

# Deep Reinforcement Learning

## Markov Decision Processes and Q-Learning



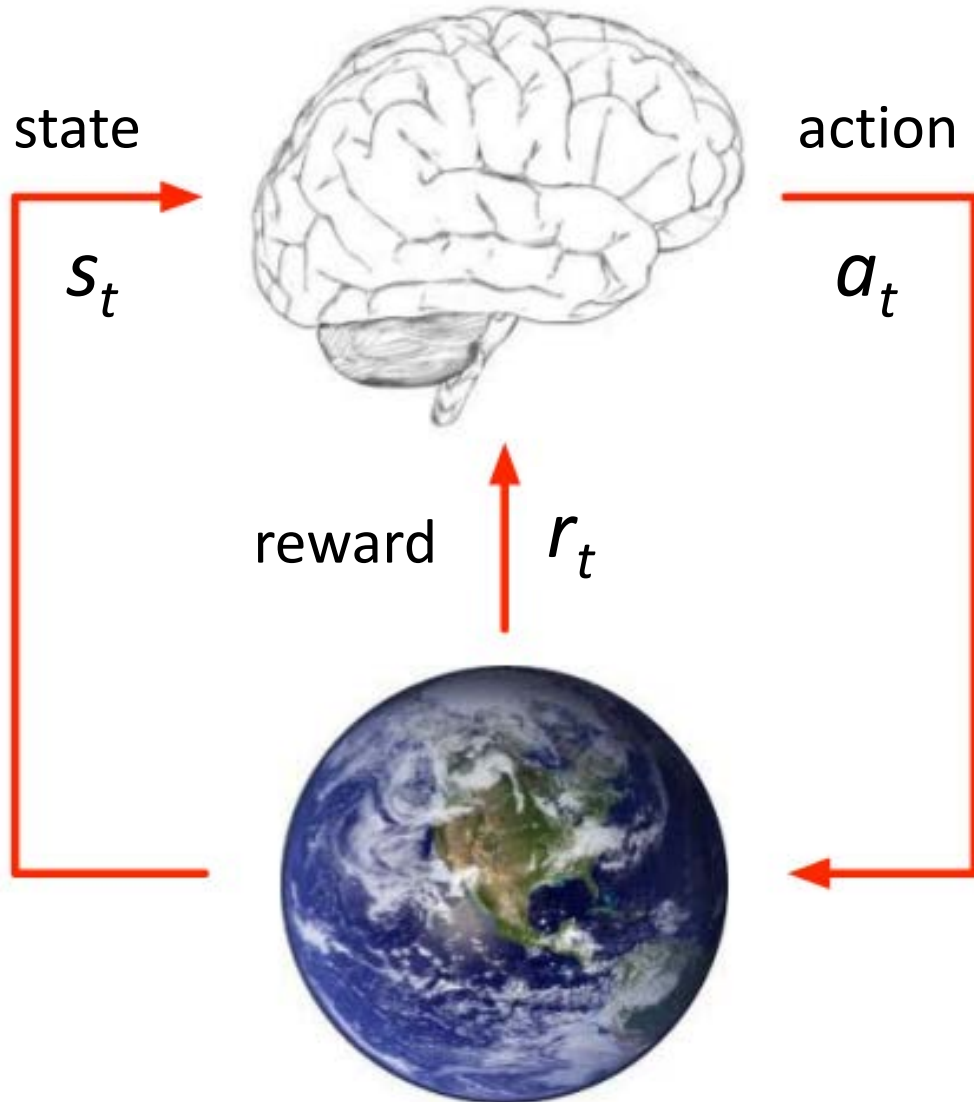
Samuel Cheng  
University of Oklahoma

# Reinforcement Learning (RL)

- Learn how to response to the environment with limited amount of information
- Unlike supervised learning: no one tell you what exactly what you supposed to do
- Unlike unsupervised learning: RL has a clear objective. Bottom line, You don't want to crash your car



# Agent and Environment



- At each step  $t$  the agent:
  - Receives state  $s_t$
  - Receives scalar reward  $r_t$
  - Executes action  $a_t$
- The environment:
  - Receives action  $a_t$
  - Emits state  $s_t$
  - Emits scalar reward  $r_t$

# Examples of RL

---

- **Control** physical systems: walk, fly, drive, swim, ...
- **Interact** with users: retain customers, personalise channel, optimise user experience, ...
- **Solve** logistical problems: scheduling, bandwidth allocation, elevator control, cognitive radio, power optimisation, ..
- **Play** games: chess, checkers, Go, Atari games, ...
- **Learn** sequential algorithms: attention, memory, conditional computation, activations, ...

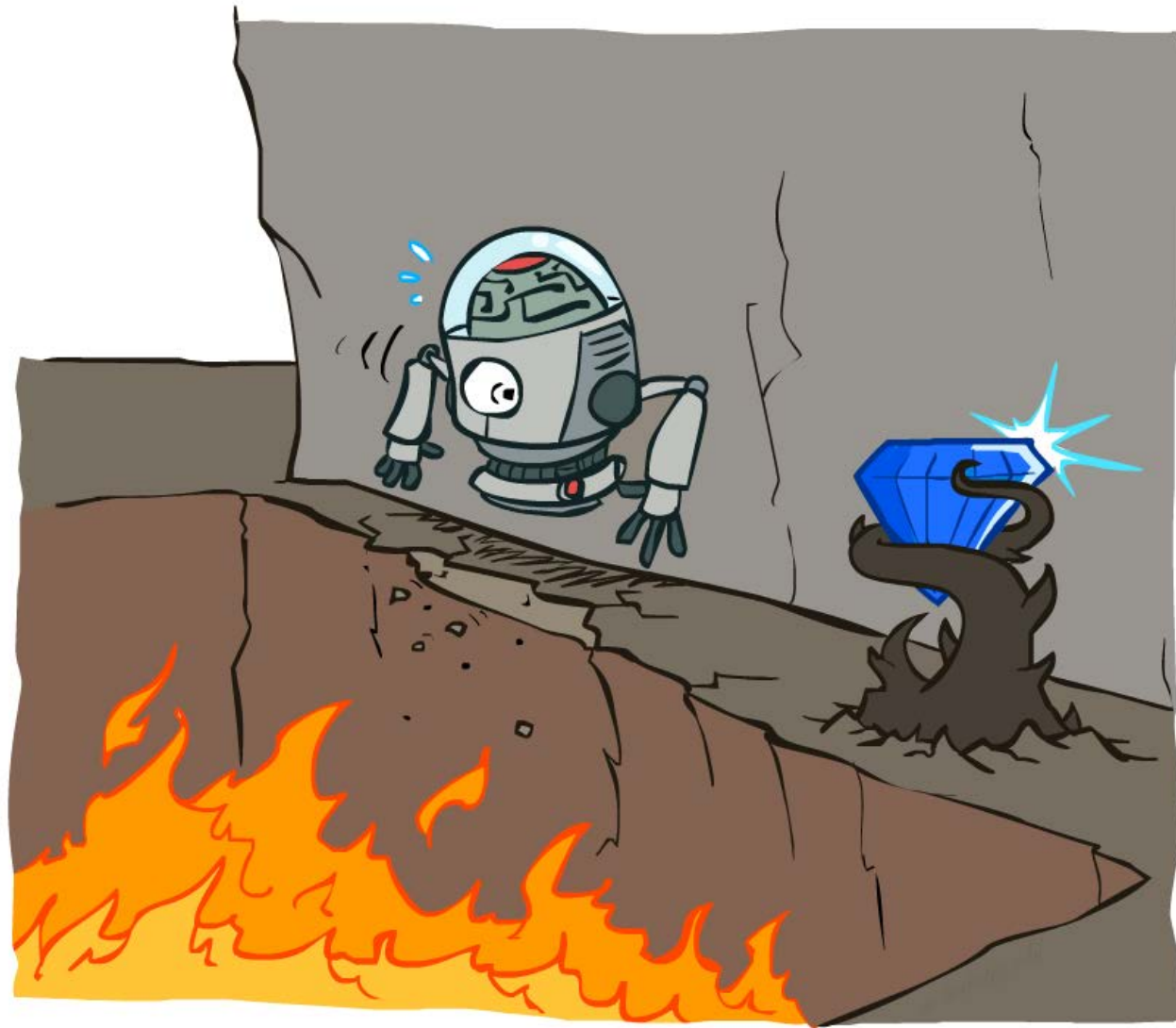
# Today's Plan

---

- Learn how to response assuming that we DO know what the environment is like (we have a map): **non-deterministic search** and **Markov decision process**
- Learn how to response even when we DON'T know what the environment is like (we don't have a map): **Q-learning**
- Learn how to response when we DON'T know what the environment is like and without being drowned in information: **approximate Q-learning, deep Q-learning**

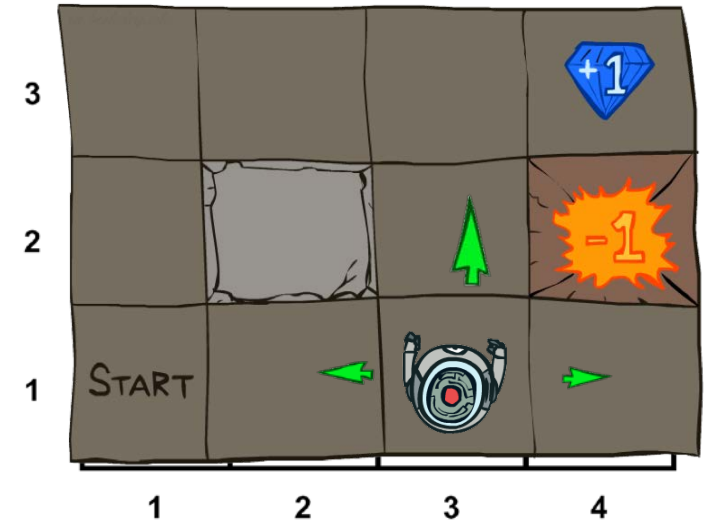
# Non-Deterministic Search

---



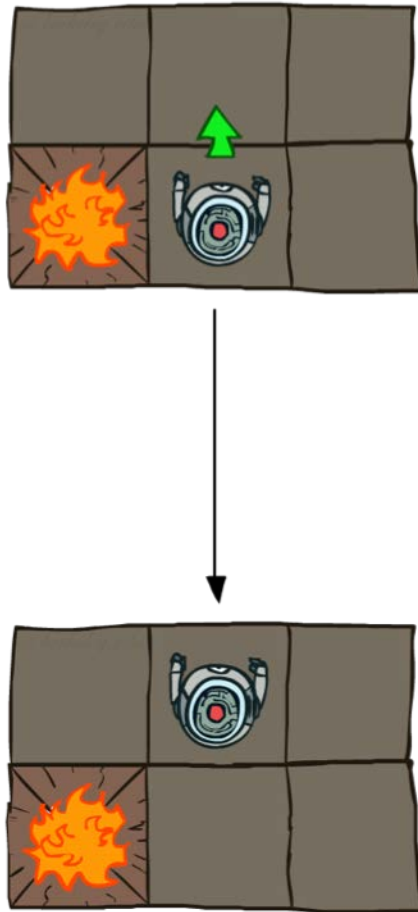
# Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards

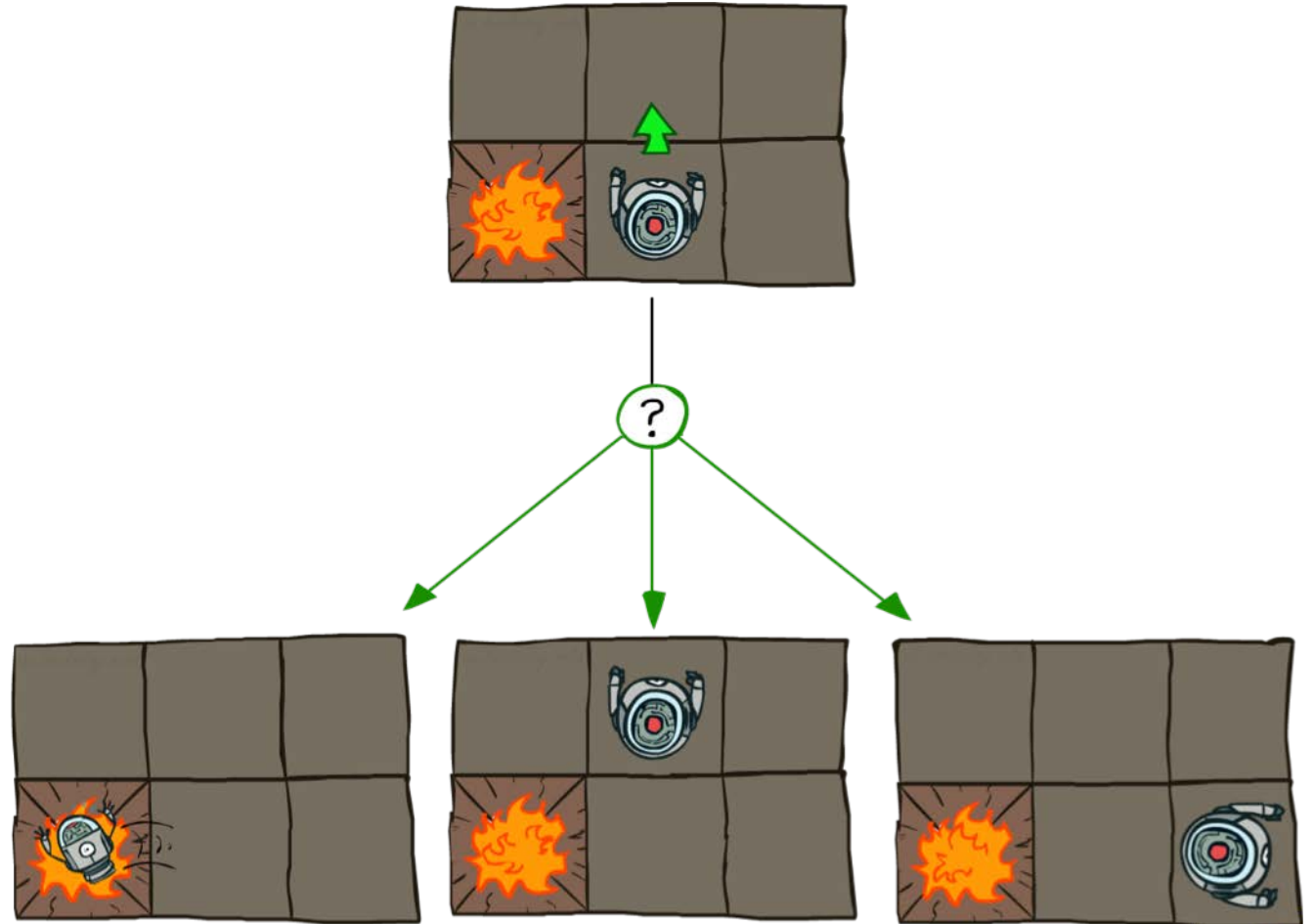


# Grid World Actions

Deterministic Grid World



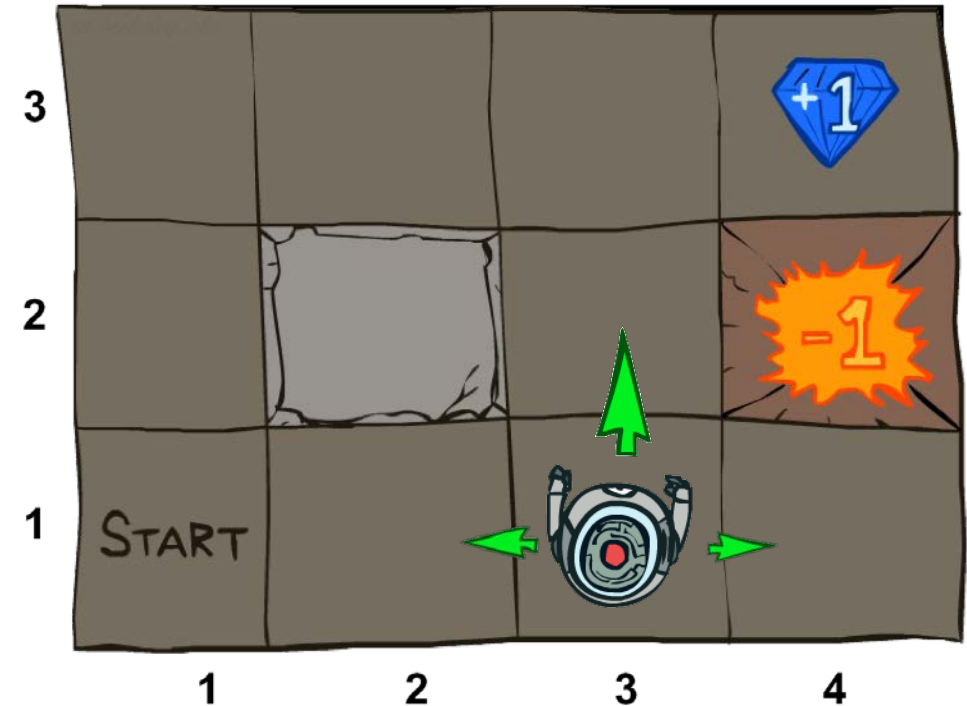
Stochastic Grid World





# Markov Decision Processes

- An MDP is defined by:
  - A **set of states**  $s \in S$
  - A **set of actions**  $a \in A$
  - A **transition function**  $T(s, a, s')$ 
    - Probability that  $a$  from  $s$  leads to  $s'$ , i.e.,  $P(s' | s, a)$
    - Also called the model or the dynamics
  - A **reward function**  $R(s, a, s')$ 
    - Sometimes just  $R(s)$  or  $R(s')$
  - A **start state**
  - Maybe a **terminal state**
- MDPs are **non-deterministic search problems**
  - One way to solve them is with expectimax search
  - We'll have a new tool soon



# What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent
- For Markov decision processes, “Markov” means action outcomes depend only on the current state

$$\begin{aligned} P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \dots, S_0 = s_0) \\ = \\ P(S_{t+1} = s' | S_t = s_t, A_t = a_t) \end{aligned}$$

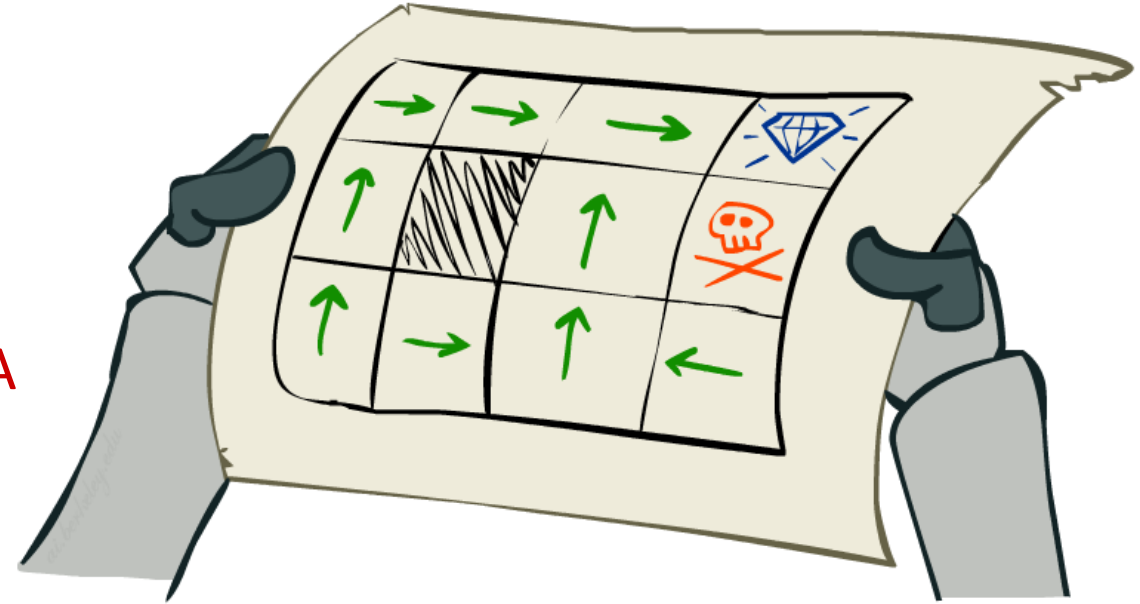
- This is just like search, where the successor function could only depend on the current state (not the history)



Andrey Markov  
(1856-1922)

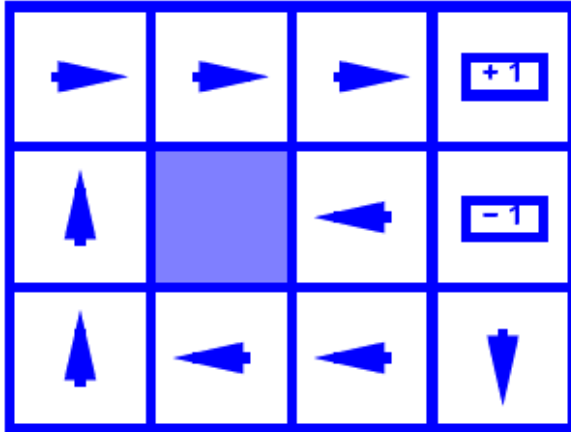
# Policies

- In deterministic single-agent search problems, we wanted an optimal **plan**, or sequence of actions, from start to a goal
- For MDPs, we want an optimal **policy**  $\pi^*: S \rightarrow A$ 
  - A policy  $\pi$  gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent
- Expectimax didn't compute entire policies
  - It computed the action for a single state only

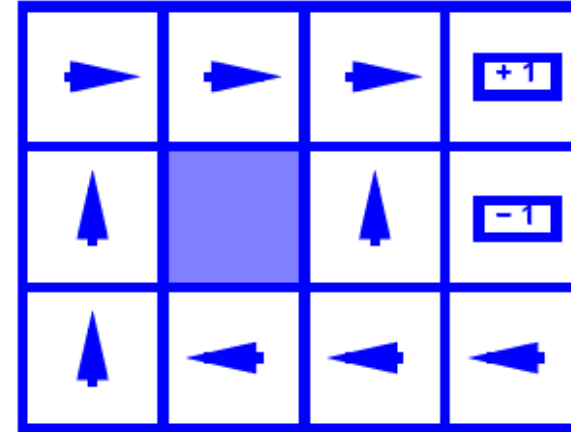


Optimal policy when  $R(s, a, s') = -0.03$   
for all non-terminals  $s$

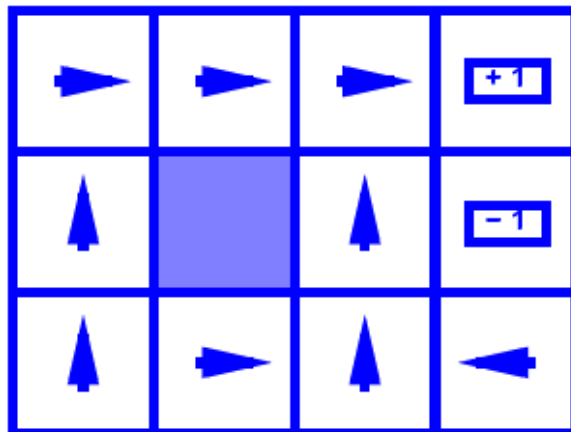
# Immediate Rewards Affect Optimal Policies



