Lecture 4: Computational Cognitive Modeling

Reinforcement Learning (pt. 1)

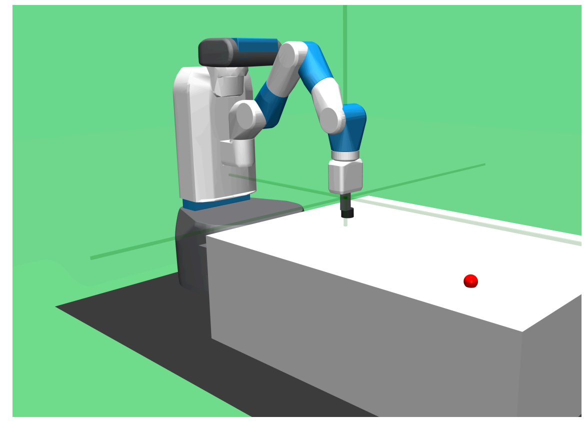
course website:
https://brendenlake.github.io/CCM-site/

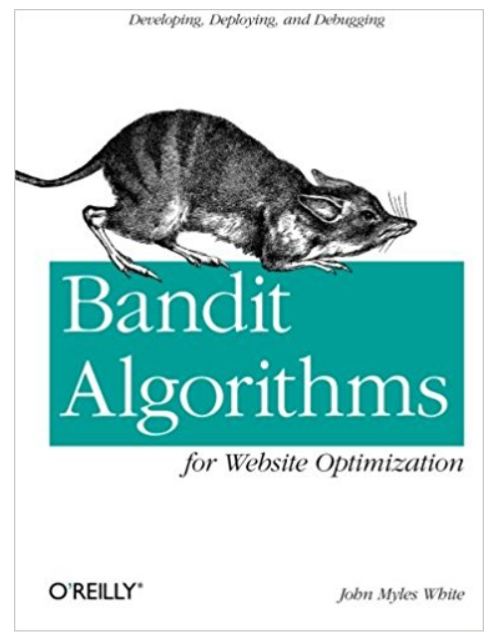
Types of learning (from a ML perspective)

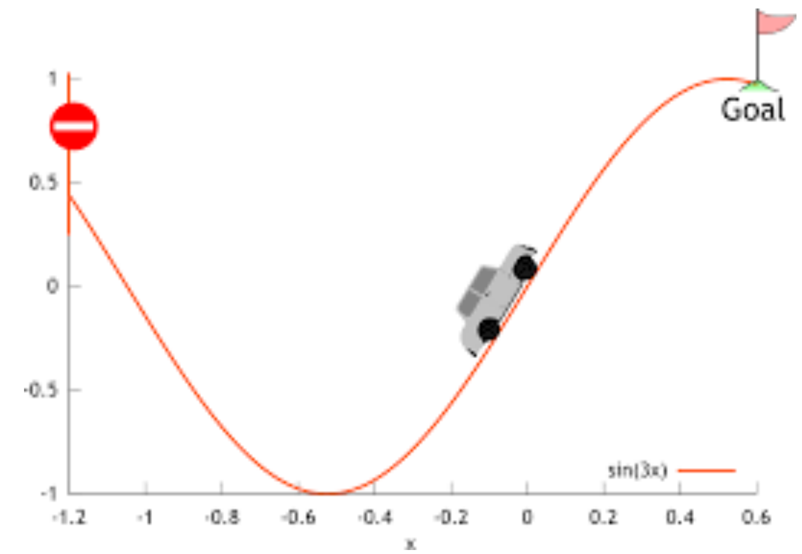
- **Supervised learning**
  - Works by instructing the learning system what output to give for each input. Corrective feedback that is diagnostic ("this not that").
- **Unsupervised learning**
  - No feedback. Works by detecting the correlational and covariation structure of the input to find generalizable patterns.
- **Reinforcement learning**
  - Evaluating feedback (good) but not corrective. Goal is to take actions or make decisions in order to earn higher rewards.

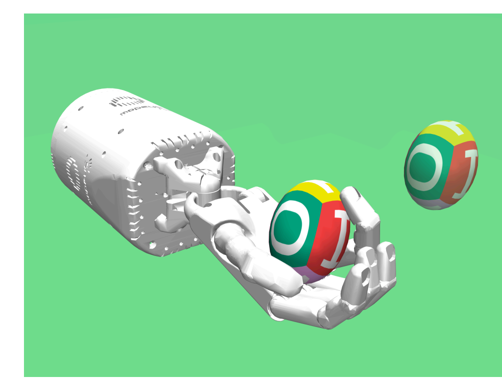
Example applications: Dynamic control problems

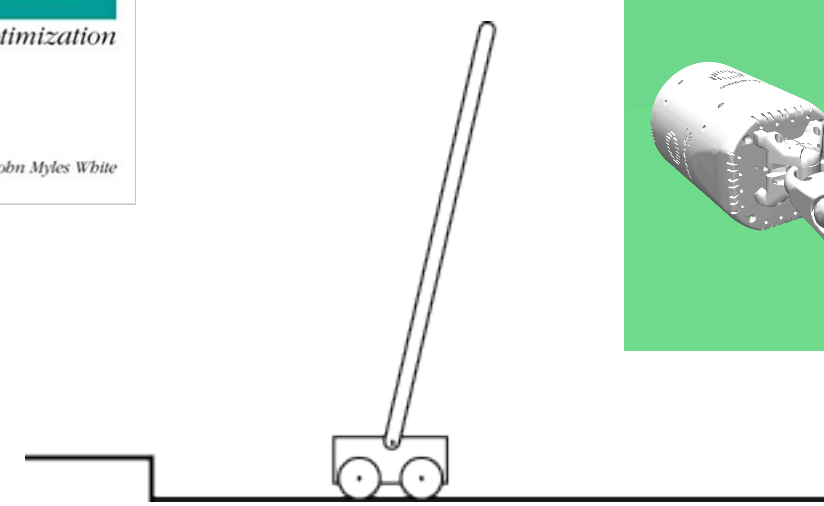
- Autonomous helicopter flight
- Robot legged locomotion
- Cell-phone network routing
- Marketing strategy selection
- Factory control
- Efficient web-page indexing
- A/B testing

