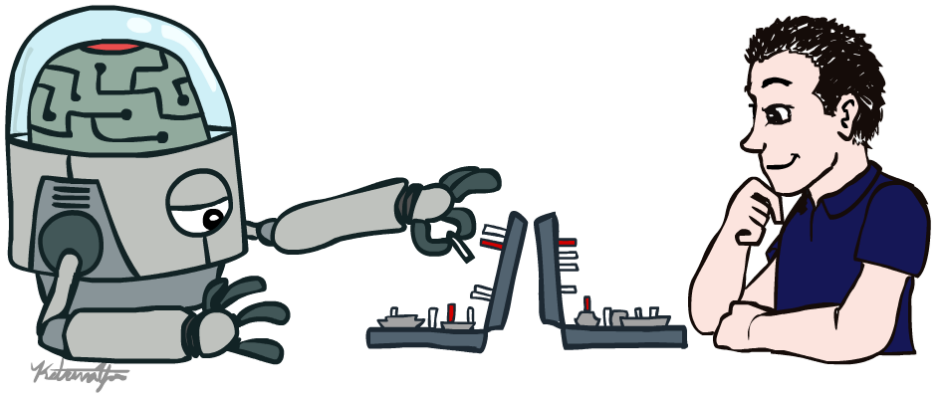
Introduction to Reinforcement Learning



[These slides were originally created by Dan Klein and Pieter Abbeel for *CS188 Intro to AI* at UC Berkeley (http://ai.berkeley.edu) – Edited by Aleksis Pirinen for *FMAN45 Machine Learning* at Lund University]

Human Saccade-and-Fixate Search

[S. Mathe, C. Sminchisescu, *Actions in the eye: dynamic gaze datasets and learnt saliency models for visual recognition*, PAMI 2015]

Human Saccade-and-Fixate Search

[S. Mathe, C. Sminchisescu, *Actions in the eye: dynamic gaze datasets and learnt saliency models for visual recognition*, PAMI 2015]

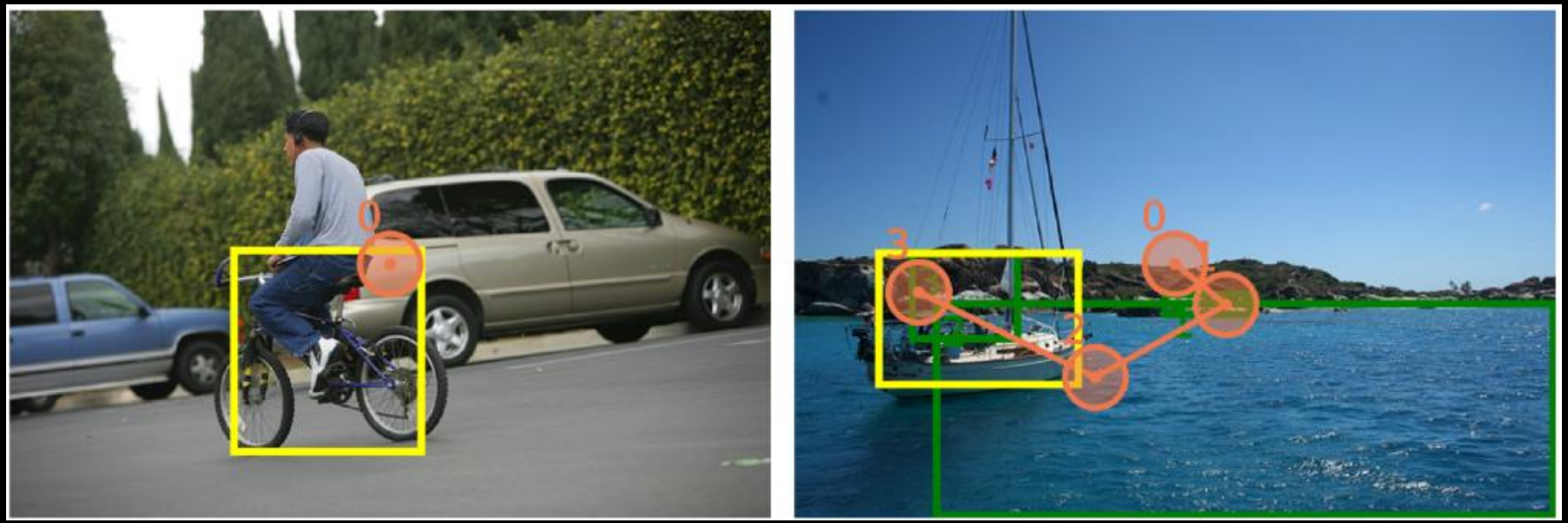
Human Saccade-and-Fixate Search

[S. Mathe, C. Sminchisescu, *Actions in the eye: dynamic gaze datasets and learnt saliency models for visual recognition*, PAMI 2015]

Human Saccade-and-Fixate Search

[S. Mathe, C. Sminchisescu, *Actions in the eye: dynamic gaze datasets and learnt saliency models for visual recognition*, PAMI 2015]

Saccade-and-Fixate Search via Reinforcement Learning



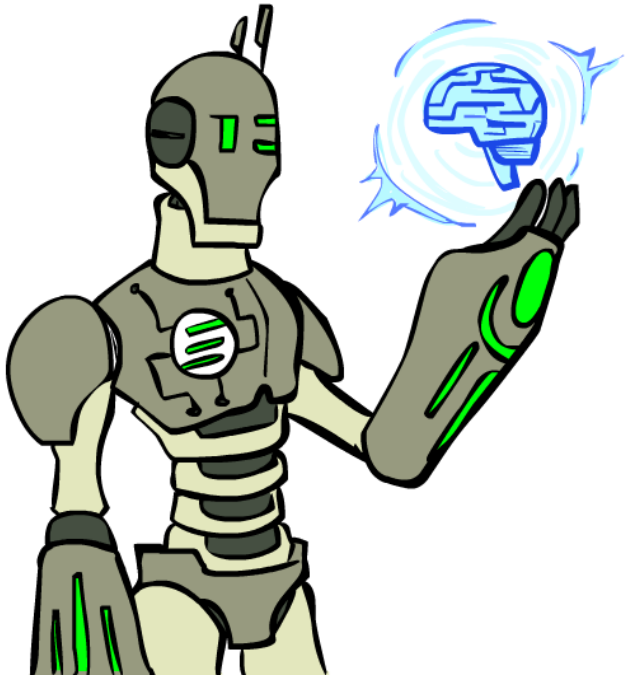
S. Mathe, A. Pirinen, C. Sminchisescu, *Reinforcement Learning for Visual Object Detection,* CVPR 2016

DeepDrive: Self-Driving Cars from Video Games

[http://deepdrive.io/]

Today

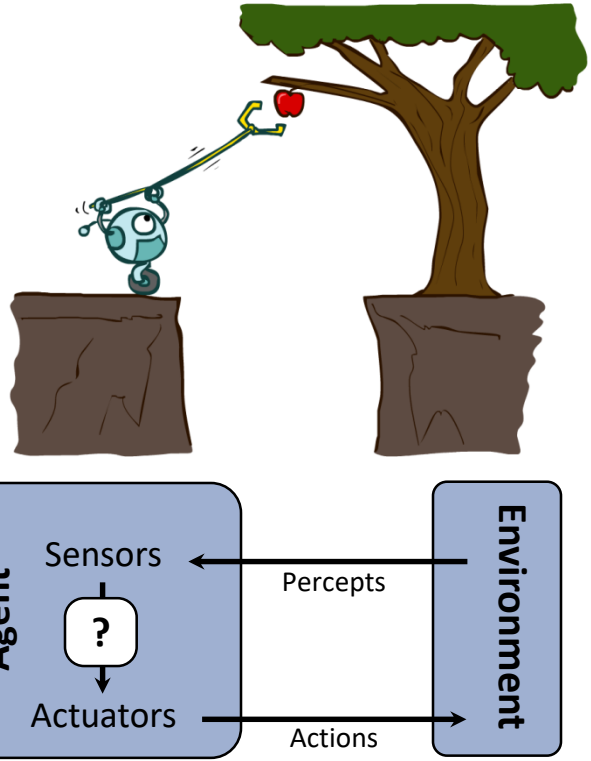
- Basics of Reinforcement Learning
  - Decision making and planning
  - Markov decision processes (MDPs)
  - Value-based policy evaluation and policy search in MDPs
  - Learning to act in unknown MDPs = Reinforcement learning
- Introduction to Assignment 4
  - Learning to play the game *Snake*



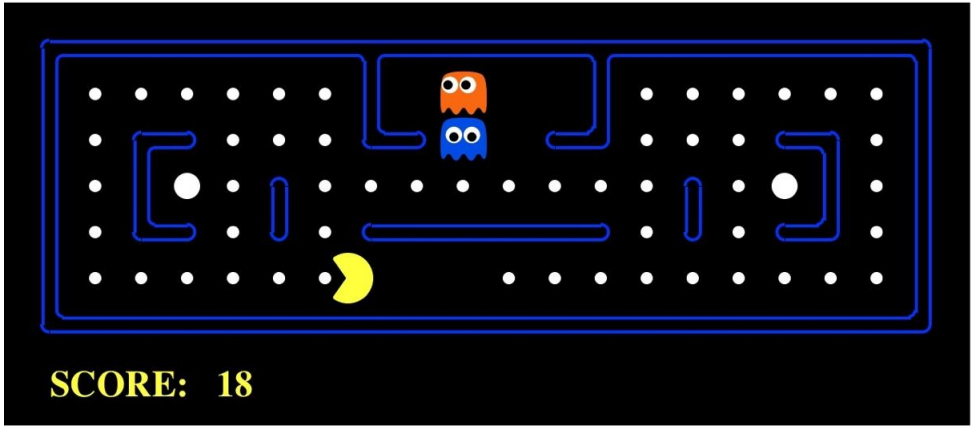
[These slides were originally created by Dan Klein and Pieter Abbeel for *CS188 Intro to AI* at UC Berkeley (http://ai.berkeley.edu) – Edited by Aleksis Pirinen for *FMAN45 Machine Learning* at Lund University]

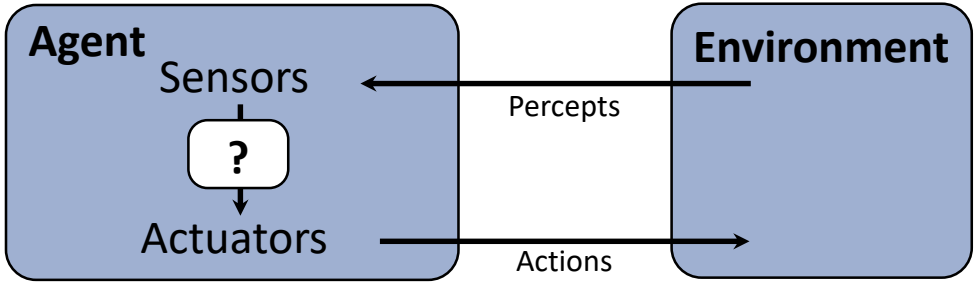
Rational Agents

- An **agent** is an entity that *perceives* and *acts*
- A **rational agent** selects actions that maximize its (expected) **utility**
- **Utility** defined over *sequence* of actions
- Characteristics of the **percepts, environment,** and **action space** dictate techniques for selecting rational actions



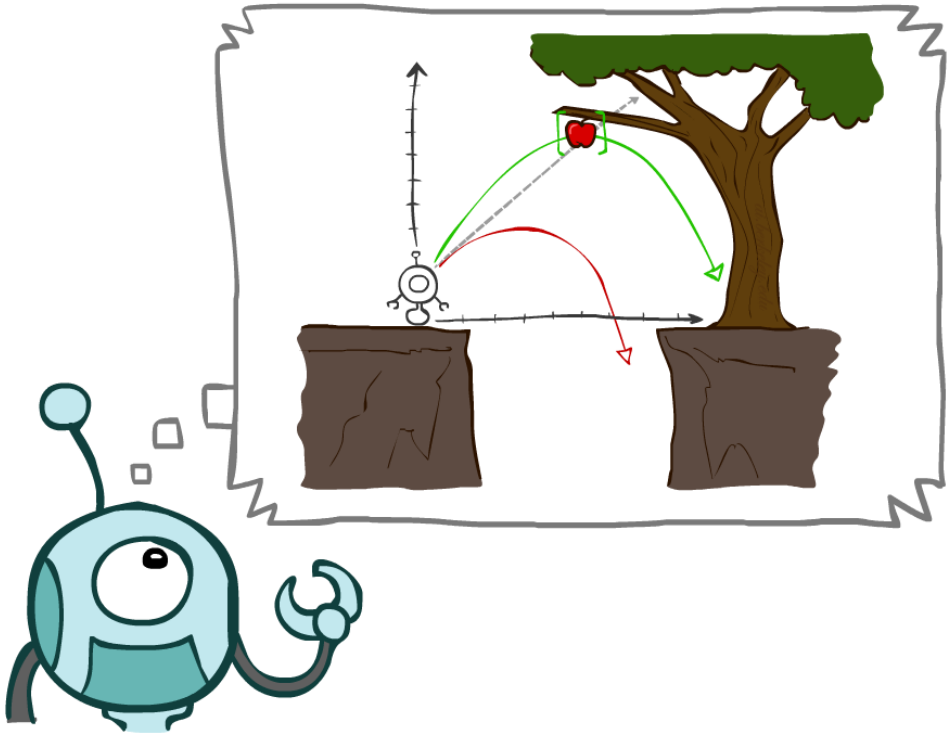
Pac-Man as an Agent





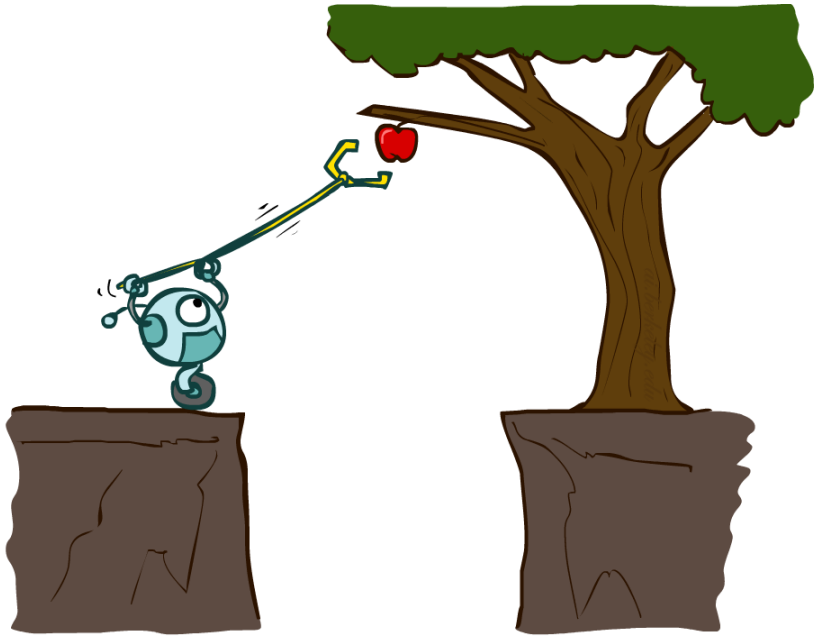
Pac-Man is a registered trademark of Namco-Bandai Games, used here for educational purposes