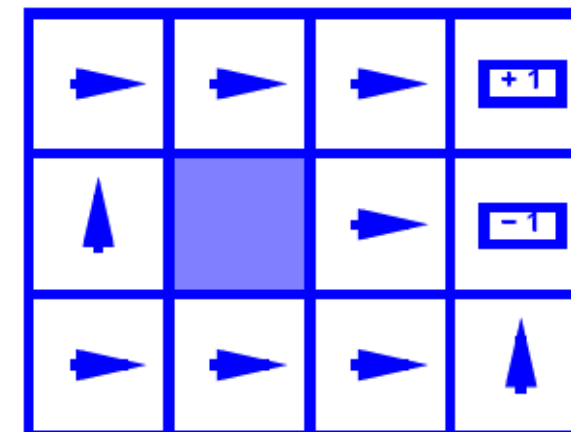
$R(s) = -0.01$



$R(s) = -0.03$



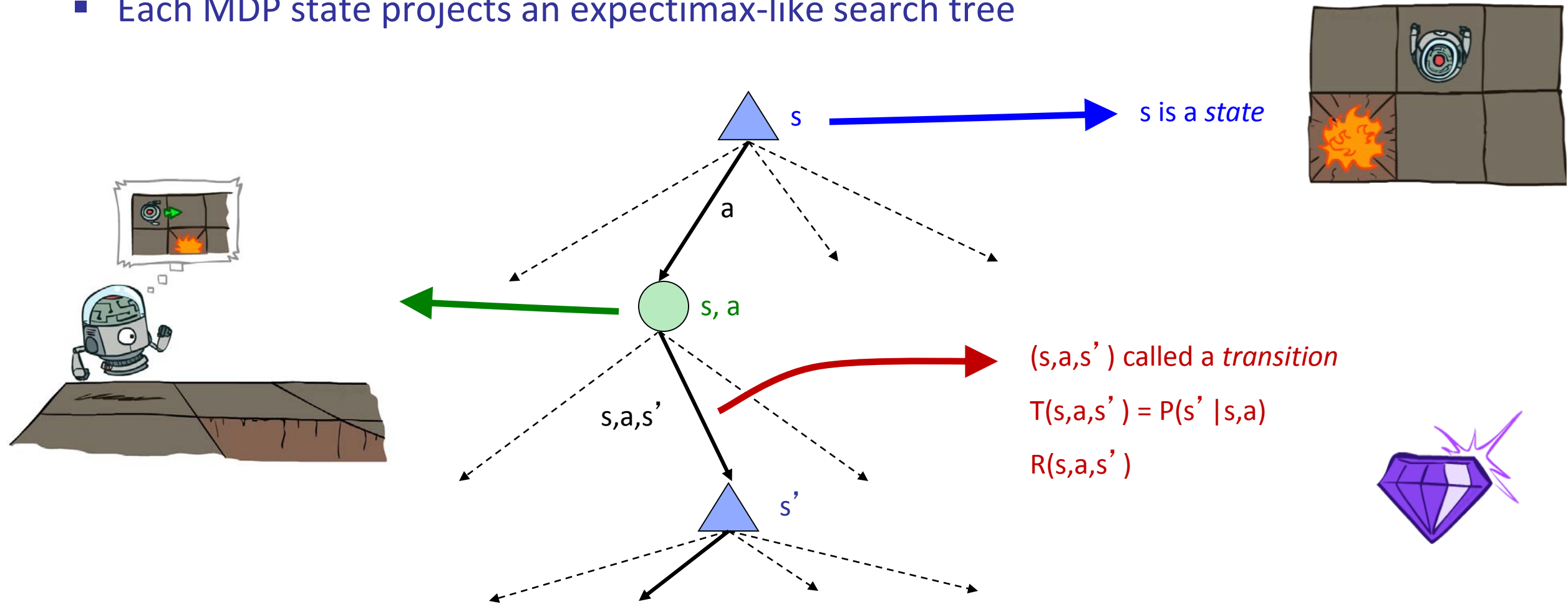
$R(s) = -0.4$



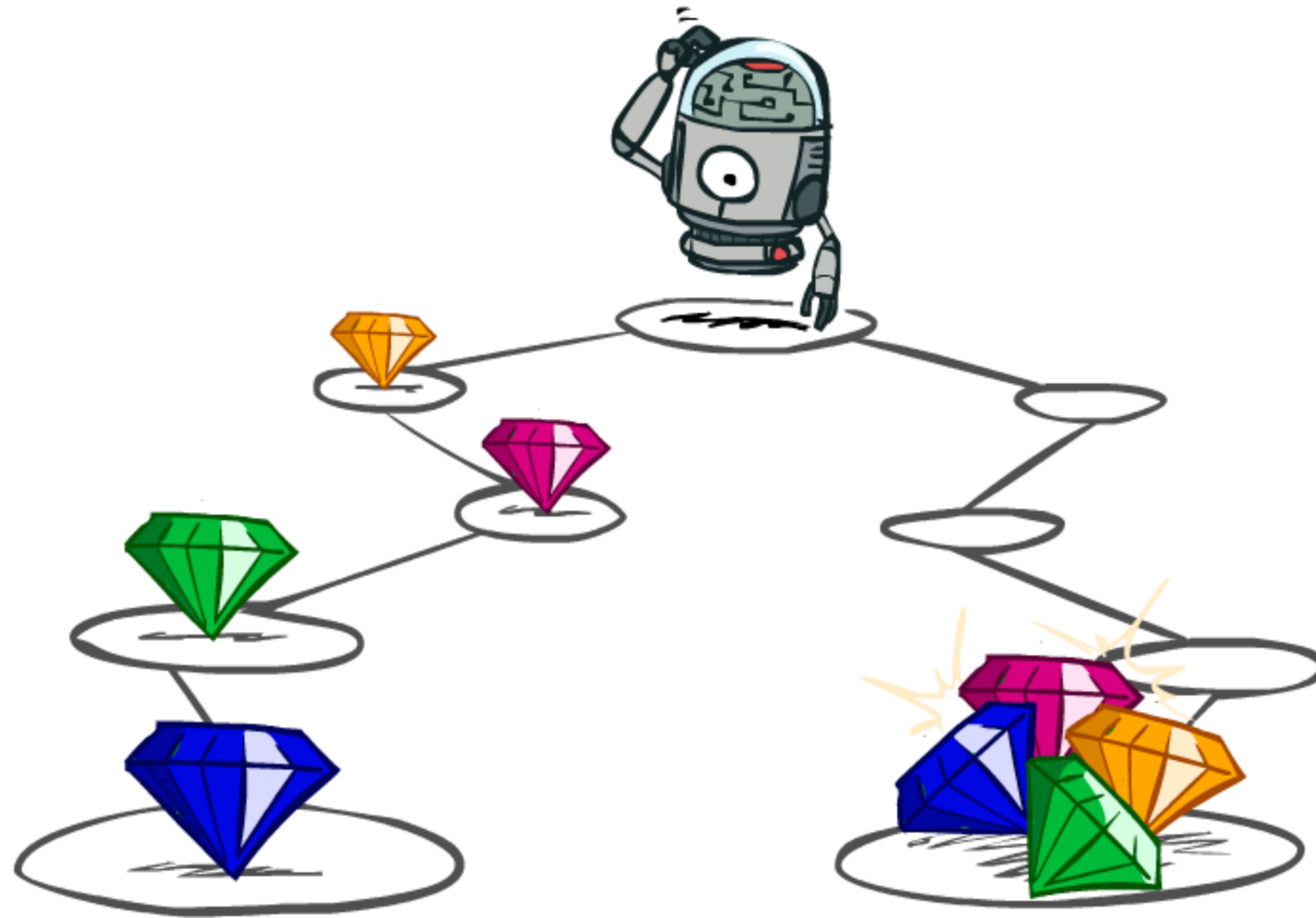
$R(s) = -2.0$

# MDP Search Trees

- Each MDP state projects an expectimax-like search tree

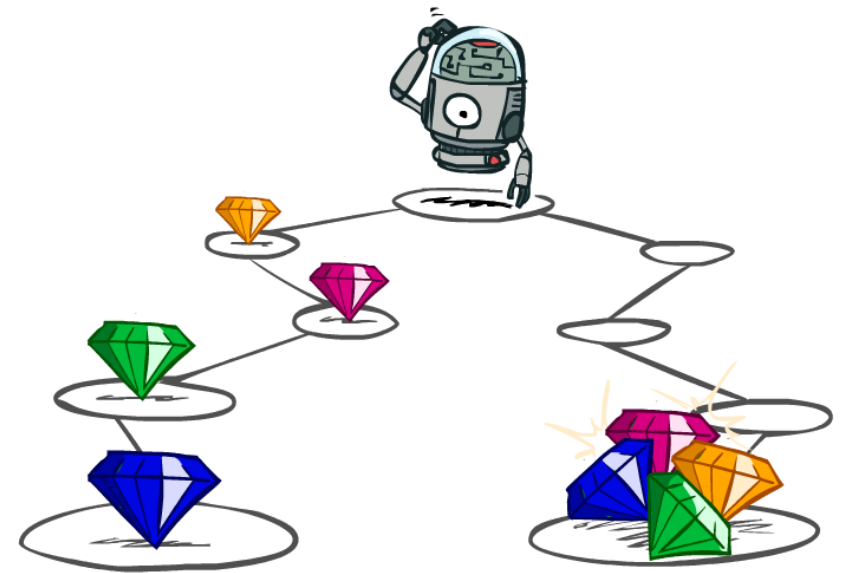


# Utilities of Sequences



# Utilities of Sequences

- What preferences should an agent have over reward sequences?
- More or less?  $[1, 2, 2]$  or  $[2, 3, 4]$
- Now or later?  $[0, 0, 1]$  or  $[1, 0, 0]$



# Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially



1

Worth Now



$\gamma$

Worth Next Step



$\gamma^2$

Worth In Two Steps



# Discounting

- How to discount?

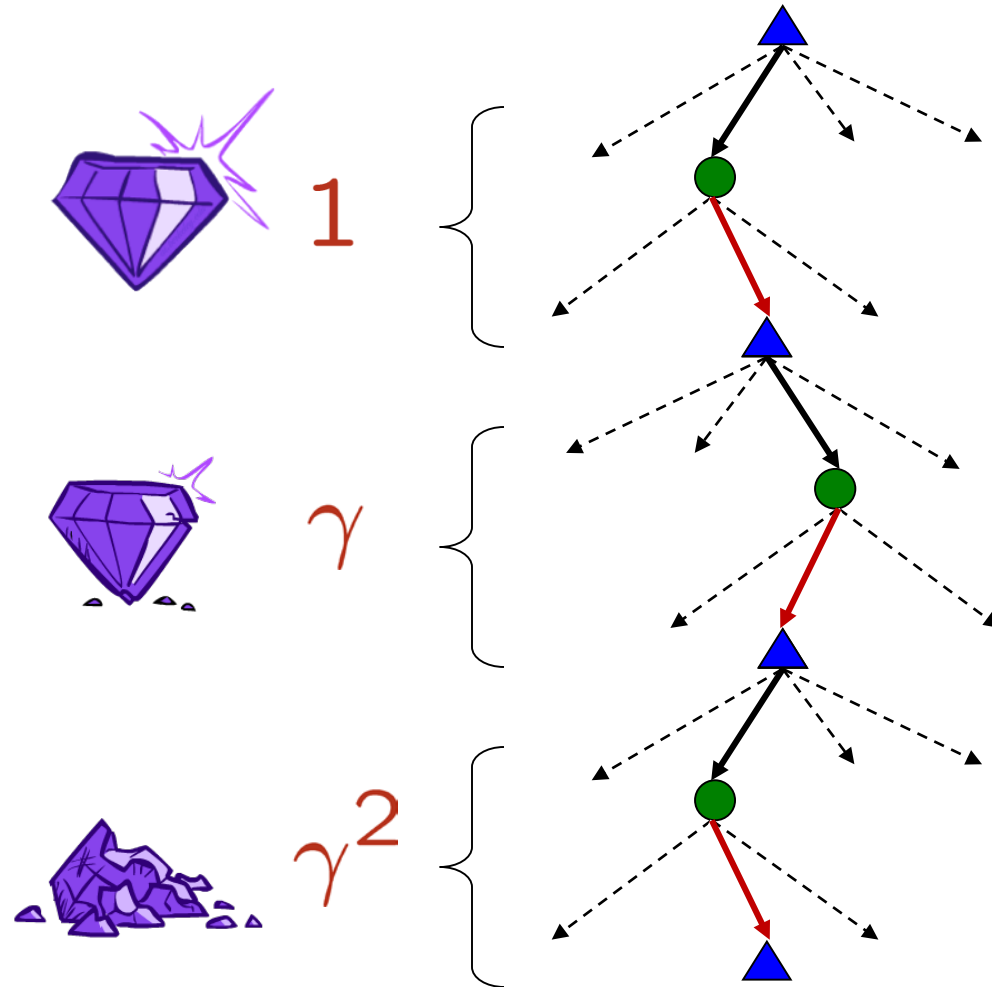
- Each time we descend a level, we multiply in the discount once

- Why discount?

- Sooner rewards probably do have higher utility than later rewards
- Also helps our algorithms converge

- Example: discount of 0.5

- $U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3$
- $U([1,2,3]) < U([3,2,1])$



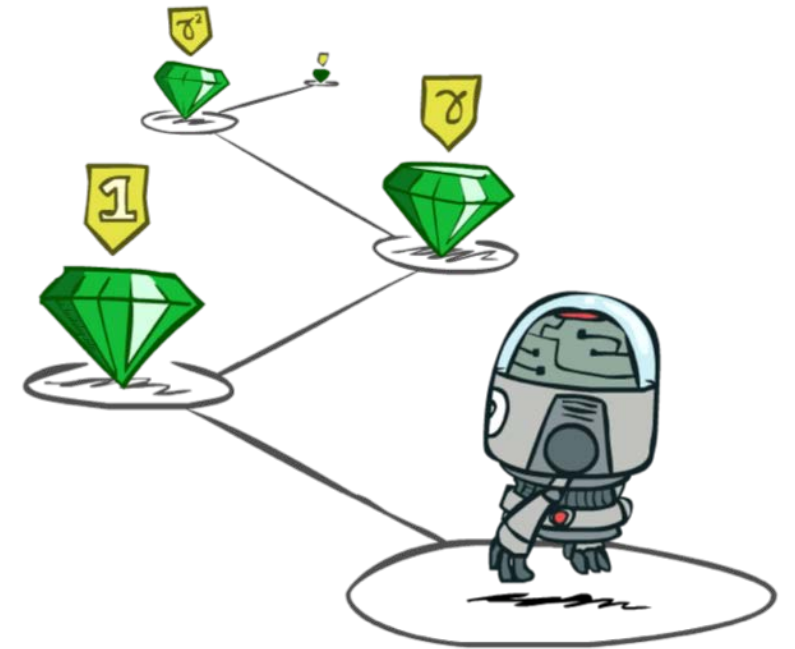
# Stationary Preferences

- Theorem: if we assume **stationary preferences**:

$$[a_1, a_2, \dots] \succ [b_1, b_2, \dots]$$



$$[r, a_1, a_2, \dots] \succ [r, b_1, b_2, \dots]$$



- Then: there are only two ways to define utilities

- Additive utility:  $U([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \dots$

- Discounted utility:  $U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \dots$

# Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?

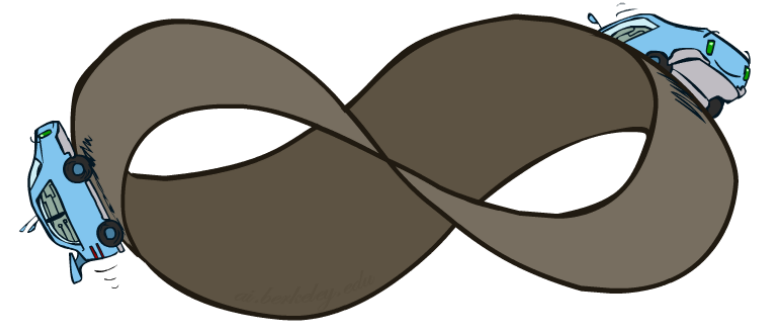
- Solutions:

- Finite horizon: (similar to depth-limited search)
  - Terminate episodes after a fixed T steps (e.g. life)
  - Gives nonstationary policies ( $\pi$  depends on time left)

- Discounting: use  $0 < \gamma < 1$

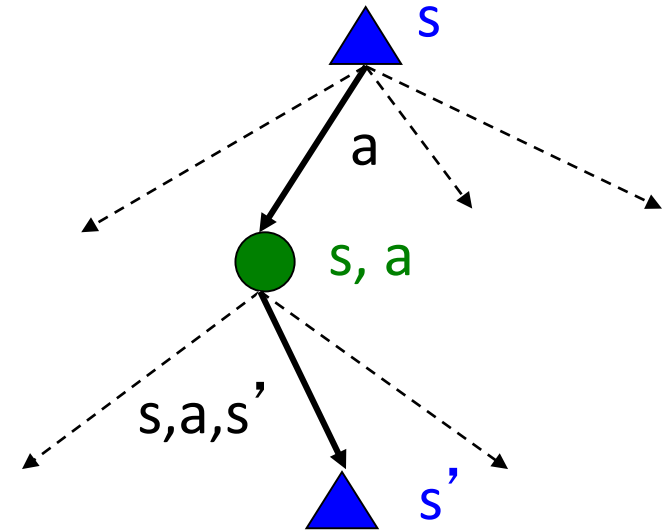
$$U([r_0, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max} / (1 - \gamma)$$

- Smaller  $\gamma$  means smaller “horizon” – shorter term focus
- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

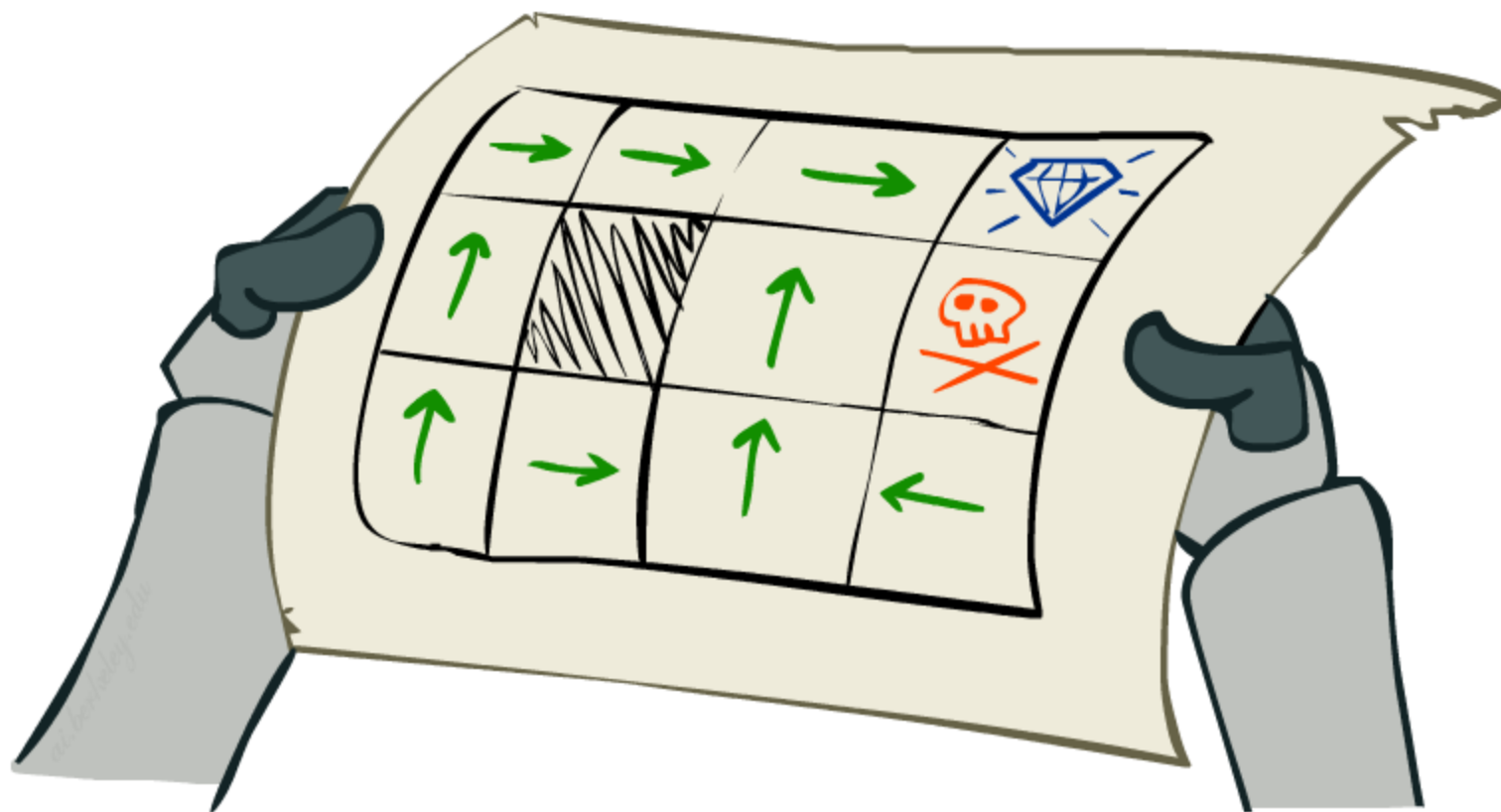


# Recap: Defining MDPs

- Markov decision processes:
  - Set of states  $S$
  - Start state  $s_0$
  - Set of actions  $A$
  - Transitions  $P(s' | s, a)$  (or  $T(s, a, s')$ )
  - Rewards  $R(s, a, s')$  (and discount  $\gamma$ )
- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards

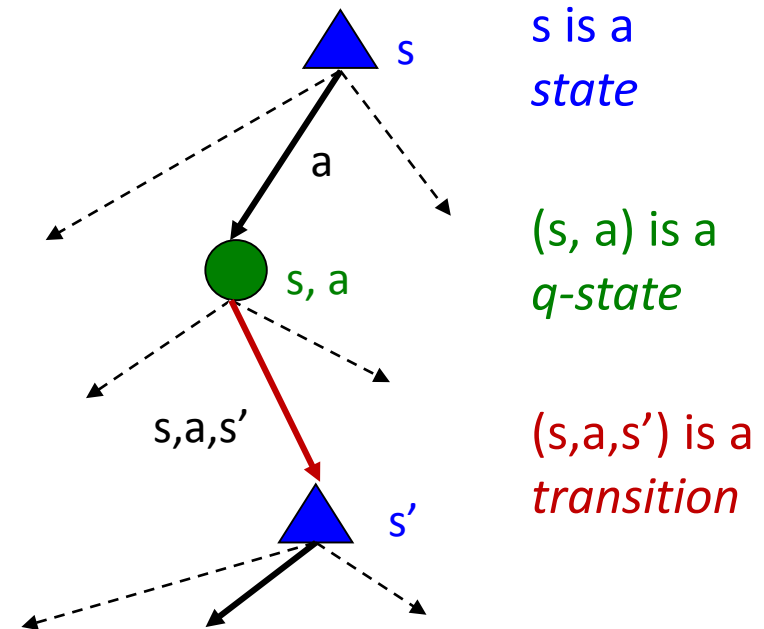


# Solving MDPs



# Optimal Quantities

- The value (utility) of a state  $s$ :  
 $V^*(s)$  = expected utility starting in  $s$  and acting optimally
- The value (utility) of a q-state  $(s,a)$ :  
 $Q^*(s,a)$  = expected utility starting out having taken action  $a$  from state  $s$  and (thereafter) acting optimally
- The optimal policy:  
 $\pi^*(s)$  = optimal action from state  $s$

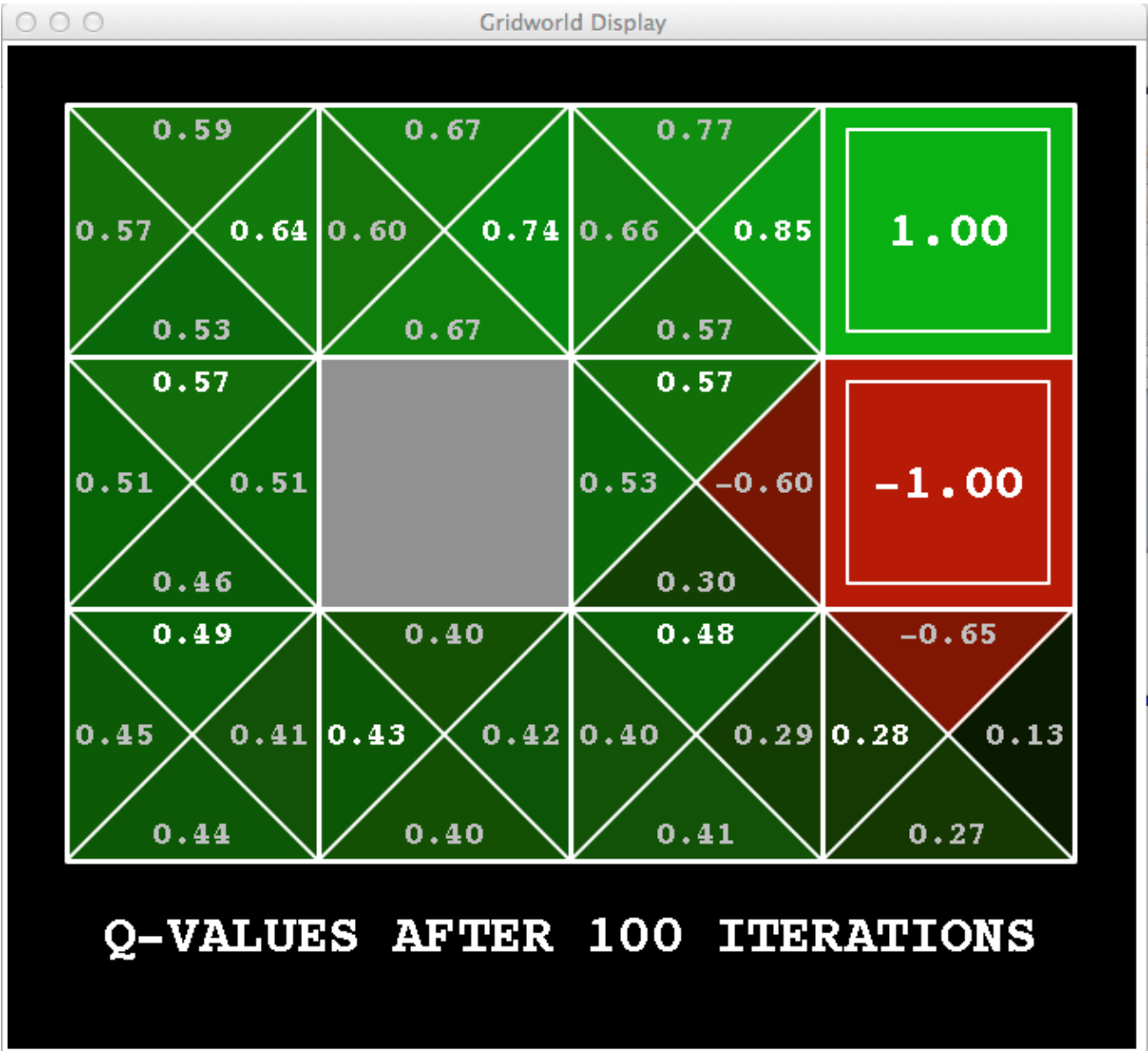


# Snapshot of Demo – Gridworld V Values



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# Snapshot of Demo – Gridworld Q Values



Noise = 0.2  
Discount = 0.9  
Living reward = 0



# How to Find Values of States? Bellman Equations

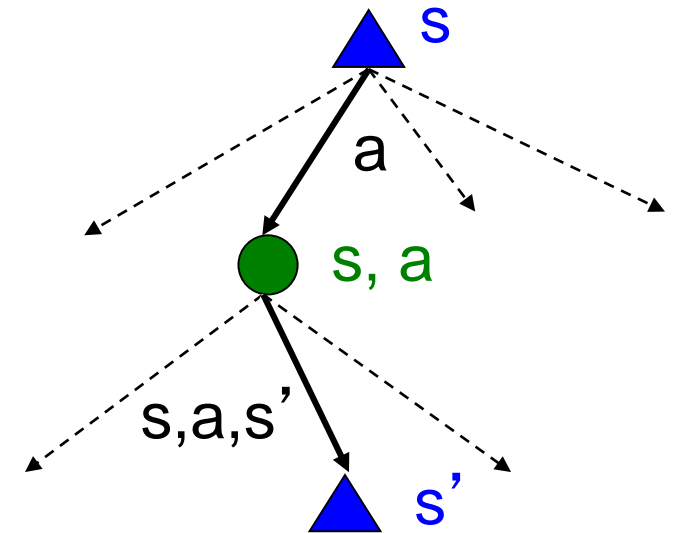
- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!