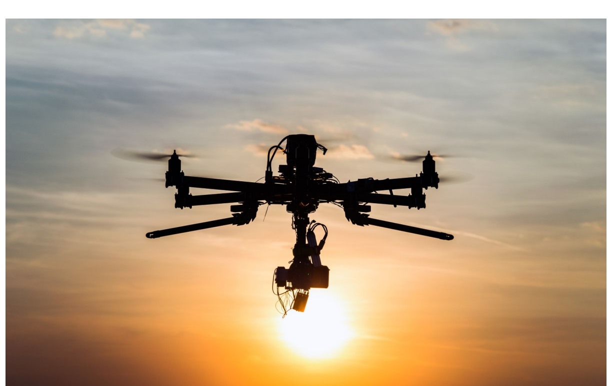




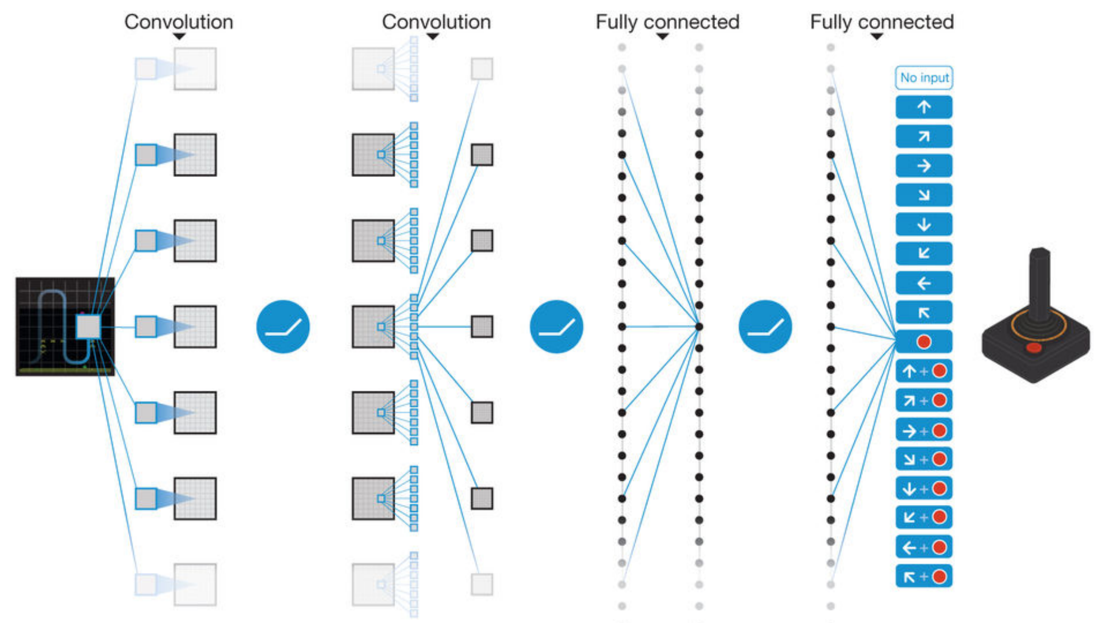


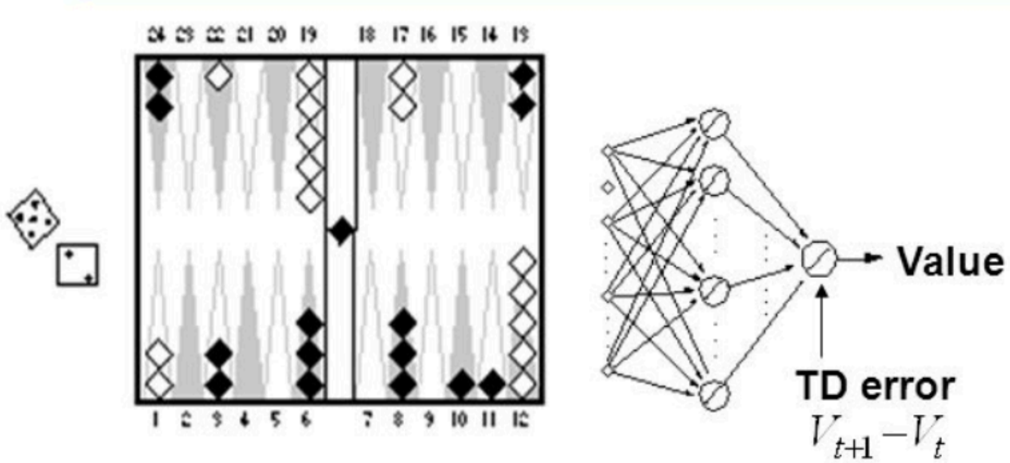






Example applications: Sequential decision making problems, games





Tesauro, 1992–1995

Action selection by 2–3 ply search

Reinforcement Learning

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

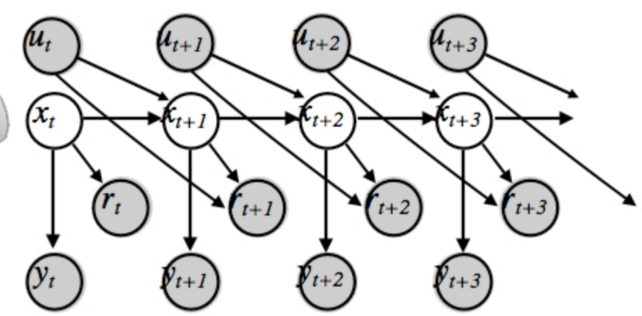
Computational
Why do things work the way they do?
What is the goal of the computation?
What are the unifying principles?

maximize:

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

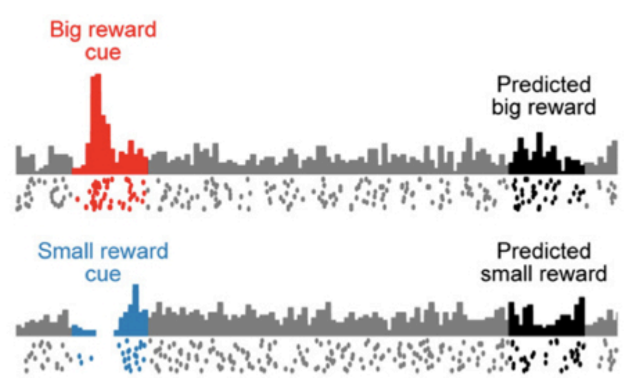
Bellman

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



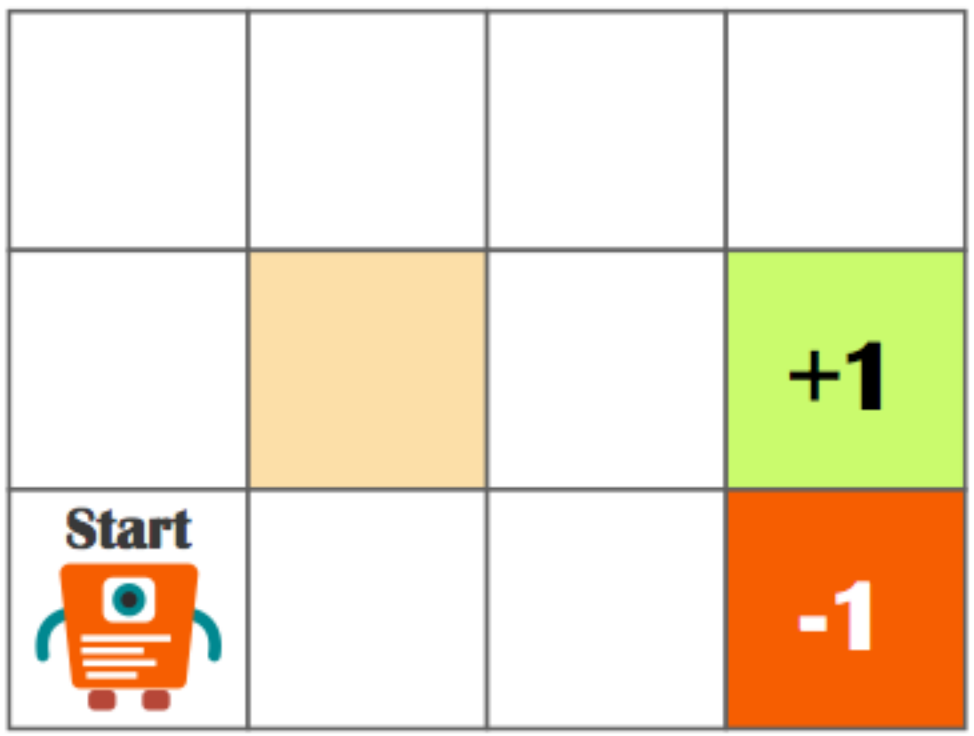
Dynamic programming, TD methods, Monte Carlo

