Introduction to Reinforcement Learning

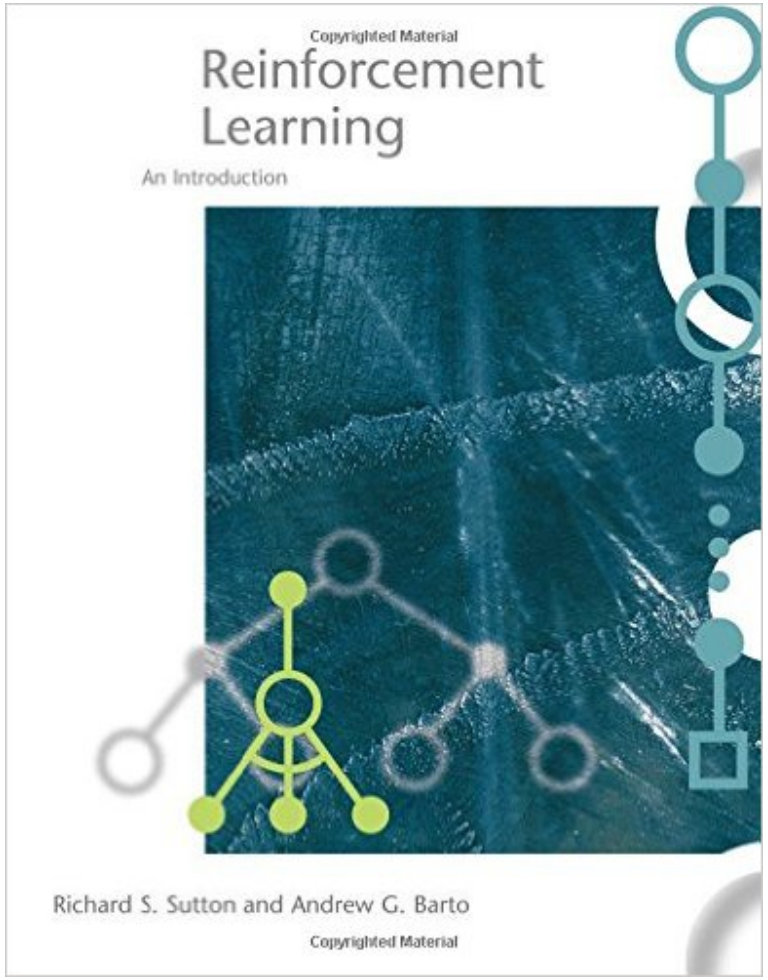
Gabriela Tavares

gtavares@google.com

April 18, 2019

Lecture outline

- Part 1: Dynamic Programming and Monte Carlo
- Part 2: Temporal Difference Learning
- Part 3: Survey of Deep RL Topics

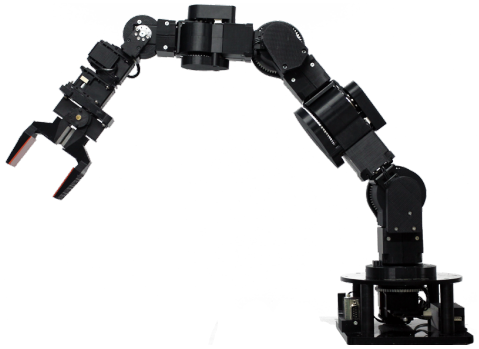


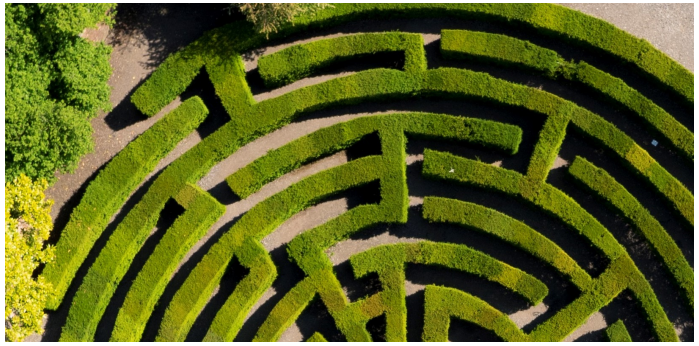
Rich Sutton

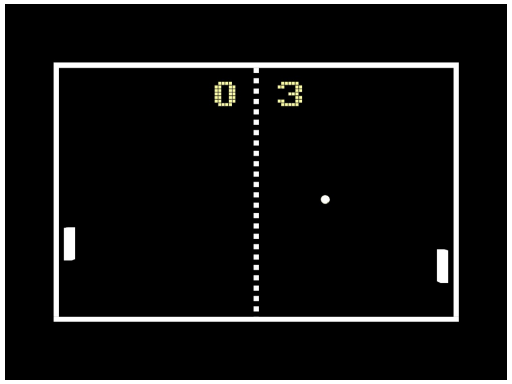
Andy Barto

Uses of Reinforcement Learning

Reinforcement Learning solves problems of control: choosing the right (best) action at the right time.





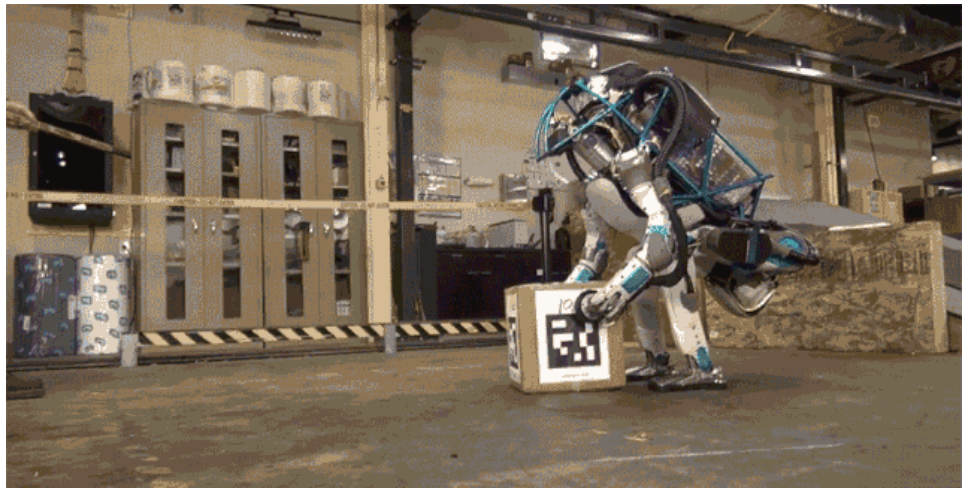


Applications of Reinforcement Learning

Some successful applications include:

- Backgammon (Tesauro, 1994)
- Inventory management (Van Roy et al., 1996)
- Dynamic channel allocation (Singh and Bertsekas, 1997)
- Elevator scheduling (Crites and Barto, 1998)
- Robocop soccer (Stone and Velloso, 1999)
- Helicopter control (Ng, 2003; Abbeel and Ng, 2006)
- HIV treatment strategies (Ernst et al., 2006)
- Playing Pong (Mnih et al., 2015) and Go (Silver et al., 2016)

Also: robots!

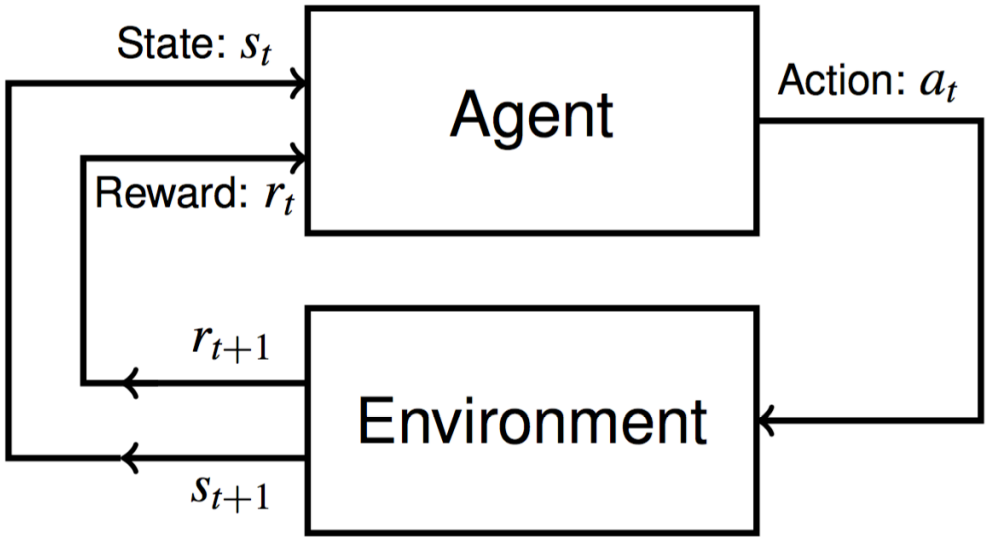


Dynamic Programming and Monte Carlo

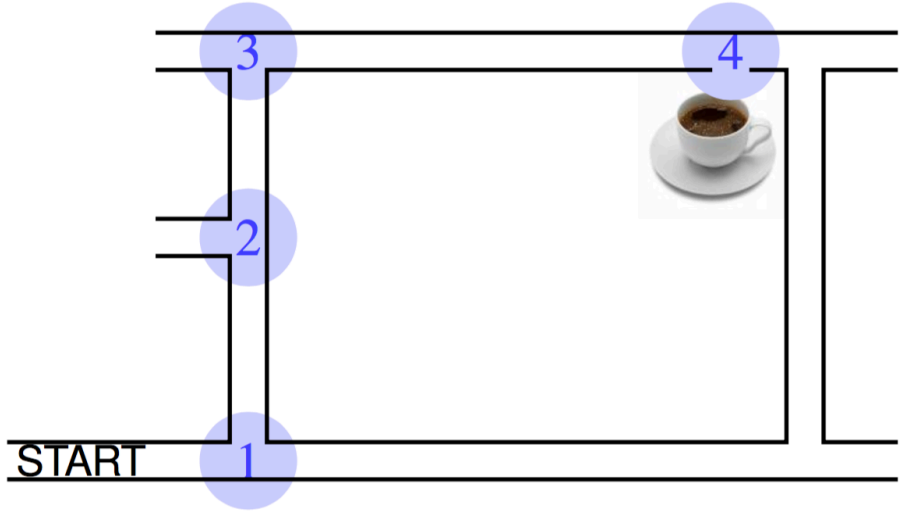
Agent-environment interface

We want to model how agents interact with their environment.

- At each time step t, the environment is in some state (configuration) s_t
- At time t, the agent perceives state s_t and chooses action a_t
- The environment changes to state s_{t+1}
- At time $t+1$, the agent perceives state s_{t+1} and receives reward r_{t+1}



An example: coffee policy



States are 1, 2, 3 and 4.

A policy for this task is:

$\pi(1)$ = turn left

$\pi(2)$ = straight on

$\pi(3)$ = turn right

$\pi(4)$ = go through door

More generally we use a probabilistic policy:

$$\pi_t(s, a) = p(a_t = a | s_t = s)$$

Markov Decision Processes

Markov Decision Processes (MDPs) describe a class of control problems and consist of:

- A set of states, S.
- A set of actions, A.
- A transition function, t, where $t(s, a, s') = p(s'|s, a)$ for $s, s' \in S$ and $a \in A$.
- A reward function, r, where $r(s, a, s')$ is the reward for moving from s to s' under a.
- A policy π takes the current state and gives an action in response.
  - This can be deterministic, i.e., $\pi(s) = a$
  - Or probabilistic, i.e., $\pi(s, a) = p(a|s, \pi)$

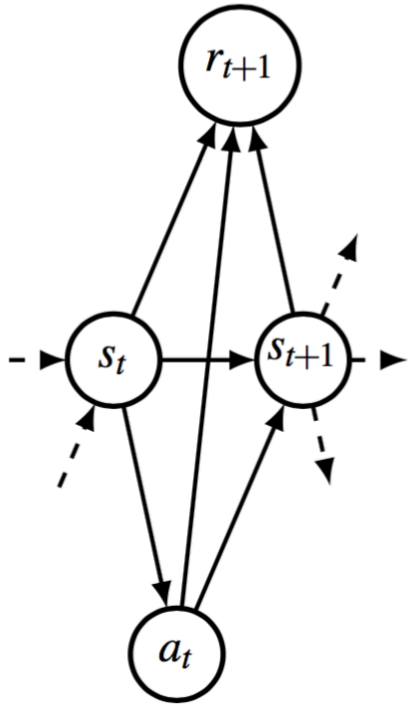
Modeling Markov Decision Processes

The Markov property/assumption:

$$p(s_{t+1}, r_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0) = p(s_{t+1}, r_{t+1} | s_t, a_t)$$

States: s_t, s_{t+1} Actions: a_t Rewards: r_{t+1}

- Variable s_t is the state of the world at time t
- Agent takes action a_t and the world state changes to s_{t+1}
- This transition gives a reward r_{t+1}
- We can also think in terms of costs (i.e. negative rewards)



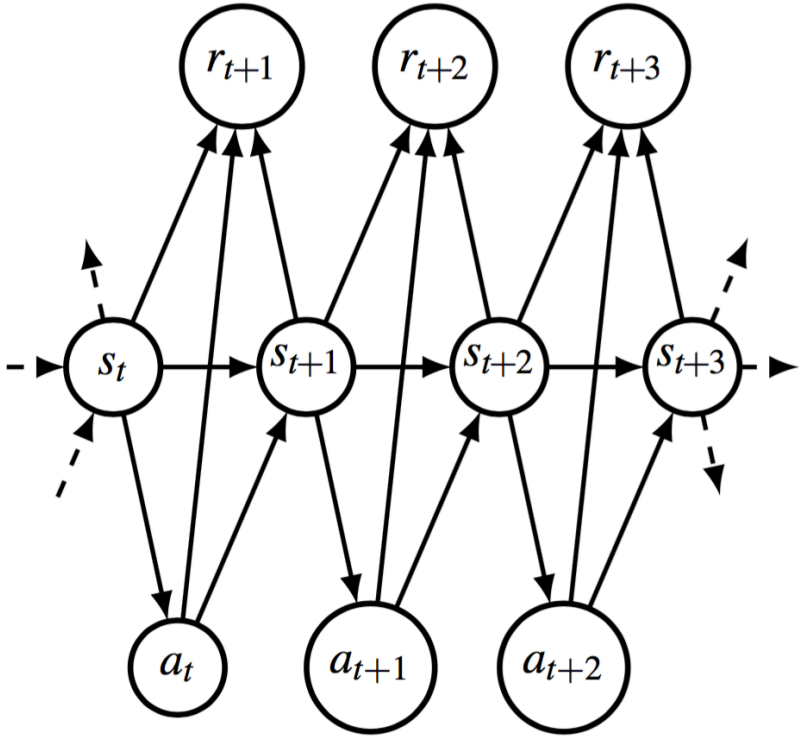
Modeling Markov Decision Processes

Recall that:

$\pi_t(s, a) = p(a_t = a | s_t = s)$

$t(s, a, s') = p(s_{t+1} = s' | s_t = s, a_t = a)$

$r(s, a, s')$ is a reward for going from s to s' under a.



What do we want to achieve?

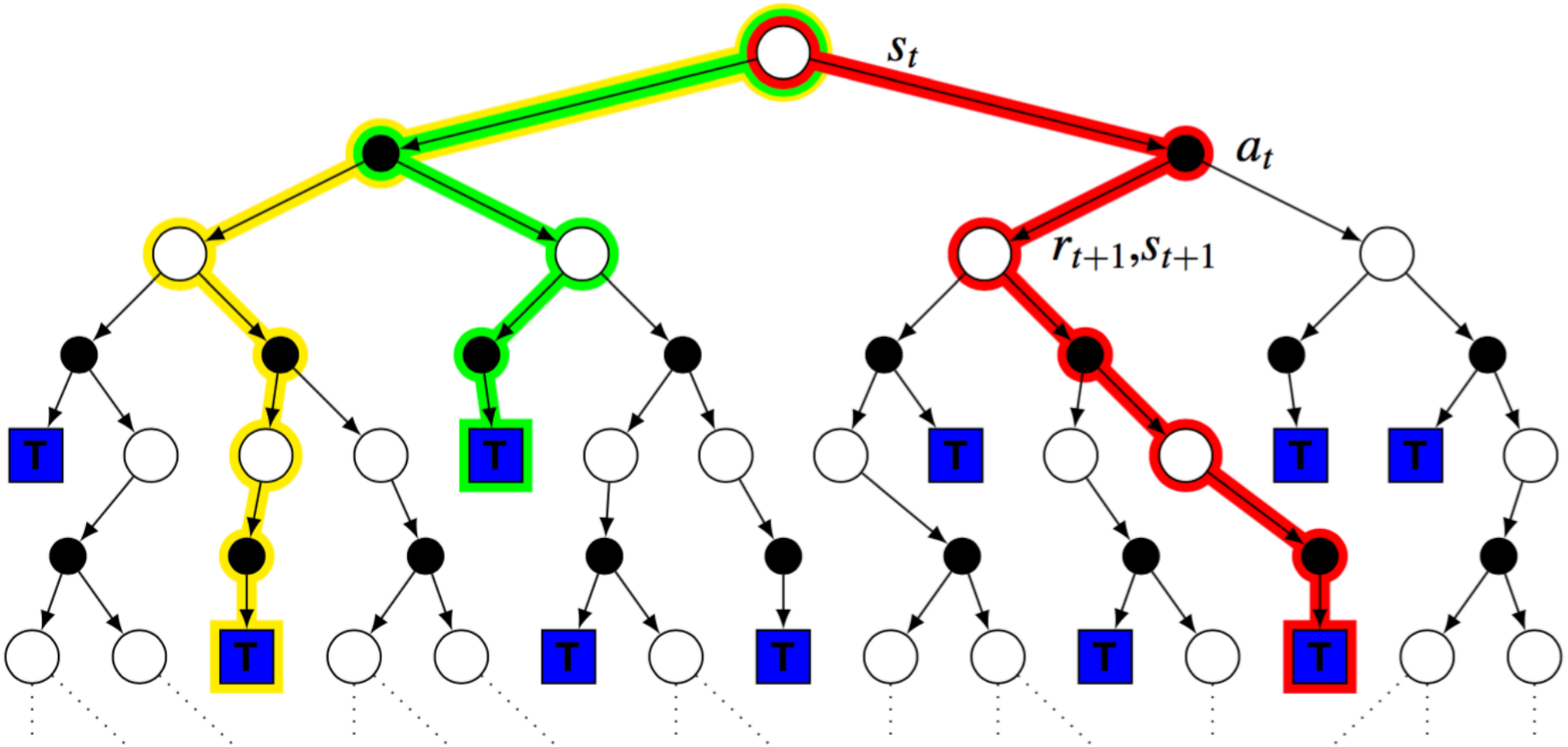
The question we want to answer is: how do we find the best policy for controlling the MDP?

- For this we need to know how to rank or score policies.
- An obvious choice is to obtain the expected performance of a policy. So we must be able to evaluate performance.
- We often need more details than this: we may want to score states and actions as well as policies.

Traces

How do we evaluate policy performance?

We begin by considering traces.



Aggregating rewards

- A **trace** is a state-action-reward sequence, $\tau = s_0, a_0, r_1, s_1, a_1, r_2, \ldots, r_N, s_N = \tau_{[0,N]}$
- A **return** measures the value of τ,

$$R(\tau) = \sum_{k=0}^{N} \gamma_k r_{k+1}$$

- γ_k is the discount factor at time k. Three common choices are:
  - A simple sum, $\gamma_k = 1$.
  - Average reward, $\gamma_k = \frac{1}{N}$.
  - Geometric discount, $\gamma_k = \gamma^k$, with $0 < \gamma < 1$.
- We will use geometric discount:

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots$$

Values

- The **expected return** predicts average trace return for policy π:

$$J(\pi) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{k+1} \,|\pi\right)$$

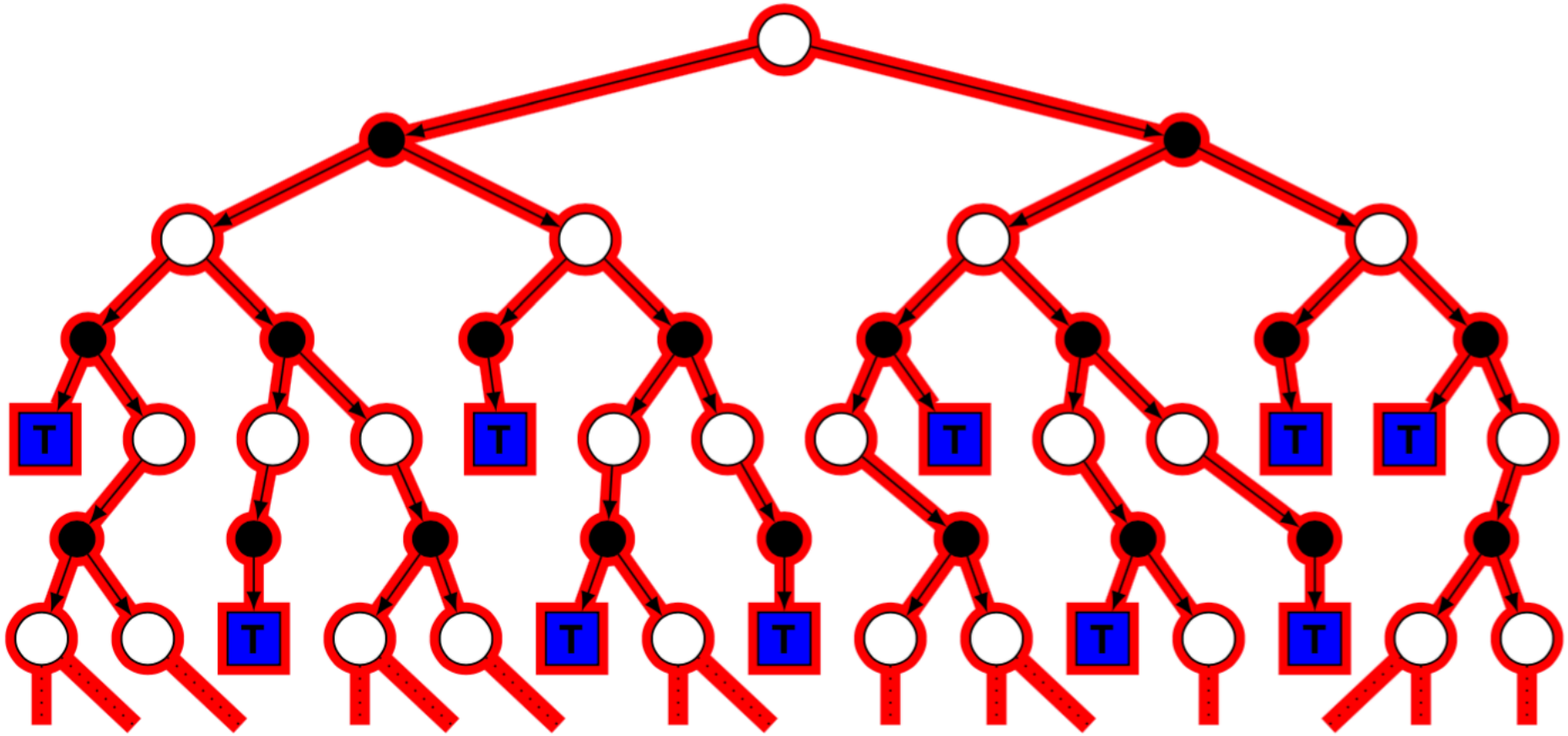
- The **value function** is the expected return given a state:

$$V^{\pi}(s) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\pi, s_t = s\right)$$

- The **Q function** (state-action value function) is the expected return given a state and action:

$$Q^{\pi}(s,a) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\pi, s_t = s, a_t = a\right)$$

Brute force estimates



Brute force estimates

Naively, we might want to estimate these values by performing a weighted sum over all possible traces.

