Lecture 4: Computational Cognitive Modeling

Reinforcement Learning (pt. 1)

email address for instructors:
instructors-ccm-spring2019@nyuccl.org

course website:
https://brendenlake.github.io/CCM-site/

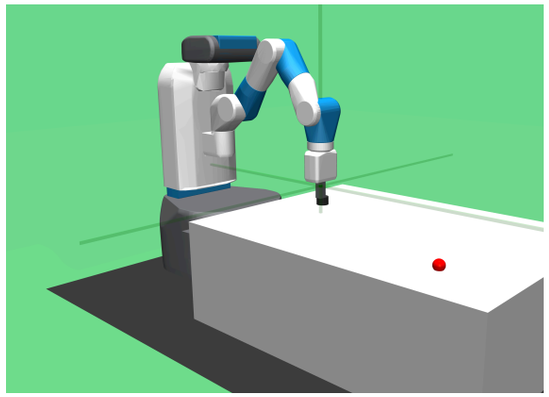
Types of learning (from a ML perspective)

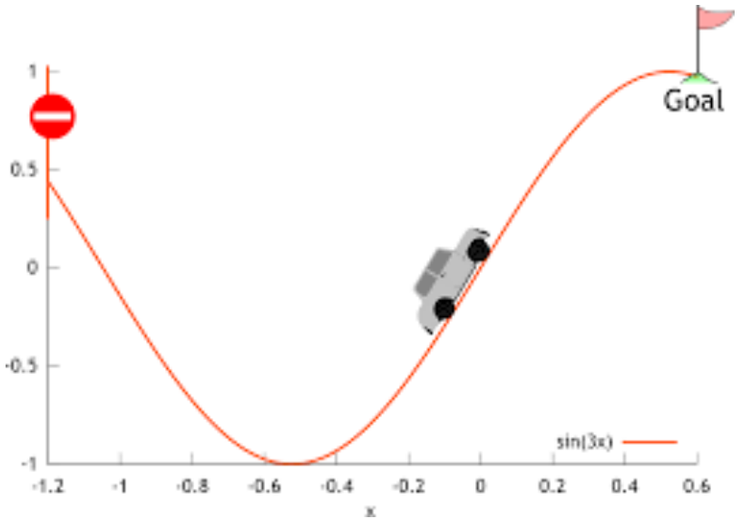
- **Supervised learning**
    - Works by instructing the learning system what output to give for each input. Corrective feedback that is diagnostic (“this not that”).
- **Unsupervised learning**
    - No feedback. Works by detecting the correlational and covariation structure of the input to find generalizable patterns.

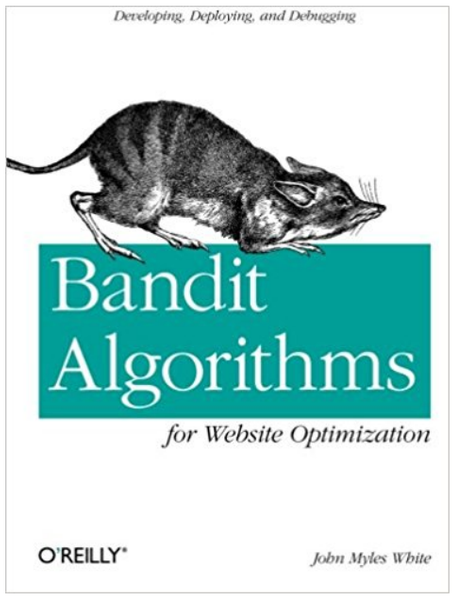
- **Reinforcement learning**
    - Evaluating feedback (good) but not corrective. Goal is to take actions or make decisions in order to earn higher rewards.

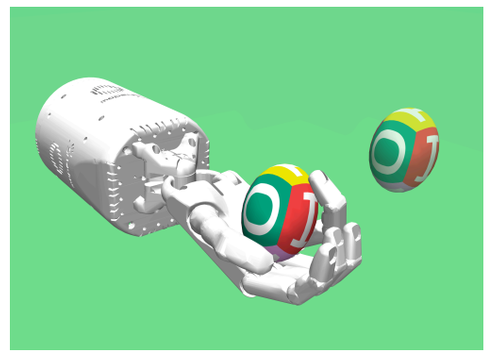
Example applications: Dynamic control problems

- Autonomous helicopter flight
- Robot legged locomotion
- Cell-phone network routing
- Marketing strategy selection
- Factory control
- Efficient web-page indexing
- A/B testing

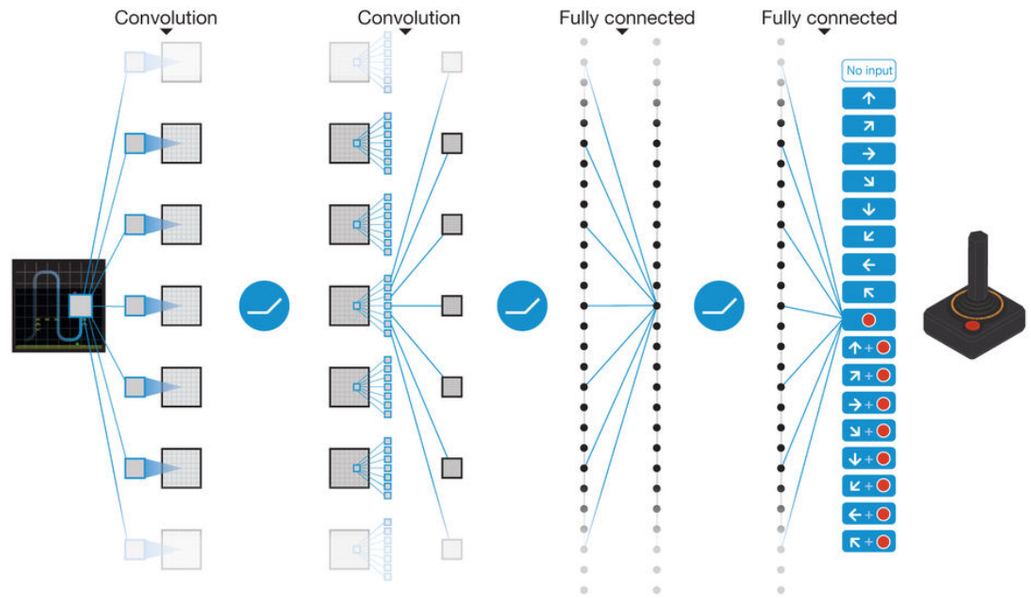


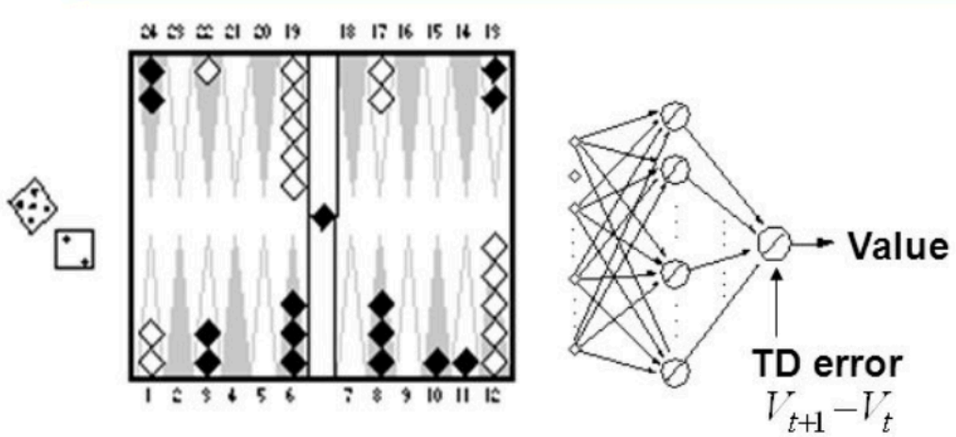






Example applications: Sequential decision making problems, games

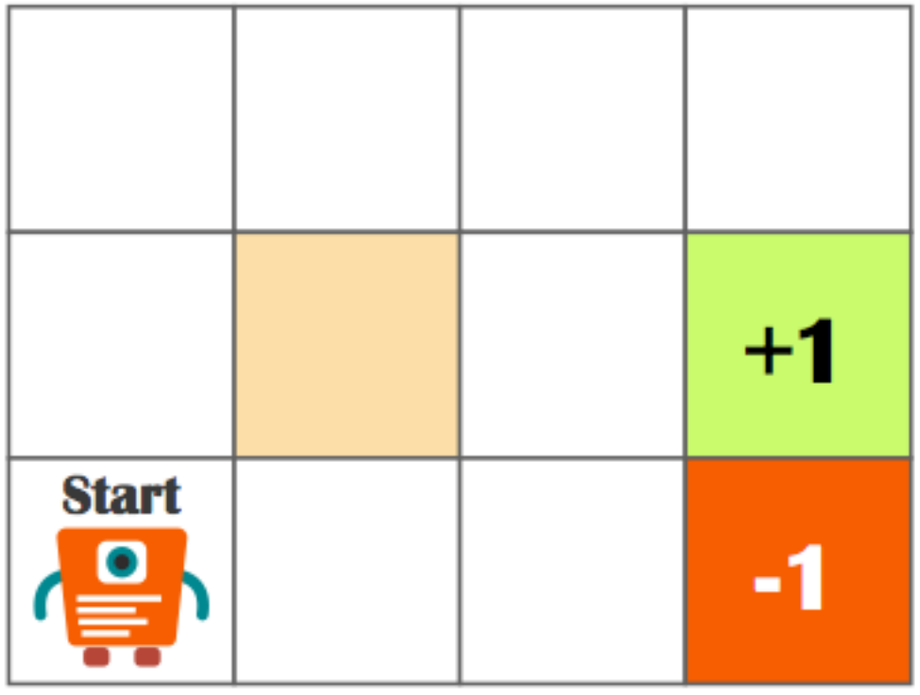




Tesauro, 1992–1995

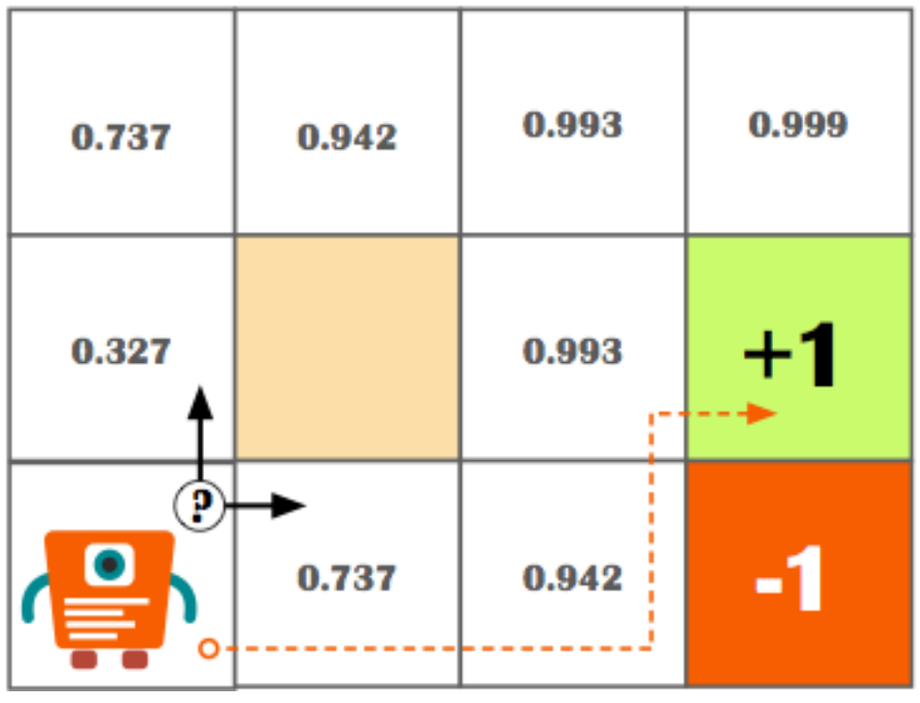
Action selection by 2–3 ply search

Canonical example



- Grid cells are rooms the robot can be in (aka “states”)
- Actions are **move up, move down, move left, move right**
- The orange cell is an obstacle
- The +1 is the reward for entering that state
- The -1 is the penalty for entering that cell
- **What route should the agent take?**