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



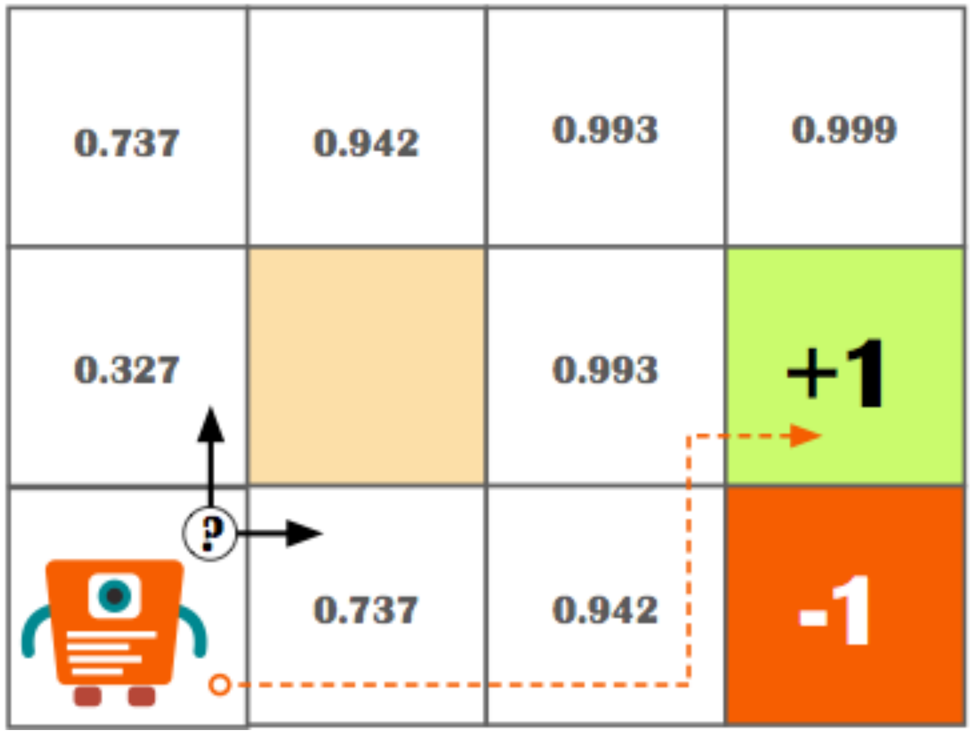
Neural firing patterns, prediction errors, system level neuroscience

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?**

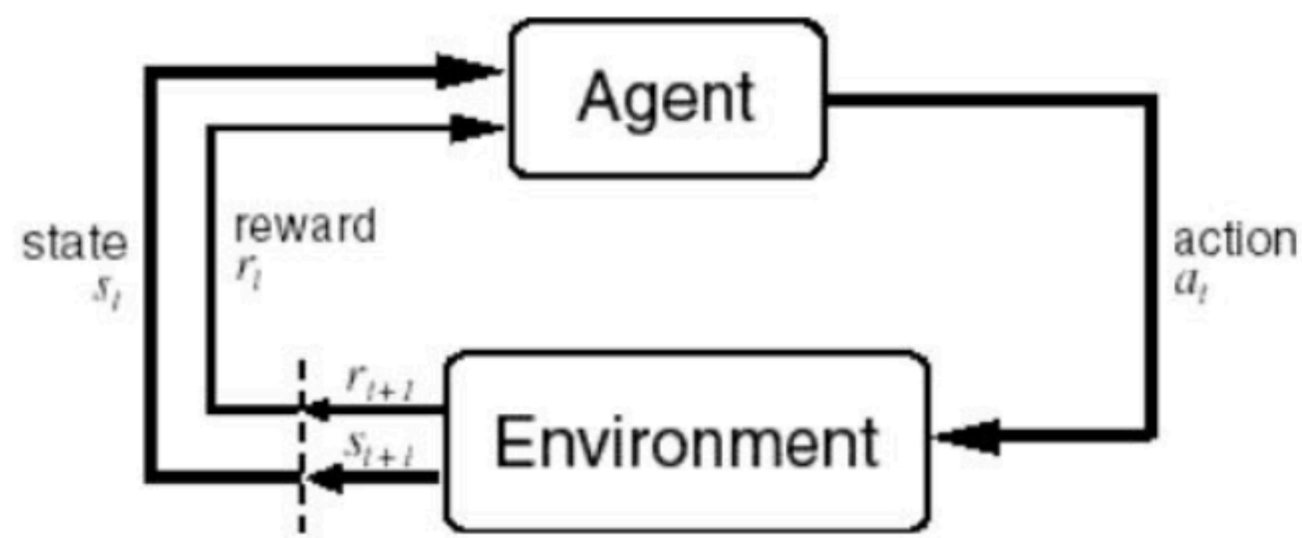
Canonical example



- 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

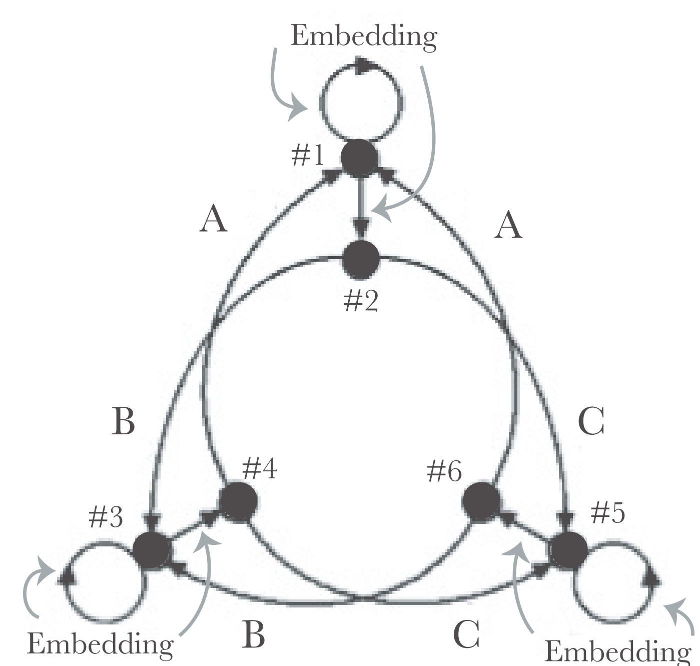
- The environment
- Reward function
- Policy
- Value function
- Model of the environment



The environment

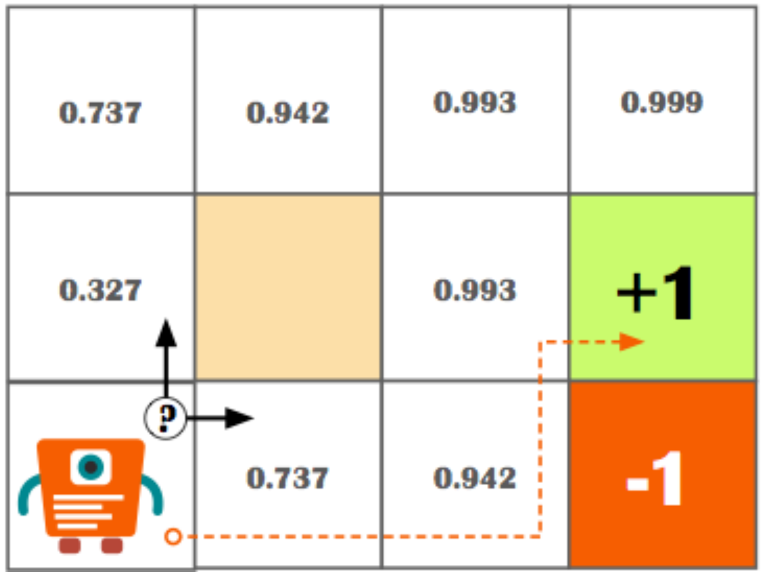
- Modeled as a Markov Decision Process
- At its essence means that the system is defined by the one-step dynamics
- Simply put, the distribution of future states and rewards depend only on where you are now and what action you take

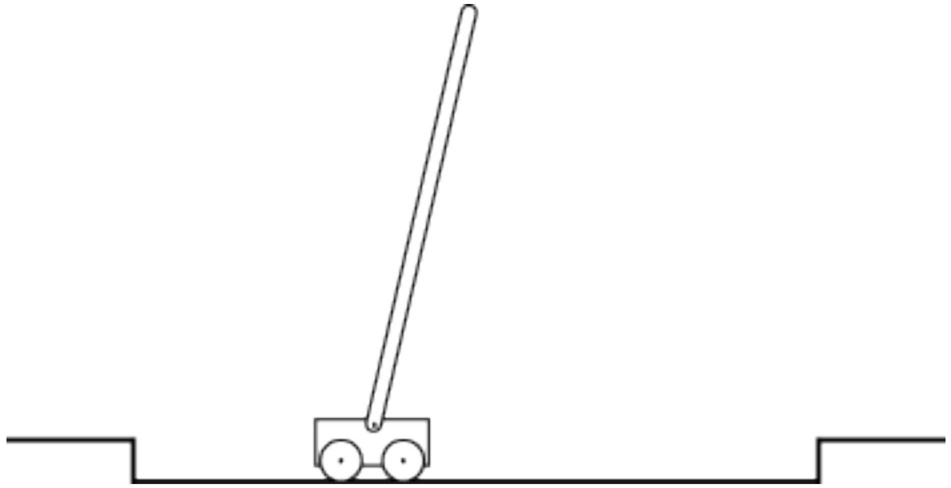
$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}$$



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?





