\eta$ to something that doesn’t decay as quickly (for example, $\eta = 1/(n + \tau)$ so that newer examples are favored. The motivation is that as learning proceeds, later samples $u_t$ will be more reliable than earlier ones.

The second equivalent formulation is making the update look like a stochastic gradient update. Indeed, if think about each $Q_\pi(s, a)$ as the input and $u$ as 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 $Q_\pi$ (which has dimensionality $S \times A$) and there is one feature template $(s, a)$.

The stochastic gradient descent view will become particularly relevant when we use non-trivial features on $(s, a)$.

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 nstep_episodes, 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.
### SARSA

Data (following policy \( \pi \)):

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

**Algorithm: model-free Monte Carlo**

When receive \((s, a, u)\):

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

**Algorithm: SARSA**

When receive \((s, a, r, s', a')\):

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

### Comparison

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

**Key idea: bootstrapping**

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

- \( u \) is only based on one path, so could have large variance, need to wait until end
- \( \hat{Q}_\pi(s', a') \) based on estimate, which is more stable, update immediately

### Question

Which of the following algorithms allows you to estimate \( Q^{\text{opt}}(s, a) \) (select all that apply)?

- model-based Monte Carlo
- model-free Monte Carlo
- SARSA
- Advanced: There is also a version of these algorithms that estimates the value function \( V_\pi \) instead of \( Q_\pi \). Value functions aren’t enough to choose actions unless you actually know the transitions and rewards. Nonetheless, these are useful in game playing where you actually know the transition and rewards, but the state space is just too large to compute the value function exactly.

- 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 episode lengthens, it becomes an increasingly less representative sample of what could happen. Can we produce 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.

- 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 \( Q_\pi(s', a') \) 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 \( r + \gamma (\hat{Q}_\pi(s', a') + \lambda Q_\pi(s', a') \) [two rewards]. We can even combine all of these updates, which results in an algorithm called SARSA(\( \lambda \)), where \( \lambda \) 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 \( V_\pi \) instead of \( Q_\pi \). Value functions aren’t enough to choose actions unless you actually know the transitions and rewards. Nonetheless, these are useful in game playing where you actually know the transition and rewards, but the state space is just too large to compute the value function exactly.

- Model-based Monte Carlo estimates the transitions and rewards, which fully specifies the MDP. With the MDP, you can estimate anything you want, including computing \( Q^{\text{opt}}(s, a) \).
- 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 \( \pi \). These will not provide direct estimates of \( Q^{\text{opt}}(s, a) \).
Recall our goal is to get an optimal policy, which means estimating $Q_{\text{opt}}$. The situation is as follows: Our two methods (model-free Monte Carlo and SARSA) are model-free, but only produce estimates $Q_{\pi}$. We have one algorithm, model-based Monte Carlo, which can be used to produce estimates of $Q_{\text{opt}}$, but is model-based. Can we get an estimate of $Q_{\text{opt}}$ in a model-free manner? The answer is yes, and Q-learning is an off-policy 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.

To derive Q-learning, it is instructive to look back at the MDP recurrence for $Q_{\text{opt}}$. There are several changes that take us from the MDP 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 $Q_{\text{opt}}(s,a)$ 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.

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.
• The naive solution is to explore using the optimal policy according to the estimated Q-value $\hat{Q}_{opt}(s, a)$.
• 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 number of episodes to 10000, nothing new gets learned.

The naive solution is to explore using the optimal policy according to the estimated Q-value $\hat{Q}_{opt}(s, a)$.
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 number of episodes to 10000, nothing new gets learned.

No exploitation, all exploration

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

Run (or press ctrl-enter)

(3,1) | (2,1) | (1,1) | (2,2) | (1,2) | (3,2)
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50

Average utility: 1.95

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

No exploitation, all exploration

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

Run (or press ctrl-enter)

(3,1) | (2,1) | (1,1) | (2,2) | (1,2) | (3,2)
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50
-50 -50 -50 -50 -50 -50

Average 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

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

Generalization

Problem: large state spaces, hard to explore

Average utility: 0.44

Now we turn to another problem with vanilla Q-learning.
- 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 \( \epsilon \) and exploits with probability \( 1 - \epsilon \).
- Over time, it is natural to let \( \epsilon \) 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.

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:

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

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

Problem: doesn’t generalize to unseen states/actions

Function approximation

Key idea: linear regression model
Define features \( \phi(s, a) \) and weights \( w \):

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

Example: features for volcano crossing

\[
\begin{align*}
\phi_1(s, a) &= 1[a = \text{W}] \\
\phi_7(s, a) &= 1[s = (5, \ast)] \\
\phi_2(s, a) &= 1[a = \text{E}] \\
\phi_8(s, a) &= 1[s = (\ast, 6)] \\
\end{align*}
\]

Algorithm: Q-learning with function approximation

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

\[
w \leftarrow w - \eta \left[ \hat{Q}_{\text{opt}}(s, a; w) - (r + \gamma \hat{V}_{\text{opt}}(s')) \right] \phi(s, a)
\]

Implied objective function:

\[
\left( \hat{Q}_{\text{opt}}(s, a; w) - (r + \gamma \hat{V}_{\text{opt}}(s')) \right)^2
\]

• 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 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 \( \phi \) included the feature \( 1[s = (5, \ast)] \). 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).

• 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 \(w\) now instead of \( \hat{Q}_{\text{opt}} \). The chain rule takes care of the rest.
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 feedback

Reinforcement learning (flying helicopters, Q-learning):
- Stateful, partial feedback

Key idea: partial feedback
Only learn about actions you take.

Key idea: state
Rewards depend on previous actions ⇒ can have delayed rewards.

Summary so far

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

Roadmap

Reinforcement learning
Monte Carlo methods
Bootstrapping methods
Covering the unknown
Summary

States and information

stateless
state
full feedback supervised learning (binary classification)
partial feedback multi-armed bandits reinforcement learning
Deep reinforcement learning

just use a neural network for $Q_{opt}(s, a)$

Playing Atari [Google DeepMind, 2013]:

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

Deep reinforcement learning

- Policy gradient: train a policy $\pi(a | s)$ (say, a neural network) to directly maximize expected reward
- Google DeepMind’s AlphaGo (2016)
- Andrej Karpathy’s blog post

http://karpathy.github.io/2016/05/31/rl

Applications

Autonomous helicopters: control helicopter to do maneuvers in the air

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

Elevator scheduling; send which elevators to which floors to maximize throughput of building

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