- Recursive definition of value (**Bellman** eqns):

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

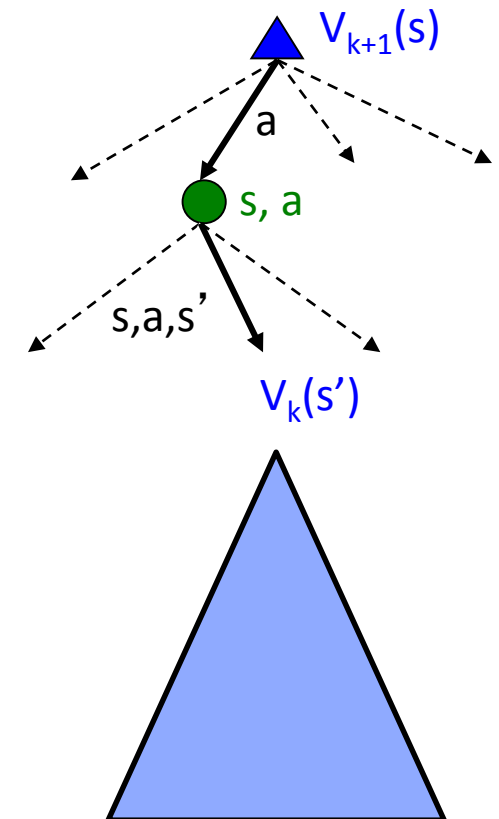


# Value Iteration

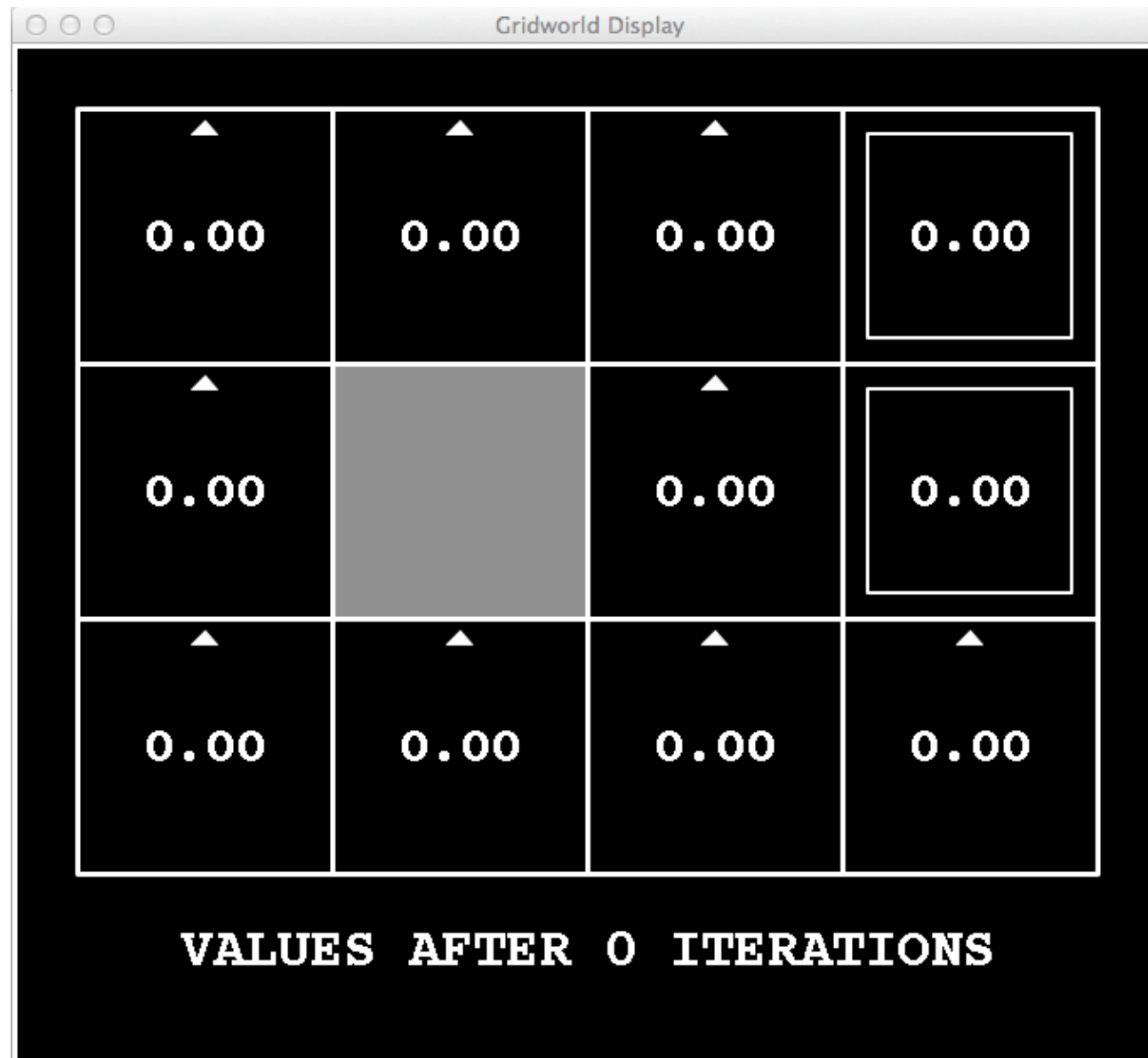
- Start with  $V_0(s) = 0$ : no time steps left means an expected reward sum of zero
- Given vector of  $V_k(s)$  values, do one ply of expectimax from each state:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

- Repeat until convergence
- Complexity of each iteration:  $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

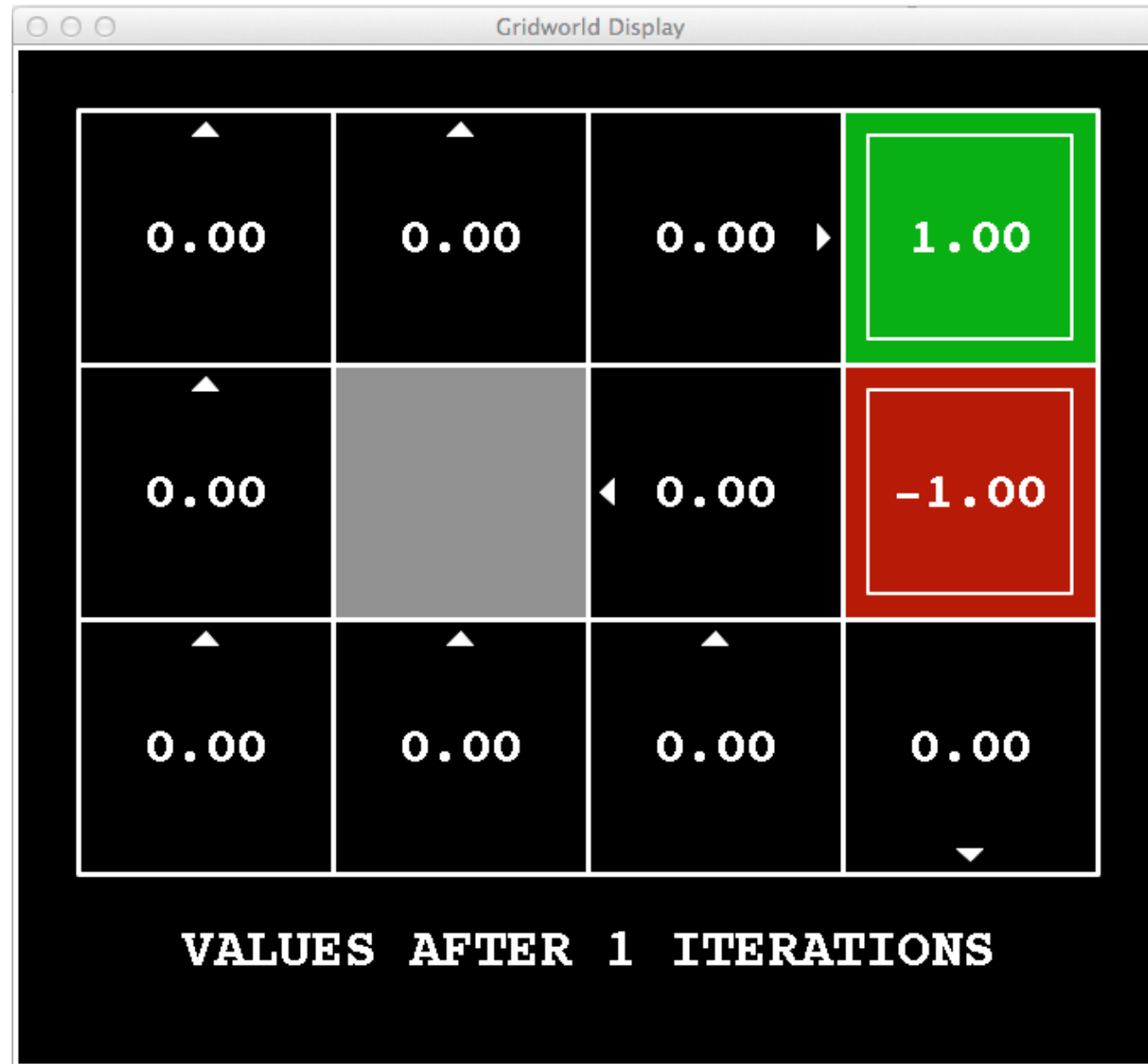


# k=0



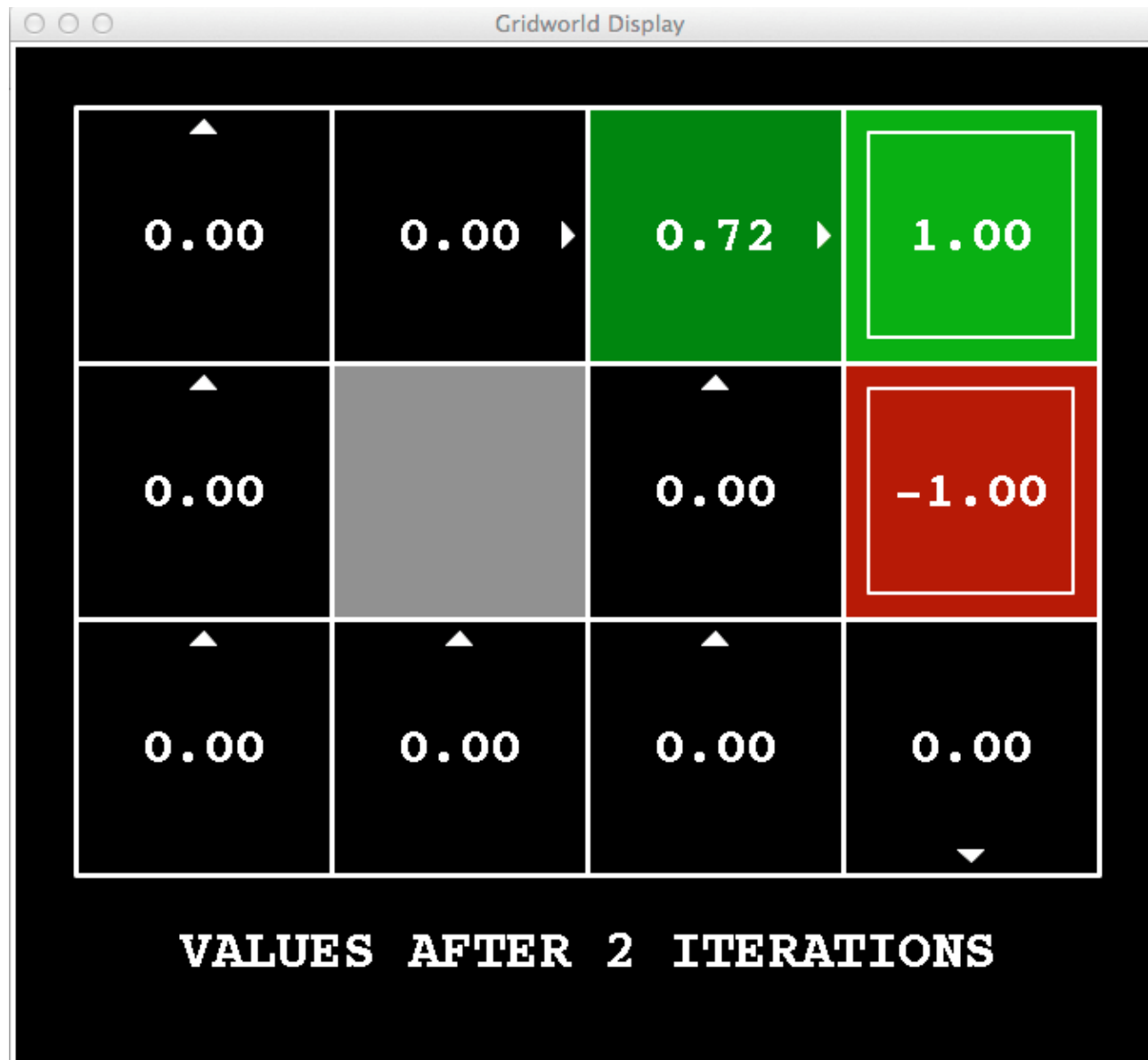
Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=1



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=2



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=3



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=4



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=5



Noise = 0.2  
Discount = 0.9  
Living reward = 0



# k=6



Noise = 0.2  
Discount = 0.9  
Living reward = 0

k=7



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=8



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=9



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=10



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=11



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=12



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=100



Noise = 0.2  
Discount = 0.9  
Living reward = 0



# Summary of Value Iteration

- Bellman equations **characterize** the optimal values:

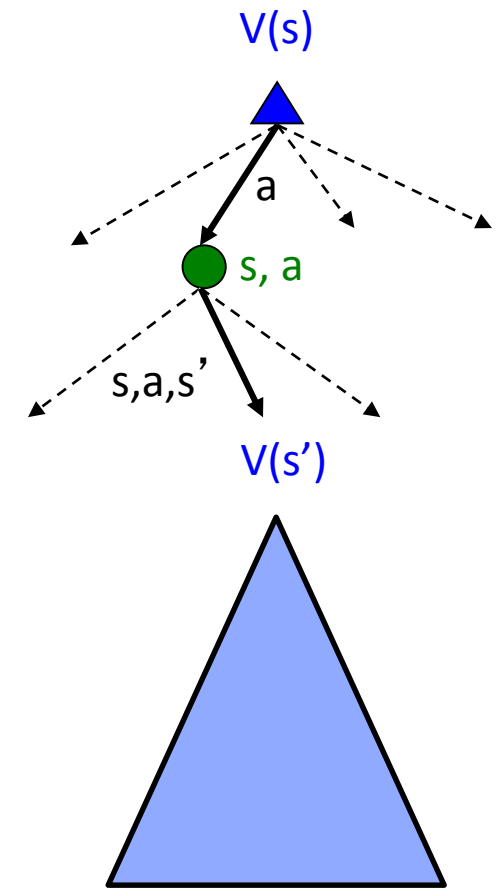
$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

- Value iteration **computes** them:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

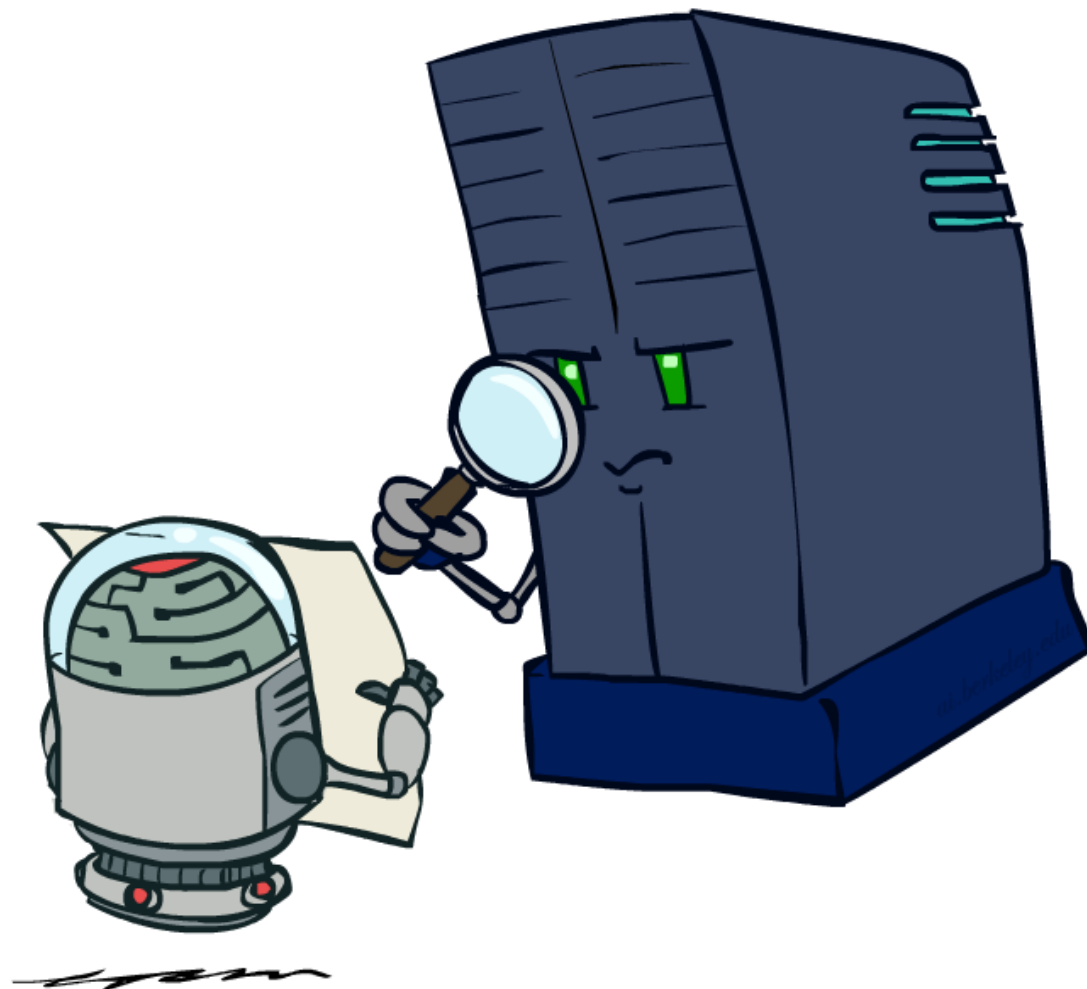
- Value iteration is just a fixed point solution method

- ... though the  $V_k$  vectors are also interpretable as time-limited values



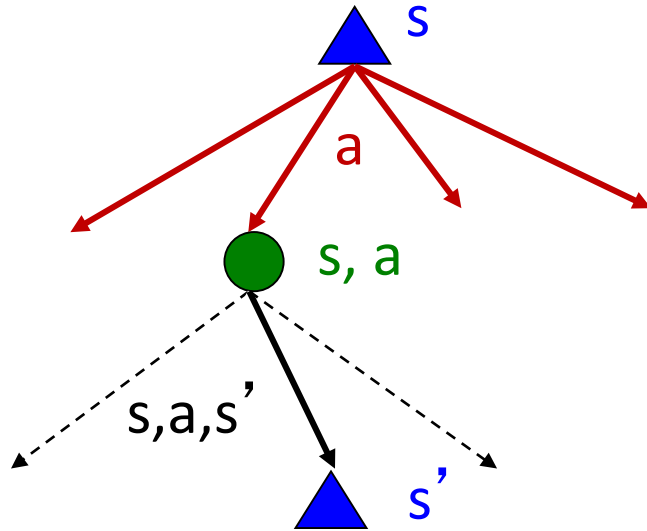
# Policy Evaluation

---

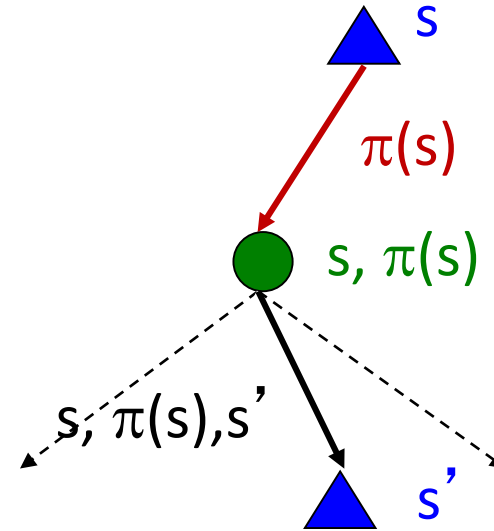


# Fixed Policies

Do the optimal action



Do what  $\pi$  says to do

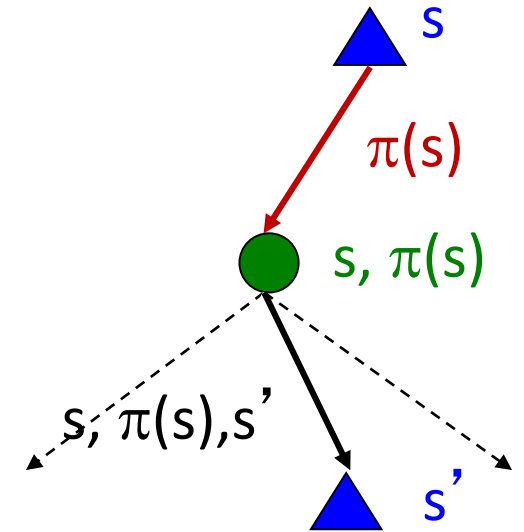


- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy  $\pi(s)$ , then the tree would be simpler – only one action per state
  - ... though the tree's value would depend on which policy we fixed

# Utilities for a Fixed Policy

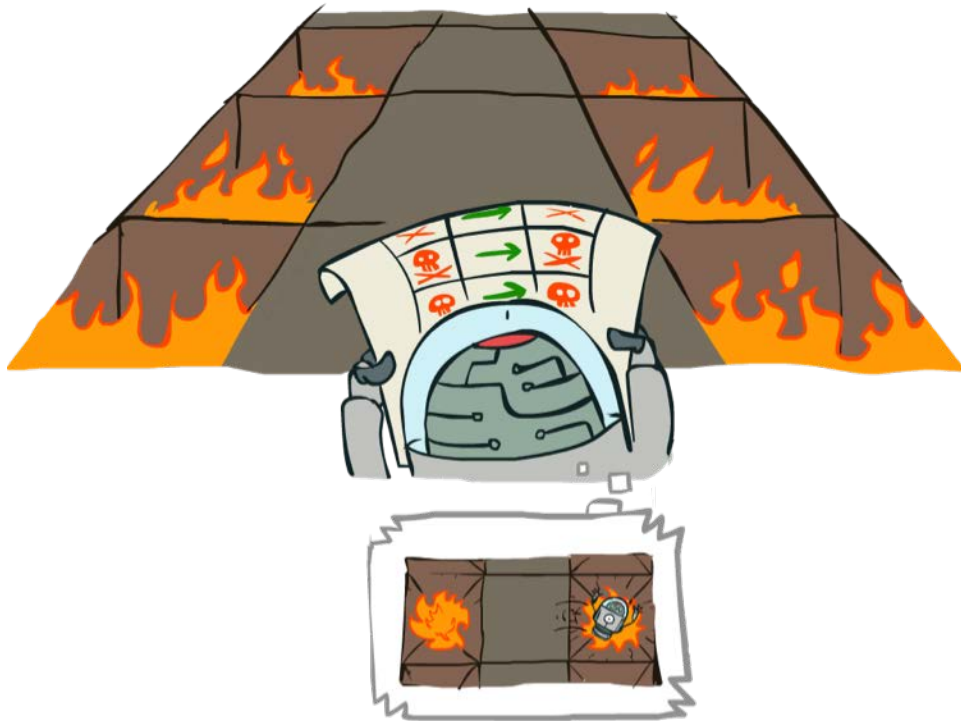
- Another basic operation: compute the utility of a state  $s$  under a fixed (generally non-optimal) policy
- Define the utility of a state  $s$ , under a fixed policy  $\pi$ :  
 $V^\pi(s)$  = expected total discounted rewards starting in  $s$  and following  $\pi$
- Recursive relation (one-step look-ahead / Bellman equation):

