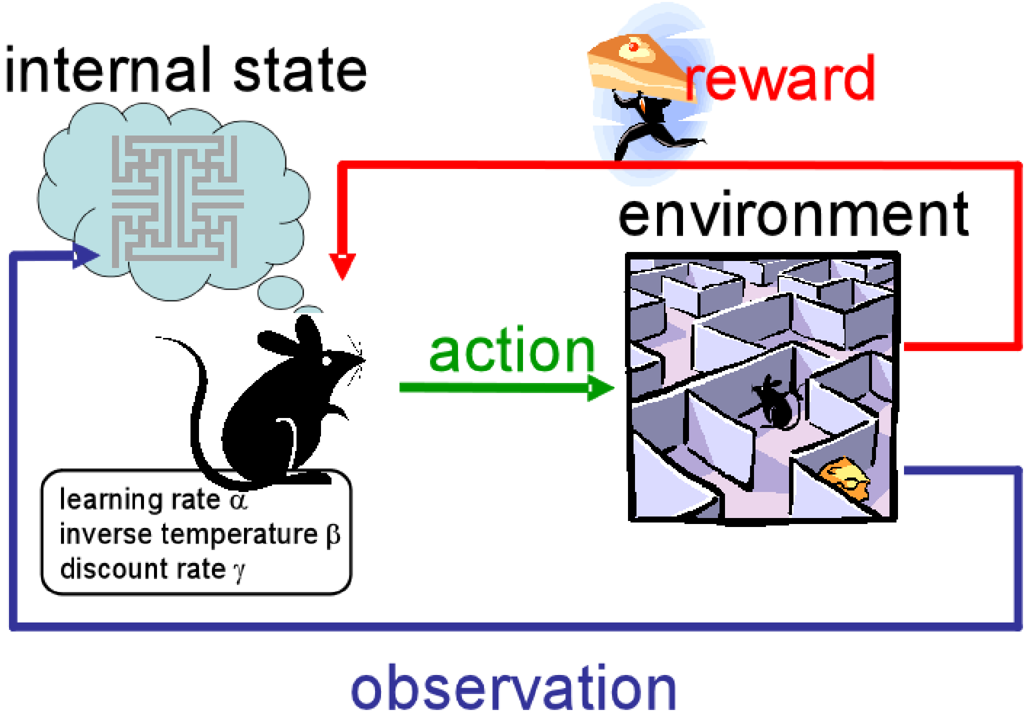
University of Ljubljana, Faculty of Computer and Information Science

Reinforcement learning



Prof Dr Marko Robnik-Šikonja
Intelligent Systems, Edition 2025

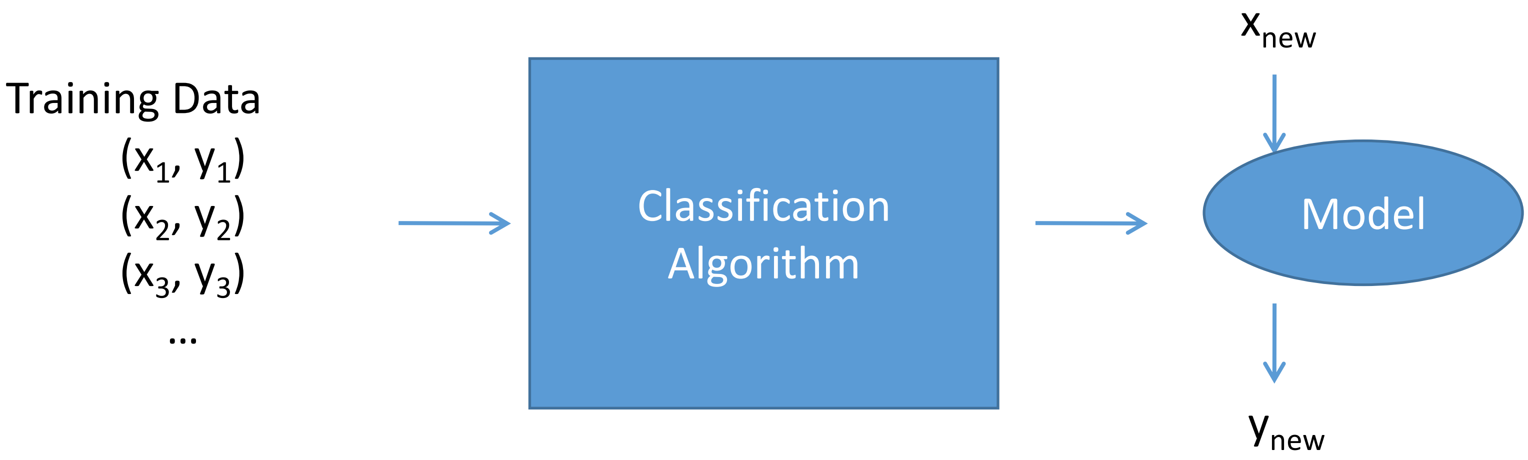
References

- R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction, 2018, 2nd edition (the book is freely available)
- many papers, tutorials, online courses
- Recently, a revival due to deep reinforcement learning and the use of reinforcement learning in LLM preference alignment

some slides are courtesy of Andrew Barto, Peter Bodik and Lisa Torrey

Machine Learning

- Classification:
  - Given
    - Training data
  - Learn
    - A model for making a single prediction or decision



Training Data

(x_1, y_1)

(x_2, y_2)

(x_3, y_3)

…

Classification Algorithm

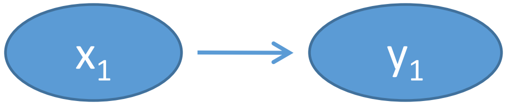
Model

x_{new}

y_{new}

Animal/Human Learning

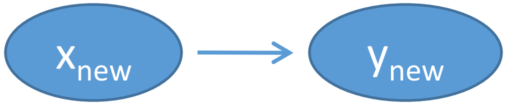
Memorization



x_1

y_1

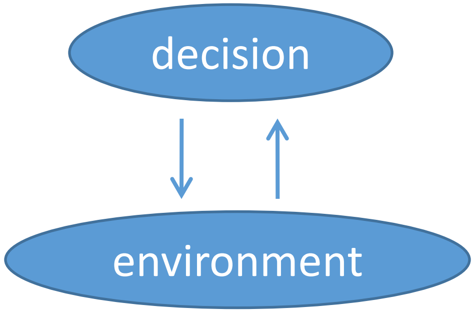
Classification



x_{new}

y_{new}

Procedural



decision

environment

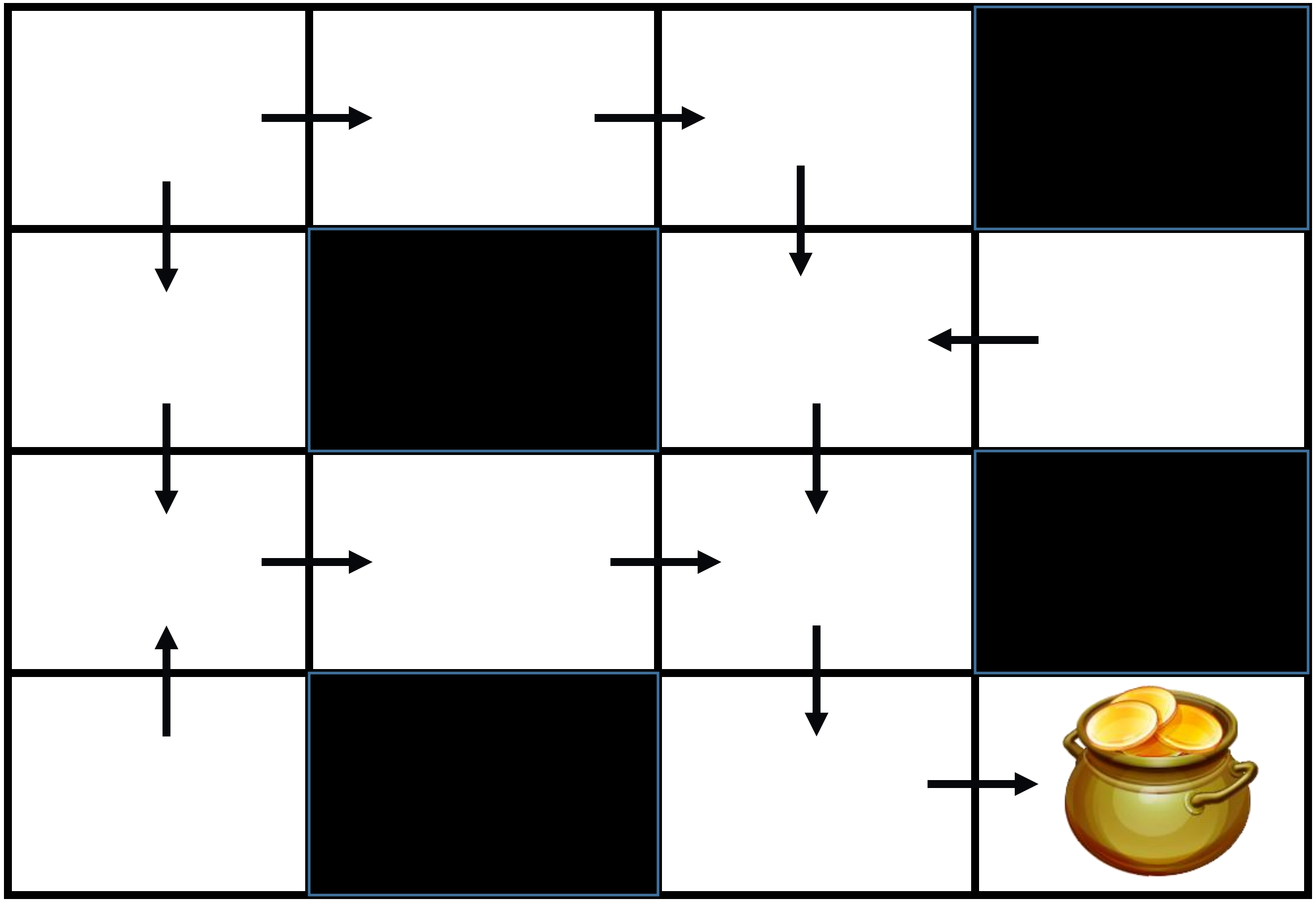
Other?

Procedural Learning

- Learning how to act to accomplish goals
  - Given
    - Environment that contains rewards
  - Learn
    - A policy for acting

- Important differences from classification
  - You don’t get examples of correct answers
  - You have to try things in order to learn

A Good Policy

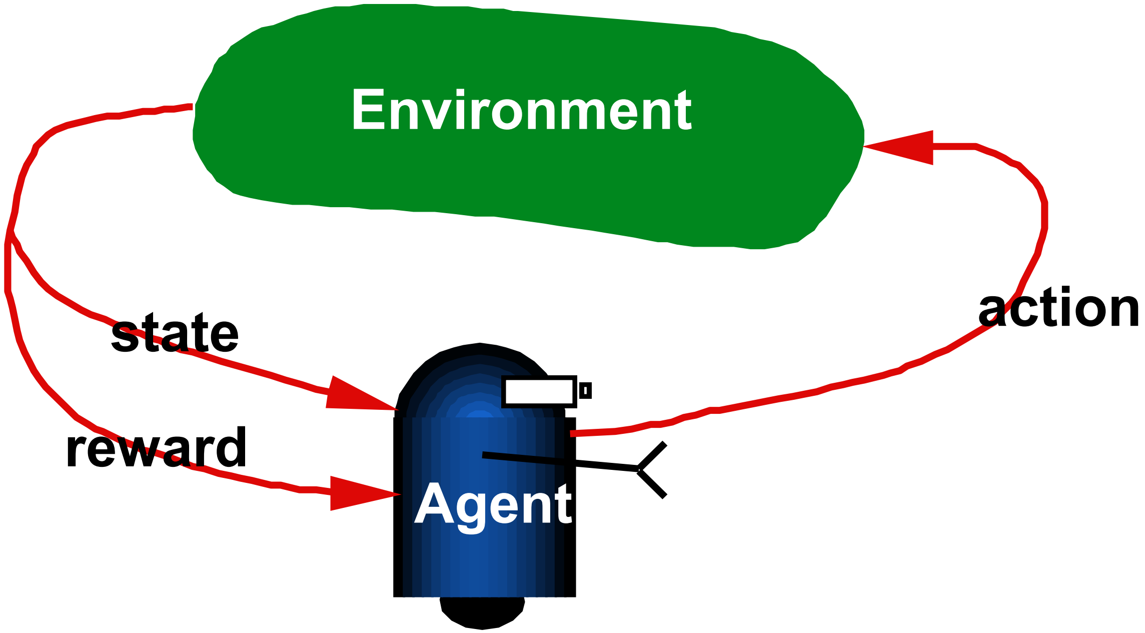


Introduction to Reinforcement Learning

- Reinforcement learning (RL)
- Questionable name stemming from behavioristic psychology (behavior reinforcement)
- Agent learning in the environment, performing actions
- Getting feedback from the environment (award, punishment), not necessary immediately
- Trying to learn a policy leading to goals
- An example: playing a game without knowing the rules; after 1000 moves, an opponent declares: You lost.

Agent

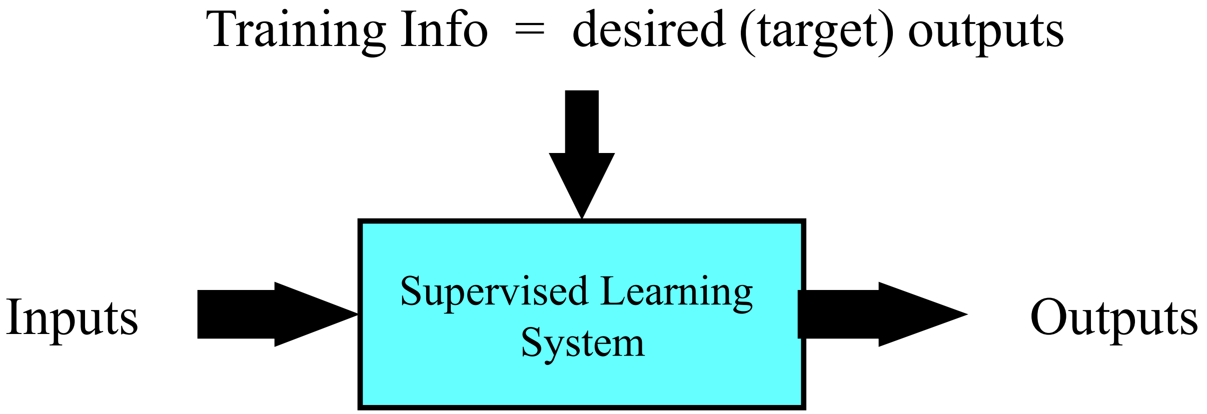
- Temporally situated
- Continual learning and planning
- Objective is to affect the environment
- Environment is stochastic and uncertain



What is Reinforcement Learning?

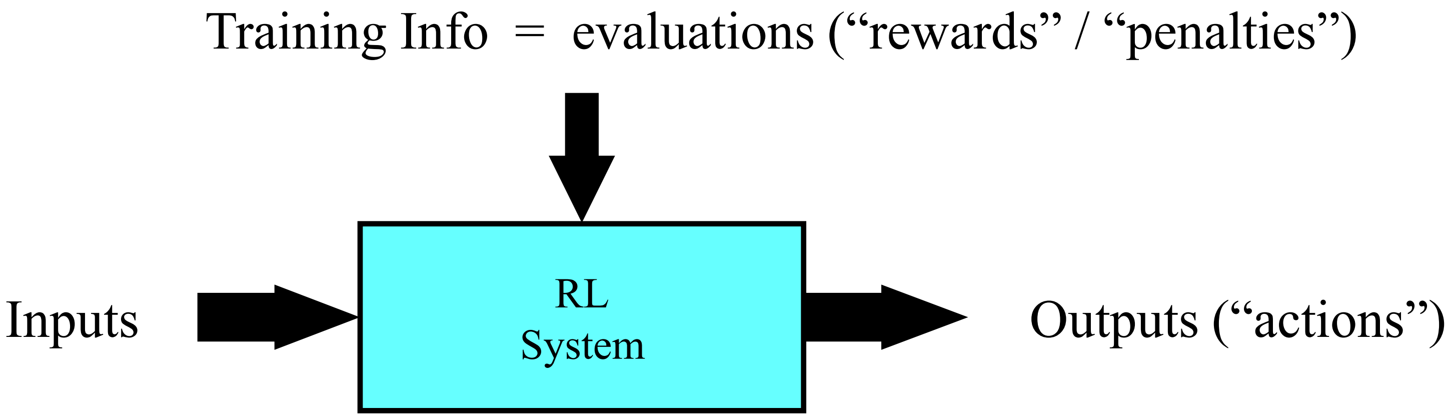
- Learning from interaction
- Goal-oriented learning: short-term and possible long-term awards
- Learning about, from, and while interacting with an external environment
- Learning what to do—how to map situations to actions—so as to maximize a numerical reward signal
- Agent discovers which action in what circumstances gives the highest award
- Agent can build a model of its environment
- RL is not supervised learning; it is about trial and error search, exploring, and getting information from the environment

Supervised Learning


Training Info = desired (target) outputs
Inputs
Supervised Learning System
Outputs

Error = (target output – actual output)