- Expected return:

$$J(\pi) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{k+1} \,|\pi\right) = \sum_{\tau} p(\tau|\pi) R(\tau)$$

- State-value function:

$$V^{\pi}(s) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\pi, s_t = s\right) = \sum_{\tau} p(\tau|\pi, s_0 = s) R(\tau)$$

- State-action value function:

$$Q^{\pi}(s,a) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\pi, s_t = s, a_t = a\right) = \sum_{\tau} p(\tau|\pi, s_0 = s, a_0 = a) R(\tau)$$

More on traces

Consider the trace $\tau = s_0, a_0, r_1, s_1, a_1, r_2, \ldots, r_N, s_N$. We can relate certain properties of traces using t and π:

$$p_\tau(s_t = s, a_t = a|\pi) = \pi(s, a)\, p(s_t = s|\pi)$$

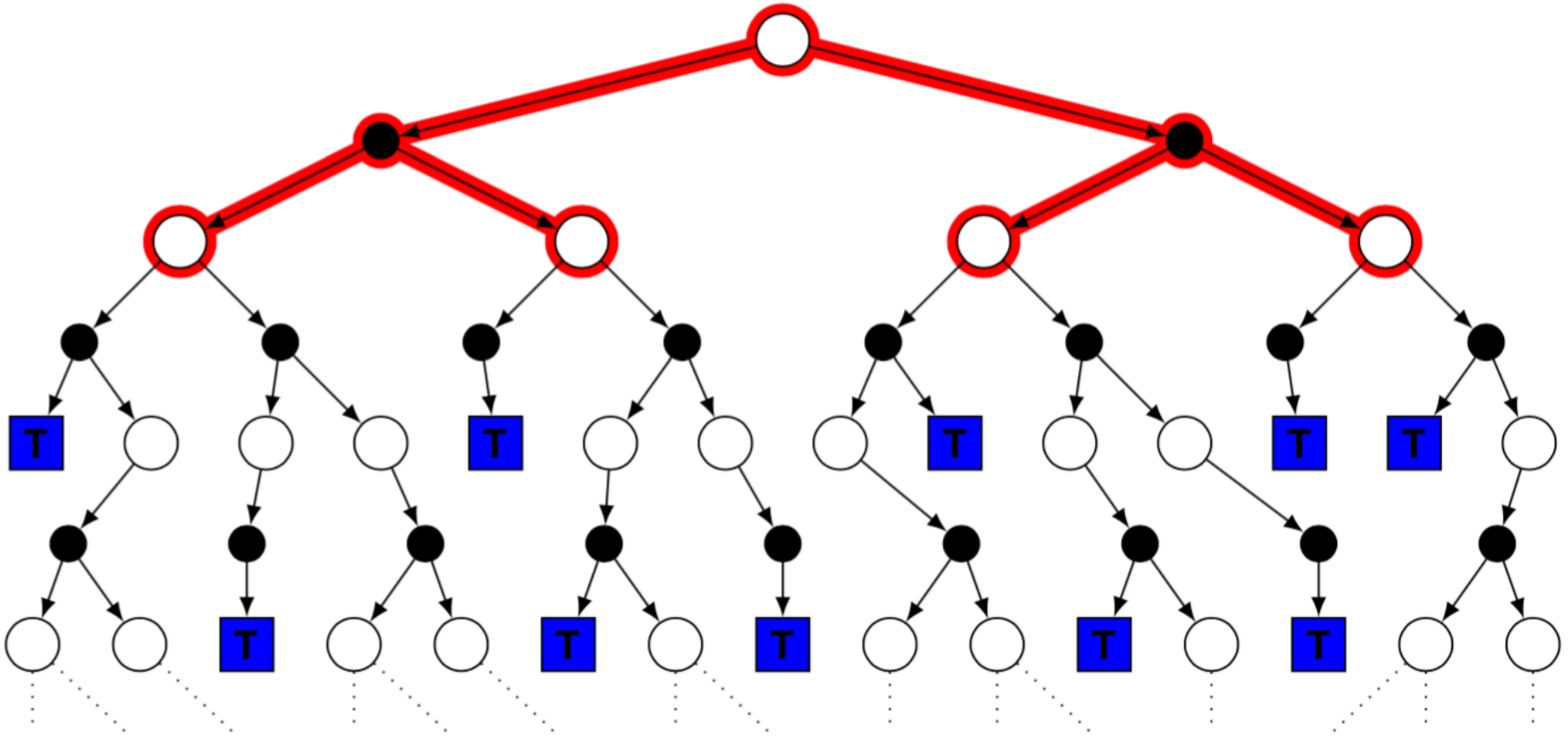
and

$$p_\tau(s_t = s, a_t = a, s_{t+1} = s'|\pi) = t(s, a, s')p(s_t = s, a_t = a|\pi) = t(s, a, s')\pi(s, a)p(s_t = s|\pi)$$

And the probability of a trace:

$$p(\tau|\pi) = p(s_0)\prod_{t=0}^{N-1} p(a_t|s_t)p(s_{t+1}|s_t, a_t) = p(s_0)\prod_{t=0}^{N-1} \pi(s_t, a_t)t(s_t, a_t, s_{t+1})$$

A smarter way



A smarter way

Using the Markov property we can rewrite V^{π}:

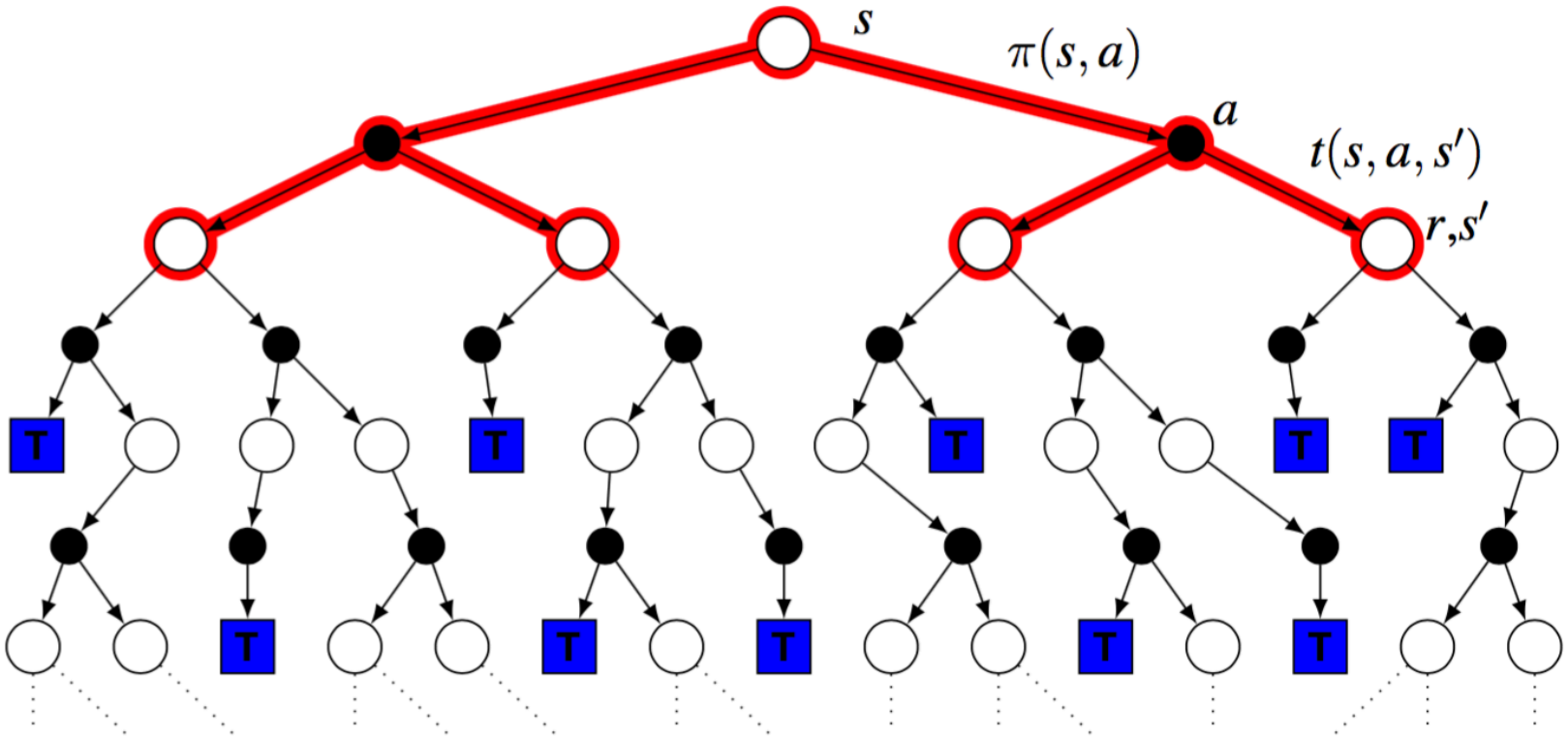
$$V^{\pi}(s) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} \,|\pi, s_t = s\right) = \sum_{a} \pi\,(s,a) \sum_{s'} t(s,a,s')\left(r(s,a,s') + \gamma V^{\pi}(s')\right)$$

This is called the **Bellman equation** for fixed π.

We can prove this using the Markov property and our definitions of $V^{\pi}(s)$, $p(\tau|s_0 = s, \pi)$ and $R(\tau)$.

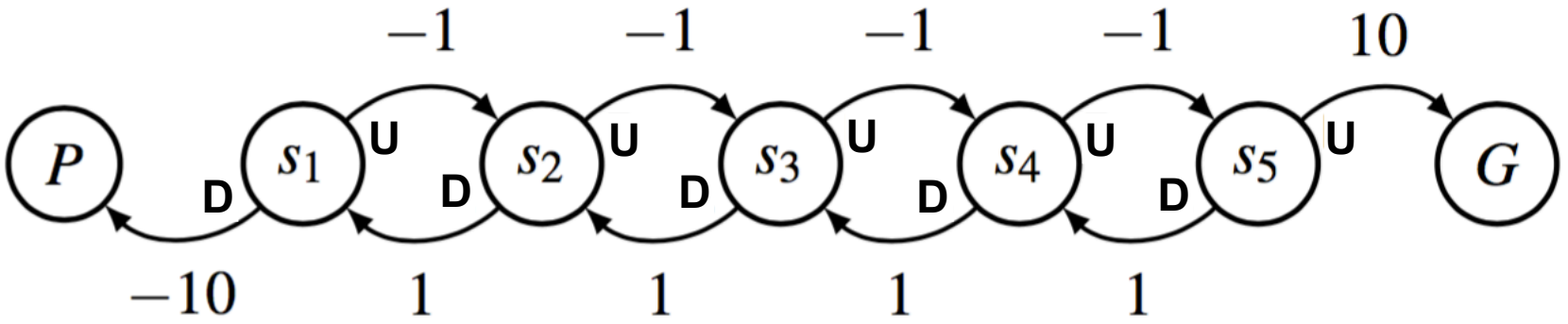
(…but we won't)

Dynamic Programming estimates



The Bellman equation allows us to look just one step ahead in the tree and use the estimates located there.

Example: a very simple MDP (stair climbing)



The start state is s_3. States P and G are absorbing. $\gamma = 0.9$.

- Begin with an unbiased random policy: for all states s, $\pi(s, D) = \pi(s, U) = \frac{1}{2}$.
- We want to obtain the value $V^{\pi}(s)$ at each state s.
- What is the best policy, π^*? How will $V^{\pi}(s)$ help us find π^*?

Estimating state values

How do we estimate V^{π} for some MDP?

- We begin with estimates $\hat{V}(s) = 0$ for all states s.
- We can get improved estimates using updates based on the Bellman equation:

  For all s,

$$\hat{V}'(s) \leftarrow \sum_{a} \pi(s,a) \sum_{s'} t(s,a,s') \left(r(s,a,s') + \gamma \hat{V}(s') \right)$$

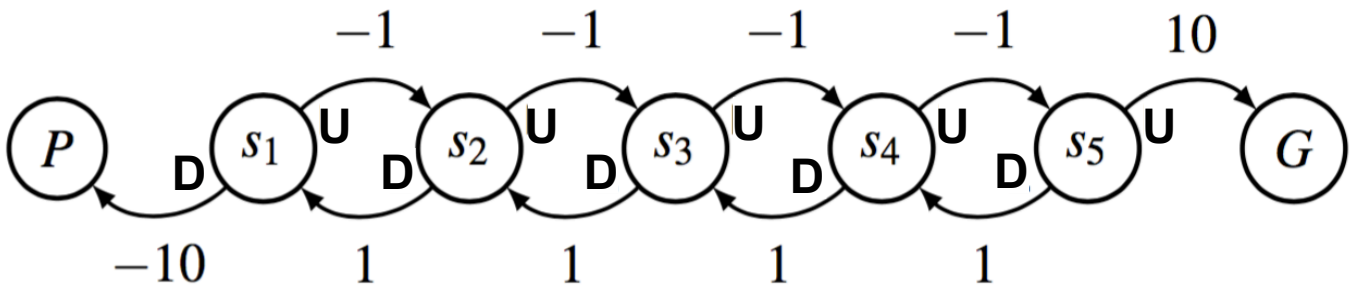
- We set each $\hat{V}(s) \leftarrow \hat{V}'(s)$, and repeat the above update until the desired accuracy.

Policy Evaluation algorithm

1: **procedure** POLICYEVALUATION(π,θ)
2: Initialise $\hat{V}(s) = 0$ for all $s \in S$
3: **repeat**
4: **for all** $s \in S$ **do**
5: $\hat{V}'(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} t(s,a,s') \left(r(s,a,s') + \gamma \hat{V}(s')\right)$
6: $\Delta \leftarrow \max_{s \in S} \mid \hat{V}'(s) - \hat{V}(s) \mid$
7: **for all** $s \in S$ **do**
8: $\hat{V}(s) \leftarrow \hat{V}'(s)$
9: **until** $\Delta < \theta$

Policy Evaluation on stair climbing MDP

Policy evaluation for $\pi(s, D) = \pi(s, U) = \frac{1}{2}$, with $\gamma = 0.9$:



$\hat{V}_0$:	0.0	0.0	0.0	0.0	0.0	0.0	0.0
$\hat{V}_1$:	0.0	−5.5	0.0	0.0	0.0	5.5	0.0
$\hat{V}_2$:	0.0	−5.5	−2.48	0.0	2.48	5.5	0.0
$\hat{V}_3$:	0.0	−6.61	−2.48	0.0	2.48	6.61	0.0
$\hat{V}_4$:	0.0	−6.61	−2.98	0.0	2.98	6.61	0.0
$\hat{V}_\infty$:	0.0	−6.90	−3.10	0.0	3.10	6.90	0.0

More on Policy Evaluation

Policy Evaluation:

- Estimates the value function, i.e. $\forall s$, $\hat{V}(s) \approx V^{\pi}(s)$.
- Requires direct access to transition probabilities and reward function (i.e., the model of the MDP).
- Requires the Markov property.
- Can use any initial values for $\hat{V}_0$.
- Can be performed in-place, i.e., using only one copy of the estimates $\hat{V}$ (this gives a small performance gain).

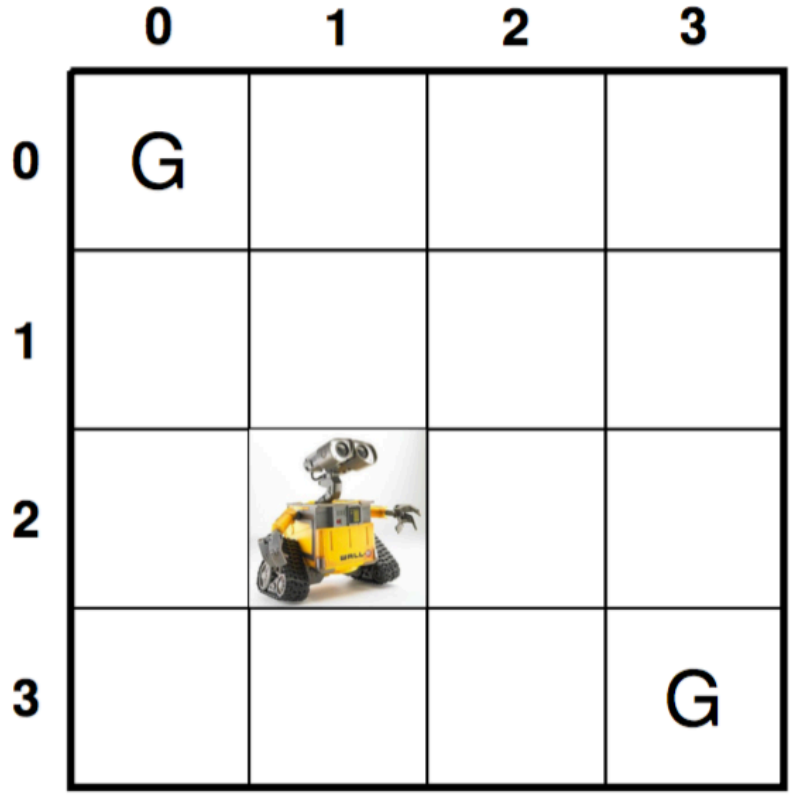
Using $\hat{V}$

A grid world example:

- Actions R, U, L and D
- Action effects are deterministic
- (0,0) and (3,3) are absorbing (goal) states
- $r = -1$ on all transitions
- $\pi(s, a) = \frac{1}{4}$ for all s and a

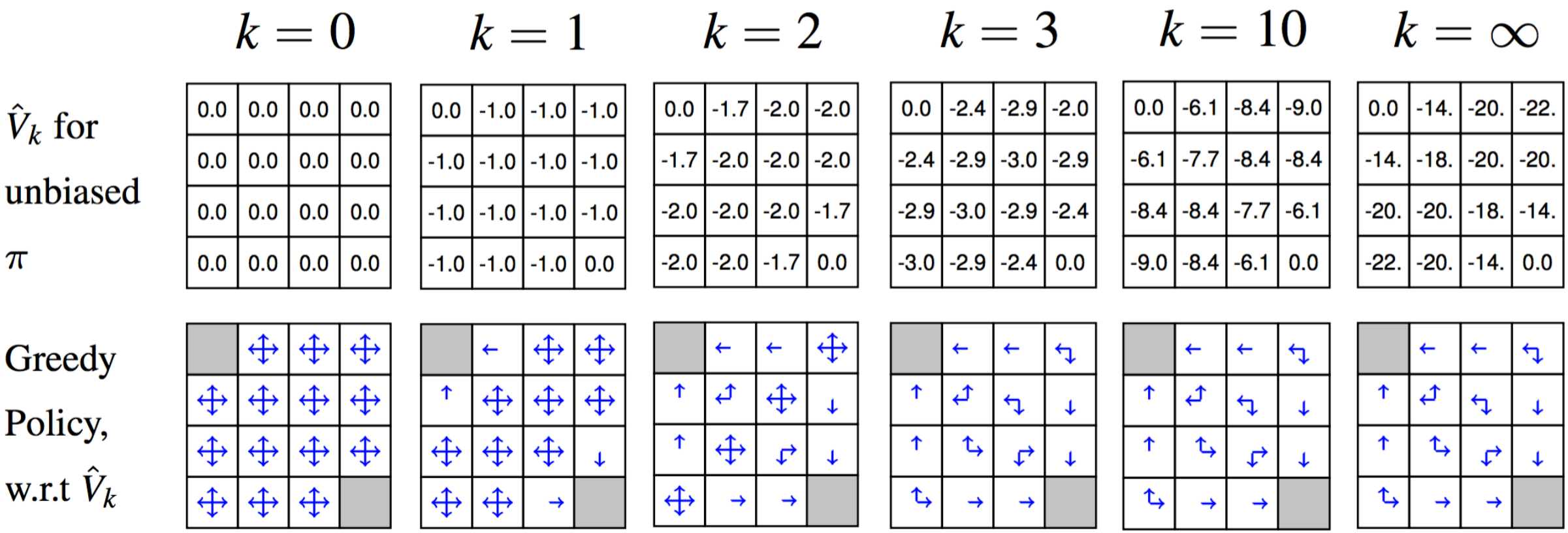
What is the best policy?

Can we use estimates $\hat{V}(s)$ to find it?