Agents that Plan



Planning Agents

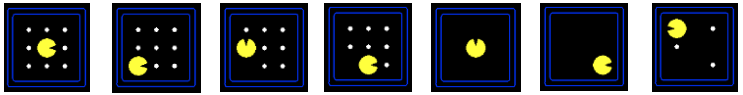
- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a **model** of how the world evolves in response to actions
  - Must formulate a **goal**
  - Compute consequences of actions in a **simulation**, rather than actually performing actions “in the wild”
  - Consider how the world WOULD BE



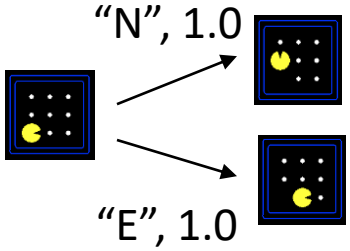
Search Problems

Search Problems

- A search problem consists of:
  - **State space**



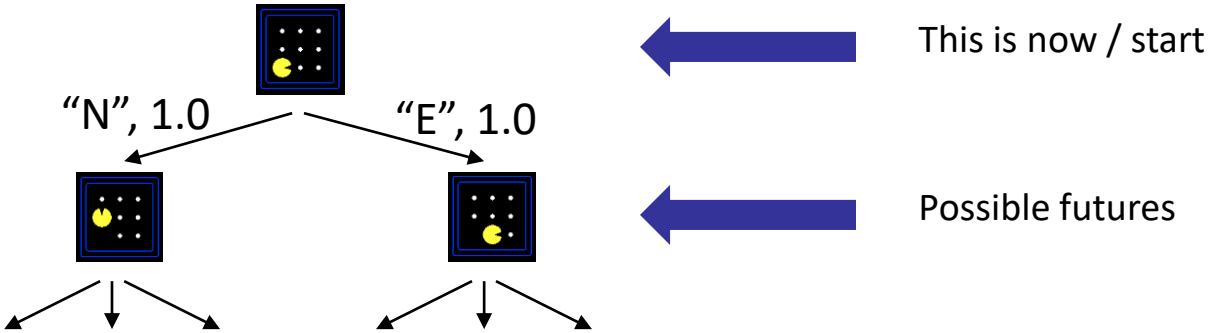
  - **Successor function**
    (with actions, costs)



  - **Start state** and a **goal test**

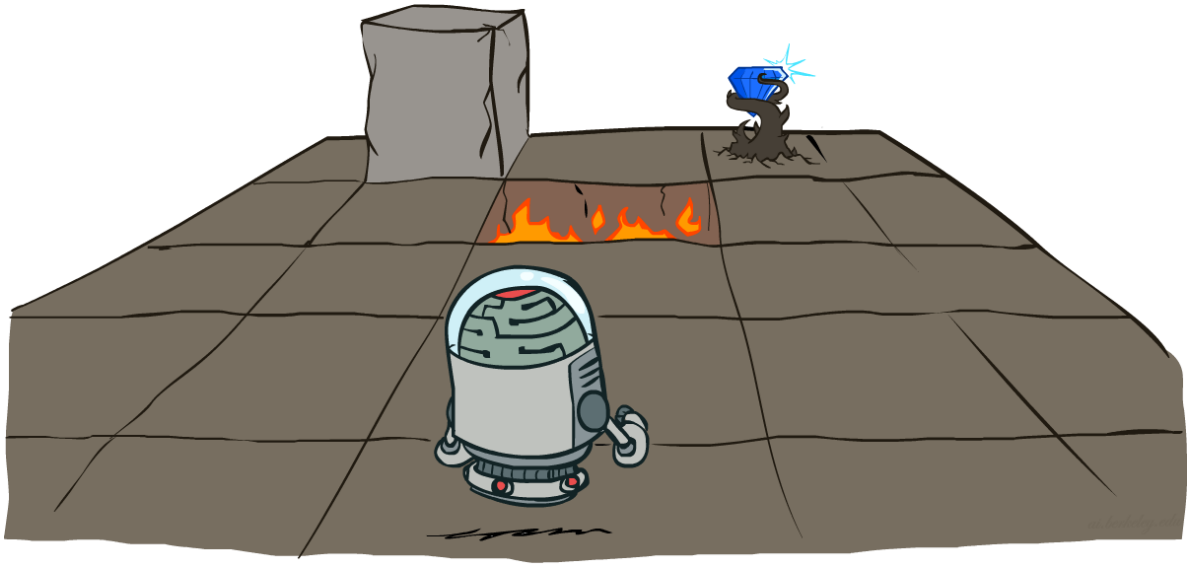
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

Search Trees

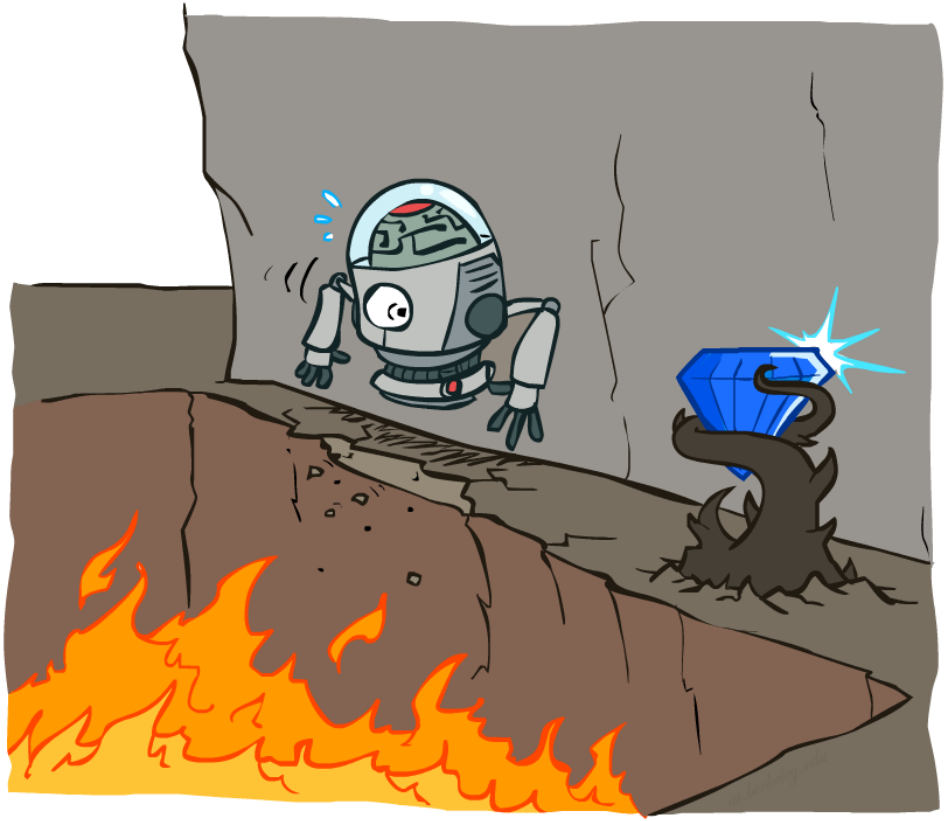


- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

Markov Decision Processes

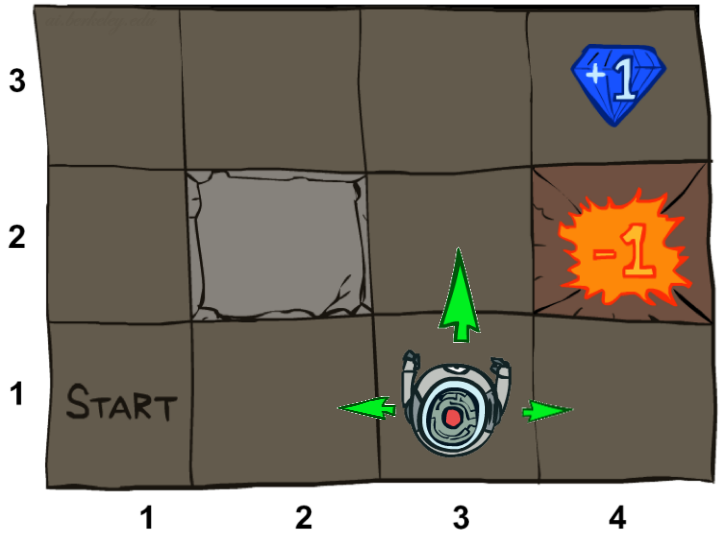


Non-Deterministic Search



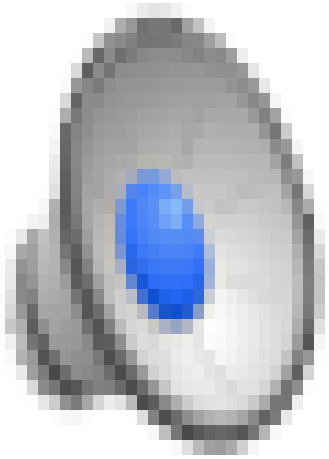
Example: Grid World

- 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
  - "Exit" action at terminal states (jewel / fire pit)



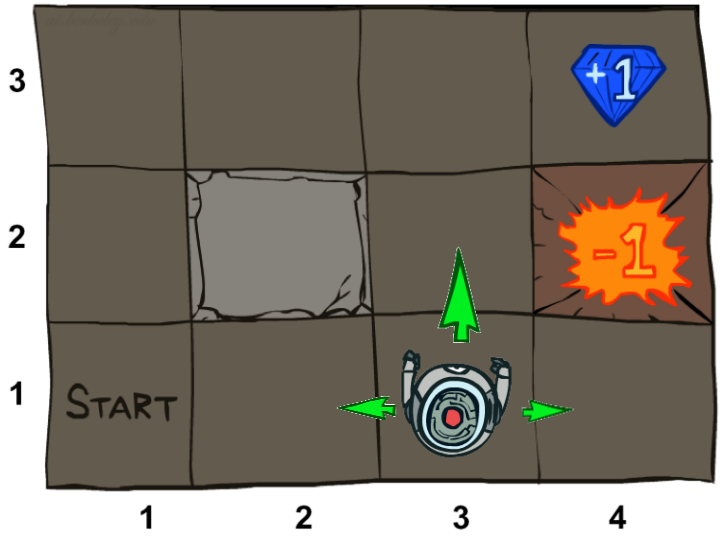
- The agent receives **rewards each time step**
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end: good (blue jewel, +1) or bad (fire pit, -1)
- Goal: **maximize expected sum of rewards** (i.e., maximize expected utility)

Video Demo of Grid World



Markov Decision Processes (MDPs)

- 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*
    - Similar to successor function, except may be stochastic
  - 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**

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

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

$$=$$

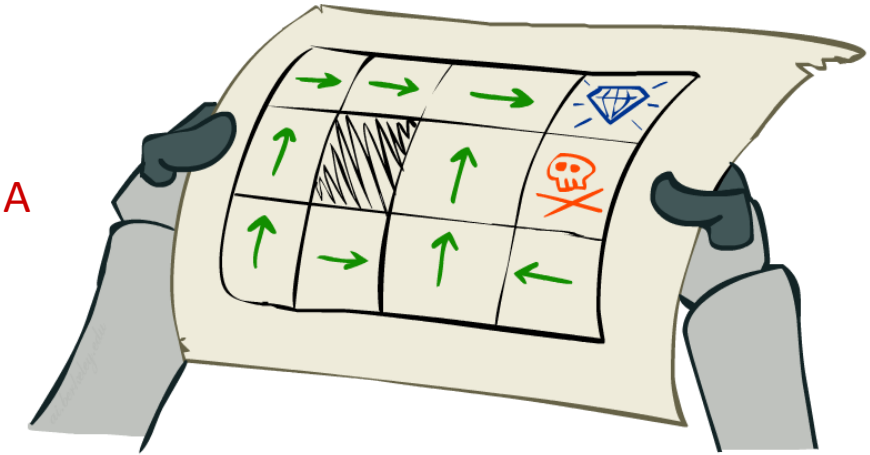
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

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

Andrey Markov
(1856-1922)

Policies

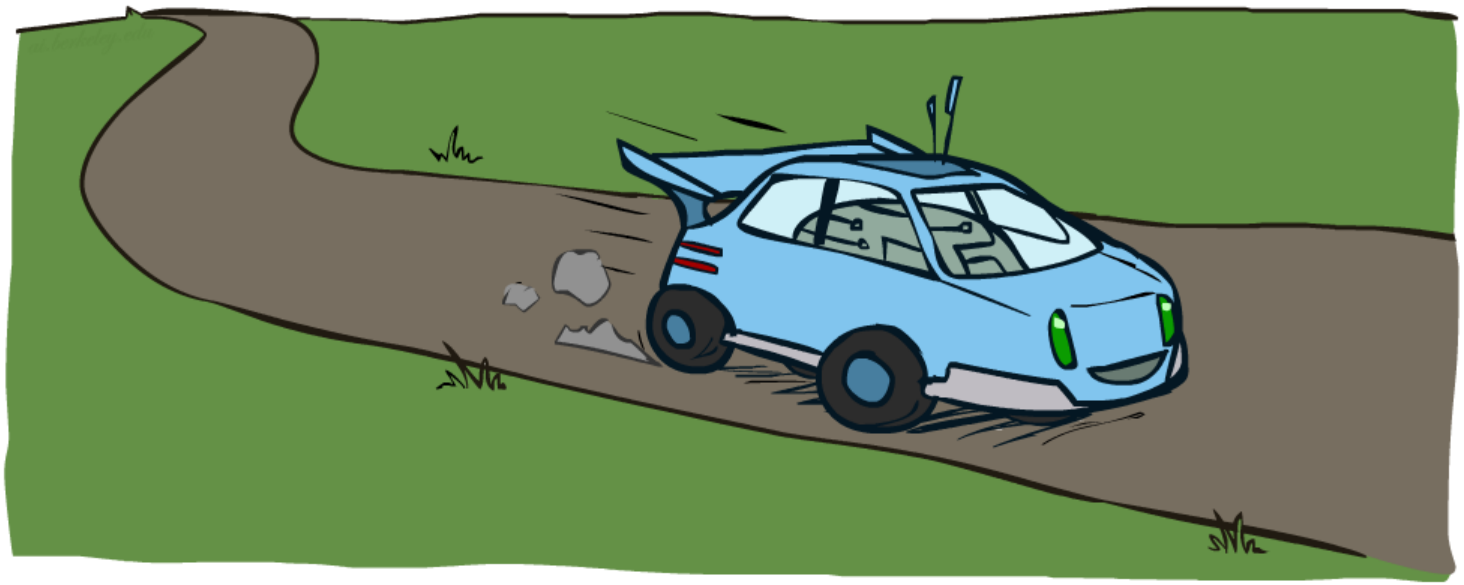
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected sum of rewards (utility) if followed