Canonical example



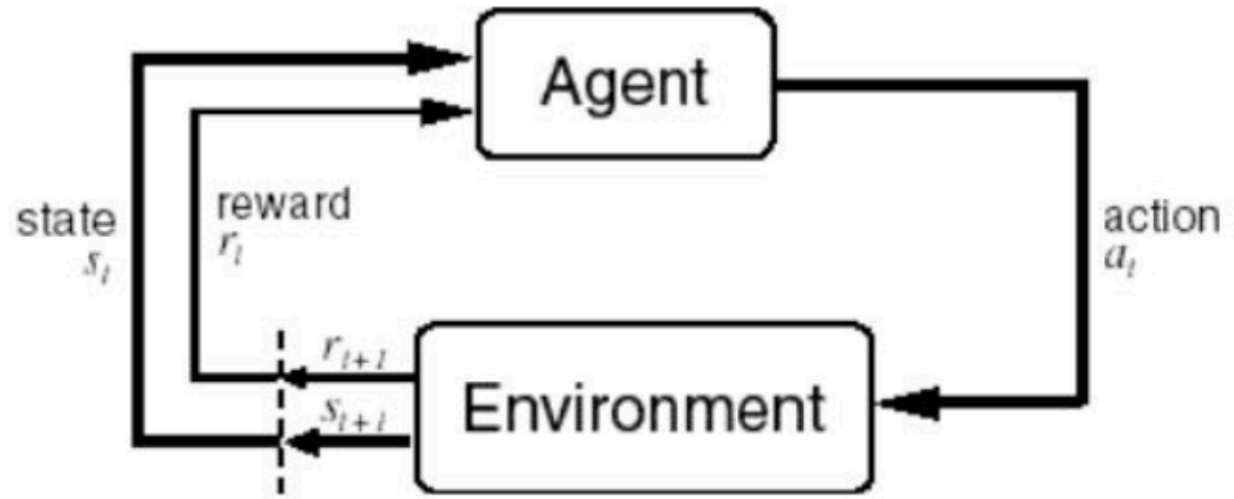
- Solution takes the form of values or numbers assigned to each state (or state-action pair) that determine how good they are in the long run.

- A good/optimal decision strategy (policy) can be determined by choosing to move to the immediate state with the highest value.

- **Reinforcement learning is a family of methods for solving these types of problems that borrows from research on human/animal learning and from research in operations/ planning.**

Key Components of RL Systems

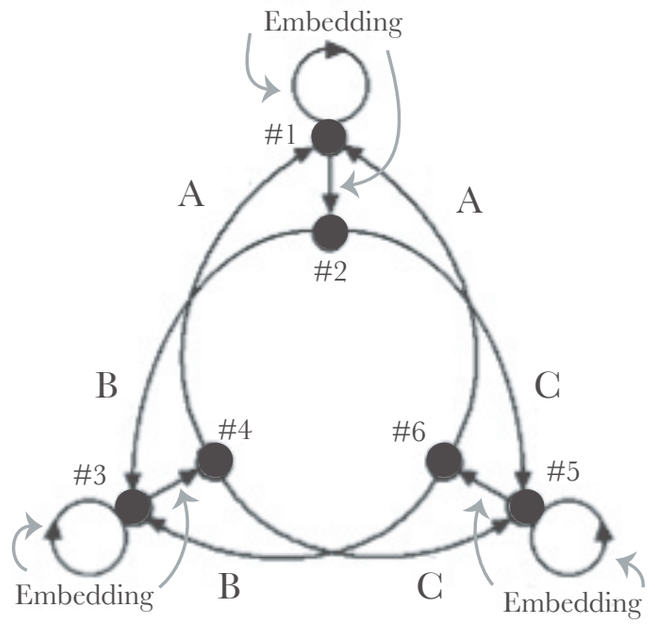
- The environment
- Reward function
- Policy
- Value function
- Model of the environment



The environment

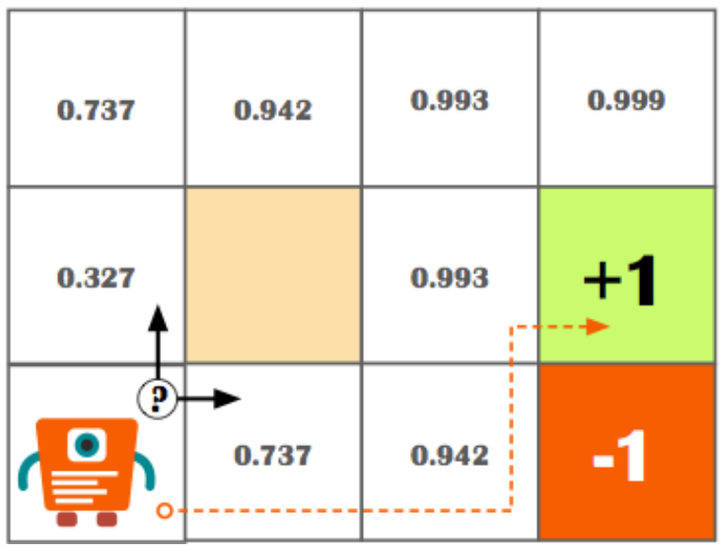
- Modeled as a Markov Decision Process
- At its essence means that the system is defined by the one-step dynamics
- Simply put, the distribution of future states and rewards depend only on where you are now and what action you take

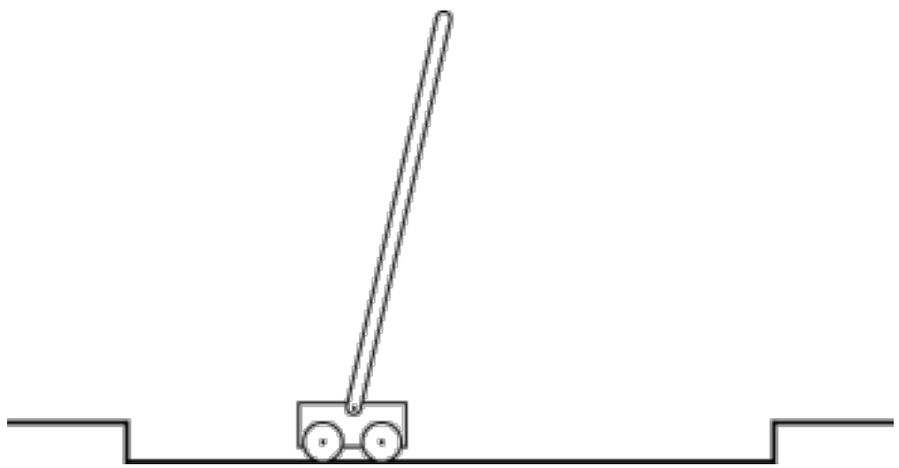
$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}$$



What are the states?

- **Robot navigation:** different rooms that you could be in. Transitions are like moving between doorways that connect the rooms
- **Pole balancing:** the position of the cart on the x-axis, the angle of the pole, possibly momentum or other terms
- Generally could be defined in terms of features (even down the pixels) as they are in DeepMind Atari Deep RL.
- What are they for humans?





Reward Function

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

- Typically a single number which indicates how good or bad the current state is.
- The overall goal of the agent is to maximize the discounted reward it gets over the long term.
- The parameter (gamma) determines how much weight is given to immediate version delayed rewards.
- Reward are the immediate, primary sensory feedback from a particular state, in contrast to value functions

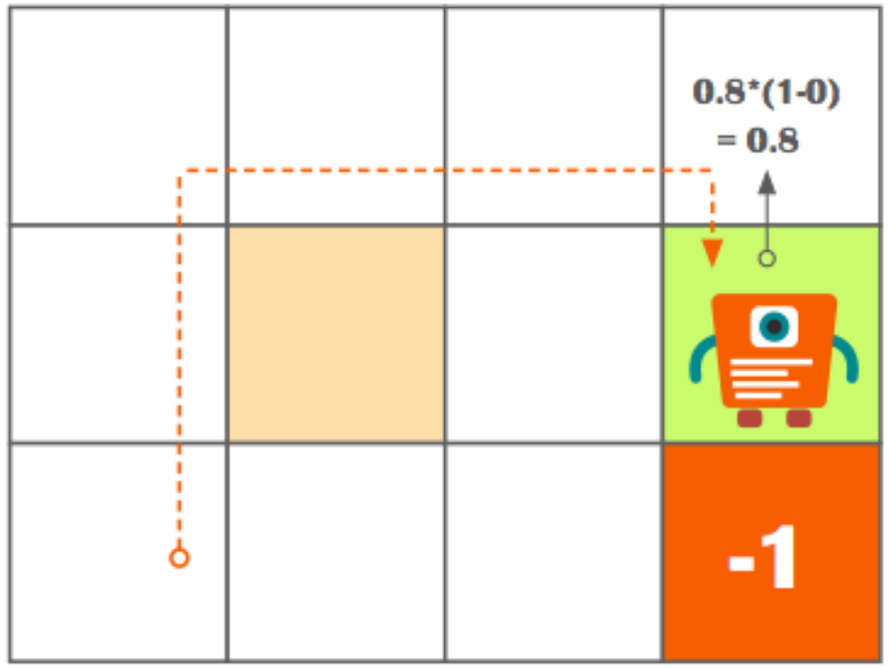
Policy

- The rules for how an agent should act. A full set of stimulus-response rules or associations.

- Represented as π, where $\pi(s, a)$ means the probability of selecting action **a** in state **s**.

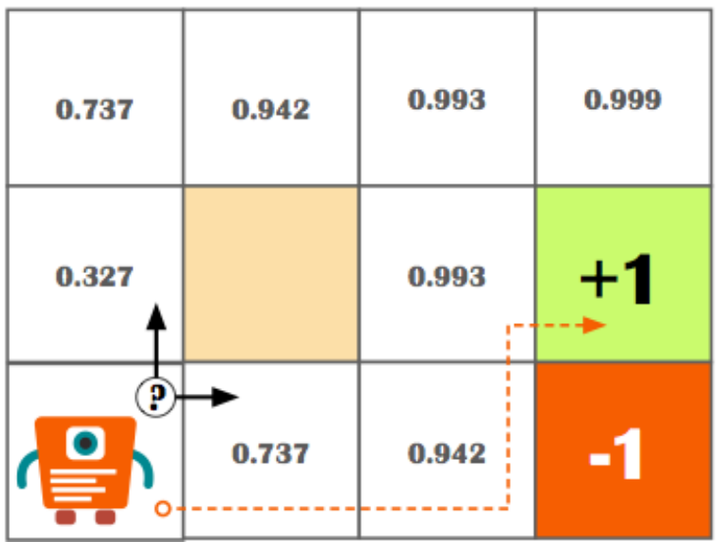
- Policies can be explicitly stochastic in nature or deterministic.

- **What we are trying to learn: a good policy for the environment we face.**



Value Function

$$V^{\pi} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s\}$$



- The long run total that an agent can expect to make in the future starting from that state. So unlike rewards these are not immediate short-term values, but based on a longer temporal window.

- Since the goal is to maximize reward over the long term, in a sense you are tying to choose actions or states associated with higher values.