(Sutton and Barto)

Policy Evaluation on grid world MDP



(Sutton and Barto)

Policy Iteration algorithm

Using the estimates $\hat{V}$, we can **greedily** choose a better (deterministic) policy, π', where:

$$\pi'(s) = \arg\max_a \mathbb{E}(r_{t+1} + \gamma \hat{V}(s_{t+1}) | s_t = s, a_t = a)$$

$$\pi'(s) = \arg\max_a \sum_{s'} t(s, a, s')(r(s, a, s') + \gamma \hat{V}(s'))$$

We can then alternate between the policy evaluation step, E, and a policy improvement step, I, until there is no further change in the policy:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi^* \xrightarrow{E} V^*$$

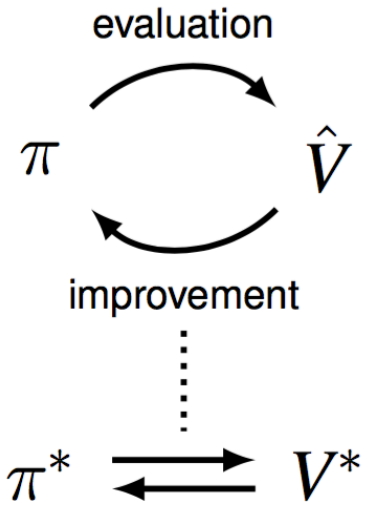
(this is similar to the EM algorithm)

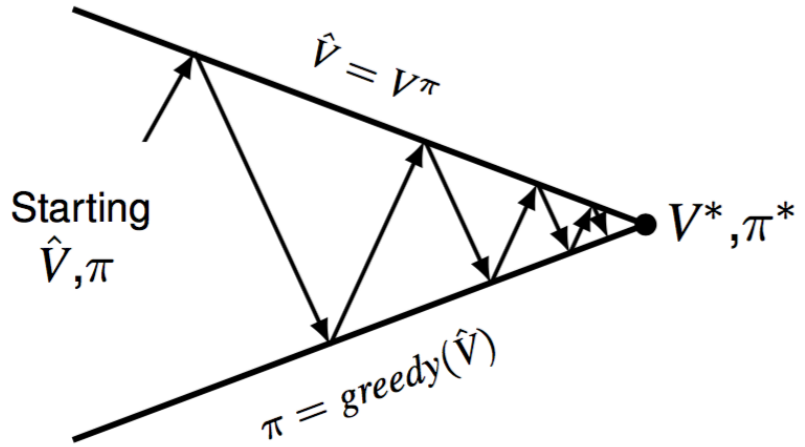
Policy Iteration algorithm

1: **procedure** POLICYITERATION(θ)
2: Initialise $\hat{V}(s) = 0$ and $\pi(s) \in A$ for all $s \in S$.
3: **repeat**
4: $\hat{V} \leftarrow$ POLICYEVALUATION(π,θ)
5: *stable* $\leftarrow$ *true*
6: **for all** $s \in S$ **do**
7: $b \leftarrow \pi(s)$
8: $\pi(s) \leftarrow \arg\max_a \sum_{s'} t(s,a,s') \left(r(s,a,s') + \gamma \hat{V}(s')\right)$
9: **if** $b \neq \pi(s)$ **then**
10: *stable* $\leftarrow$ *false*
11: **until** *stable* $=$ *true*

Dynamic Programming

- The order with which we update states does not matter, as long as we visit every state sufficiently often.
- The Policy Iteration algorithm is a type of Generalized Policy Iteration (GPI).
- Another example is **Value Iteration**, which replaces the policy evaluation step in Policy Iteration with a single backup of each state, making it more efficient.





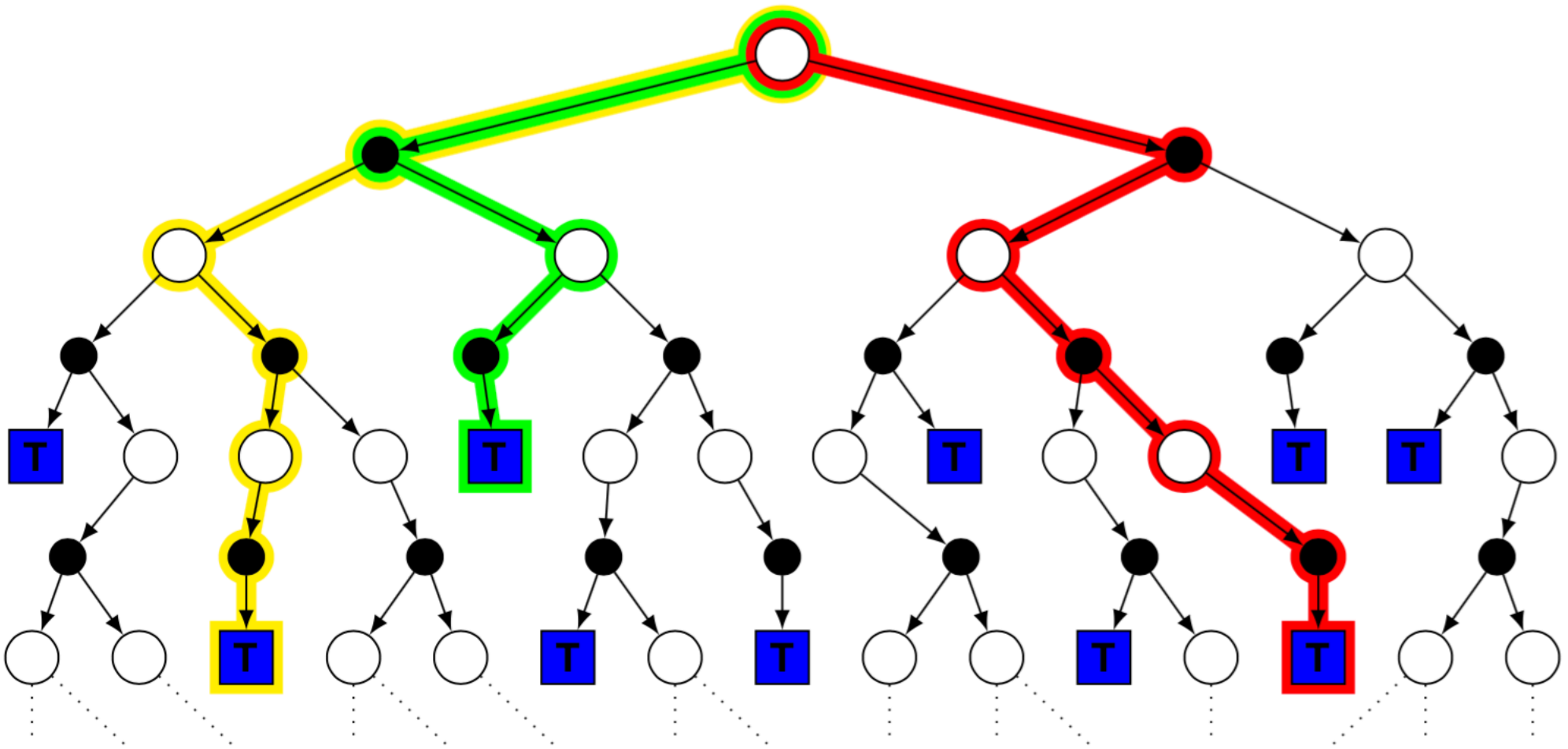
Sampling traces

Is there a better way of estimating $V^{\pi}(s)$?

We can use π to explore the MDP by interacting with it. If we sample a sufficient number of traces $\{\tau_i\}_1^N$ from state s, then

$$V^{\pi}(s) = \sum_{\text{all } \tau} p(\tau|\pi, s_0 = s)R(\tau) \approx \frac{1}{N}\sum_{i=1}^{N} R(\tau_i)$$

Monte Carlo exploration



First-visit Monte Carlo estimation algorithm

1: **procedure** MONTECARLOESTIMATION(π, N)
2: **Init**
3: $Returns(s) \leftarrow$ an empty list, for all $s \in S$.
4: **EndInit**
5: **for** N iterations **do**
6: Get trace, τ, using π.
7: **for all** s appearing in τ **do**
8: $R \leftarrow$ return from first appearance of s in τ.
9: Append R to $Returns(s)$
10: **for all** $s \in S$ **do**
11: $\hat{V}(s) \leftarrow$ average($Returns(s)$)
12: **return** $\hat{V}$

The game of blackjack

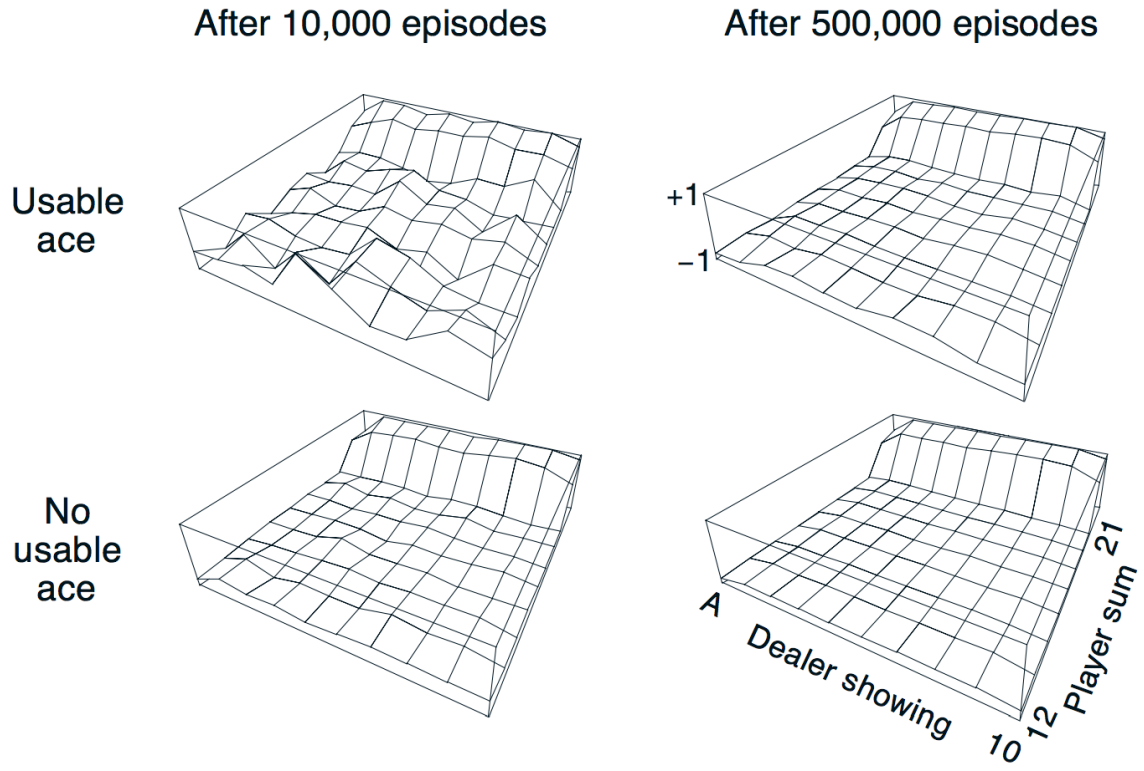
Rules:

- Play against the dealer.
- A hand's value is the sum of the cards' values, with J = Q = K = 10 and A = 1 or 11.
- You want to get closer to 21 than the dealer, but no higher or you bust.
- Start with 2 cards. Actions are hit (get another card) or stick (end your round).

Example of Monte Carlo estimation

Consider the game of blackjack. Is this an MDP?

We can use Monte Carlo estimation with a policy that only sticks on 20 or 21.

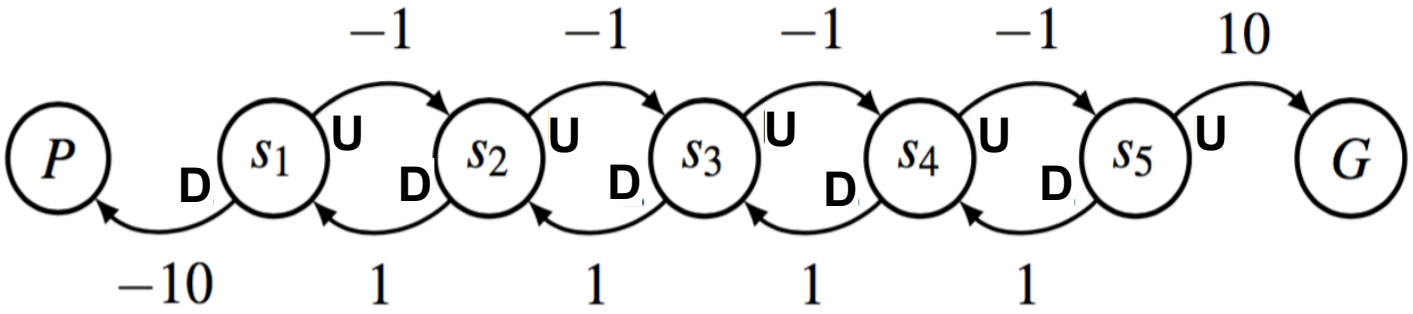


(Sutton and Barto)

First-visit Monte Carlo estimation on stair climbing MDP

MC estimation for our stair climbing MDP with $\pi(s, D) = \pi(s, U) = \frac{1}{2}$ and $\gamma = 0.9$.

First generate a trace, for instance: $\tau = s_3, U, -1, s_4, D, 1, s_3, U, -1, s_4, U, -1, s_5, U, 10, G$



$\hat{V}_0$:	0.0	0.0	0.0	0.0	0.0
$\hat{V}_1$:	0.0	0.0	4.92	6.58	10.

and after many traces…

$\hat{V}_\infty$:	−6.90	−3.10	0.0	3.10	6.90

Monte Carlo overview

- Methods that collect traces by exploration are collectively called Monte Carlo.
- There is no need to access the model of the MDP.
- Can be first state visit or every state visit.

How do we update our policy (choose better actions)?

Another way to choose actions

The Q function (or state-action value function) is the expected return given a state and action:

$$Q^{\pi}(s,a) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^{k} r_{t+1} \,|\pi, s_0 = s, a_0 = a\right)$$

Relating the functions V^{π} and Q^{π}:

$$V^{\pi}(s) = \sum_{a} \pi(s,a)\, Q^{\pi}(s,a)$$

$$Q^{\pi}(s,a) = \sum_{s'} t(s,a,s')(r(s,a,s') + \gamma\, V^{\pi}(s'))$$

Using the state-action value function

- The Q function can be used to choose the best action at a particular state.
- We can define the **greedy** policy as:

$$\pi_{\text{greedy}}(s) = \arg\max_a Q^{\pi}(s, a)$$

- We no longer need the model of the MDP, just the Q function.
- And we can learn estimates $\hat{Q}$ with Monte Carlo, in the same way as we learned $\hat{V}$.

First visit Monte Carlo Q estimation

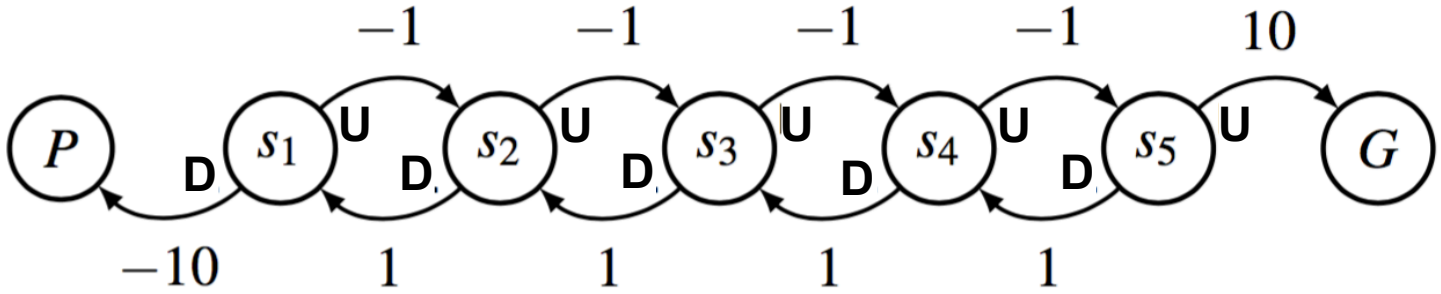
State-action value estimation is almost identical to state value estimation.

1: **procedure** MONTECARLOESTIMATION(π, N)
2: **Init**
3: $Returns(s, a) \leftarrow$ an empty list, for all $s \in S$ and $a \in A$.
4: **EndInit**
5: **for** N iterations **do**
6: Get trace, τ, using π.
7: **for all** (s,a) appearing in τ **do**
8: $R \leftarrow$ return from first appearance of (s, a) in τ.
9: Append R to $Returns(s, a)$
10: **for all** $s \in S$ and $a \in A$ **do**
11: $\hat{Q}(s, a) \leftarrow$ average($Returns(s, a)$)
12: **return** $\hat{Q}$

Monte Carlo Q estimation

On stair climbing MDP, with $\pi(s, D) = \pi(s, U) = \frac{1}{2}$, $\gamma = 0.9$, and first trace

$$\tau = s_3, U, -1, s_4, D, 1, s_3, U, -1, s_4, U, -1, s_5, U, 10, G$$



	(D,U)	(D,U)	(D,U)	(D,U)	(D,U)
$\hat{Q}_0$:	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
$\hat{Q}_1$:	$(0,0)$	$(0,0)$	$(0,4.92)$	$(6.58,8.)$	$(0,10.)$

and after many traces…

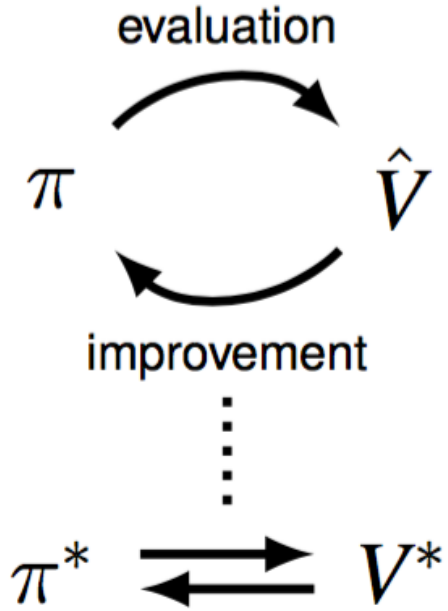
$\hat{Q}_\infty$:	(-10., -3.8)	(-5.2, -1.0)	(-1.8, 1.8)	(1.0, 5.2)	(3.8, 10.)

What is the greedy policy w.r.t. $\hat{Q}_\infty$?

What if we had started with policy $\pi(s) = D$ for all $s \in S$?

A return to Policy Iteration

- Policy Iteration (repeated estimation and improvement) can work for Q values, just as it works for V values.
- But if we use Monte Carlo estimation, we must be careful that we continue to explore all states and actions. Why?
- And why is this a problem if we are using greedy policies?



Exploration-exploitation trade-off

ε-soft policies

Consider a policy where at each state visit, we follow a deterministic policy with probability $(1-\varepsilon)$, $0 < \varepsilon \ll 1$, and act randomly with probability ε.

- For any ε-soft policy, π,

$$\pi(s,a) = \begin{cases} 1-\varepsilon+\dfrac{\varepsilon}{|A|} & \text{the dominant action, } a\text{, at } s \\ \dfrac{\varepsilon}{|A|} & \text{otherwise} \end{cases}$$

- All ε-soft policies are guaranteed to explore all states and actions.
- Dominant actions are taken far more often.

ε-greedy policies

The ε-greedy policy is the closest ε-soft policy to a greedy policy.

- For $\hat{Q}$, the ε-greedy policy, $\pi = \varepsilon\text{−greedy}(\hat{Q})$, is given by

$$\pi(s,a) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{|A|} & \text{for } a = \arg\max_{a'} \hat{Q}(s,a') \\ \dfrac{\varepsilon}{|A|} & \text{otherwise} \end{cases}$$

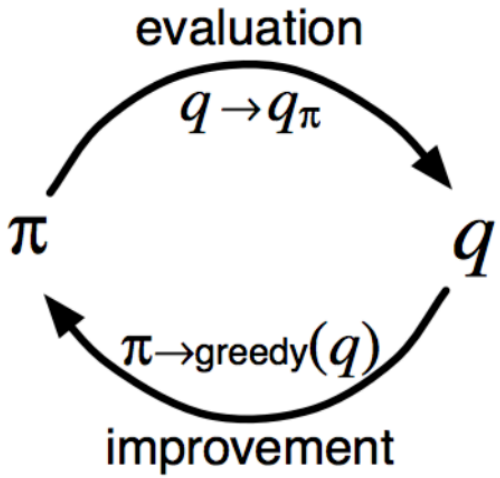
- We can use Policy Improvement with ε-greedy policies to get the best such policy $\tilde{\pi}^*$ (we write $\tilde{Q}^{\tilde{\pi}^*} = \tilde{Q}^*$).
- If ε is small, then the absolute best policy is $\pi^* = \text{greedy}(\tilde{Q}^*)$.

ε-soft Policy Iteration

Interleave Monte Carlo estimation, E, with periodic ε-greedy policy improvement, I_ε, to find the best ε-greedy policy, $\tilde{\pi}^*$.

$$\pi_0 \xrightarrow{E} Q^{\pi_0} \xrightarrow{I_\varepsilon} \pi_1 \xrightarrow{E} Q^{\pi_1} \xrightarrow{I_\varepsilon} \dots \xrightarrow{I_\varepsilon} \tilde{\pi}^* \xrightarrow{E} \tilde{Q}^* \xrightarrow{I} \pi^*$$

Methods that follow the policy they are estimating are known as **on-policy**.