Reinforcement Learning



Objective: get as much reward as possible

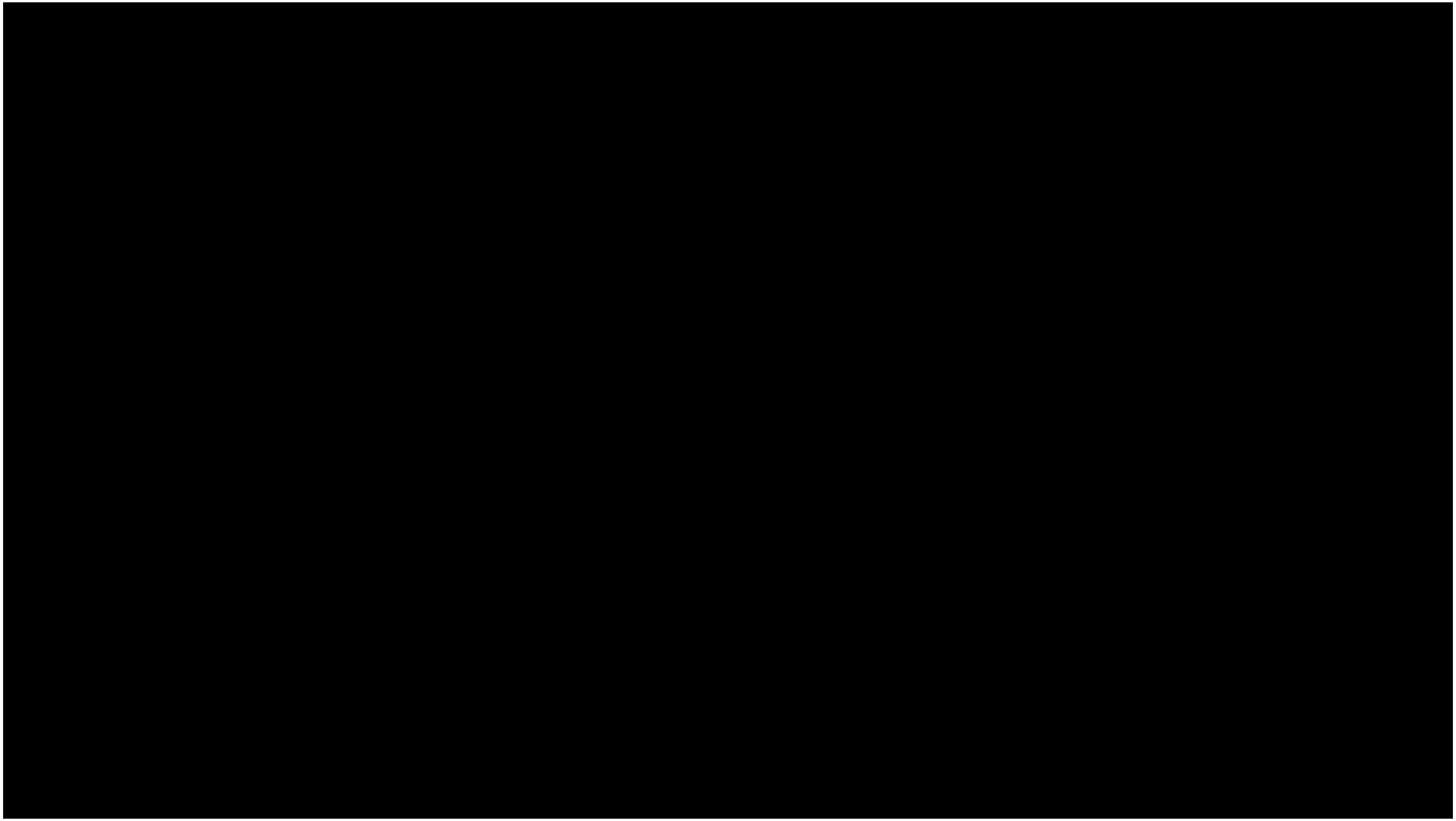
Key Features of RL

- Learner is not told which actions to take
- Trial-and-error search
- Possibility of delayed reward
  - Sacrifice short-term gains for greater long-term gains
- The need to ***explore*** and ***exploit***
- Considers the whole problem of a goal-directed agent interacting with an uncertain environment

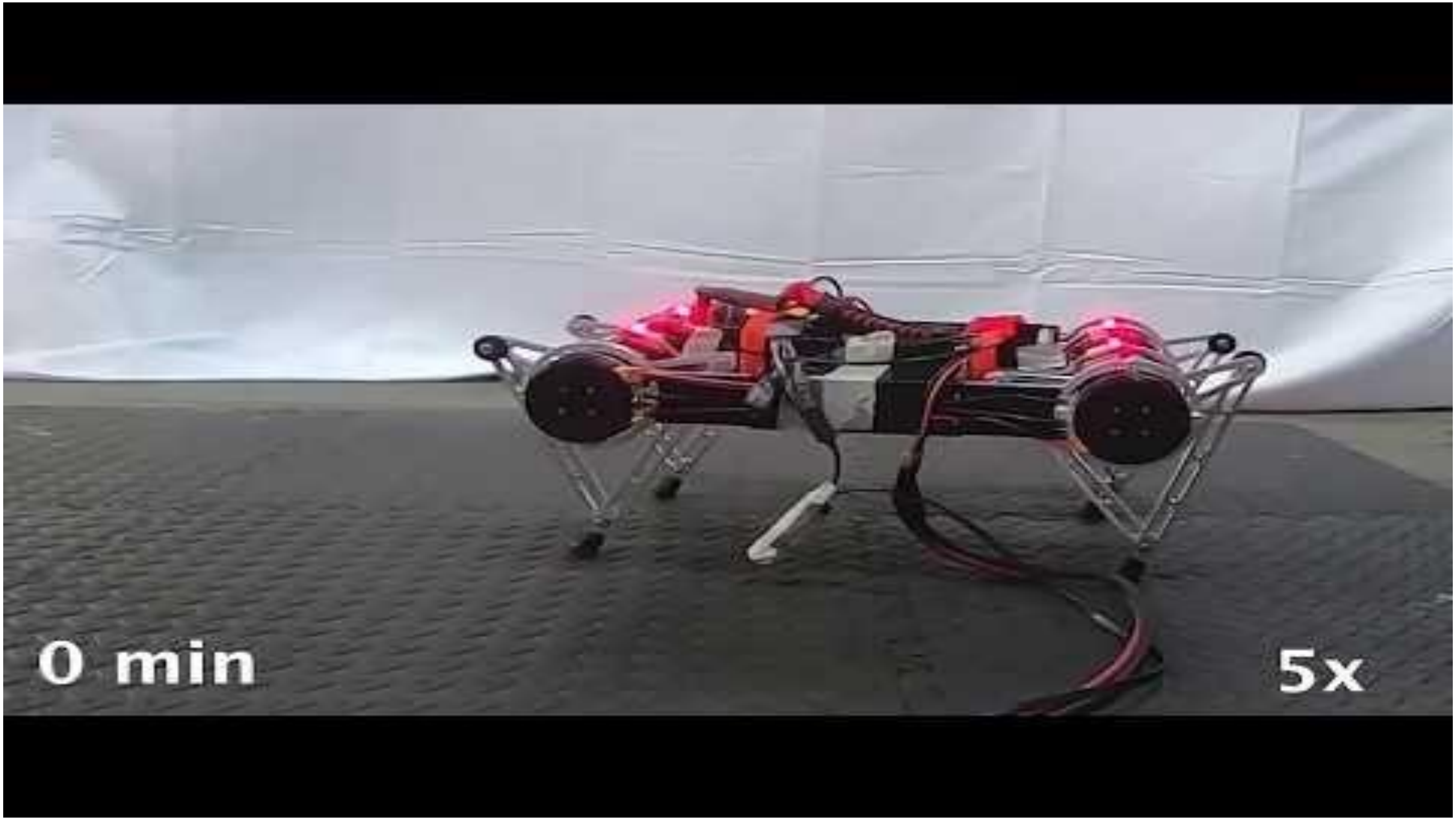
RL successful applications

- Robocup Soccer
- Financial asset management/Inventory management
- Dynamic Channel assignment in mobile communications
- Controlling elevators, industrial controllers, robots …
- Robots: navigation, grasping, moving …
- Games: backgammon (TD-Gammon, Jellyfish), Go (AlphaGo in combination with deep neural networks), Atari video games, poker, chess
- in LLMs: used in RLHF

Example video: Atari game Breakout



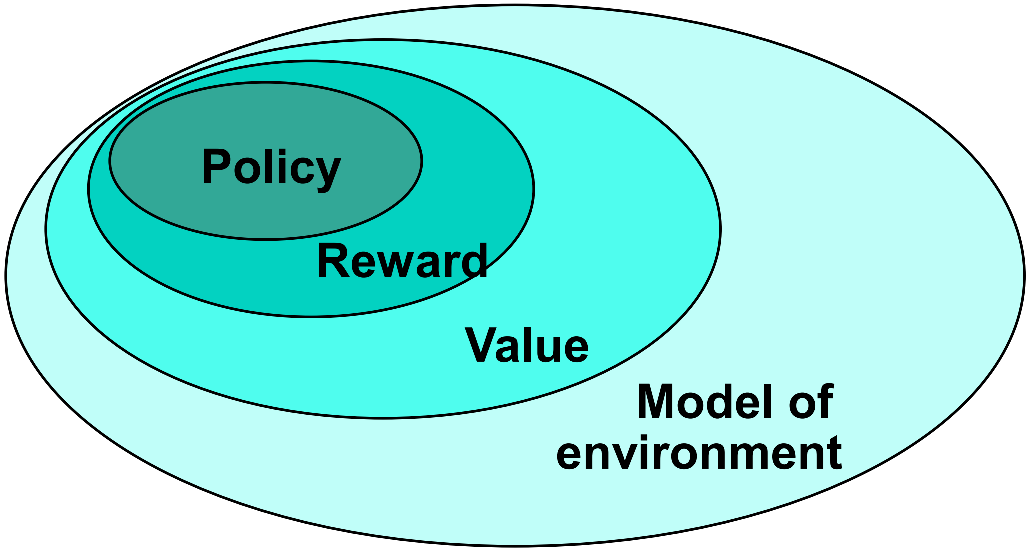
Example video:Robot training



Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, Sergey Levine. Learning to Walk via Deep Reinforcement Learning. Robotics: Science and Systems (RSS). 2019.

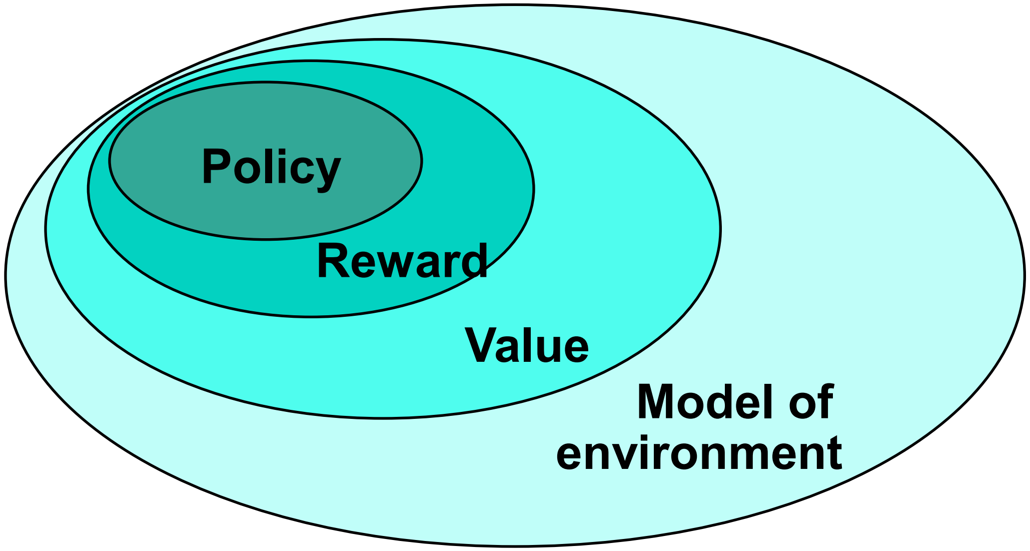
Example video: Bipedals

Components of RL 1/2



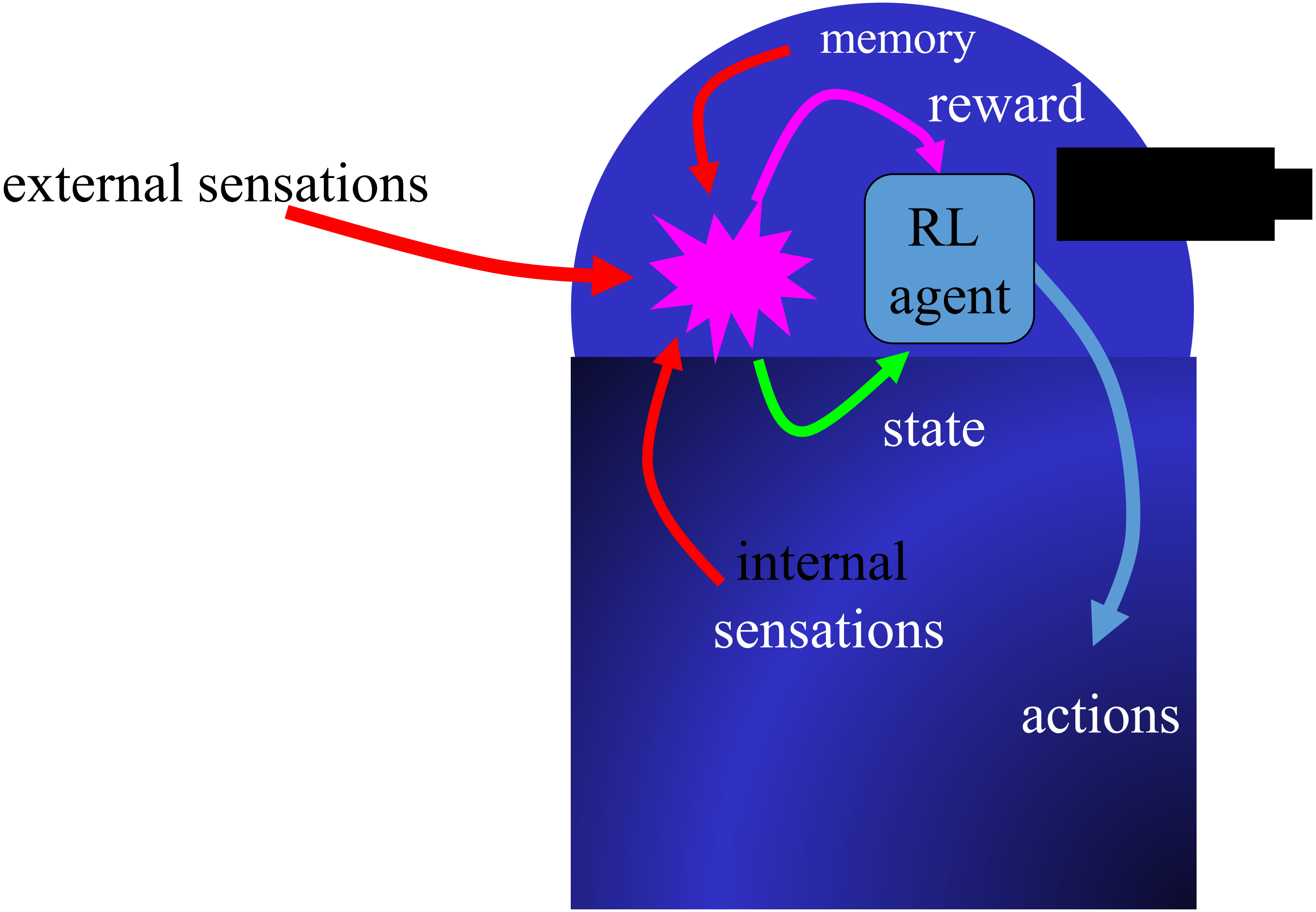
- **Policy**: what to do?
  - Defines agents choices and actions in a given time
  - Represented with rules, table, neural networks etc.
  - Result of search, planning, stochastic, etc.
- **Reward:** what is good?
  - Feedback from environment, agent tries to maximize it

Components of RL 2/2



- **Value**: internal representation of what is good, it *predicts* reward
  - Agent’s expectation of what can be expected in given state (long-term)
  - Implicitly contains evaluation of next states
  - Value has to be learned; use repetitions and sampling to estimate the value
- **Model**: what follows what
  - Internal representation of the environment
  - Agent can evaluate values and actions without performing them
  - Optional component