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

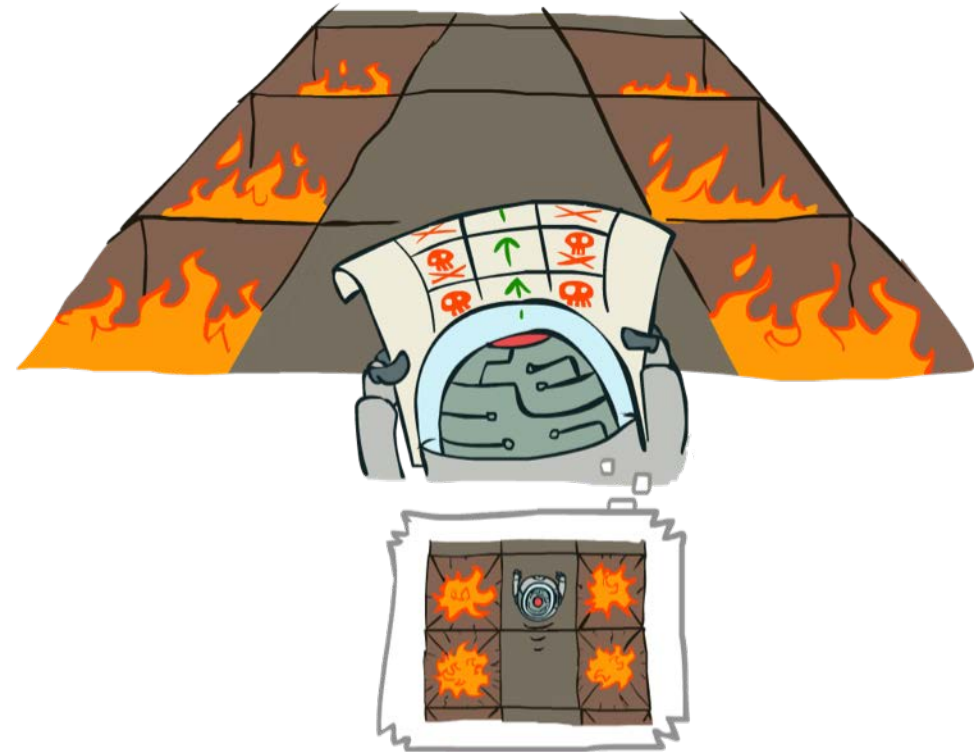


# Example: Policy Evaluation

Always Go Right



Always Go Forward

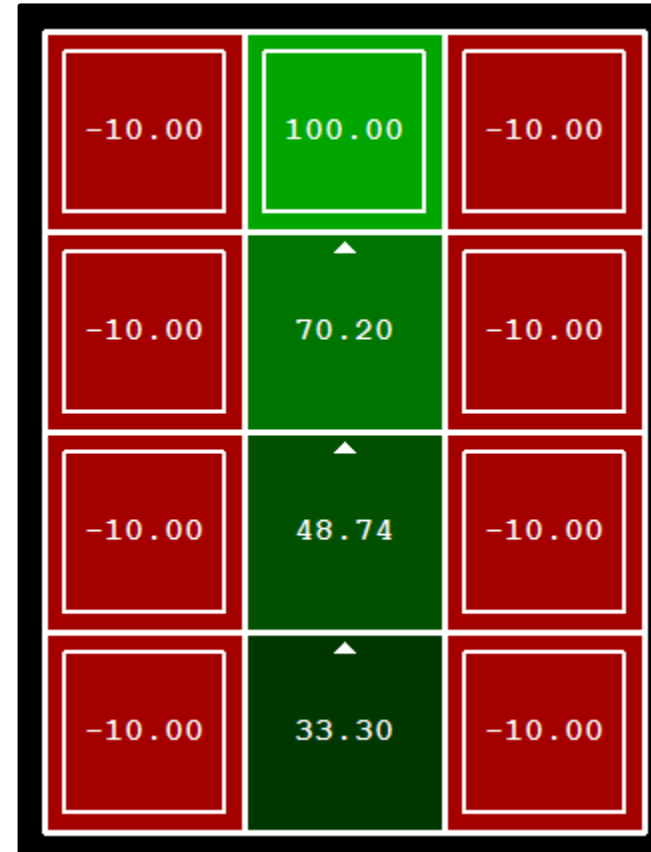


# Example: Policy Evaluation

Always Go Right



Always Go Forward



# Policy Evaluation

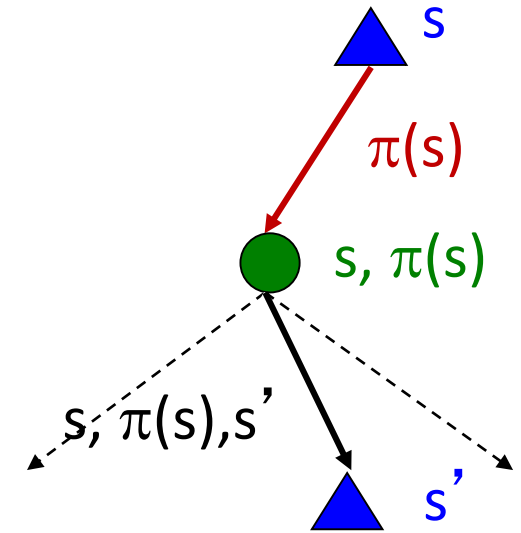
- How do we calculate the  $V$ 's for a fixed policy  $\pi$ ?
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

$$V_0^\pi(s) = 0$$

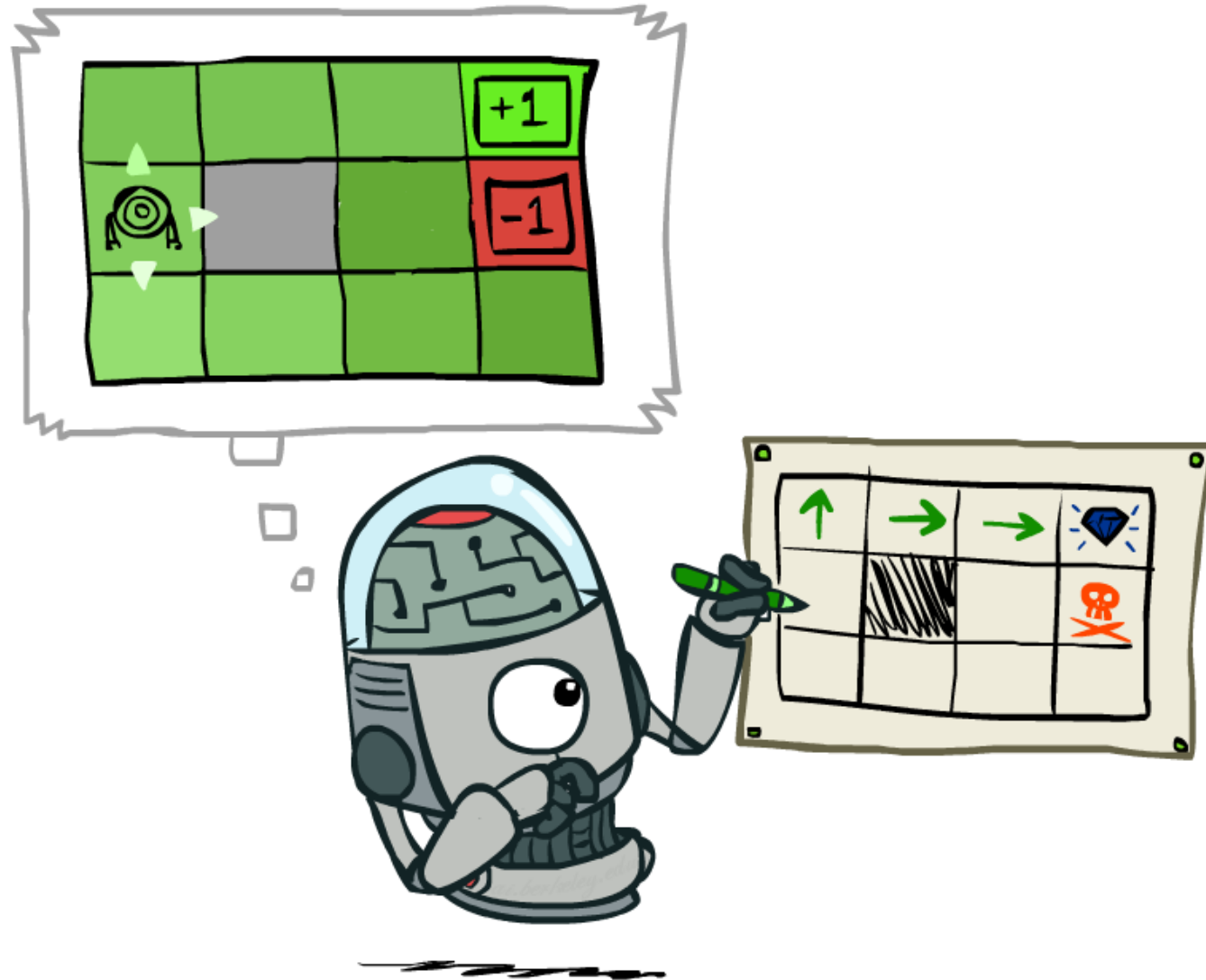
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

Efficiency:  $O(S^2)$  per iteration

- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)



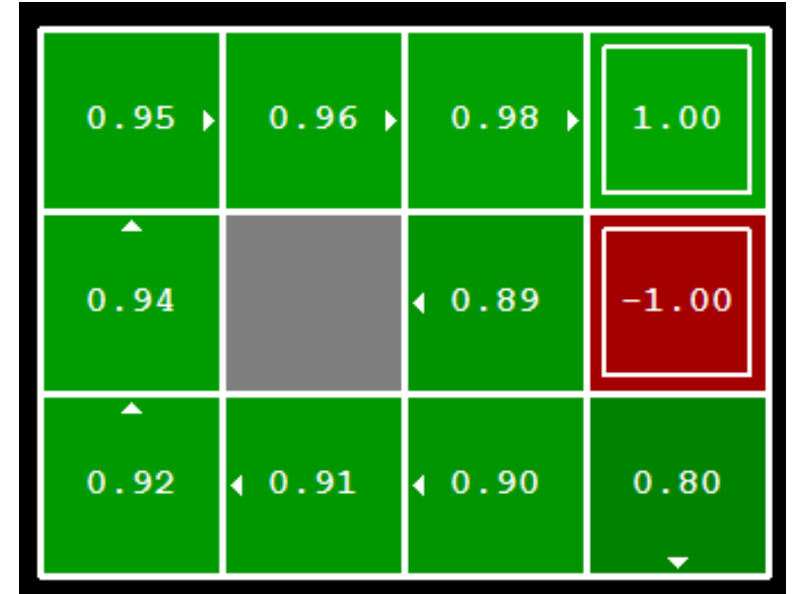
# Policy Extraction





# Computing Actions from Values

- Let's imagine we have the optimal values  $V^*(s)$
- How should we act?
  - It's not too bad...
- We need to do a mini-expectimax (one step)



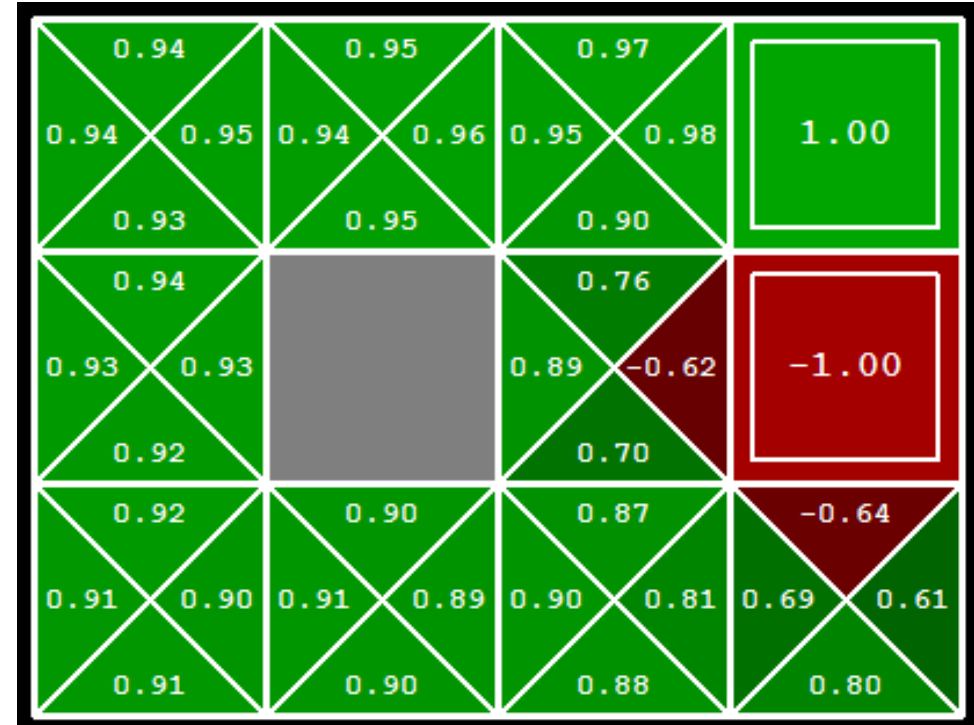
$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

- This is called **policy extraction**, since it gets the policy implied by the values

# Computing Actions from Q-Values

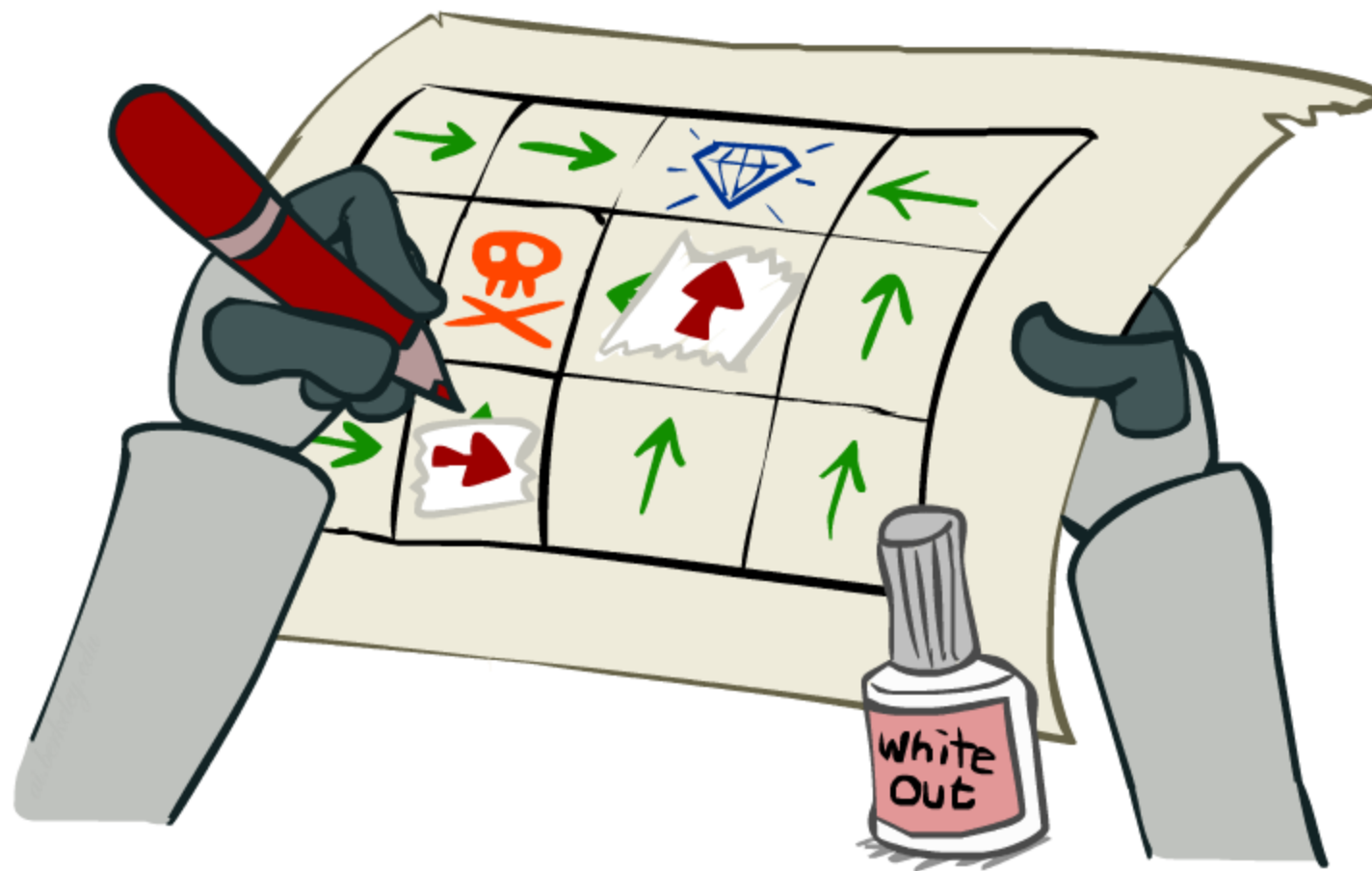
- Let's imagine we have the optimal q-values:
- How should we act?
  - Completely trivial to decide!

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$



- Important lesson: actions are easier to select from q-values than values!

# Policy Iteration

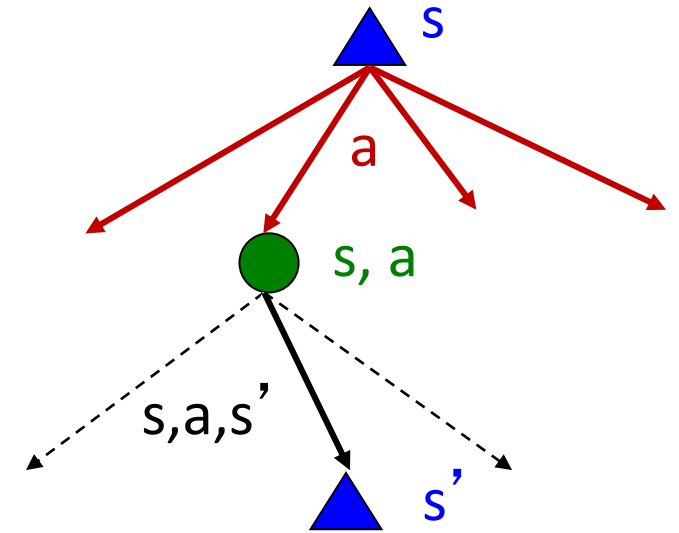


# Problems with Value Iteration

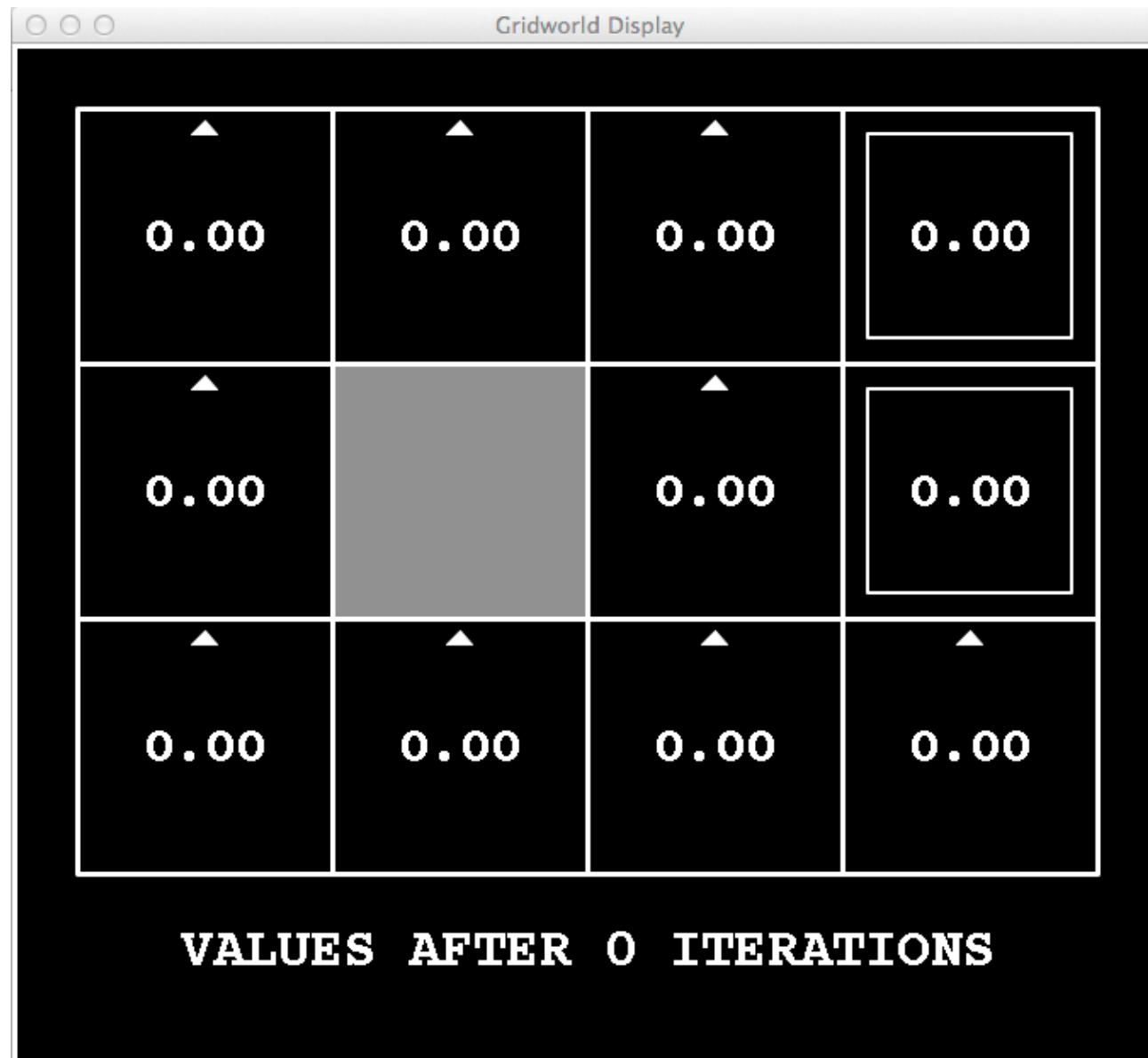
- Value iteration repeats the Bellman updates:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

- Problem 1: It's slow –  $O(S^2A)$  per iteration
- Problem 2: The “max” at each state rarely changes
- Problem 3: The policy often converges long before the values