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

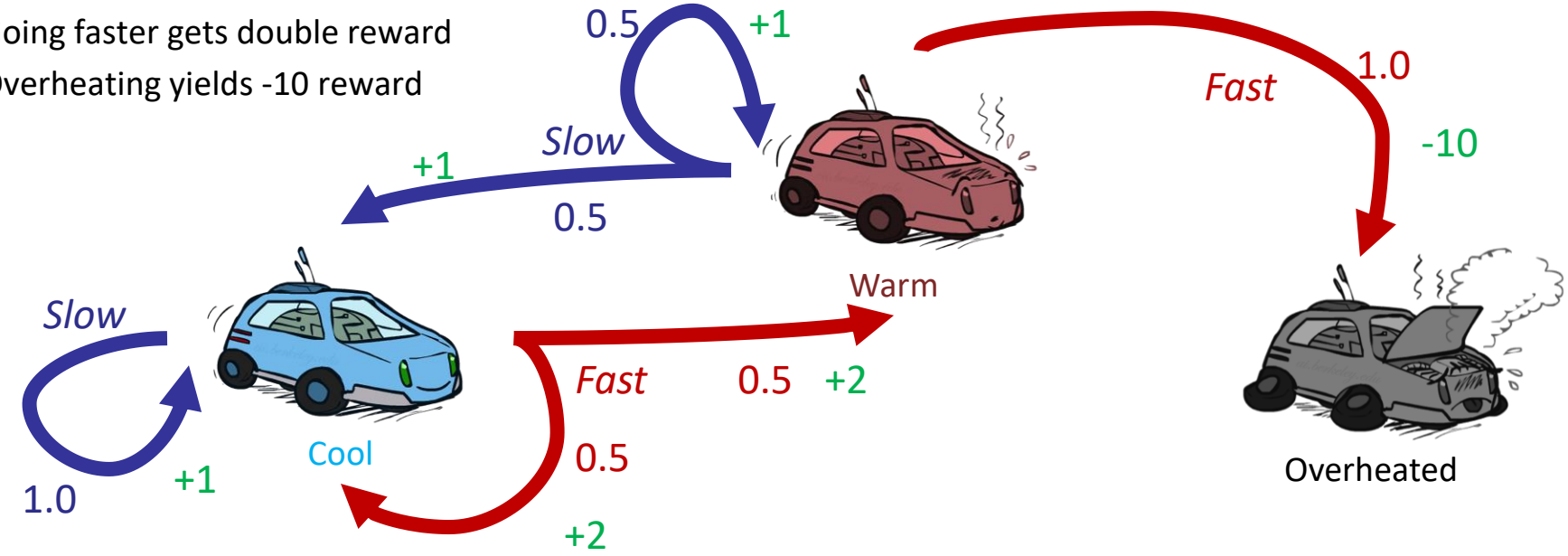
Recall: Stochastic transitions!

Example: Racing

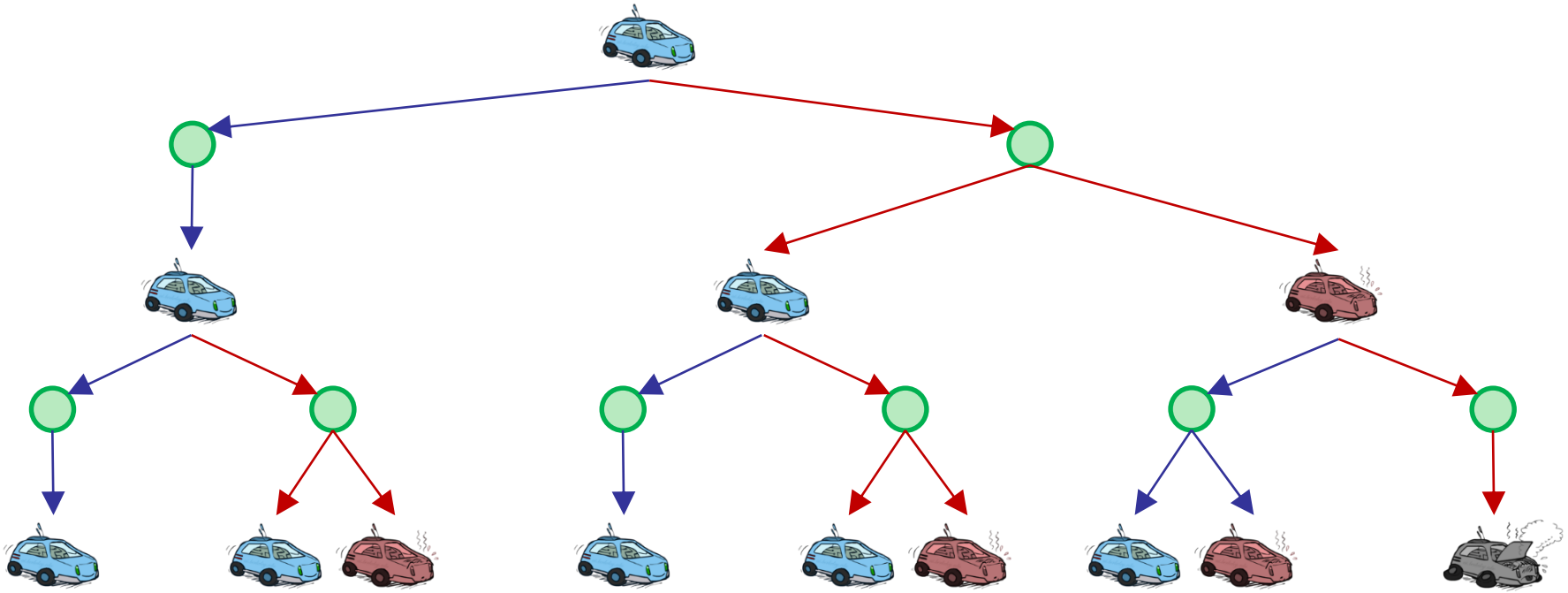


Example: Racing

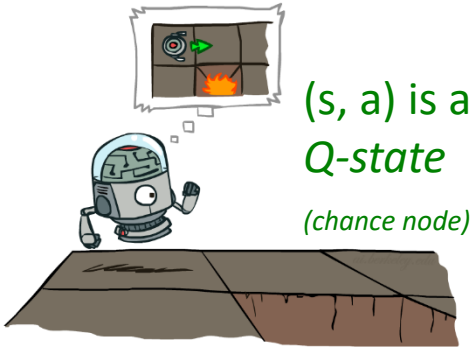
- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated (terminal)
- Two actions: *Slow*, *Fast*
- Going faster gets double reward
- Overheating yields -10 reward



Racing Search Tree

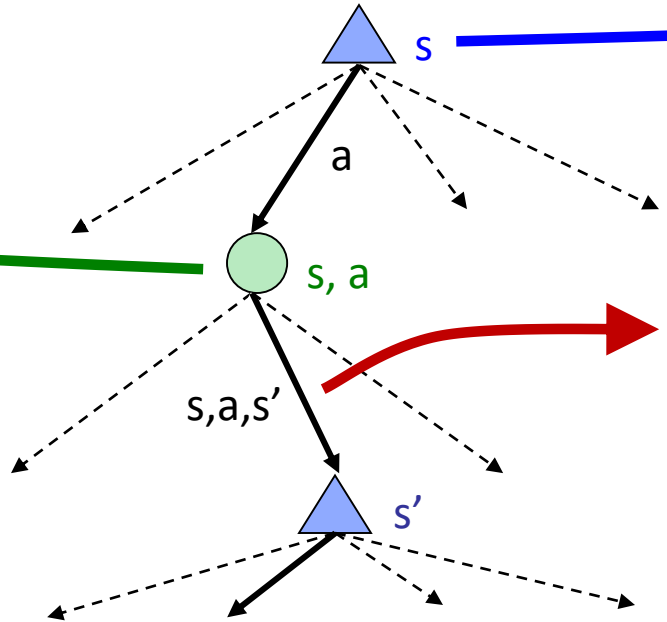


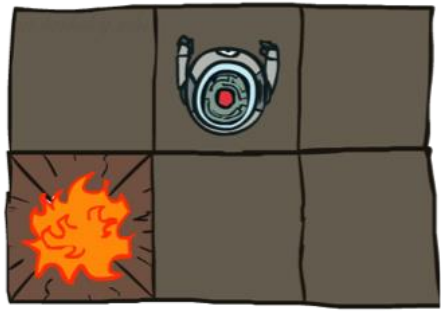
MDP Search Trees



(s, a) is a *Q-state*

(chance node)





(s,a,s') is called a *transition*

T(s,a,s') = P(s'|s,a)

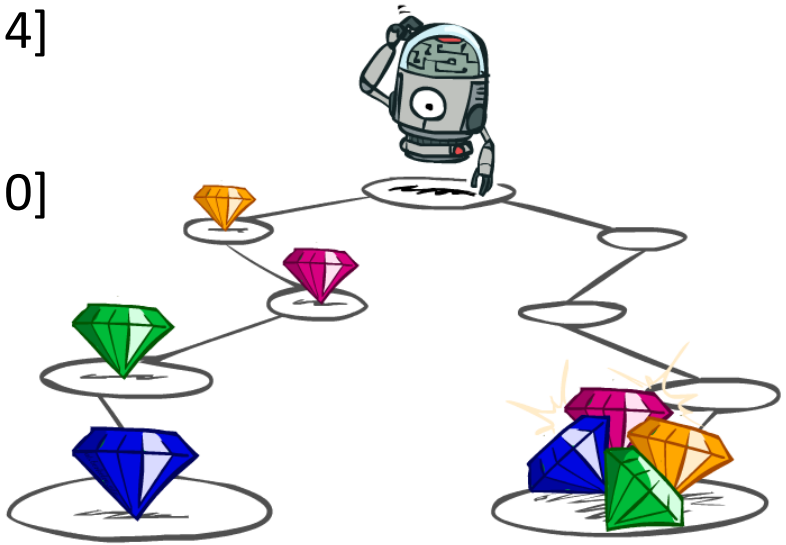
R(s,a,s')

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 expected sum of rewards**
- It's also reasonable to **prefer rewards *now* rather than later**
- **One solution:** *values of rewards* ***decay exponentially***

1

Worth Now

γ

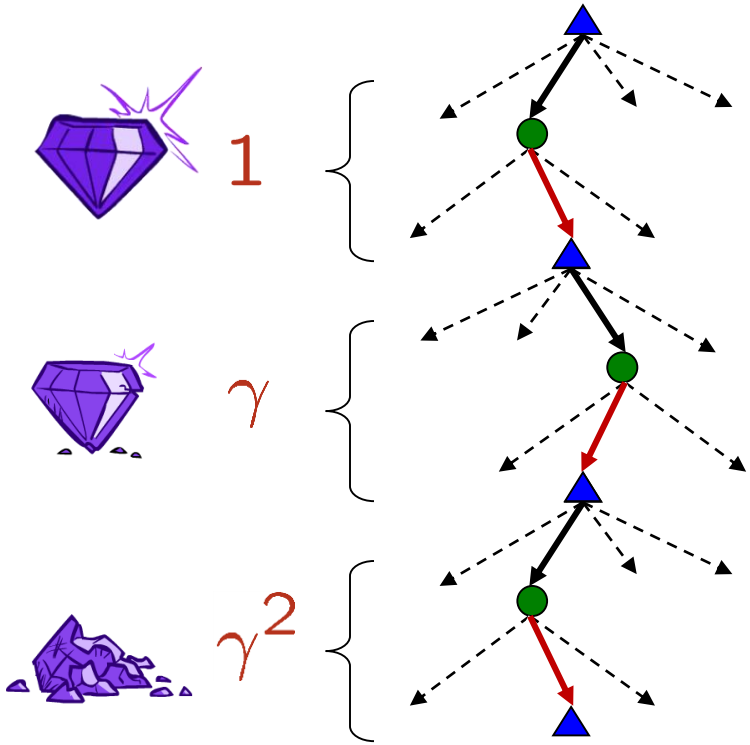
Worth Next Step

γ^2

Worth In Two Steps

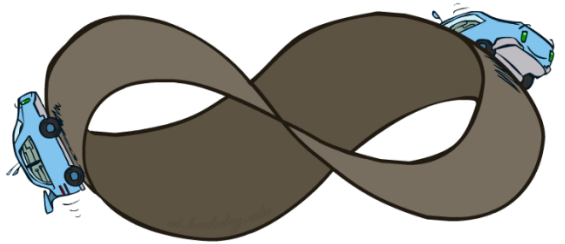
Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once with what we see at the next level
- Why discount?
  - **Sooner rewards** probably do have **higher utility** than later rewards
  - Also **helps our algorithms converge**



Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:
  - **Finite horizon**:
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)



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

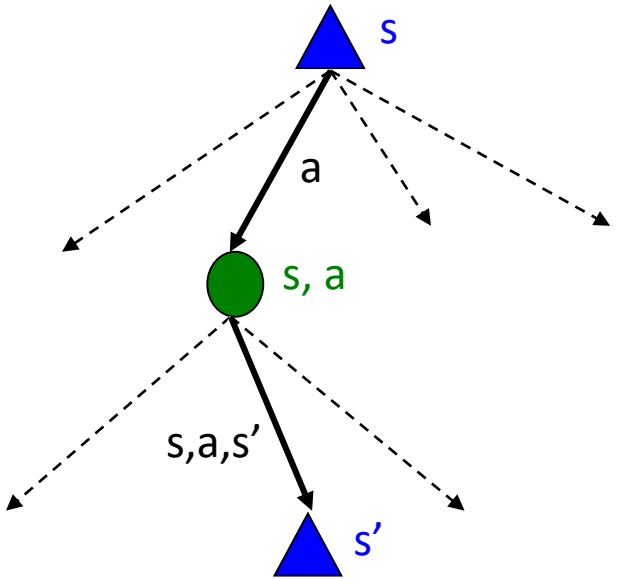
$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

    - Smaller γ means smaller "horizon" – shorter term focus
  - **Absorbing state**: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing, but we need to tweak it so that (cool,slow) may lead to overheating too)

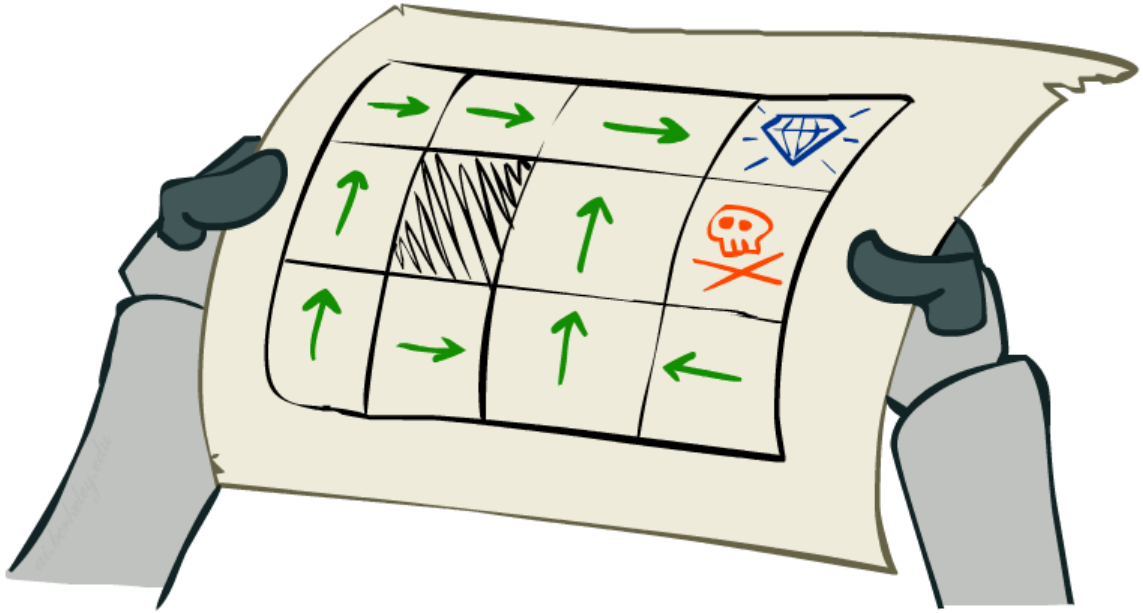
Recap: Defining MDPs

- 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 γ

- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = Sum of (discounted) rewards