Agent from the point of view of RL



memory

reward

external sensations

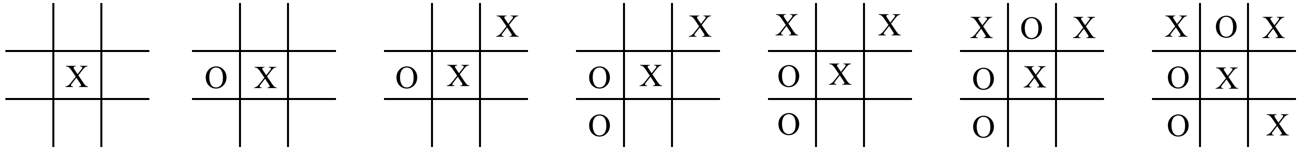
RL agent

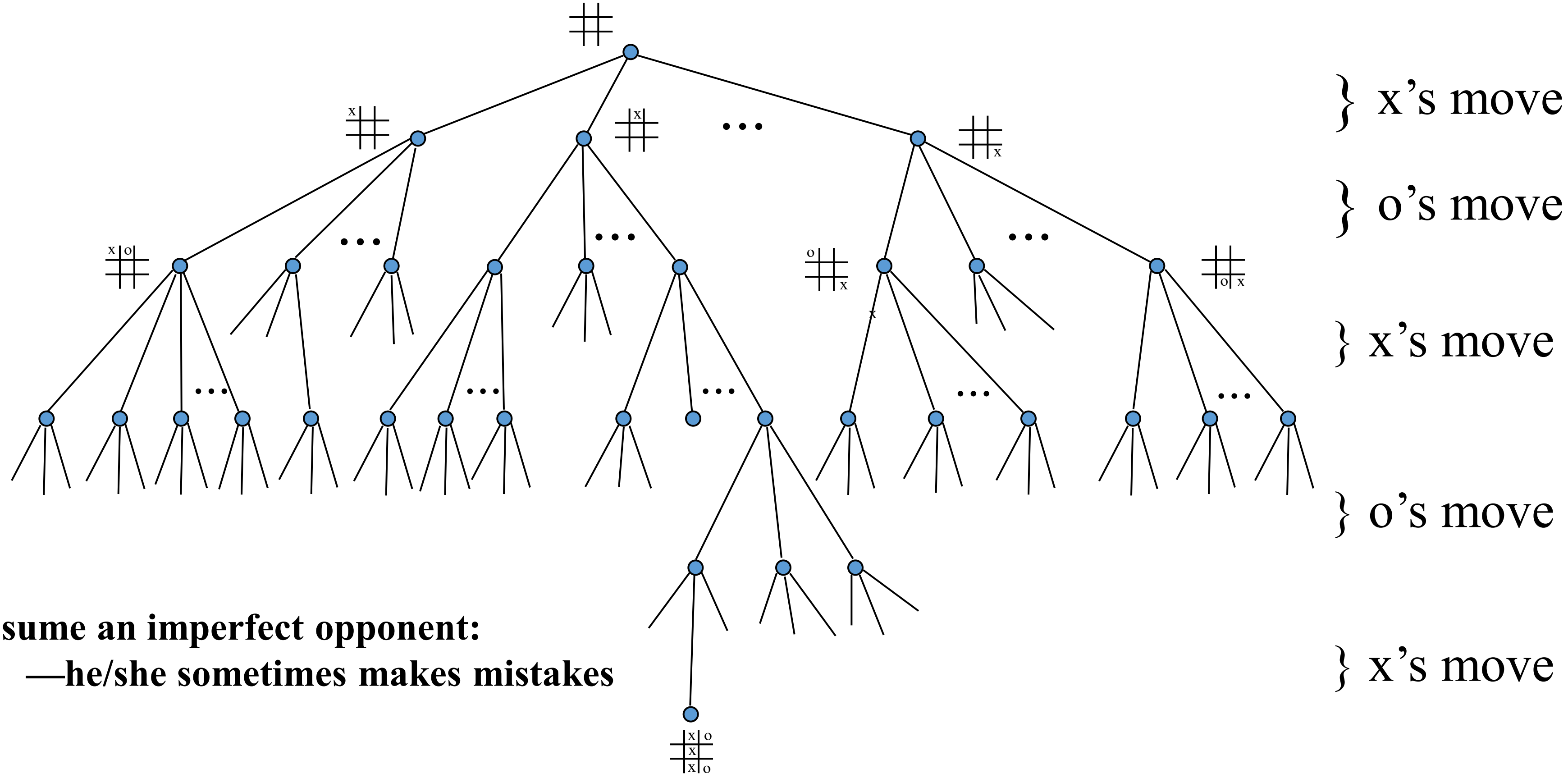
state

internal sensations

actions

An Example: Tic-Tac-Toe

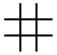
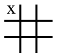
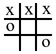
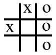
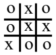




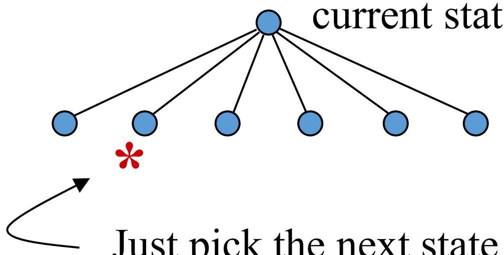
Assume an imperfect opponent:
—he/she sometimes makes mistakes

An RL Approach to Tic-Tac-Toe

1. Make a table with one entry per state:

State	$V(s)$ – estimated probability of winning	
	.5	?
	.5	?
⋮	⋮	
	1	win
⋮	⋮	
	0	loss
⋮	⋮	
	0	draw

2. Now play lots of games.
To pick our moves,
look ahead one step:



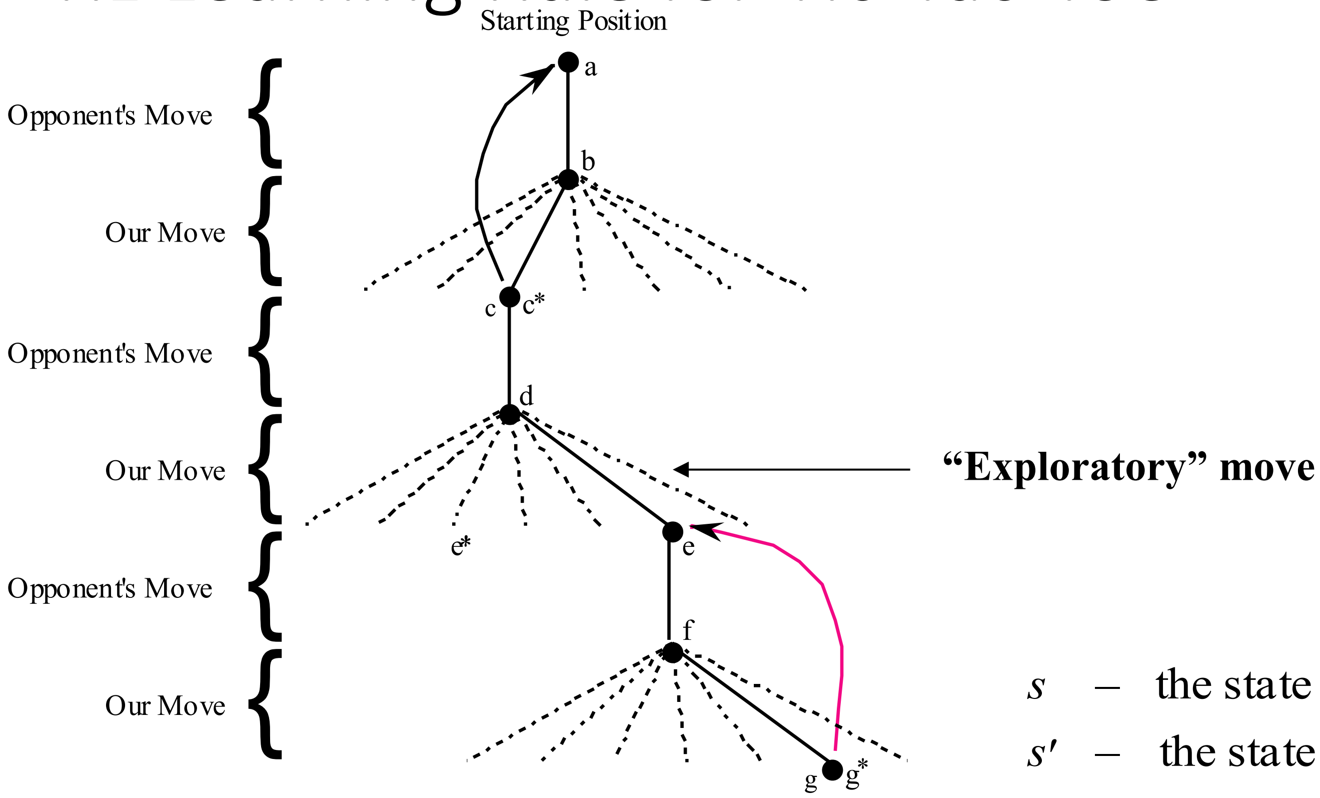
current state

various possible next states

Just pick the next state with the highest estimated prob. of winning — the largest $V(s)$; a ***greedy*** move.

But 10% of the time pick a move at random; an ***exploratory move***.

RL Learning Rule for Tic-Tac-Toe



s – the state before our greedy move

s' – the state after our greedy move

We increment each $V(s)$ toward $V(s')$ – a ***backup***:

$$V(s) \leftarrow V(s) + \alpha\big[V(s') - V(s)\big]$$

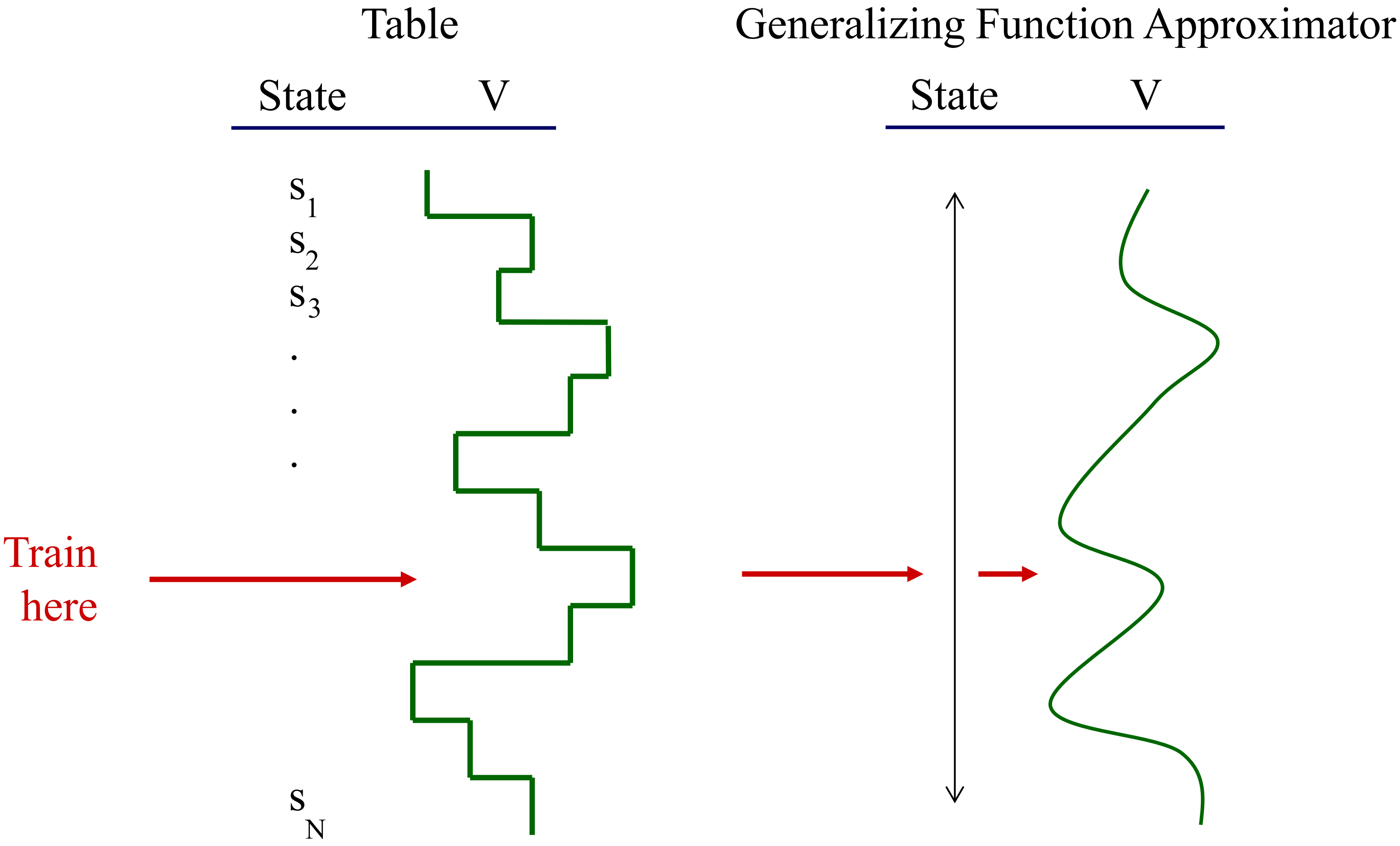
a small positive fraction, e.g., $\alpha = .1$

the ***step - size parameter***

How can we improve this TTT player?

- Take advantage of symmetries
  - representation/generalization
  - How might this backfire?
- Do we need “random” moves? Why?
  - Do we always need a full 10%?
- Can we learn from “random” moves?
- Can we learn offline?
  - Pre-training from self play?
  - Using learned models of opponent?
- . . .

E.g., generalization



Table

State V

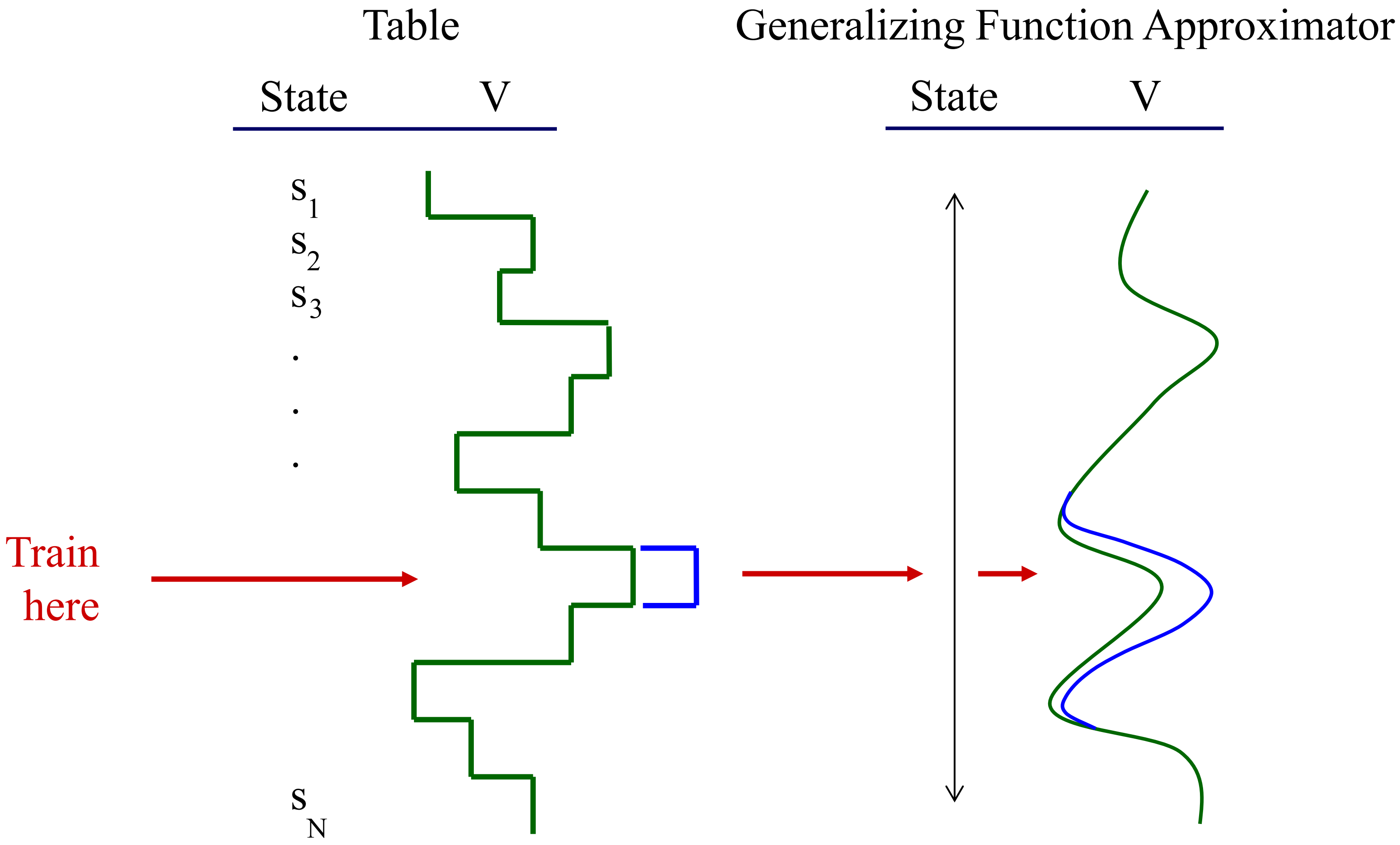
s_1
s_2
s_3
.
.
.
s_N

Train here

Generalizing Function Approximator

State V

E.g., generalization



Table

Generalizing Function Approximator

State V

State V

s_1

s_2

s_3

.

.

.

s_N

Train here

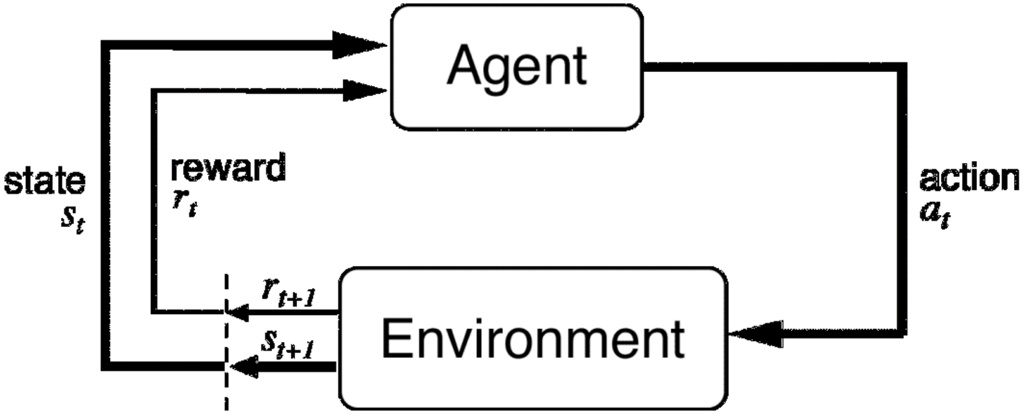
Tic-Tac-Toe is just a toy example

- Finite, small number of states
- One-step look-ahead is always possible
- State completely observable
- . . .

✹ RL is not limited to a finite number of states; in problems with infinite or very large number of states we only generate states encountered during search

✹ RL is not limited to games or opponent’s response

The Agent-Environment Interface



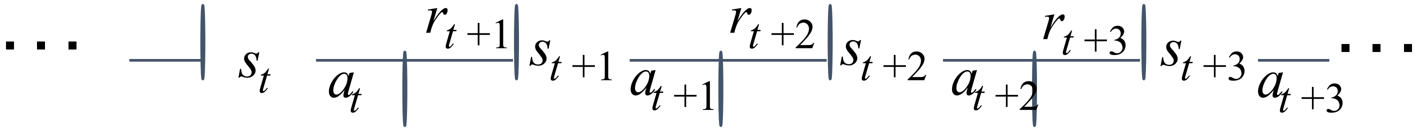
Agent and environment interact at discrete time steps : $t = 0, 1, 2, \dots$

Agent observes state at step t : $s_t \in S$

produces action at step t : $a_t \in A(s_t)$

gets resulting reward : $r_{t+1} \in \Re$

and resulting next state : s_{t+1}



The Agent Learns a Policy

Policy at step t, π_t :

a mapping from states to action probabilities

$\pi_t(s, a) =$ probability that $a_t = a$ when $s_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.
- How to represent a policy? With a table, function, ML model, etc.