- The value function is essentially a stand-in for what you will get in the long run from a particular choice and is important in learning

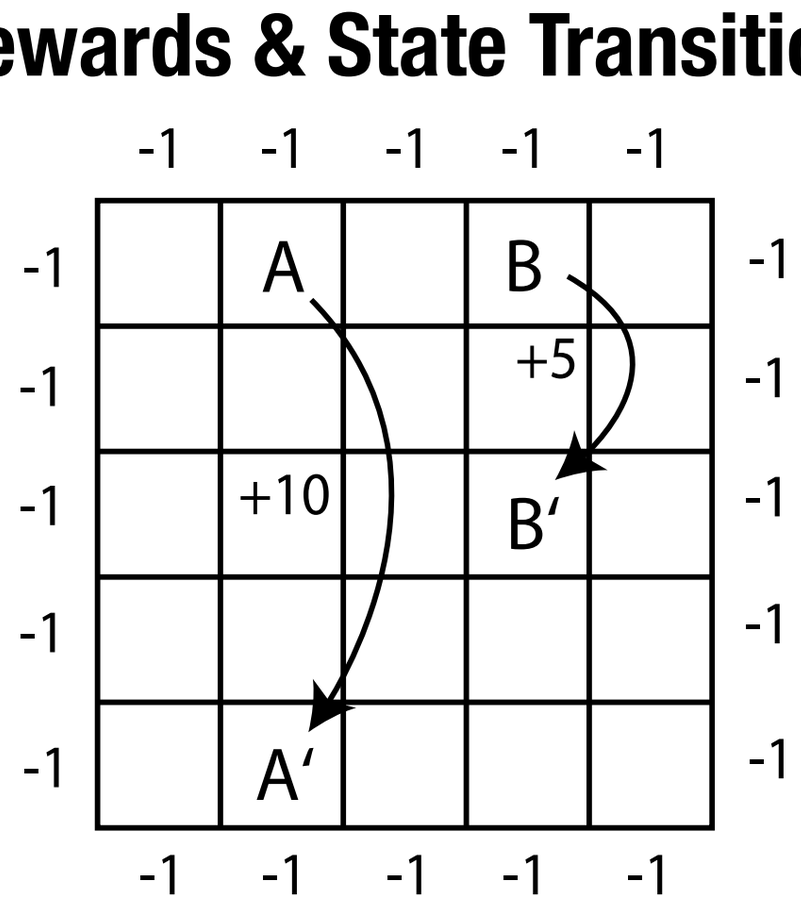
Model of Environment

- Knowledge about the way the world works
- Essentially means a representation of the Markov process itself such as the probabilities of moving from state a to state b given that you take action c
- **Not absolutely necessary** (e.g., Monte Carlo methods or temporal difference learning can operate with out an explicit model!!)

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

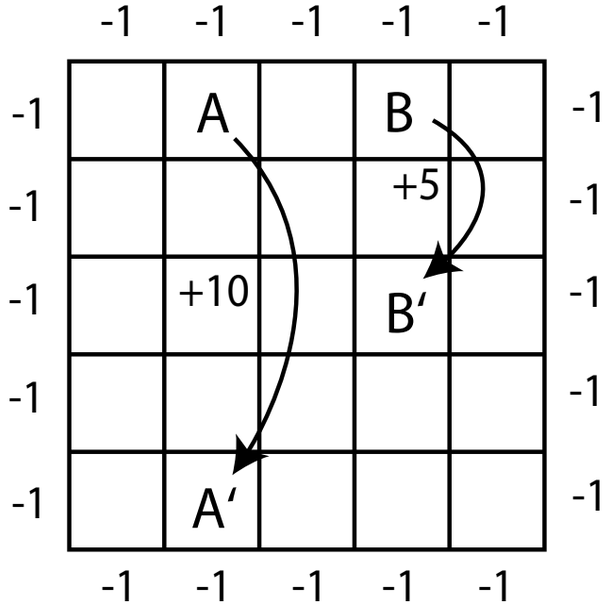
$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

An example: Behaving optimally in a known world

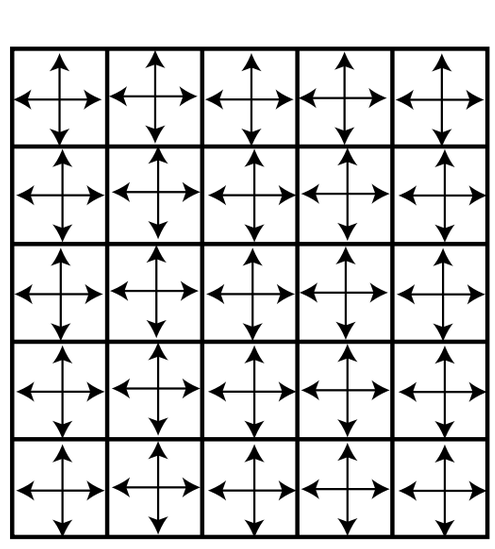


An example: Behaving optimally in a known world

Rewards & State Transitions



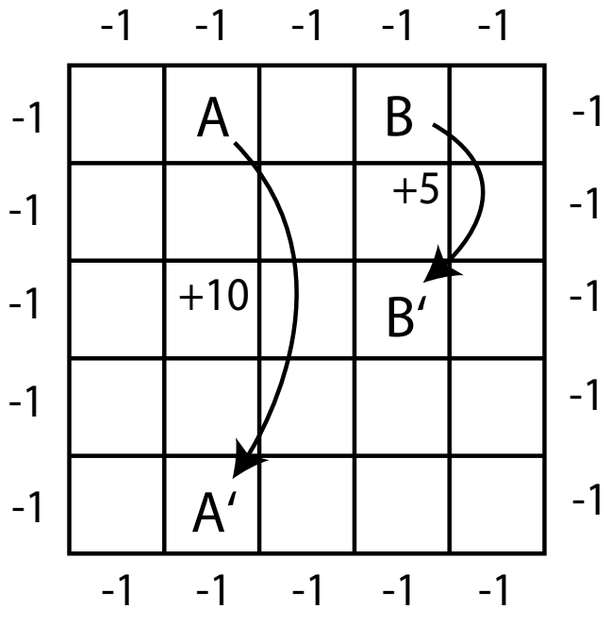
Agent's Policy (π)



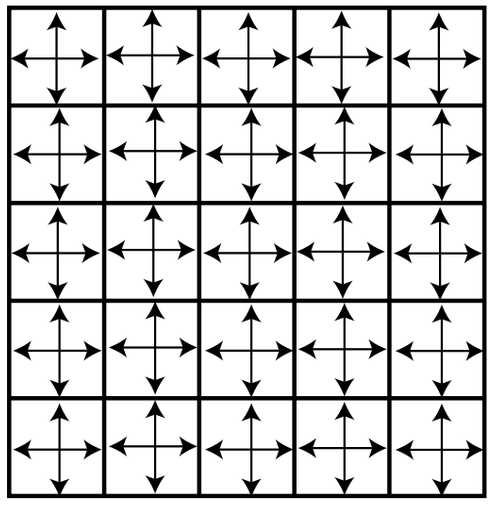
$\gamma = 0$

An example: Behaving optimally in a known world

Rewards & State Transitions

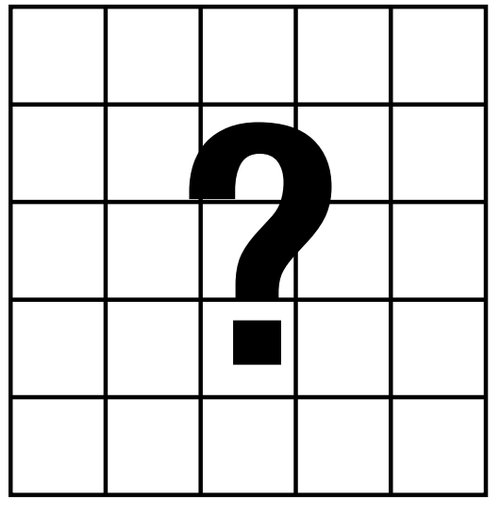


Agent's Policy (π)



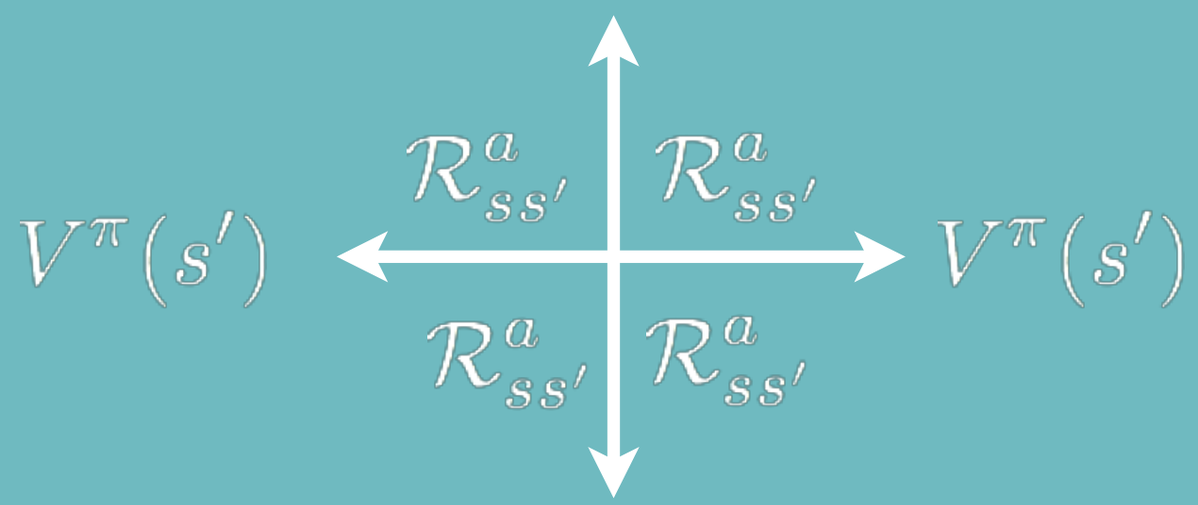
$\gamma = 0$

Value Function (V)



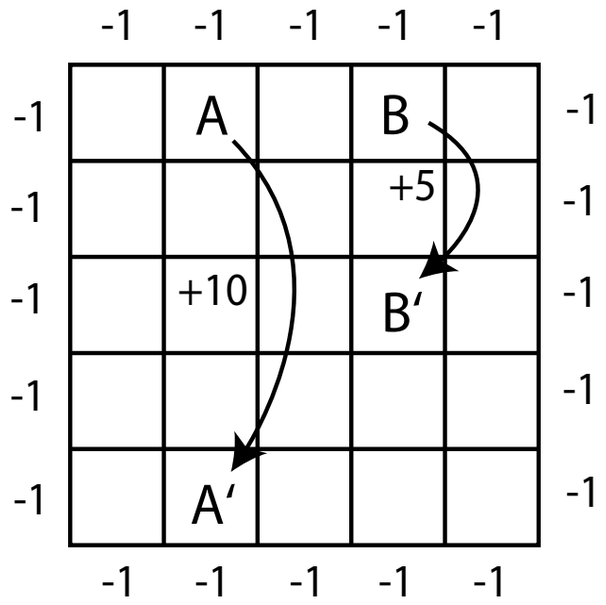
Bellman Equation

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

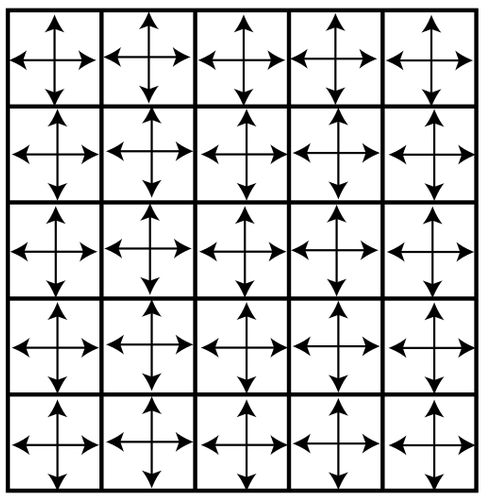


- If we know the function $\mathcal{P}^{a}_{ss'}$ and $\mathcal{R}^{a}_{ss'}$ (i.e. have perfect knowledge of the environment), we can easily solve for $V^{\pi}(s)$ by simply solving the systems of equations under our policy

Rewards & State Transitions



Agent's Policy (π)