Solving MDPs



Optimal Quantities

- The optimal value of a state s:

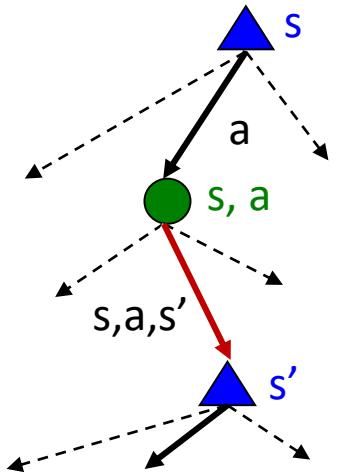
  $V^*(s)$ = expected utility starting in s and acting optimally

- The optimal value 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

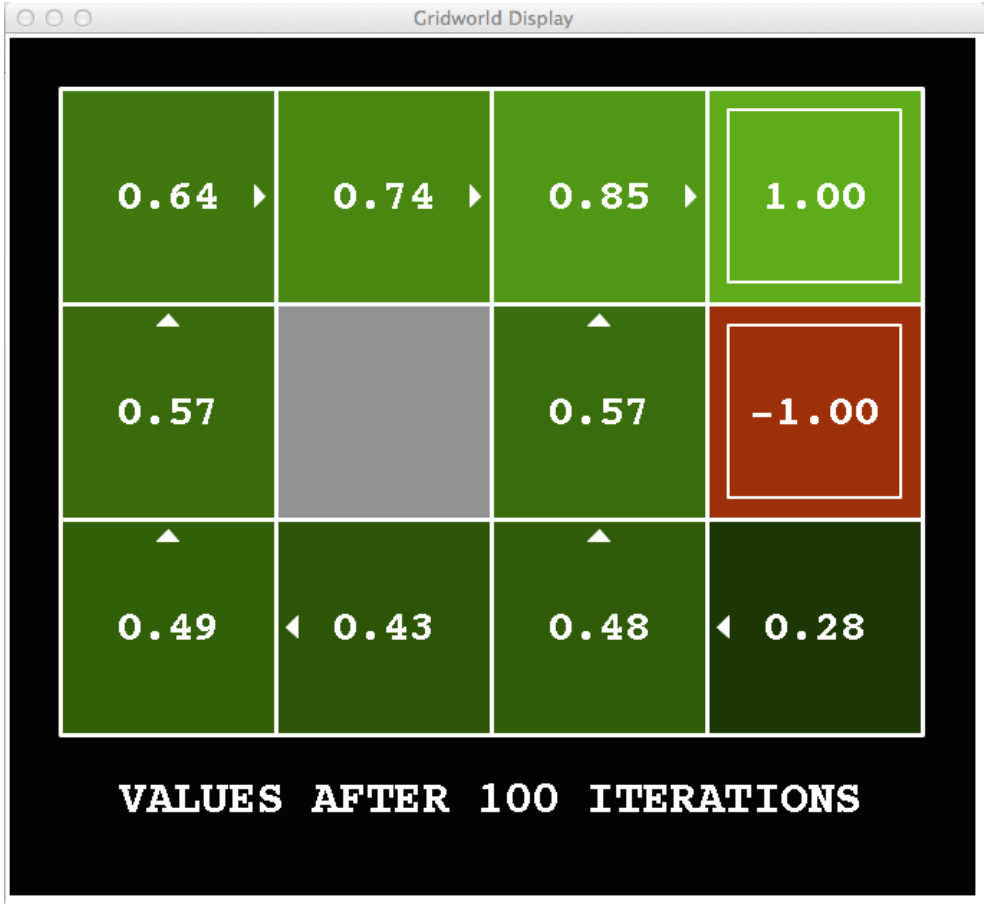


s is a *state (max node)*

(s, a) is a *q-state (chance node)*

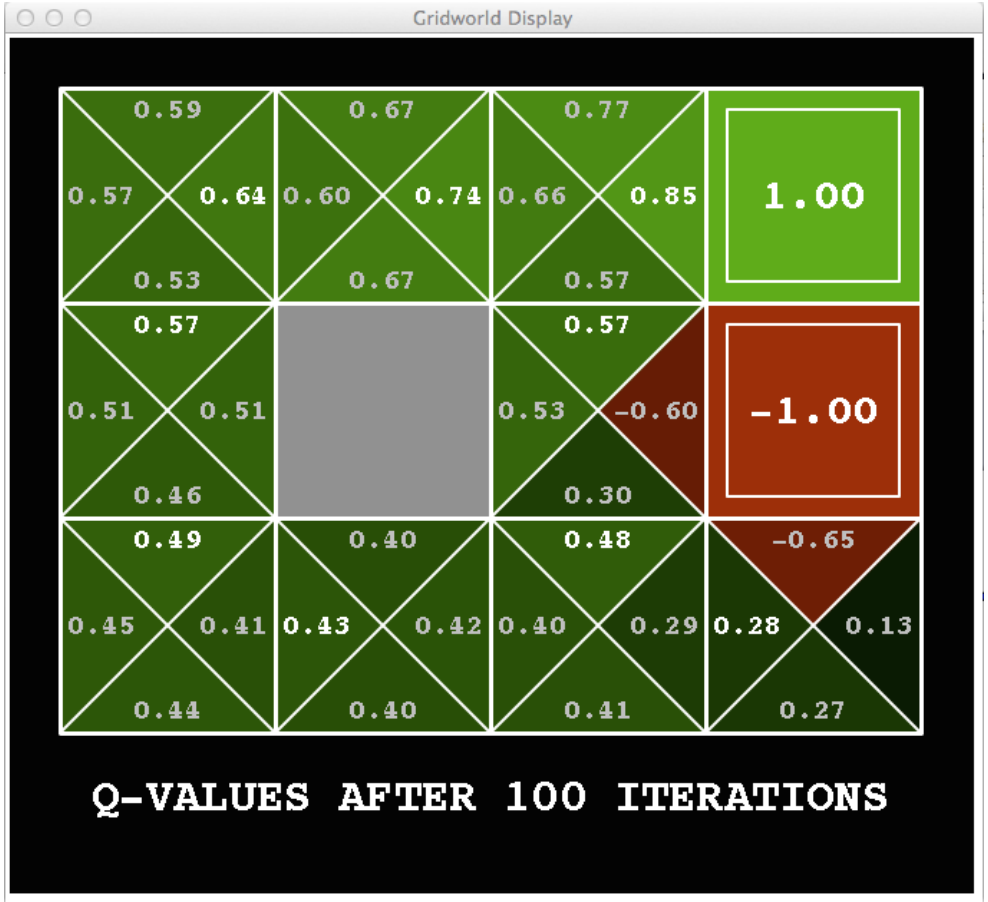
(s,a,s') is a *transition*

Snapshot of Demo – Grid World V-Values



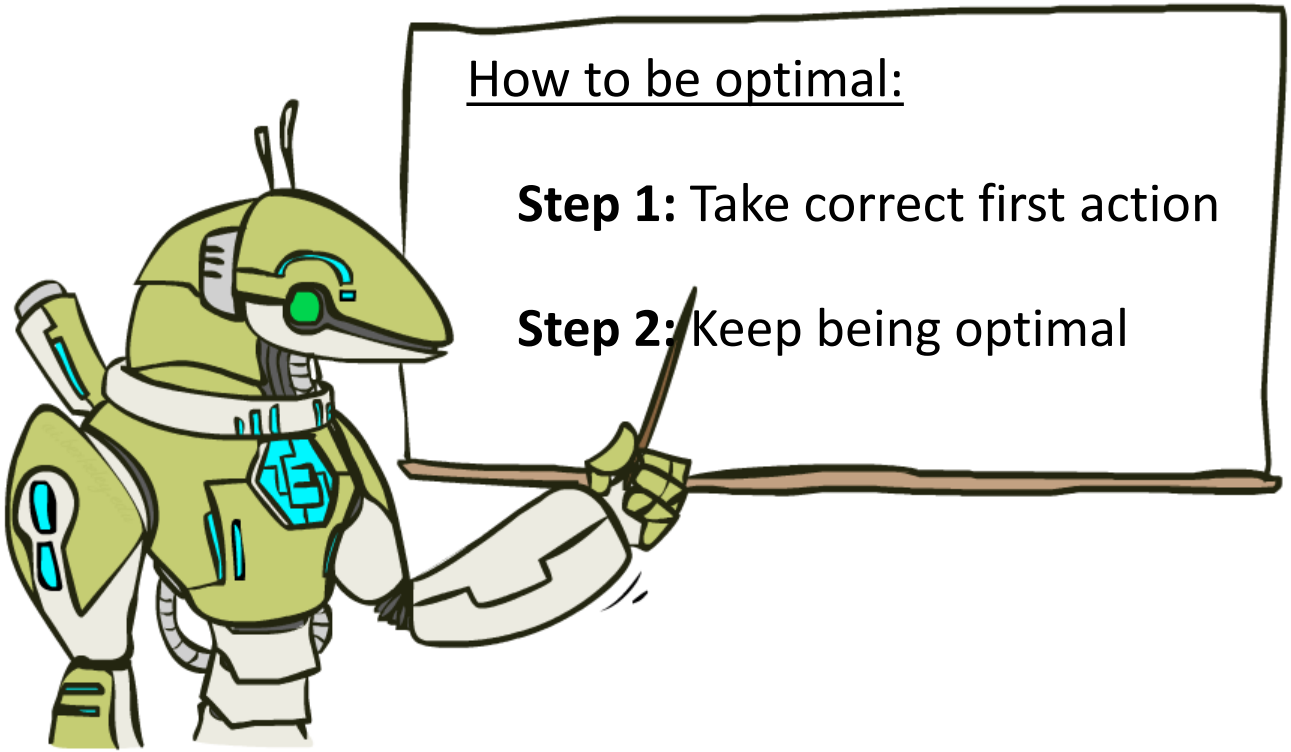
Noise = 0.2
Discount = 0.9
Living reward = 0

Snapshot of Demo – Grid World Q-Values



Noise = 0.2
Discount = 0.9
Living reward = 0

The Bellman Optimality Equations



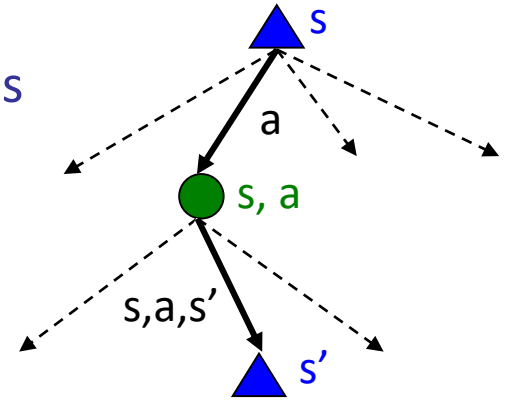
The Bellman Optimality Equations

- Definition of "optimal utility" is a simple one-step lookahead relationship amongst optimal utility values

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

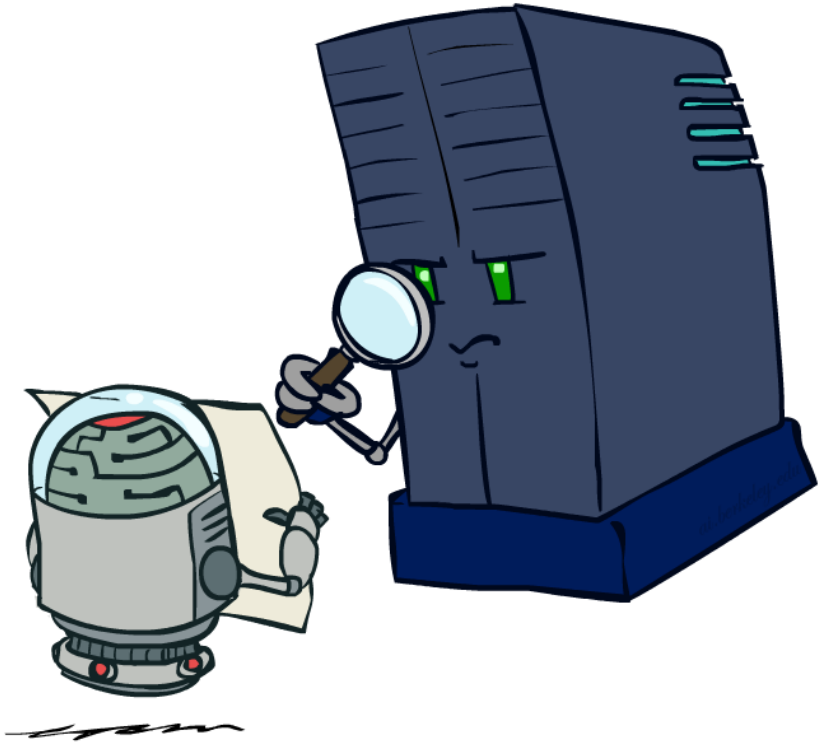
$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

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



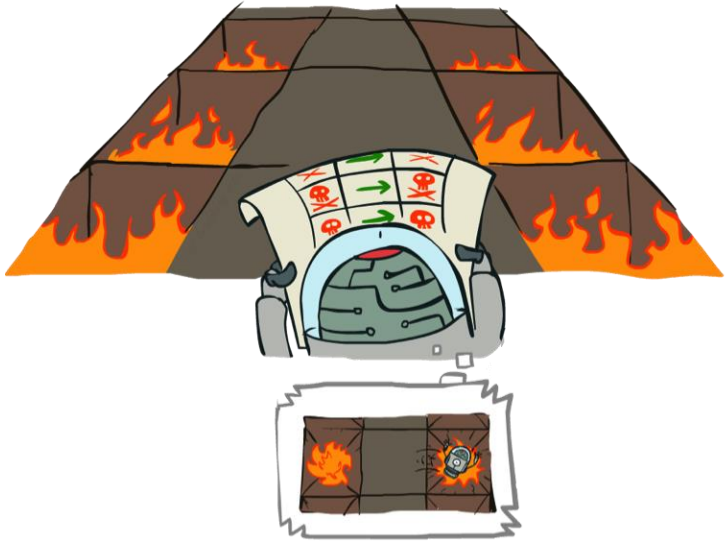
- These are the Bellman optimality equations, and they **characterize optimal values**

Policy Evaluation

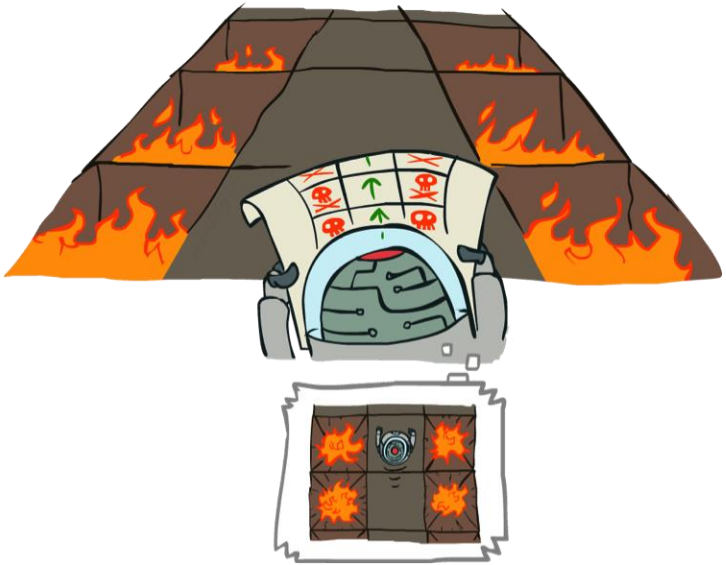


Example: Policy Evaluation

Always Go Right



Always Go Forward



Example: Policy Evaluation

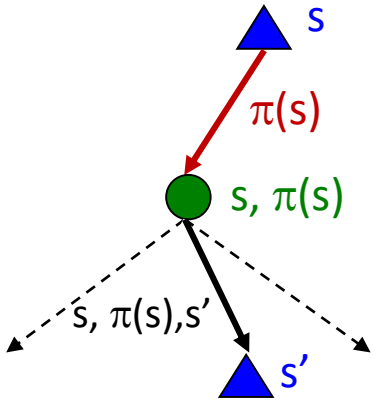
Always Go Right

Always Go Forward

Values for a Fixed Policy

- Define the value of a state s, under a fixed policy π:

  $V^{\pi}(s)$ = expected total discounted rewards starting in s and following π



- Recursive relation (one-step lookahead / Bellman equation):

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

- Note: No outer maxing (as for optimal policies)!