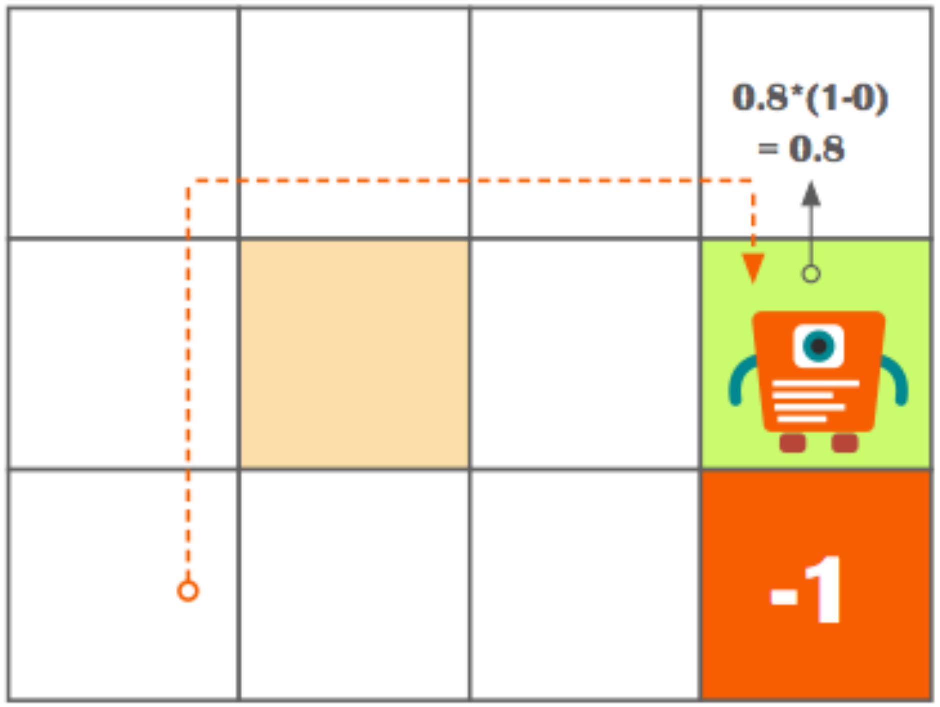
Reward Function

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

- 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

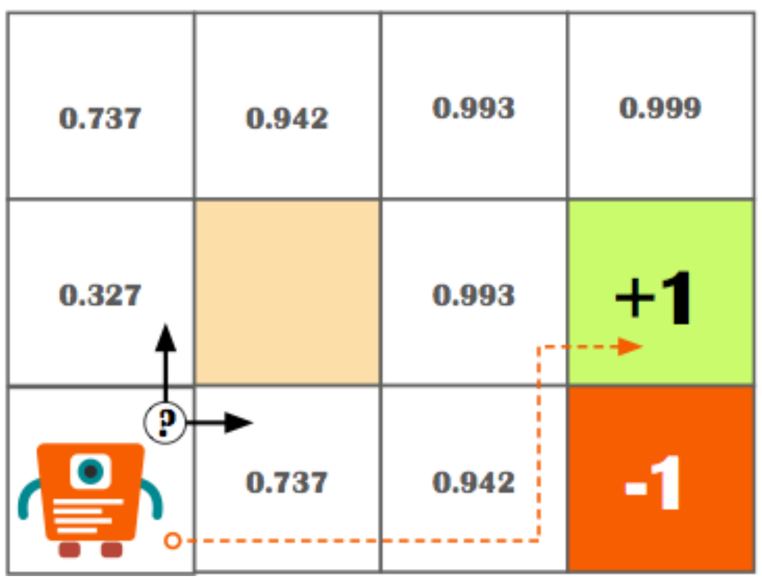
Policy

- 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

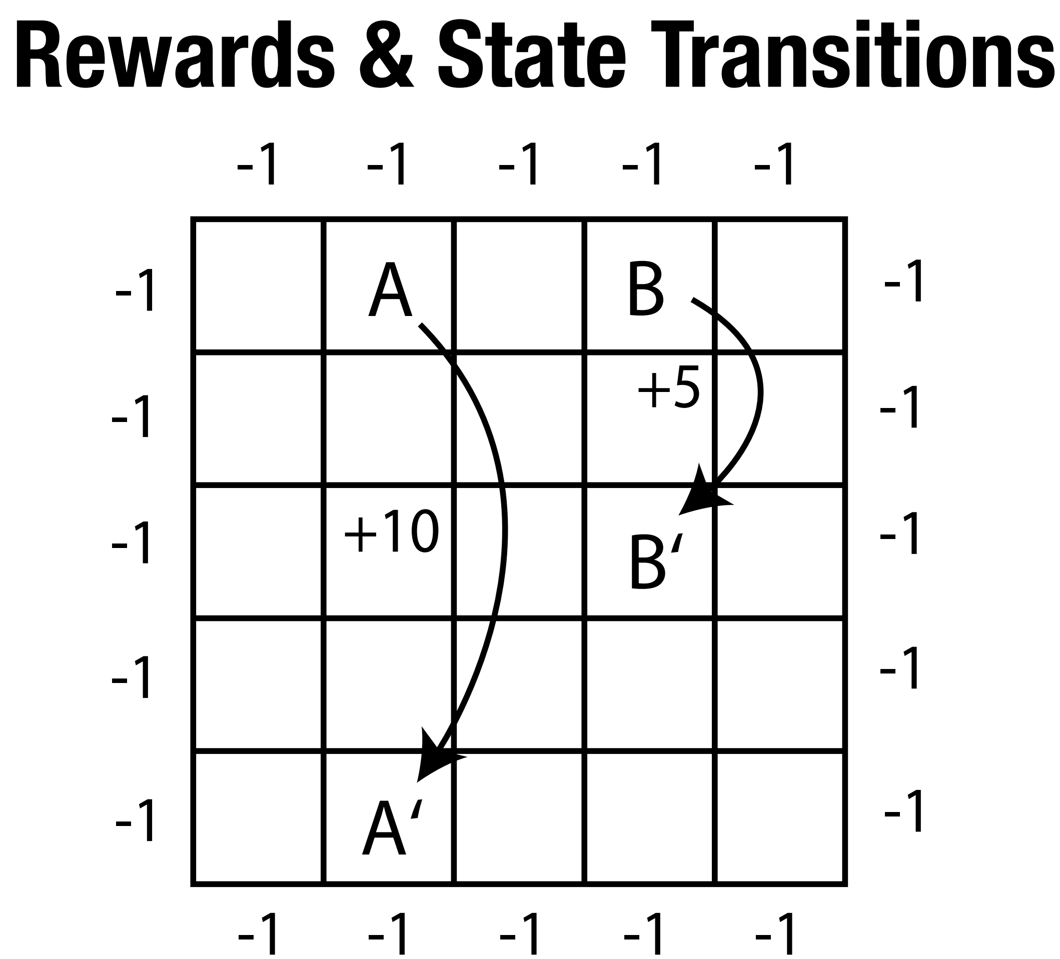
Model of Environment

- 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!!)