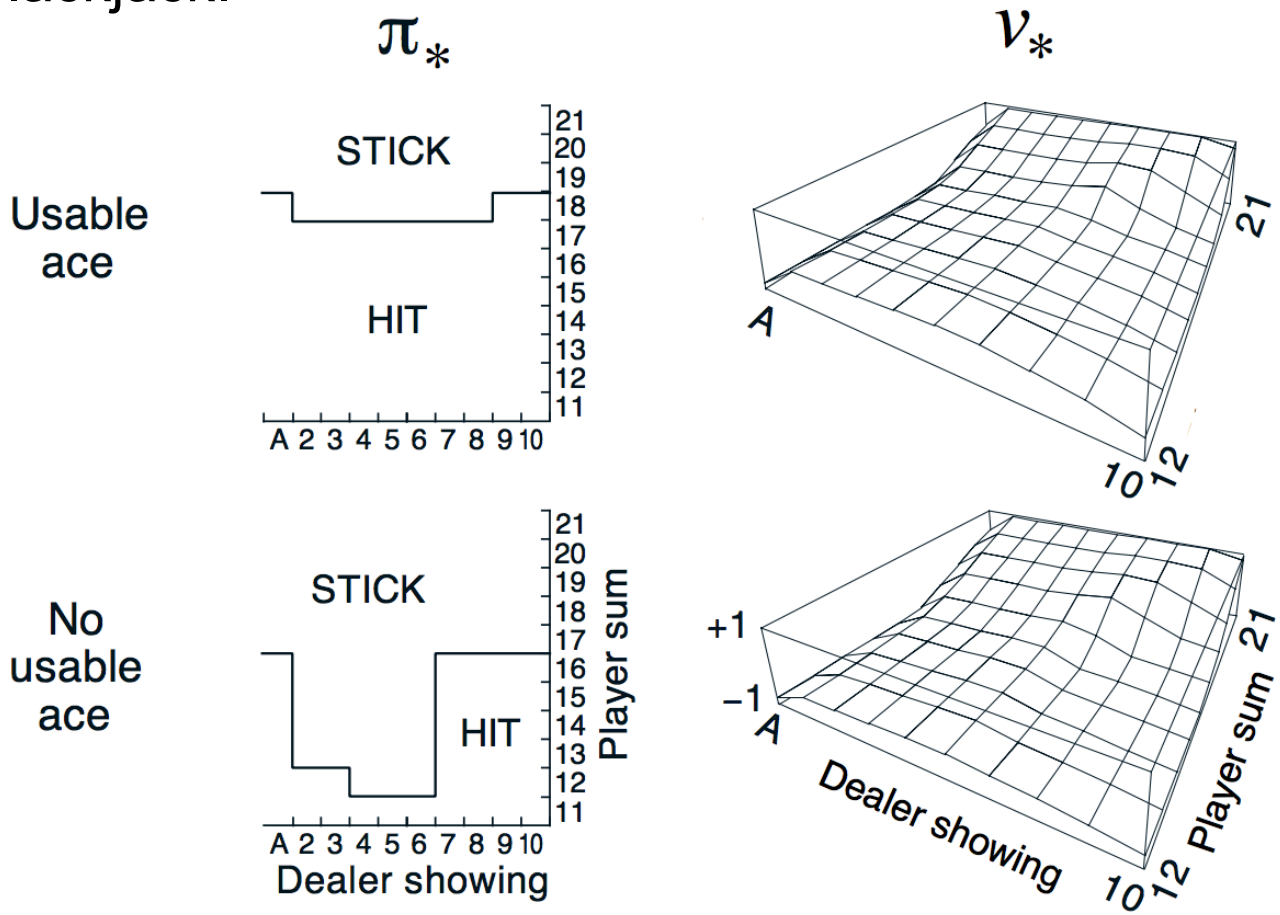


On-policy Monte Carlo batch optimization

1: **procedure** MONTECARLOBATCHOPTIMISATION(n, N)
2: **Init**
3: $\hat{Q}(s, a) \leftarrow$ arbitrary value, for all $s \in S$, $a \in A$.
4: $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
5: **EndInit**
6: **for** n batches **do**
7: $\hat{Q} \leftarrow$ MONTECARLOESTIMATION(π, N)
8: $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
9: **return** *greedy*$(\hat{Q})$

Example of Monte Carlo optimization

We can use Monte Carlo optimization to find the optimal policy in the game of blackjack.



(Sutton and Barto)

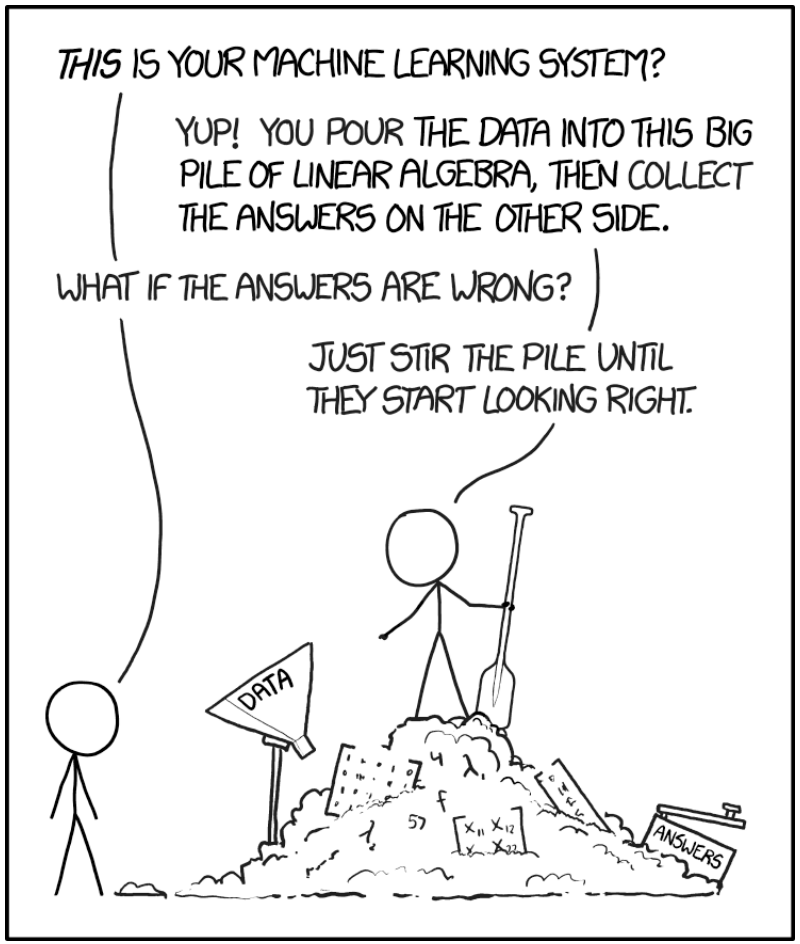
Summary

- Markov Decision Processes (MDPs) describe a class of control problems.
- The state value function (V) predicts the future return from a given state.
- The state-action value function (Q) predicts the future return from a given state and action.
- Policy Evaluation uses Bellman updates to estimate the value function.
- Policy Improvement (greedily) chooses a new policy based on the value function.
- Policy Iteration combines these steps to converge on the optimal policy (an example of GPI).

Summary

- Monte Carlo estimation can also be used for policy evaluation.
- Monte Carlo estimation learns $\hat{Q}$ (or $\hat{V}$) from experience.
- The algorithm must continually explore all states and actions.
- ε-soft policies can be used for on-policy learning.
- Note: we can also perform off-policy learning, meaning we use one fixed policy to explore the environment, but optimize a different policy (more on this later).

5-minute break



(xkcd)

Temporal Difference Learning

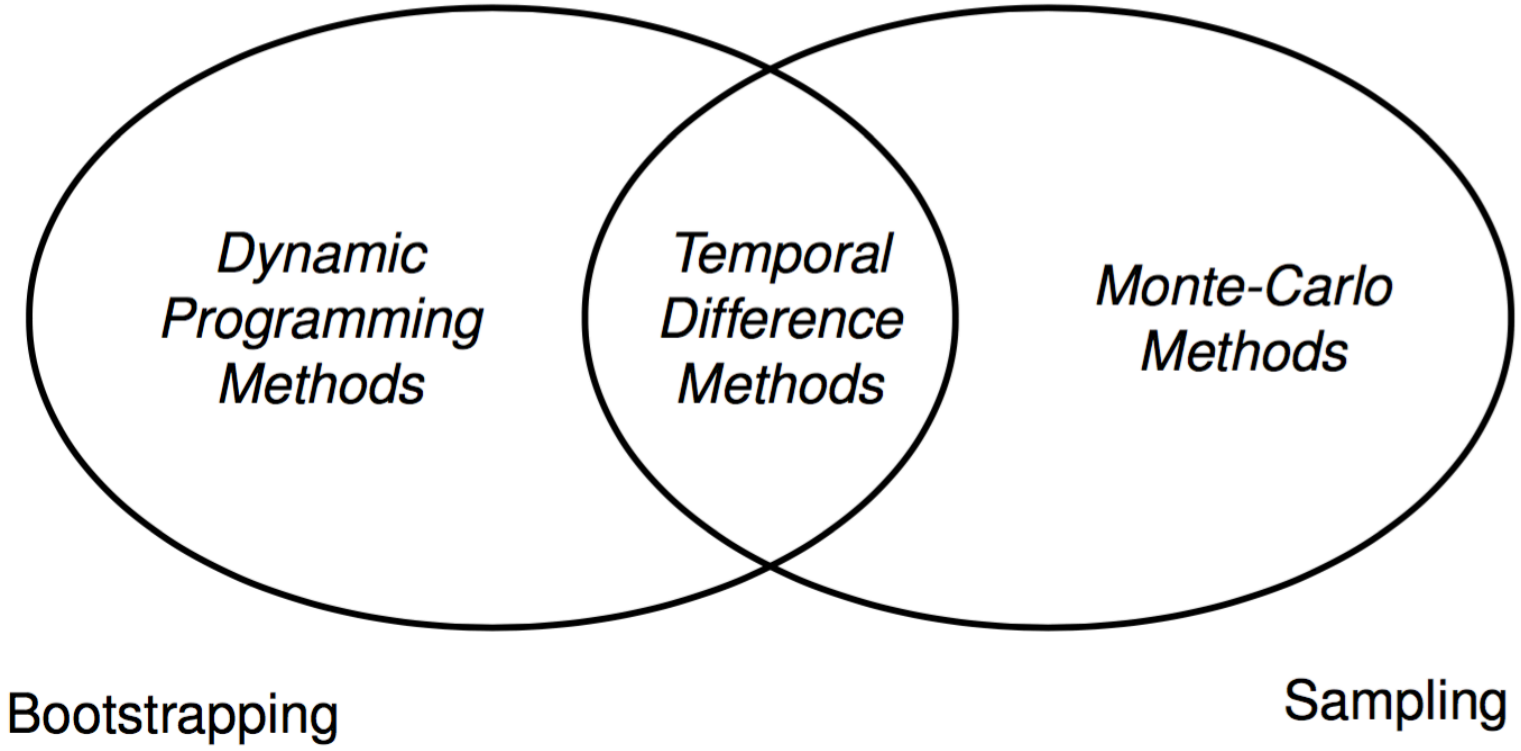
The best of both worlds

- Both Dynamic Programming and Monte Carlo approaches have desirable properties:
  - **Bootstrapping**: Dynamic Programming updates value estimates based on other value estimates (efficient use of data).
  - **Sampling**: Monte Carlo methods sample states and rewards directly from the environment (no need to know the model of the MDP).

Can we have both?

The best of both worlds

Temporal difference (TD) learning sits between Dynamic Programming and Monte Carlo methods.



Temporal Difference learning

The iterative Monte Carlo V^{π} update can be written as

$$\hat{V}'(s_t) \leftarrow \hat{V}(s_t) + \alpha(\underbrace{R_t}_{\text{sample}} - \underbrace{\hat{V}(s_t)}_{\text{running estimate}})$$

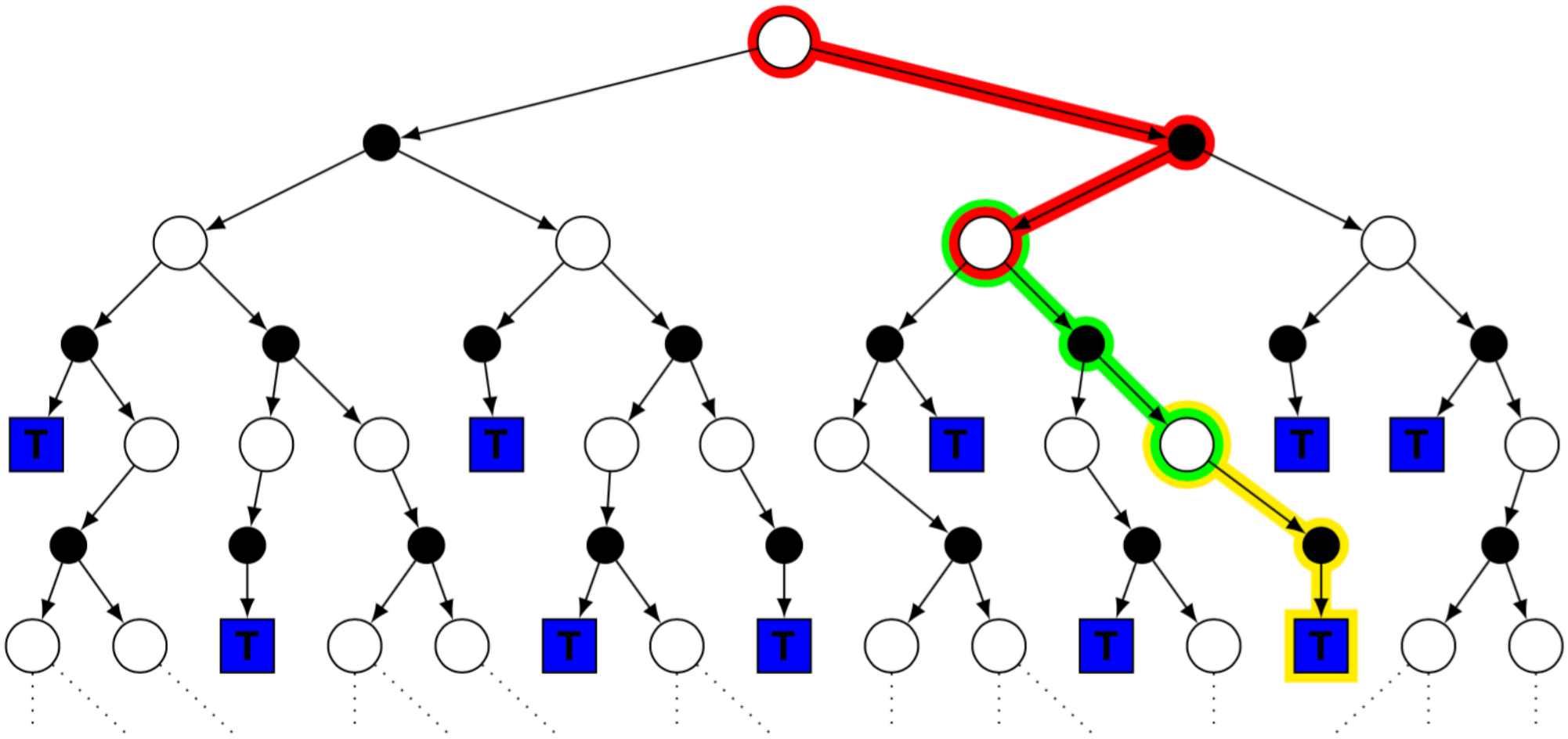
Temporal Difference methods can perform a similar update, but after every time step (as opposed to every sampled trace), i.e.,

$$\hat{V}'(s_t) \leftarrow \hat{V}(s_t) + \alpha(\underbrace{r_{t+1} + \gamma\hat{V}(s_{t+1})}_{\text{sample}} - \underbrace{\hat{V}(s_t)}_{\text{running estimate}})$$

This is possible because

$$V^{\pi}(s) = \mathbb{E}(R_t|s_t = s, \pi) = \mathbb{E}\left(\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s, \pi\right) = \mathbb{E}(r_{t+1} + \gamma V^{\pi}(s_{t+1})|s_t = s, \pi)$$

Temporal Difference estimates



TD estimation

1: **procedure** TD-ESTIMATION(π)
2: **Init**
3: $\hat{V}(s) \leftarrow$ arbitrary value, for all $s \in S$.
4: **EndInit**
5: **repeat**(For each episode)
6: Initialise s
7: **repeat**(For each step of episode)
8: a action chosen from π at s
9: Take action a; observe r, and next state, s'
10: $\delta \leftarrow r + \gamma \hat{V}(s') - \hat{V}(s)$ → δ is called the temporal difference error
11: $\hat{V}(s) \leftarrow \hat{V}(s) + \alpha\delta$
12: $s \leftarrow s'$
13: **until** s is absorbing state
14: **until** Done

Sarsa learning

Just as we did with Monte Carlo learning, we can use TD for on-policy optimization.

- Use TD updates to learn state-action values with

temporal difference error

$$\hat{Q}'(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha\big(r_{t+1} + \gamma \hat{Q}(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t)\big)$$

sample running estimate

- Continually update π to be ε-greedy($\hat{Q}$)
- Repeated runs from start to an absorbing state (episodes)
- This algorithm is called **Sarsa** because it uses s_t, a_t, r_{t+1}, s_{t+1} and a_{t+1} values

Sarsa: on-policy TD learning

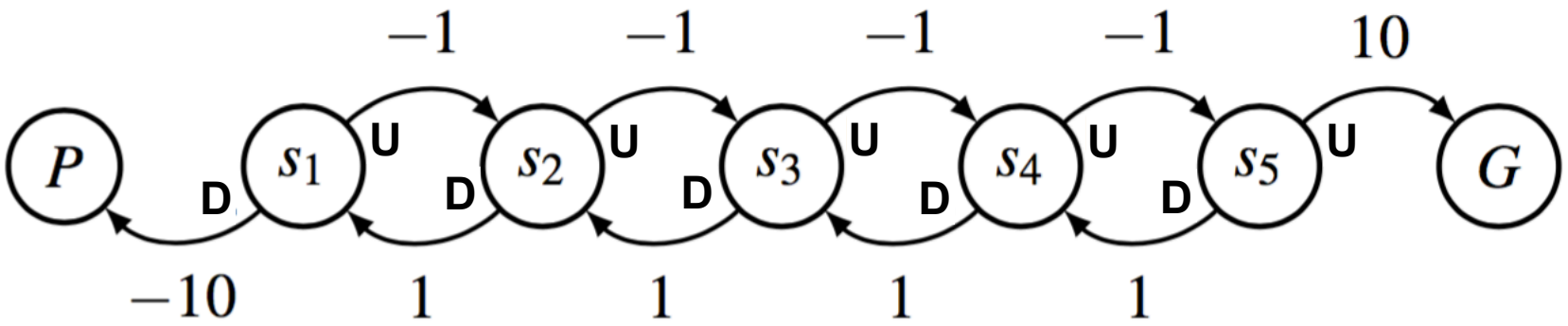
1: **procedure** SARSA
2: **Init**
3: $\hat{Q}(s,a) \leftarrow$ arbitrary value, for all $s \in S$, $a \in A$.
4: $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
5: **EndInit**
6: **repeat**(For each episode)
7: Initialise s
8: a action chosen from π at s
9: **repeat**(For each step of episode)
10: Take action a; observe r, and next state, s'
11: a' action chosen from π at s'
12: $\delta \leftarrow r + \gamma \hat{Q}(s',a') - \hat{Q}(s,a)$
13: $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha\delta$
14: $s \leftarrow s'$; $a \leftarrow a'$
15: $\pi \leftarrow \varepsilon$-*greedy*$(\hat{Q})$
16: **until** s is absorbing state
17: **until** Done

Sarsa on the stair climbing MDP

Consider Sarsa on our stair climbing MDP, with trace

$$\tau = s_3, U, -1, s_4, D, 1, s_3, U, -1, s_4, U, -1, s_5, U, 10, G$$

- What state-action value updates occur and at what points?
- What policy updates occur and at what points?

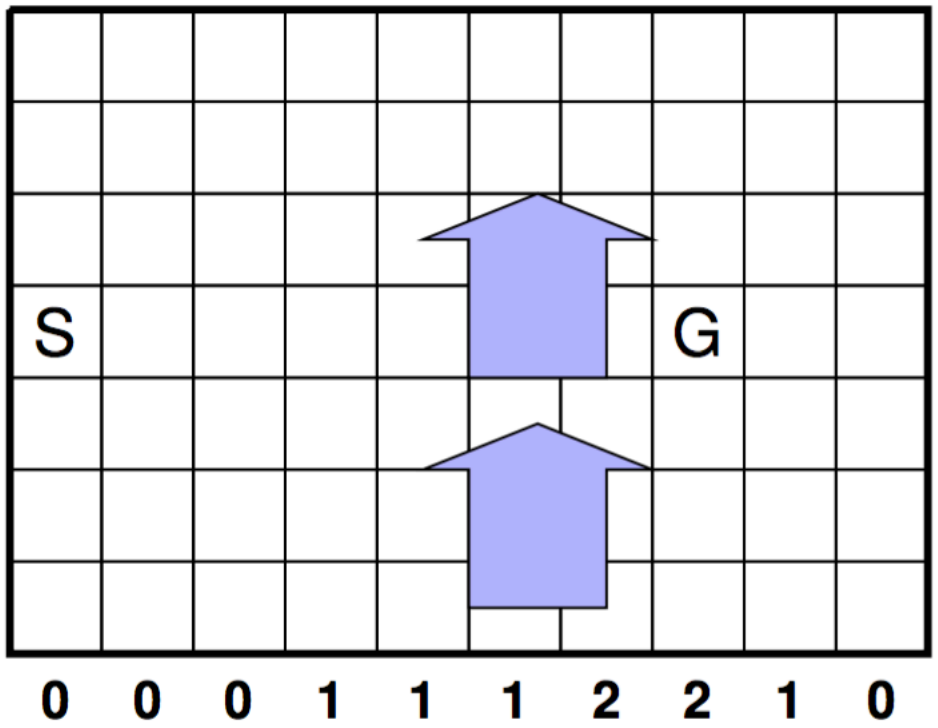


Another example: the windy grid world

Windy grid world:

- Actions are R, U, L and D.
- Action effects are influenced by upwards blowing wind.
- Start state is S; G is an absorbing state.
- Rewards are -1 for every step.

