Bandit Problems and Model-free Reinforcement Learning

CS 4641 B: Machine Learning (Summer 2020)

Miguel Morales

07/08/2020

Outline

Bandit Problems

Prediction Problem

Control Problem

Outline

Bandit Problems

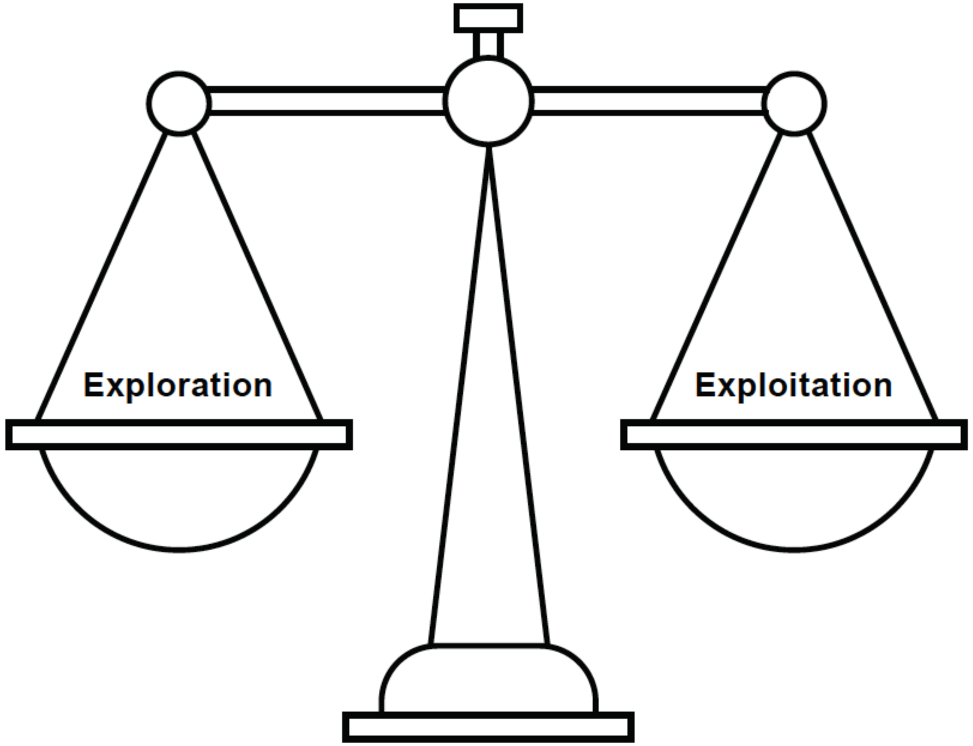
Prediction Problem

Control Problem

> Uncertainty and expectation are the joys of life. Security is an insipid thing.
>
> — William Congreve
> English playwright and poet of the Restoration period and political figure in the British Whig Party

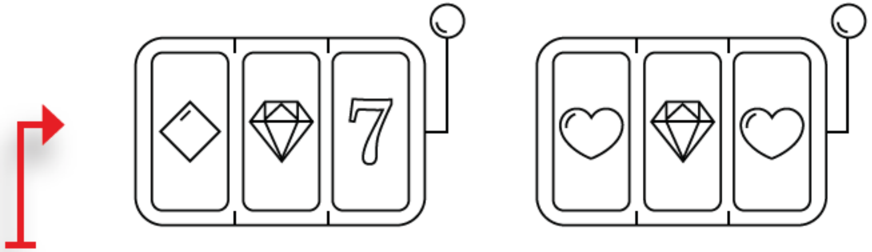
Exploration vs. Exploitation

- Planning methods assume we have a “map” of the environment. But what if we don’t?
- We need to explore to gain information about the environment.
- But exploring cause us to missed opportunities we could otherwise exploit.
- There is a tradeoff between exploration and exploration.



Exploration vs. Exploitation

- Multi-armed bandits (MAB) are a special case of a RL problem in which the size of the state space and horizon equal one.
- MAB have multiple actions, a single state, and a greedy horizon; you can also think of it as a "many-options single-choice" environment.
- The name comes from slot machines (bandits) with multiple arms to choose from (more realistically: multiple slot machines to choose from).



(1) A 2-armed bandit is a decision-making problem with two choices. You need to try them both sufficient to correctly asses each option. So, how do you best hand the exploration-exploitation tradeoff?

Bandit Problems

Show Me the Math

Multi-armed bandit

(1) MABs are MDPs with a single non-terminal state, and a single time step per episode.

$$MAB = \mathcal{MDP}(\mathcal{S} = \{s\}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{S}_\theta = \{s\}, \gamma = 1, \mathcal{H} = 1)$$

(2) The Q-function of action *a* is the expected reward given *a* was sampled.

$$q(a) = \mathbb{E}[R_t | A_t = a]$$

(3) The best we can do in a MAB is represented by the optimal V-function, or selecting the action that maximizes the Q-function.

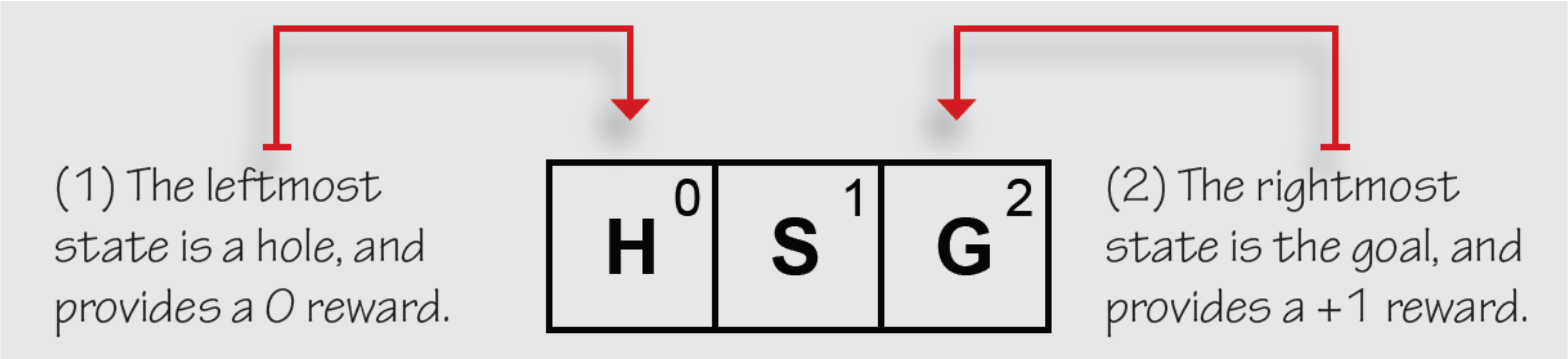
$$v_* = q(a_*) = \max_{a \in A} q(a)$$

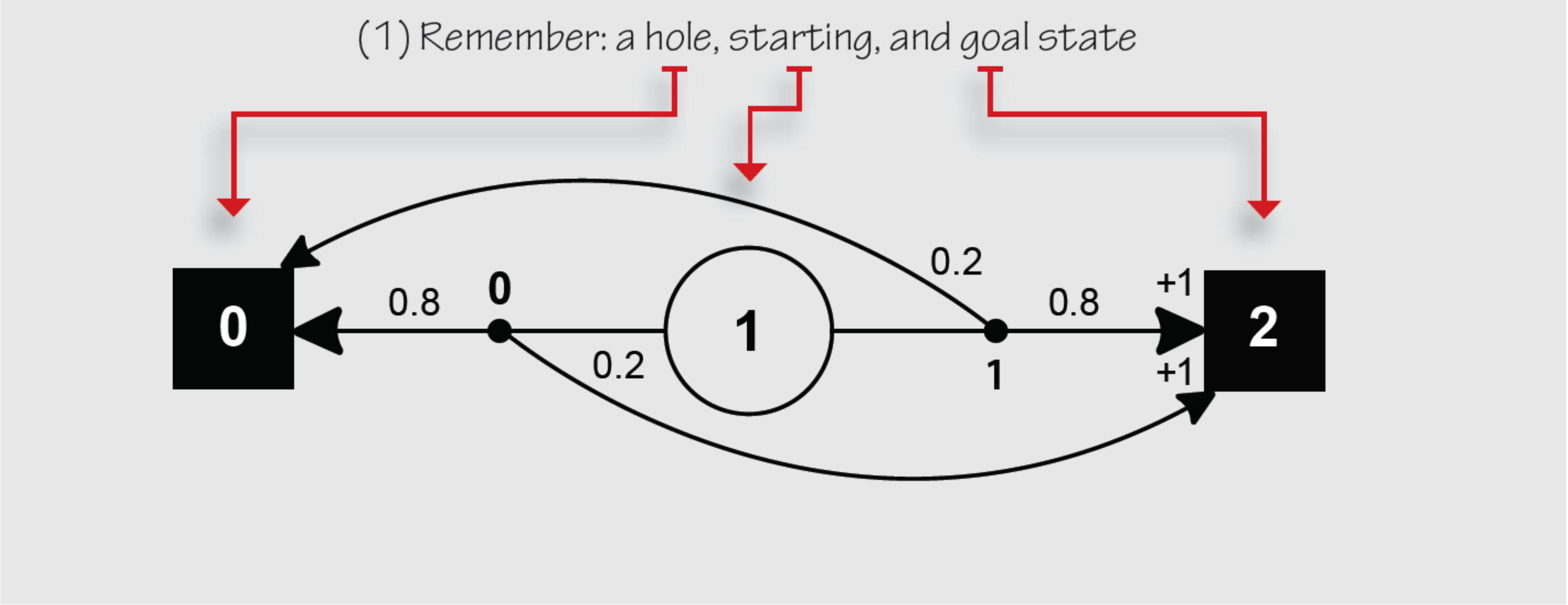
(4) The optimal action, is the action that maximizes the optimal Q-function, and optimal V-function (only 1 state).