Policy Evaluation

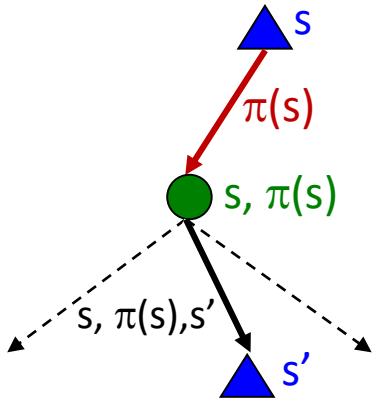
- How do we calculate the V's for a fixed policy π?
- Recall: Recursive Bellman equation for $V^{\pi}(s)$

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

- Turn recursive **Bellman equations into updates (fixed-point solution)**

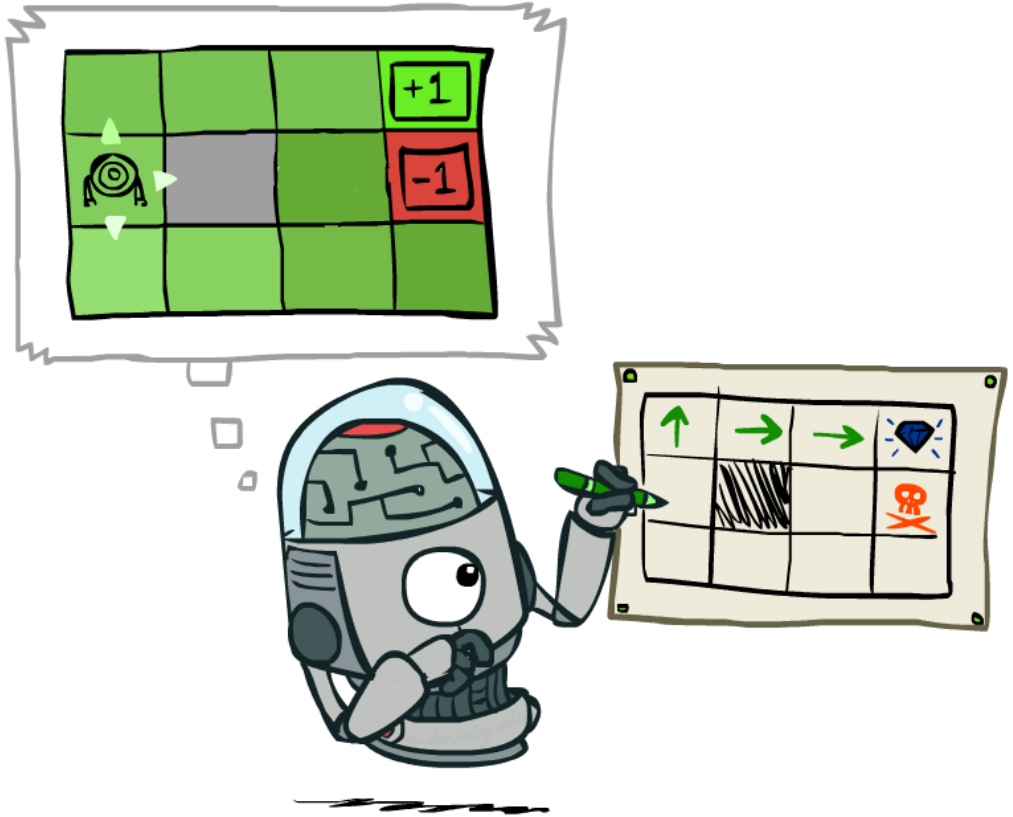
$$V_0^{\pi}(s) = 0$$

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



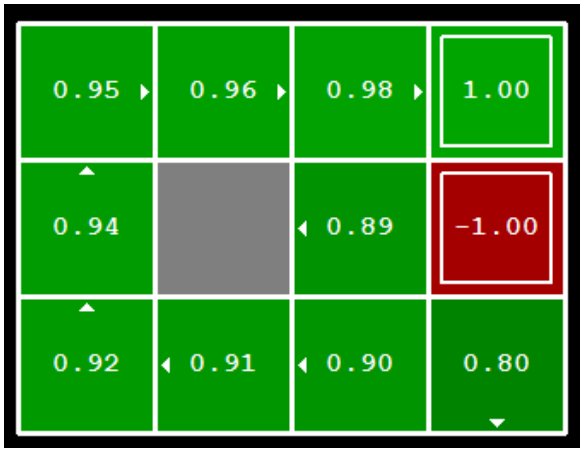
- Theorem: **will converge to unique values corresponding to π** if finite-depth tree or if $0 < \gamma < 1$
  - Basic idea: Bound $|V_k - V_{k+1}|$ by some finite C times γ^k

Policy Extraction



Computing Actions from Values

- Let's imagine we have the optimal values V*(s)
- How should we act?
  - It's not obvious!
- We need to max for a one-step lookahead:

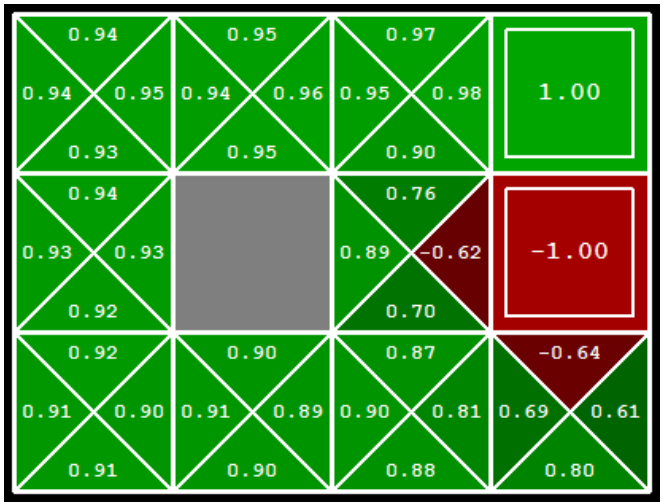


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- 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:

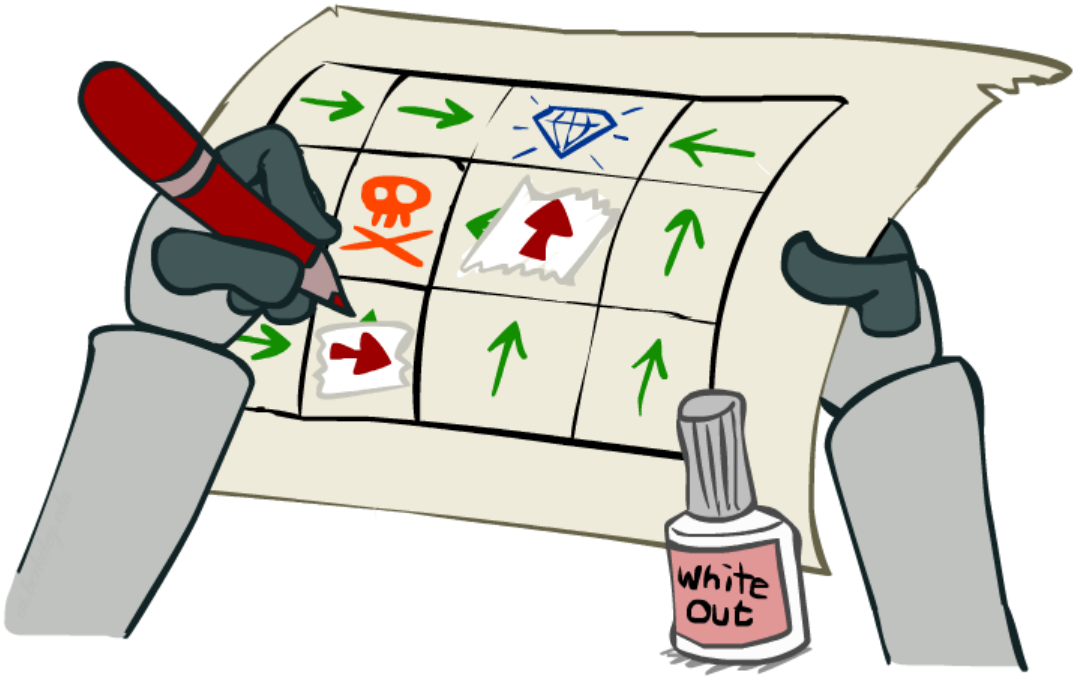


- How should we act?
  - Completely **trivial to decide**!

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

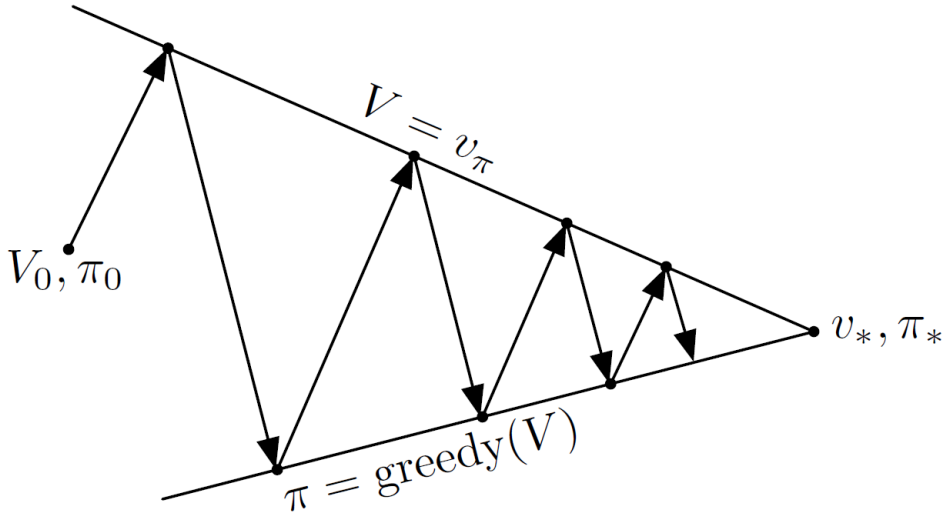
- Important lesson: **actions are easier to select from Q-values than values**!
  - Will come back to this when talking about reinforcement learning

Policy Iteration



Policy Iteration

- Alternating approach for optimal values and optimal policy:
  - Step 1: Policy evaluation: compute values for *current fixed policy* until convergence
  - Step 2: Policy improvement: update policy using one-step lookahead with resulting converged (*not* necessarily optimal) values from step 1
  - Repeat steps until policy converges



[Image source: ***Reinforcement Learning: An Introduction (2nd edition)***, Sutton & Barto 2016]

Policy Iteration – Alternating Steps

- **Policy evaluation:** For *fixed current policy* π, find values with policy evaluation
  - Iterate until values converge:

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

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

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

Policy Iteration – Full Algorithm

Policy iteration (using iterative policy evaluation)

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

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

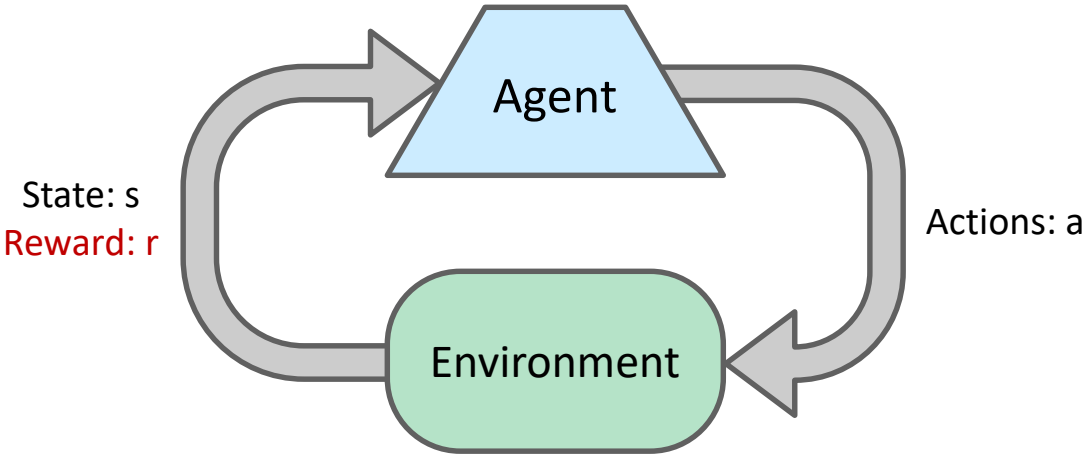
[*Reinforcement Learning: An Introduction (2nd edition)*, Sutton & Barto 2016]

Summary: MDP Algorithms

- So you want to….
  - Compute optimal values: use 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 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!

Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) A
  - A model T(s,a,s')
  - A reward function R(s,a,s')
- Still looking for a policy $\pi(s)$

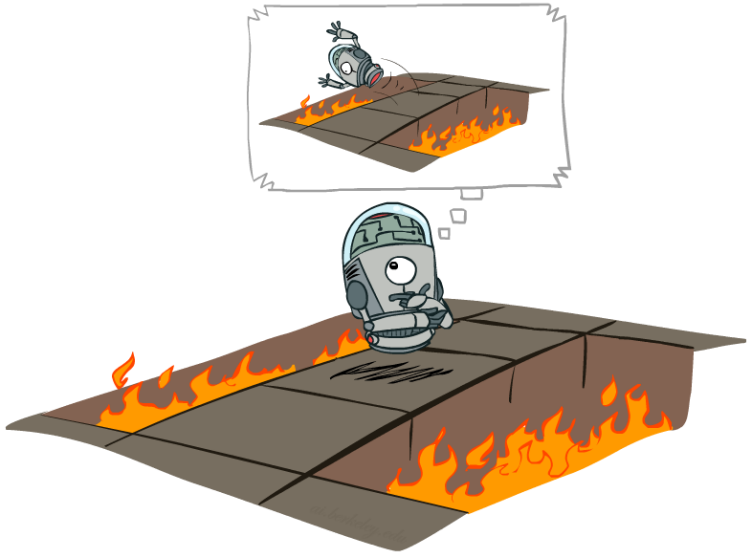
Cool

Warm

Overheated

- New twist: don't know T or R
  - I.e. we don't know which states are good or what the actions do
  - Must actually **try actions and states to learn which are good/bad**

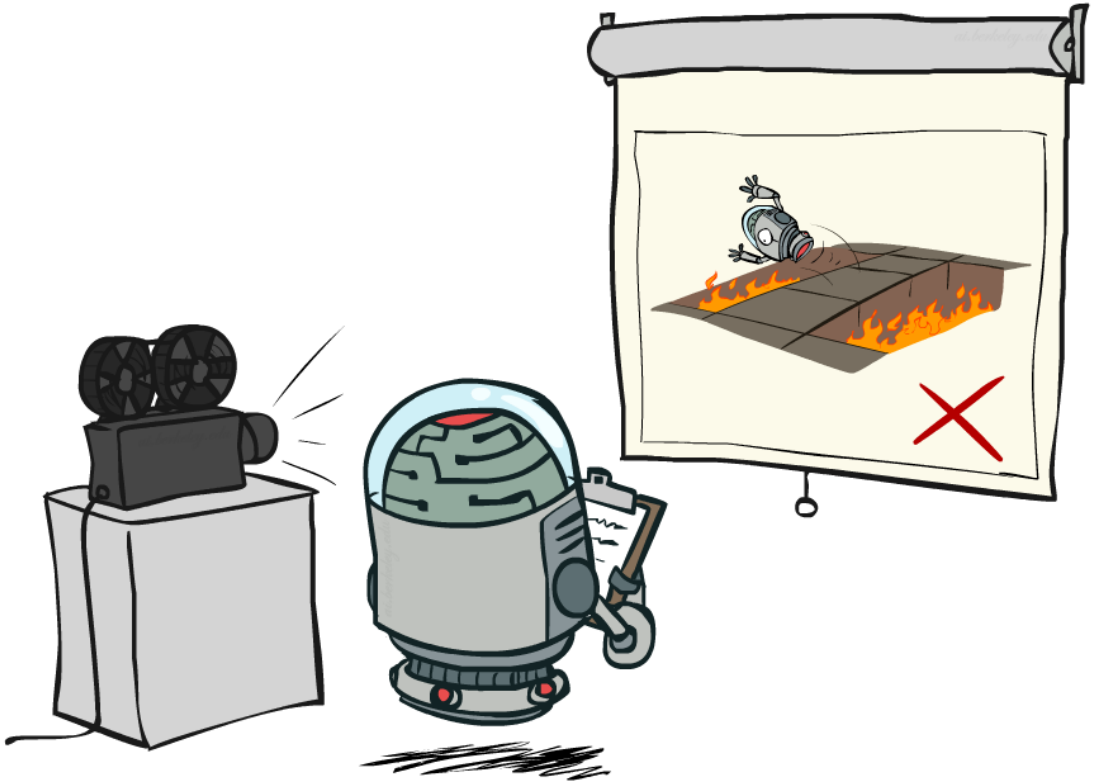
Offline (MDPs) vs. Online (RL)



Offline Solution

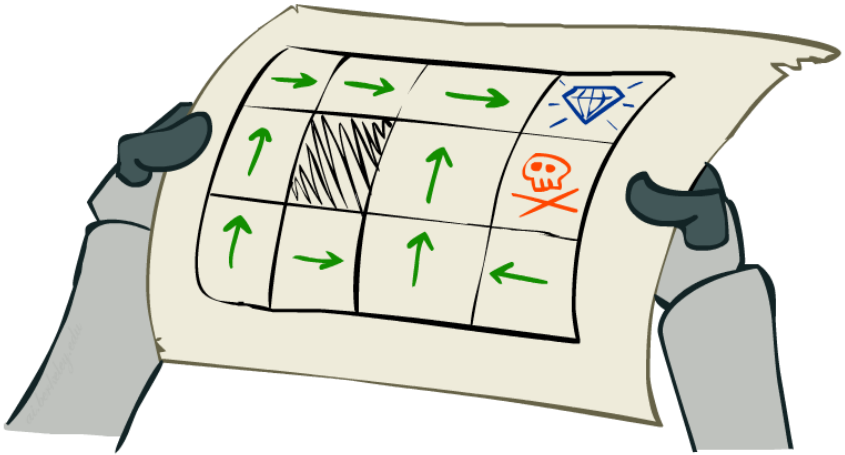
Online Learning

Passive Reinforcement Learning



Passive Reinforcement Learning

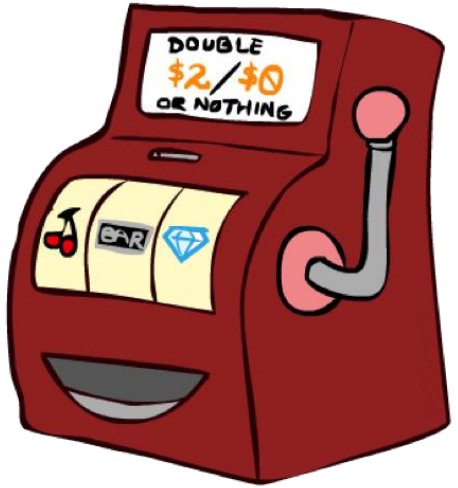
- Simplified task: policy evaluation
  - Input: a fixed policy $\pi(s)$
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - Goal: learn the state values



- In this case:
  - Learner is "along for the ride"
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - This is NOT offline planning! You actually take actions in the world according to the given policy

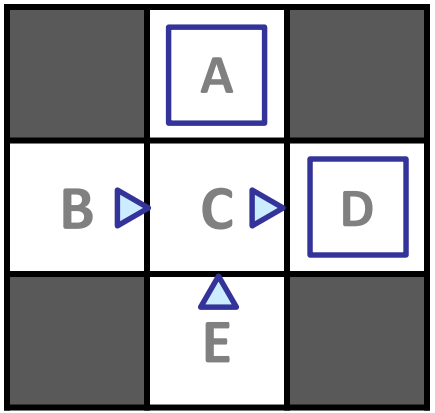
Direct Evaluation

- Goal: Compute values for each state under π
- Idea: Average together observed sample values
  - Act according to π
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - Average those samples
  - This is just simple Monte-Carlo simulation
- This is called direct evaluation



Example: Direct Evaluation

Input Policy π



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

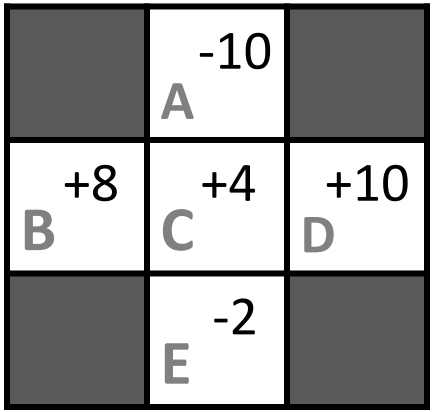
Episode 3

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

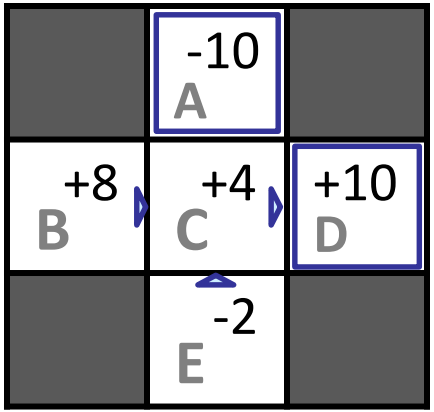
Output Values



Problems with Direct Evaluation

- What's good about direct evaluation?
  - It's easy to understand
  - It doesn't require any knowledge of T, R
  - It eventually computes the correct average values, using just sample transitions
- What bad about it?
  - It wastes information about state connections
  - Each state must be learned separately
  - So, it takes a long time to learn

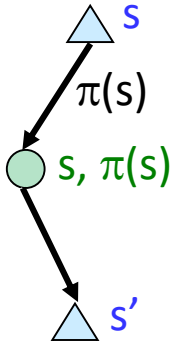
Output Values



If B and E both go to C under this policy, how can their values be different?

Temporal Difference Learning (TD Learning)

- Big idea: **learn from every experience**!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often



- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

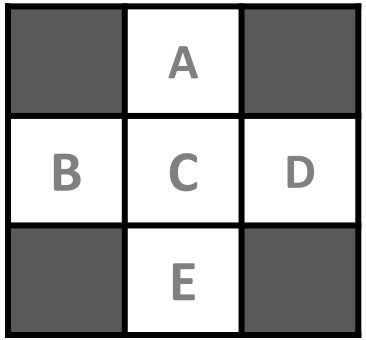
Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$

Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Example: Temporal Difference Learning

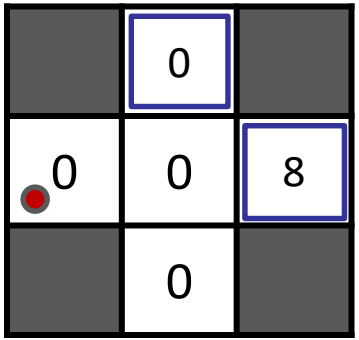
States



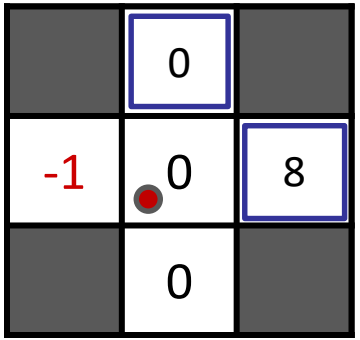
Assume: $\gamma = 1$, $\alpha = 0.5$

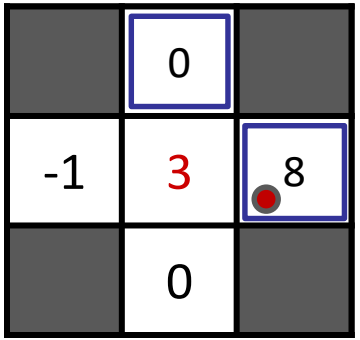
Observed Transitions

B, east, C, -2



C, east, D, -2

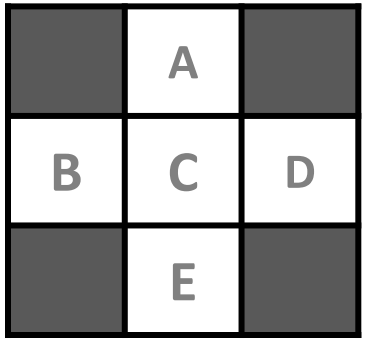




$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

Example: Temporal Difference Learning

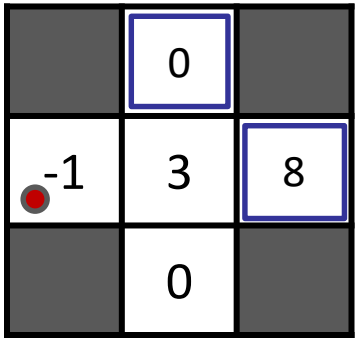
States



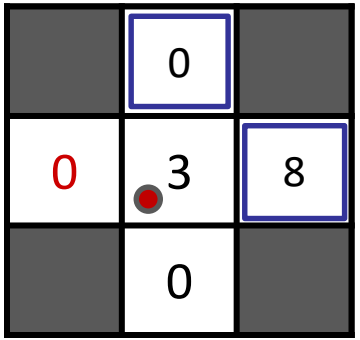
Assume: $\gamma = 1$, $\alpha = 1/2$

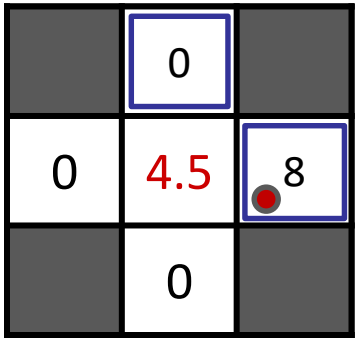
Observed Transitions

B, east, C, -2



C, east, D, -2

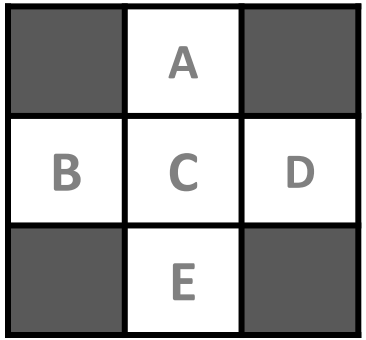




$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

Example: Temporal Difference Learning

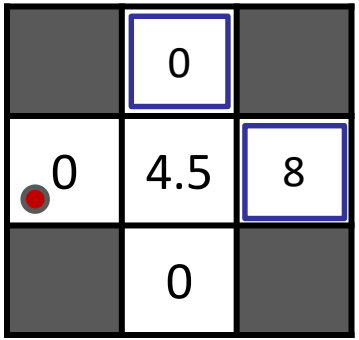
States



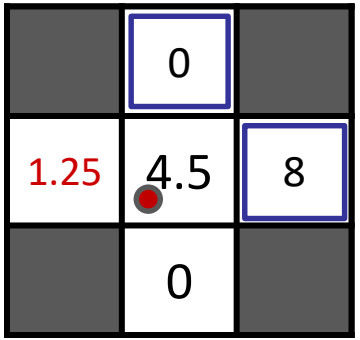
Assume: $\gamma = 1$, $\alpha = 1/2$

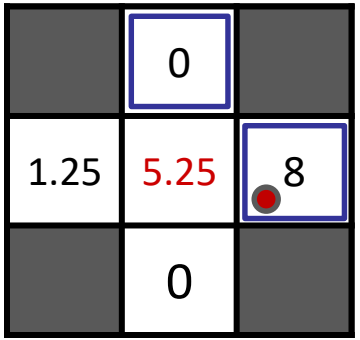
Observed Transitions

B, east, C, -2



C, east, D, -2

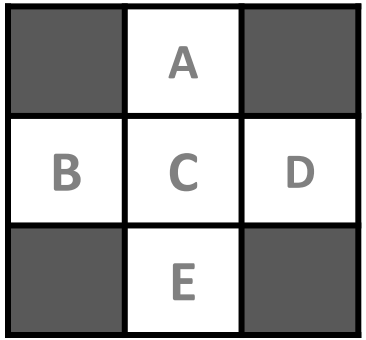




$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

Example: Temporal Difference Learning

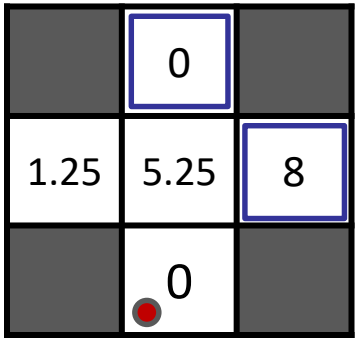
States



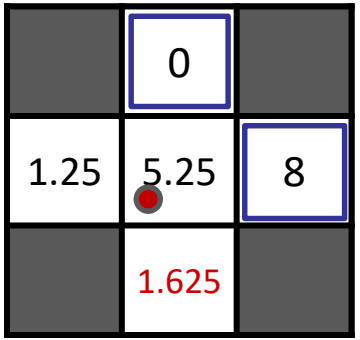
Assume: $\gamma = 1$, $\alpha = 1/2$

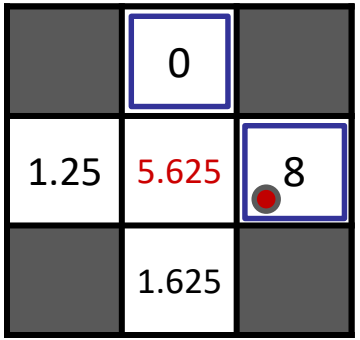
Observed Transitions

E, North, C, -2



C, east, D, -2





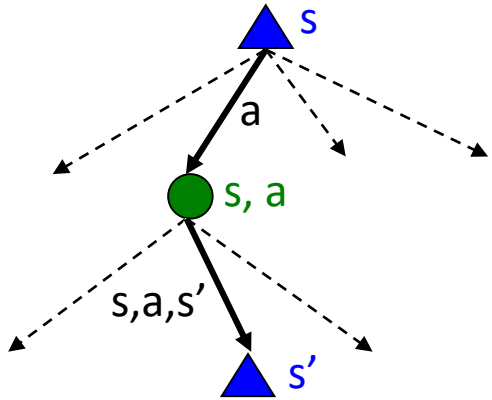
$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