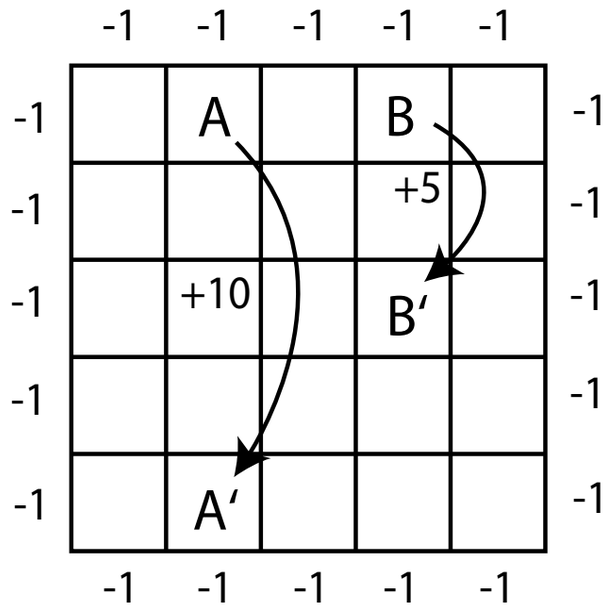


$\gamma = 0$

Value Function (V)

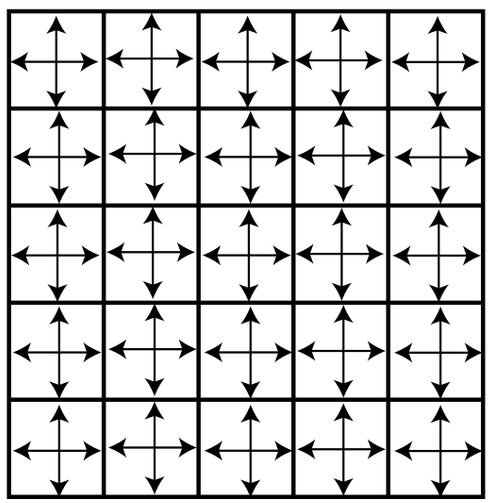
-0.5	10	-0.25	5	-0.5
-0.25	0	0	0	-0.25
-0.25	0	0	0	-0.25
-0.25	0	0	0	-0.25
-0.5	-0.25	-0.25	-0.25	-0.5

Rewards & State Transitions



$\gamma = 0.9$

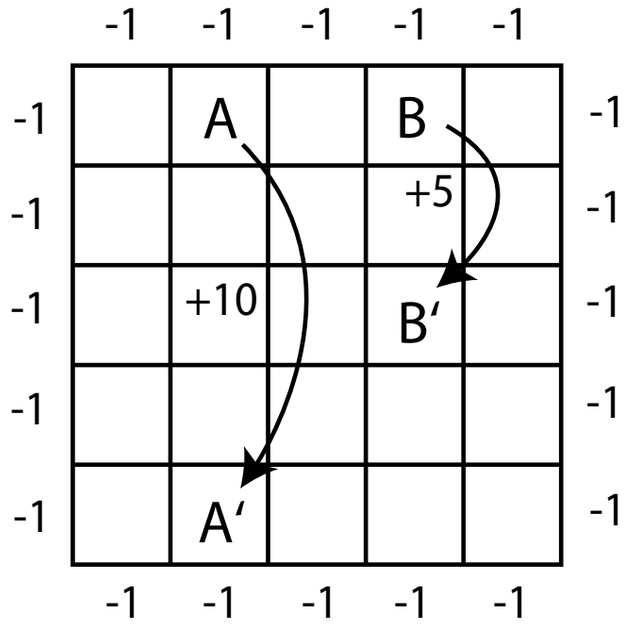
Agent's Policy (π)



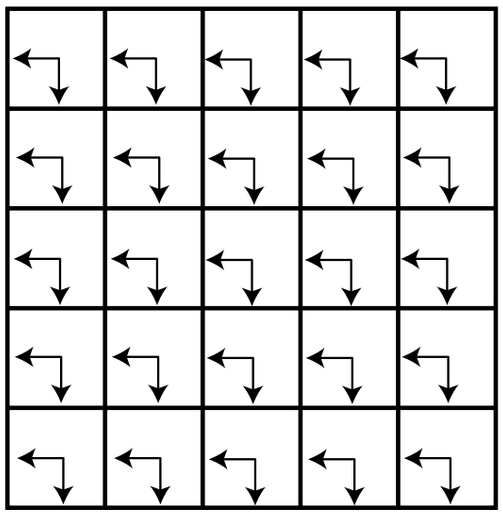
Value Function (V)

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Rewards & State Transitions



Agent's Policy (π)



$\gamma = 0.9$

Value Function (V)

-7.2	1.8	-1.9	-0.5	-2.4
-7.7	-6.8	-6.0	-5.4	-4.9
-8.3	-7.4	-6.7	-6.1	-5.6
-9.0	-8.1	-7.4	-6.8	-6.3
-9.9	-9.1	-8.4	-7.7	-7.2

Solution methods for policy evaluation

v_k for the Random Policy

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

iterations

- Generally this can be solved in one step as a complicated system of equations (the value of each state is defined in terms of all the other variables and so you have N variables and N equations)
- More practically for large problems (think about scale of a game like Go) there is an iterative solution where you initialize the values all to zero and then update each one in light of of the Bellman recurrence relation.
- Each pass brings you closer and closer to the true values and will converge to the actual stationary Bellman values.

Finding the optimal policy via Value Iteration

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s') \right]$$

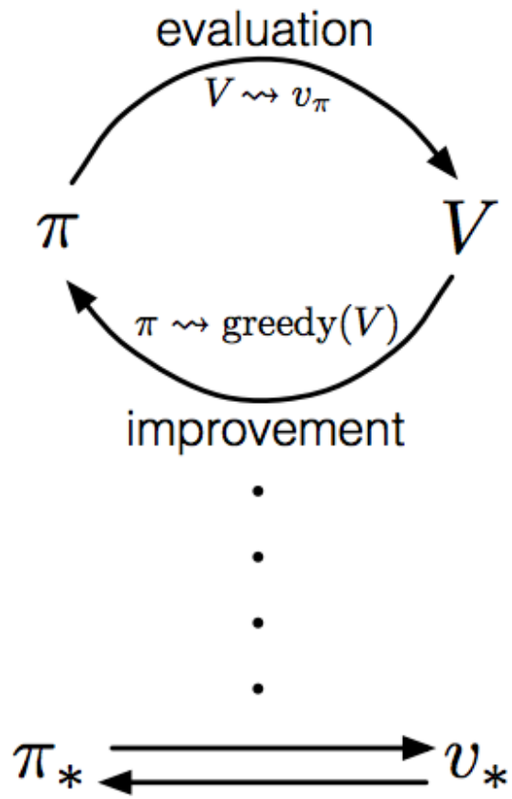
- Sort of like the Expectation-Maximization algorithm iterate between evaluation and policy improvement cycles!
- This is known as **value iteration**

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$



Finding the optimal policy via Value Iteration

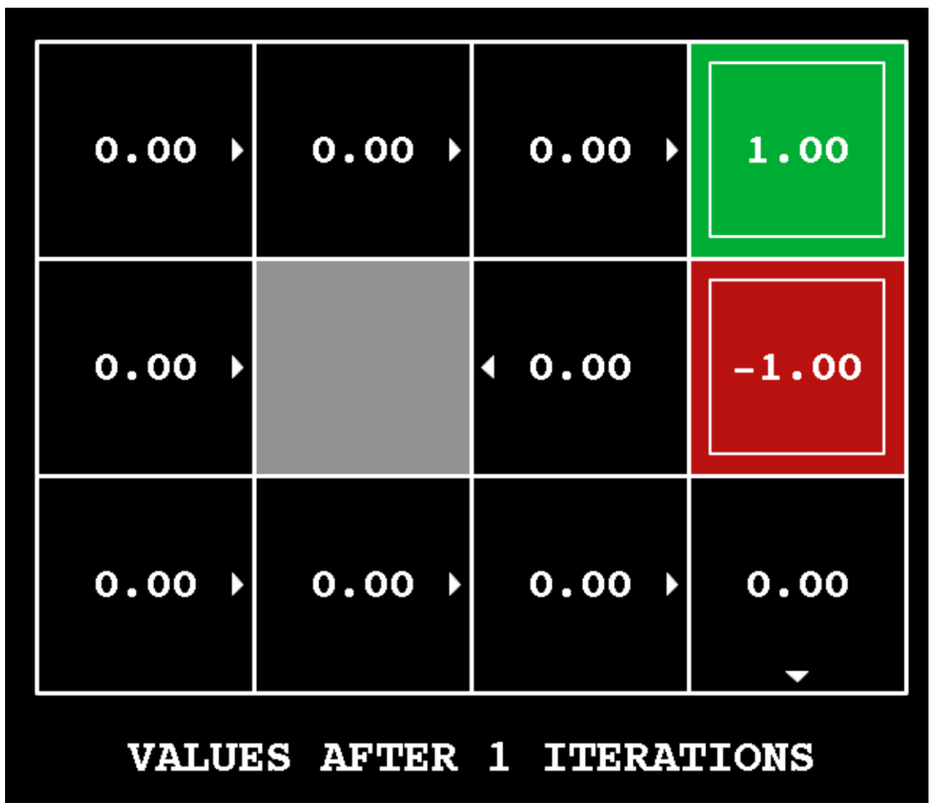
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

noise = 0.2
gamma = 0.9



Finding the optimal policy via Value Iteration

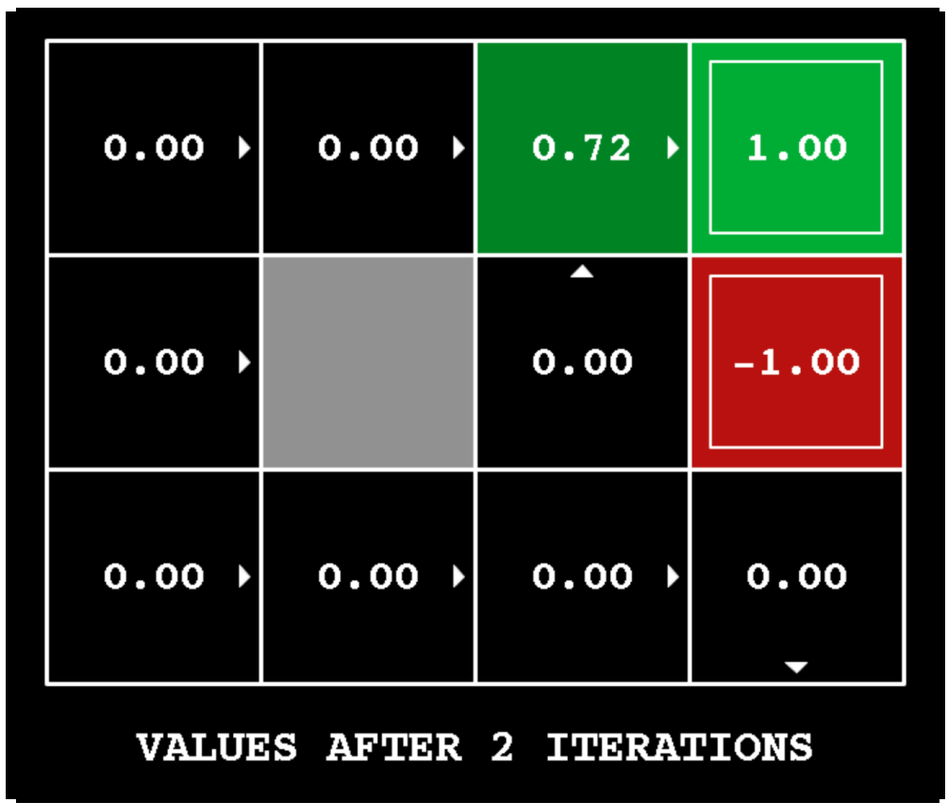
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

noise = 0.2
gamma = 0.9



0.72 = 0.8*(0+0.9*1.0)