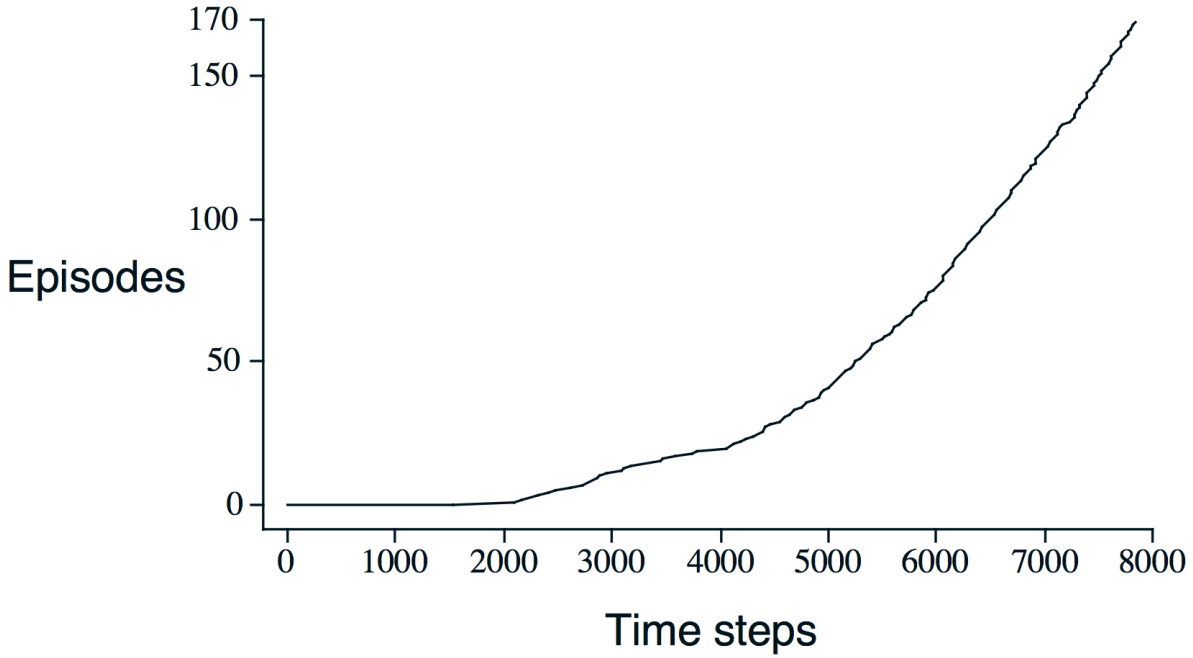
Why would Monte Carlo learning perform badly in this environment?

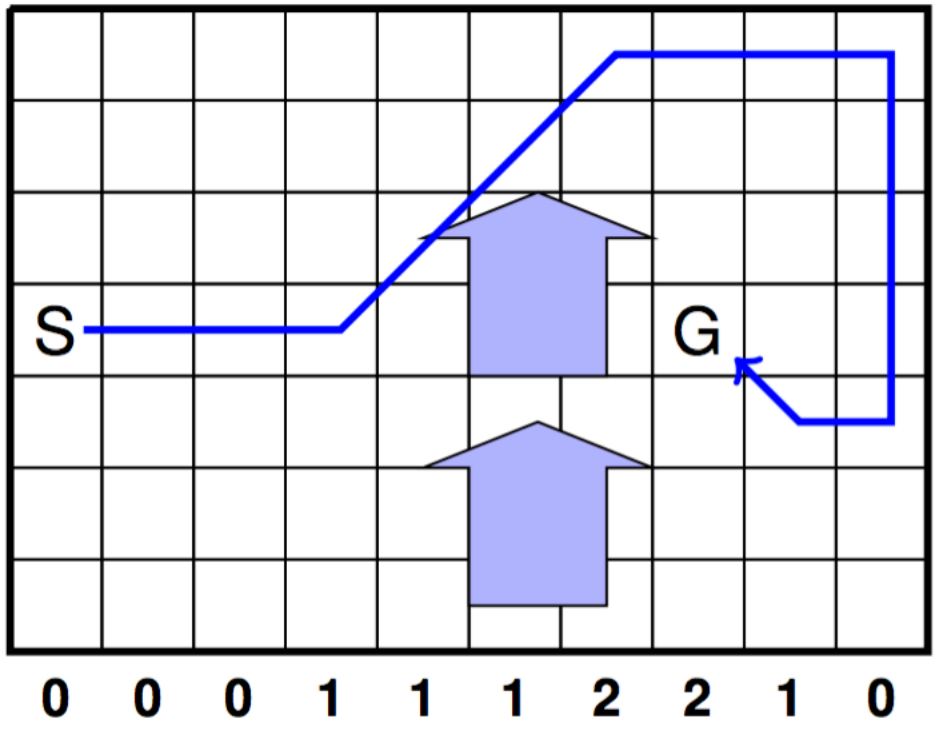


(Sutton and Barto)

Another example: the windy grid world

Sarsa with $\gamma = 0.9$, $\varepsilon = 0.1$ and $\alpha = 0.1$:





(Sutton and Barto)

Advantages of Sarsa

Sarsa has some advantages over Monte Carlo learning:

- Sarsa improves within an episode, while Monte Carlo must wait until the end.
- TD methods accelerate learning by using bootstrapping.
- TD methods can be much lighter weight (they do not need to store whole traces, or lists of returns).

Off-policy TD learning

An alternative to Sarsa is called **off-policy** TD learning.

- The controlling policy ensures exploration.
- The state-action value estimates keep track of greedy estimates using the update

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t))$$

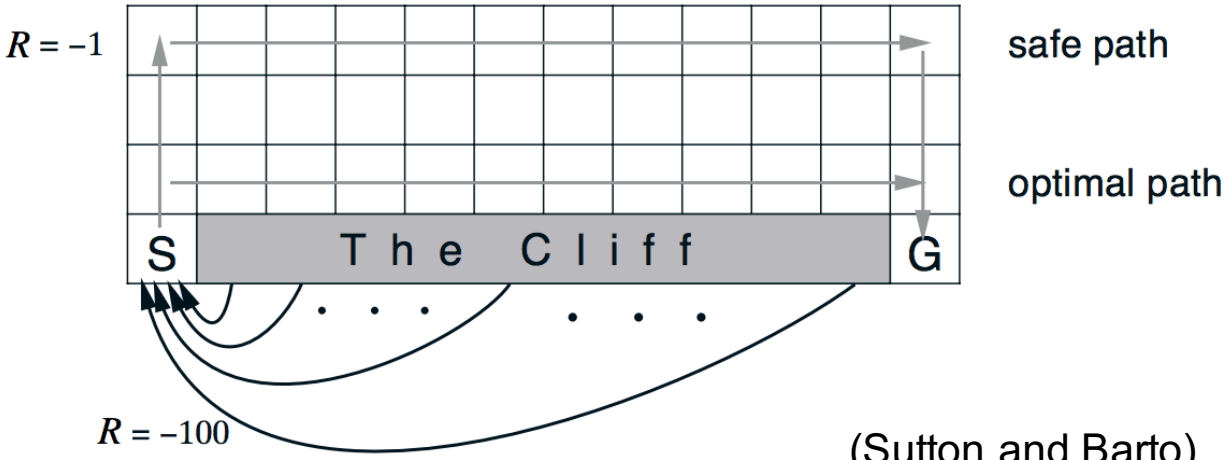
Why is this considered an off-policy method?

Q-learning: off-policy TD learning

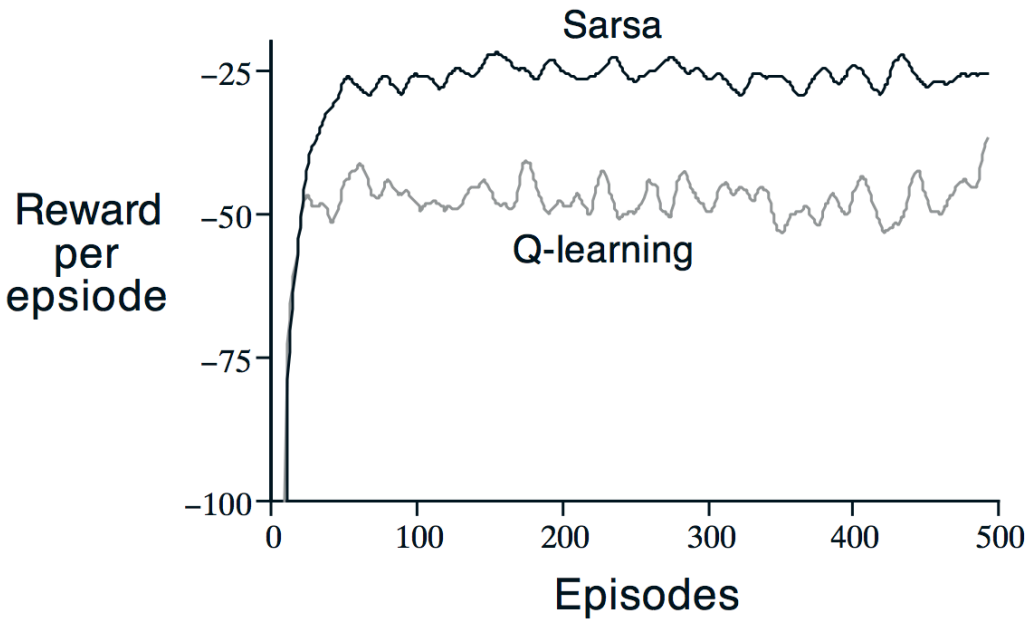
procedure Q-LEARNING
  Init
    $\hat{Q}(s,a) \leftarrow$ arbitrary value, for all $s \in S$, $a \in A$.
    $\tilde{\pi} \leftarrow \varepsilon$-*greedy*$(\hat{Q})$ ▷ the control policy
  EndInit
  repeat(For each episode)
    Initialise s
    repeat(For each step of episode)
      a action chosen from $\tilde{\pi}$ at s
      Take action a; observe r, and next state, s'
      $\delta \leftarrow r + \gamma \max_{a^*} \hat{Q}(s', a^*) - \hat{Q}(s,a)$
      $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha\delta$
      $\tilde{\pi} \leftarrow \varepsilon$-*greedy*$(\hat{Q})$ ▷ update control policy
      $s \leftarrow s'$
    until s is absorbing state
  until Done
  $\pi \leftarrow$ *greedy*$(\hat{Q})$ ▷ output optimal policy
  return π

Comparing Sarsa and Q-learning

Grid world with deterministic action effects. $r = -1$ except when falling off the cliff, in which case $r = -100$.



(Sutton and Barto)



Q-learning online performance is worse than Sarsa.

Sarsa learns the safe path, while Q-learning learns the optimal path.

Why?

Softmax policy

As an alternative to the ε-greedy policy, we can use a probabilistic policy such as **softmax**.

- For $\hat{Q}$, the softmax policy, $\pi = \mathrm{softmax}(\hat{Q})$, is given by

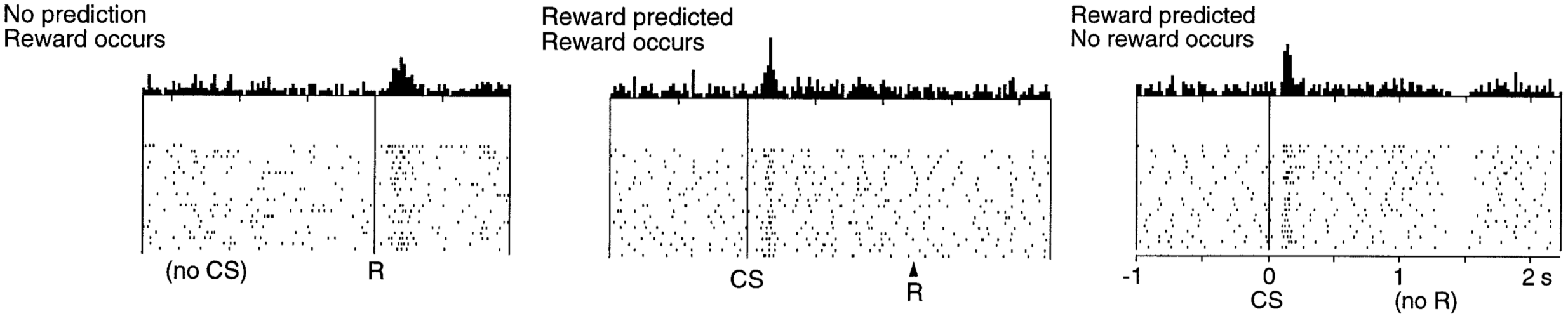
$$\pi(s,a) = P(a_t|s_t) = \frac{e^{Q(s_t,a_t)/\tau}}{\sum_{i=1}^{n} e^{Q(s_t,i)/\tau}}$$

- τ is called the **temperature** parameter. With high temperatures, all actions have nearly the same probability; with lower temperatures, the expected rewards will increasingly affect the action probabilities.
- In practice, we can also use the inverse temperature, $\beta = \frac{1}{\tau}$.

Evidence for TD error in the macaque brain

- There is experimental evidence that dopaminergic neurons report rewards according to a reward prediction error signal (TD error)

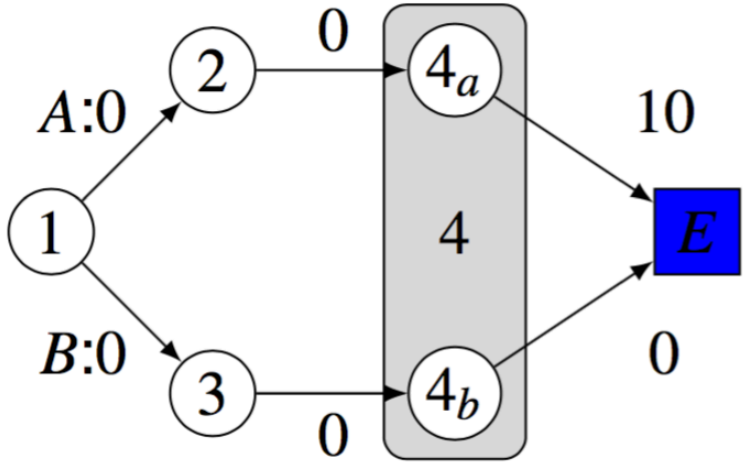
$$\hat{V}'(s_t) \leftarrow \hat{V}(s_t) + \alpha(\underbrace{R_t}_{\text{sample}} - \underbrace{\hat{V}(s_t)}_{\text{running estimate}})$$



(Schultz et al., 1997)

Problems with hidden states

What method would you use to estimate the value function for the following **hidden-state** process? In states 4_a and 4_b the agent observes 4.



The Markov assumption fails:

$$p(s_{t+1}, r_{t+1} | s_t, a_t) \neq p(s_{t+1}, r_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0)$$

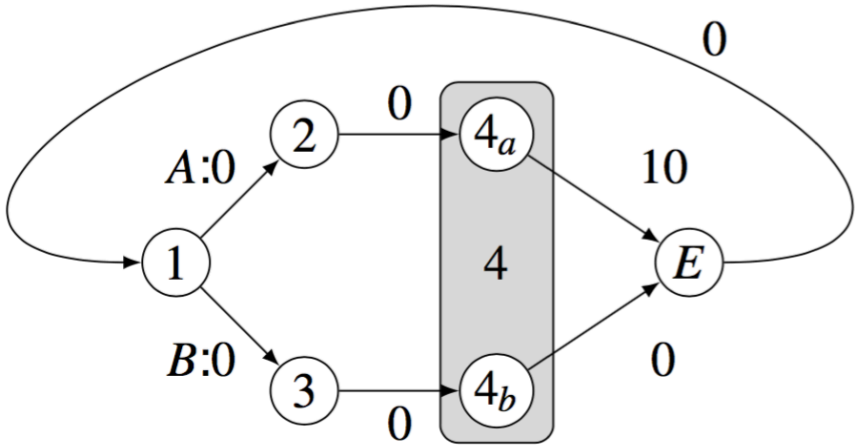
Note that TD updates for states 2 and 3 have the same fixed point:

$$\hat{V}(2) \leftarrow \hat{V}(2) + \alpha\left(0 + \gamma\hat{V}(4) - \hat{V}(2)\right)$$
$$\hat{V}(3) \leftarrow \hat{V}(3) + \alpha(0 + \gamma\hat{V}(4) - \hat{V}(3))$$

Monte Carlo doesn't have this problem.

Problems with hidden states

What about the following process?



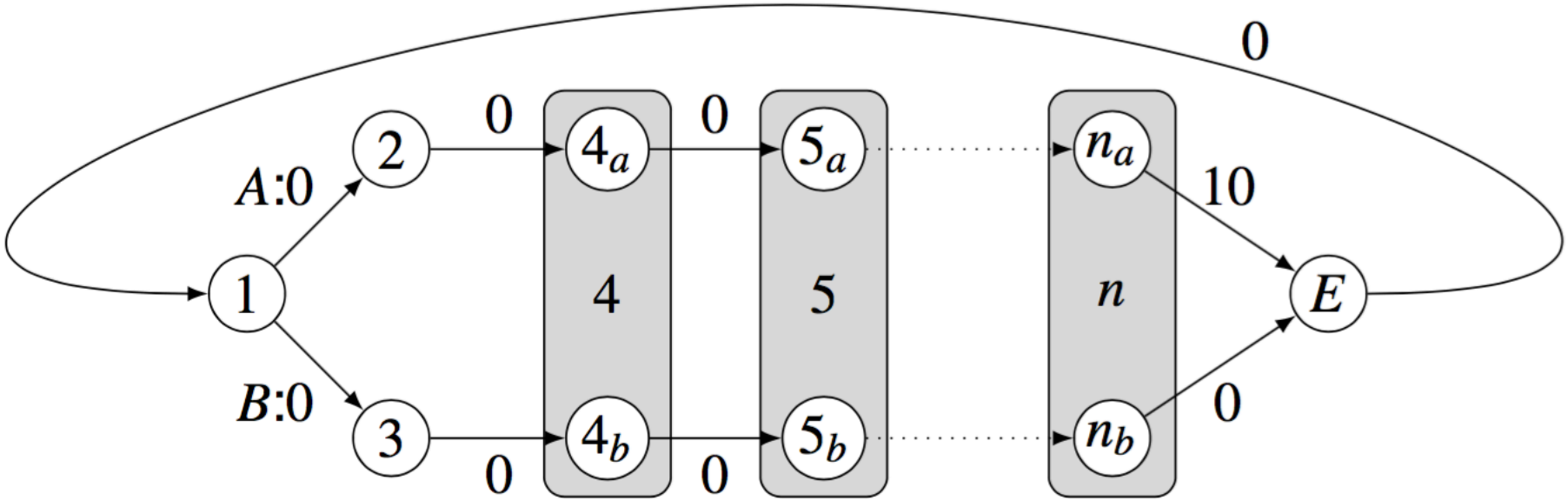
In this case, Monte Carlo learning will not work, because the process is not episodic.

What about a two-step TD-like update?

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha \underbrace{\left(r_{t+1} + \gamma r_{t+2} + \gamma^2 \hat{V}(s_{t+2}) - \hat{V}(s_t)\right)}_{\text{TD error}}$$

Problems with hidden states

We may still have problems though…



n-step updates

What we can do for 2 steps, we can do for n steps…

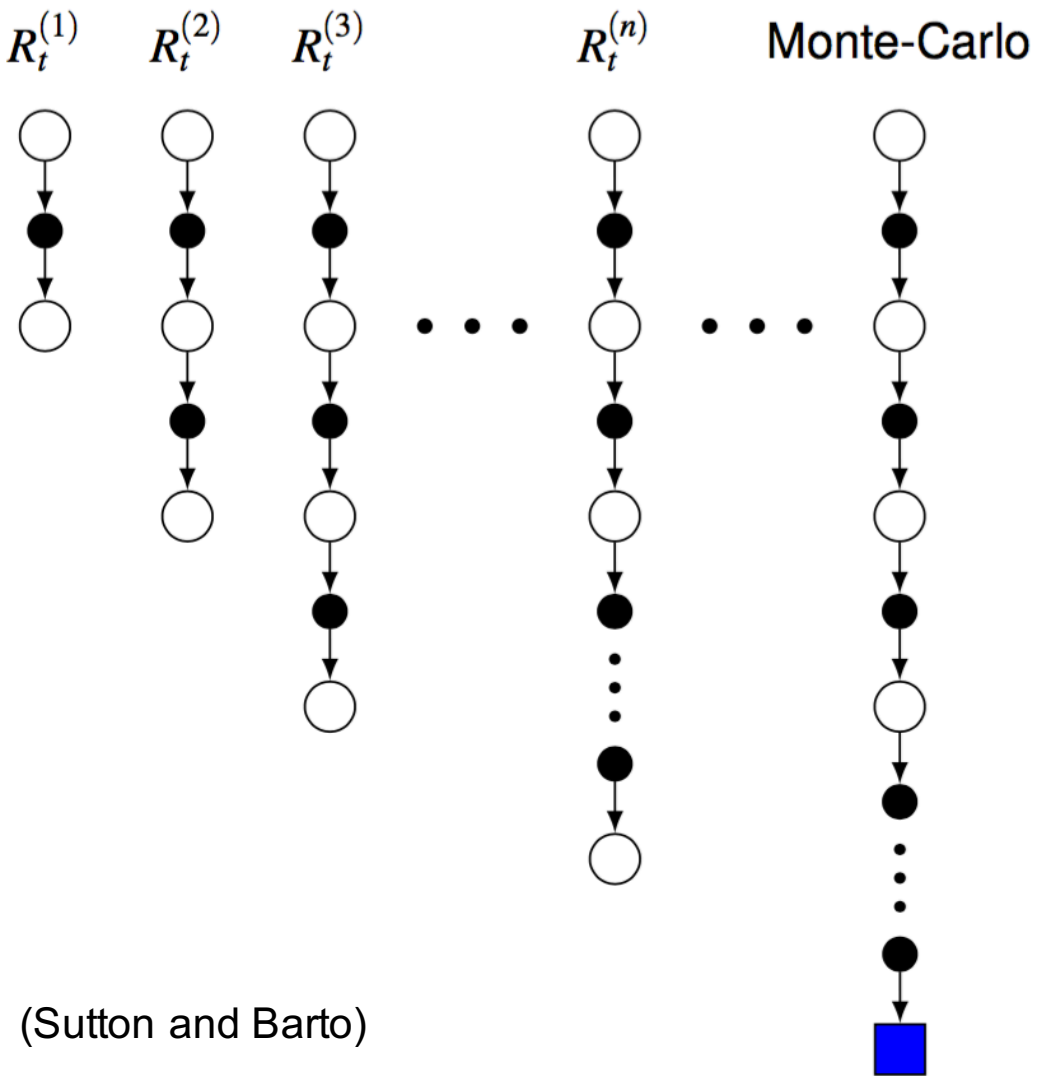
$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha\left(\underbrace{R_t^{(n)}}_{\text{sample}} - \underbrace{\hat{V}(s_t)}_{\text{running estimate}}\right)$$

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{V}(s_{t+n})$$

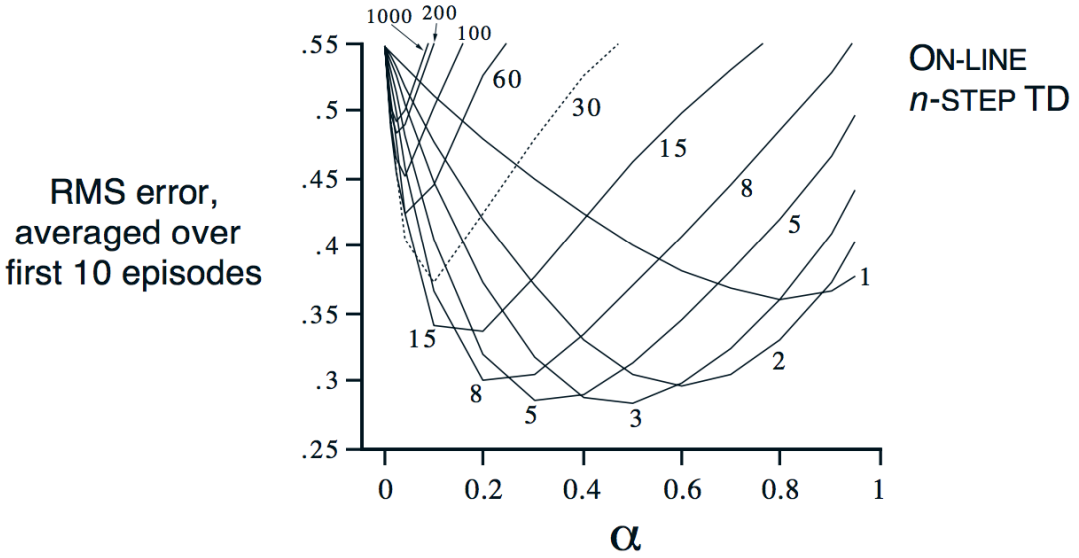
What value of n should we use?