Problems with TD Value Learning

- TD value learning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- However, if we want to turn values into a (new) policy, we're sunk:

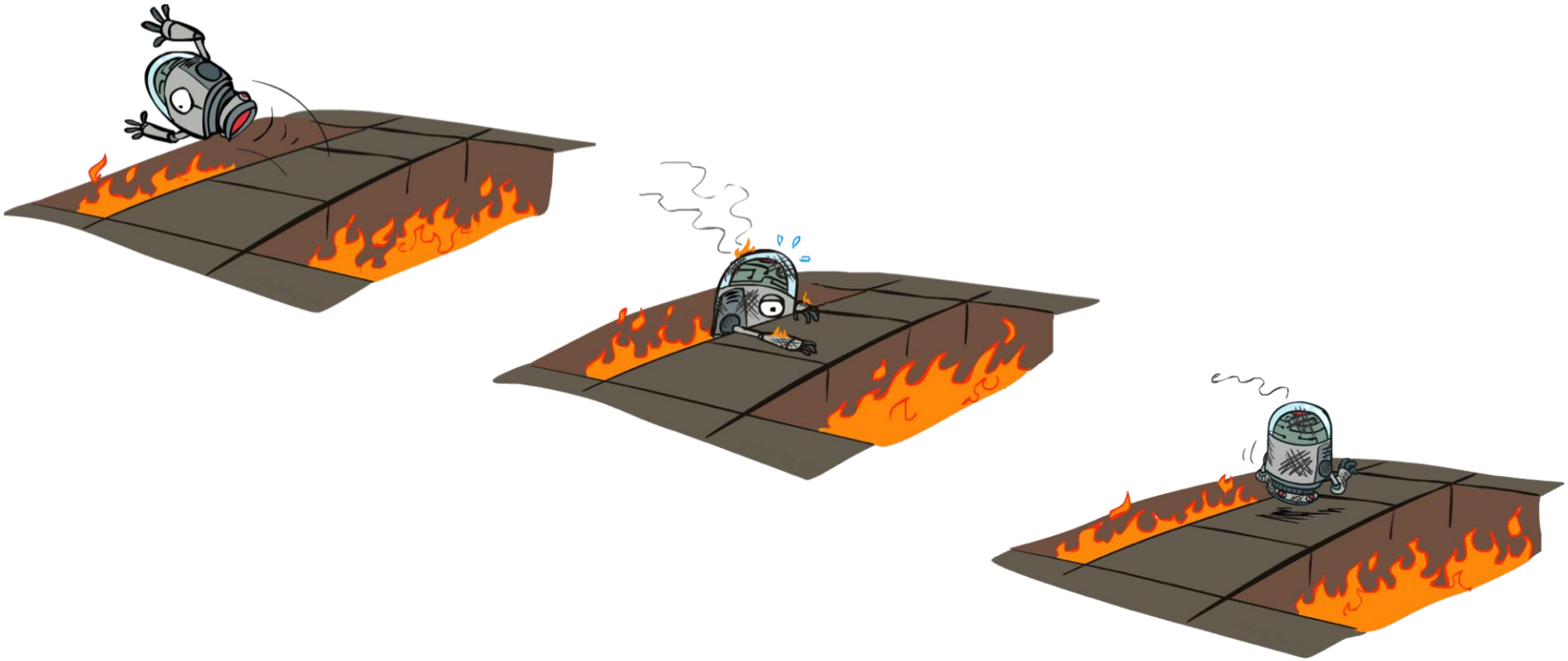
$$\pi(s) = \arg\max_a Q(s,a)$$

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



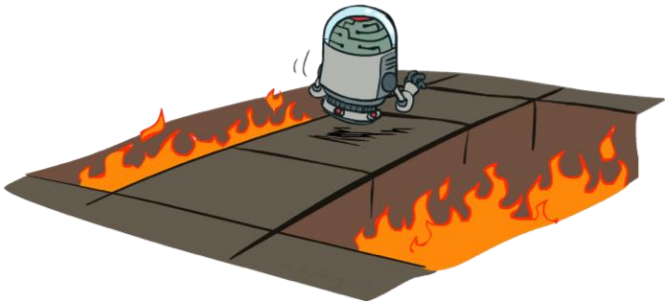
- Idea: **learn Q-values, <u>not</u> values!**
- Makes action selection model-free too! ☺

Active Reinforcement Learning



Active Reinforcement Learning

- Full reinforcement learning: optimal policies (like policy iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values



- In this case:
  - **Learner makes choices!** (C.f. passive, where we only evaluate)
  - Fundamental tradeoff: **exploration vs. exploitation**
  - This is NOT offline planning! You actually take actions in the world and find out what happens…

Detour (known MDPs): Q-Value Iteration

- We want to look at Q instead of V, since **Q allows for simple action selection**:

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

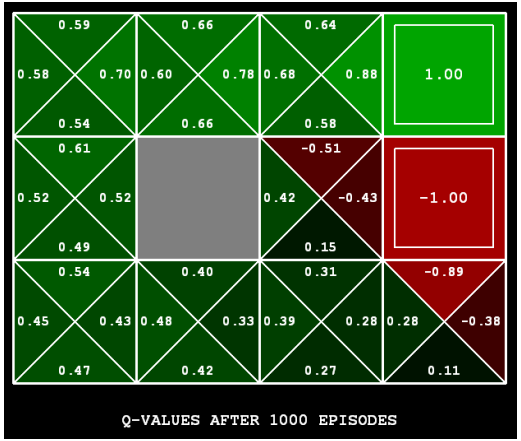
- Bellman optimality equation for Q^* turned into updates

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

- Start with $Q_0(s,a) = 0$

- Given Q_k, calculate the depth k+1 Q-values for all Q-states:

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



Q-Learning (back to unknown MDPs)

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

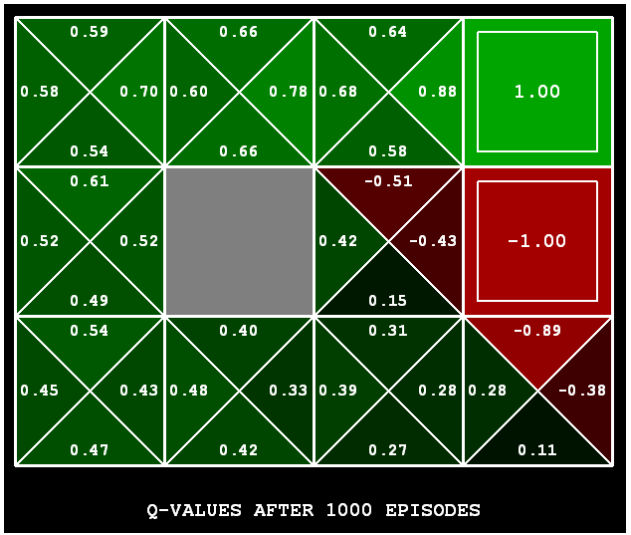
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

- 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:

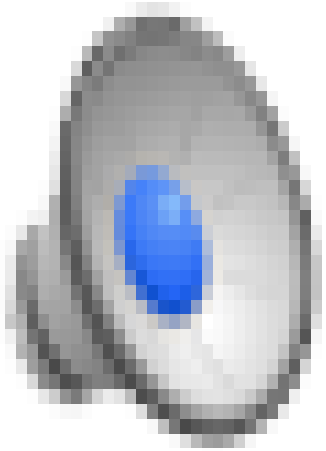
$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

  - Incorporate the new estimate into a running average:

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

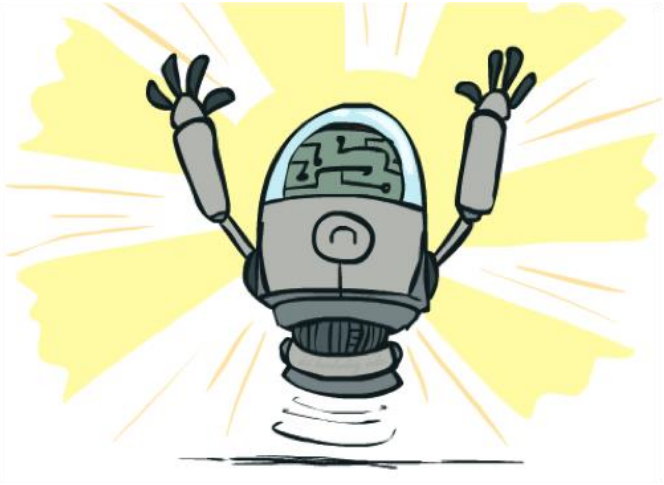


Video Demo Q-Learning -- Gridworld



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 (!)**



The Story So Far: MDPs and RL

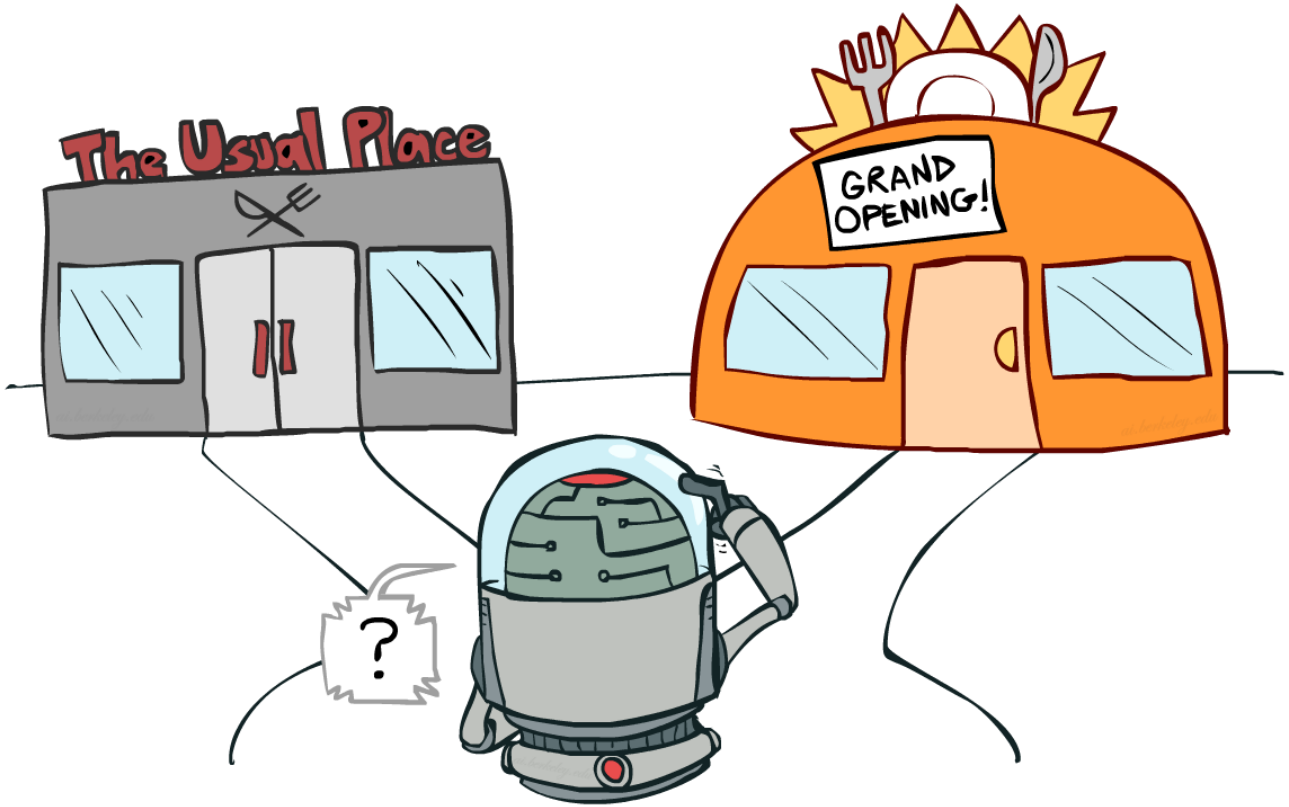
Known MDP: Offline Solution

Goal	Technique
Compute V*, Q*, π*	Policy iteration (PI)
Evaluate a fixed policy π	Policy evaluation (PE)