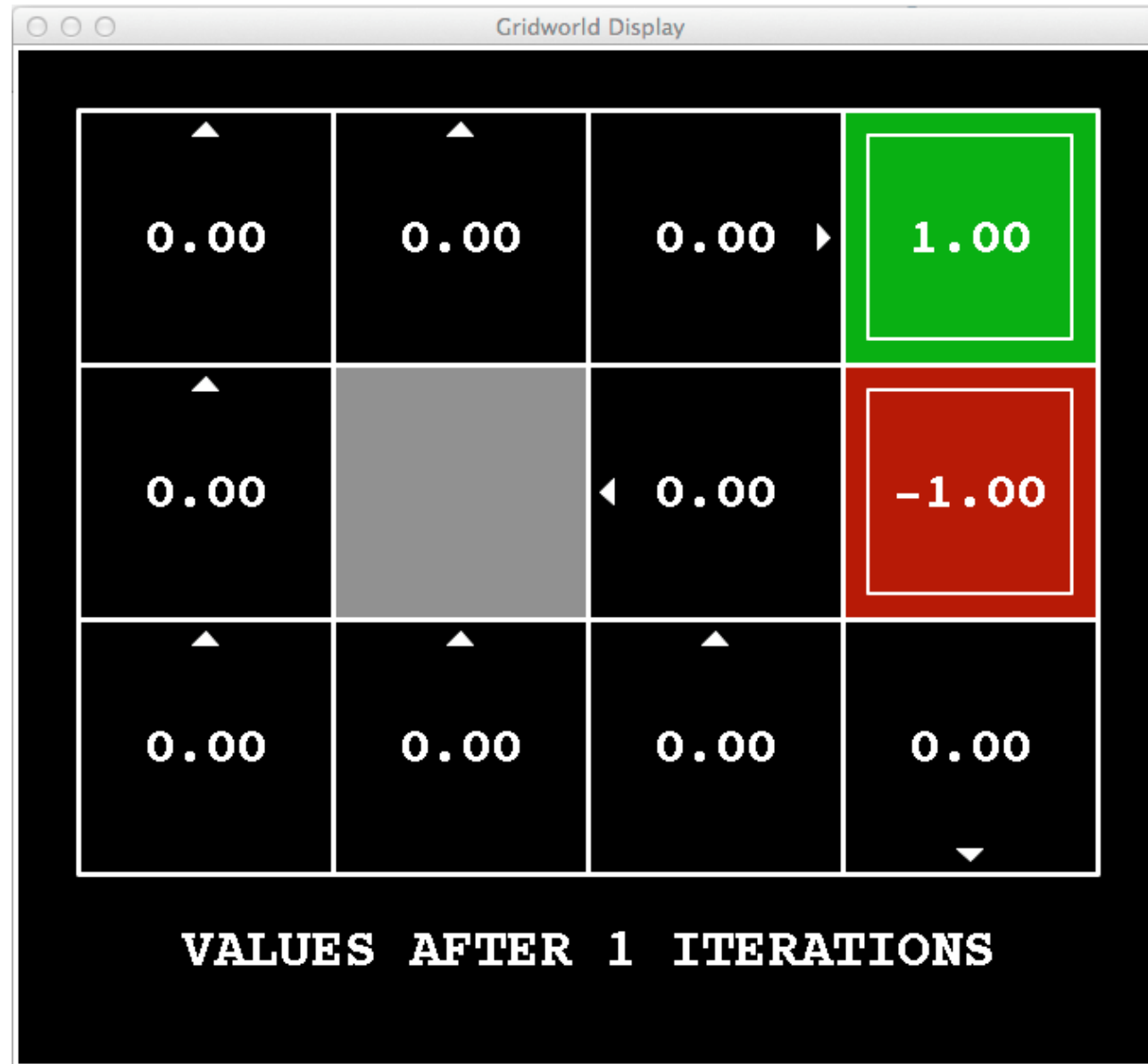


# k=0



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=1



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=2



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=3



Noise = 0.2  
Discount = 0.9  
Living reward = 0

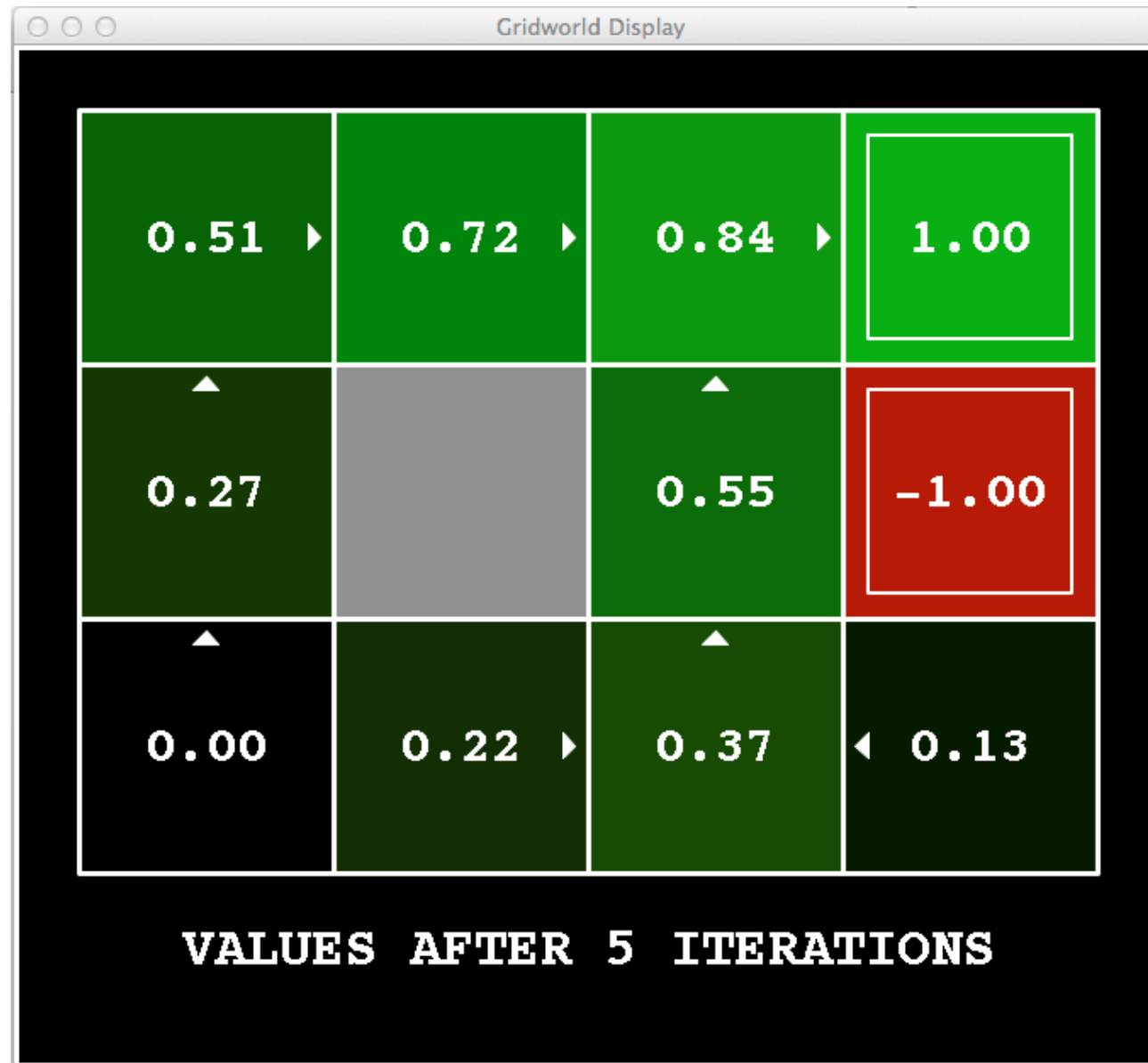


# k=4



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=5



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=6



Noise = 0.2  
Discount = 0.9  
Living reward = 0

k=7



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=8



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=9



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=10



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=11



Noise = 0.2  
Discount = 0.9  
Living reward = 0



# k=12



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# k=100



Noise = 0.2  
Discount = 0.9  
Living reward = 0

# “Policy” Iteration

- Alternative approach for optimal values:
  - **Step 1: Policy evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - **Step 2: Policy improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges
- This is called **policy iteration**
  - It's still optimal!
  - Can converge (much) faster under some conditions

# Policy Iteration

- Evaluation: For fixed current policy  $\pi$ , find values with policy evaluation:
  - Iterate until values converge:

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') [R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')]$$

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^{\pi_i}(s')]$$

# Comparison

---

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)
- Both are dynamic programs for solving MDPs

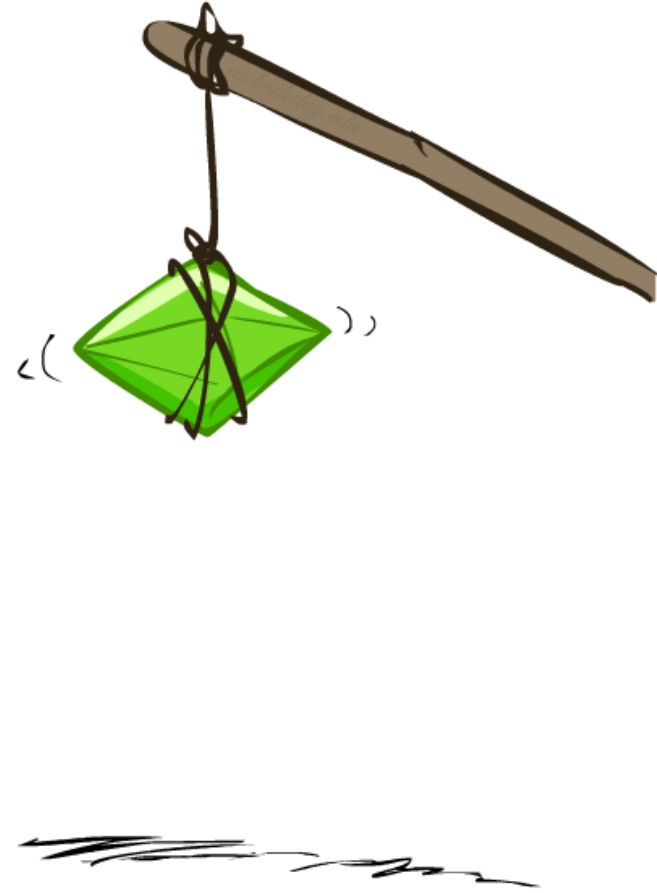
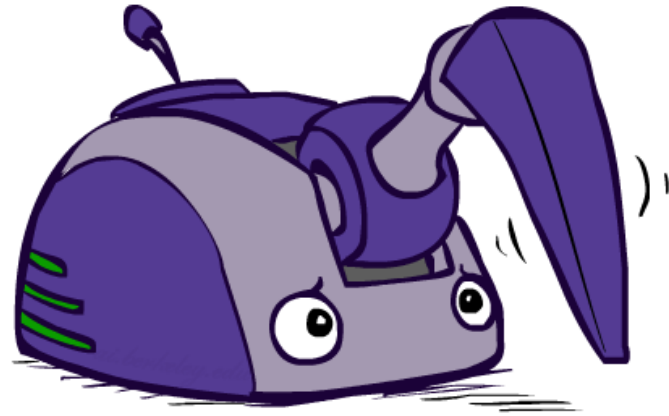
# Summary: MDP Algorithms

---

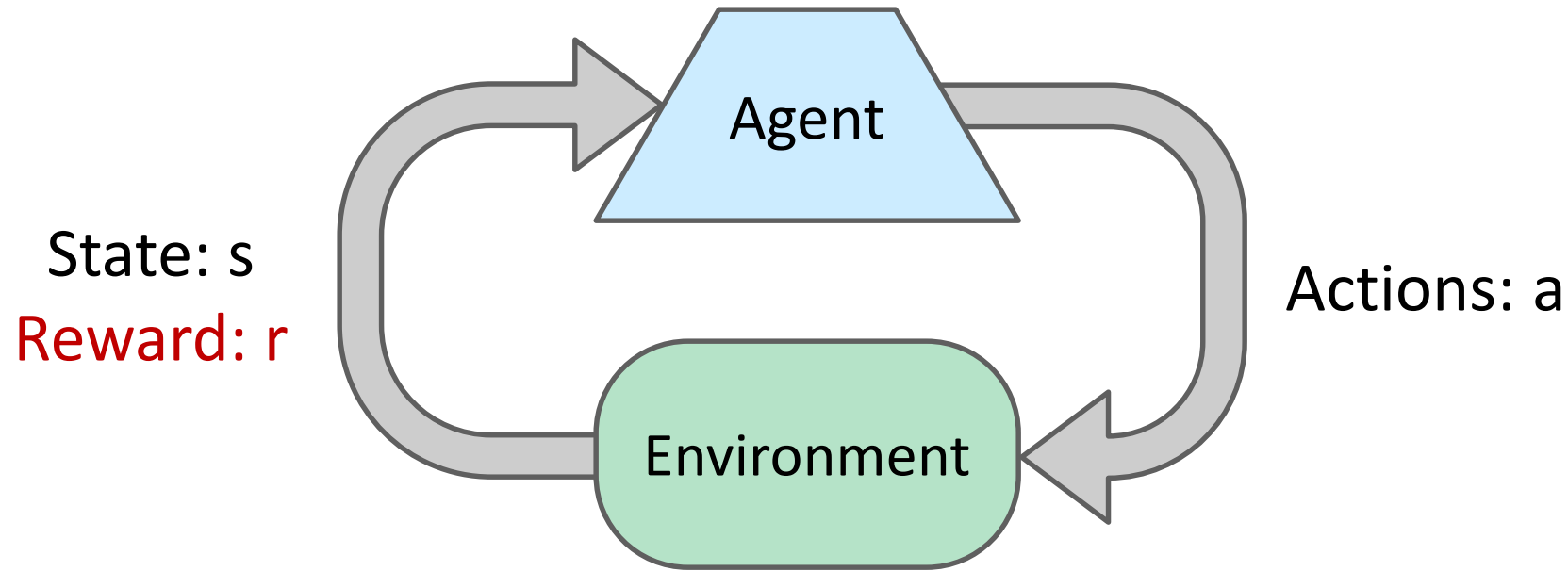
- So you want to....
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)
- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions

# Reinforcement Learning

---



# Reinforcement Learning



- Basic idea:
  - Receive feedback in the form of **rewards**
  - Agent's utility is defined by the reward function
  - Must (learn to) act so as to **maximize expected rewards**
  - All learning is based on observed samples of outcomes!



# Example: Learning to Walk



Initial



A Learning Trial



After Learning [1K Trials]

# Example: Learning to Walk



Initial

# Example: Learning to Walk



Training

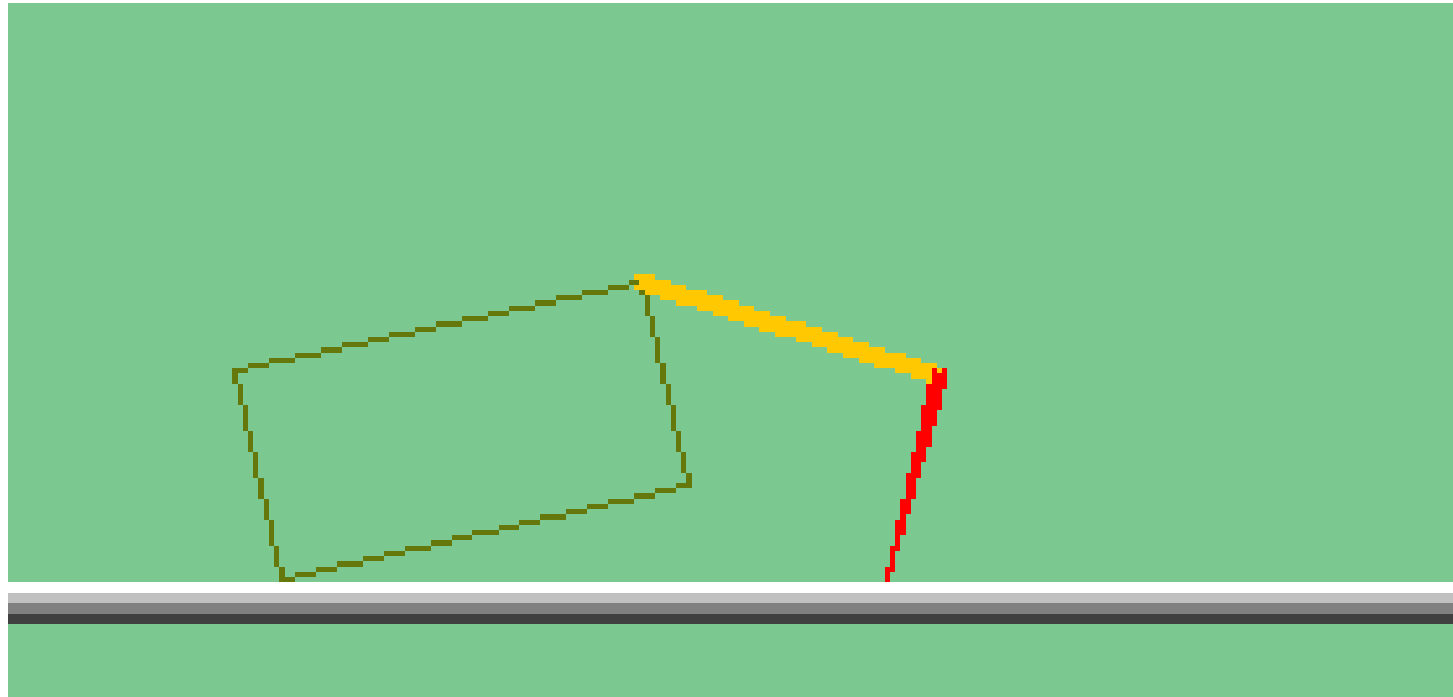
# Example: Learning to Walk



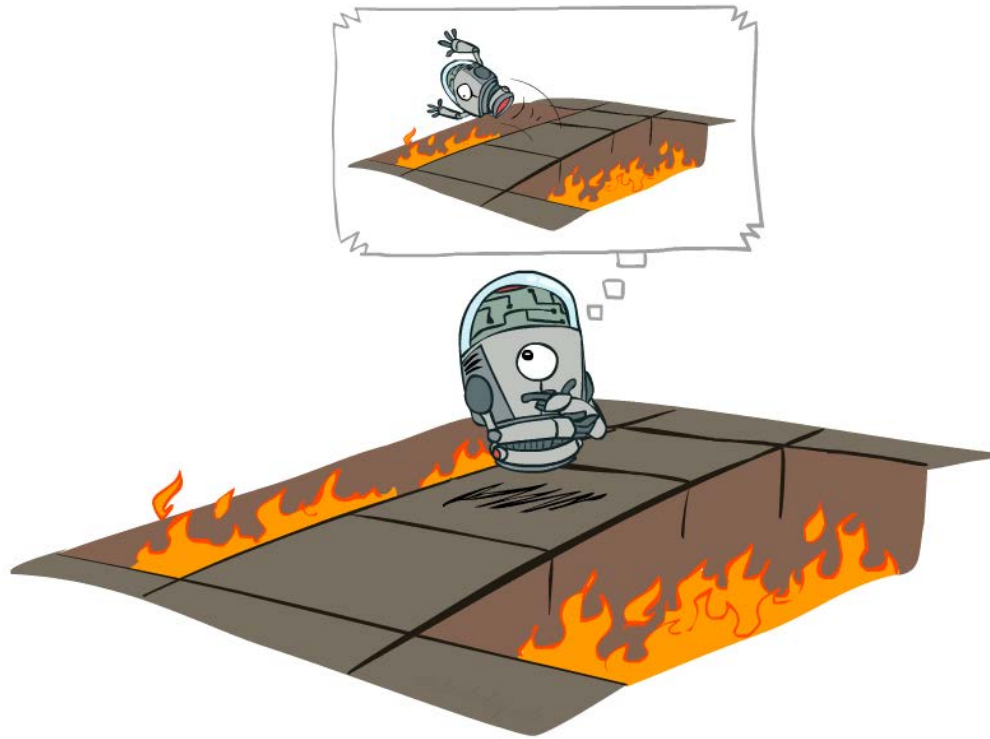
Finished

# The Crawler!

---



# Offline (MDPs) vs. Online (RL)



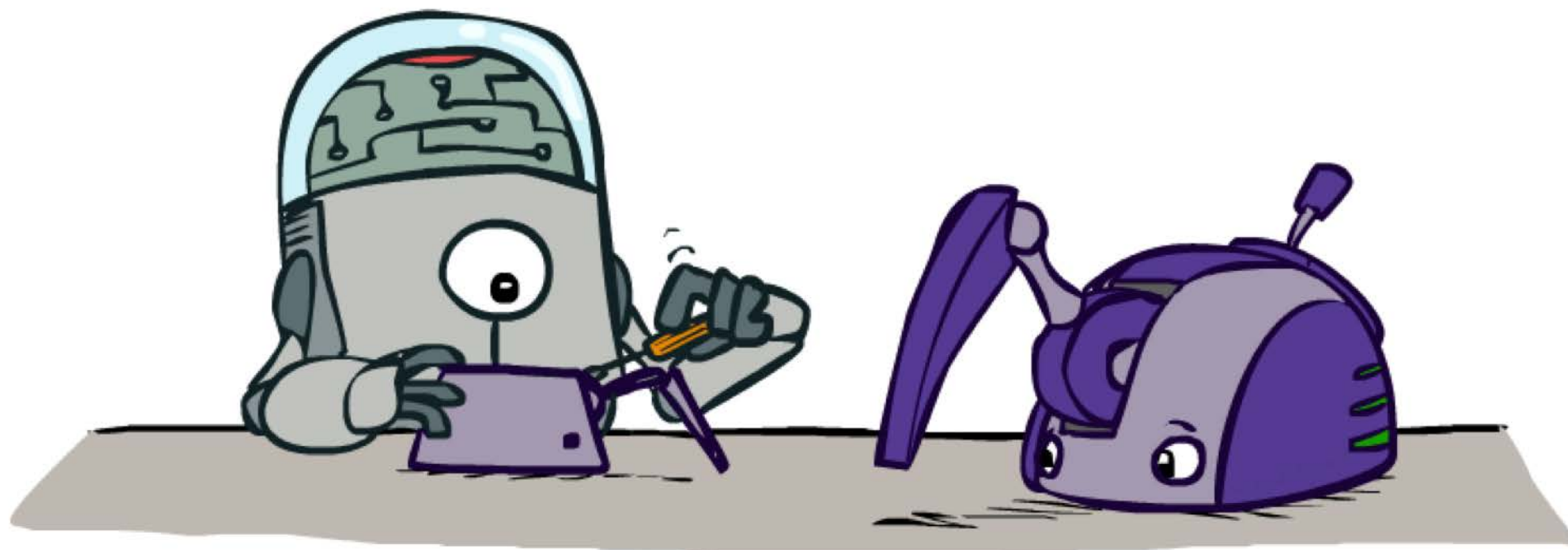
Offline Solution



Online Learning

# Model-Based Learning

---



# Model-Based Learning

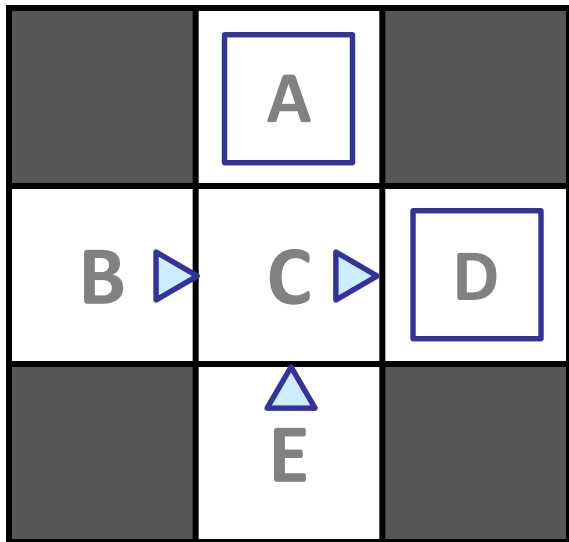
- **Model-Based Idea:**
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct
- **Step 1: Learn empirical MDP model**
  - Count outcomes  $s'$  for each  $s, a$
  - Normalize to give an estimate of  $\hat{T}(s, a, s')$
  - Discover each  $\hat{R}(s, a, s')$  when we experience  $(s, a, s')$
- **Step 2: Solve the learned MDP**
  - For example, use value iteration, as before





# Example: Model-Based Learning

Input Policy  $\pi$



Assume:  $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1  
C, east, D, -1  
D, exit, x, +10

Episode 2

B, east, C, -1  
C, east, D, -1  
D, exit, x, +10

Episode 3

E, north, C, -1  
C, east, D, -1  
D, exit, x, +10

Episode 4

E, north, C, -1  
C, east, A, -1  
A, exit, x, -10

Learned Model

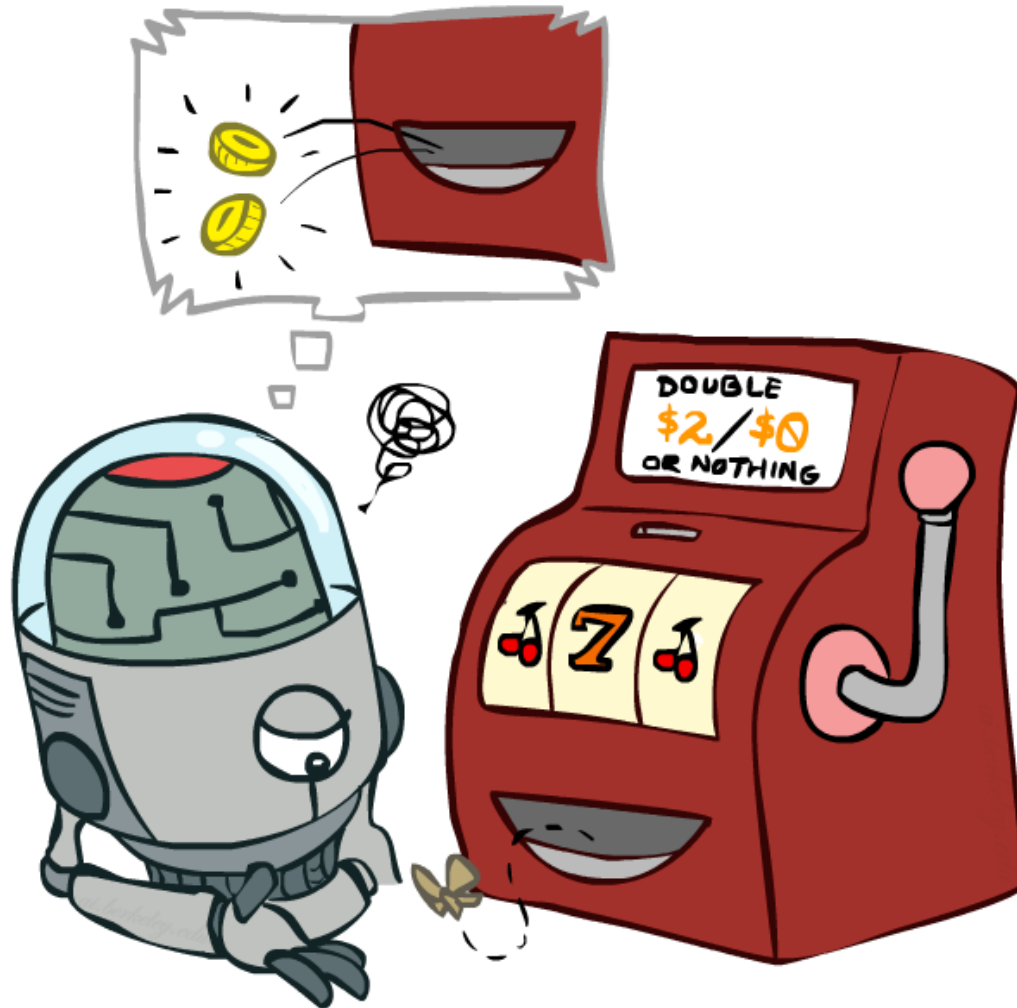
$$\hat{T}(s, a, s')$$

T(B, east, C) = 1.00  
T(C, east, D) = 0.75  
T(C, east, A) = 0.25  
...

$$\hat{R}(s, a, s')$$

R(B, east, C) = -1  
R(C, east, D) = -1  
R(D, exit, x) = +10  
...