- $R_t^{(n)}$ is the **truncated return**
- A larger n can be better at avoiding hidden state problems…
- ...but we must wait longer before we update.

A graphical view of truncated returns



(Sutton and Barto)



Average RMS error over the first 10 episodes for n-step TD estimation on a 19-state random walk task (no hidden states) with different values of n.

Linear combinations of truncated returns

With the Markov assumption:

- Each truncated return is a sample from V^π

$$\mathbb{E}\left(R_t^{(n)} \middle| s_t = s\right) = V^\pi(s)$$

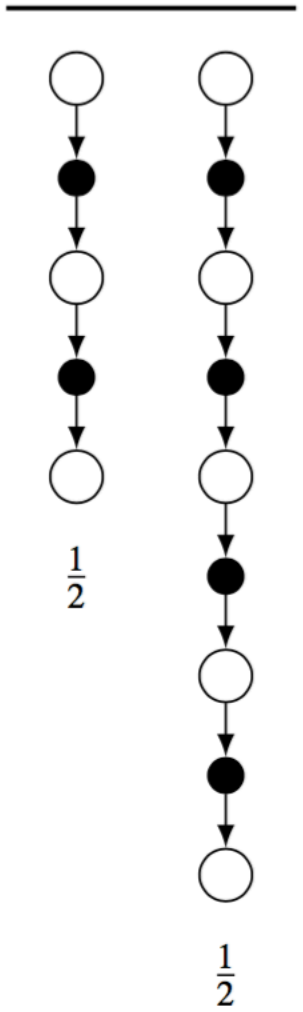
- For instance, we can average over 2-step and 4-step returns:

$$\mathbb{E}\left(\frac{R_t^{(2)} + R_t^{(4)}}{2} \middle| s_t = s\right) = \frac{V^\pi(s) \ + V^\pi(s)}{2} = V^\pi(s)$$

- Any normalized linear combination will do:

$$\mathbb{E}\left(\frac{\sum_i w_i R_t^{(n_i)}}{\sum_i w_i} \middle| s_t = s\right) = \frac{\sum_i w_i V^\pi(s)}{\sum_i w_i} = V^\pi(s)$$

- Even infinite sums are possible.



TD(λ): forward view

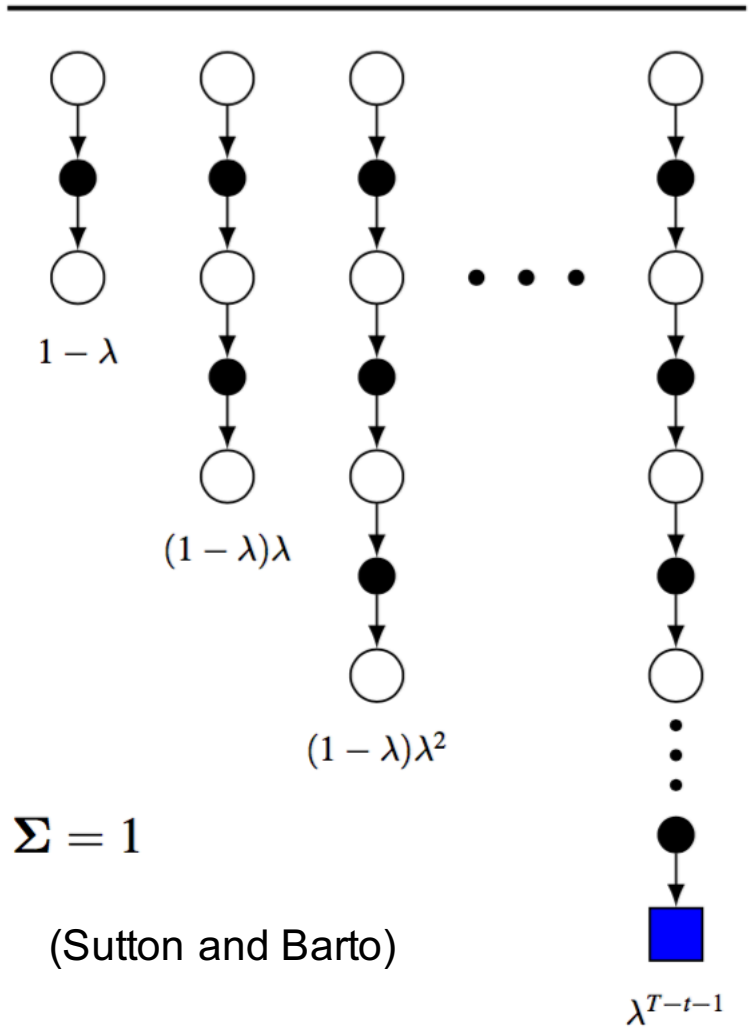
- For $\lambda \in [0,1)$, $\sum_{n=0}^{\infty} \lambda^n = \frac{1}{1-\lambda}$
- Define the λ-return (normalized sum) as

$$R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

- If the episode terminates at time T,

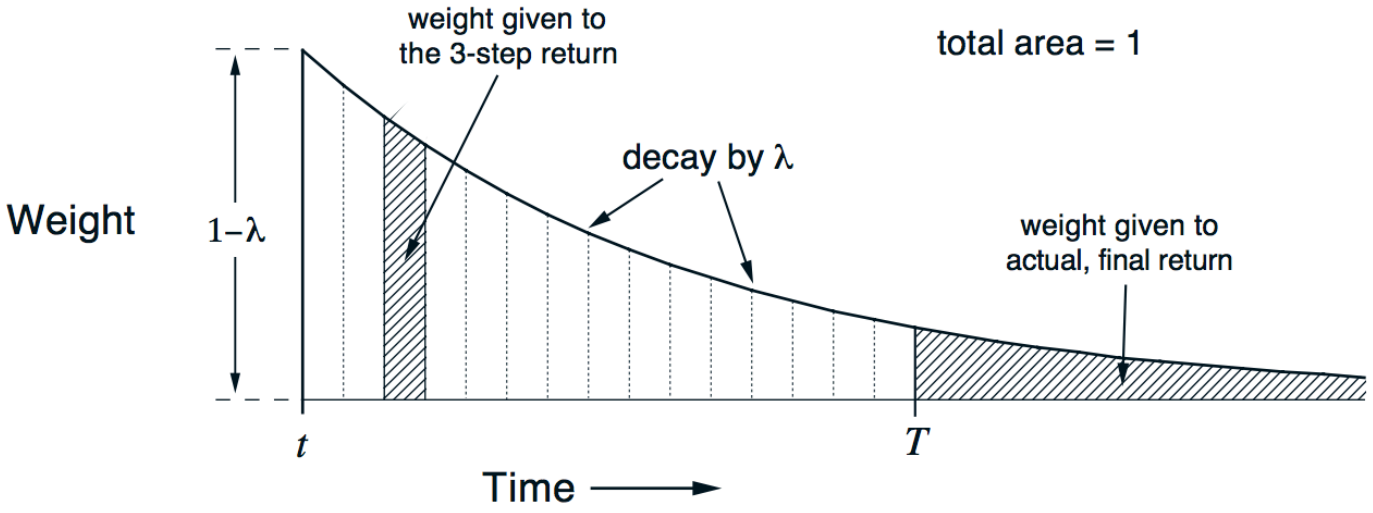
$$R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

- TD(0) is standard TD (1-step updates).
- TD(1) is equivalent to Monte Carlo.

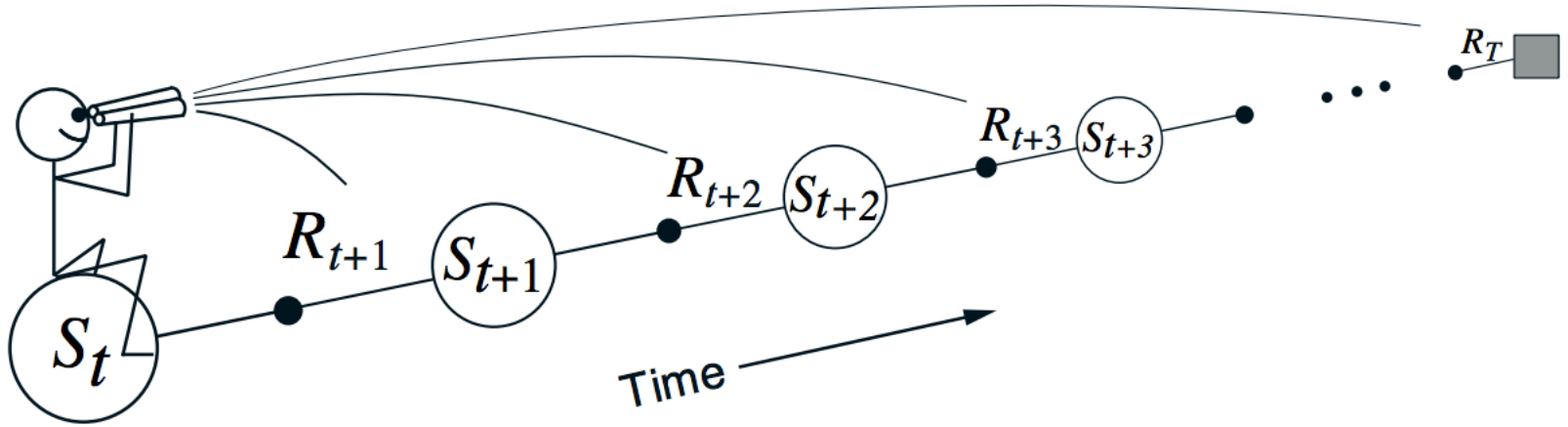


(Sutton and Barto)

TD(λ): forward view



Weighting given by λ-return to each n-step return (from Sutton and Barto).



The forward (theoretical) view of TD(λ) (from Sutton and Barto). We know how we want to update $V^{\pi}(s_t)$, but when can we apply the update?

Rewriting λ-returns using the TD error

- At each time t, we begin collecting a new update

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha\left(R_t^{\lambda} - \hat{V}(s_t)\right) = \hat{V}(s_t) + \alpha \sum_{n=t}^{\infty} (\gamma\lambda)^{n-t}\delta_n$$

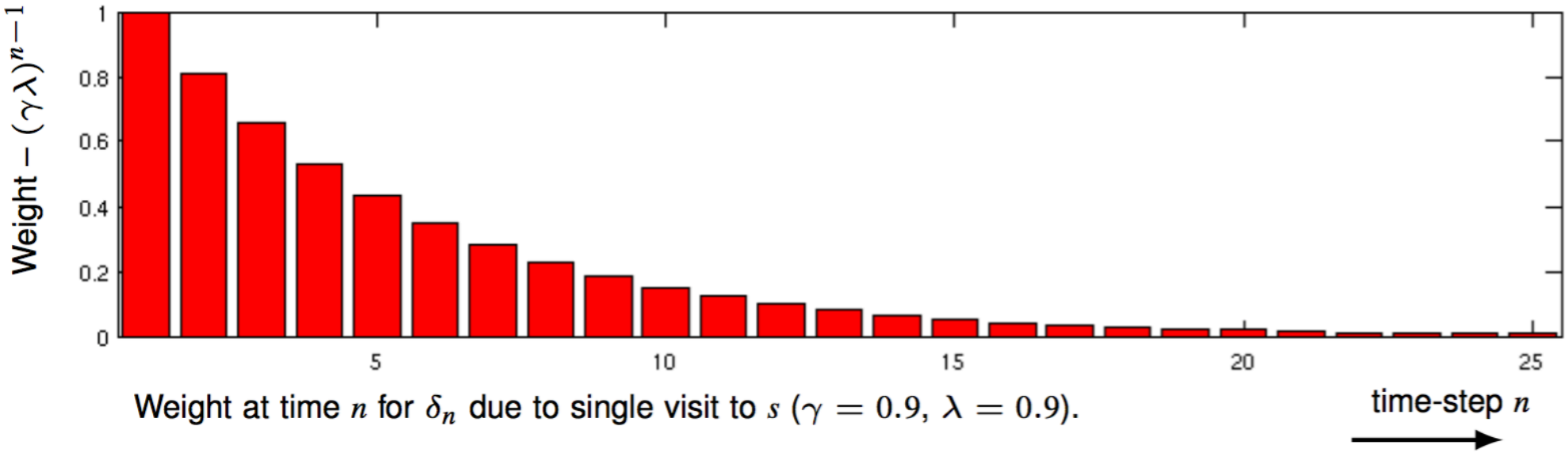
- The important thing is that we have rewritten the update in terms of single-step TD errors:

$$\delta_n = r_{n+1} + \gamma\hat{V}(s_{n+1}) - \hat{V}(s_t)$$

- At each time n we can apply a bit more of this update.

The weights to the TD error update

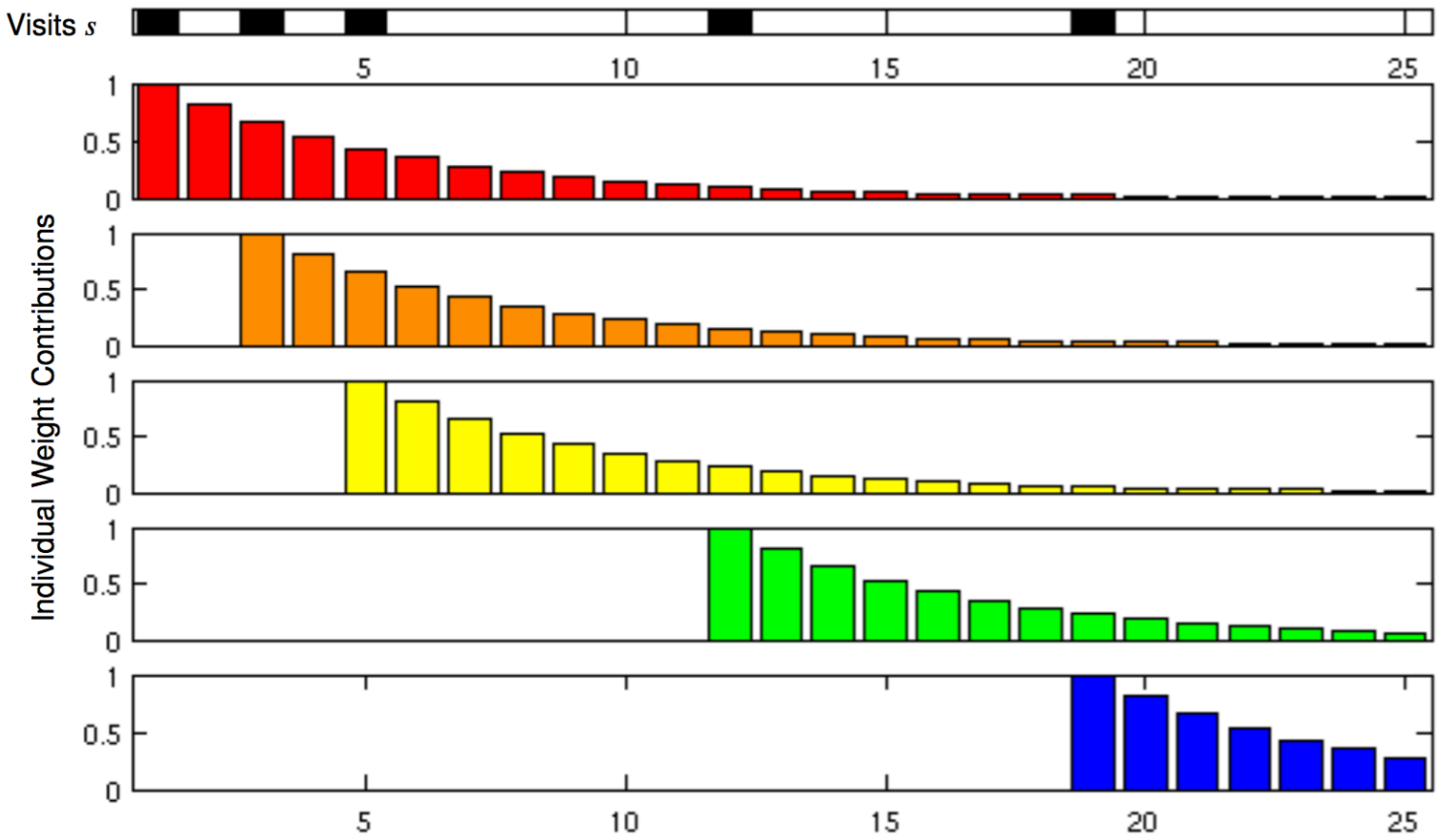
If we only visit state s at time $t = 1$, then the weights $(\gamma\lambda)^{n-t}$ we apply to δ_n for our update at each time n decay like this:



Weight at time n for δ_n due to single visit to s ($\gamma = 0.9$, $\lambda = 0.9$).

Multiple visits to the same state

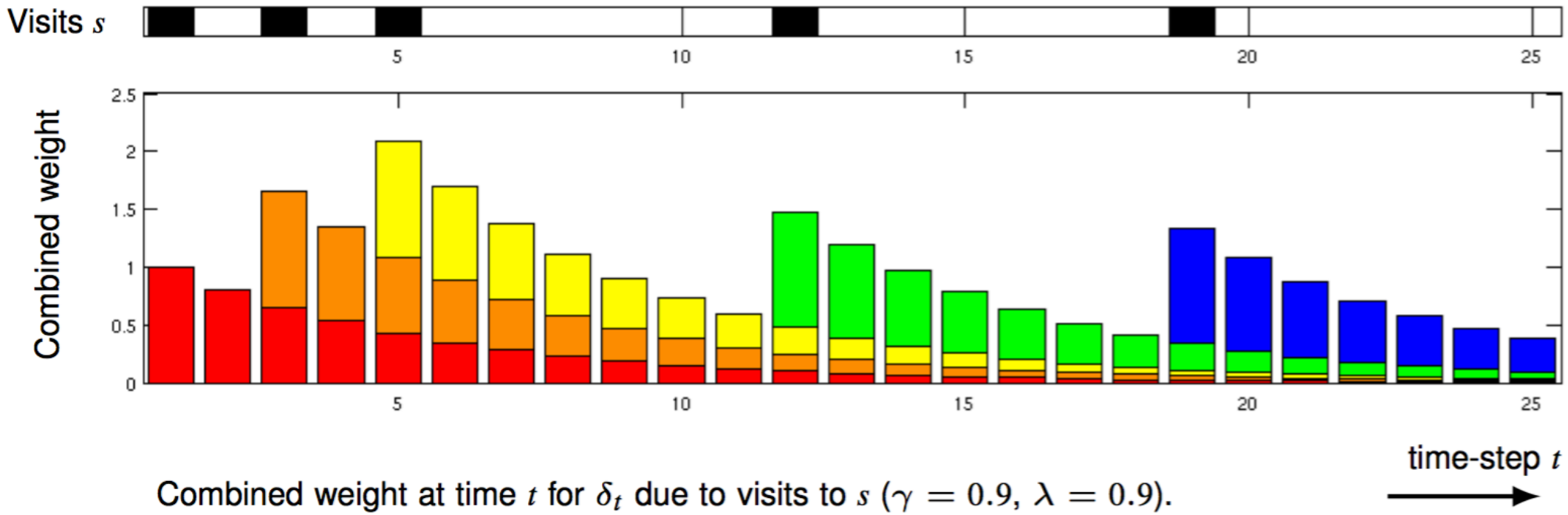
If we visit s at time $t = 1, 3, 5, 12$ and 19:



Individual weight contributions at time t for δ_t due to visits to s ($\gamma = 0.9$, $\lambda = 0.9$).

Combining the TD error update

If we visit s at time $t = 1$, 3, 5, 12 and 19:



Combined weight at time t for δ_t due to visits to s ($\gamma = 0.9$, $\lambda = 0.9$).

Eligibility traces

Eligibility traces keep track of the states we have visited, or rather, they keep track of the weights with which we need to update them.