Getting the degree of abstraction right

- Time steps need not refer to fixed intervals of real time.
- Actions can be low level (e.g., voltages to motors), or high level (e.g., accept a job offer), “mental” (e.g., shift in focus of attention),
- States can be low-level “sensations”, or they can be abstract, symbolic, based on memory, or subjective (e.g., the state of being “surprised” or “lost”).
- An RL agent is not like a *whole* animal or robot.
- Reward computation is in the agent’s environment because the agent cannot change it arbitrarily.
- The environment is not necessarily unknown to the agent, only incompletely controllable.

Goals and Rewards

- Is a scalar reward signal an adequate notion of a goal?—maybe not, but it is surprisingly flexible.
- A goal should specify **what** we want to achieve, not **how** we want to achieve it.
- A goal must be outside the agent's direct control—thus outside the agent.
- The agent must be able to measure success:
  - explicitly;
  - frequently during its lifespan.
- Human learning: we often get considerable better feedback on errors (not just a numeric score)

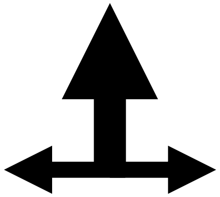
Robot in a room

			+1
	▨		-1
START			

actions: UP, DOWN, LEFT, RIGHT

UP

80%	move UP
10%	move LEFT
10%	move RIGHT

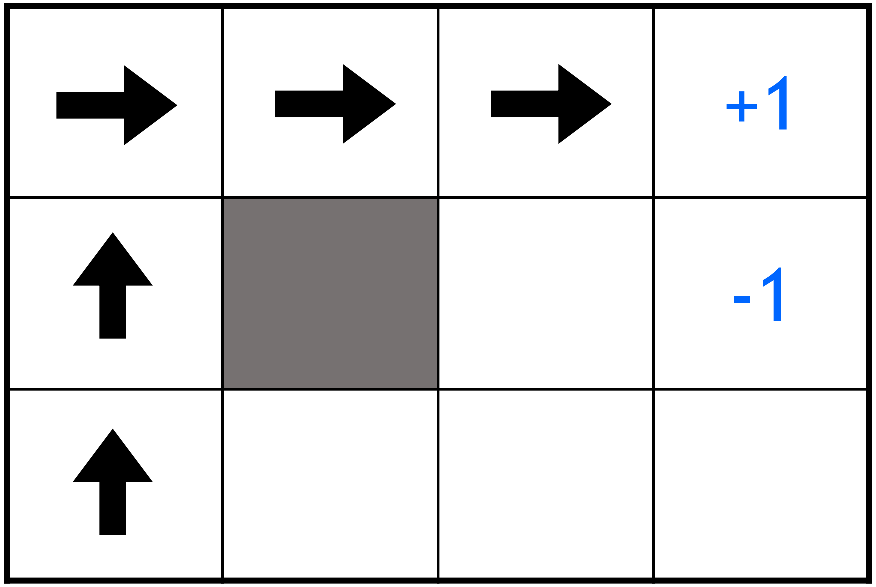


reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

- states
- actions
- rewards

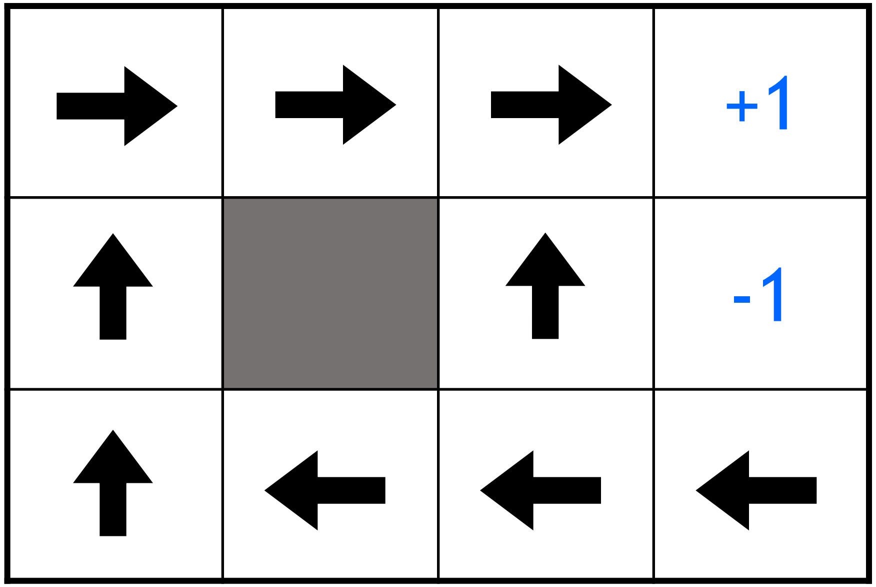
- what is the solution?

Is this a solution?



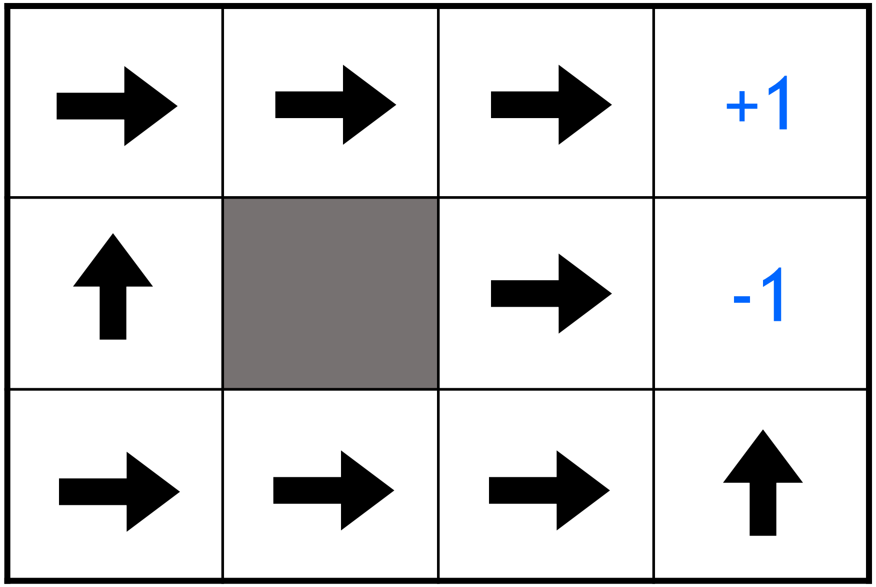
- only if actions are deterministic
  - not in this case (actions are stochastic)
- solution/policy
  - mapping from each state to an action

Optimal policy



→	→	→	+1
↑	■	↑	-1
↑	←	←	←

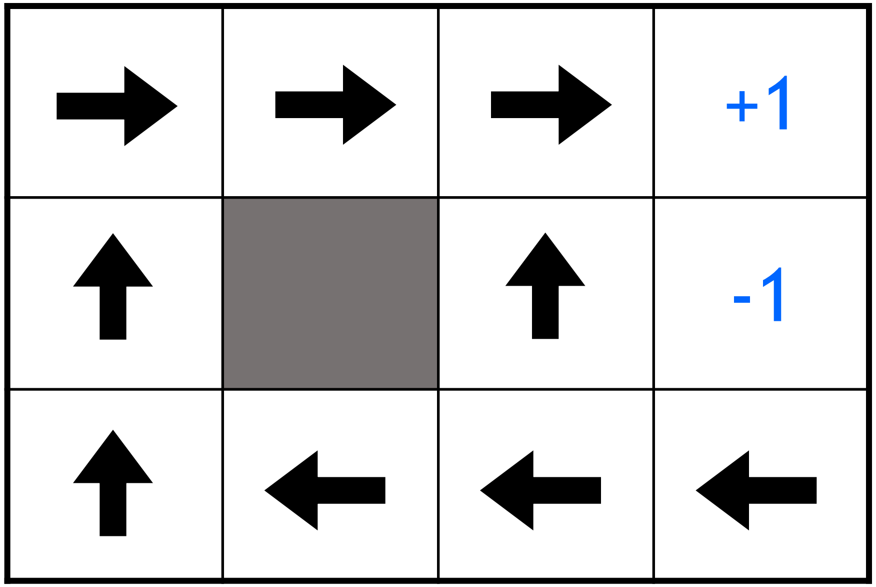
Reward for each step -2



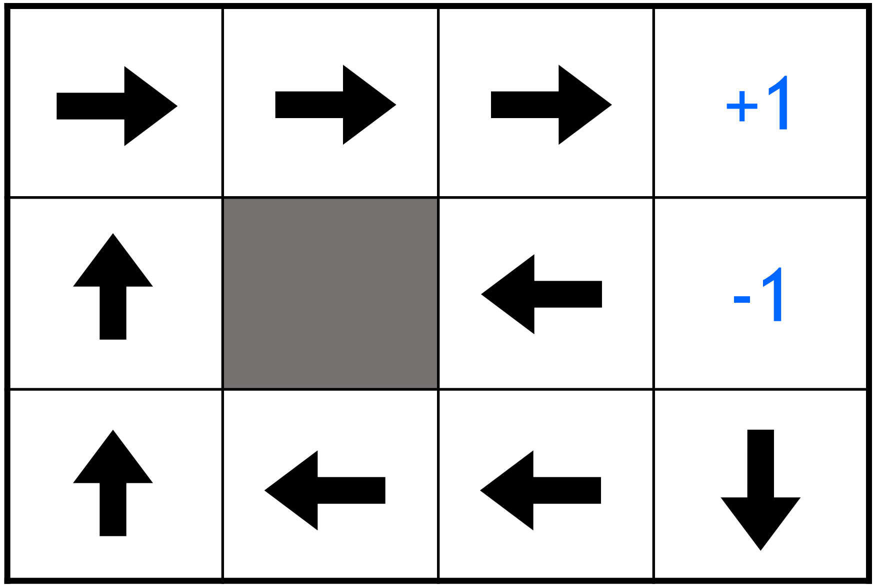
Reward for each step: -0.1

→	→	→	+1
↑	▨	↑	-1
↑	→	↑	←

Reward for each step: -0.04



Reward for each step: -0.01



Reward for each step: +0.01

↓	←	←	+1
↓	■	←	-1
←	←	←	↓

Returns

Suppose the sequence of rewards after step t is :

$$r_{t+1}, r_{t+2}, r_{t+3}, \ldots$$

What do we want to maximize?

In general,

we want to maximize the **expected return**, $E\{R_t\}$, for each step t.

Episodic tasks: interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze.

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

where T is a final time step at which a **terminal state** is reached, ending an episode.

Returns for Continuing Tasks

Continuing tasks: interaction does not have natural episodes.

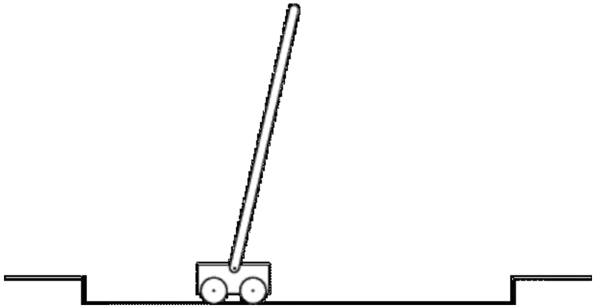
Discounted return:

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

where $\gamma, 0 \le \gamma \le 1,$ is the **discount rate**.

shortsighted $0 \leftarrow \gamma \rightarrow 1$ farsighted

An example: cart and pole



Avoid **failure:** the pole falling beyond a critical angle or the cart hitting end of track.

As an **episodic task** where episode ends upon failure:

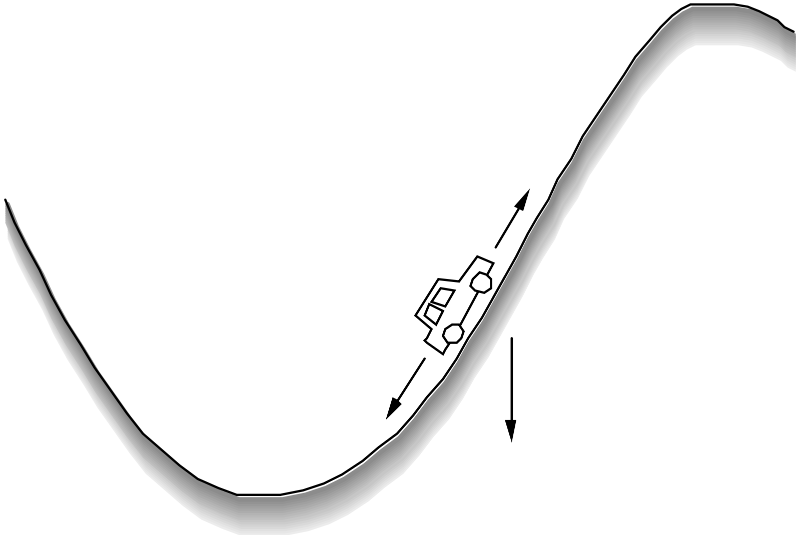
$$\text{reward} = +1 \text{ for each step before failure}$$
$$\Rightarrow \quad \text{return} = \text{number of steps before failure}$$

As a **continuing task** with discounted return:

$$\text{reward} = -1 \text{ upon failure; } 0 \text{ otherwise}$$
$$\Rightarrow \quad \text{return} = -\gamma^k, \text{ for } k \text{ steps before failure}$$

In either case, return is maximized by avoiding failure for as long as possible.

Another Example



Get to the top of the hill as quickly as possible.

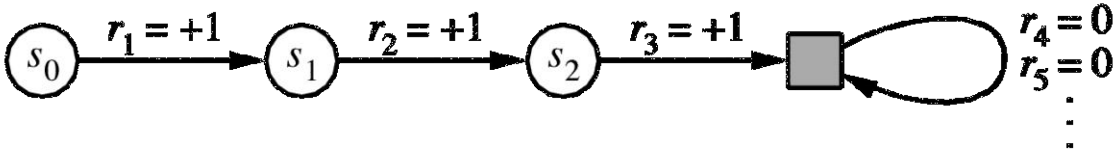
reward $= -1$ for each step where **not** at top of hill

$\Rightarrow$ return $=$ $-$ number of steps before reaching top of hill

Return is maximized by minimizing number of steps to reach the top of the hill.

A Unified Notation

- In episodic tasks, we number the time steps of each episode starting from zero.
- We usually do not have to distinguish between episodes, so we write s_t instead of $s_{t,j}$ for the state at step *t* of episode *j*.
- Think of each episode as ending in an absorbing state that always produces reward of zero:



- We can cover all cases by writing

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

where γ can be 1 only if a zero reward absorbing state is always reached.

Episodic task – finite horizon

- In time *t,* the agent is interested in *h* further states
- Rewards in that time are r_{t+1}, r_{t+2}, r_{t+3}, ..., r_{t+h}

  $R_t = r_{t+1} + r_{t+2} + r_{t+3} + \dots + r_{t+h}$
- The agent maximizes expected reward in that period

$$\max E(R_t) = \max E(\sum_{k=1}^{h} r_{t+k})$$

Finite horizon

- Two optimal behaviors
  - h-step optimal action: on step 1, do an action which is optimal under assumption that h-1 actions will follow, on step 2 do an action which is optimal under assumption that h-2 actions will follow ...
  - h-step receding action: on each step do an action which is optimal under assumption that h actions will follow
- Limited look-ahead
- Suitability of finite horizon: episodic missions (e.g., labyrinth)

Continuous tasks

- No natural end, but …

… nearer actions are more important than more distant ones

- Agent optimizes infinite sequence of rewards
- Rewards are geometrically discounted
- rewards: $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4,}$ ... for $0 < \gamma < 1$
- γ (discount factor) can be interpreted as interest rate, a trick to bound an infinite sum, probability of surviving another step, short/far-sightedness

$$\max E(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}), \quad 0 < \gamma < 1$$

Average reward model

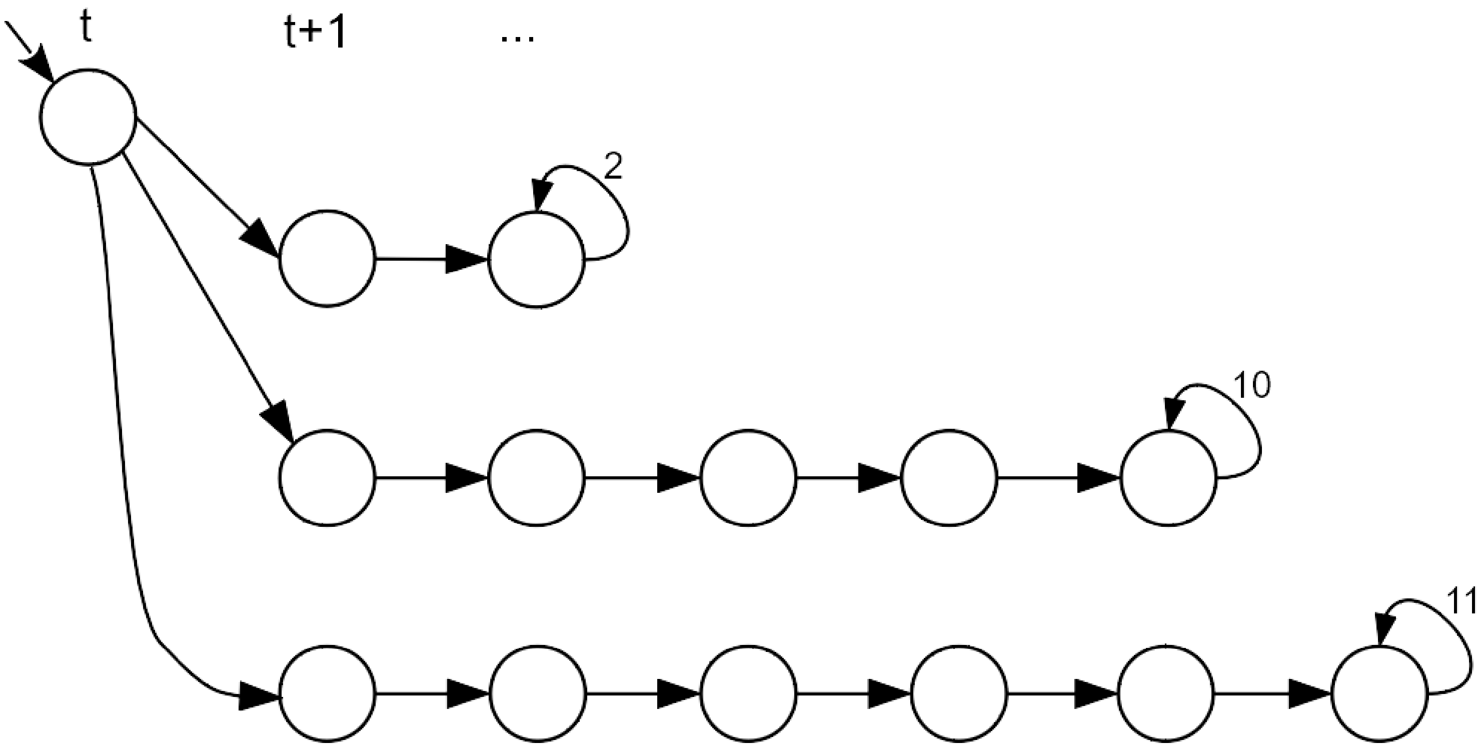
- Agent optimizes long-term average reward

$$\lim_{h\to\infty} E(\frac{1}{h}\sum_{k=1}^{h} r_{t+k})$$

- Downside: does not know the difference between near and distant rewards

An example: rewards

1. finite horizont, h=4
2. infinite horizont, γ=0.9
3. average expected reward



The Markov Property

- By “the state” at step *t*, we mean whatever information is available to the agent at step *t* about its environment.
- The state can include immediate “sensations,” highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations so as to retain all “essential” information, i.e. it should have the **Markov Property**:

$$\Pr\left\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right\} =$$

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

for all s', r, and histories $s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$.