# Model-Free Learning



# Example: Expected Age

Goal: Compute expected age of a group of students

Known  $P(A)$

$$E[A] = \sum_a P(a) \cdot a = 0.35 \times 20 + \dots$$

Without  $P(A)$ , instead collect samples  $[a_1, a_2, \dots, a_N]$

Unknown  $P(A)$ : "Model Based"

$$\hat{P}(a) = \frac{\text{num}(a)}{N}$$
$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

Why does this work? Because eventually you learn the right model.

Unknown  $P(A)$ : "Model Free"

$$E[A] \approx \frac{1}{N} \sum_i a_i$$

Why does this work? Because samples appear with the right frequencies.

# Detour: Q-Value Iteration

- Value iteration: find successive (depth-limited) values

- Start with  $V_0(s) = 0$ , which we know is right
- Given  $V_k$ , calculate the depth  $k+1$  values for all states:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

- But Q-values are more useful, so compute them instead

- Start with  $Q_0(s,a) = 0$ , which we know is right
- Given  $Q_k$ , calculate the depth  $k+1$  q-values for all q-states:

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$$

# Q-Learning

- Q-Learning: sample-based Q-value iteration

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right]$$

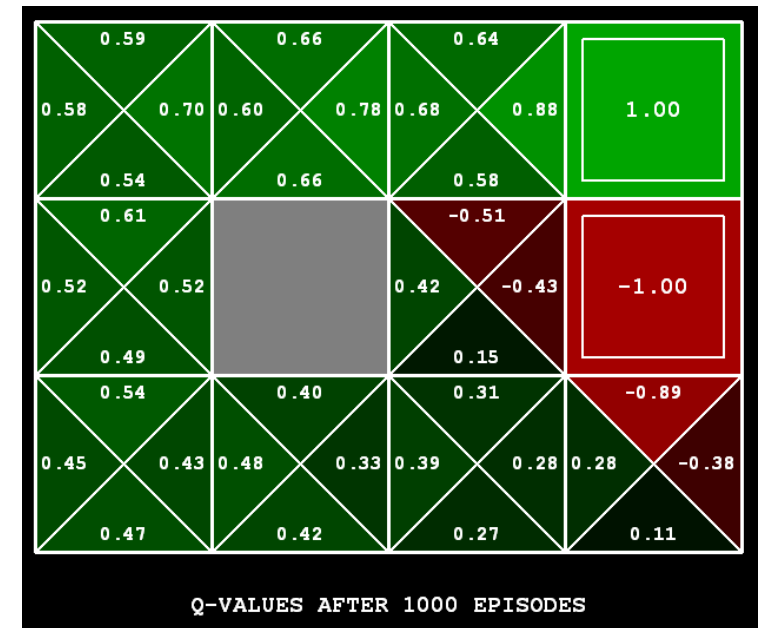
- Learn  $Q(s,a)$  values as you go

- Receive a sample  $(s,a,s',r)$
- Consider your old estimate:  $Q(s, a)$
- Consider your new sample estimate:

$$sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$

- Incorporate the new estimate into a running average:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha) [sample]$$

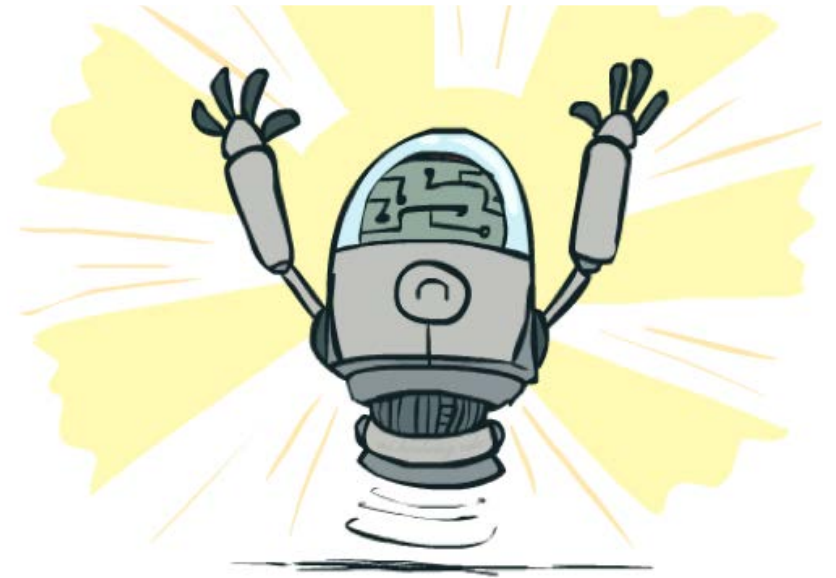


[Demo: Q-learning – gridworld (L10D2)]

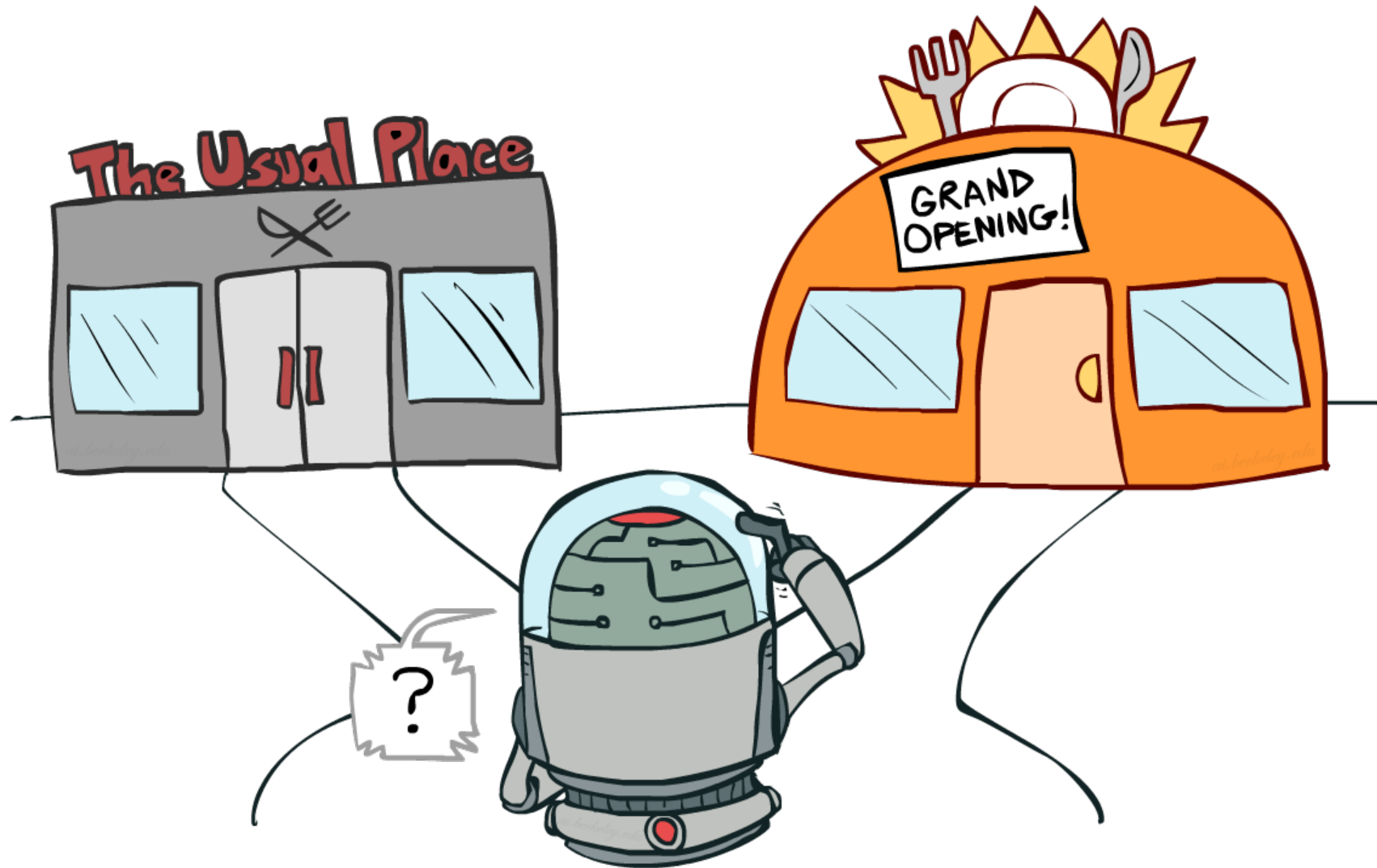
[Demo: Q-learning – crawler (L10D3)]

# Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called **off-policy learning**
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - ... but not decrease it too quickly
  - Basically, in the limit, it doesn't matter how you select actions (!)



# Exploration vs. Exploitation



# How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions ( $\epsilon$ -greedy)
    - Every time step, flip a coin
    - With (small) probability  $\epsilon$ , act randomly
    - With (large) probability  $1-\epsilon$ , act on current policy
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
    - One solution: lower  $\epsilon$  over time
    - Another solution: exploration functions



[Demo: Q-learning – manual exploration – bridge grid (L11D2)]

[Demo: Q-learning – epsilon-greedy -- crawler (L11D3)]



# Exploration Functions

- When to explore?

- Random actions: explore a fixed amount
- Better idea: explore areas whose badness is not (yet) established, eventually stop exploring

- Exploration function

- Takes a value estimate  $u$  and a visit count  $n$ , and returns an optimistic utility, e.g.  $f(u, n) = u + k/n$

Regular Q-Update:  $Q(s, a) \leftarrow_{\alpha} R(s, a, s') + \gamma \max_{a'} Q(s', a')$

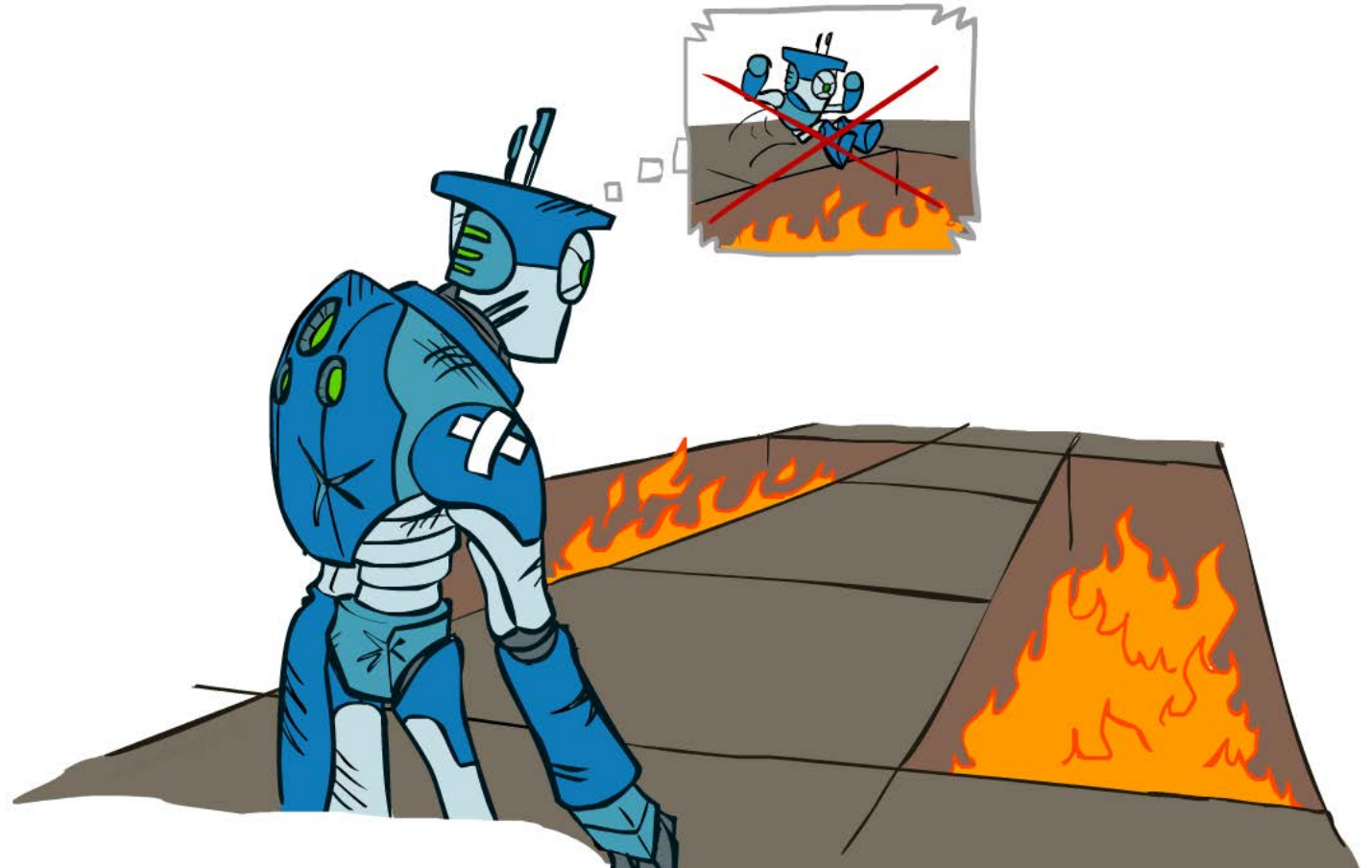
Modified Q-Update:  $Q(s, a) \leftarrow_{\alpha} R(s, a, s') + \gamma \max_{a'} f(Q(s', a'), N(s', a'))$

- Note: this propagates the “bonus” back to states that lead to unknown states as well!

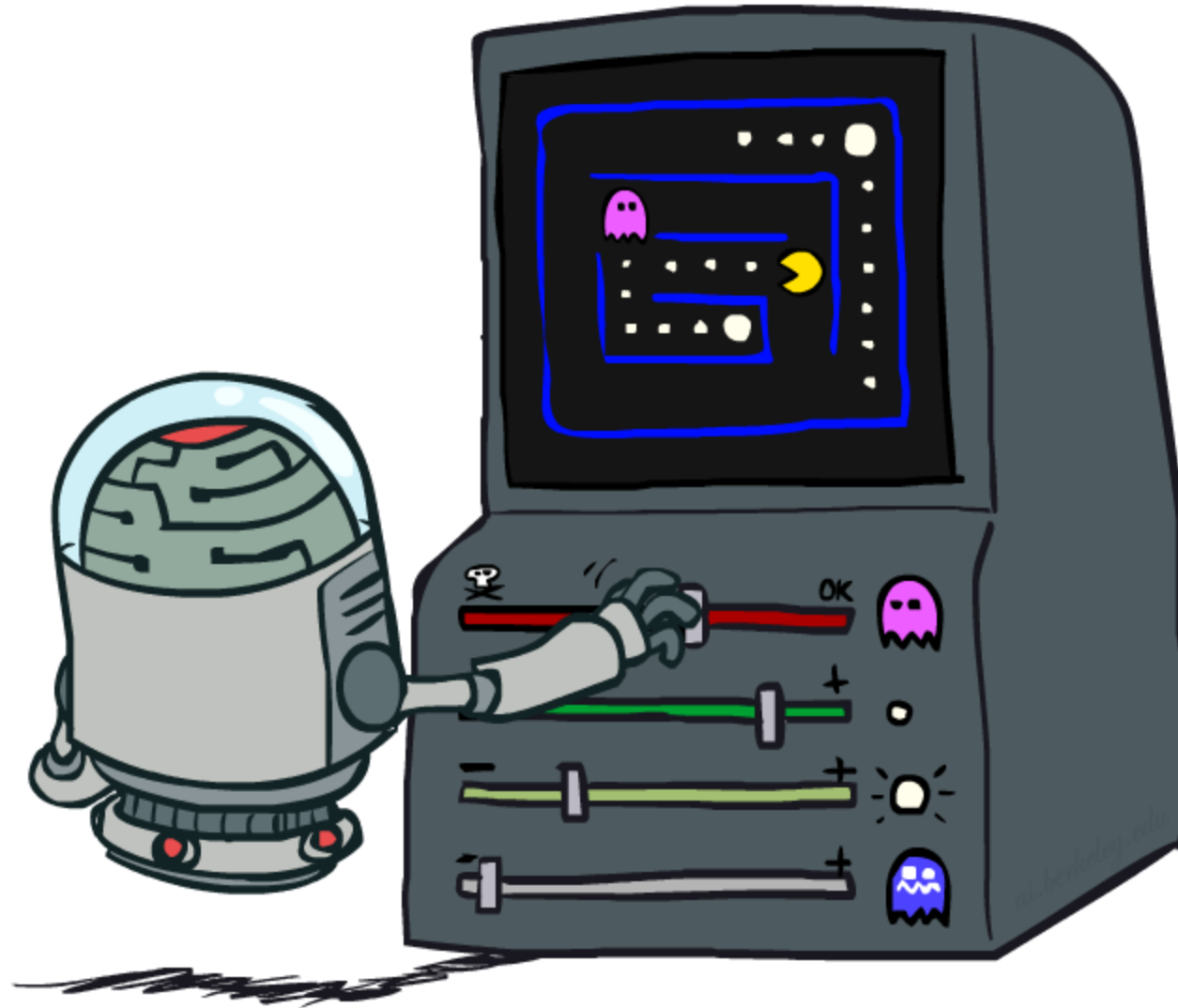


# Regret

- Even if you learn the optimal policy, you still make mistakes along the way
- Regret is a measure of your total mistake cost: the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards
- Minimizing regret goes beyond learning to be optimal – it requires optimally learning to be optimal
- Example: random exploration and exploration functions both end up optimal, but random exploration has higher regret

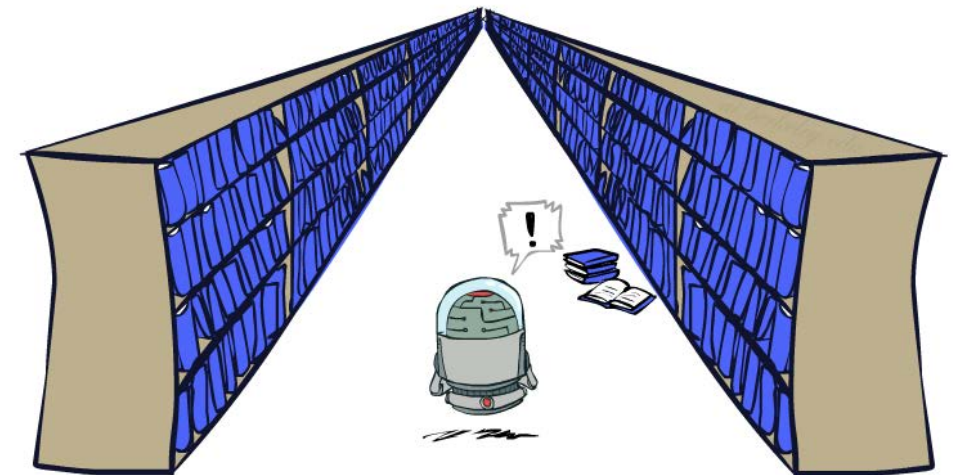
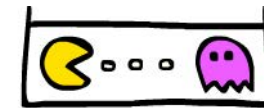
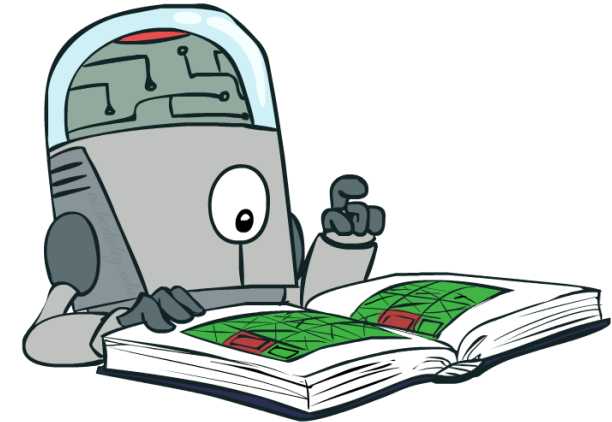


# Approximate Q-Learning



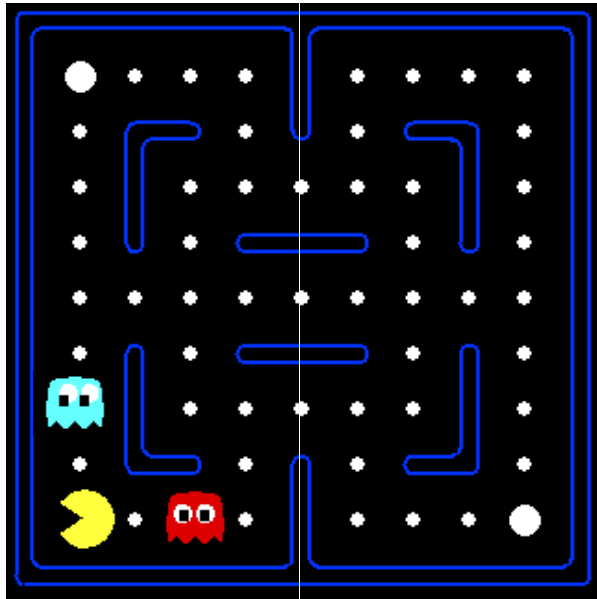
# Generalizing Across States

- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning, and we'll see it over and over again

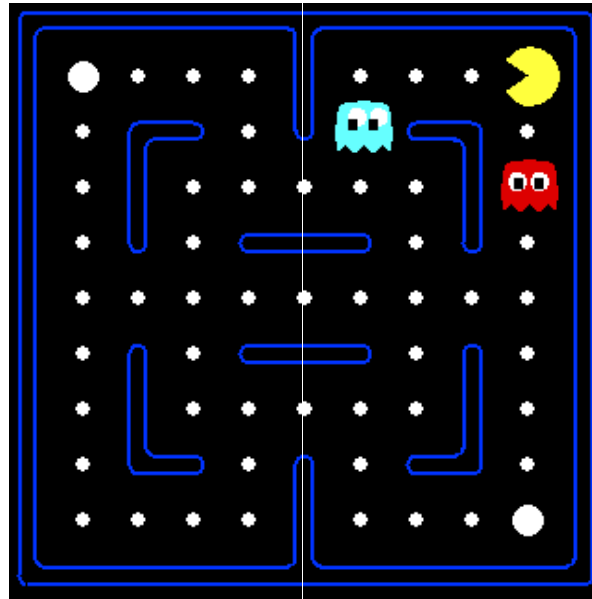


# Example: Pacman

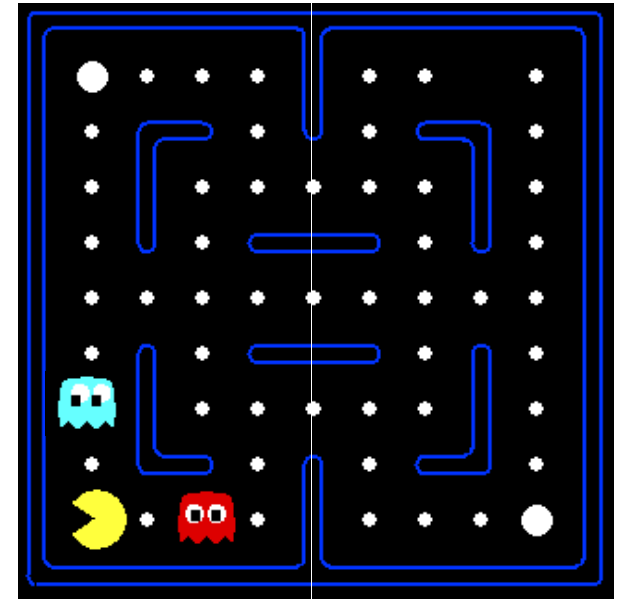
Let's say we discover through experience that this state is bad:



In naïve q-learning, we know nothing about this state:



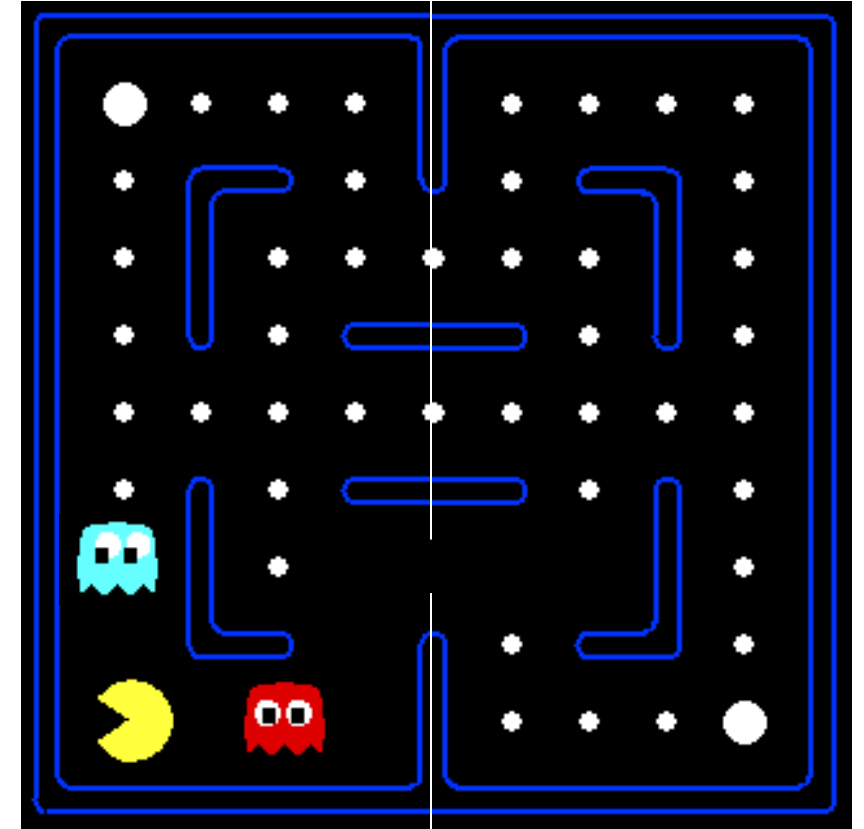
Or even this one!