Unknown MDP: Model-Free Reinforcement Learning

Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	TD value learning

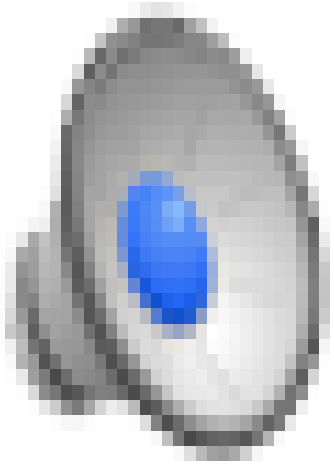
Exploration vs. Exploitation



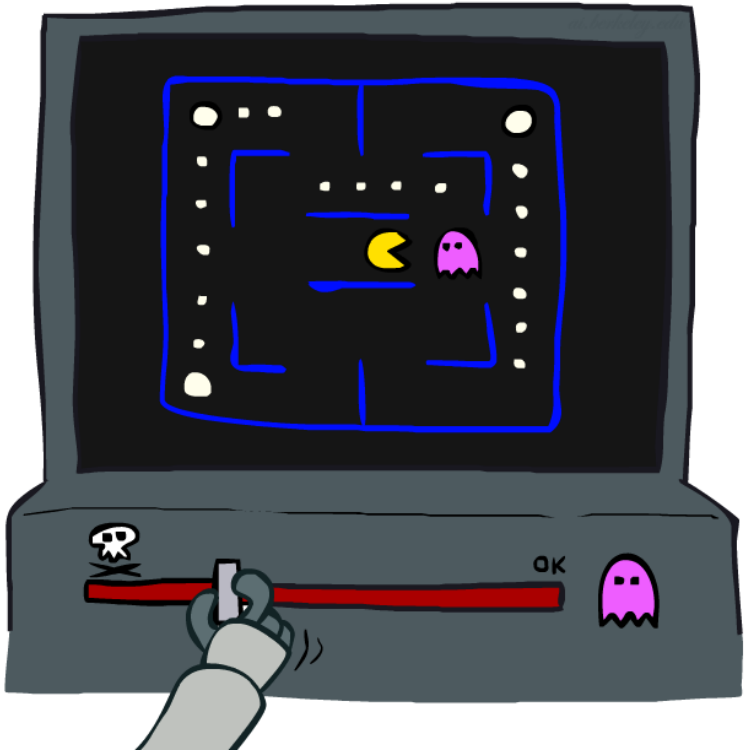
How to Explore (for Q-learning)?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip an "ε-weighted" coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, 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 ε over time
    - Another solution: Have separate train / test phases (set ε = 0 in testing; **do this in Assignment 4**)

Video Demo Q-Learning Auto Cliff Grid

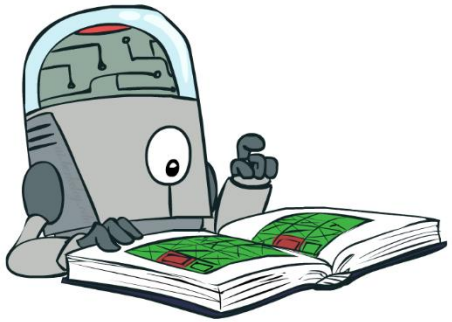


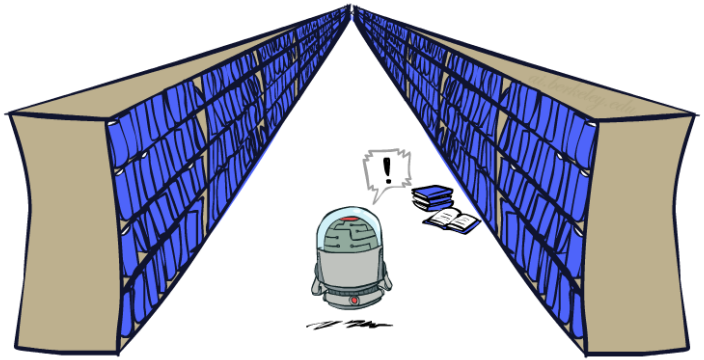
Approximate Q-Learning



Generalizing Across States

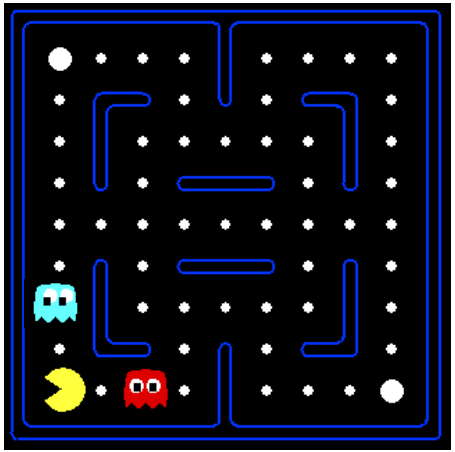
- Basic Q-Learning keeps a table of all Q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the Q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training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



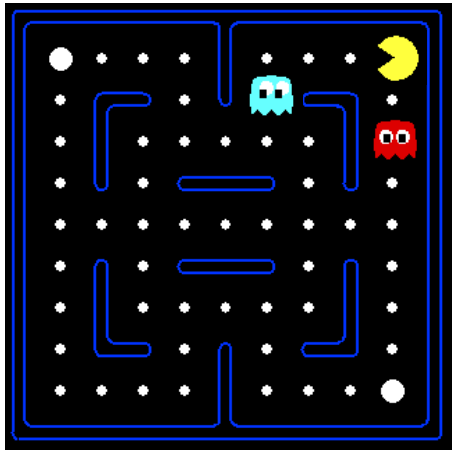


Example: Pacman

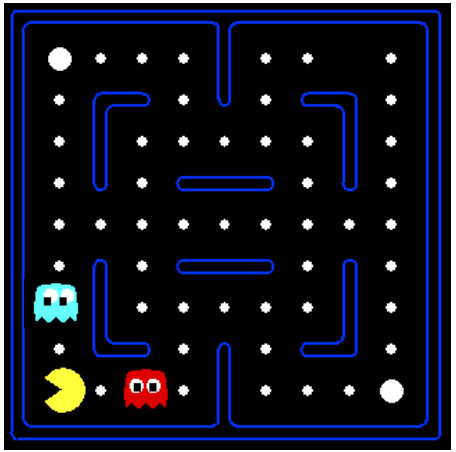
Let's say we discover through experience that this state is bad:



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

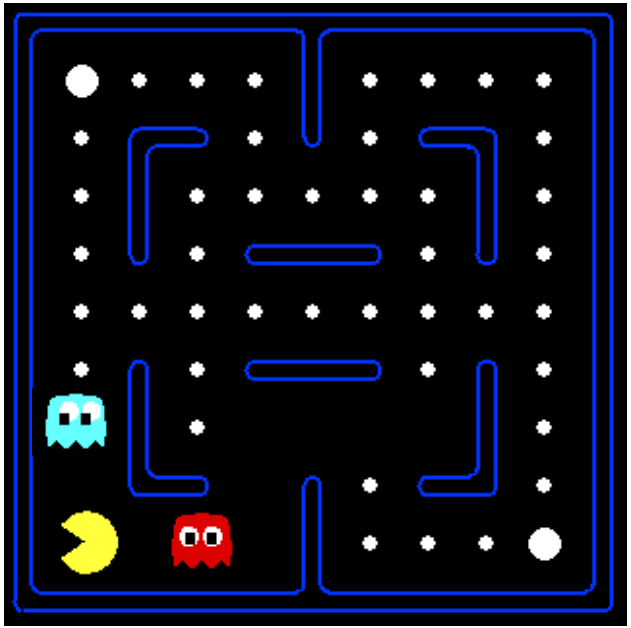


Or even this one!



Feature-Based Representations

- Solution: describe a state using a **vector of features (properties)**
  - Features are functions from states to real numbers that capture important properties of the state
  - Example features (for values):
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - 1 / (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. "Does action North in state X move Pacman closer to food?" (0/1))



Linear Value (and Q-value) Functions

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

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

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

- 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

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

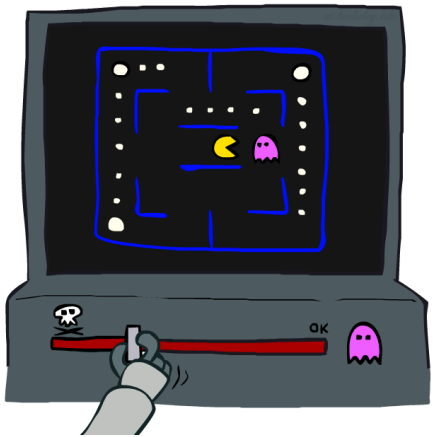
- Q-learning with linear Q-functions:

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

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

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

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

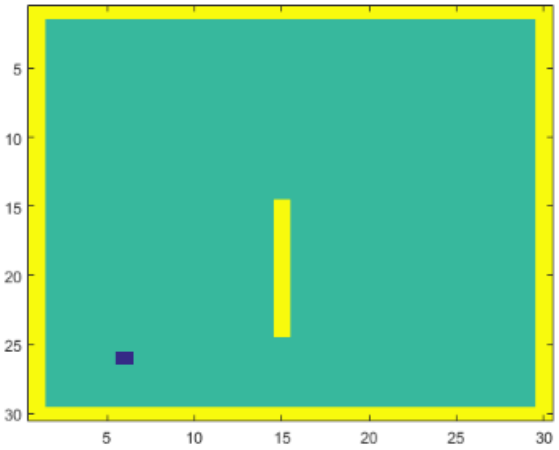


- Intuitive interpretation:
  - **Adjust weights of *active* features**
  - E.g., if something unexpectedly bad happens, **blame the features that had significant magnitude**: disprefer all states with that state's features

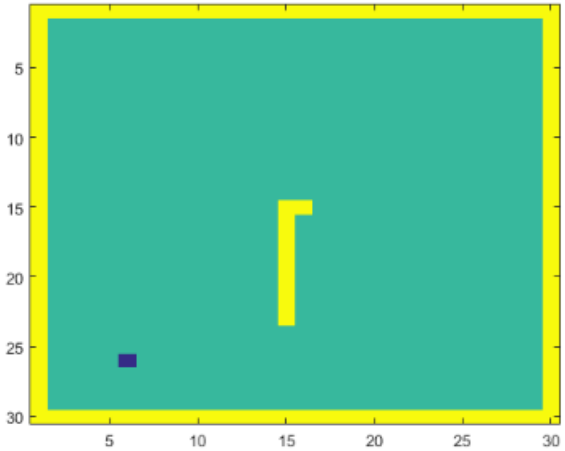
Assignment 4: Learning to Play *Snake*

- **Goal:** Familiarize yourself with today's content in the context of a classic video game called *Snake*
- 8 questions
  - First five are theoretical:
    - **State spaces** (question 1)
    - **Bellman equations** (questions 2-5)
  - Last three are programmatical/experimental:
    - Small *Snake* game via **policy iteration** (question 6)
    - Small *Snake* game via **tabular (full) Q-learning** (question 7)
    - Large *Snake* game via **approximate Q-learning** (question 8)
- ***Everything you need is in today's slides and in the assignment description!***

Snake Example (part 1 of 3)

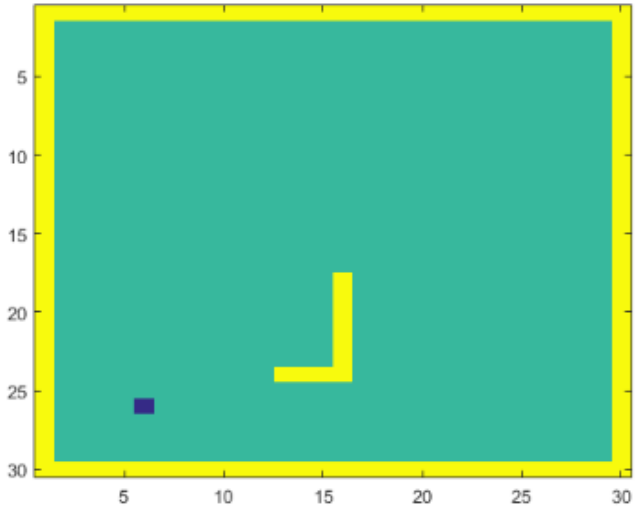


(a) Initial state. The length of the snake is 10 pixels. It forms a straight line of pixels, with its head located at the center of the grid, facing a random direction (N/E/S/W); in this case the direction is north (N). An apple is located at a random location, in this case in $(m, n) = (26, 6)$.

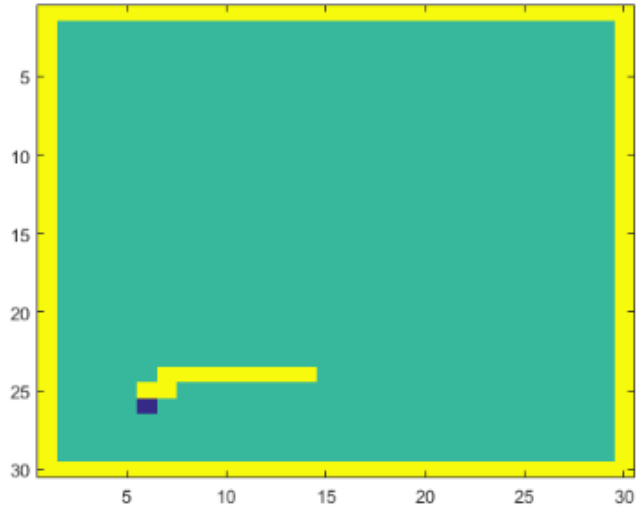


(b) At each time-step, the player must perform precisely one of three possible actions (movements): *left*, *forward*, or *right* (relative to the direction of the head). In this example, the player chooses to go right as their first action. The body of the snake moves along, keeping the snake intact.

Snake Example (part 2 of 3)

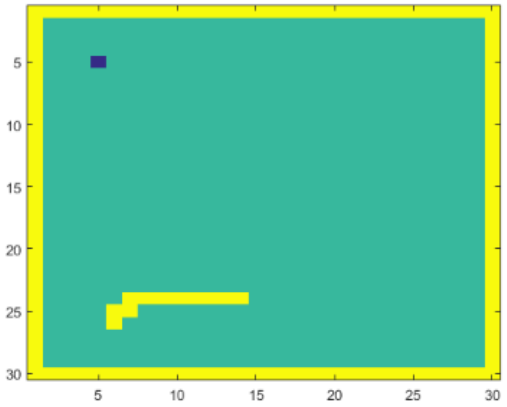


(c) After a few more time-steps, this is where the snake is located. Its direction is west. The initial apple has not yet been eaten.

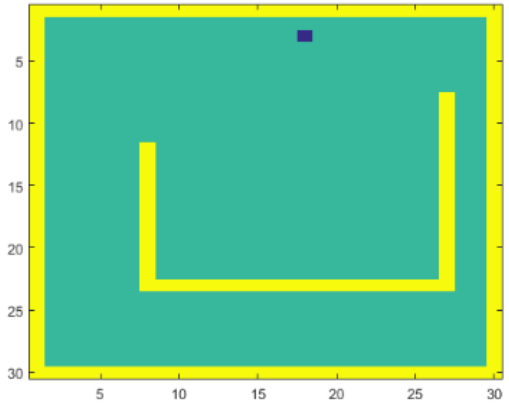


(d) After a few more time-steps, this is where the snake is located. Its direction is west. The initial apple has not yet been eaten.

Snake Example (part 3 of 3)



(e) Given the state shown in Figure 1d, the player chooses to go left; the new snake direction is south. The head of the snake eats the apple, causing the length of the snake to increase by 1; the body remains in the same place during the next movement, growing one pixel in the movement direction. The apple disappears and a new one is randomly placed on an empty pixel.



(f) After several more time-steps, the snake has gotten significantly longer by eating apples. Each apple eaten gives one point. The game ends when the player chooses an action such that the head ends up in an occupied space (meaning the body of the snake or a border). The final score is the total number of apples eaten during the game.

Reading

***Reinforcement Learning: An Introduction (2nd edition)*, Sutton & Barto 2016**

- **Chapter 3**
  - Markov Decision Processes
- **Chapter 4.1 – 4.3**
  - Policy Evaluation
  - Policy Improvement
  - Policy Iteration (= policy evaluation + policy improvement, alternating steps)
- **Chapter 6.1, 6.2, 6.5**
  - Temporal Difference (TD) Prediction
  - Advantages of TD Prediction Methods
  - Q-learning: Off-Policy TD Control