Markov Decision Processes

- If a reinforcement learning task has the Markov Property, it is basically a **Markov Decision Process (MDP)**.
- If state and action sets are finite, it is a **finite MDP**.
- To define a finite MDP, you need to give:
  - **state and action sets**
  - one-step “dynamics” defined by **transition probabilities**:

$$\mathbf{P}^{a}_{ss'} = \Pr\left\{s_{t+1} = s' \,\middle|\, s_t = s, a_t = a\right\} \quad \text{for all } s, s' \in S, a \in A(s).$$

  - **reward probabilities**:

$$\mathbf{R}^{a}_{ss'} = E\left\{r_{t+1} \,\middle|\, s_t = s, a_t = a, s_{t+1} = s'\right\} \quad \text{for all } s, s' \in S, a \in A(s).$$

An Example of Finite MDP

Recycling Robot

- At each step, robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge.
- Searching is better but runs down the battery; if runs out of power while searching, has to be rescued (which is bad).
- Decisions made on basis of current energy level: `high`, `low`.
- Reward = number of cans collected

Recycling Robot MDP

$S = \{\texttt{high}, \texttt{low}\}$

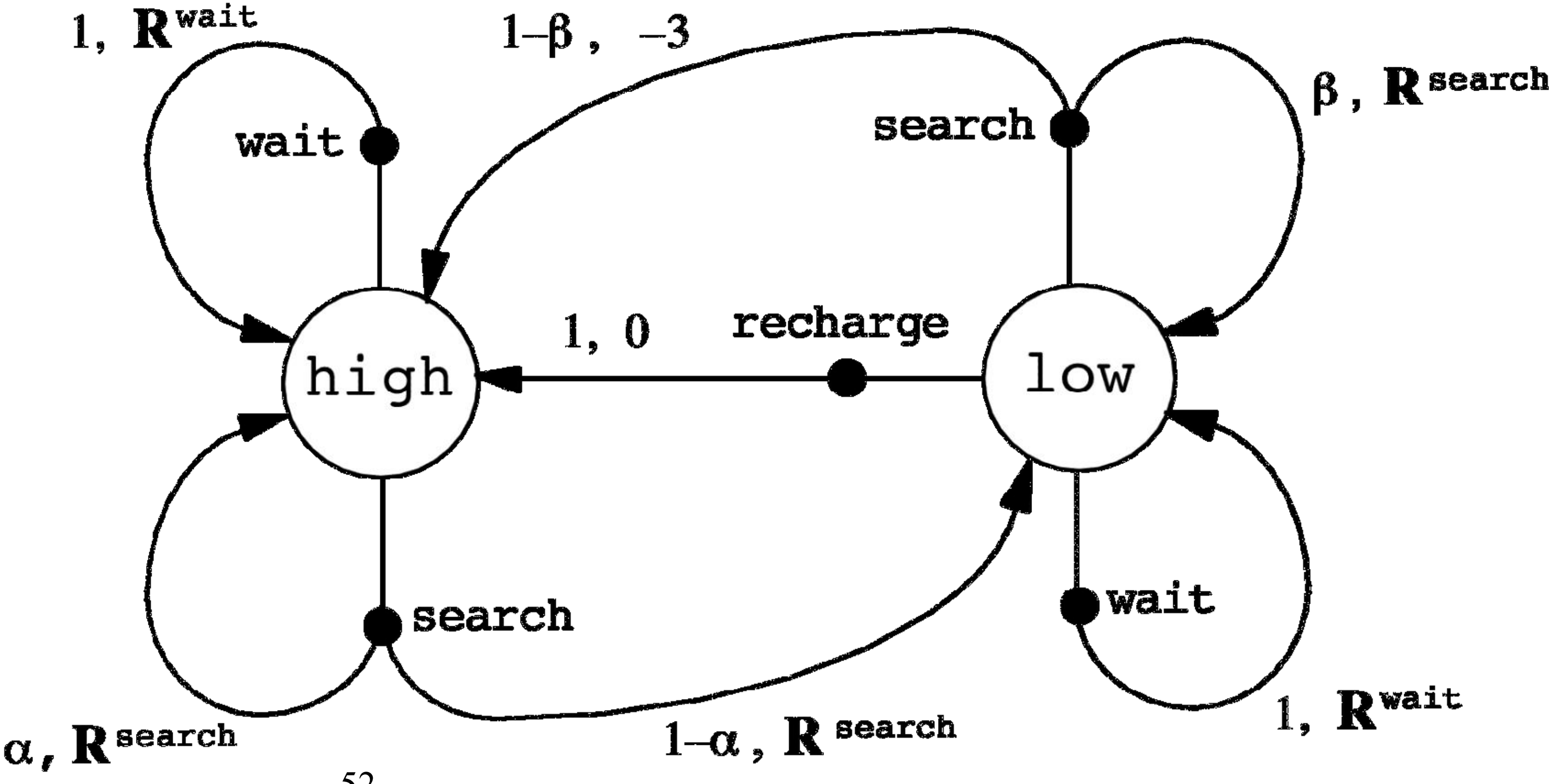
$A(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$

$A(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$

$\mathbf{R}^{\texttt{search}}$ = expected no. of cans while searching

$\mathbf{R}^{\texttt{wait}}$ = expected no. of cans while waiting

$\mathbf{R}^{\texttt{search}} > \mathbf{R}^{\texttt{wait}}$



Value Functions

- The **value of a state** is the expected return starting from that state; depends on the agent's policy:

State - value function for policy π :

$$V^{\pi}(s) = E_{\pi}\left\{R_t \mid s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}$$

- The **value of taking an action in a state under policy** π is the expected return starting from that state, taking that action, and thereafter following π :

Action - value function for policy π :

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

Bellman Equation for policy π

The basic idea:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} \cdots \\ &= r_{t+1} + \gamma \left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} \cdots \right) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

So:

$$\boxed{\begin{aligned} V^\pi(s) &= E_\pi \left\{ R_t \mid s_t = s \right\} \\ &= E_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s \right\} \end{aligned}}$$

Or, without the expectation operator:

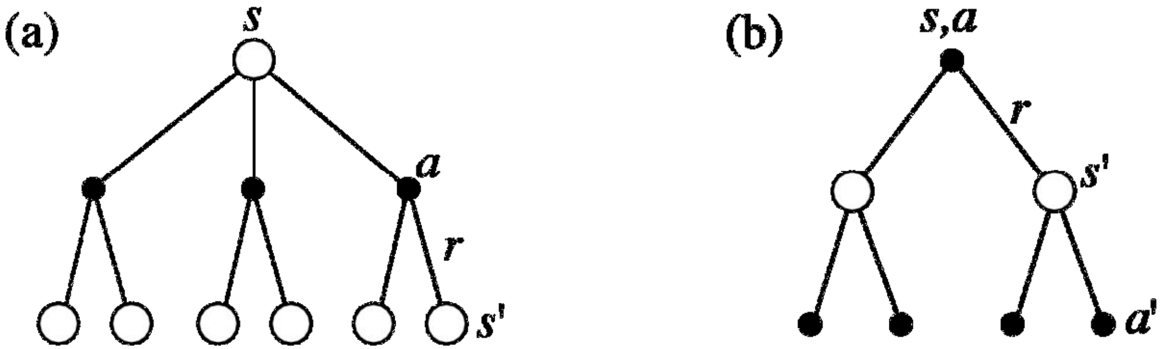
$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} \mathbf{P}^a_{ss'} \left[\mathbf{R}^a_{ss'} + \gamma V^\pi(s') \right]$$

More on the Bellman Equation

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

This is a set of equations (in fact, linear), one for each state. The value function for π is its unique solution.

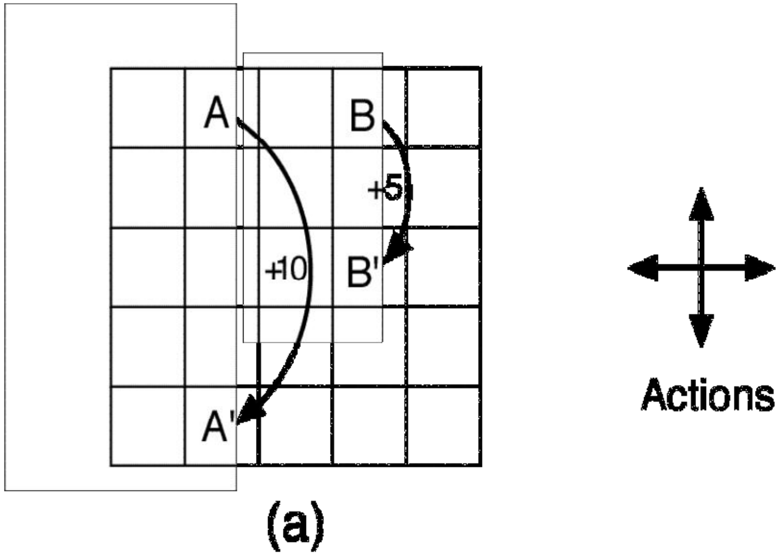
Backup diagrams:



for V^{π} for Q^{π}

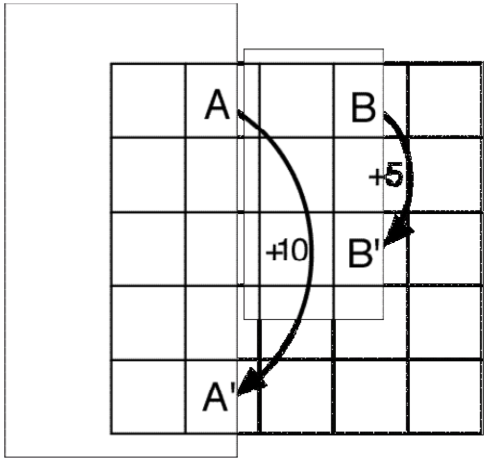
Gridworld

- Actions: `north`, `south`, `east`, `west`; deterministic.
- If it would take agent off the grid: no move but reward = –1
- Other actions produce reward = 0, except actions that move agent out of special states A and B, as shown.

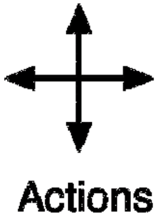


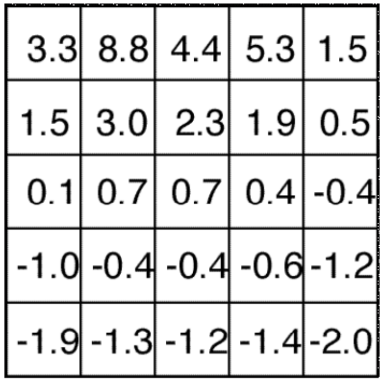
Gridworld

- Actions: `north`, `south`, `east`, `west`; deterministic.
- If would take agent off the grid: no move but reward = –1
- Other actions produce reward = 0, except actions that move agent out of special states A and B as shown.



(a)





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

(b)

State-value function for equiprobable random policy; $\gamma = 0.9$

Optimal Value Functions

- For finite MDPs, policies can be **partially ordered**:

$$\pi \geq \pi' \quad \text{if and only if} \quad V^{\pi}(s) \geq V^{\pi'}(s) \ \text{ for all } s \in S$$

- There are always one or more policies that are better than or equal to all the others. These are the **optimal policies**. We denote them all π*.
- Optimal policies share the same **optimal state-value function**:
- Optimal policies also share the same **optimal action-value function**:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad \text{for all } s \in S$$

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \ \text{ for all } \ s \in S \text{ and } a \in A(s)$$

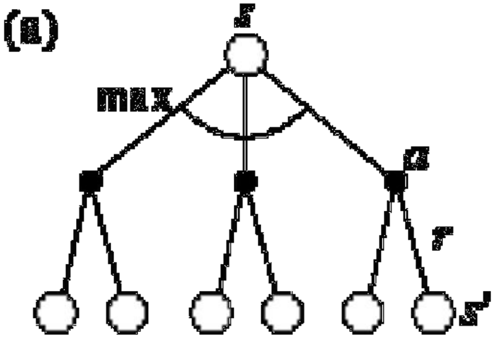
This is the expected return for taking action a in state s and thereafter following an optimal policy.

Bellman Optimality Equation for *V**

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$
\begin{aligned}
V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s,a) \\
&= \max_{a \in A(s)} E\left\{ r_{t+1} + \gamma V^*(s_{t+1}) \,\middle|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in A(s)} \sum_{s'} \mathbf{P}^a_{ss'} \left[\mathbf{R}^a_{ss'} + \gamma V^*(s') \right]
\end{aligned}
$$

The relevant backup diagram:



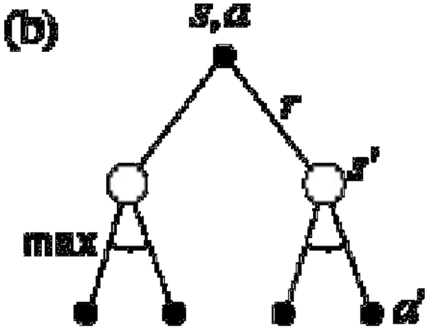
V^* is the unique solution of this system of nonlinear equations.

Bellman Optimality Equation for *Q**

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1},a') \middle| s_t = s, a_t = a\right\}$$

$$= \sum_{s'} \mathbf{P}^a_{ss'} \left[\mathbf{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')\right]$$

The relevant backup diagram:



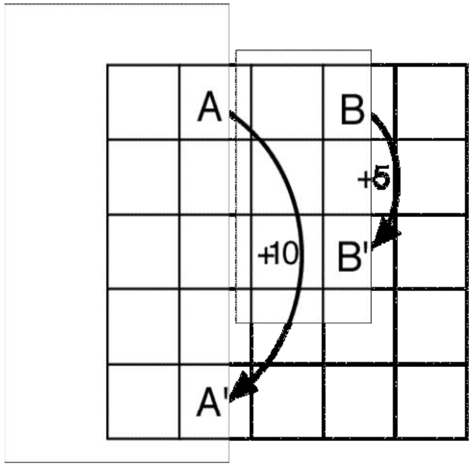
Q^* is the unique solution of this system of nonlinear equations.

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to V^* is an optimal policy.

Therefore, given V^*, one-step-ahead search produces the long-term optimal actions.

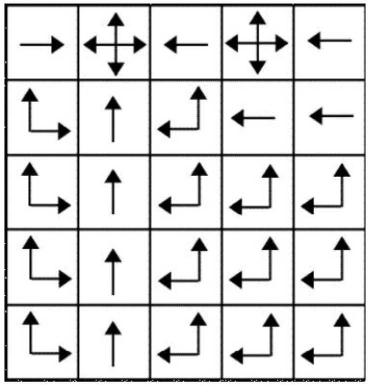
E.g., back to the gridworld:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b) V^*



c) ≠* π^*

What About Optimal Action-Value Functions?

Given Q^*, the agent does not even have to do a one-step-ahead search:

$$\boxed{\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s,a)}$$

Solving the Bellman Optimality Equation

- Finding an optimal policy by solving the Bellman optimality equation requires the following:
  - accurate knowledge of environment dynamics;
  - enough space and time to do the computation;
  - the Markov property.
- How much space and time do we need?
  - polynomial in number of states (via dynamic programming methods),
  - BUT, number of states is often huge (e.g., backgammon has about 10^{20} states).
- We usually have to settle for approximations.
- Many RL methods can be understood as approximately solving the Bellman optimality equation.