Finding the optimal policy via Value Iteration

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

noise = 0.2
gamma = 0.9

0.78 = 0.8*(0+0.9*1.0)+
0.2(0.4*0.9*0.72)

0.52 = 0.8*(0+0.9*0.72)

…

Finding the optimal policy via Value Iteration

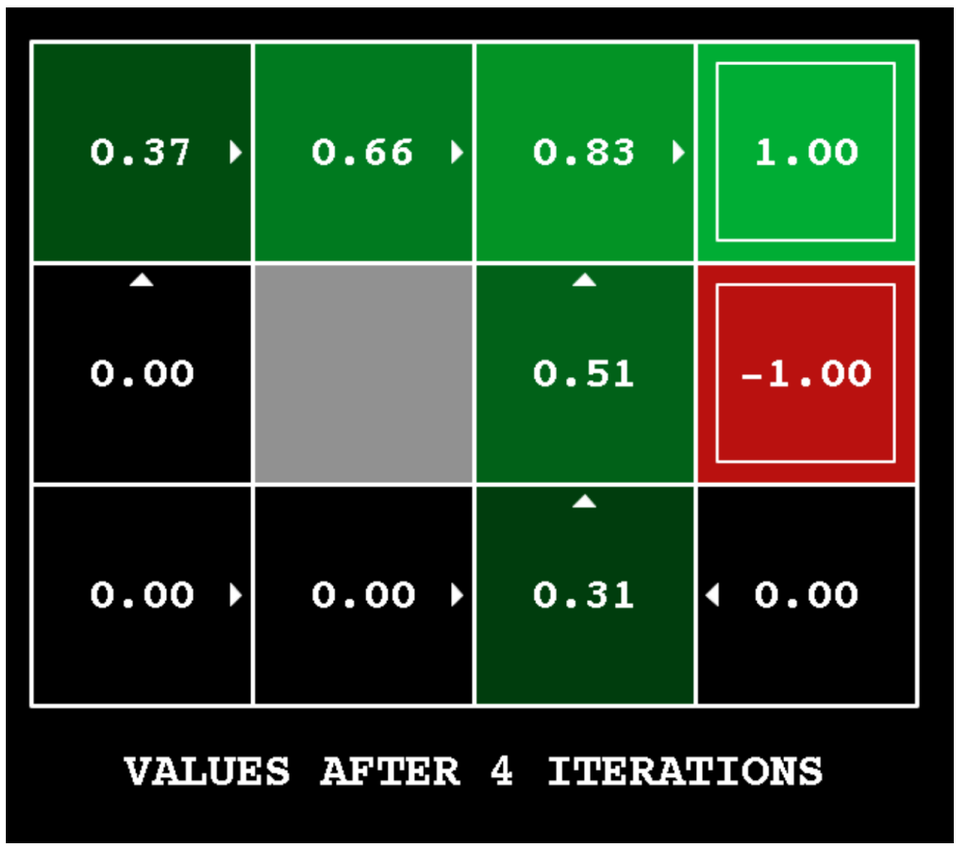
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$

noise = 0.2
gamma = 0.9



Finding the optimal policy via Value Iteration

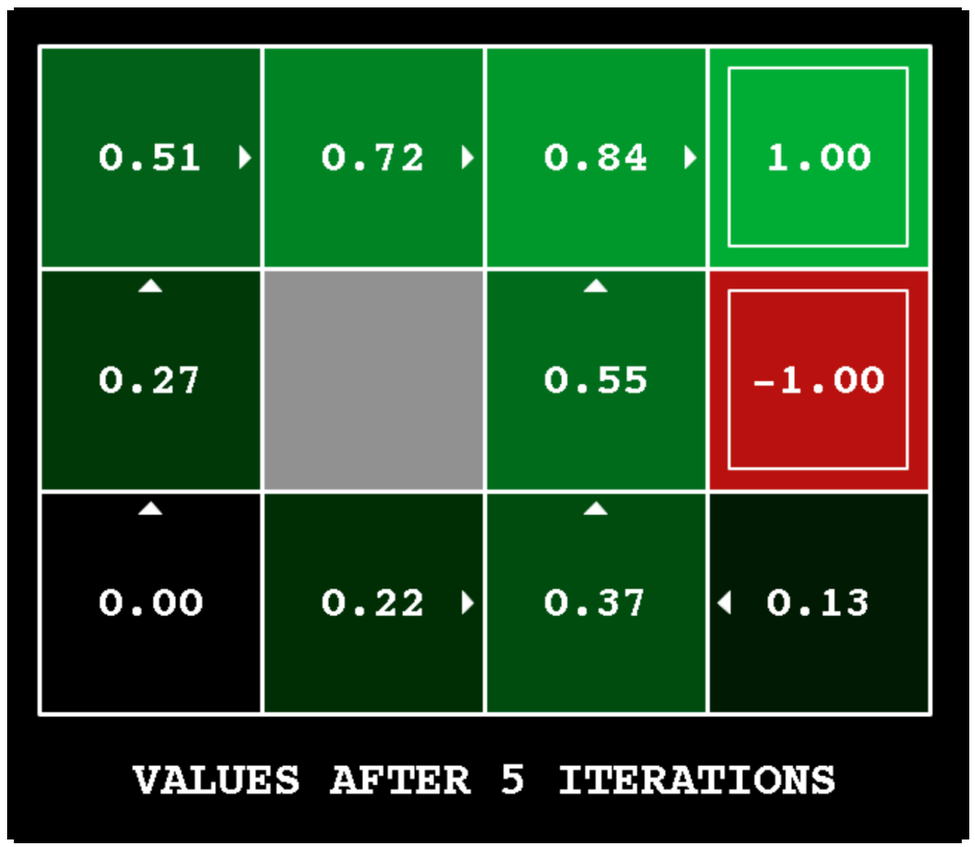
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

noise = 0.2
gamma = 0.9



Finding the optimal policy via Value Iteration

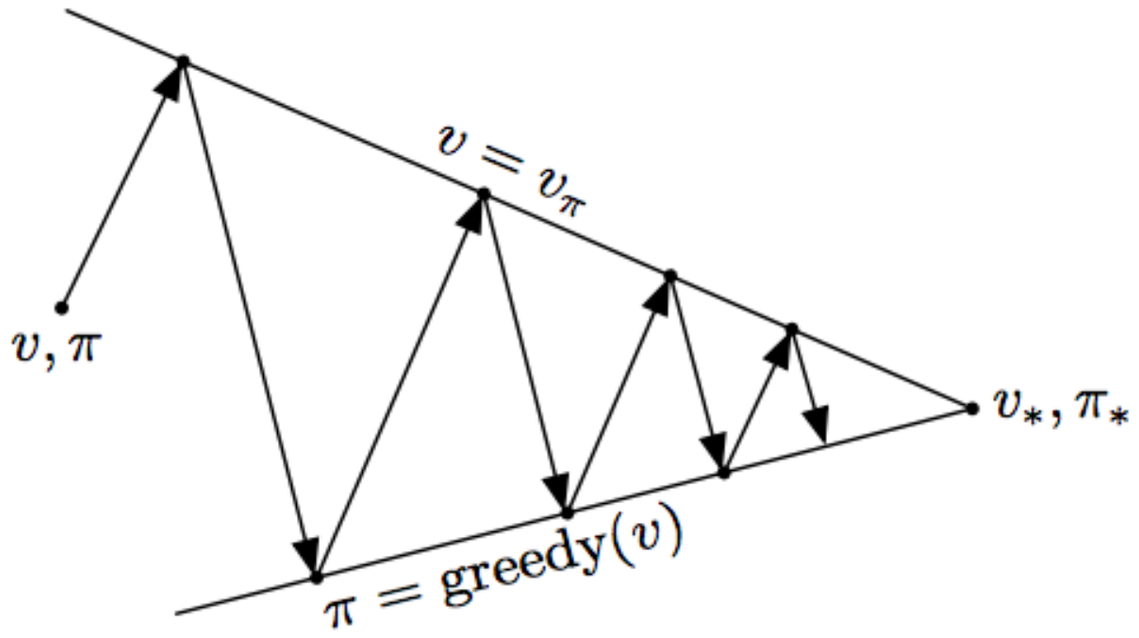
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

noise = 0.2
gamma = 0.9



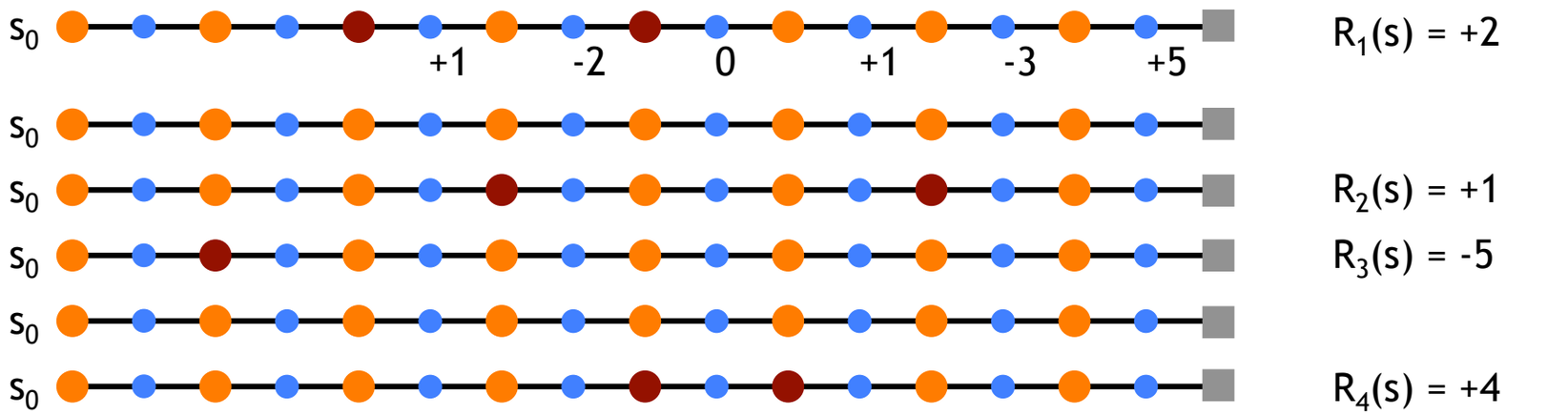
Problem

You need to know a model of the world which is not always available to agents or can’t be clearly specified in advance!

Need to track and represent all the states which can be memory intensive. Unlikely to be solved by people.

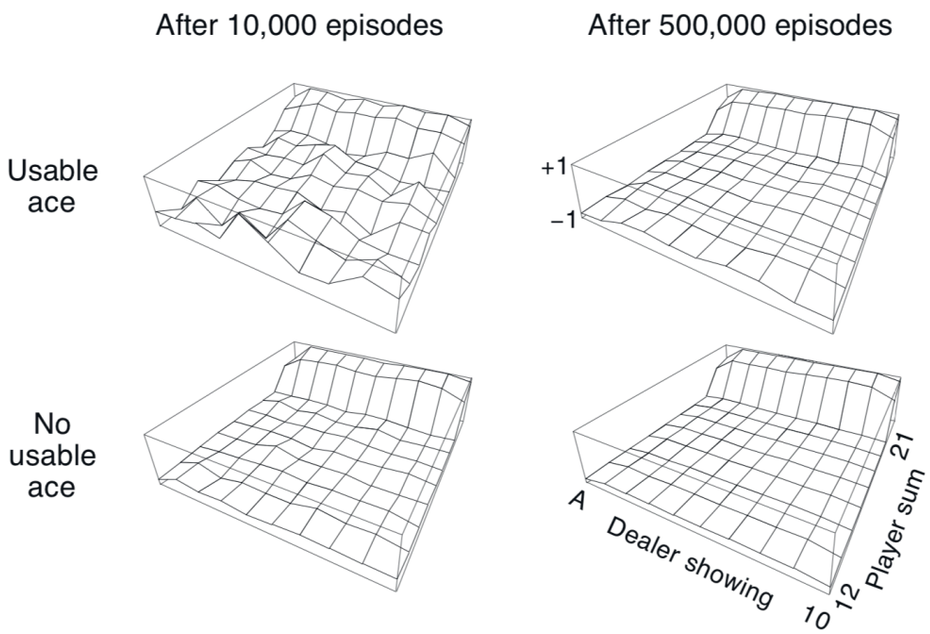
Monte Carlo Methods

- **don't need full knowledge of environment**: just experience, or simulated experience
- but similar to DP: policy evaluation, policy improvement
- **want to estimate V(s):** expected return starting from s and following π. estimate as average of observed returns in state s
- **first-visit MC**: average returns following the first visit to state s



$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

Monte Carlo Methods



- **downside:** no bootstrapping as in dynamic programming
- **upside:** no bootstrapping as in dynamic programming… expense of updating doesn't depend on the total number of states in the problem.