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

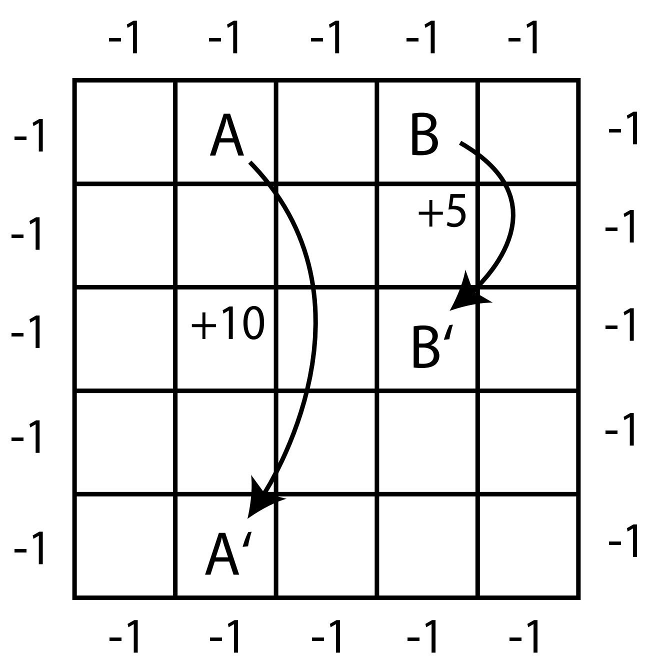
$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

An example: Behaving optimally in a known world

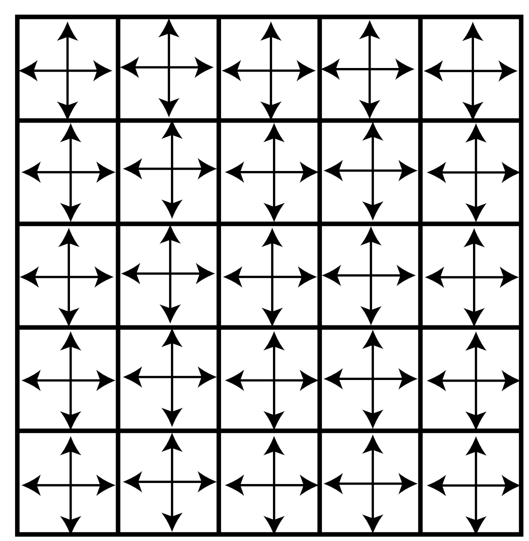


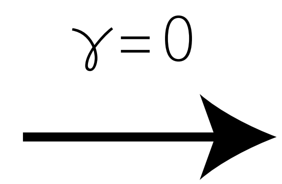
An example: Behaving optimally in a known world

Rewards & State Transitions

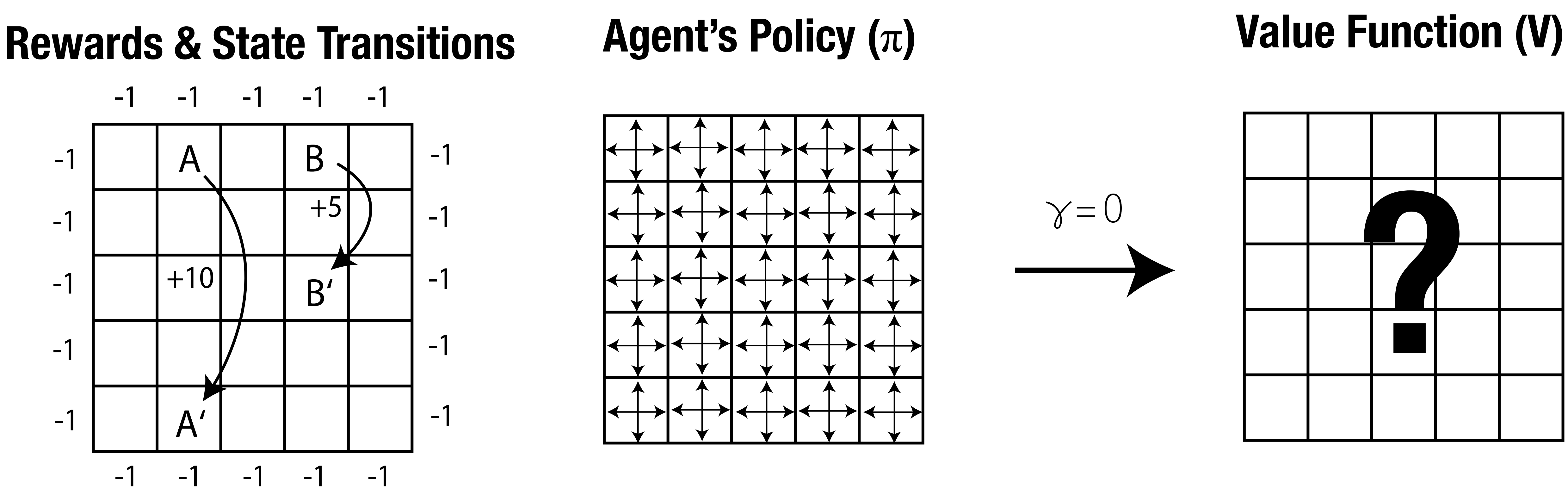


Agent's Policy (π)