$$a_* = \operatorname{argmax}_{a \in A} q(a)$$

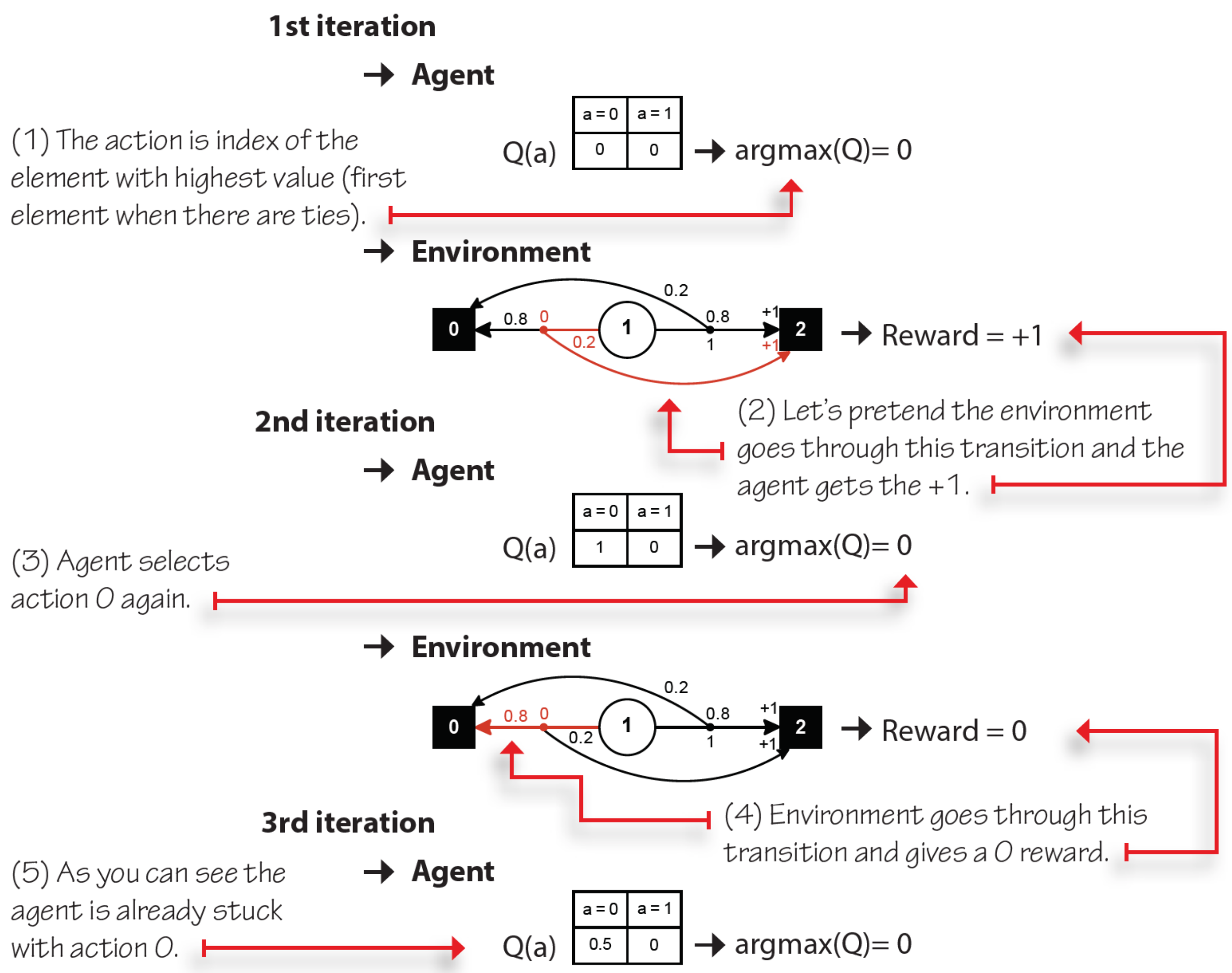
$$q(a_*) = v_*$$

Slippery Bandit Walk environment

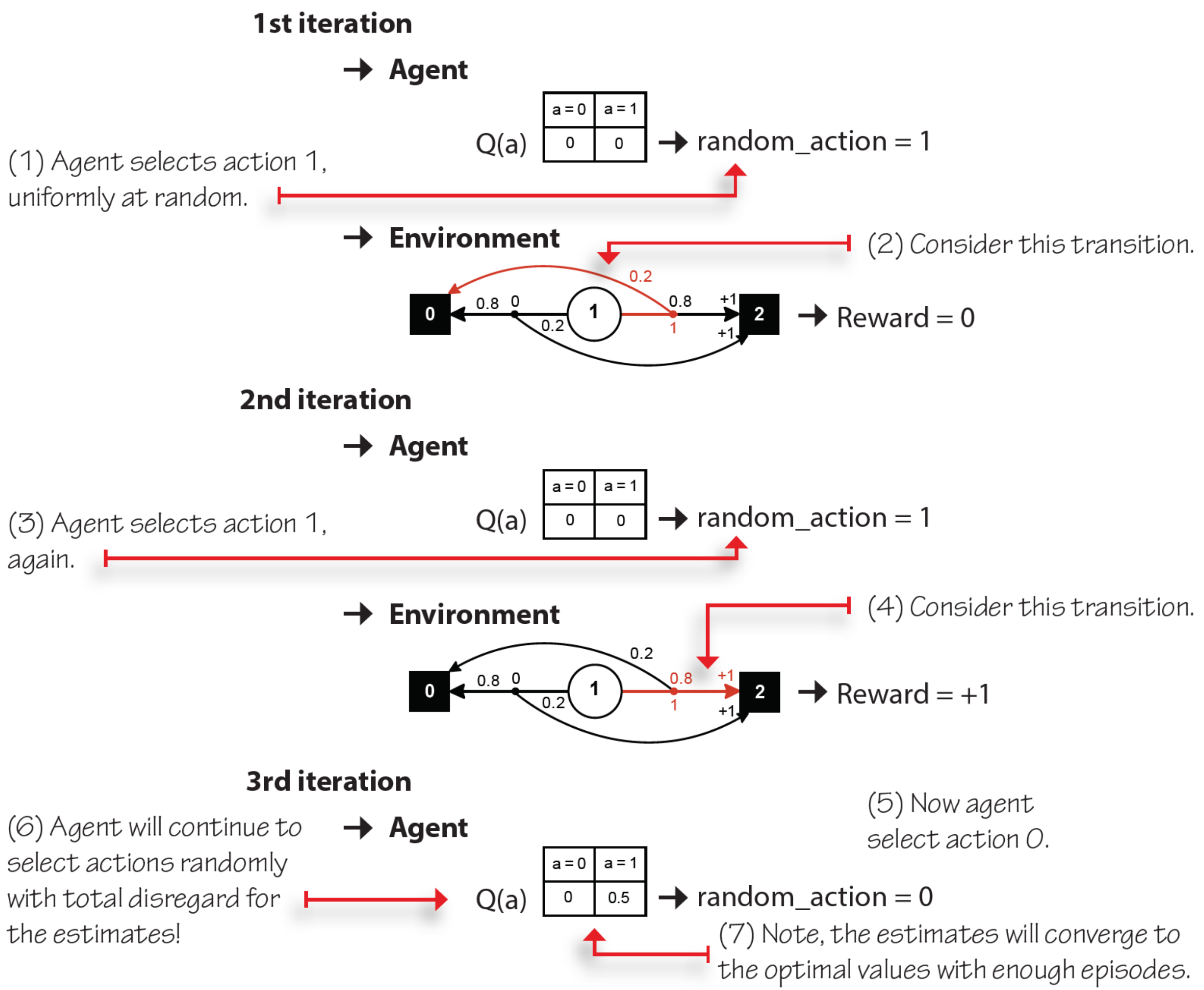




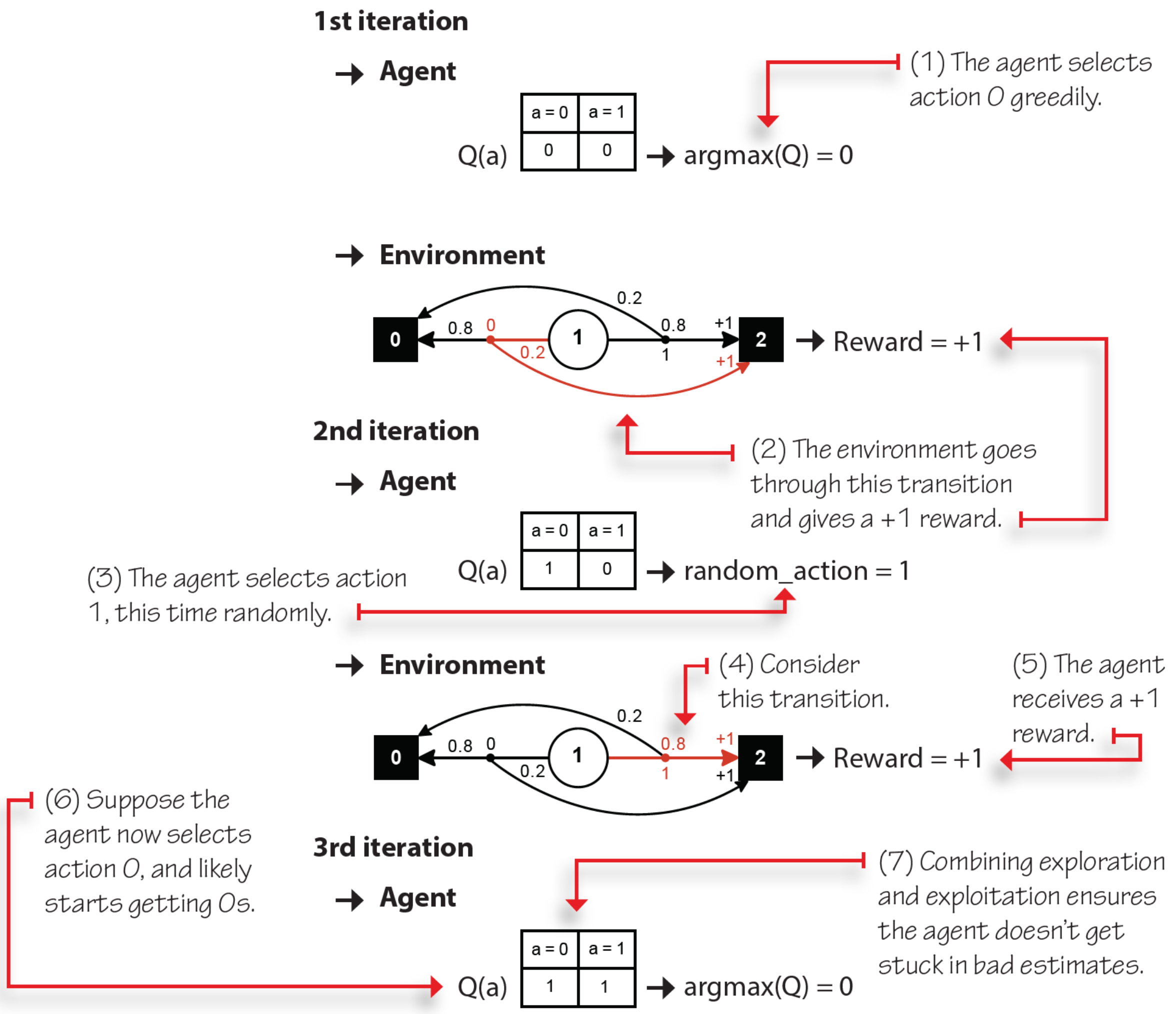
Greedy strategy: Always exploit



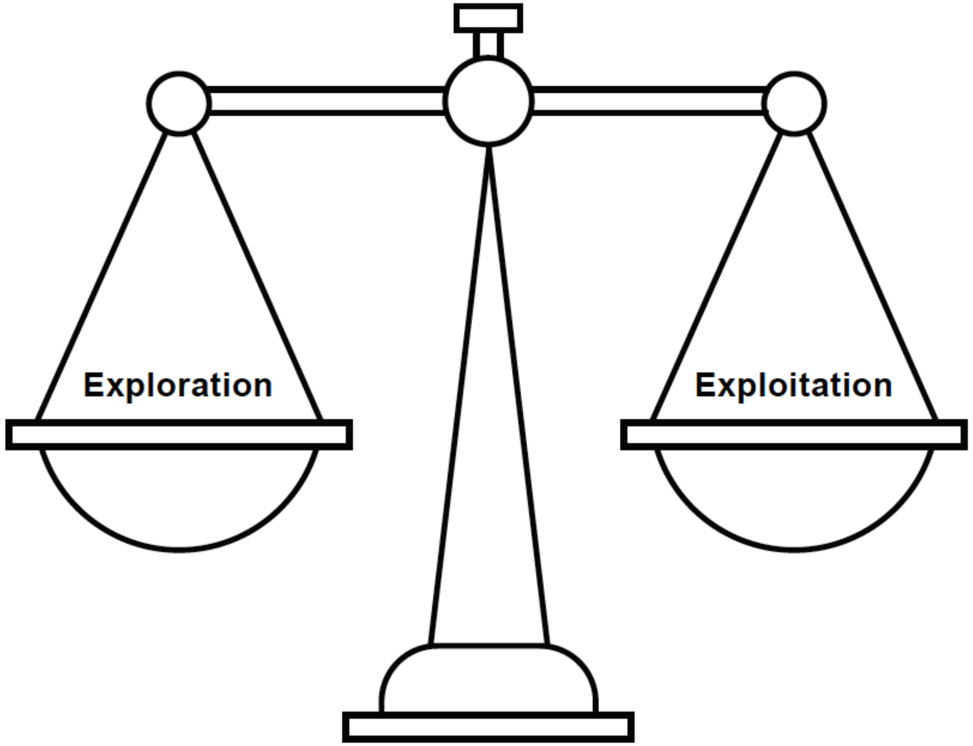
Random strategy: Always explore



Epsilon-Greedy strategy: Always always greedy, sometimes random



Recap: Bandit Problems



Recommended reading.
Reinforcement Learning: An introduction (chapter 2)
http://incompleteideas.net/book/the-book-2nd.html

Outline

Bandit Problems

Prediction Problem

Control Problem

“I conceive that the great part of the miseries of mankind are brought upon them by false estimates they have made of the value of things.”

— Benjamin Franklin
Founding Father of the United States
an author, politician, inventor, and a civic activist.

Georgia Tech
CREATING THE NEXT

Prediction Problem

- Estimate the value of policies; evaluate policies under feedback that is simultaneously sequential and evaluative.
- This is not policy optimization but is equally important for finding optimal policies.
- Having perfect estimates makes policy improvement trivial.

Terminology recap

With An RL Accent

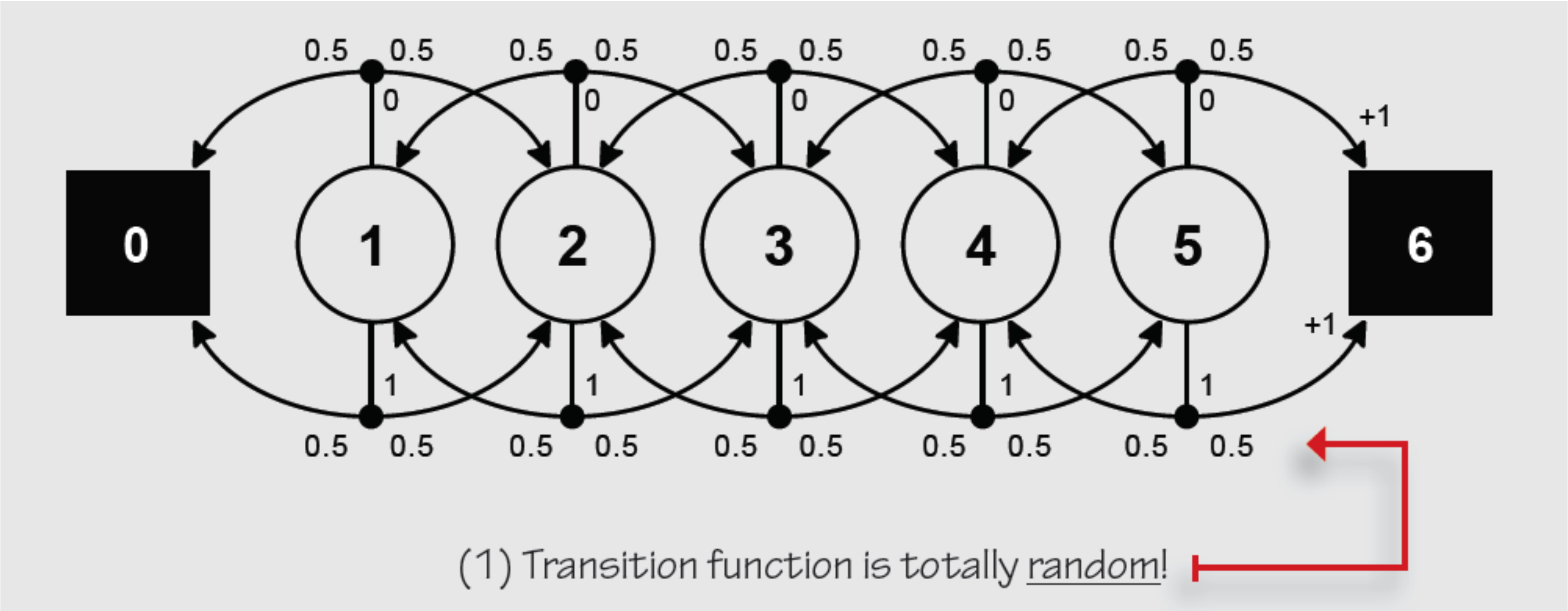
Reward vs. Return vs. Value function