Dynamic programming

- main idea
  - use value functions to structure the search for good policies
  - need a perfect model of the environment
- two main components
  - policy evaluation: compute V^{π} from π
  - policy improvement: improve π based on V^{π}

  - start with an arbitrary policy
  - repeat evaluation/improvement until convergence

Policy evaluation/improvement

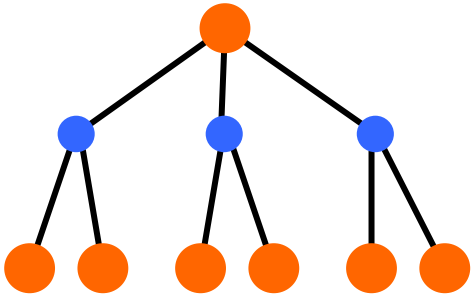
- policy evaluation: π -> V^π
  - Bellman eqn's define a system of n eqn's
  - could solve, but will use iterative version

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P_{ss'}^{a} [r_{ss'}^{a} + \gamma V_k(s')]$$

  - start with an arbitrary value function V_0, iterate until V_k converges

- policy improvement: V^π -> π'

$$\begin{aligned}\pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a \sum_{s'} P_{ss'}^{a} \left[r_{ss'}^{a} + \gamma V^\pi(s') \right]\end{aligned}$$



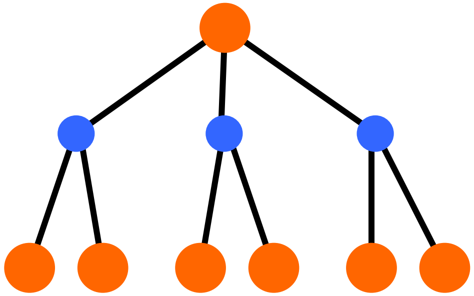
  - π' either strictly better than π, or π' is optimal (if π = π')

Policy/Value iteration

- Policy iteration

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

  - two nested iterations; too slow
  - don't need to converge to V^{π_k}
    - just move towards it



- Value iteration

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V_k(s') \right]$$

  - use Bellman optimality equation as an update
  - converges to V*

Using dynamic programming

- need complete model of the environment and rewards
    - robot in a room
        - state space, action space, transition model
- can we use DP to solve
    - robot in a room?
    - backgammon?
    - helicopter?
- DP bootstraps
    - updates estimates on the basis of other estimates

Monte Carlo methods

- don’t need full knowledge of environment
    - just experience, or
    - simulated experience

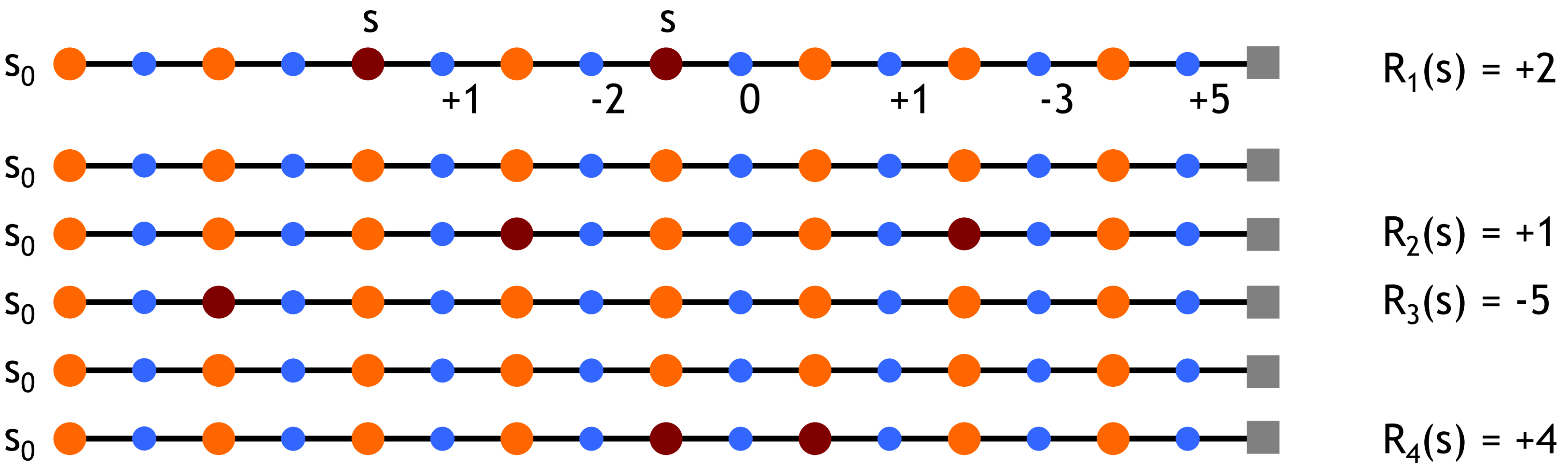
- averaging sample returns
    - defined only for episodic tasks

- but similar to DP
    - policy evaluation, policy improvement

Monte Carlo policy evaluation

- want to estimate $V^\pi(s)$
  = expected return starting from s and following π
  - estimate as average of observed returns in state s

- first-visit MC
  - average returns following the first visit to state s



$R_1(s) = +2$

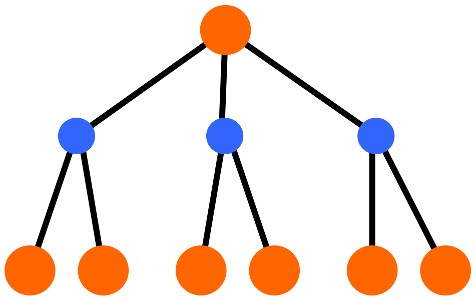
$R_2(s) = +1$

$R_3(s) = -5$

$R_4(s) = +4$

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

Monte Carlo control



- V^{π} not enough for policy improvement
  - need exact model of environment

$$\pi'(s) = \arg\max_a Q^{\pi}(s,a)$$

- estimate $Q^{\pi}(s,a)$

- MC control

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

  - update after each episode

- non-stationary environment

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

- a problem
  - greedy policy won't explore all actions

Maintaining exploration

- key ingredient of RL

- deterministic/greedy policy won’t explore all actions
  - don’t know anything about the environment at the beginning
  - need to try all actions to find the optimal one

- maintain exploration
  - use *soft* policies instead: $\pi(s,a)>0$ (for all s,a)

- ε-greedy policy
  - with probability 1-ε perform the optimal/greedy action
  - with probability ε perform a random action

  - will keep exploring the environment
  - slowly move it towards greedy policy: ε -> 0

Simulated experience

- 5-card draw poker
  - s_0: A♣, A♦, 6♠, A♥, 2♠
  - a_0: discard 6♠, 2♠
  - s_1: A♣, A♦, A♥, A♠, 9♠ + dealer takes 4 cards
  - return: +1 (probably)
- DP
  - list all states, actions, compute P(s,a,s')
    - P([A♣,A♦,6♠,A♥,2♠], [6♠,2♠], [A♠,9♠,4]) = 0.00192
- MC
  - all you need are sample episodes
  - let MC play against a random policy, or itself, or another algorithm

Summary of Monte Carlo

- don’t need model of environment
  - averaging of sample returns
  - only for episodic tasks

- learn from:
  - sample episodes
  - simulated experience

- can concentrate on “important” states
  - don’t need a full sweep

- no bootstrapping
  - less harmed by violation of Markov property

- need to maintain exploration
  - use soft policies

Value Iteration

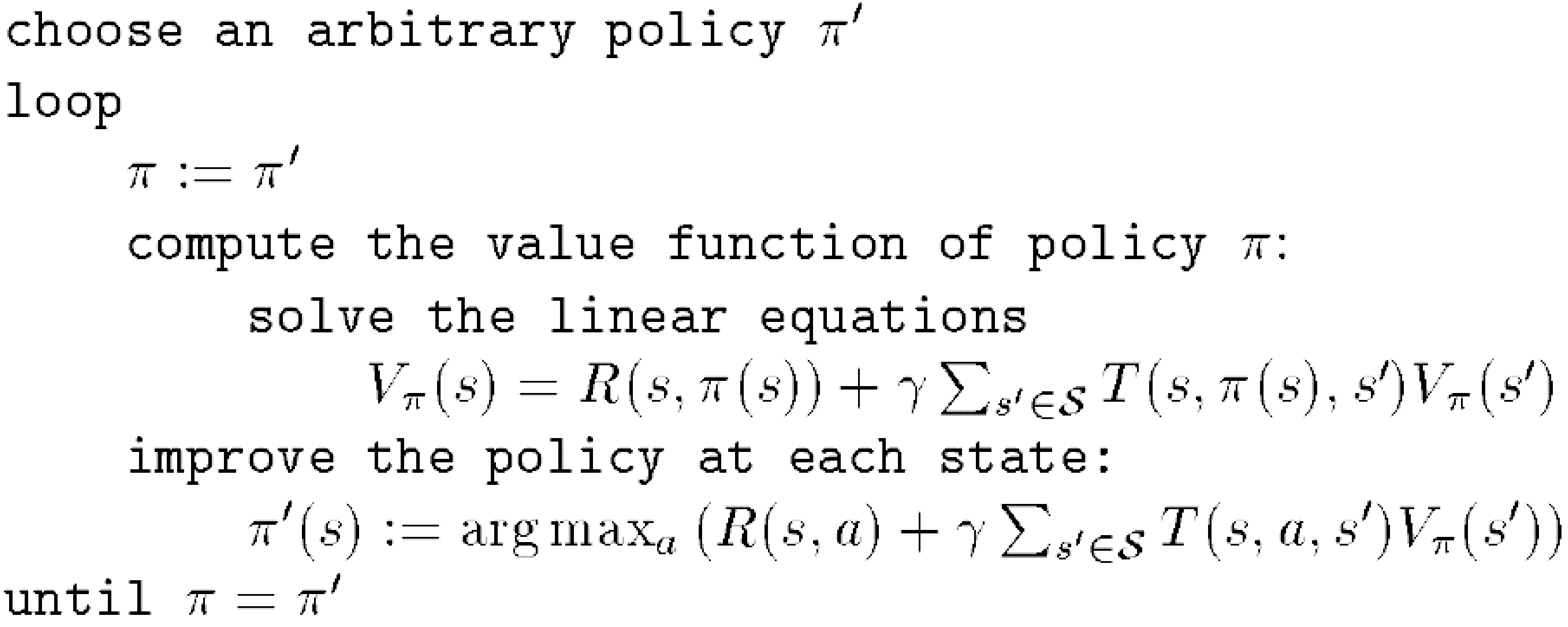
```
void valueIteration() {
  initialize $V(s)$ arbitrarily
  do{
    foreach ( $s \in S$ ) {
      foreach ( $a \in A$ ) {
          $Q(s,a) = R(s,a)+\gamma \Sigma_{s' \in S} T(s,a,s')V(s')$
           $\mathrm{V}(s) = \mathrm{max}_a Q(s,a)$
      }
    }
   } while ( ! policy good enough ) ;
}
```

Algorithm updates values backwards (from final states)

Value iteration: convergence

- Theorem: If the maximum difference between two successive value functions is less than ε, then the value of the greedy policy, (the policy obtained by choosing, in every state, the action that maximizes the estimated discounted reward, using the current estimate of the value function) differs from the value function of the optimal policy by no more than $2\varepsilon \gamma/(1-\gamma)$ at any state.
- An effective stopping criterion for the algorithm
- Value iteration is very flexible. The assignments to V need not be done in strict order but instead can occur asynchronously in parallel, provided that the value of every state gets updated infinitely often on an infinite run.

Policy iteration



```
choose an arbitrary policy $\pi'$
loop
    $\pi := \pi'$
    compute the value function of policy $\pi$:
        solve the linear equations
            $V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') V_\pi(s')$
    improve the policy at each state:
        $\pi'(s) := \arg\max_a \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') V_\pi(s')\right)$
until $\pi = \pi'$
```

The value function of a policy is just the expected infinite discounted reward that will be gained, at each state, by executing that policy. It can be determined by solving a set of linear equations. Once we know the value of each state under the current policy, we consider whether the value could be improved by changing the first action taken. If it can, we change the policy to take the new action whenever it is in that situation. This step is guaranteed to strictly improve the performance of the policy. When no improvements are possible, then the policy is guaranteed to be optimal.

Approximate solutions

- Learning with time differences (TD),
  a model is not needed, incremental, difficult for analysis
- Dynamic programming,
  mathematically well defined problems with exact and complete description of the environment
- Monte Carlo methods,
  model is not necessary, conceptually simple, not incremental, sampling complete trajectories in interaction with environment (or model of environment)
- Efficiency, convergence

TD(λ) learning

- Learning with time differences
- Previous states receive a portion of the difference to successors
- For λ=0

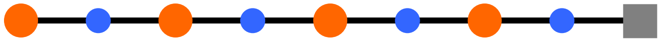
  $V(s_t) = V(s_t) + c(V(s_{t+1})- V(s_t))$
- c is a parameter, slowly decreasing during learning assuring convergence
- For $\lambda > 0$, more than just immediate successors are taken into account (speed)

Temporal Difference Learning

- combines ideas from MC and DP
  - like MC: learn directly from experience (don’t need a model)
  - like DP: bootstrap
  - works for continuous tasks, usually faster then MC

- constant-alpha MC:
  - have to wait until the end of episode to update

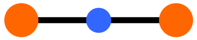
$$V(s_t) \leftarrow V(s_t) + \alpha \left[\underbrace{R_t} - V(s_t) \right]$$



target

- simplest TD
  - update after every step, based on the successor

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



MC vs. TD

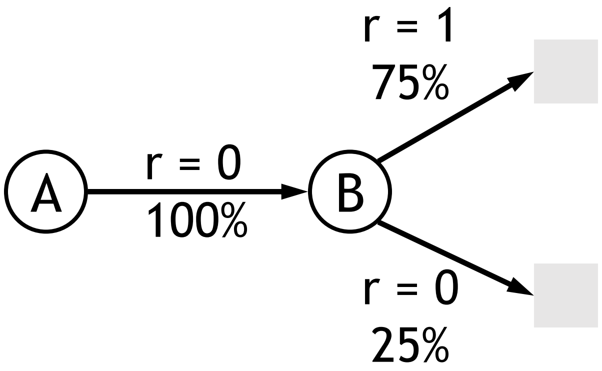
- observed the following 8 episodes:

  A – 0, B – 0   B – 1   B – 1   B - 1
  B – 1   B – 1   B – 1   B – 0

- MC and TD agree on V(B) = 3/4

- MC: V(A) = 0
  - converges to values that minimize the error on training data

- TD: V(A) = 3/4
  - converges to ML estimate of the Markov process



Q-learning

- previous algorithms: on-policy algorithms
  - start with a random policy, iteratively improve
  - converge to optimal

- Q-learning: off-policy
  - use any policy to estimate Q

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

  - Q directly approximates Q* (Bellman optimality eqn)
  - independent of the policy being followed
  - only requirement: keep updating each (s,a) pair

- Sarsa

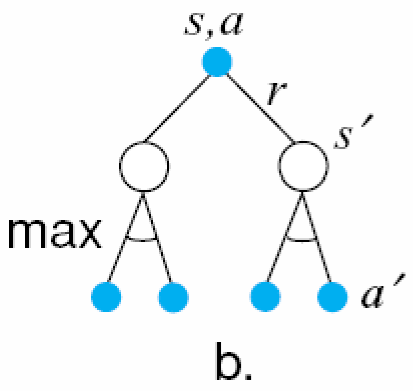
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Q learning

- Watkins, 1989
- The most popular variant of time difference learning
- One step ahead

$Q(s_t,a_t) = (1\text{-}c)\ Q(s_t,a_t) + c(r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t))$

$0 <= c, \gamma <= 1$



Q-Learning: Definitions

- Current state: s
- Current action: a
- Transition function: $\delta(s, a) = s'$

Markov property: this is independent of previous states given current state

- Reward function: $r(s, a) \in R$
- Policy $\pi(s) = a$

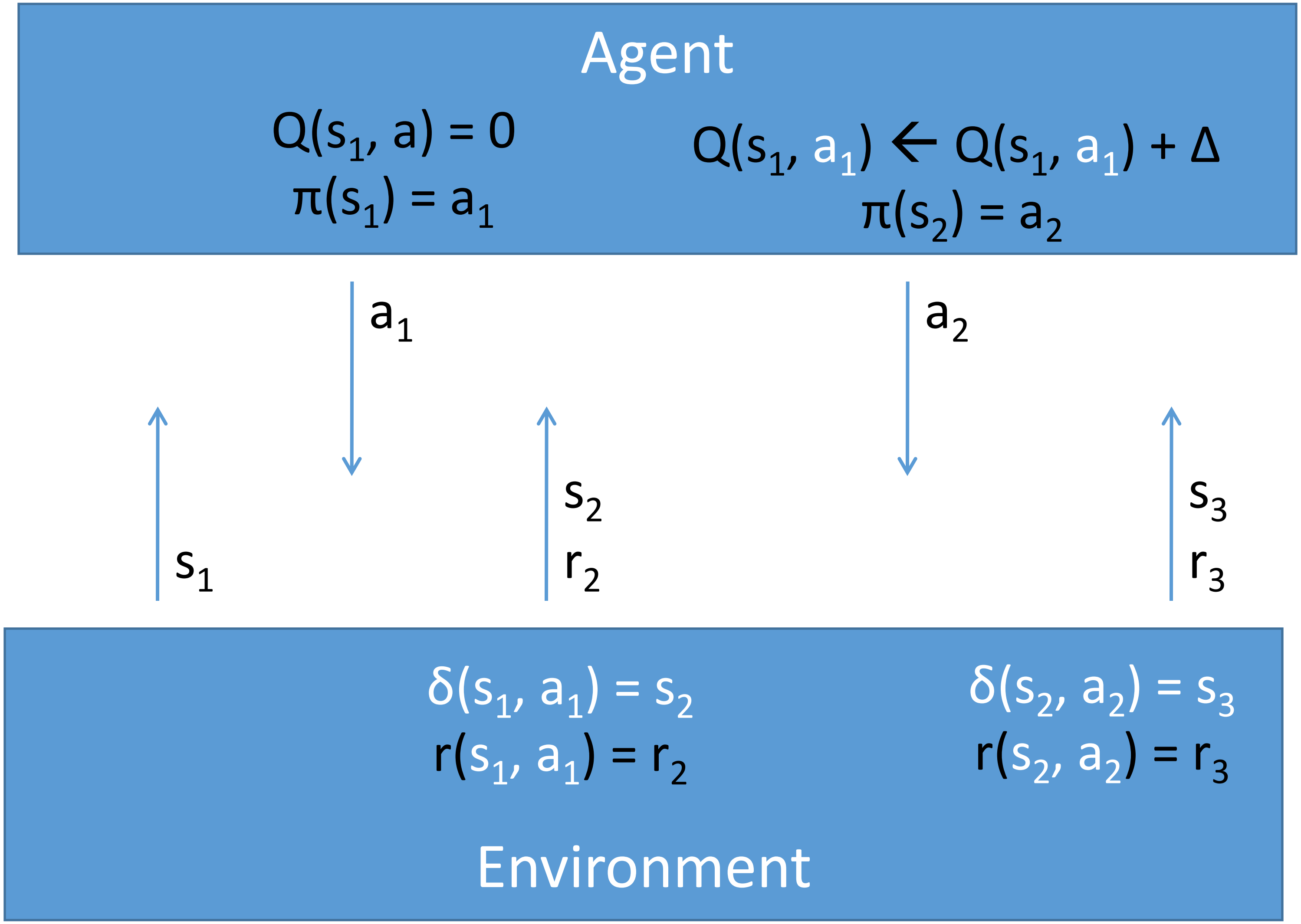
In classification we'd have examples $(s, \pi(s))$ to learn from

- $Q(s, a) \approx$ value of taking action a from state s

The Q-function

- ***Q(s, a)*** estimates the *discounted cumulative reward*
  - Starting in state ***s***
  - Taking action ***a***
  - Following the current policy thereafter
- Suppose we have the optimal Q-function
  - What's the optimal policy in state ***s***?
  - The action $\boldsymbol{argmax_b\ Q(s, b)}$
- But we don't have the optimal Q-function at first
  - Let's act as if we do
  - And updates it after each step so it's closer to optimal
  - Eventually it will be optimal!