- Initialize every $\hat{z}_0(s) = 0$ and at t we update for each s

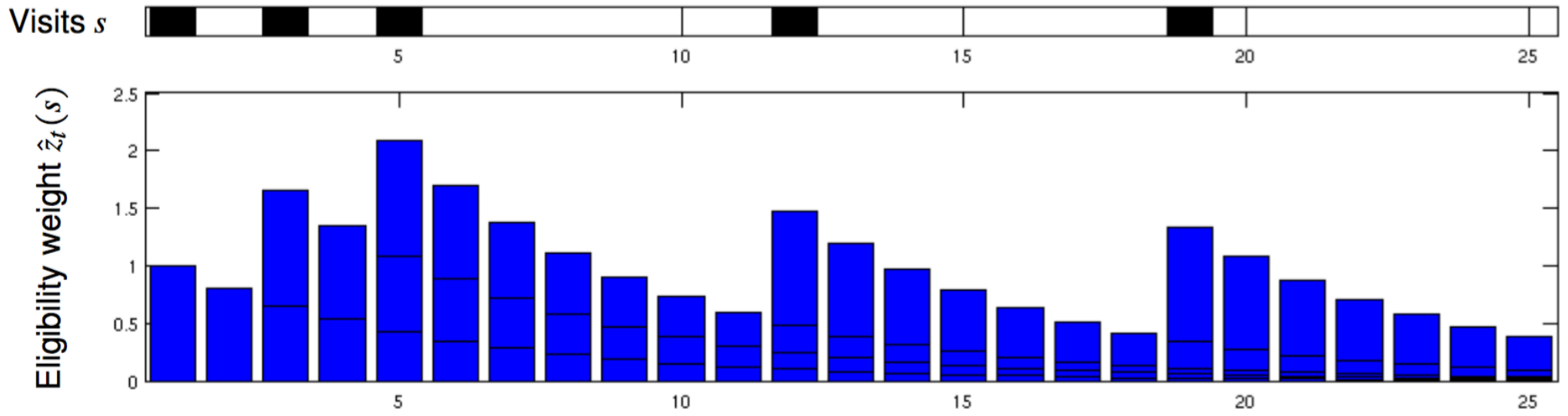
$$\hat{z}_0(s) = \begin{cases} \gamma\lambda\hat{z}_{t-1}(s) + 1 & \text{If } s_t = s \\ \gamma\lambda\hat{z}_{t-1}(s) & \text{otherwise} \end{cases}$$

- At each time t, we update every state s with

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha\delta_t\hat{z}_t(s)$$

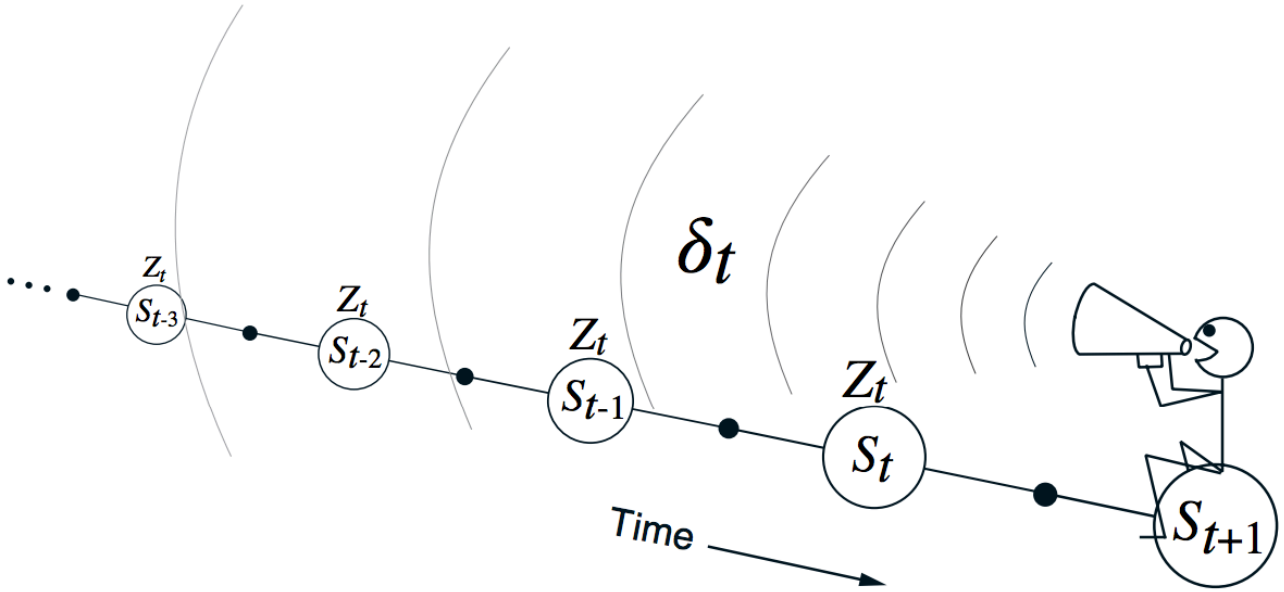
Weights of the eligibility trace

If we visit s at time $t = 1, 3, 5, 12$ and 19:

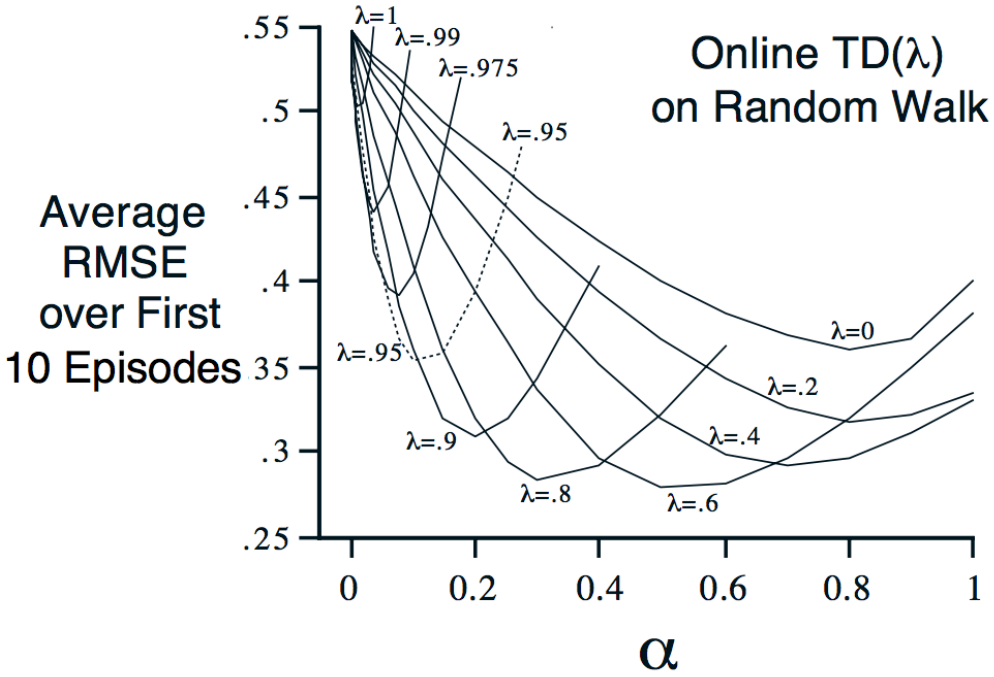


Eligibility weights for $\hat{z}_t(s)$ for various times due to repeated visits to s ($\gamma = 0.9$, $\lambda = 0.9$).

TD(λ): the backwards view



The backward or mechanistic view. Each update depends on the current TD error combined with traces of past events (Sutton and Barto).



Average RMS error over the first 10 episodes for TD(λ) estimation on a 19-state random walk task (no hidden states) with different values of λ.

TD(λ) algorithm

procedure TD[λ]-ESTIMATION(π,λ)
  Init
    $\hat{V}(s) \leftarrow$ arbitrary value, for all $s \in S$.
    $\hat{z}(s) \leftarrow 0$, for all $s \in S$.
  EndInit
  repeat(For each episode)
    Initialise s
    repeat(For each step of episode)
      a action chosen from π at s
      Take action a; observe r, and next state, s'
      $\delta \leftarrow r + \gamma \hat{V}(s') - \hat{V}(s)$
      $\hat{z}(s) \leftarrow \hat{z}(s) + 1$
      for all $\tilde{s} \in S$ **do**
        $\hat{V}(\tilde{s}) \leftarrow \hat{V}(\tilde{s}) + \alpha\delta\hat{z}(\tilde{s})$
        $\hat{z}(\tilde{s}) \leftarrow \gamma\lambda\hat{z}(\tilde{s})$
      end for
      $s \leftarrow s'$
    until s is absorbing state
  until Done
end procedure

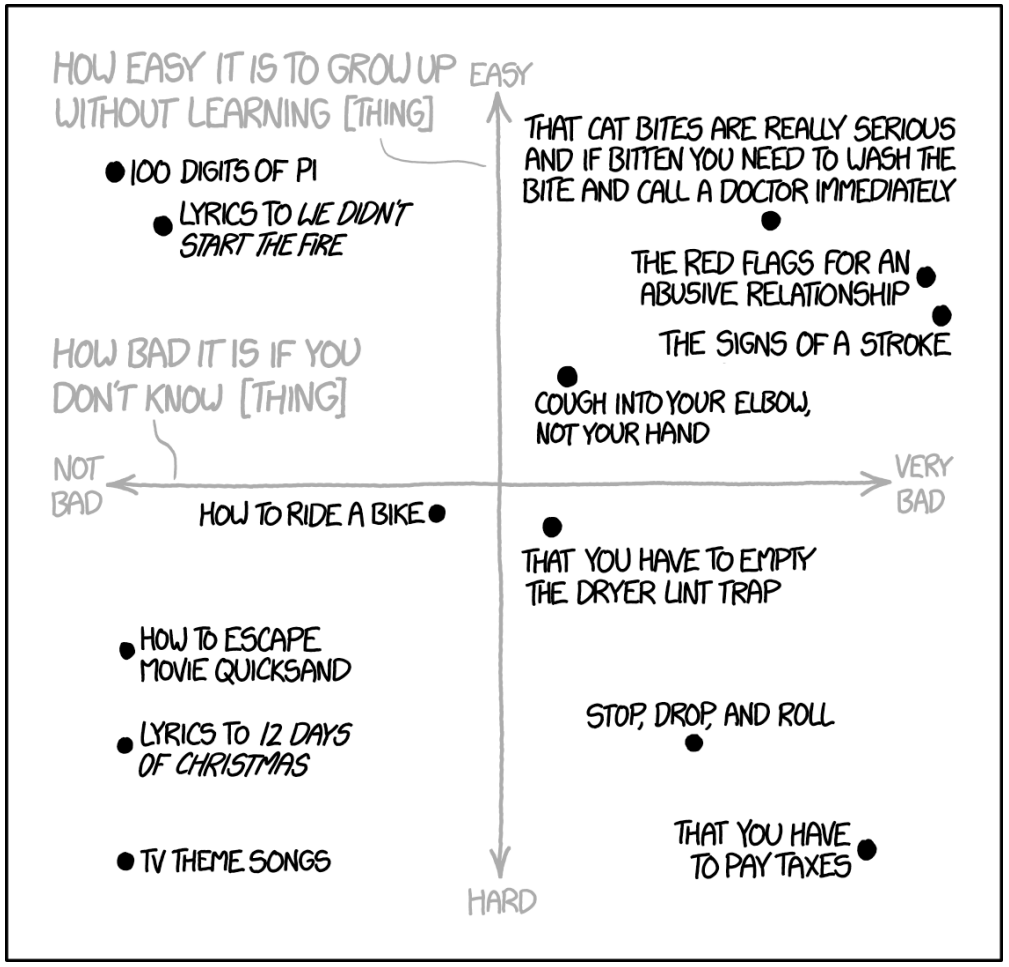
Summary

- Dynamic Programming (DP)
  - Needs access to model dynamics
  - Uses bootstrapping
- Monte Carlo (MC) methods
  - Samples paths from the environment
  - Does not need to know the MDP
  - Care is needed to ensure exploration
  - On- and off-policy methods available
- Temporal Difference (TD) learning
  - Combines best of DP and MC
  - Can be lighter weight
  - Learns within an episode
  - On- and off-policy methods available

Summary

- The Markov assumption may not hold when we have hidden states.
- Monte Carlo can still work; single-step TD methods may not.
- TD(λ) is a linear combination of n-step updates (forward view).
- Eligibility traces capture weights we can apply to TD error at each step (backward view).
- TD(0) ≡ standard TD. TD(1) ≡ Monte Carlo.

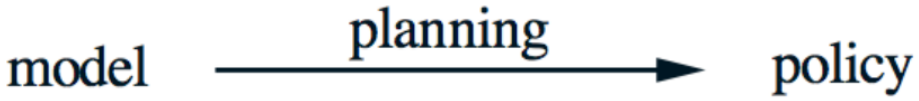
5-minute break



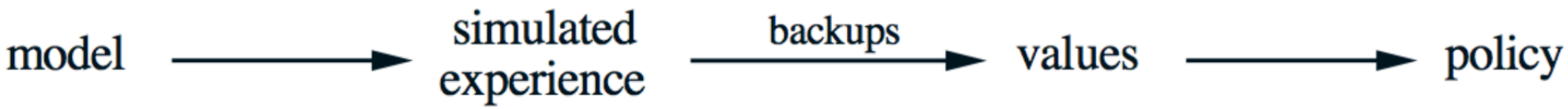
(xkcd)

Models and planning

- Dynamic programming methods are **planning** methods, while Monte Carlo and TD are **learning** methods.
- Planning means taking a model as input and producing or improving a policy for interacting with the modeled environment:



- Planning methods compute value functions as an intermediate step toward improving the policy, and value functions are computed through backup operations applied to simulated experience:

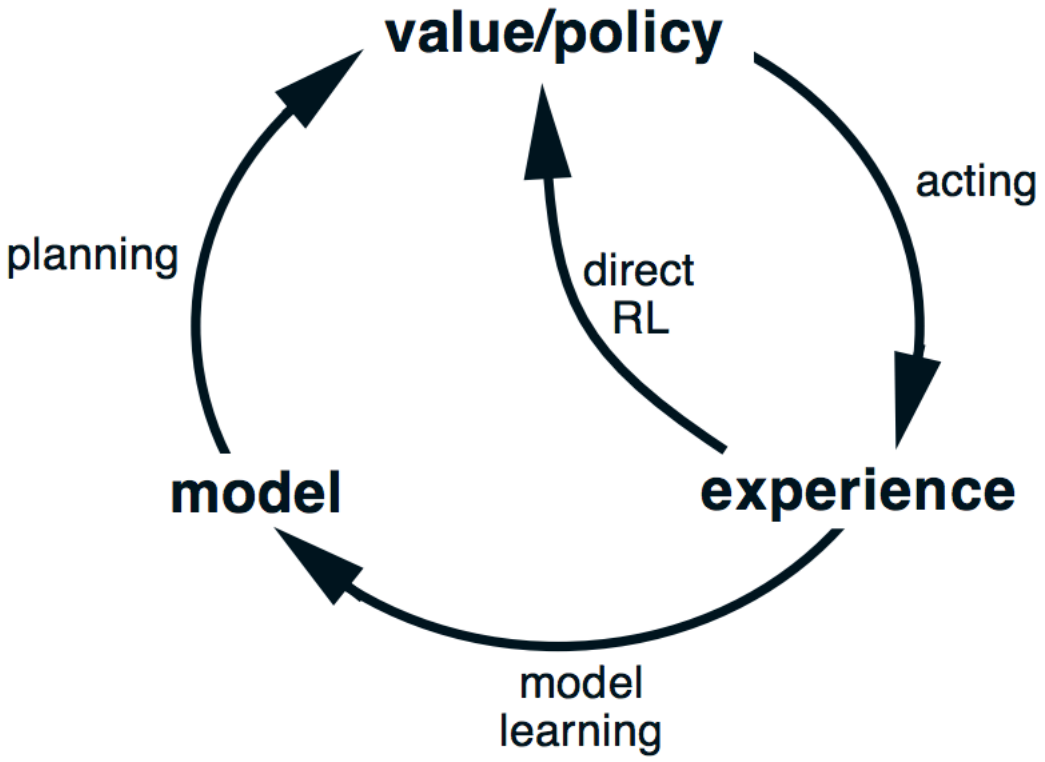


Models and planning

- Learning methods also estimate value functions through backup operations.
- But while planning methods use simulated experience generated by a model, learning methods use real experience generated by the environment.
- In many cases, we can replace the backup step of a planning method with a learning algorithm.
- Planning in small, incremental steps can be very efficient, because we can redirect planning at any time with little wasted computation.

Integrating planning, acting and learning

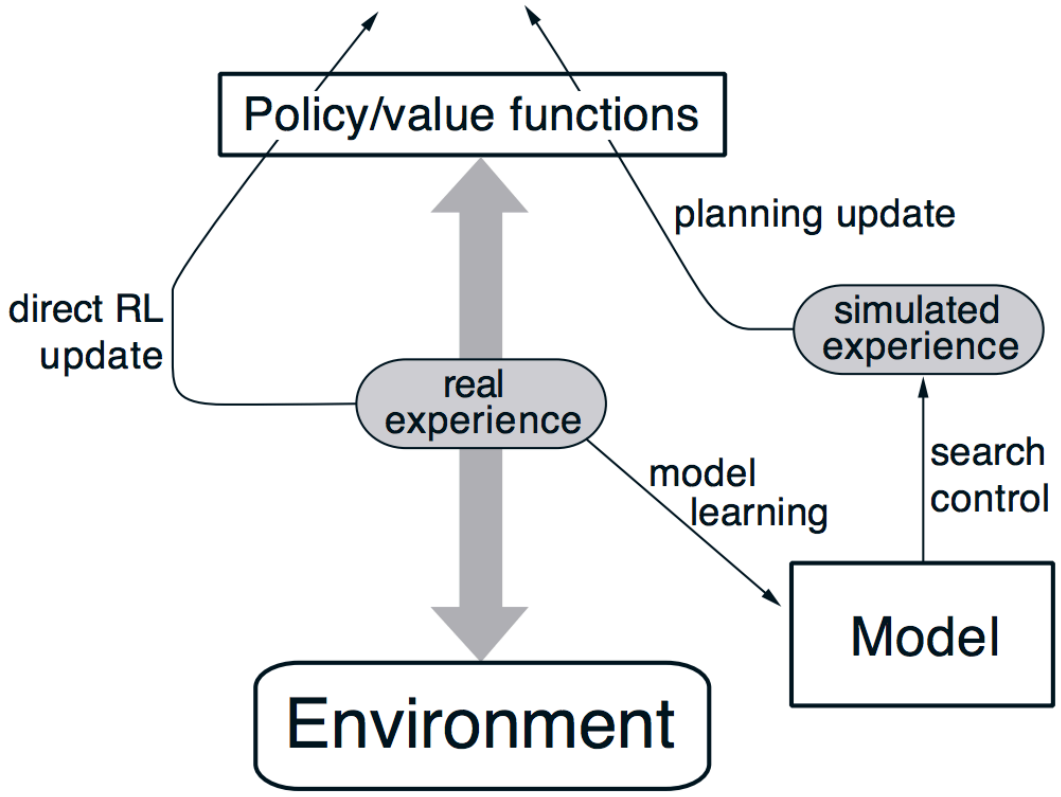
- In a **planning agent**, experience is used to improve the model (model learning) as well as to improve the value function and policy (direct reinforcement learning)
- Improving the value function and policy directly is called **direct RL**
  - Simpler and not affected by biases in the model design
- Improving them indirectly through the model is called **indirect RL**
  - Makes better use of limited experience, so achieves better policy with fewer environmental interactions



(Sutton and Barto)

The Dyna architecture

- **Dyna** is an architecture that is more effective than pure learning methods and more computationally efficient than pure planning methods.
- It simultaneously uses experience to build a model, uses experience to adjust the policy, and uses the model to adjust the policy.



(Sutton and Barto)

Dyna-Q algorithm

procedure DYNA-Q(N, k)
  Init
    $\hat{Q}(s,a) \leftarrow$ arbitrary value, for all $s \in S$ and $a \in A$
    Initialize $Model(s,a)$ for all $s \in S$ and $a \in A$
    $s \leftarrow$ sample a (nonterminal) state
  EndInit
  for N iterations **do**
    $a \leftarrow \epsilon$-greedy(s,Q)
    Take action a; observe reward r and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$ → Use experience to improve the value function/policy
    $Model(s,a) \leftarrow s', r$ → Use experience to improve the model
    $s'' \leftarrow s'$
    for k iterations **do**
      $s \leftarrow$ random previously observed state
      $a \leftarrow$ action previously taken in s
      $s', r \leftarrow Model(s,a)$
      $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$ → Use the model to improve the value function/policy
    $s \leftarrow s''$

Dyna-Q algorithm

procedure DYNA-Q(N, k)
  Init
    $\hat{Q}(s, a) \leftarrow$ arbitrary value, for all $s \in S$ and $a \in A$
    Initialize $Model(s, a)$ for all $s \in S$ and $a \in A$
    $s \leftarrow$ sample a (nonterminal) state
  EndInit
  for N iterations **do**
    $a \leftarrow \epsilon$-greedy(s, Q)
    Take action a; observe reward r and next state s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $Model(s, a) \leftarrow s', r$ → assumes a deterministic environment
    $s'' \leftarrow s'$
    for k iterations **do**
      $s \leftarrow$ random previously observed state
      $a \leftarrow$ action previously taken in s
      $s', r \leftarrow Model(s, a)$
      $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s''$

Summary

- Dyna Q is a hybrid architecture which is a middle ground between model-based RL (Dynamic Programming) and model-free RL (TD learning).
- It is more effective than TD learning methods and more efficient than Dynamic Programming methods.
- Dyna-Q simultaneously uses experience to improve the value function and policy and to improve the model of the environment.

Survey of Deep RL Topics

Limitations of simple state representation

- Using simple state representations can be impractical for large state spaces (curse of dimensionality).
- This is not usually a problem with actions.
  - For instance, in 9x9 Go: $|S| = 10^{38}$ and $|A| = 81$
- Another limitation is that we would like to generalize experience to unseen states.
- A powerful alternative way of representing states is as a set of weights over features.
- Each state is represented by a **feature vector**, $\phi(s) \in \mathbb{R}^n$.