Reward: Refers to the *one-step reward signal* the agent gets: the agent observes a state, selects an action, and it receives a reward signal. The reward signal is the core of RL, but it is *not* what the agent is trying to maximize! Again, the agent is not trying to maximize the reward! Realize that while your agent maximizes the one-step reward, in the long-term, is getting less than it could.

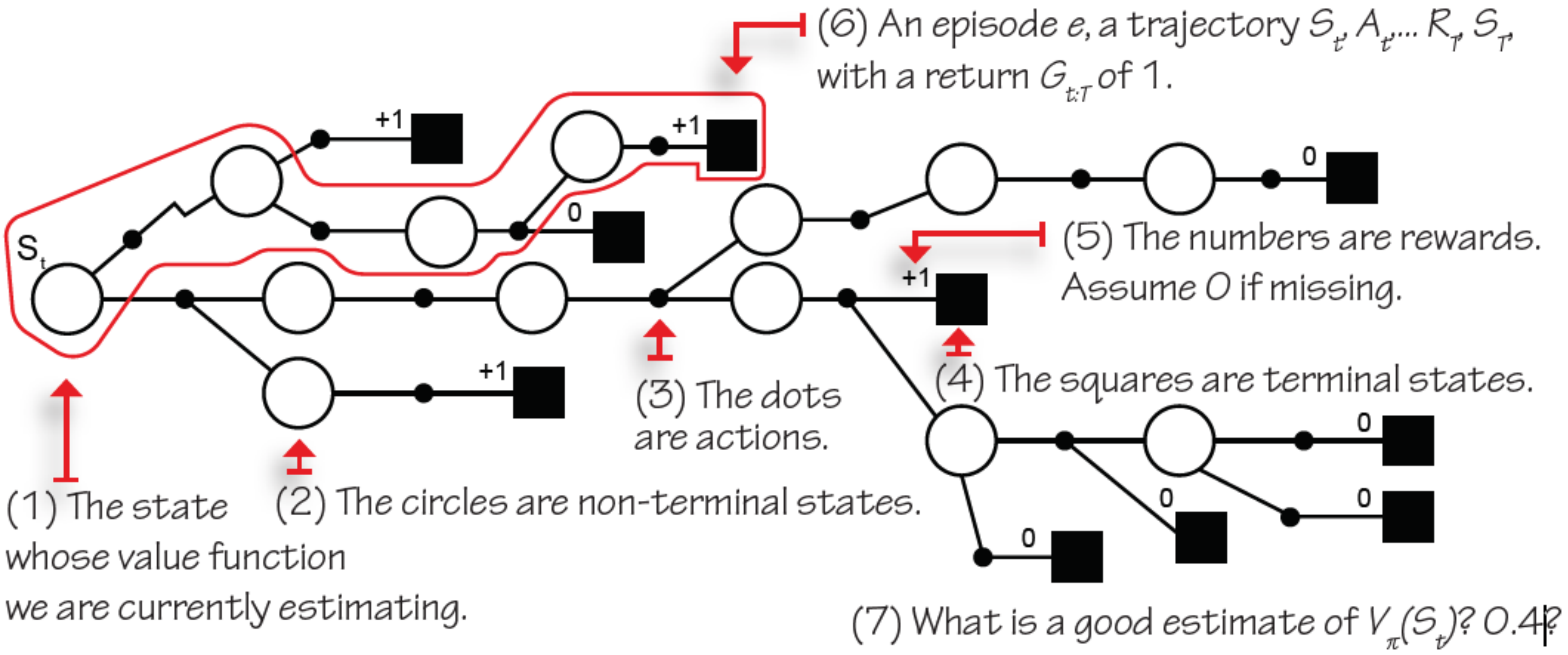
Return: Refers to the *total discounted rewards*. Returns are calculated from any state and usually go until the end of the episode. That is when a terminal state is reached the calculation stops. Returns are often referred to as *total reward*, *cumulative reward*, sum of rewards, and are commonly *discounted*: *total discounted reward*, *cumulative discounted reward, sum of discounted reward*. But, it is basically the same: a return tells you how much reward your agent *obtained* in an episode. As you can see, returns are better indicators of performance because they contain a long-term sequence, a single-episode history of rewards. But the return is *not* what an agent tries to maximize, either! An agent that attempts to obtain the highest possible return may find a policy that takes it through a noisy path; sometimes, this path will provide a high return, perhaps most of the time a low one.