- ***V(s)* not enough for policy improvement:** need exact model of environment
  - **Instead estimate *Q(s,a)*:** $\pi'(s) = \arg\max_a Q^\pi(s,a)$
- **Update after each episode**

$$\pi_0 \rightarrow^E Q^{\pi_0} \rightarrow^I \pi_1 \rightarrow^E Q^{\pi_1} \rightarrow^I \ldots \rightarrow^I \pi^* \rightarrow^E Q^*$$

- **a problem:** greedy policy won't explore all actions! Need to balance explore-exploit

Monte Carlo Methods Summary

- Don’t need model of environment!
    - averaging of sample returns from actual experience or even simulated play (e.g., self-play in two player games, etc…)
- can concentrate on “important” states: don’t need a full sweep of all states because no bootstrapping
- need to maintain **exploration** (EXPLORE-EXPLOIT DILEMMA — next week!)

Summary

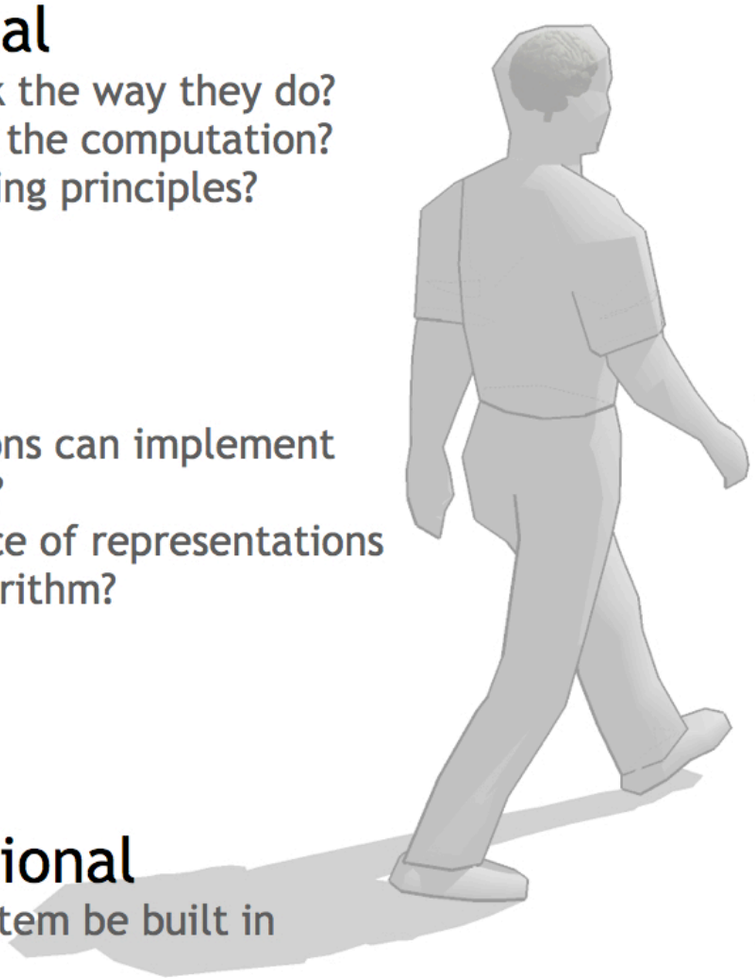
- The goal of the RL agent is to maximize reward over the long term.
- The way this is implemented is through a function which determines the value of various “states” or “situations” under a certain policy
- Once you know how to evaluate a policy, there are a number of ways to actually arrive at optimal policies
    - **Value iteration**: move back and forth from evaluating policy to changing policy to optimize estimated values
    - **Monte Carlo:** Run direct simulations (or experience) forward and tally returns following each state
- **CRITICALLY**, there is more than one way to solve the RL problem depending on if you represent the world model or not.

Five minute break

Next time: Evaluating the world when you don't know anything about it

Three levels of description *(David Marr, 1982)*

Computational
Why do things work the way they do?
What is the goal of the computation?
What are the unifying principles?

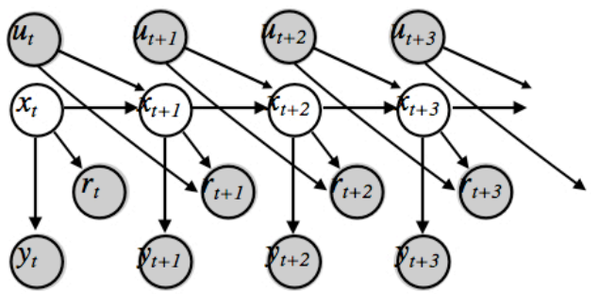


maximize:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

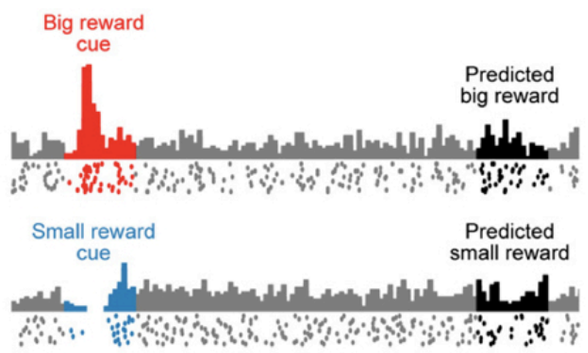
Bellman

Algorthmic
What representations can implement such computations?
How does the choice of representations determine the algorithm?



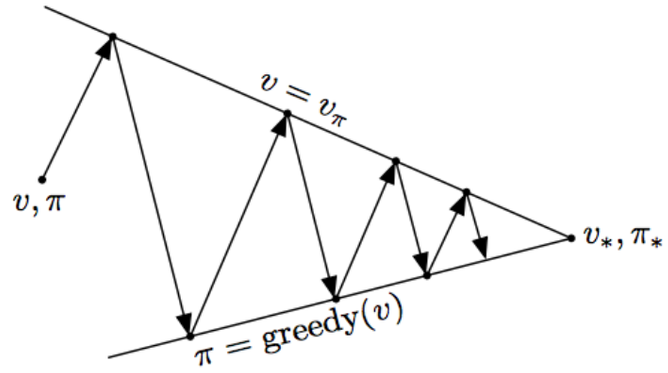
Dynamic programming, TD methods, Monte Carlo

Implementational
How can such a system be built in hardware?
How can neurons carry out the computations?

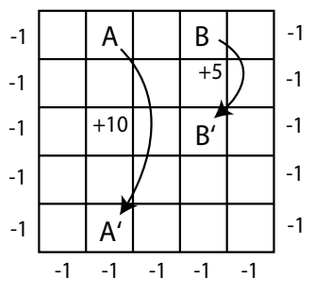


Neural firing patterns, prediction errors, system level neuroscience

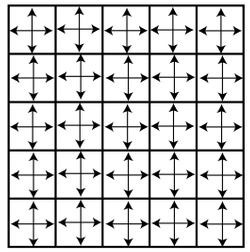
Dynamic Programming/Value iteration



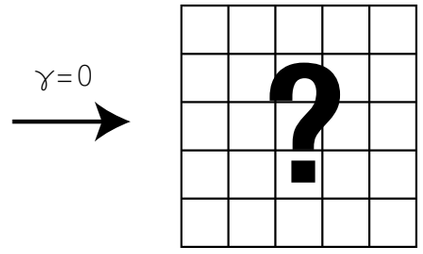
Rewards & State Transitions



Agent's Policy (π)

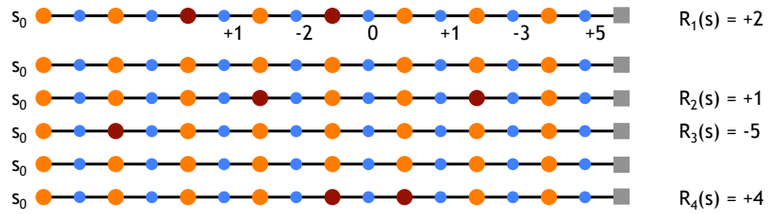


Value Function (V)



- Generally require "model" of environment (i.e., knowledge of state transitions, reward, and policy at at point in environment)
- Curse of dimensionality
- Proveably converges to optimal
- Solution benefits from "bootstrapping"

Monte Carlo



$V^\pi(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

- Does **not** require "model"
- May not even estimate some part of environment
- Converange more sensitive to issues like sufficient exploration
- Solution does not benefit from "bootstrapping"

Blending the ideas….

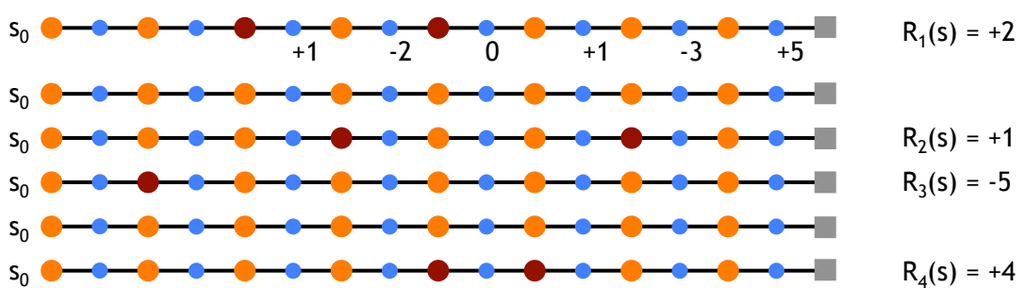
- The first-visit MC algorithm has following steps

  Let R be the return following first visit to state s. Append R to list Returns[s]. V(s) = average(Returns[s])

- Incremental implementation:

$$V(s) = V(s) + \frac{1}{n(s)}[R - V(s)]$$

where n(s) is number of first visits to s.

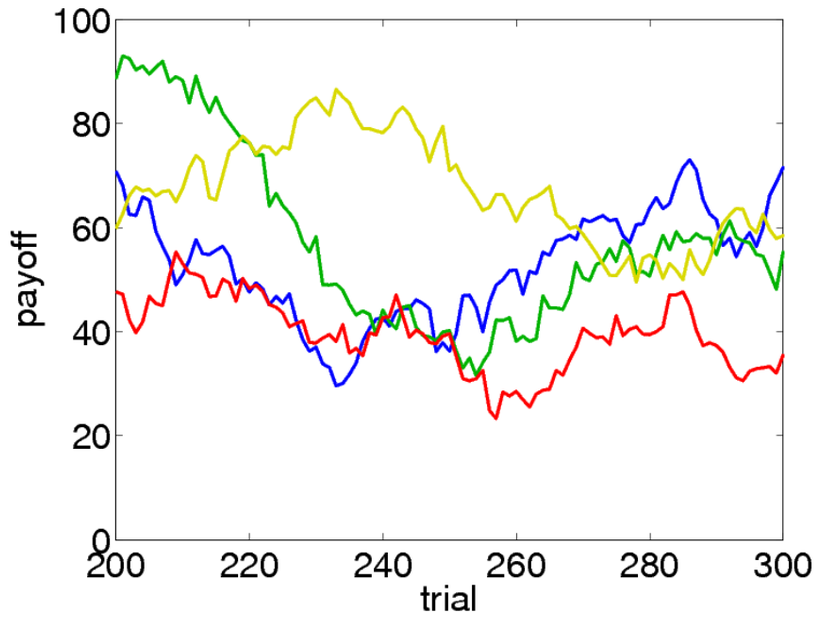


Blending the ideas….