Q-Learning: The Procedure



Agent

$Q(s_1, a) = 0$

$\pi(s_1) = a_1$

$Q(s_1, a_1) \leftarrow Q(s_1, a_1) + \Delta$

$\pi(s_2) = a_2$

s_1

a_1

s_2
r_2

a_2

s_3
r_3

$\delta(s_1, a_1) = s_2$

$r(s_1, a_1) = r_2$

$\delta(s_2, a_2) = s_3$

$r(s_2, a_2) = r_3$

Environment

Q-Learning: Updates

- The basic update equation

$$Q(s,a) \longleftarrow r(s,a) + \max_b Q(s',b)$$

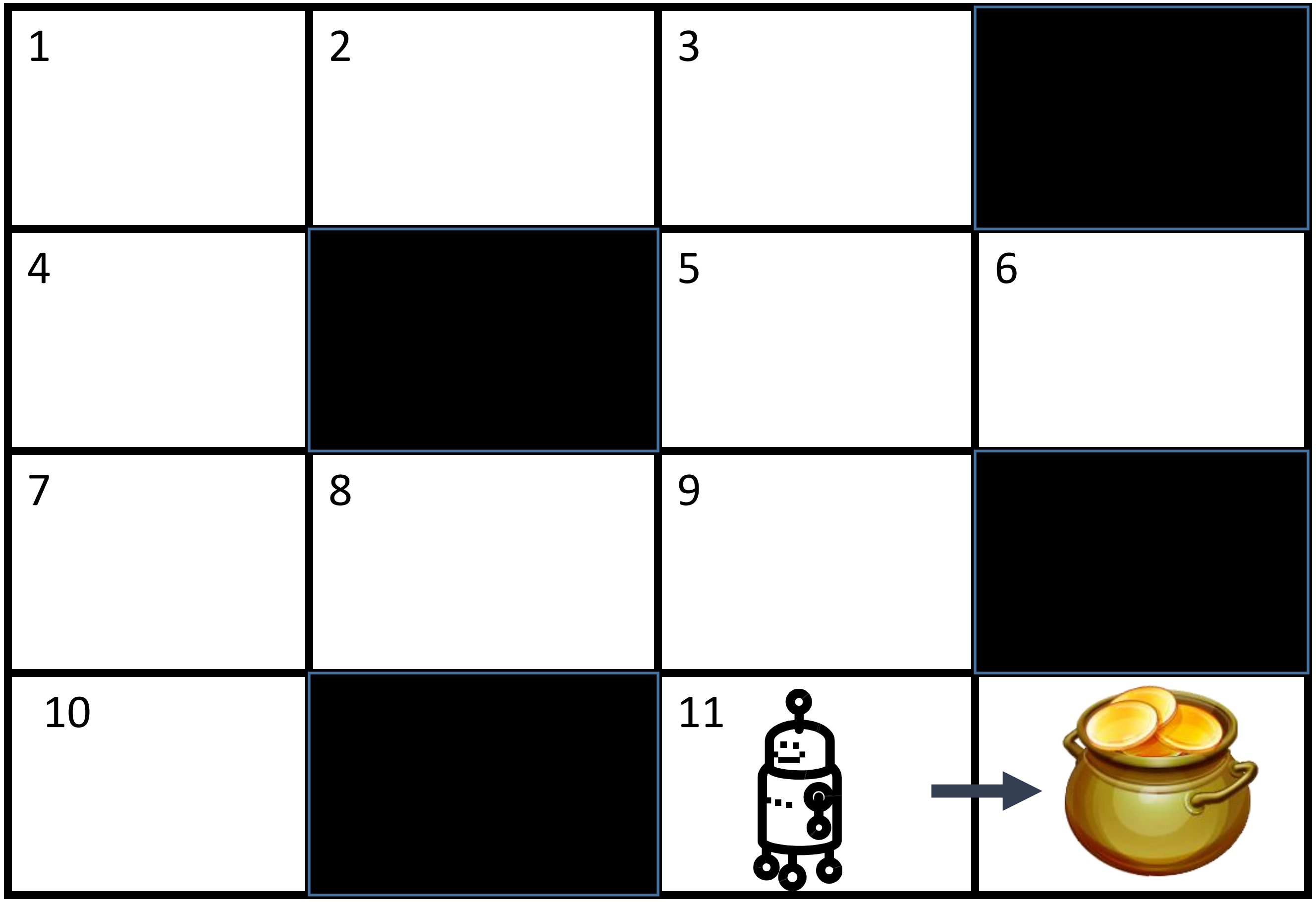
- With a discount factor to give later rewards less impact

$$Q(s,a) \longleftarrow r(s,a) + \gamma \max_b Q(s',b)$$

- With a learning rate for non-deterministic worlds

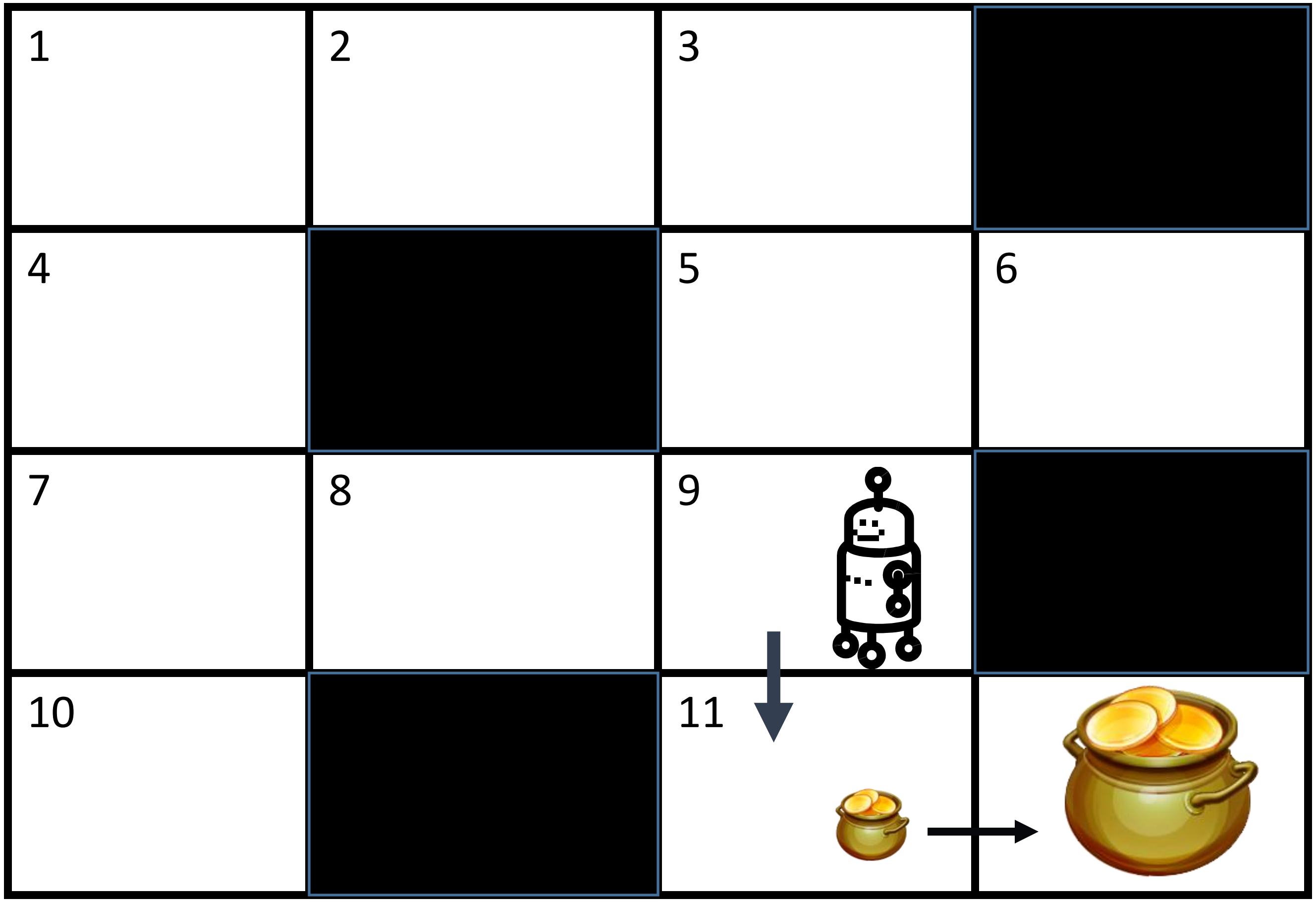
$$Q(s,a) \longleftarrow [1-\alpha]Q(s,a) + \alpha[r(s,a) + \gamma \max_b Q(s',b)]$$