Value function: Refers to the *expectation of returns*. That means, sure, we want high returns, but high in *expectation (on average)*. So, if the agent is in a very noisy environment, or if the agent is using a stochastic policy, it's all just fine. The agent is trying to maximize the *expected total discounted reward*, after all: value functions.

The Random Walk environment



Monte-Carlo prediction



Monte-Carlo prediction equations

(1) **WARNING:** I'm heavily abusing notation to make sure you get the whole picture. In specific, you need to notice when each thing is calculated. For instance, when you see a subscript *t:T*, that just means it is derived from time step *t* until the final time step, *T*. When you see *T*, that means it is computed at the end of the episode at the final time step *T*.

(2) As a reminder, the action-value function is the expectation of returns. This is a *definition* good to remember.

$$v_\pi(s) = \mathbb{E}_\pi[G_{t:T} \mid S_t = s]$$

(3) And the returns are the total discounted reward.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$$

(4) So, in MC, the first thing we do is sample the policy for a trajectory.

$$S_t, A_t, R_{t+1}, S_{t+1}, \ldots, R_T, S_T \sim \pi_{t:T}$$

(5) Given that trajectory, we can calculate the return for all states encountered.

(6) Then, add up the per-state returns.

$$T_T(S_t) = T_T(S_t) + G_{t:T}$$

(7) And, increment a count (more on this later.)

$$N_T(S_t) = N_T(S_t) + 1$$

(8) We can simply estimate the expectation using the empirical mean. So, the estimated state-value function for a state is just the mean return for that state.

$$V_T(S_t) = \frac{T_T(S_t)}{N_T(S_t)}$$

(9) As the counts approach infinity, the estimate will approach the true value

$$N(s) \rightarrow \infty \quad V(s) \rightarrow v_\pi(s)$$

(10) But, notice that means can be calculated incrementally. So, there is no need to keep track of the sum of returns for all states. This equation is equivalent, just more efficient.

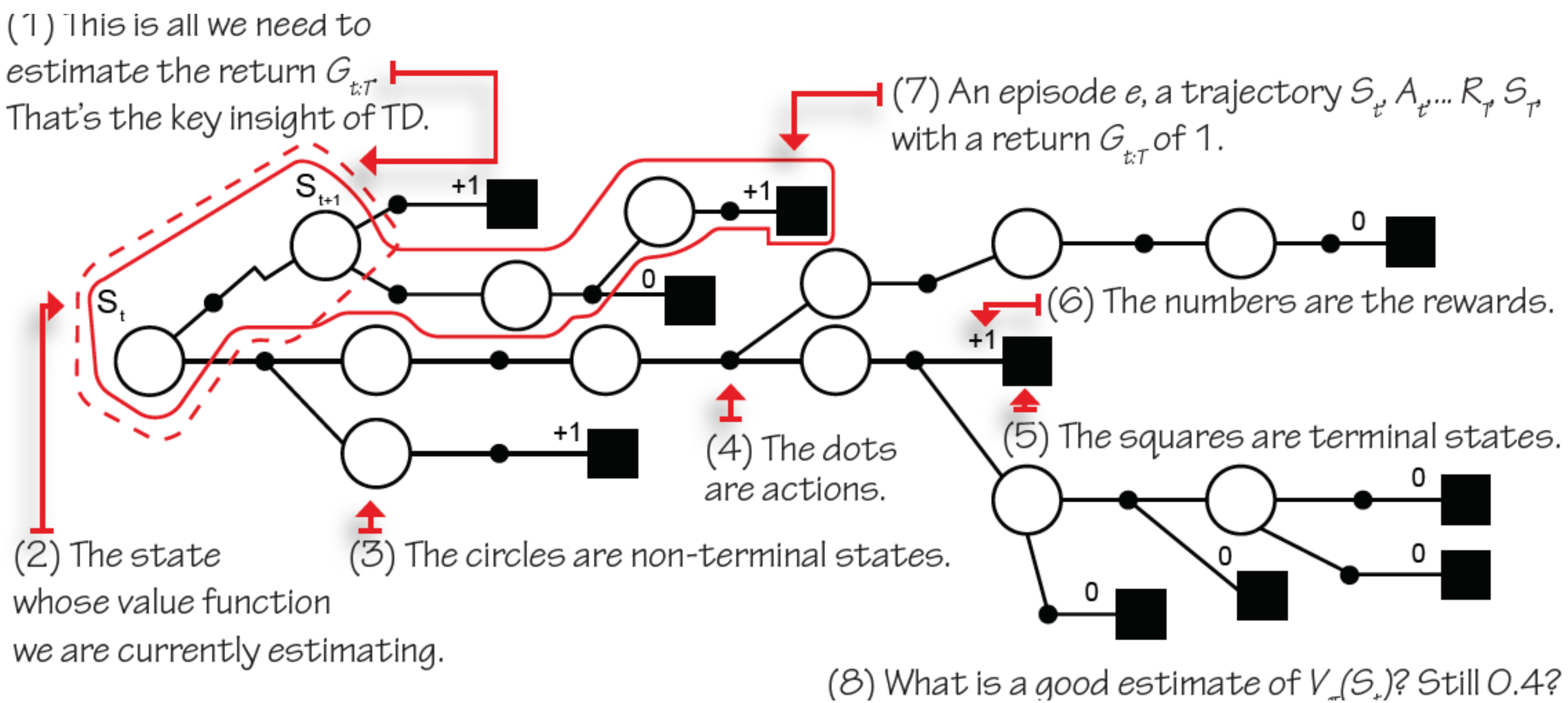
$$V_T(S_t) = V_{T-1}(S_t) + \frac{1}{N_t(S_t)}\left[G_{t:T} - V_{T-1}(S_t)\right]$$

(11) On this one, we just replace the mean for a learning value that can be time dependent, or constant.

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \left[\overbrace{\underbrace{G_{t:T}}_{\text{MC target}} - V_{T-1}(S_t)}^{\text{MC error}}\right]$$

(12) Notice that V is calculated only at the end of an episode, time step T, because G depends on it.

Temporal-Difference Learning



Temporal-Difference Learning equations

(1) We again start from the definition of the state-value function.

$$v_\pi(s) = \mathbb{E}_\pi[G_{t:T} \mid S_t = s]$$

(2) And the definition of the return.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$$

(3) From the return, we can rewrite the equation by grouping up some terms. Check it out.

$$\begin{aligned} G_{t:T} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots + \gamma^{T-2} R_T) \\ &= R_{t+1} + \gamma G_{t+1:T} \end{aligned}$$

(4) Now, the same return has a recursive style.

(5) We can use this new definition to also rewrite the state-value function definition equation.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_{t:T} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1:T} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned}$$

(6) And because the expectation of the returns from the next state is simply the state-value function of the next state, we get.

(7) This means we could estimate the state-value function on every time step.

(8) We roll out a single interaction step.

$$S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{t:t+1}$$

(9) And can obtain an estimate *V(s)* of the true state-value function $v_\pi(s)$ a different way than with MC.

(10) The key difference to realize is we are now estimating $v_\pi(s_t)$ with an estimate of $v_\pi(s_{t+1})$. We are using an estimated, not an actual return.

$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \Big[\overbrace{\underbrace{R_{t+1} + \gamma V_t(S_{t+1})}_{\text{TD target}} - V_t(S_t)}^{\text{TD error}} \Big]$$

(11) A big win is we can now make updates to the state-value function estimates *V(s)* every time step.

MC vs. TD learning

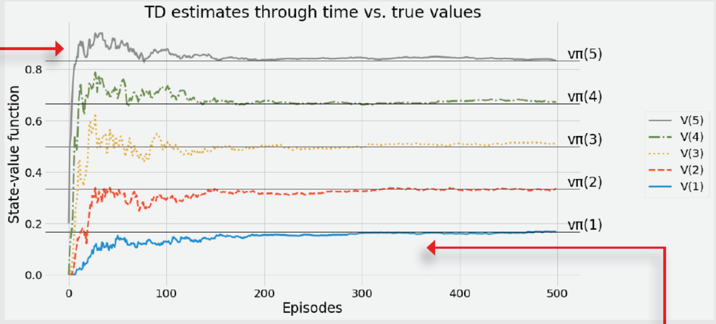
Tally It Up

MC and TD both nearly converge to the true state-value function

(1) Here I'll be showing only First-Visit Monte-Carlo prediction (FVMC) and Temporal-Difference Learning (TD). If you head to the Notebook for this chapter, you'll also see the results for Every-Visit Monte-Carlo prediction, and some additional plots that may be of interest to you!

(2) Take a close look at these plots. These are the running state-value function estimates $V(s)$ of an all-left policy in the Random Walk environment. As you can see in these plots, both algorithms show near-convergence to the true values.

(3) Now, see the difference trends of these algorithms. FVMC running estimates are very noisy, they jump back and forth around the true values.



(4) TD running estimates don't jump as much, but they are off center for most of the episodes. For instance $V(5)$ is usually higher than $v_\pi(5)$, while $V(1)$ is usually lower than $v_\pi(1)$. But if you compare those values with FVMC estimates, you notice a different trend.