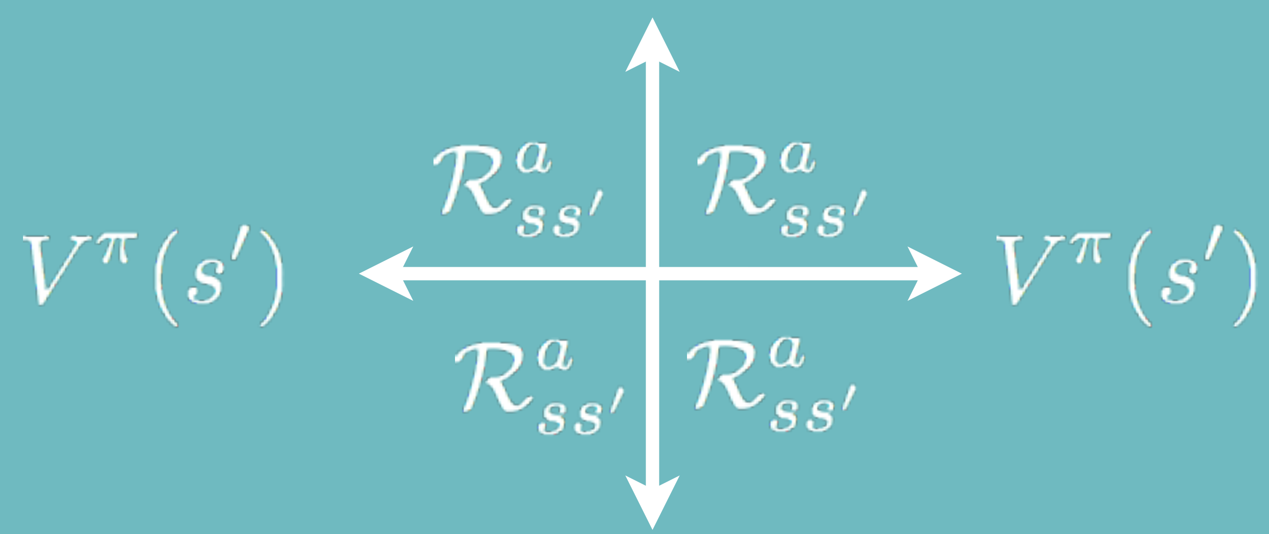


An example: Behaving optimally in a known world



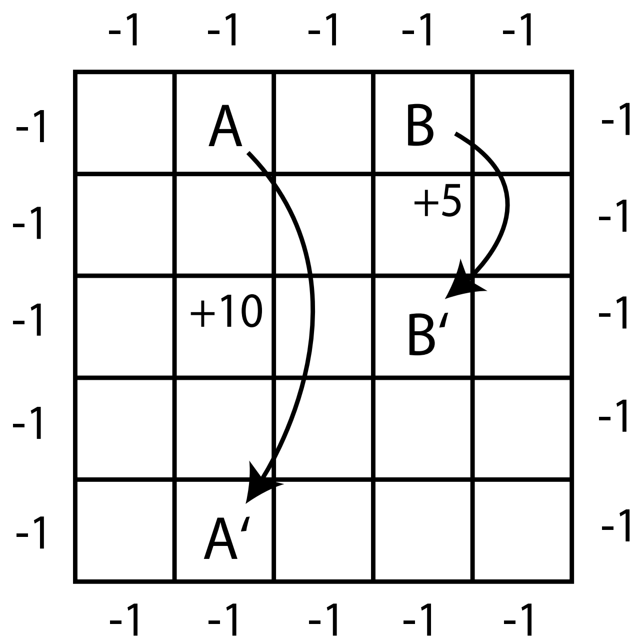
Bellman Equation

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

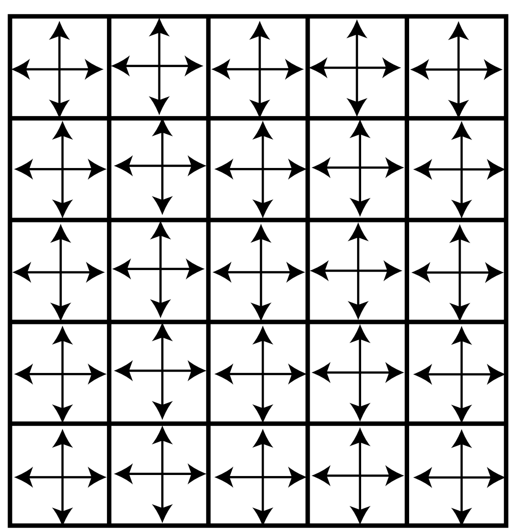


- If we know the function $\mathcal{P}^{a}_{ss'}$ and $\mathcal{R}^{a}_{ss'}$ (i.e. have perfect knowledge of the environment), we can easily solve for $V^{\pi}(s)$ by simply solving the systems of equations under our policy

Rewards & State Transitions



Agent's Policy (π)

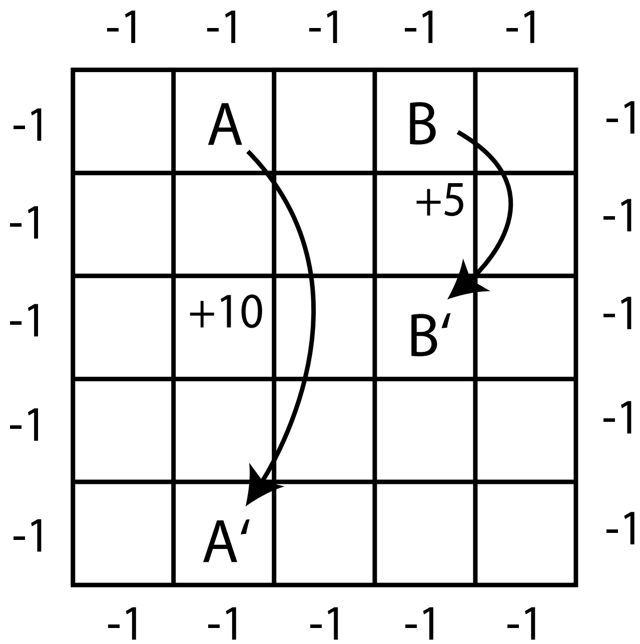


$\gamma = 0$

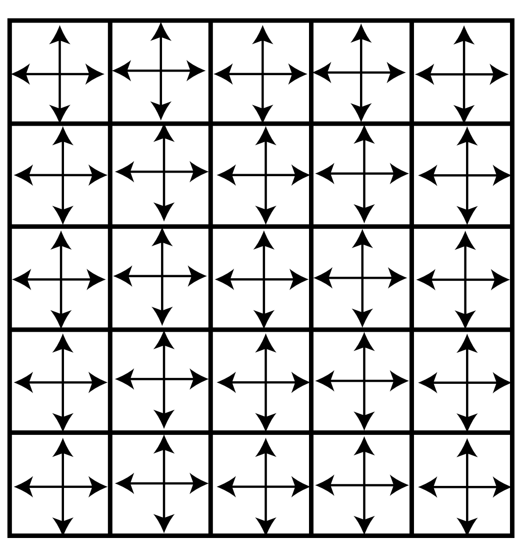
Value Function (V)

-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



Agent's Policy (π)

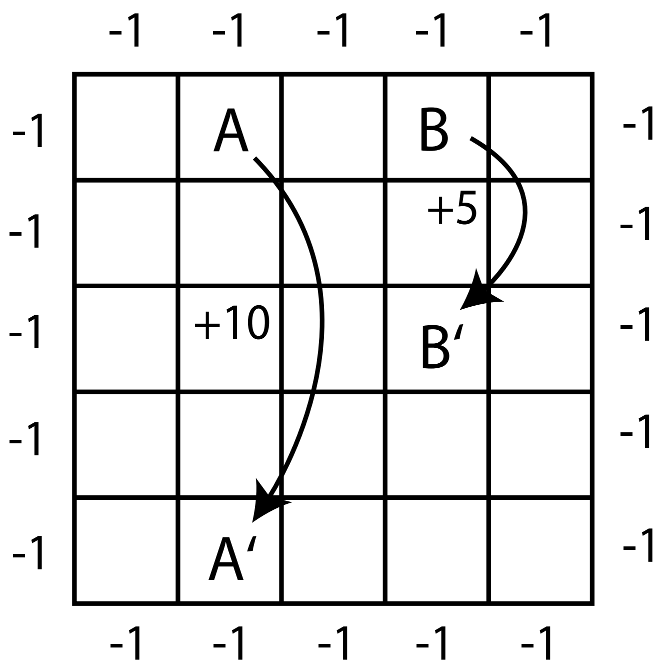


$\gamma = 0.9$

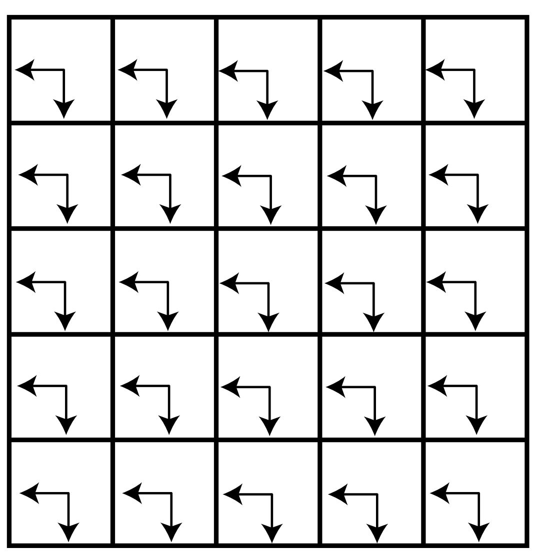
Value Function (V)