Function approximation

- Each state is represented by a **feature vector**, $\phi(s) \in \mathbb{R}^n$.
- We approximate the V and Q functions with weights:

$$V^{\pi}(s) \approx \mathbf{v}^T \phi(s) \qquad Q^{\pi}(s,a) \approx \mathbf{q}_a^T \phi(s)$$

- The Markov assumption may hold for states:

$$p(s_{t+1}, r_{t+1} | s_t, a_t) = p(s_{t+1}, r_{t+1} | s_t, a_t, \dots, s_0, a_0)$$

- ...but not for feature vectors:

$$\text{for } \phi(s_t) = \phi_t, \; p(\phi_{t+1}, r_{t+1} | \phi_t, a_t) \neq p(\phi_{t+1}, r_{t+1} | \phi_t, a_t, \dots, \phi_0, a_0)$$

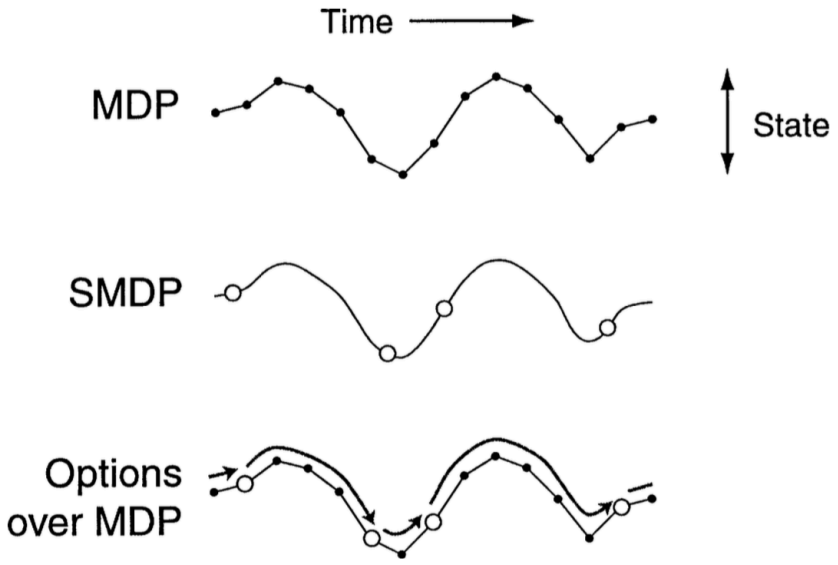
- This is called **function approximation**.
- To estimate the weights, there are Monte Carlo and TD(λ) based algorithms available.

Hierarchical RL: SMDPs

- Human decision making very often requires choice among temporally extended courses of action over a broad range of time scales.
- Traditional MDPs do not involve temporal abstraction or temporally extended action. There is no notion of a course of action persisting over time.
- **Semi-Markov decision processes** (SMDPs) allow us to model continuous-time discrete-event systems.
- Actions in an SMDP take variable amounts of time and model temporally-extended courses of actions.

Hierarchical RL: Options framework

- **Options** are courses of action within an MDP whose results are state transitions of extendable and variable duration.
- Options are a generalization of primitive actions which can be temporally extended.
- A fixed set of options defines a discrete-time SMDP embedded within the MDP.

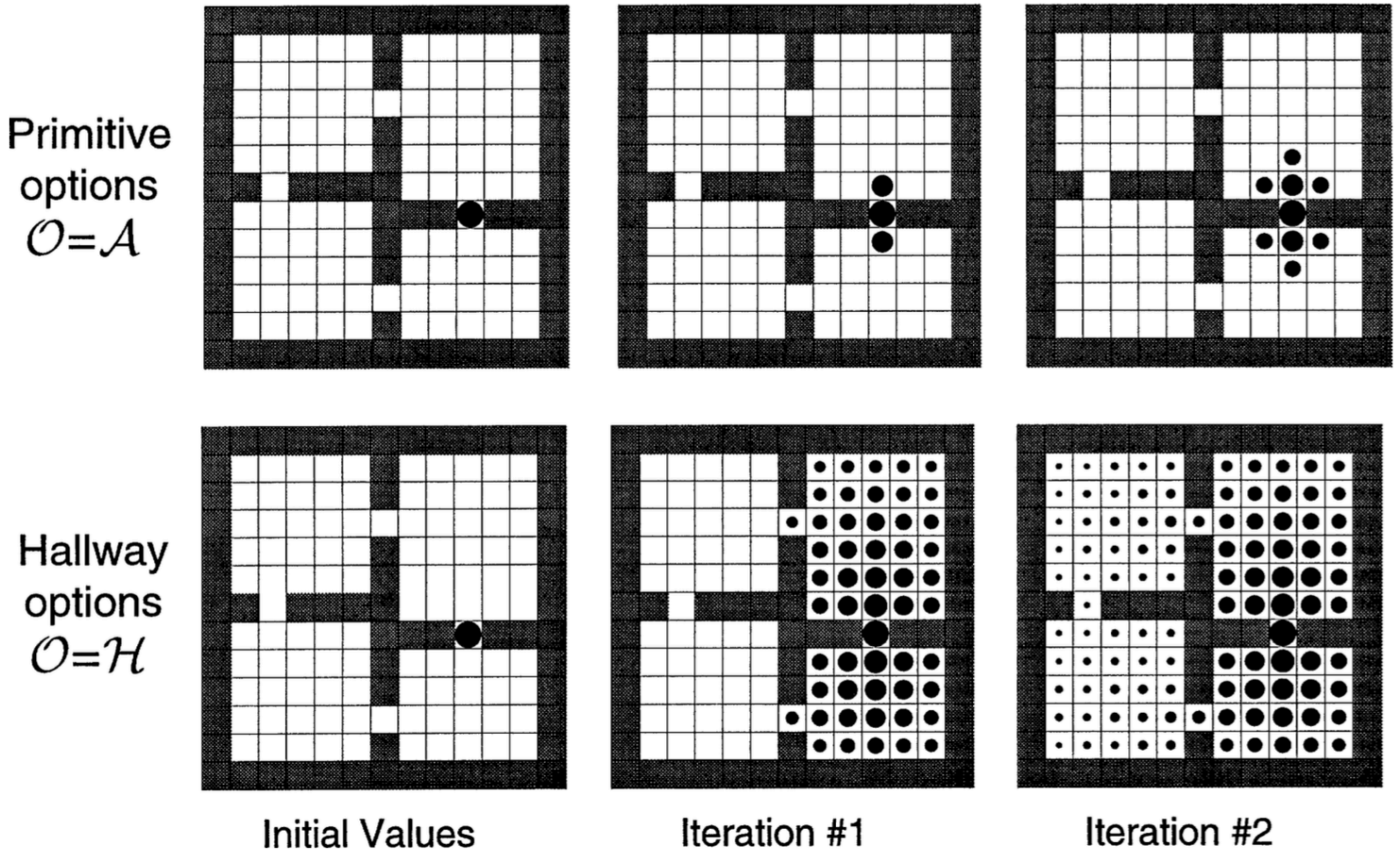


(Sutton et al., 1998)

Options framework

- An option consists of:
  - An initiation set $I \subseteq S$: the states where the option is available
  - A policy $\pi : S \times A \to [0, 1]$: the policy according to which actions are taken while the option is in effect
  - A termination condition $\beta : S^+ \to [0, 1]$: the probability that the option terminates at each state
- We can easily adapt the Bellman equations for V and Q functions to SMDPs and options.
- Planning and learning methods can also be adapted.

SMDP example: multi-room grid world



(Sutton et al., 1998)

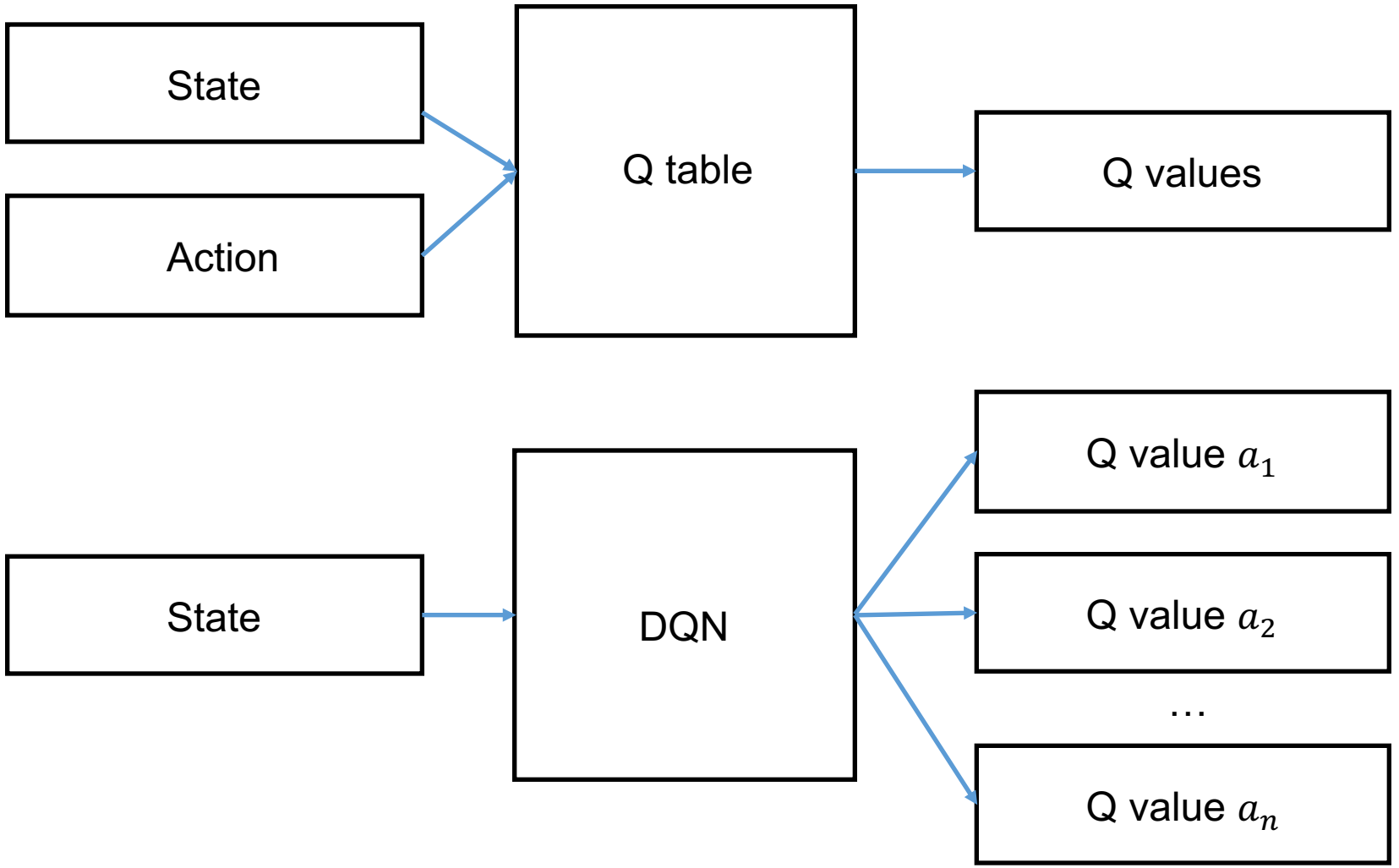
Deep Q-learning

- We can use the Q-learning algorithm with feature vectors instead of states (function approximation):

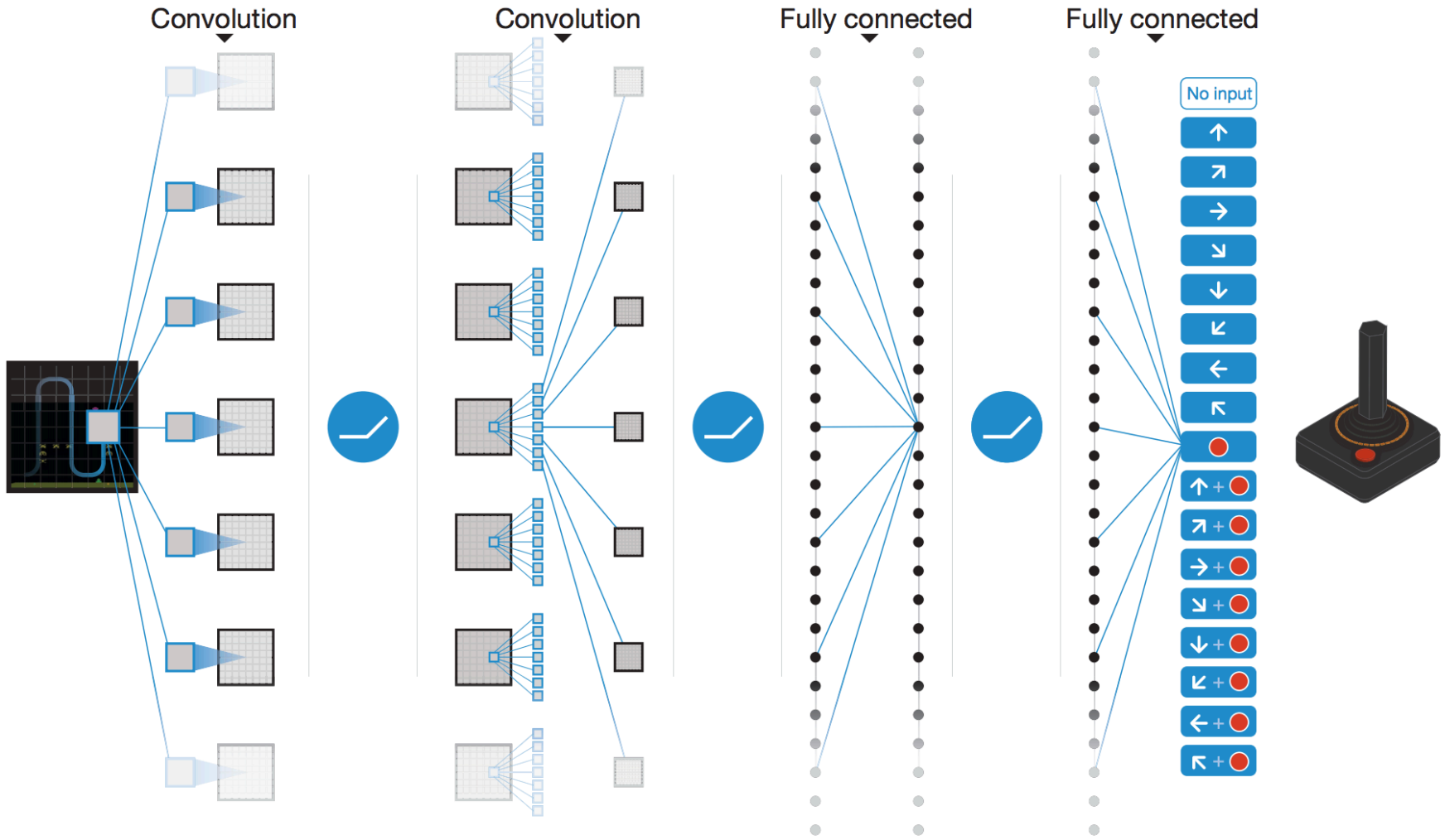
$$Q^{\pi}(s,a) \approx \mathbf{q}_a^T \phi(s)$$

- We learn the Q values for each state-action pair $Q^{\pi}(s,a)$ by estimating the weights $\mathbf{q_a}$ for each action.
- Estimating weights is what deep networks do best!
- We use a Deep Q Network (DQN).

Q-learning vs. Deep Q-learning



Playing ATARI games with DQN



(Mnih et al., 2015)

Deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

→ Experience replay

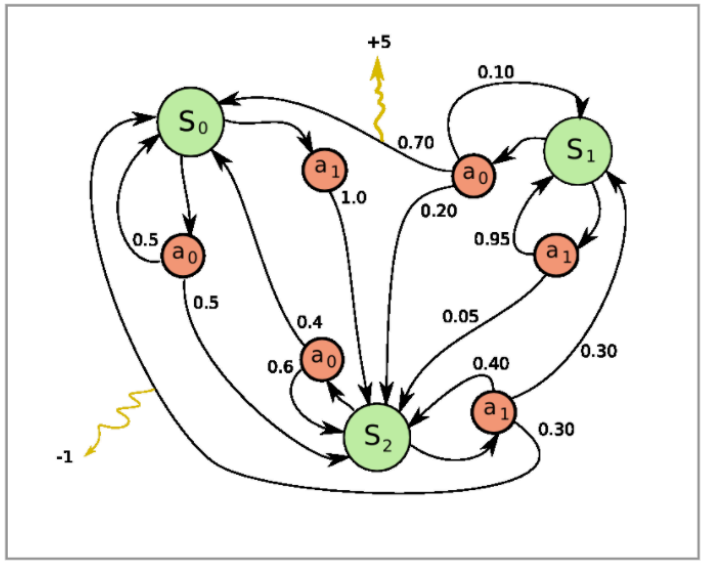
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

→ Q-learning update

    Every C steps reset $\hat{Q} = Q$
  End For
End For

(Mnih et al., 2015)

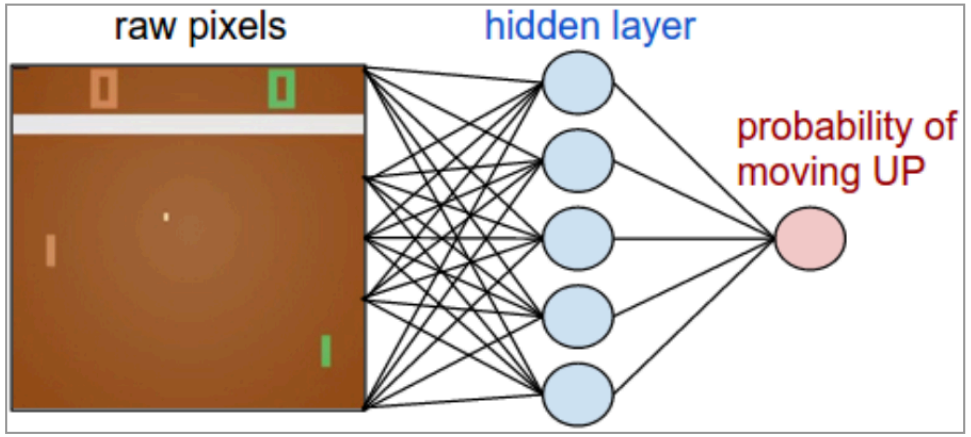
Pong as an MDP



(Andrej Karpathy's blog)

Pong as an MDP

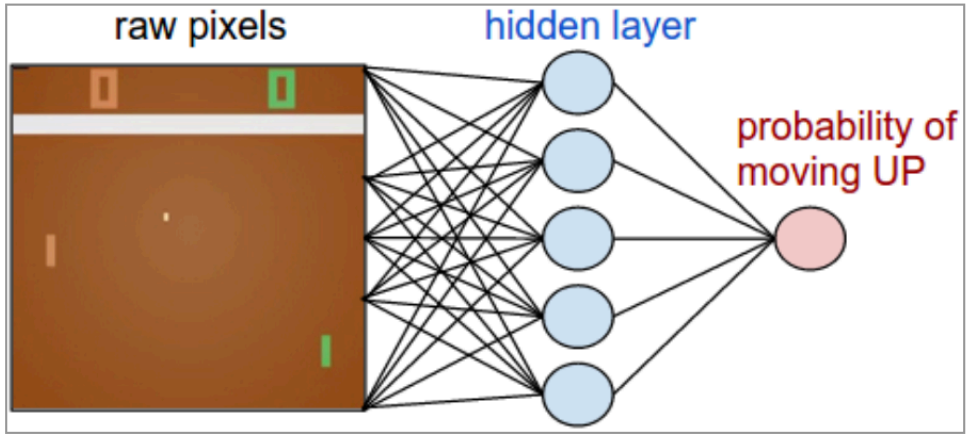
- The network receives an image frame as input (current state of the game).
- Choices available: move the paddle up or down.
- The output of the network is a probability: $p(\mathrm{up})$.
- For each action the state of the game changes, and we get a reward: +1 if the ball goes past the opponent's paddle, -1 if we miss the ball, and 0 otherwise.
- We want to find the weights W that give us the best action at each game state.



(Andrej Karpathy's blog)

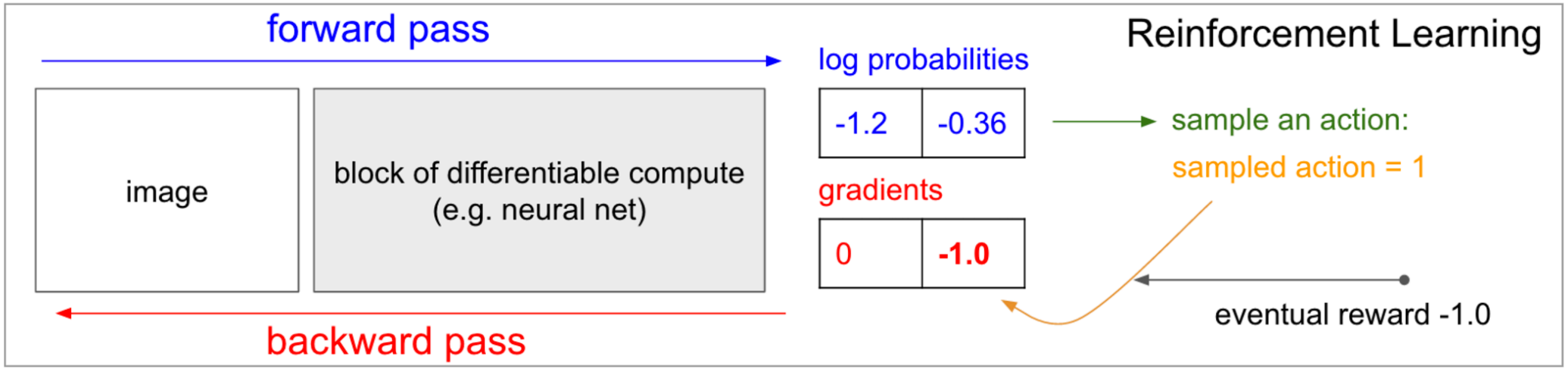
Pong as an MDP

- Most actions will have a reward of zero.
- When we finally get a non-zero reward, how do we know which of our many actions led to it?
- This is called the **credit assignment problem**.



(Andrej Karpathy's blog)

Policy gradients



(Andrej Karpathy's blog)

The End