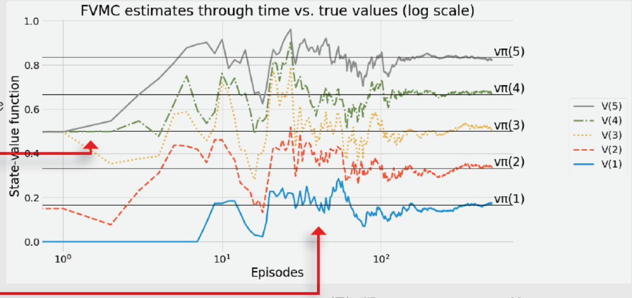
Tally It Up

MC estimates are noisy, TD estimates off target

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \Big[\overbrace{\underbrace{G_{t:T}}_{\text{MC target}} - V_{T-1}(S_t)}^{\text{MC error}} \Big]$$

(1) If we get a close-up (log-scale plot) these trends, you will see what's happening. MC estimates jump around the true values. This is because of the high variance of the MC targets.

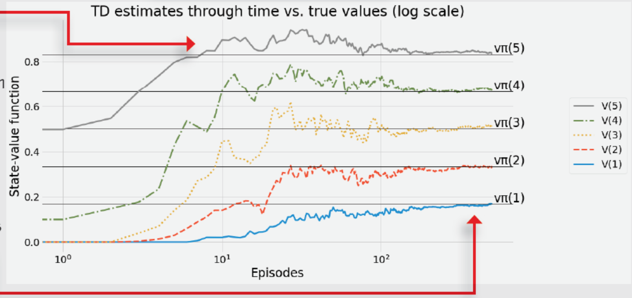
(2) A couple of pros though; first you can see all estimates get close to their true values very early on. Also, the estimates jump around the true values.



$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \Big[\overbrace{\underbrace{G_{t:t+1}}_{\text{TD target}} - V_t(S_t)}^{\text{TD error}} \Big]$$

(3) TD estimates are off target most of the time, but they are less jumpy. This is because TD targets are low variance, though biased. They use an estimated return for target.

(4) The bias shows, too. In the end, TD targets give up accuracy in order to become more precise. Also, they take a bit long before estimates ramp up, at least in this environment.



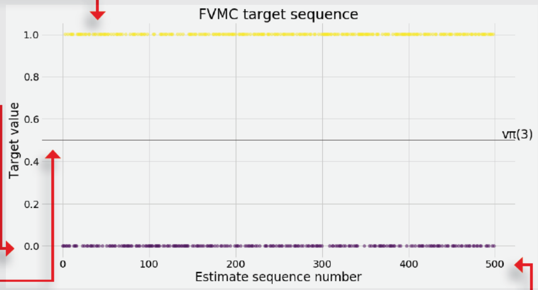
Tally It Up

MC targets high variance is evident, TD targets bias, too

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$$

(1) Here we can see the bias/variance tradeoff between MC and TD targets. Remember, the MC target is the return, which accumulates a lot of random noise. That means high variance targets.

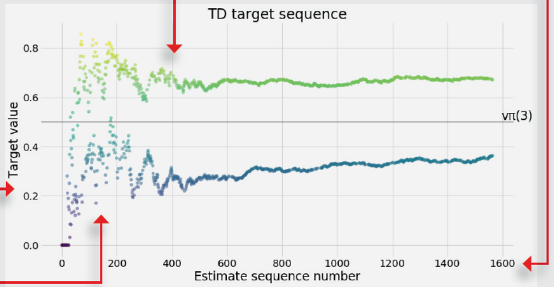
(2) These plots are showing the targets for the initial state in the RW environment. MC targets, the returns, are either 0 or 1 because the episode terminates either on the left, with a 0 return or on the right, with a 1 return, while the optimal value is 0.5!



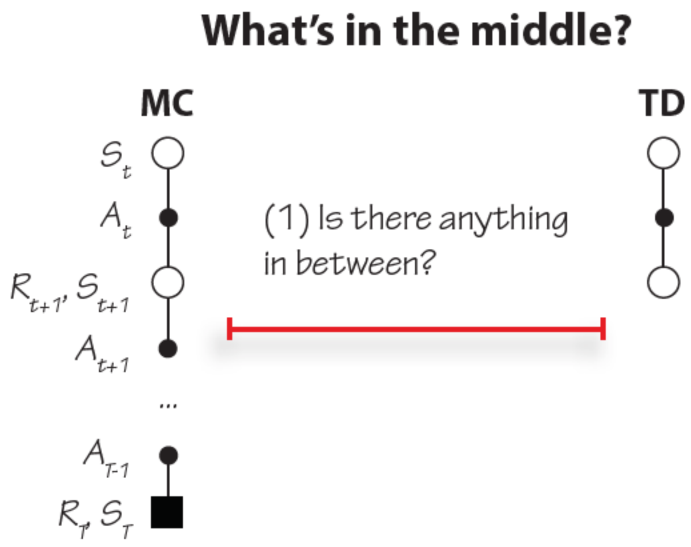
(3) TD targets are calculated using an estimated return. We use the value function to predict how much value we will get from the next state onwards. This helps us truncate the calculations and get more estimates per episode (as you can see on the x axis, we have ~1600 estimates in 500 episodes), but because we use $V_t(S_{t+1})$, which is an estimate and therefore likely wrong, TD targets are biased.

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1})$$

(4) Here you can see the range of the TD targets is much lower, MC alternates exactly between 1 and 0, TD jumps between approximately 0.7 and ~0.3, depending on which "next state" is sampled. But as the $V_t(S_{t+1})$ is an estimate, $G_{t:t+1}$ is biased, off target, inaccurate.



Is there anything in between?



n-step TD equations

Show Me the Math

N-step temporal-difference equations

$$S_t, A_t, R_{t+1}, S_{t+1}, \ldots, R_{t+n}, S_{t+n} \sim \pi_{t:t+n}$$

(1) Notice how in n-step TD we must wait n steps before we can update *V(s)*.

(2) Now, *n* doesn't have to be ∞ like in MC, or 1 like in TD. Here you get to pick. In reality *n* will be *n* or less if your agent reaches a terminal state. So, it could be less than *n*, but never more.

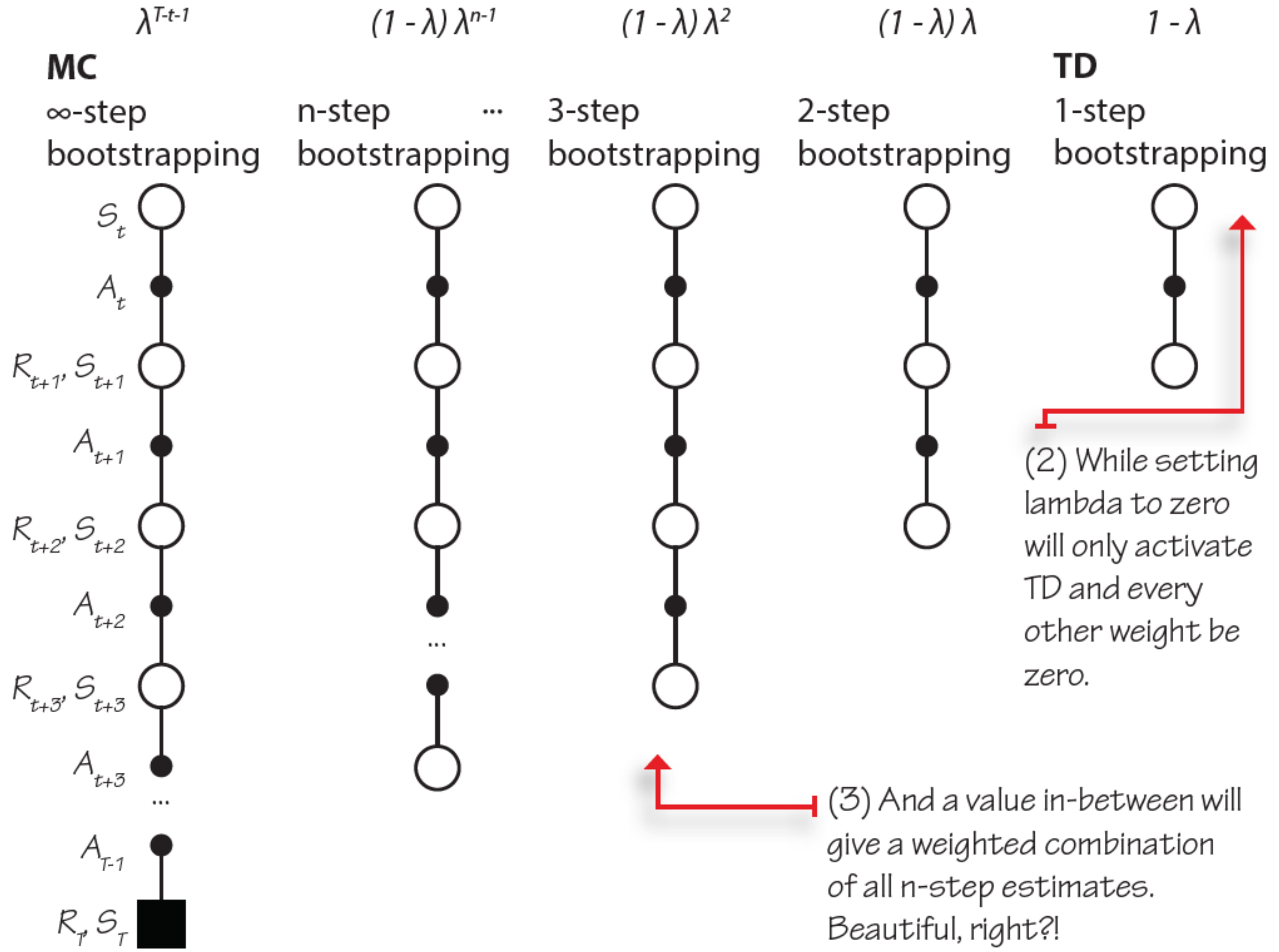
$$G_{t:t+n} = R_{t+1} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

(3) Here you see how the value function estimate gets updated approximately every *n* steps.

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha_t \Big[\overbrace{\underbrace{G_{t:t+n}}_{\text{n-step target}} - V_{t+n-1}(S_t)}^{\text{n-step error}} \Big]$$

(4) But after that, you can just plug-in that target as usual.

TD lambda



(1) First of all, what a beautiful algorithmic approach to unify MC and TD! See, if you set lambda to one, the only active weight will be MC.

(2) While setting lambda to zero will only activate TD and every other weight be zero.

(3) And a value in-between will give a weighted combination of all n-step estimates. Beautiful, right?!