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

Iterative 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

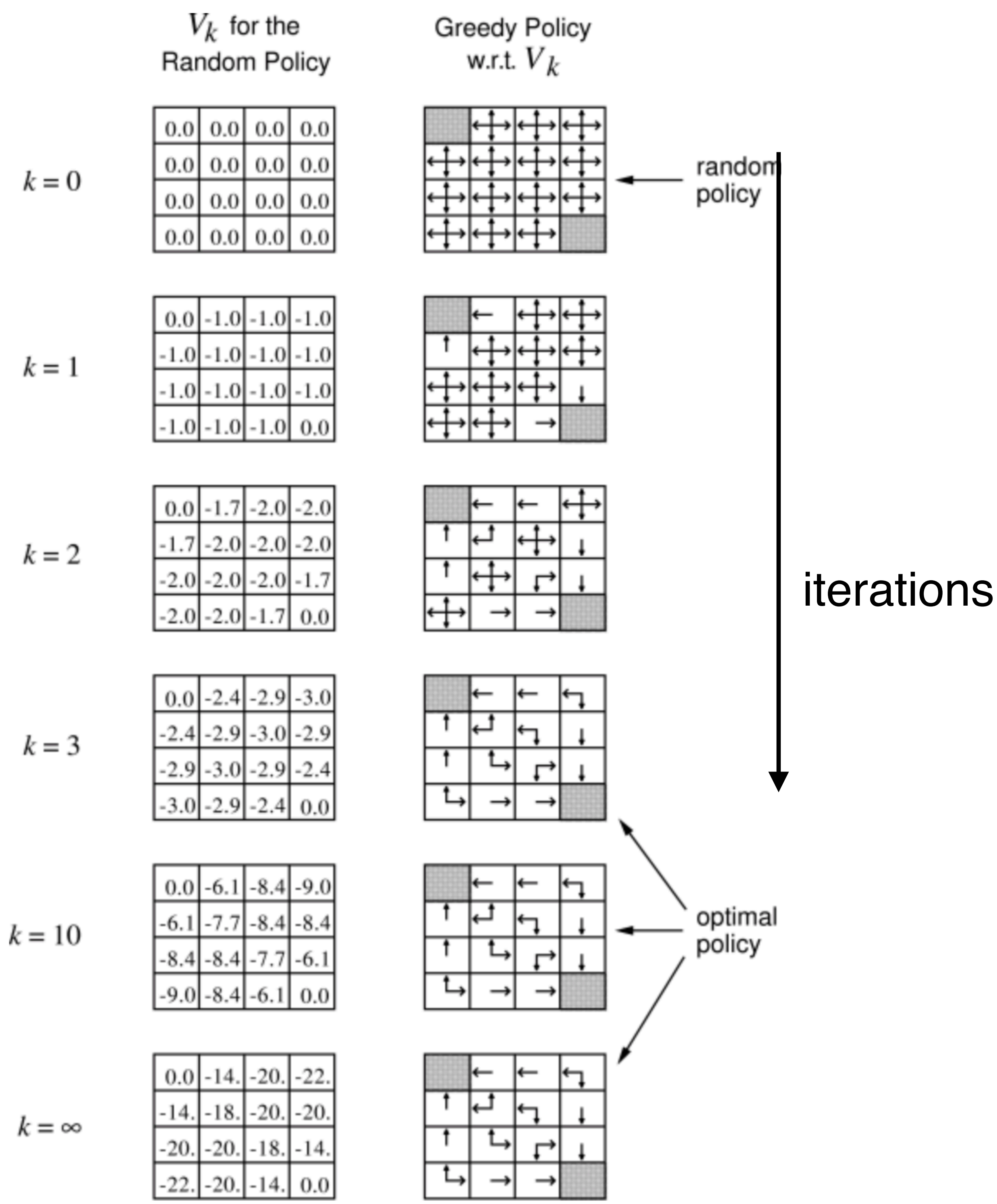
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

iterations

- Generally this can be solved in one step as a system of linear equations with |S| unknowns (the value of each state is defined in terms of all the other variables and so you have S variables and S 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.

Input π, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

Policy Improvement



- Move the current policy (which might be suboptimal) to be greedy with respect to the current value function.
- There is a policy improvement theorem that can prove that moving the policy in this way always makes the policy as good as or better than the original policy (see http://www.incompleteideas.net/book/first/ebook/node42.html)

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a E\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\right\} \\ &= \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right], \end{aligned} \quad (4.9)$$

Finding the optimal policy via Policy Iteration

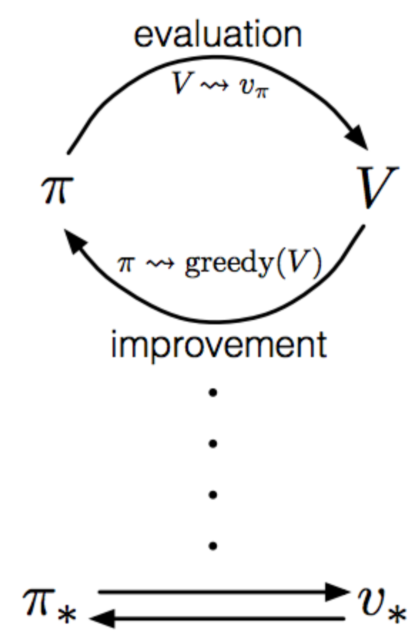
$$\pi_0 \xrightarrow{\text{E}} V^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} V^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} V^*,$$

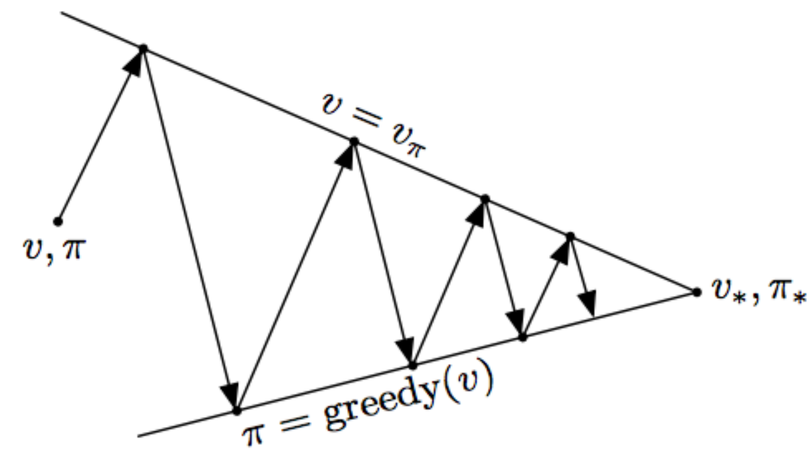
1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ For each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad b \leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s') \right]$
   $\quad$ If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

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





Problems

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.

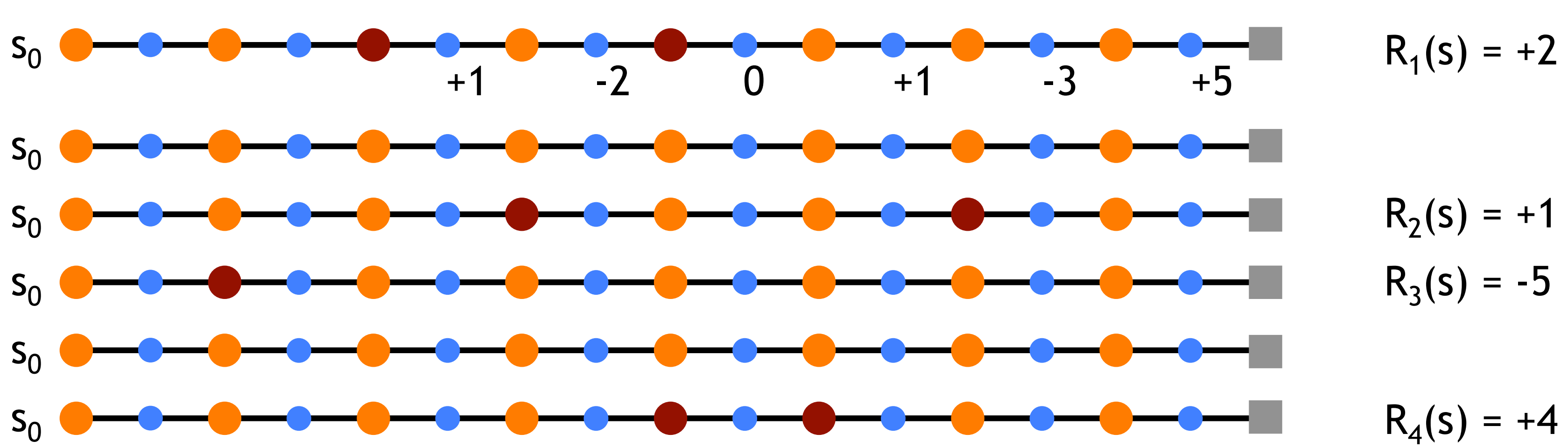
Advantages

Bootstrapping!