Now consider a constant step size Monte-carlo update:

$$V(s) = V(s) + \alpha[R - V(s)]$$

Why might this be useful?



(hint)

Temporal difference prediction

Policy evaluation is often referred to as a prediction problem: we are trying to predict how much return we'll get from being in state s and following our policy.

Monte carlo incremental update

$$V(s) = V(s) + \alpha[R - V(s)]$$

target: *actual* return from s_t to end of episode

Still have to wait until episode terminates…

Temporal Difference update TD(0):

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

target: *estimate* of the return… using BOOTSTRAPPING!

Evaluating the world when you don't know anything about it

$\alpha = 0.9 \quad \gamma = 1 \quad \pi$ - random

Initialize

$c \rightarrow f \qquad V(c) \leftarrow 0 + 0.9[100 + 0 - 0] = 90$

Evaluating the world when you don't know anything about it

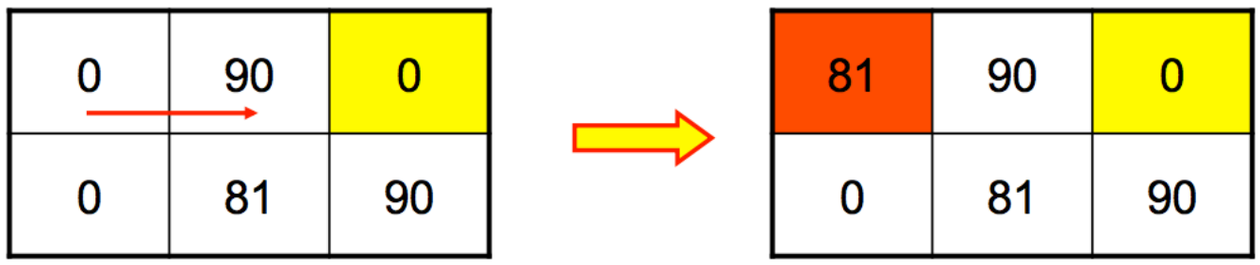
$a \rightarrow b \quad V(a) \leftarrow 0 + 0.9[0 + 0 - 0] = 0$

$e \rightarrow f \quad V(e) \leftarrow 0 + 0.9[100 + 0 - 0] = 90$

Evaluating the world when you don't know anything about it

$b \rightarrow c \quad V(b) \leftarrow 0 + 0.9[0 + 90 - 0] = 81$

$d \rightarrow e \quad V(d) \leftarrow 0 + 0.9[0 + 90 - 0] = 81$



Evaluating the world when you don't know anything about it

$a \rightarrow b \quad V(a) \leftarrow 0 + 0.9[0 + 81 - 0] \approx 73$

$c \rightarrow f \quad V(c) \leftarrow 90 + 0.9[100 + 0 - 90] = 99$

Evaluating the world when you don't know anything about it

$e \rightarrow f \quad V(e) \leftarrow 90 + 0.9[100 + 0 - 90] = 99$

$c \rightarrow b \quad V(c) \leftarrow 99 + 0.9[0 + 81 - 99] \approx 83$

Evaluating the world when you don't know anything about it

$\gamma = 0.9$

52	66	0
49	57	76

bellman solution!

Temporal difference prediction

Temporal Difference update TD(0):

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Bellman recurrence relation

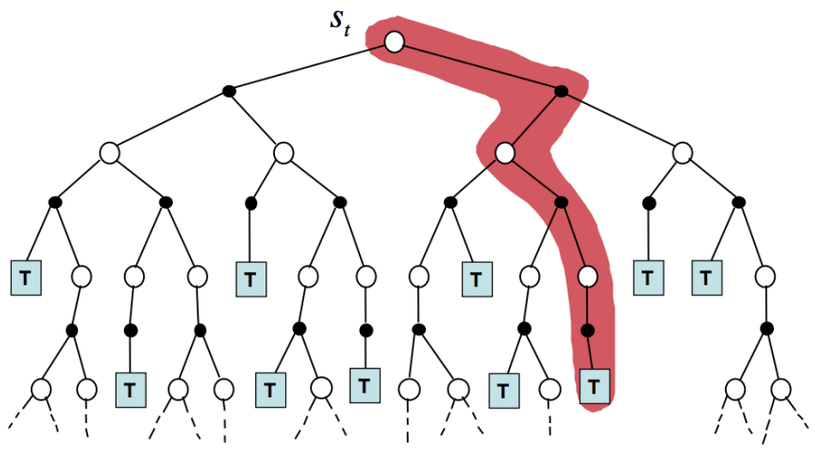
$$V^{\pi}(s) = E_{\pi}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\}$$

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'}[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')]$$

Simple Monte Carlo

$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

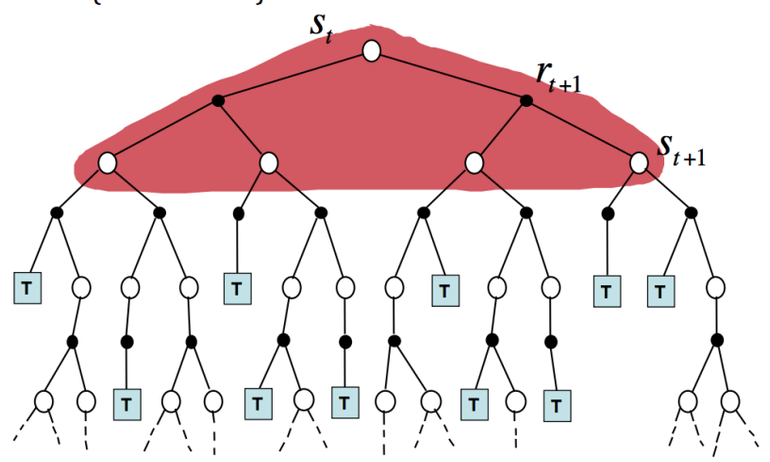
where R_t is the actual return following state s_t.



Monte Carlo uses an estimate of the actual return.

Dynamic Programming

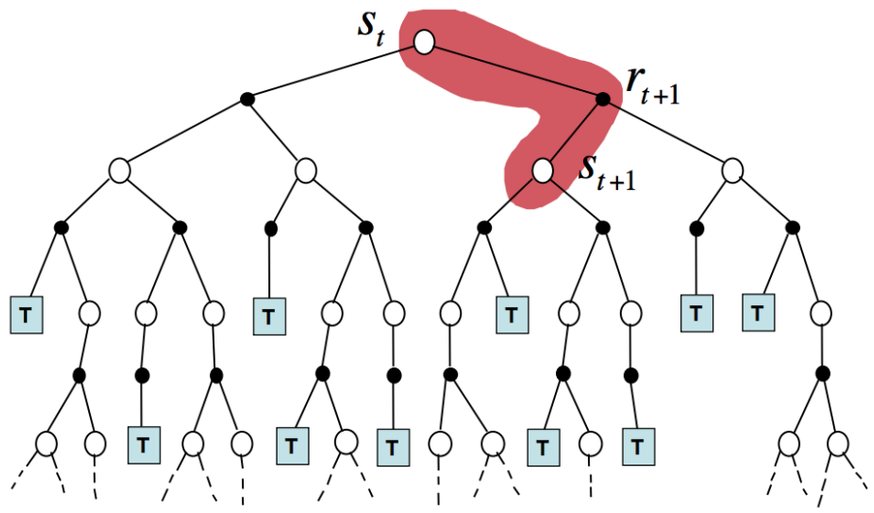
$$V(s_t) \leftarrow E_\pi\left\{r_{t+1} + \gamma V(s_t)\right\}$$



The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because V^π is not known and the current estimate is used instead.

Simplest TD Method

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$



TD samples the expected value and uses the current estimate of the value.

Advantages of TD learning methods

- Don’t need a model of the environment
- Online and incremental so can be fast (don’t need to wait until end of episode as in MC)
- Update based on actual experience (r_{t+1})
- Converges to the true values if you lower step size/learning rate as learning continues
- TD bootstraps: it updates estimate based on other estimates (like DP/value iteration).
- TD samples: updates are based on a single run/path through the state space (like MC)

TD(0) is still kind of slow: Eligibility traces

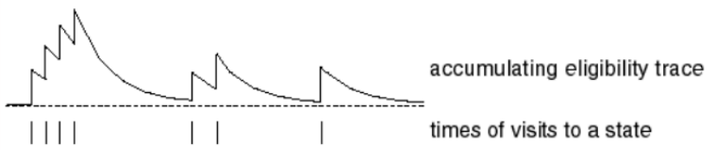
- The benefits of bootstrapping only extend between adjacent states (s to s'). As a result you have to cross that particular state transition many times for the value to "propogate" backwards

- New variable called *eligibility trace*. The eligibility trace for state at time is denoted

$$e_t(s) \in \Re^+$$

On each step, decay all traces by $\gamma\lambda$ and increment the trace for the current state by

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$



γ discount rate

λ trace-decay parameter

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

$$\Delta V_t(s) = \alpha \delta_t e_t(s)$$

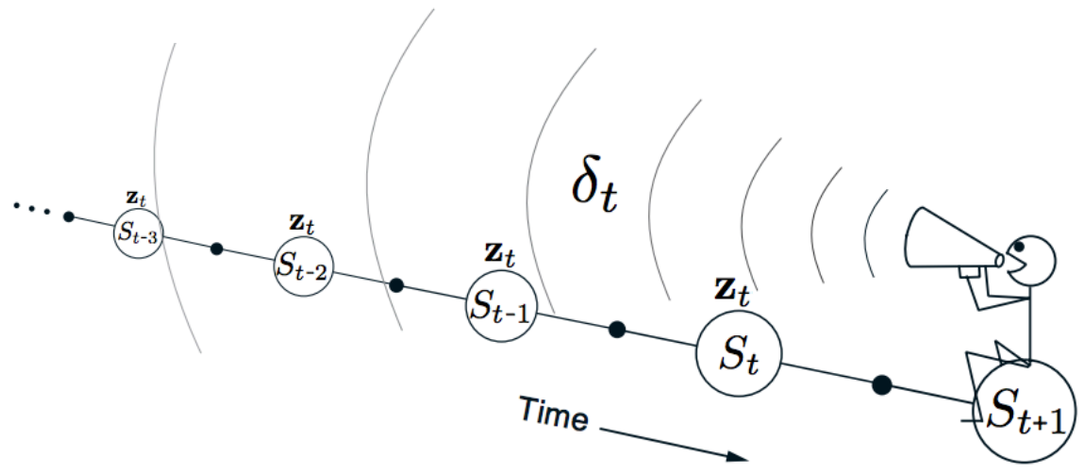
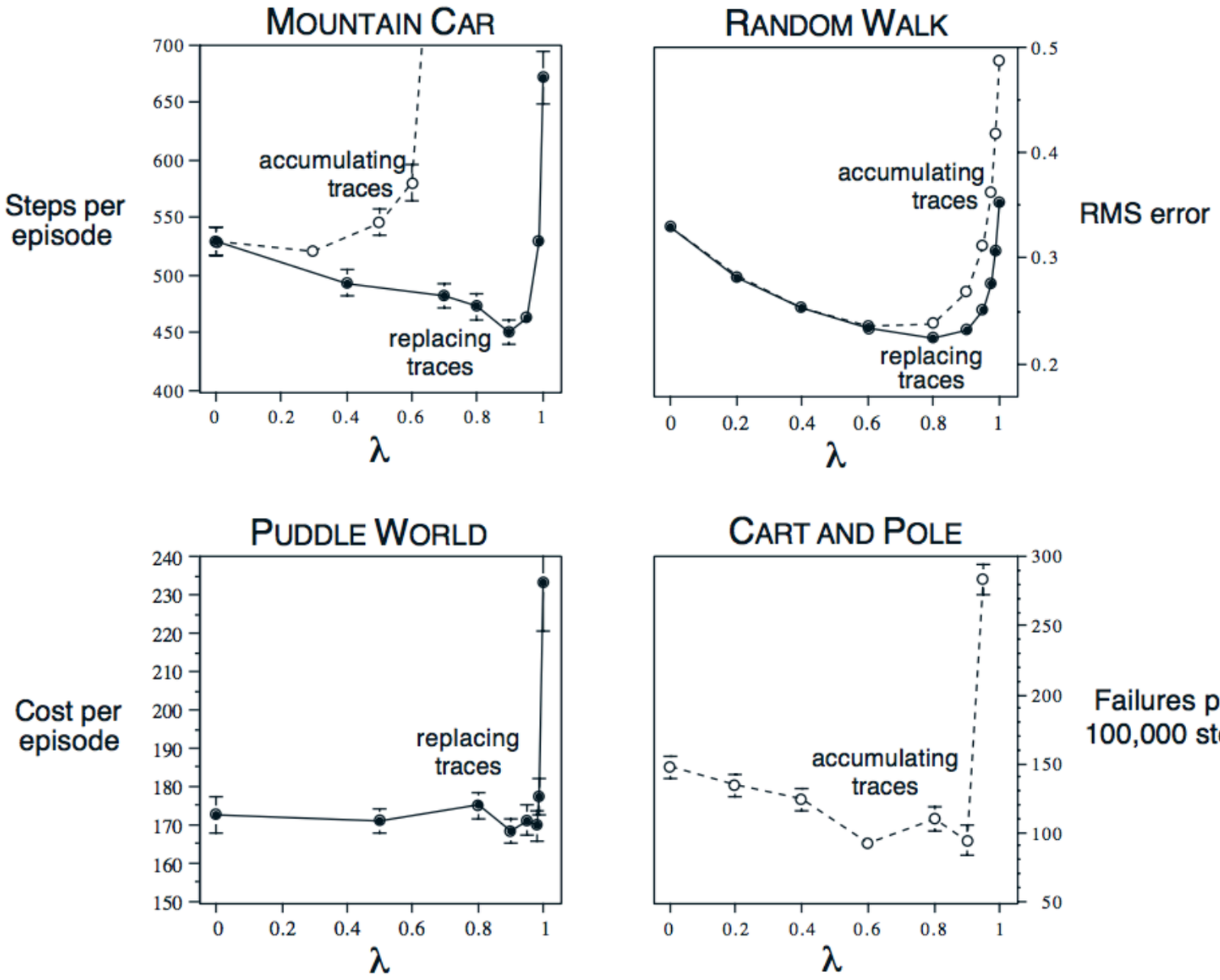


Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error combined with the current eligibility traces of past events.

TD(0) is still kind of slow: Eligibility traces



intermediate values empirically work best!

Learning for control: Learning Q-values

- Learning the value of different states can be a little obtuse because what you really want to do is learn how to act!
- Instead can make sense to learn $Q^{\pi}(s, a)$

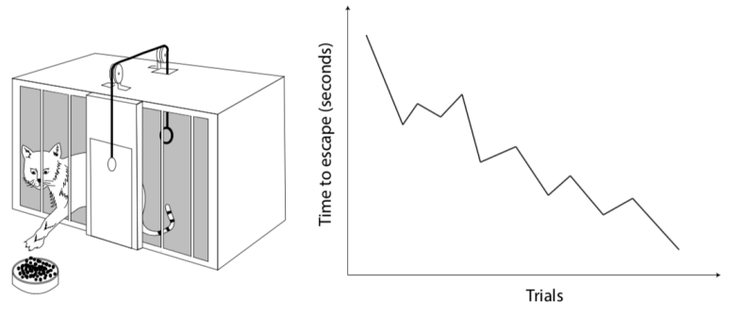


Figure 1: *Left*: An illustration of Thorndike's puzzle box experiments. *Right*: The time recorded to escape the box is reduced over repeated trials as the cat becomes more efficient at selecting the actions which lead to escape.

SARSA update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$$

- Choose a policy and estimate the Q-values using SARSA rule. Change policy toward greediness with respect to Q values.
- Converges with probability 1 to optimal policy and Q-value if you visit all state-action pairs infinitely many times and the policy converges to be a greedy policy.
- Easy to know what to do! Just choose the action with highest Q value!

Learning for control: Learning Q-values

SARSA update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$$

- Initialise $Q(s, a)$
- Repeat `many times`
  - Pick s, a
  - Repeat `each step to goal`
    - Do a, observe r, s'
    - Choose a' based on $Q(s', a')$ ϵ-`greedy`
    - $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    - $s = s'$, $a = a'$
  - Until s terminal (where $Q(s', a') = 0$)

Use with policy iteration, i.e. change policy each time to be greedy wrt current estimate of Q

sarsa is known as an on-policy learning rule…

Learning for control: Learning Q-values

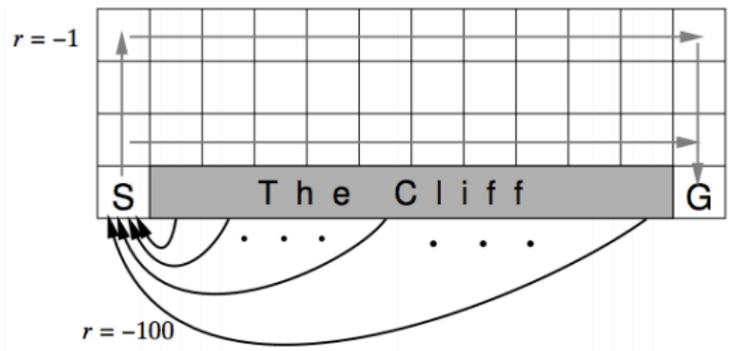
Q-learning update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)]$$

- Initialise $Q(s, a)$
- Repeat `many times`
  - Pick s `start state`
  - Repeat `each step to goal`
    - Choose a based on $Q(s, a)$ `$\epsilon$-greedy`
    - Do a, observe r, s'
    - $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    - $s = s'$
  - Until s terminal

Q-learning is known as an off-policy learning rule… always update Q value with maximally best action in next state, even if you won't necessarily take that step yourself.

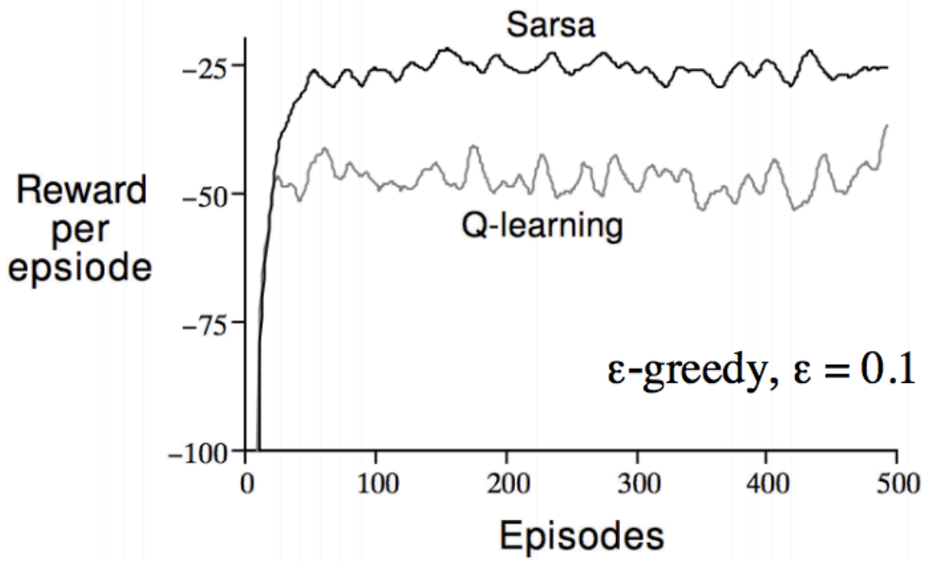
Q-learning versus SARSA (Cliffwalking)



safe path

optimal path

Reward is on all transitions -1 except those into the the region marked "The Cliff."

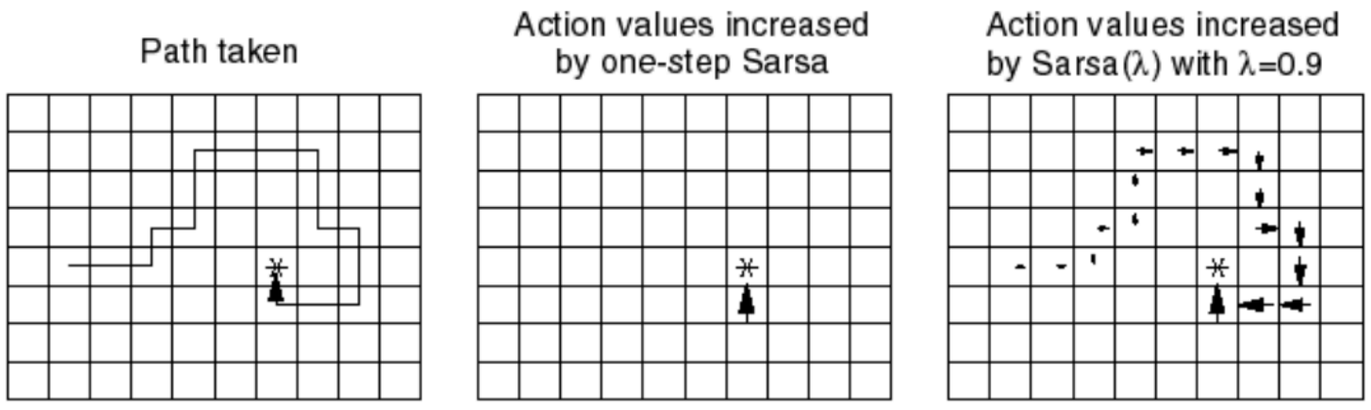


Q-learning learns quickly values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection.

Sarsa takes the action selection into account and learns the longer but safer path through the upper part of the grid.

If ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

SARSA(lambda)



- With one trial, the agent has much more information about how to get to the goal
  - not necessarily the *best* way
- Can considerably accelerate learning

Three levels of description *(David Marr, 1982)*

Computational

Why do things work the way they do?
What is the goal of the computation?
What are the unifying principles?

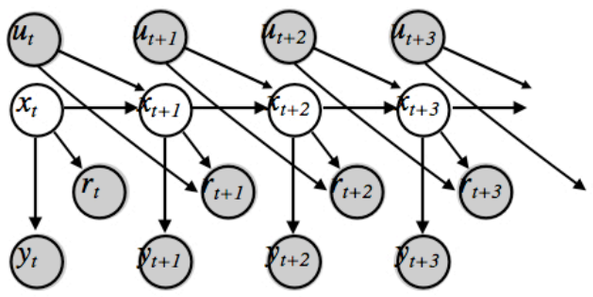
maximize:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

Bellman

Algorthmic

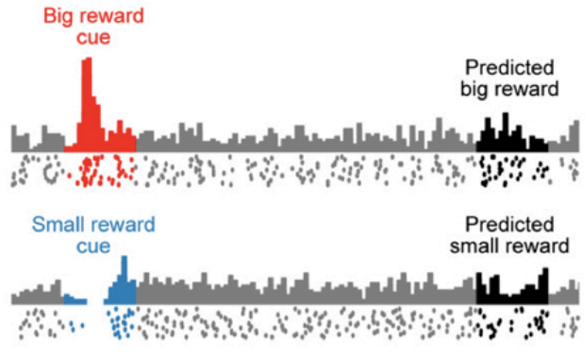
What representations can implement such computations?
How does the choice of representations determine the algorithm?



Dynamic programming, TD methods, Monte Carlo

Implementational

How can such a system be built in hardware?
How can neurons carry out the computations?



Neural firing patterns, prediction errors, system level neuroscience

Next time

- The Explore/Exploit Dilemma
- Function approximation/generalization
- Model-based versus Model-free RL