TD lambda equations

Show Me the Math

Forward-view TD(λ)

(1) Sure, this is a loaded equation, but we will unpack it below. The bottom line is that we are using all n-step returns until the final step *T*, and weighting it with an exponentially decaying value.

$$G_{t:T}^{\lambda} = \underbrace{(1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n}}_{\text{Sum of weighted returns from 1-step to T-1 steps}} + \underbrace{\lambda^{T-t-1} G_{t:T}}_{\text{Weighted final return (T)}}$$

(2) The thing is, because *T* is variable, we need to weight the actual return with a normalizing value so that all weights add up to 1.

(3) All this equation is saying is that we will calculate the one-step return and weight it with the following factor. $\mapsto$ $1-\lambda$

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1})$$

(4) And also the two-step return and weight it with this factor. $(1-\lambda)\lambda$

$$G_{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

(5) Then the same for the three-step return, and this factor. $(1-\lambda)\lambda^2$

$$G_{t:t+3} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V_{t+2}(S_{t+3})$$

(6) You do this for all n-steps... $(1-\lambda)\lambda^{n-1}$

$$G_{t:t+n} = R_{t+1} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

(7) Until your agent reaches a terminal state. Then you weight by this normalizing factor. λ^{T-t-1}

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$$

(8) Notice the issue with this approach is that you must sample an entire trajectory before you can calculate these values.

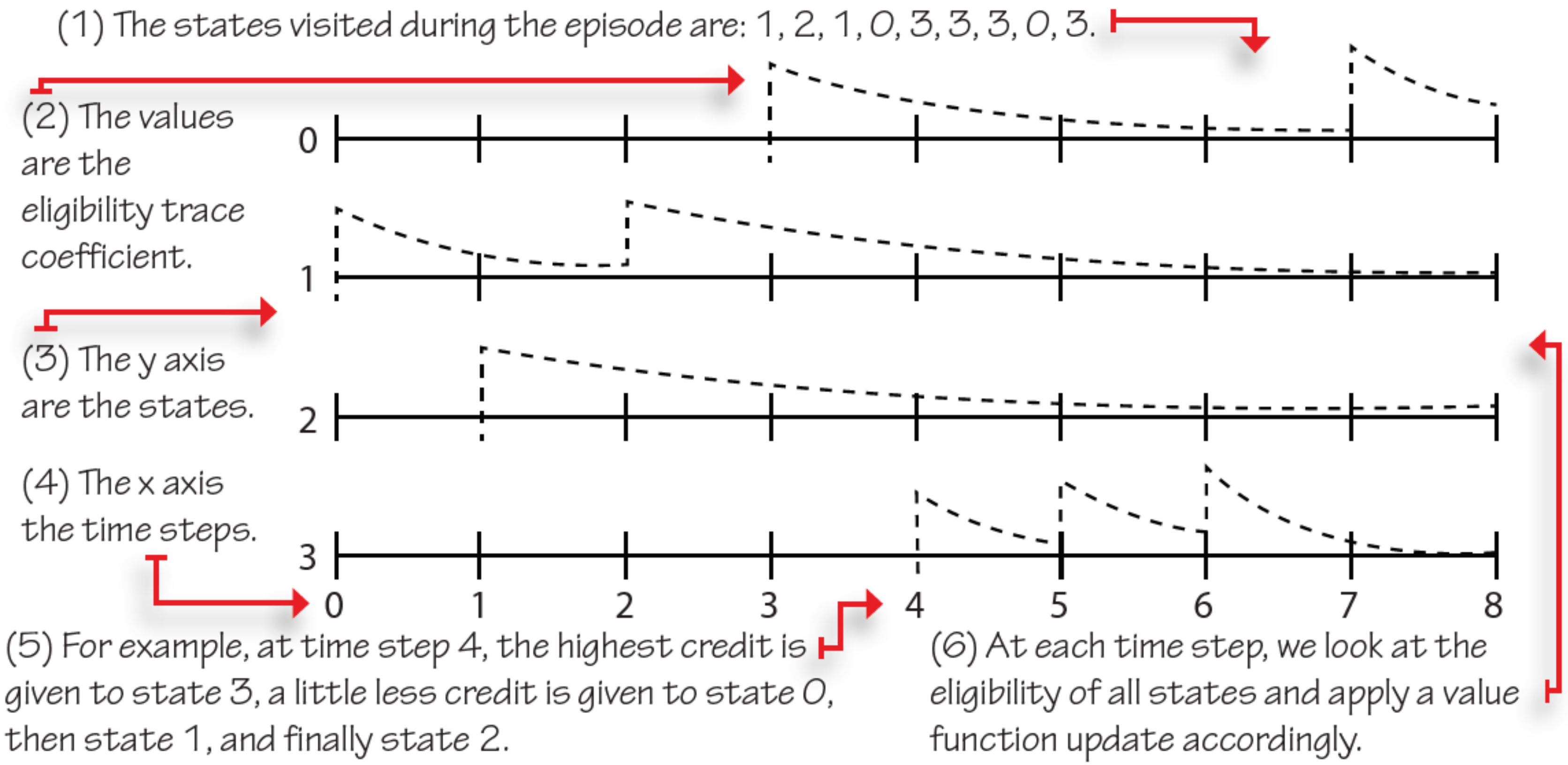
$$S_t, A_t, R_{t+1}, S_{t+1}, \ldots, R_T, S_T \sim \pi_{t:T}$$

(9) Here you have it, *V* will become available at time *T*.

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \Big[\overbrace{\underbrace{G_{t:T}^{\lambda}}_{\lambda\text{-return}} - V_{T-1}(S_t)}^{\lambda-\text{error}} \Big]$$

(10) Because of this.

Eligibility traces



TD(lambda) algorithm

Show Me the Math

Backward-view TD(λ) — TD(λ) with eligibility traces, "the" TD(λ)

(1) Every new episode we set the eligibility vector to 0. $\longmapsto$ $E_0 = 0$

(2) Then, we interact with the environment one cycle. $\longmapsto$ $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{t:t+1}$

(3) When you encounter a state S_t, make it eligible for an update... Technically, you increment its eligibility by 1. $\longmapsto$ $E_t(S_t) = E_t(S_t) + 1$

(4) We then simply calculate the TD error just as we have been doing so far.

$$\delta^{TD}_{t:t+1}(S_t) = \underbrace{R_{t+1} + \gamma V_t(S_{t+1})}_{\text{TD target}} - V_t(S_t)$$

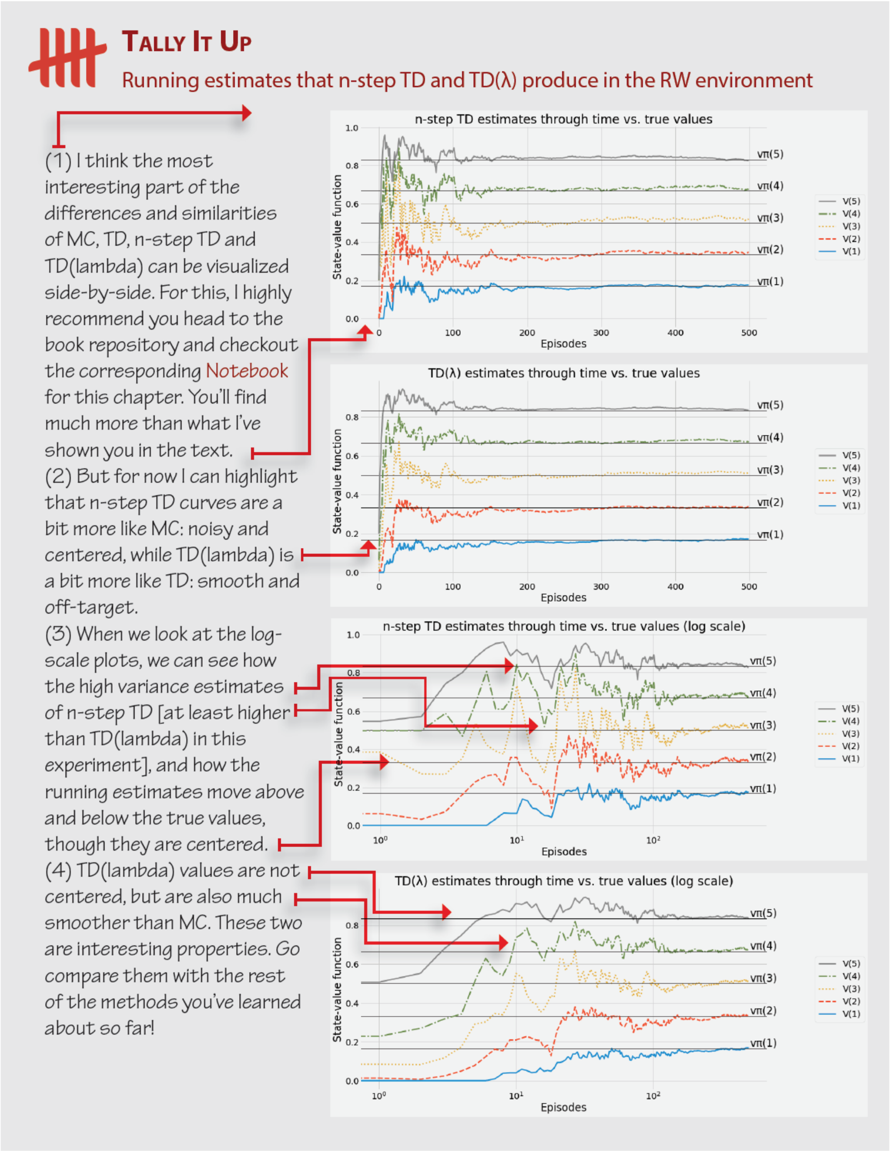
(5) However, unlike before, we update the estimated state-value function V, that is, the *entire* function at once, *every time step!* Notice I'm not using a $V_t(S_t)$, but a V_t instead. Because we are multiplying by the eligibility *vector*, all eligible states will get the corresponding credit.

$$V_{t+1} = V_t + \alpha_t \underbrace{\delta^{TD}_{t:t+1}(S_t)}_{\text{TD error}} E_t$$

(6) Finally, we decay the eligibility. $\longmapsto$ $E_{t+1} = E_t \gamma \lambda$

Recap: Prediction problem

- Allows us to accurately evaluate policies.
- Having accurate estimates makes policy improvement trivial.
- There are two core methods, Monte-Carlo prediction and Temporal-Difference Learning.
- One uses the actual returns and approximates the expectation by taking means. The other bootstraps on its own value estimates; uses partial or predicted returns.
- There are pros and cons in both!