Input π, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

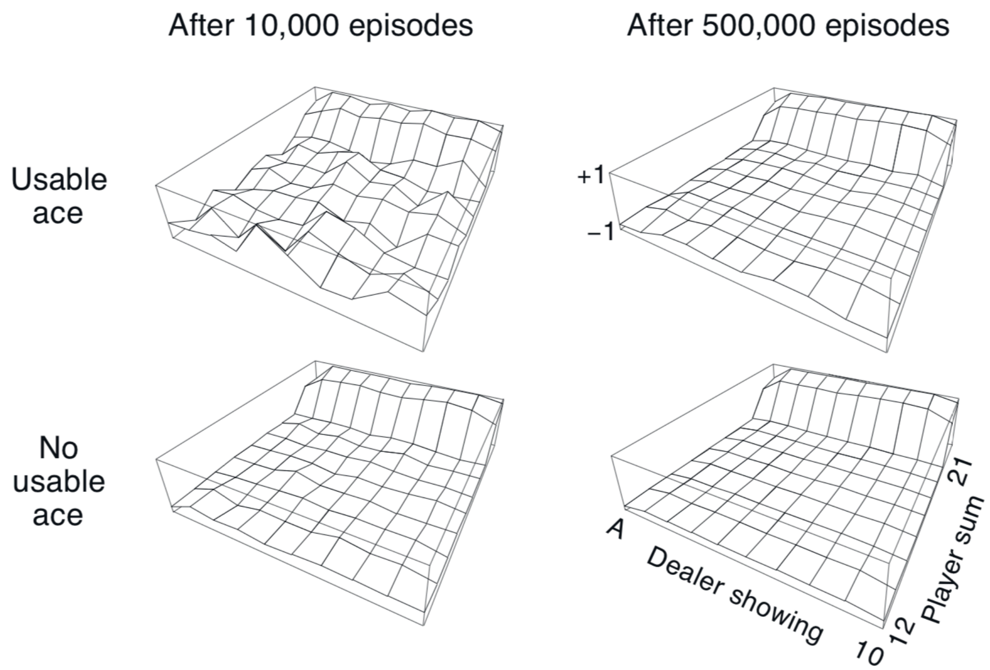
Monte Carlo Methods

- **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. 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**

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

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

Monte Carlo Methods Summary

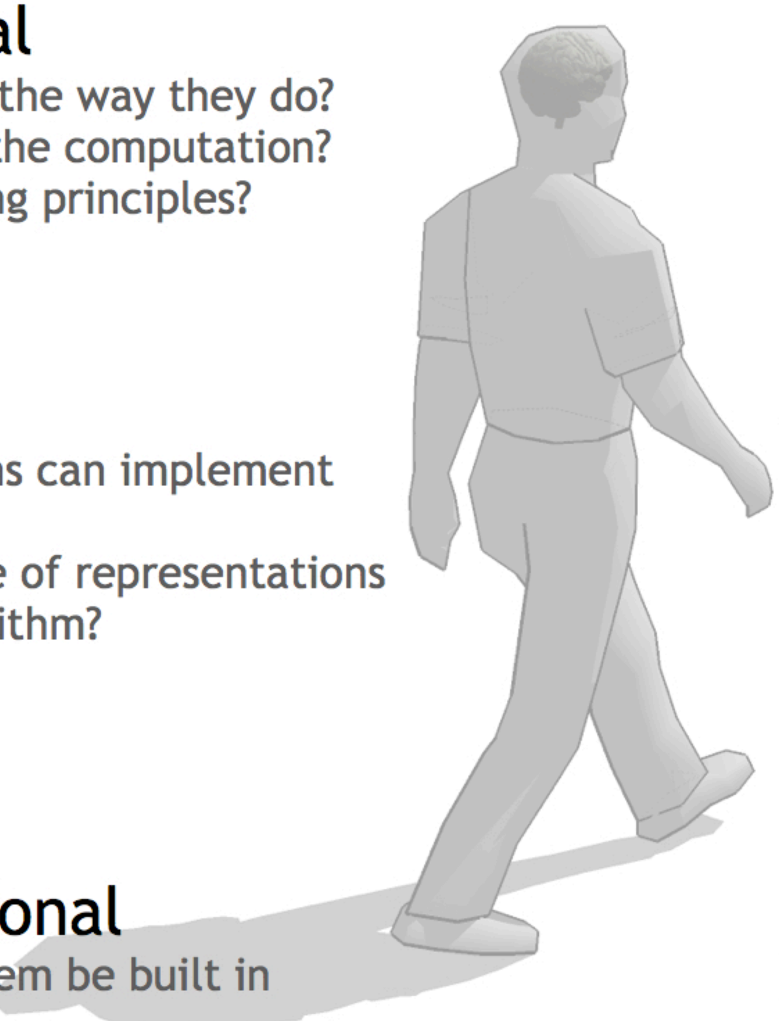
- Don't need model of environment!
    - averaging of sample returns from actual experience or even simulated play (e.g., self-play in two player games, etc…)
- can concentrate on "important" states: don't need a full sweep of all states because no bootstrapping
- need to maintain **exploration** (EXPLORE-EXPLOIT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

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

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



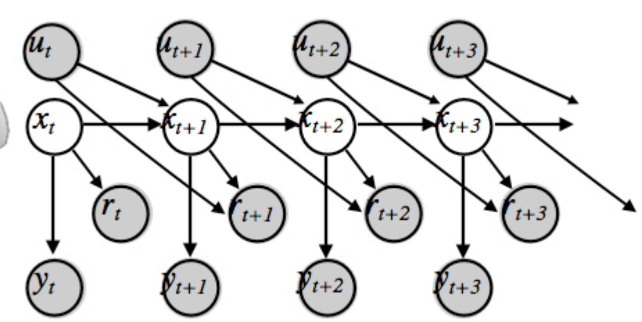
Computational
Why do things work the way they do?
What is the goal of the computation?
What are the unifying principles?

maximize:

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

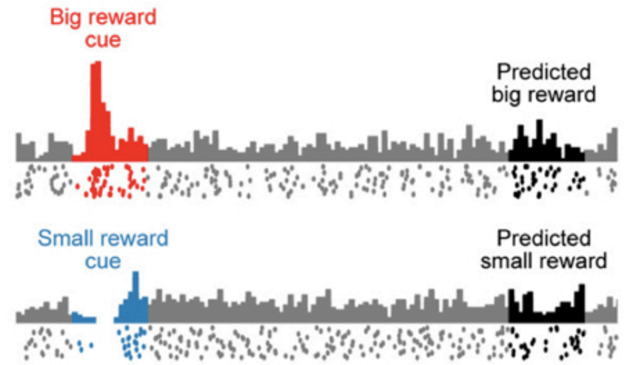
Bellman

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

Next time

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