[Demo: Q-learning – pacman – tiny – watch all (L11D5)]  
[Demo: Q-learning – pacman – tiny – silent train (L11D6)]  
[Demo: Q-learning – pacman – tricky – watch all (L11D7)]

# Feature-Based Representations

- Solution: describe a state using a vector of features (properties)
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - $1 / (\text{dist to dot})^2$
    - Is Pacman in a tunnel? (0/1)
    - ..... etc.
    - Is it the exact state on this slide?
  - Can also describe a q-state  $(s, a)$  with features (e.g. action moves closer to food)



# Linear Value Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a)$$

- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but actually be very different in value!

# Approximate Q-Learning

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a)$$

- Q-learning with linear Q-functions:

$$\text{transition} = (s, a, r, s')$$

$$\text{difference} = \left[ r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a)$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha [\text{difference}]$$

$$w_i \leftarrow w_i + \alpha [\text{difference}] f_i(s, a)$$

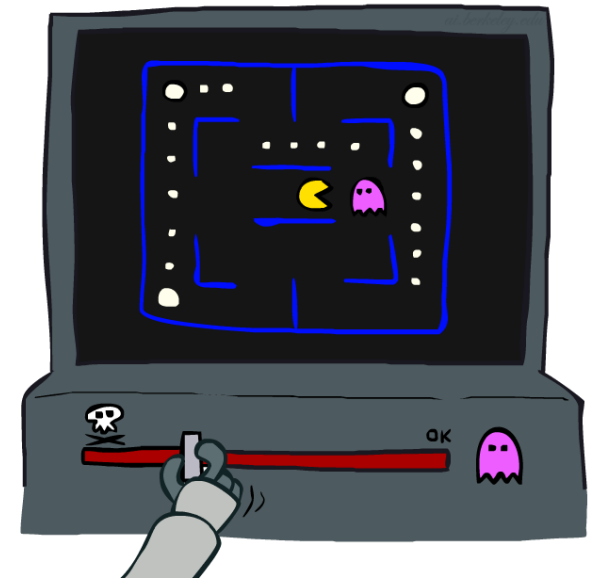
- Intuitive interpretation:

- Adjust weights of active features
- E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

- Formal justification: online least squares

Exact Q's

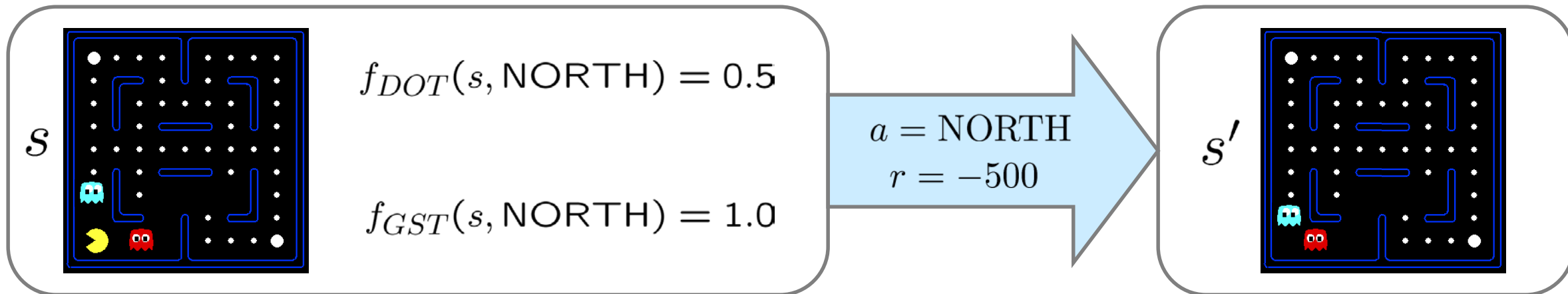
Approximate Q's





# Example: Q-Pacman

$$Q(s, a) = 4.0 f_{DOT}(s, a) - 1.0 f_{GST}(s, a)$$



$$f_{DOT}(s, \text{NORTH}) = 0.5$$

$$f_{GST}(s, \text{NORTH}) = 1.0$$

$$Q(s, \text{NORTH}) = +1$$

$$r + \gamma \max_{a'} Q(s', a') = -500 + 0$$

$$Q(s', \cdot) = 0$$

difference = -501



$$w_{DOT} \leftarrow 4.0 + \alpha [-501] 0.5$$

$$w_{GST} \leftarrow -1.0 + \alpha [-501] 1.0$$

$$Q(s, a) = 3.0 f_{DOT}(s, a) - 3.0 f_{GST}(s, a)$$

# Deep Q-Learning

- The approximate Q-learning was great if we picked the right features
- As throughout the whole course, why (deep) neural networks are great is that we can train end-to-end and do not need to handcraft features
- Simple way out: let just train a neural network to spit out the q values for a given state and action

## LETTER

doi:10.1038/nature14236

### Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>2\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Höglund<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadic<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharmhan Kumaran<sup>1</sup>, Dan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems<sup>1,2</sup>, the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms<sup>3</sup>. While reinforcement learning agents have achieved some successes in a variety of domains<sup>4,5</sup>, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks<sup>6–11</sup> to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games<sup>12</sup>. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human gamer across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

We set out to create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks—a central goal of general artificial intelligence<sup>13</sup> that has eluded previous efforts<sup>14,15</sup>. To achieve this, we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural networks known as deep neural networks. Notably, recent advances in deep neural networks<sup>16–21</sup>, in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data. We use one particularly successful architecture, the deep convolutional network<sup>22</sup>, which uses hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields—inspired by Hubel and Wiesel's seminal work on feedforward processing in early visual cortex<sup>23</sup>—thereby exploiting the local spatial correlations present in images, and building in robustness to natural transformations such as changes of viewpoint or scale.

We consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the

agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

$$Q^*(s, a) = \max_a \mathbb{E} [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi],$$

which is the maximum sum of rewards  $r_t$  discounted by  $\gamma$  at each time-step  $t$ , achievable by a behaviour policy  $\pi = P(a|s)$ , after making an observation ( $s$ ) and taking an action ( $a$ ) (see Methods)<sup>19</sup>.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as  $Q$ ) function<sup>24</sup>. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to  $Q$  may significantly change the policy and therefore change the data distribution, and the correlations between the action-values ( $Q$ ) and the target values  $r + \gamma \max_{a'} Q(s, a')$ . We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay<sup>25–28</sup> that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values ( $Q$ ) towards target values that are only periodically updated, thereby reducing correlations with the target.

While other stable methods exist for training neural networks in the reinforcement learning setting, such as neural fitted Q-iteration<sup>29</sup>, these methods involve the repeated training of networks *de novo* on hundreds of iterations. Consequently, these methods, unlike our algorithm, are too inefficient to be used successfully with large neural networks. We parameterize an approximate value function  $Q(s, a; \theta)$  using the deep convolutional neural networks shown in Fig. 1, in which  $\theta_i$  are the parameters (that is, weights) of the Q-network at iteration  $i$ . To perform experience replay we store the agent's experiences  $e_t = (s_t, a_t, r_t, s_{t+1})$  at each time-step  $t$  in a data set  $D_t = \{e_0, \dots, e_t\}$ . During learning, we apply Q-learning updates, on samples (or minibatches) of experience  $(s, a, r, s') \sim U(D)$ , drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration  $i$  uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s, a, r, s') \sim U(D)} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

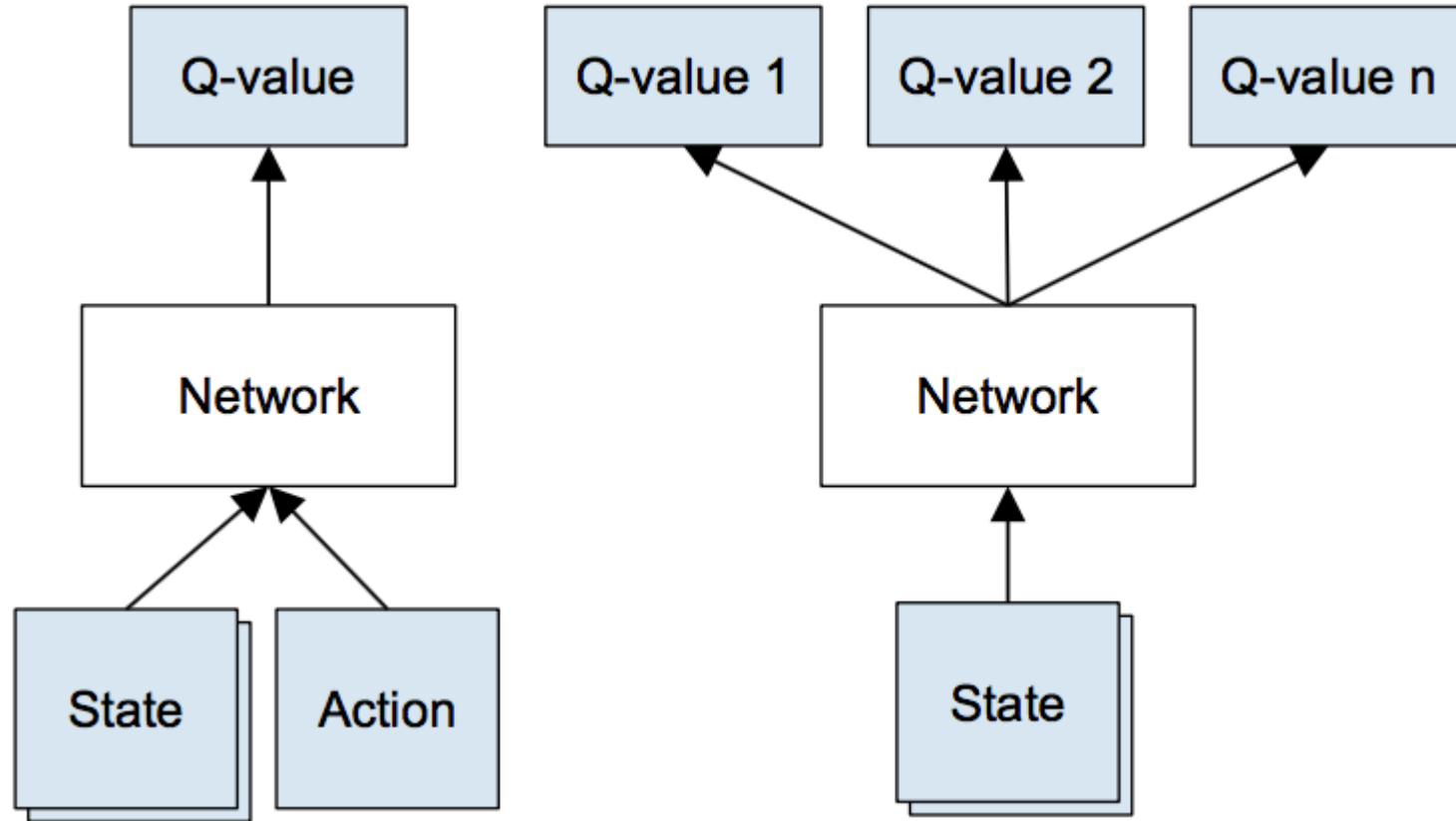
in which  $\gamma$  is the discount factor determining the agent's horizon,  $\theta_i$  are the parameters of the Q-network at iteration  $i$  and  $\theta_i^-$  are the network parameters used to compute the target at iteration  $i$ . The network parameters  $\theta_i^-$  are only updated with the Q-network parameters ( $\theta_i$ ) every  $C$  steps and are held fixed between individual updates (see Methods).

To evaluate our DQN agent, we took advantage of the Atari 2600 platform, which offers a diverse array of tasks ( $n = 49$ ) designed to be

<sup>1</sup>Google DeepMind, 5 New Street Square, London EC4A 3TW, UK.

\*These authors contributed equally to this work.

# A small trick



Left: Naive formulation of deep Q-network. Right: More optimized architecture of deep Q-network, used in DeepMind

# Network Architecture for Atari Game

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Loss function: 
$$L = \frac{1}{2} \left[ \underbrace{r + \max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}} \right]^2$$

# Stability Issues with Deep Q-Learning

---

Naïve Q-learning **oscillates** or **diverges** with neural nets

## 1. Data is sequential

- Successive samples are correlated, non-iid

## 2. Policy changes rapidly with slight changes to Q-values

- Policy may oscillate
- Distribution of data can swing from one extreme to another

## 3. Scale of rewards and Q-values

- Naïve Q-learning gradients can be large, unstable when backprop

# Stabilize Deep Q-Learning

---

## 1. Use **experience replay**

- Break correlations in data, bring us back to iid setting
- Learn from all past policies

## 2. Freeze **target Q-network**

- Avoid oscillations
- Break correlations between Q-network and target

## 3. **Clip** rewards or normalize network adaptively to sensible range

- Robust gradients

# Experience Replay

- To remove correlations, build data-set from agent's own experience
  - Take action  $a_t$  according to  $\epsilon$ -greedy policy
  - Store transition  $(s_t, a_t, r_{t+1}, s_{t+1})$  in replay memory  $D$
  - Sample random mini-batch of transitions  $(s, a, r, s')$  from  $D$
  - Optimize MSE between Q-network and Q-learning targets, e.g.,

$$L(w) = E_{s,a,r,s' \sim D} \left[ \left( r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w) \right)^2 \right]$$

# Fixed Target Q-Network

- To avoid oscillations, fix parameters used in Q-learning target
  - Compute Q-learning targets w.r.t. old, fixed parameters  $w^-$

$$r + \gamma \max_{a'} Q(s', a', w^-)$$

- Optimize MSE between Q-network and Q-learning targets

$$L(w) = E_{s,a,r,s' \sim D} \left[ \left( r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w) \right)^2 \right]$$

- Periodically update fixed parameters



# Reward/Value Range

---

- DQN clips the rewards to  $[-1, +1]$
- This prevents Q-values from becoming too large
- Ensures gradients are well-conditioned
- Can't tell difference between small and large rewards

# Improvements since Nature DQN

- **Double DQN:** Remove upward bias caused by  $\max_a Q(s, a, w)$ 
  - Current Q-network  $w$  is used to **select** actions
  - Older Q-network  $w^-$  is used to **evaluate** actions

$$I = \left( r + \gamma Q \left( s', \underset{a'}{\operatorname{argmax}} Q(s', a', w), w^- \right) - Q(s, a, w) \right)^2$$

- **Prioritized replay:** Weight experience according to surprise
  - Store experience in priority queue according to DQN error

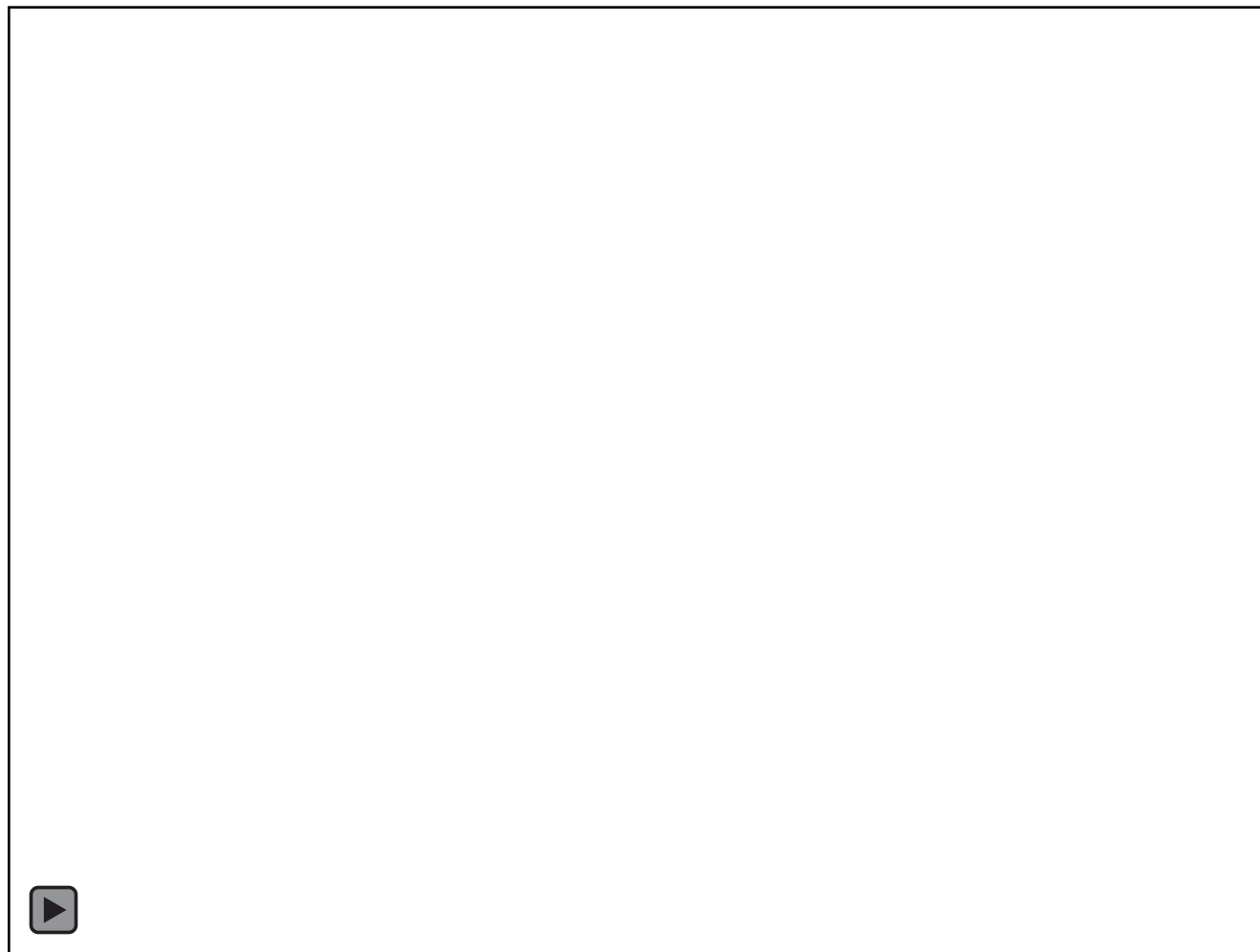
$$\left| r + \gamma \max_a Q(s', a', w^-) - Q(s, a, w) \right|$$

- **Deulling network:** Split Q-network into two channels
  - Action-independent **value function**  $V(s, v)$
  - Action-dependent **advantage function**  $A(s, a, w)$