Recommended reading.
Reinforcement Learning: An introduction (chapters 5, 6, 7, and 12)
http://incompleteideas.net/book/the-book-2nd.html

Outline

Bandit Problems

Prediction Problem

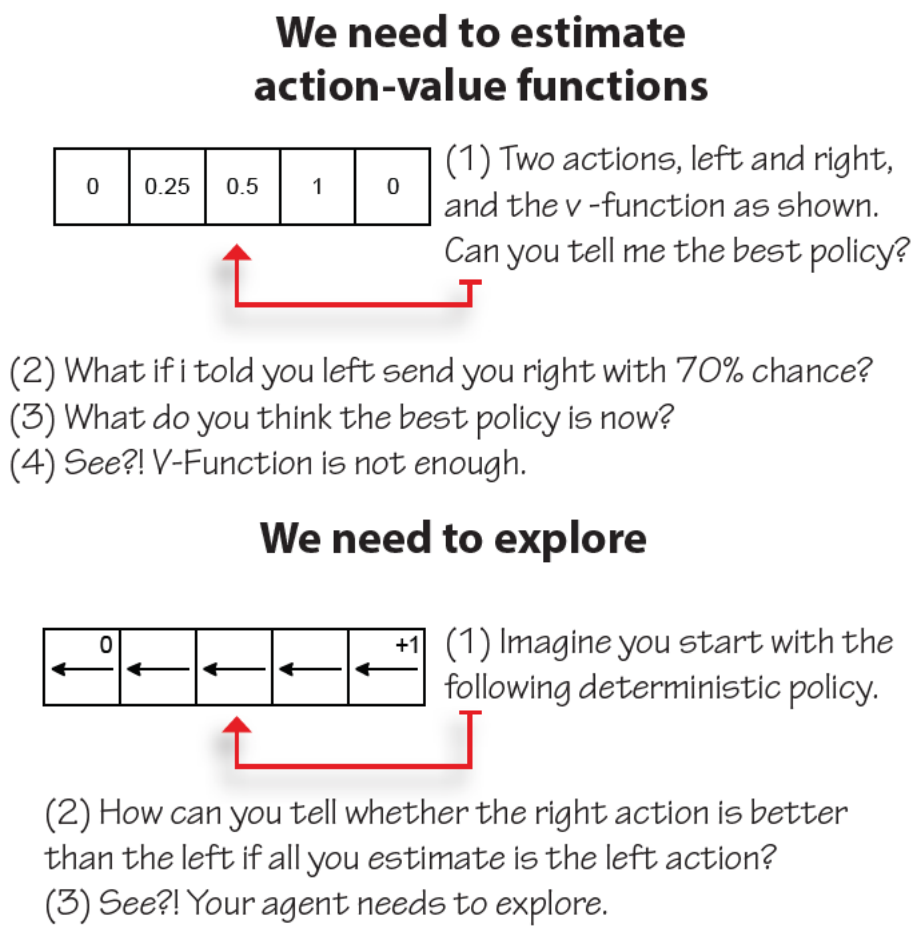
Control Problem

> “When it is obvious that the goals cannot be reached, don’t adjust the goals, adjust the action steps.”
>
> — Confucius
> Chinese teacher, editor, politician, and philosopher of the Spring and Autumn period of Chinese history

Georgia Tech
CREATING THE NEXT

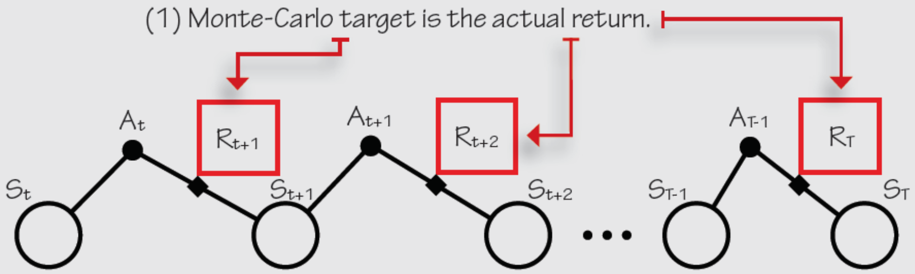
Control Problem

- Optimize policies.
- Find the best policies for any given environment.
- Needs accurate policy evaluation methods and exploration.
- This is the full reinforcement learning problem. Note: Not Deep Reinforcement Learning, which we'll discuss in next lecture.

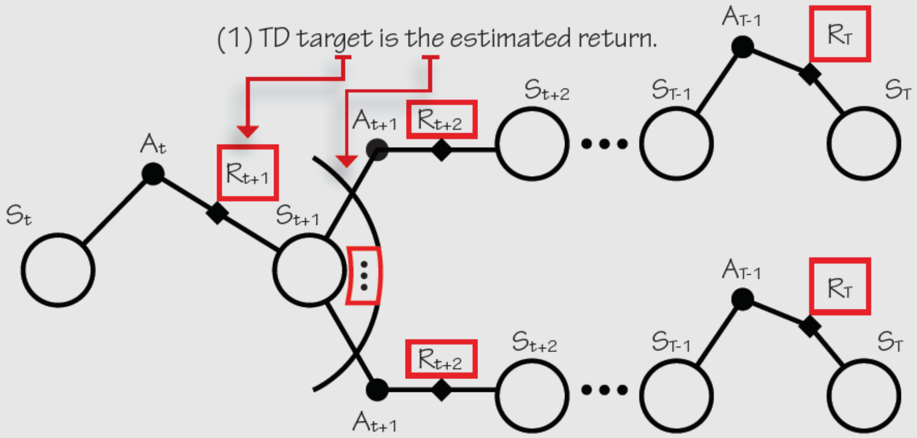


What estimation method to use?

Other important concepts worth repeating are the different ways value functions can be estimated. In general, all methods that learn value functions progressively move estimates a fraction of the error towards the targets. The general equation that most learning methods follow is: *estimate = estimate + step * error*. The *error* is simply the difference between a *sampled target* and the *current estimate*: *(target - estimate)*. The two main and opposite ways for calculating these targets are Monte-Carlo and Temporal-Difference learning.

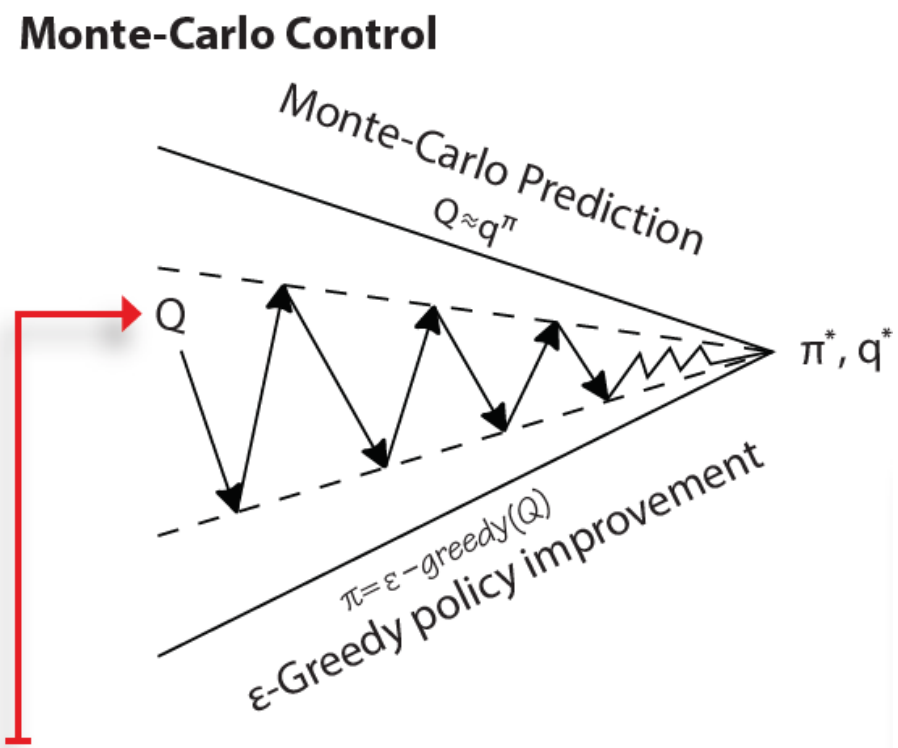


The Monte-Carlo target consists of the actual return. Really, nothing else. Monte-Carlo estimation consists of adjusting the estimates of the value functions using the empirical (observed) mean return in place of the expected (as if you could average infinite samples) return.



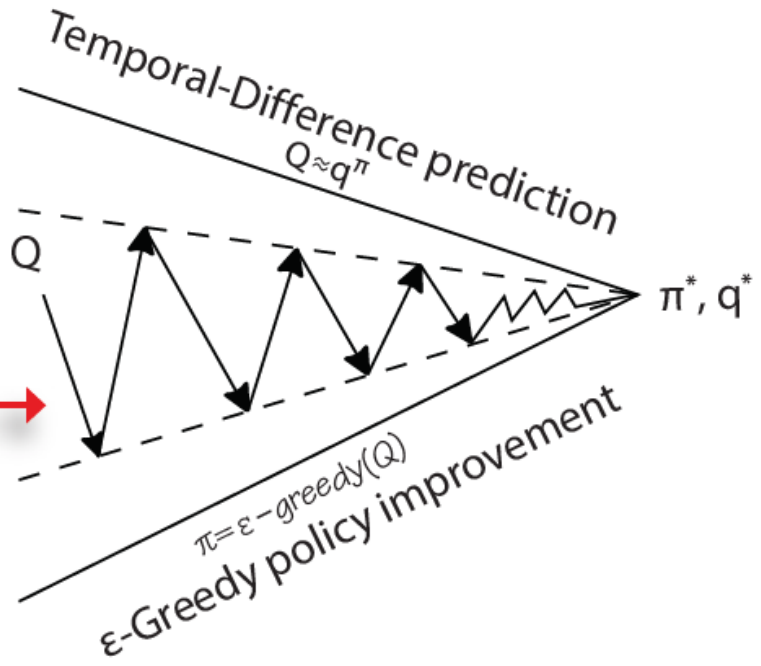
The Temporal-Difference target consists of an estimated return. Remember "bootstrapping"? It basically means using the estimated expected return from *later* states, for estimating the expected return from the *current* state. TD does that. Learning *a guess* from *a guess*. The TD target is formed by using a single reward and the estimated expected return from the next state using the running value function estimates.