$$Q(s, a) = V(s, v) + A(s, a, w)$$

# Playing Breakout

---



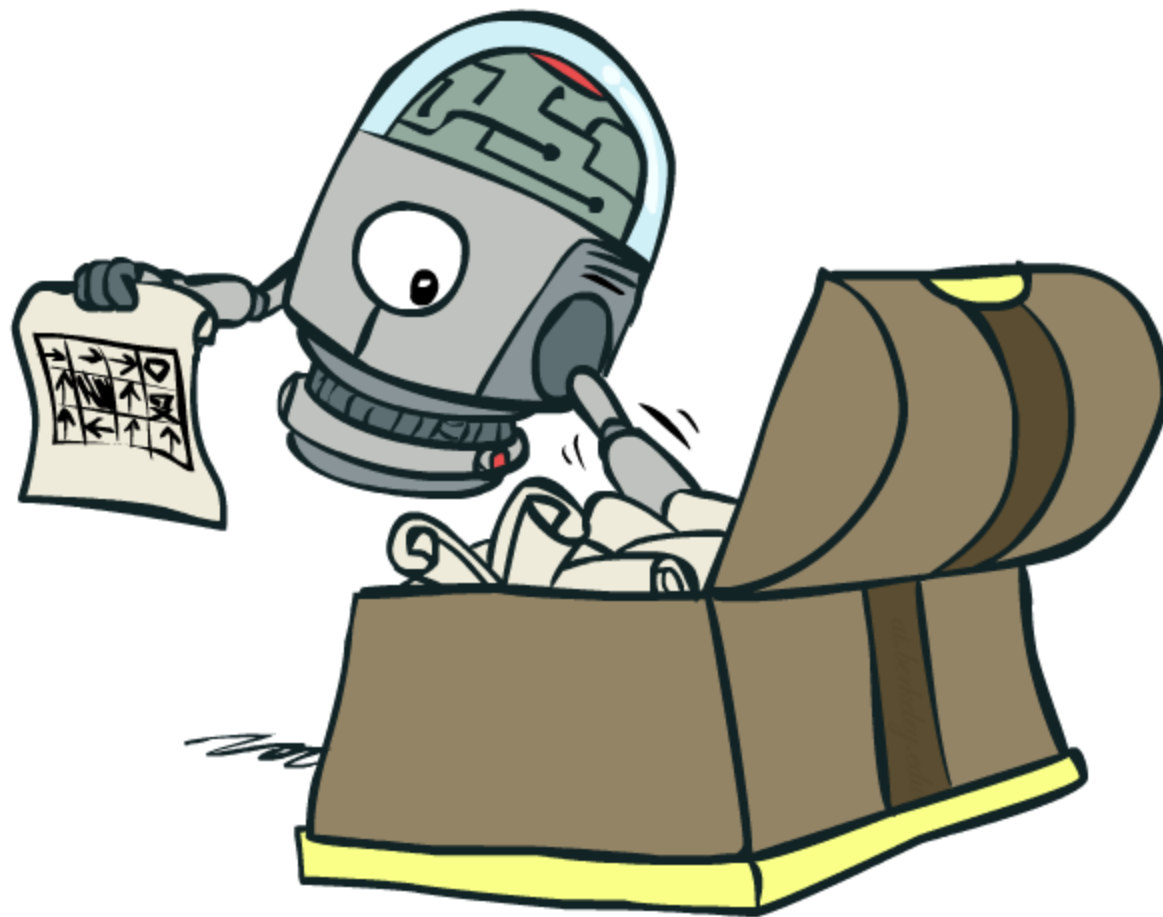
# Playing Invaders

---



# Policy Learning

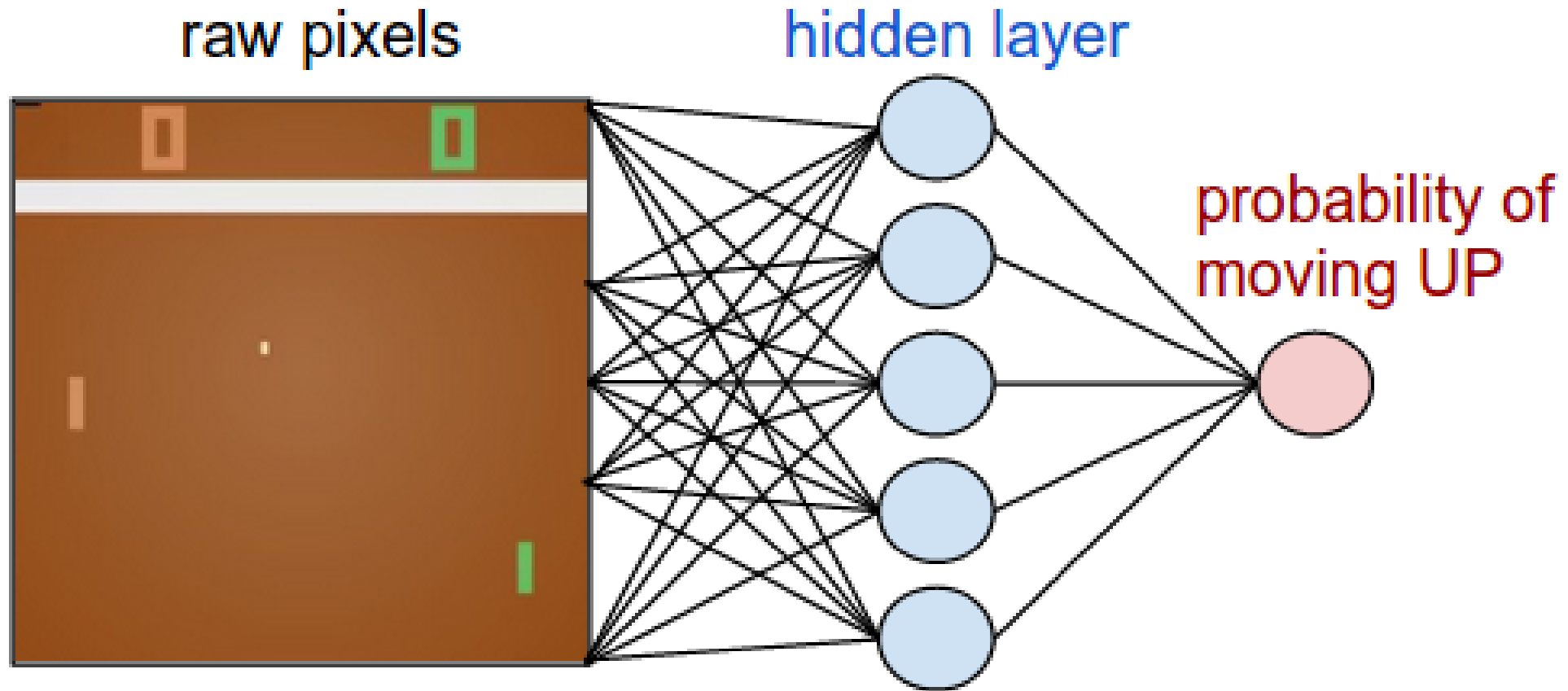
---



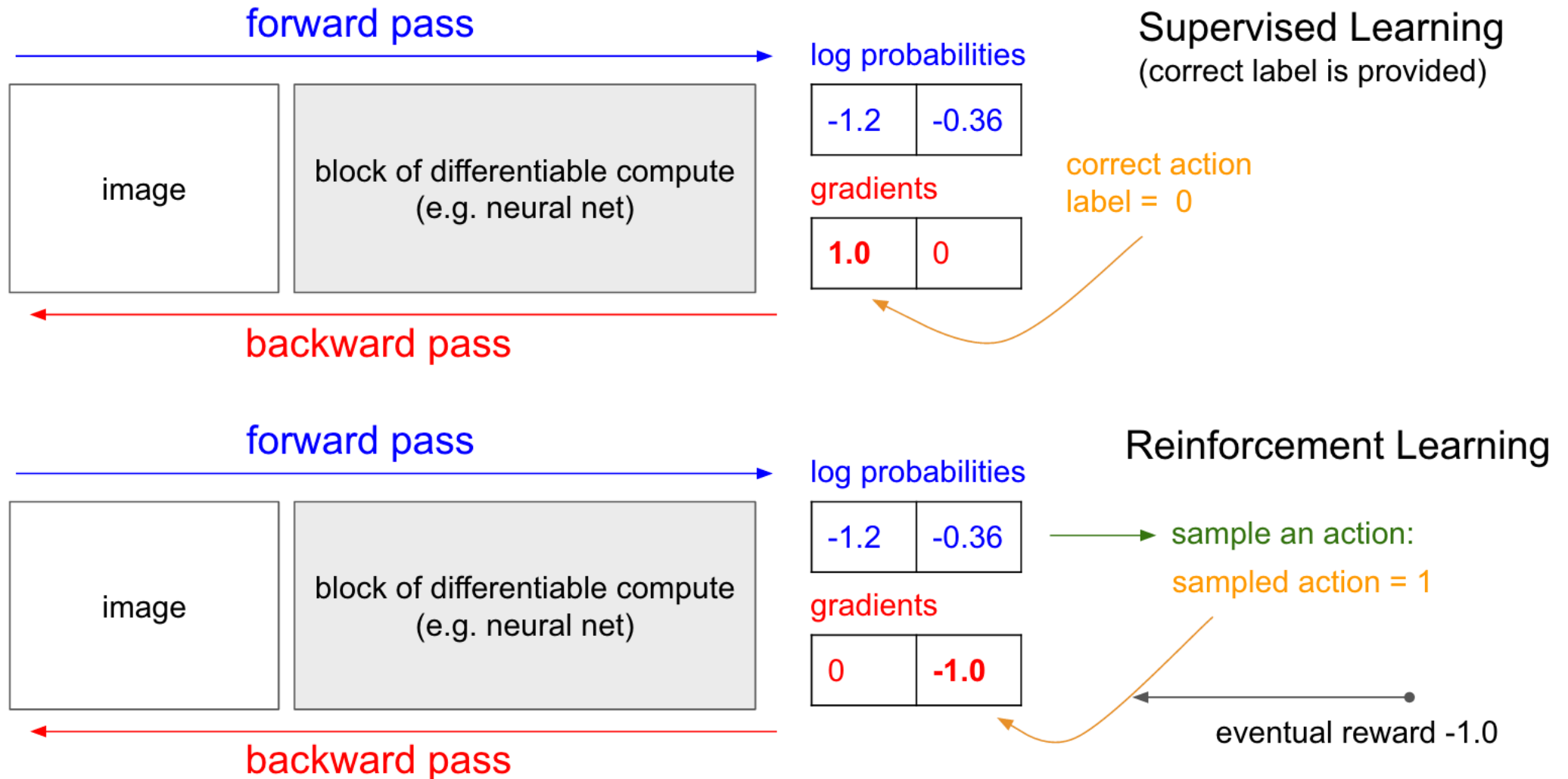
# Policy Learning

- Problem: often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate  $V / Q$  best
  - Q-learning's priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)
- Solution: learn policies that maximize rewards, not the values that predict them
- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights
- Alternative: **Build a policy network**

# Simple Policy Network

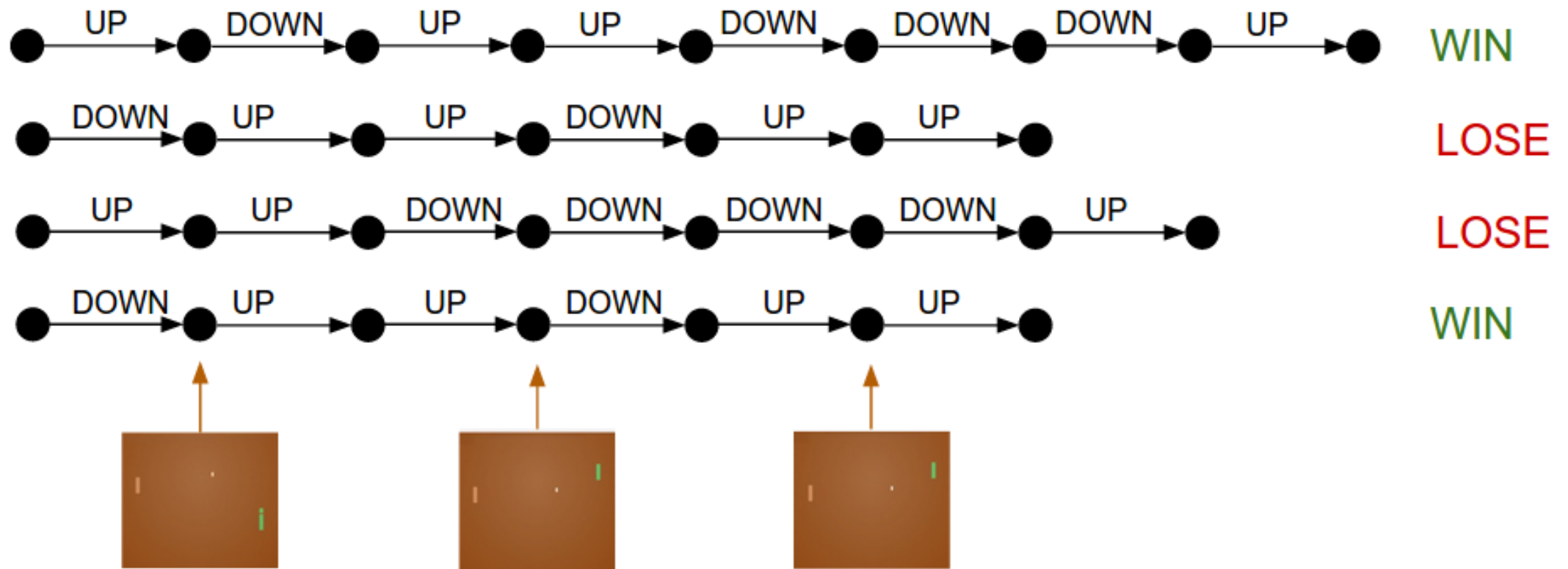


# Reinforcement Learning vs Supervised Learning





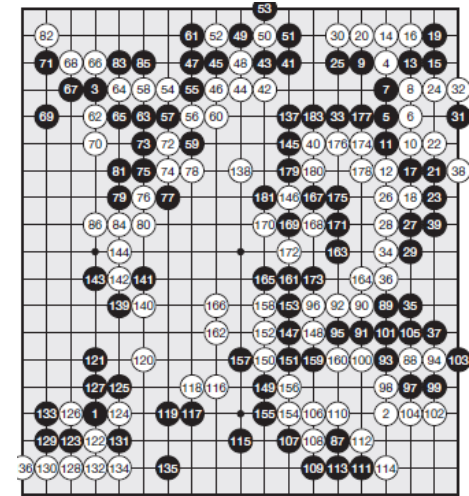
# Backprop Example



With Policy Gradients we would take the two games we won and slightly encourage every single action we made in that episode and vice versa

# AlphaGo Revisit

- Why Go is so difficult?
  - Number of combinations  $\approx 250^{180}$   
 $\approx 10^{430}$
- Comparatively, number of configurations for chess (Shannon number)  $\approx 35^{80} \approx 10^{120}$
- Number of atoms in the observable universe  $\approx 10^{82}$

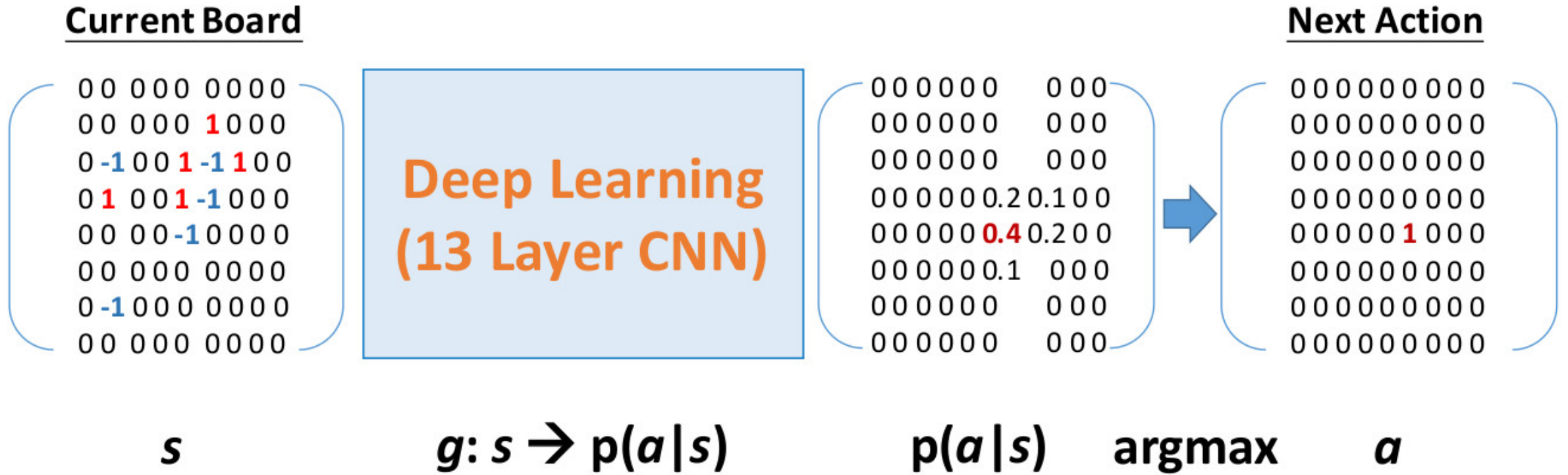


# AlphaGo's approach

---

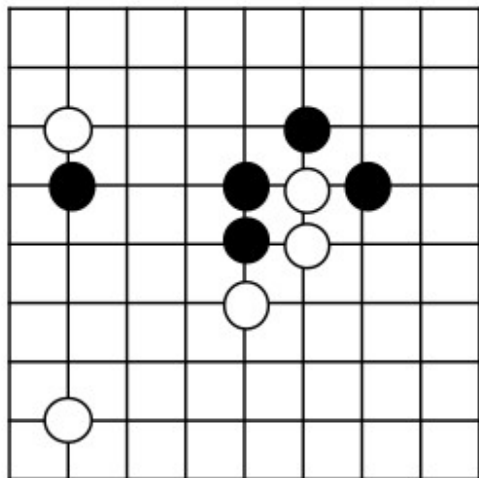
- Use a deep (CNN) policy network to imitate expert players' moves
- Use a value network to estimate the winning chance of each configuration
- ``Look ahead'' with Monte-Carlo tree search to do policy search

# AlphaGo's Policy Network



# AlphaGo's Value Network

Board position



win/loss

**Expert Moves Imitator Model  
(w/ CNN)**

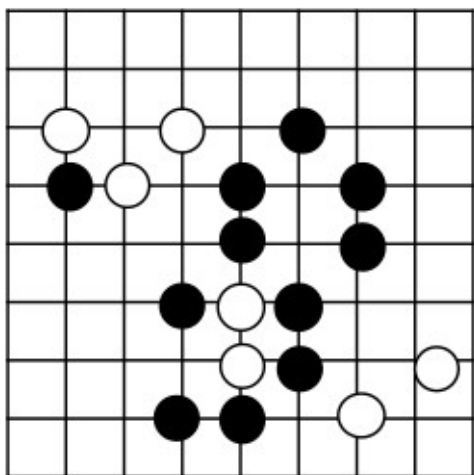
**Loss**

$z = -1$

**Training:**  $\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t.$

# AlphaGo's Value Network

Board position



win/loss

**Expert Moves Imitator Model  
(w/ CNN)**

**Win**

$z = +1$

**Training:**  $\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t.$

# Monte Carlo Tree Search

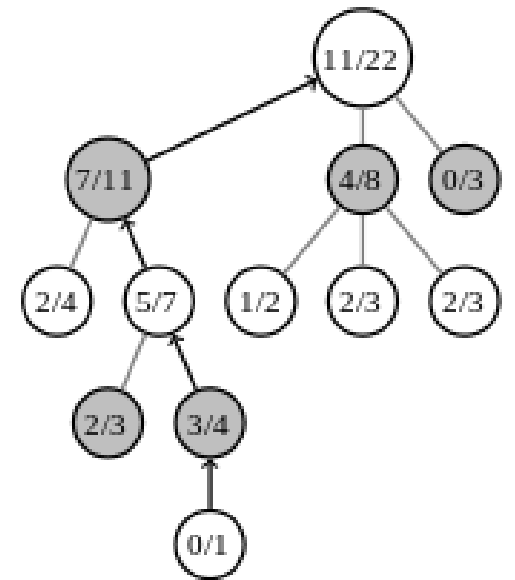
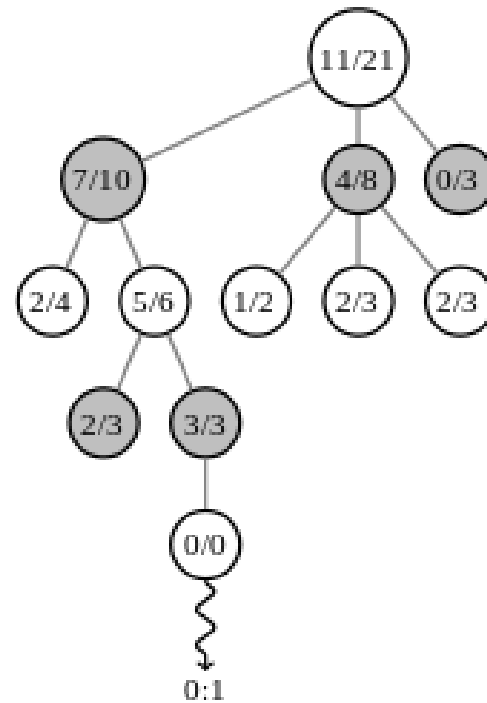
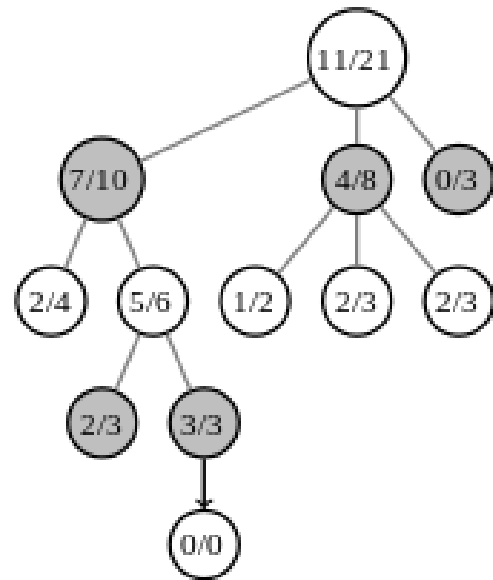
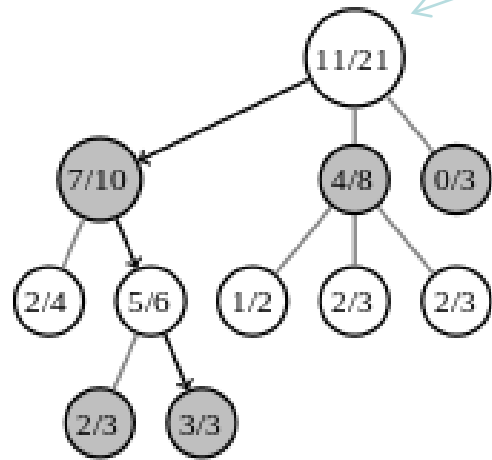
Node value: Wining rate

Selection

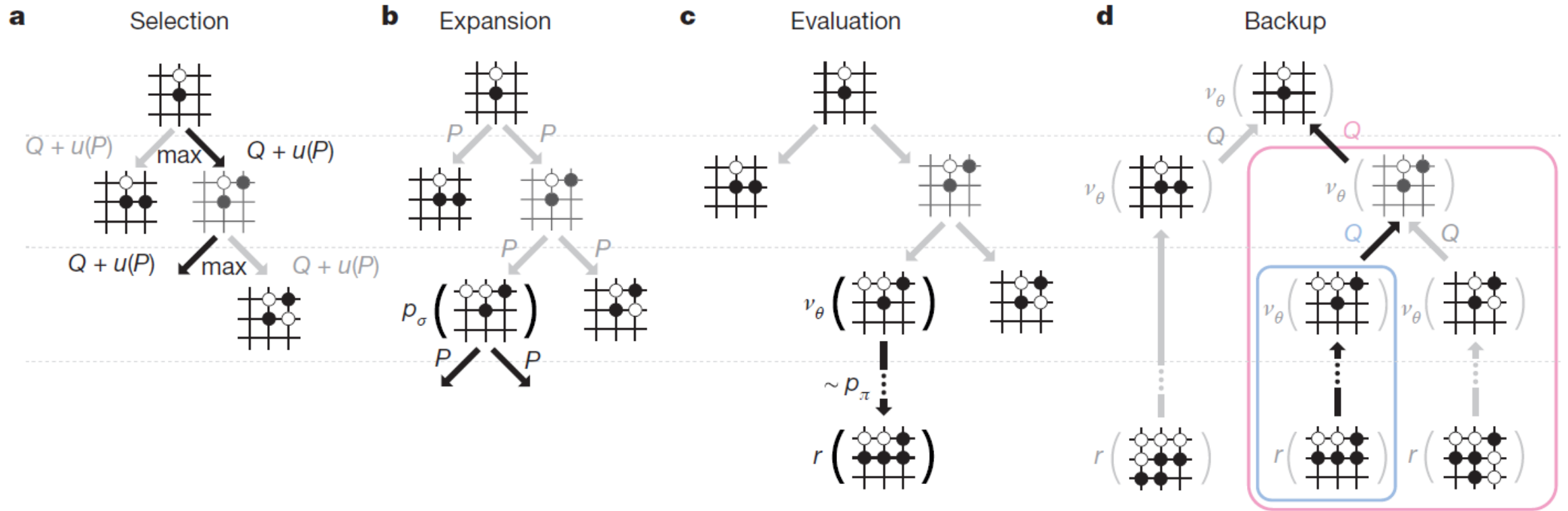
Expansion

Evaluation

Backpropagation



# Monte Carlo Tree Search (MCTS)



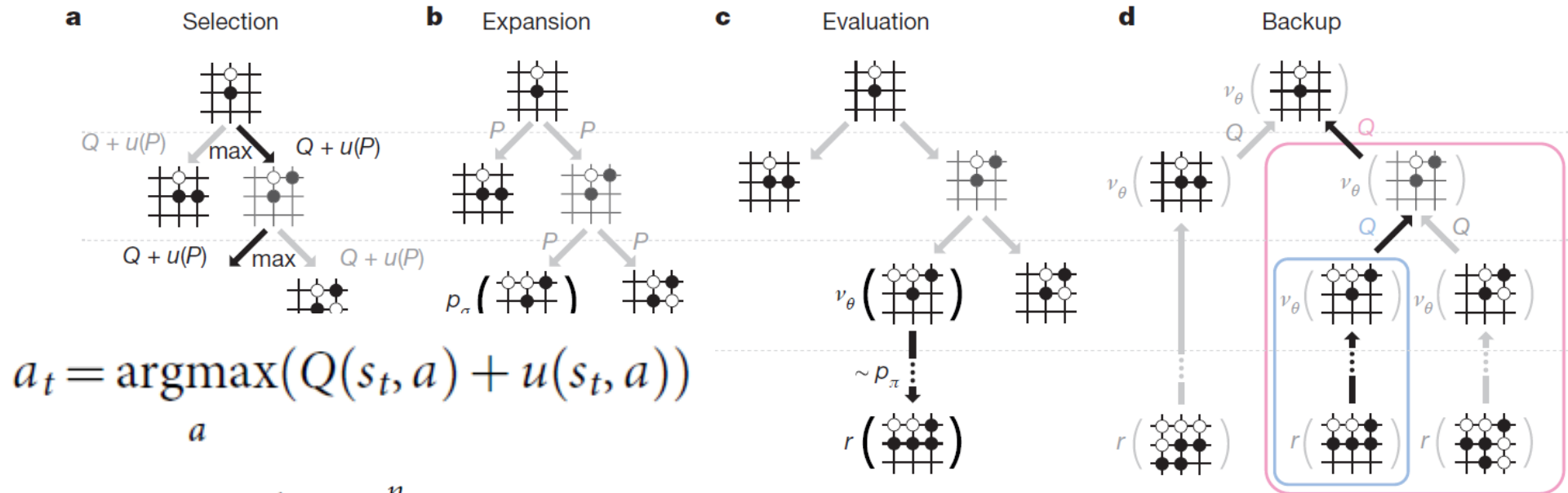
$$a_t = \operatorname{argmax}_a (Q(s_t, a) + u(s_t, a))$$

$a$

bonus



# AlphaGo in one Slide



$$a_t = \operatorname{argmax}_a (Q(s_t, a) + u(s_t, a))$$

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^n 1(s, a, i) V(s_L^i) \leftarrow \text{Winning rate estimate}$$

$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L \leftarrow \text{Winning rate predicted by fast rollout playout}$$

Winning rate predicted by value network

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)} \leftarrow \text{Action probability predicted by policy network}$$

# Conclusion

- We're done with a quick introduction on MDP, reinforcement learning, Q-learning and deep Q-learning
- Acknowledgement: thanks Dan Klein and Pieter Abbeel at UC Berkeley for making their slides available online!