Monte-Carlo Control



(3) MC Control estimates a Q-function, has a truncated MC prediction phase followed by an e-greedy policy improvement step.

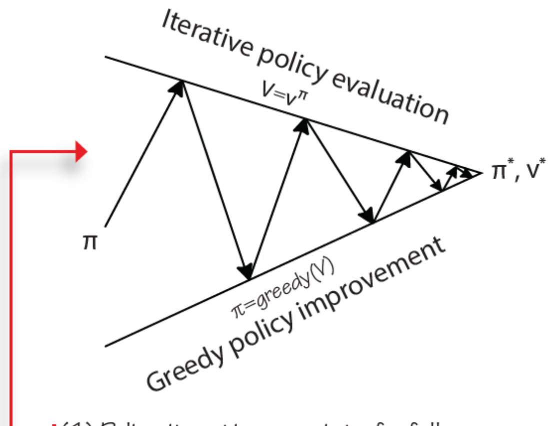
TD Control: Sarsa



(4) Sarsa has pretty much the same as MC control except a truncated TD prediction for policy evaluation.

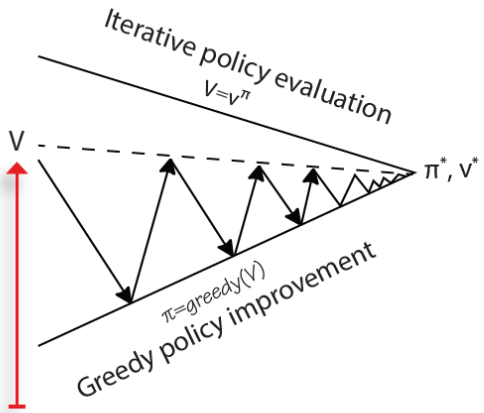
Planning methods, RL methods

Policy iteration



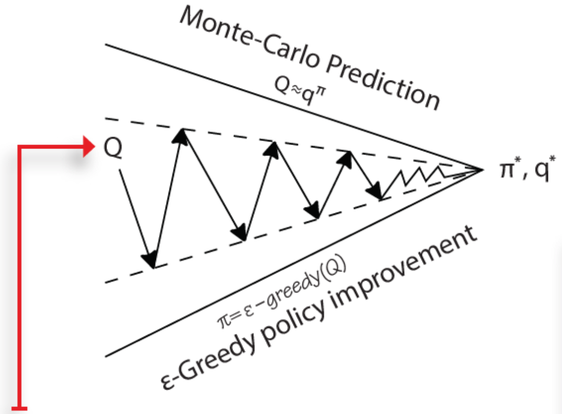
(1) Policy iteration consist of a full convergence of iterative policy evaluation alternating with greedy policy improvement.

Value iteration



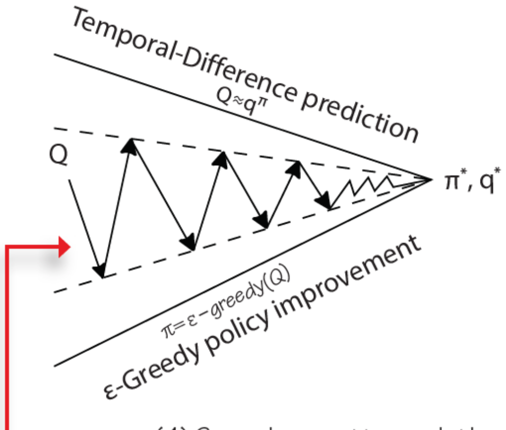
(2) Value iteration starts with an arbitrary value function and has a truncated policy evaluation step.

Monte-Carlo Control



(3) MC Control estimates a Q-function, has a truncated MC prediction phase followed by an e-greedy policy improvement step.

Sarsa



(4) Sarsa has pretty much the same as MC control except a truncated TD prediction for policy evaluation.

Q-Learning: off-policy learning

Show Me the Math

Sarsa vs. Q-learning update equations

(1) The only difference between Sarsa and Q-learning is the action used in the target.

(2) This is Sarsa update equation.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[\overbrace{\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{Sarsa target}} - Q(S_t, A_t)}^{\text{Sarsa error}} \right]$$

(3) It uses the action actually taken in the next state to calculate the target.

(4) This one is Q-learning's.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[\overbrace{\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}_{\text{Q-learning target}} - Q(S_t, A_t)}^{\text{Q-learning error}} \right]$$

(5) Q-learning uses the action with the maximum estimated value in the next state, despite the action actually taken.

On-policy vs. Off-policy learning

ŘŁ **With An RL Accent**

On-policy vs. Off-policy learning

On-policy learning: Refers to methods that attempt to evaluate or improve the policy used to make decisions. It is straightforward; think about a single policy. This policy generates behavior. Your agent evaluates that behavior and select areas of improvement based on those estimates. Your agent learns to assess and improve the same policy it uses for generating the data.

Off-policy learning: Refers to methods that attempt to evaluate or improve a policy different from the one used to generate the data. This one is more complex. Think about two policies. One produces the data, the experiences, the behavior, but your agent uses that data to evaluate, improve, and overall learn about a different policy, a different behavior. Your agent learns to assess and improve a policy different than the one used for generating the data.

Convergence: GLIE: Greedy in the Limit with Infinite Exploration and Stochastic Approximation theory.

- GLIE:
  - All state-action pairs must be explored infinitely often.
  - The policy must converge on a greedy policy.
- What this means in practice is that an e-greedy exploration strategy, for instance, must slowly decay epsilon towards zero. If it goes down too quickly, the first condition may not be met, if it decays too slowly, well, it takes longer to converge.
- Notice that for off-policy RL algorithms, such as Q-learning, the only requirement of these two that holds is the first one. The second one is no longer a requirement because in off-policy learning, the policy learned about is different than the policy we are sampling actions from. Q-learning, for instance, only requires all state-action pairs to be updated sufficiently, and that is covered by the first condition above.

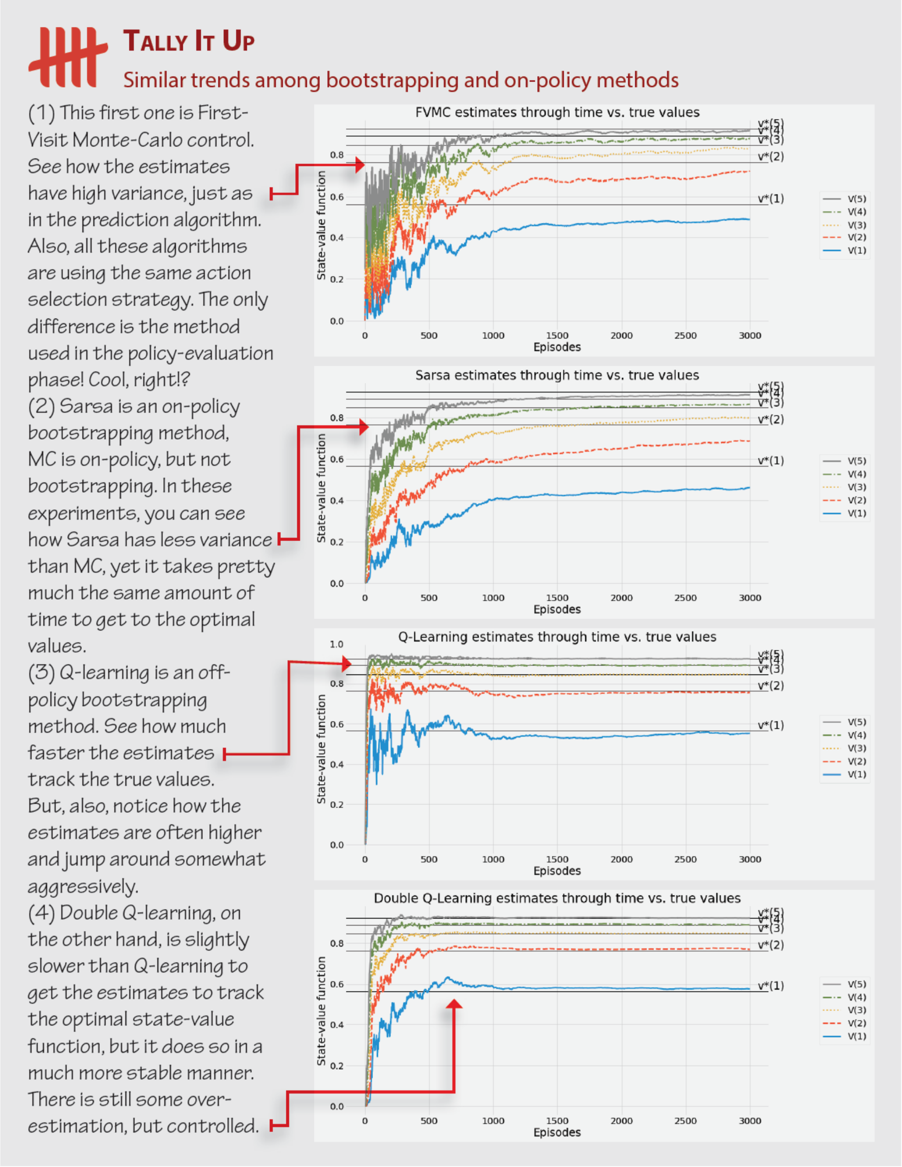
There is another set of requirements for general convergence based on Stochastic Approximation Theory that applies to all these methods. Because we are learning from samples, and samples have some variance, the estimates won't converge unless we also push the learning rate, alpha, towards zero:

- The sum of learning rates must be infinite.
- The sum of squares of learning rates must be finite.

That means you must pick a learning rate that decays but never reaches zero. For instance, if you use 1/t or 1/e, the learning rate is initially large enough to ensure the algorithm doesn't follow only a single sample too tightly but becomes small enough to ensure it finds the signal behind the noise.

Recap: Control problem

- It's what we are looking, how to find optimal control policies.
- There is a profound synergy between policy evaluation and policy improvement.
- The control problem consists on estimation action-value functions, and correctly balancing exploration and exploitation.



Recommended reading.
Reinforcement Learning: An introduction (chapters 5, 6)
http://incompleteideas.net/book/the-book-2nd.html

Thank you!