Q-Learning: Update Example



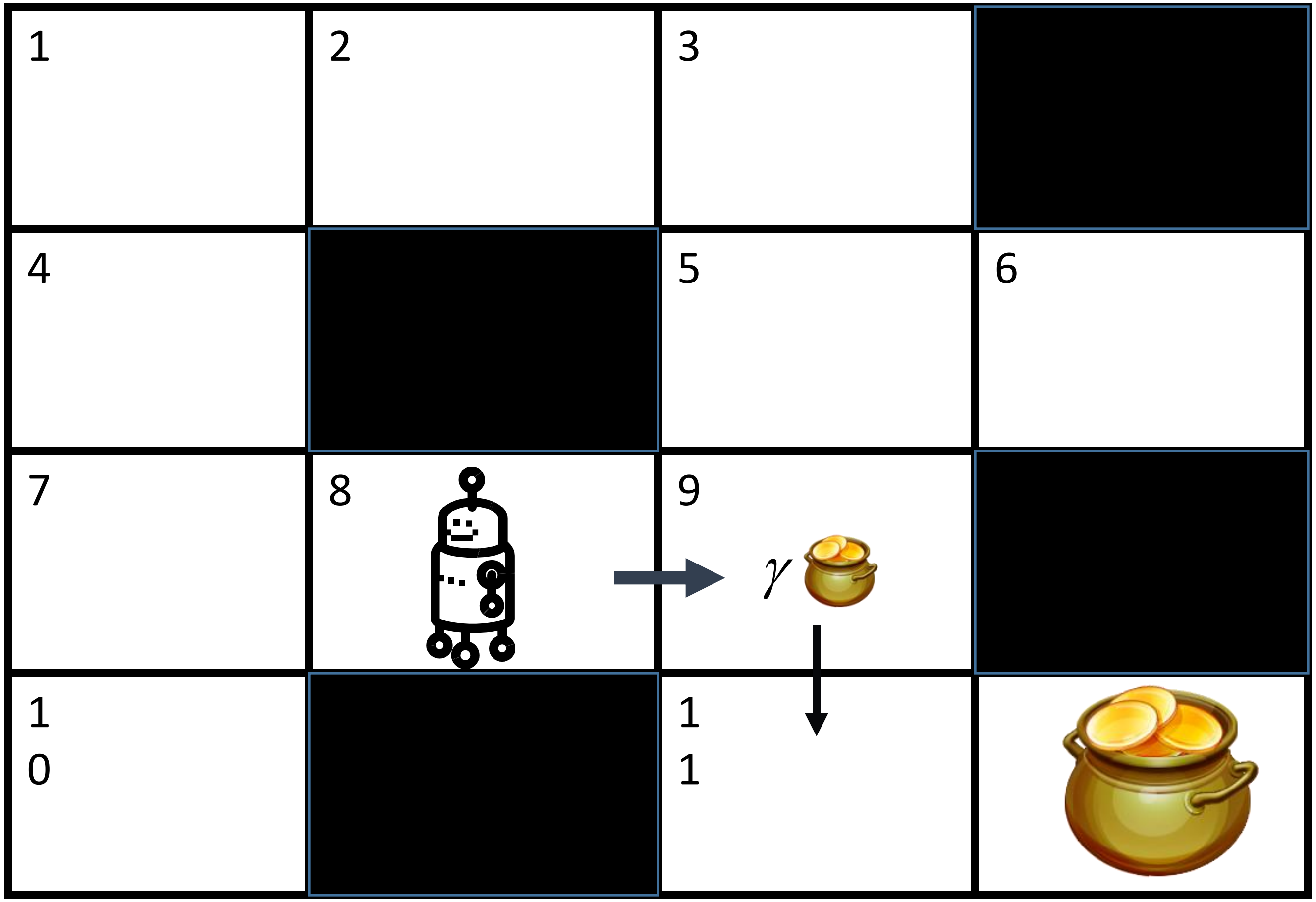
$Q(s_{11}, a_{\rightarrow}) =$

Q-Learning: Update Example



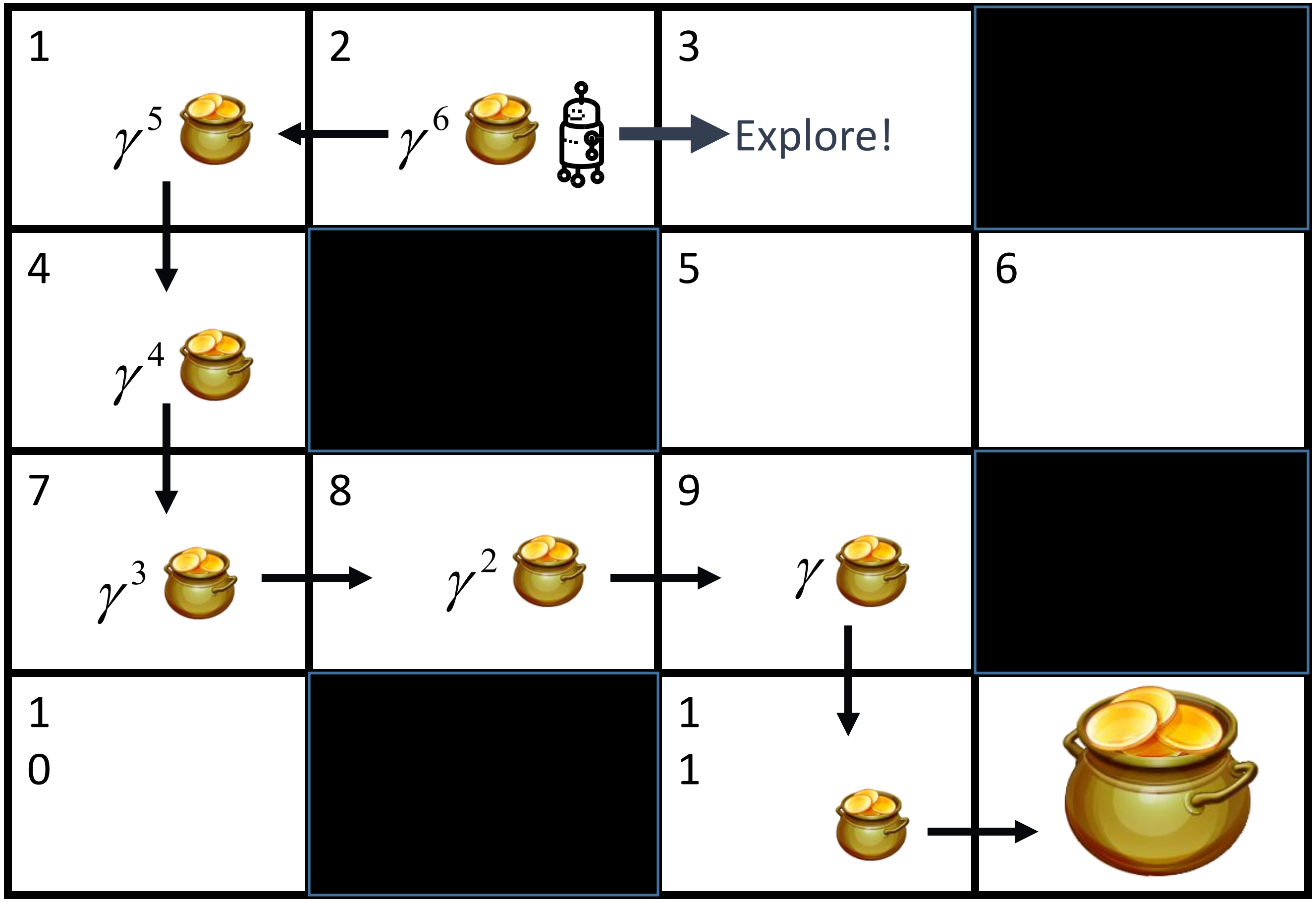
$$Q(s_9, a_\downarrow) = 0 + \gamma$$

Q-Learning: Update Example



$$Q(s_8, a_{\rightarrow}) = 0 + \gamma^2 💰$$

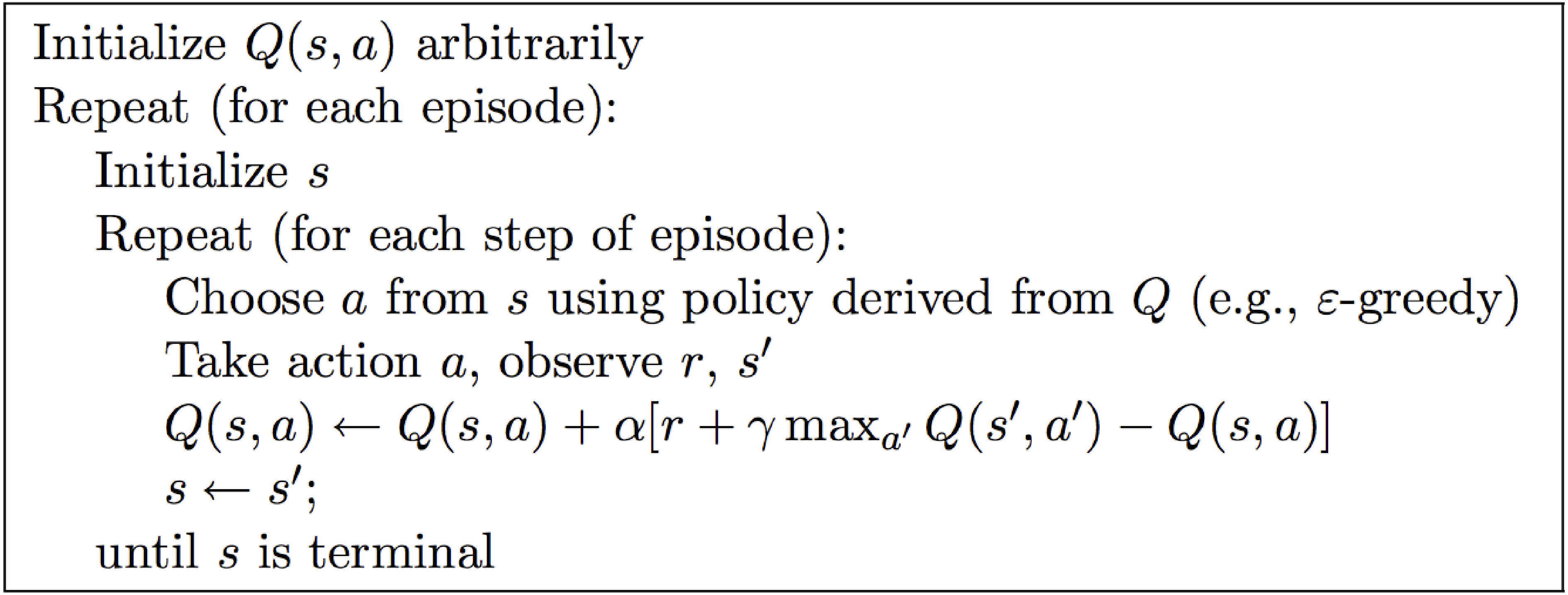
The Need for Exploration



$$\arg\max Q(s_2, a) = \leftarrow$$

$$best = \rightarrow$$

Q learning



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Explore/Exploit Tradeoff

- Can’t always choose the action with highest Q-value
  - The Q-function is initially unreliable
  - Need to explore until it is optimal

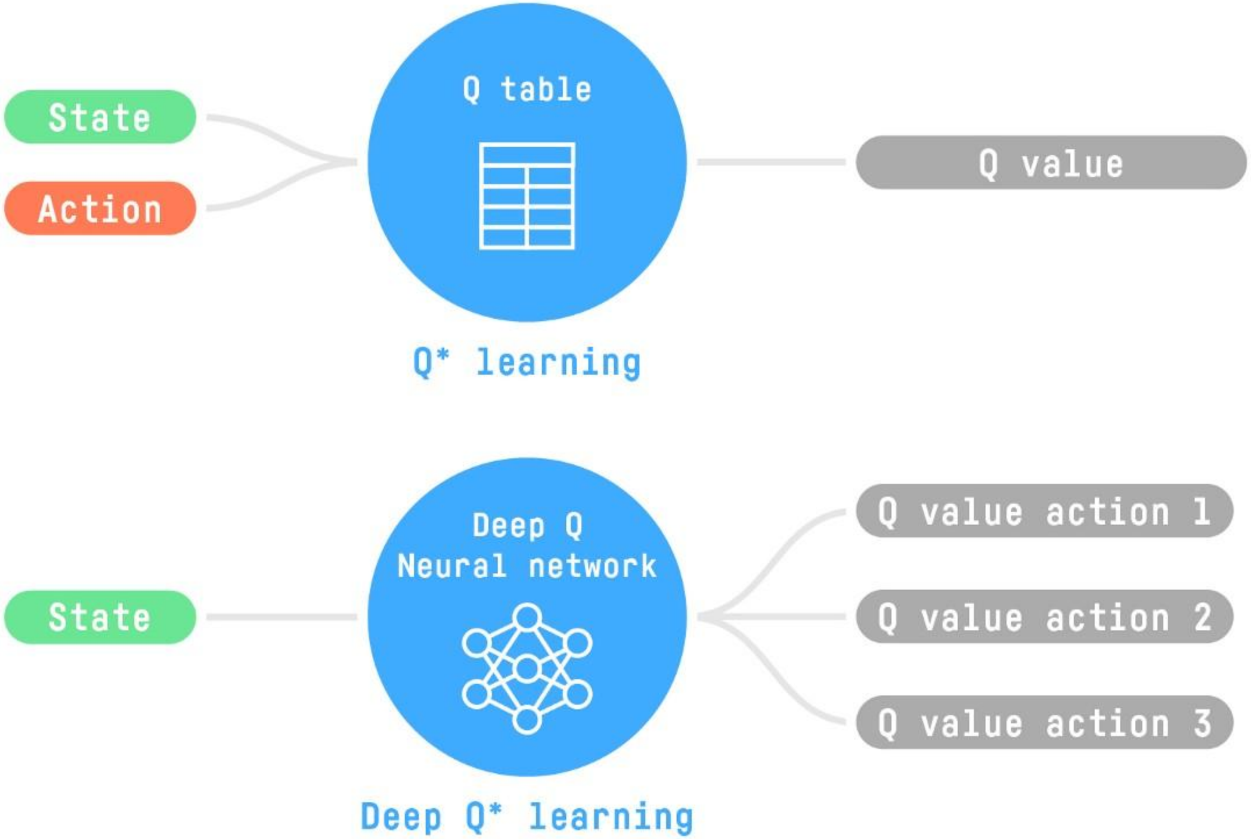
- Most common method: ε-greedy
  - Take a random action in a small fraction of steps (**ε**)
  - Decay **ε** over time

- There is some work on optimizing exploration
  - Kearns & Singh, ML 1998
  - But people usually use this simple method

Q-Learning: Convergence

- Under certain conditions, Q-learning will converge to the correct Q-function
  - The environment model doesn’t change
  - States and actions are finite
  - Rewards are bounded
  - Learning rate decays with visits to state-action pairs
  - Exploration method would guarantee infinite visits to every state-action pair over an infinite training period

Deep Q learning

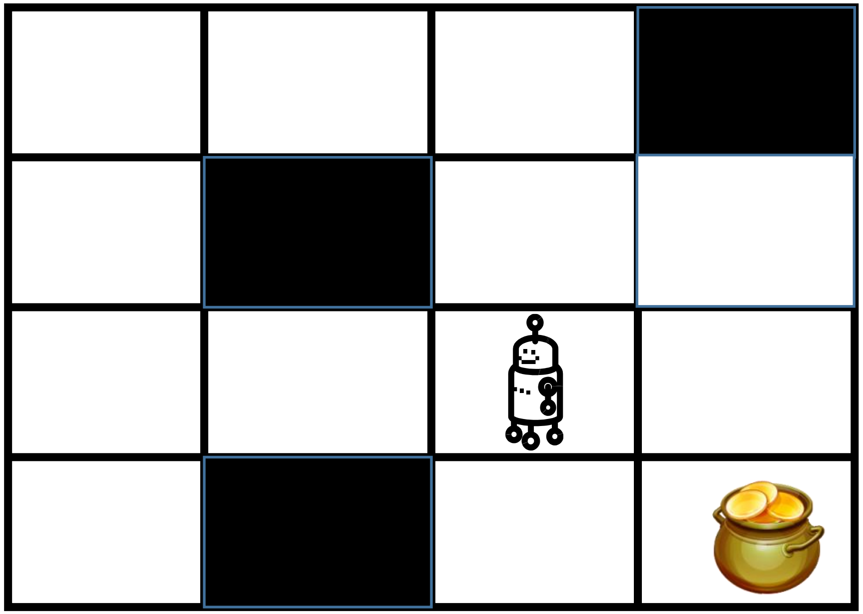


Extensions: SARSA

- SARSA: Take exploration into account in updates
  - Use the action actually chosen in updates

~~$$Q(s,a) \longleftarrow r(s,a) + \gamma \max_b Q(s',b)$$~~

$$Q(s,a) \longleftarrow r(s,a) + \gamma Q(s',a')$$

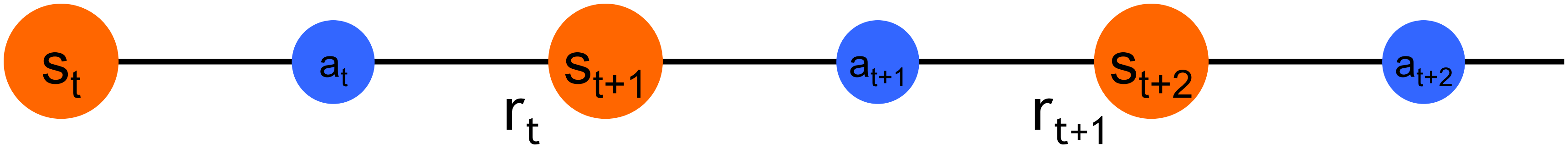


Regular: ↓=→

SARSA: ↓>→

Sarsa

- again, need Q(s,a), not just V(s)



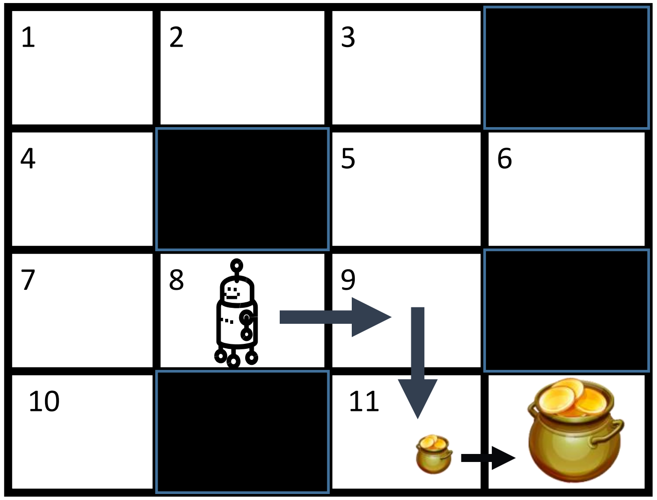
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

- control
  - start with a random policy
  - update Q and π after each step
  - again, need ε-soft policies

Extensions: Look-ahead

- Look-ahead: do updates over multiple states
  - Use some episodic memory to speed credit assignment

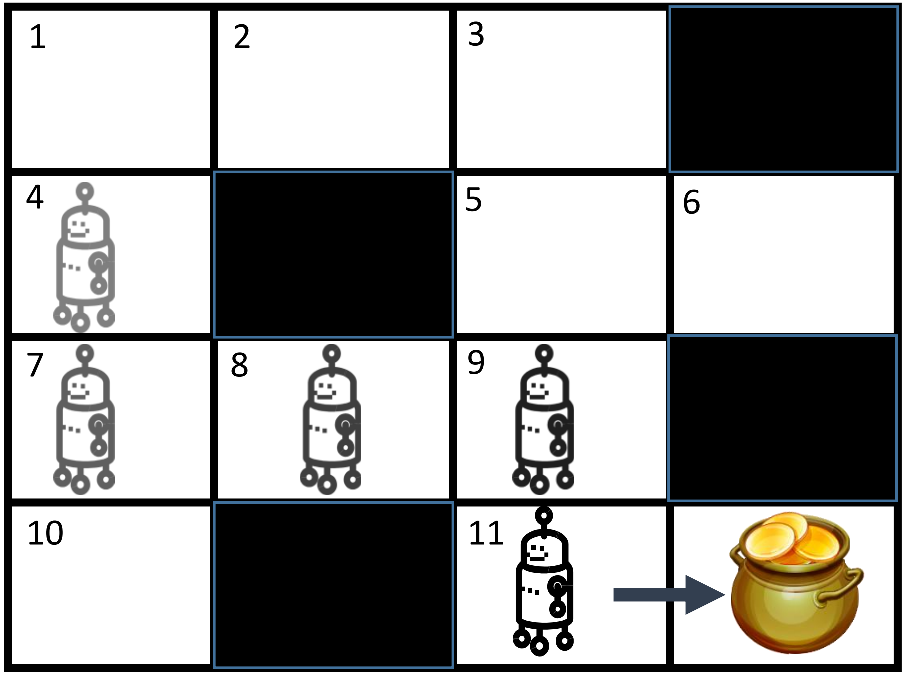
$$Q(s,a) \longleftarrow r(s,a) + \gamma r(s',a') + \gamma^2 Q(s'',a'')$$



- TD(**λ**): a weighted combination of look-ahead distances
  - The parameter **λ** controls the weighting

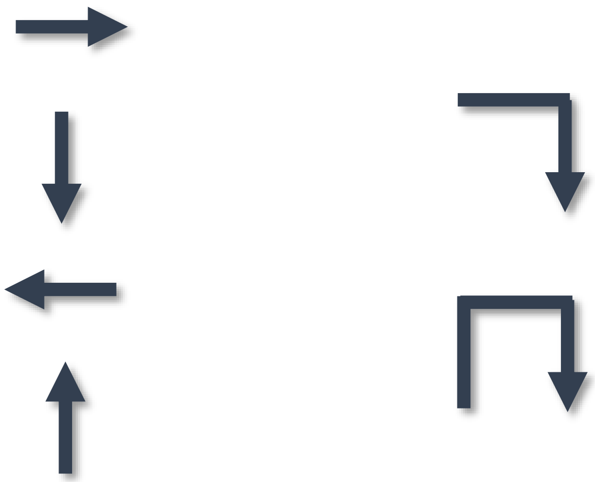
Extensions: Eligibility Traces

- Eligibility traces: Lookahead with less memory
  - Visiting a state leaves a trace that decays
  - Update multiple states at once
  - States get credit according to their trace

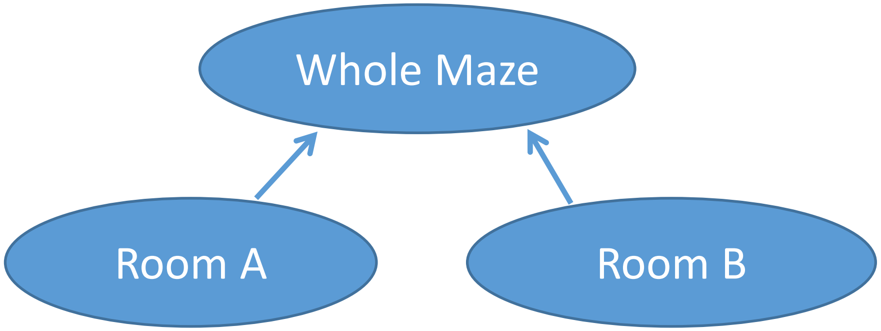


Extensions: Options and Hierarchies

- Options: Create higher-level actions



- Hierarchical RL: Design a tree of RL tasks



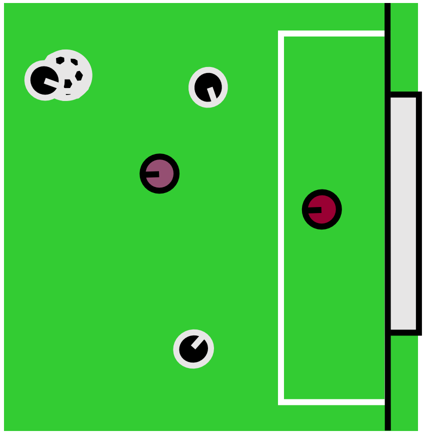
Whole Maze

Room A

Room B

Extensions: Function Approximation

- Function approximation: allow complex environments
  - The Q-function table could be too big (or infinitely big!)
  - Describe a state by a feature vector
    $\boldsymbol{f = (f_1, f_2, \ldots, f_n)}$
    - Then the Q-function can be any regression model
    - E.g. linear regression:
      $\boldsymbol{Q(s, a) = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n}$



  - Cost: convergence goes away in theory, though often not in practice
  - Benefit: generalization over similar states
  - Easiest if the approximator can be updated incrementally, like neural networks with gradient descent, but you can also do this in batches

Measuring learning performance

- Eventual convergence to optimality
  Many algorithms come with a provable guarantee of asymptotic convergence to optimal behavior. This is reassuring, but useless in practical terms.
- Speed of convergence to optimality
  Optimality is usually an asymptotic result, and so convergence speed is an ill-defined measure. More practical are
  - speed of convergence to near-optimality (how near?)
  - level of performance after a given time (what time?)
- Regret
  expected decrease in reward gained due to executing the learning algorithm instead of behaving optimally from the very beginning; these results are hard to obtain.

Challenges in Reinforcement Learning

- Feature/reward design can be very involved
  - Online learning (no time for tuning)
  - Continuous features (handled by *tiling*)
  - Delayed rewards (handled by *shaping*)
- Parameters can have large effects on learning speed
  - Tuning has just one effect: slowing it down
- Realistic environments can have partial observability
- Realistic environments can be non-stationary
- There may be multiple agents

Do Brains Perform RL?

- Should machine learning researchers care?
  - Planes don’t fly the way birds do; should machines learn the way people do?
  - But why not look for inspiration?

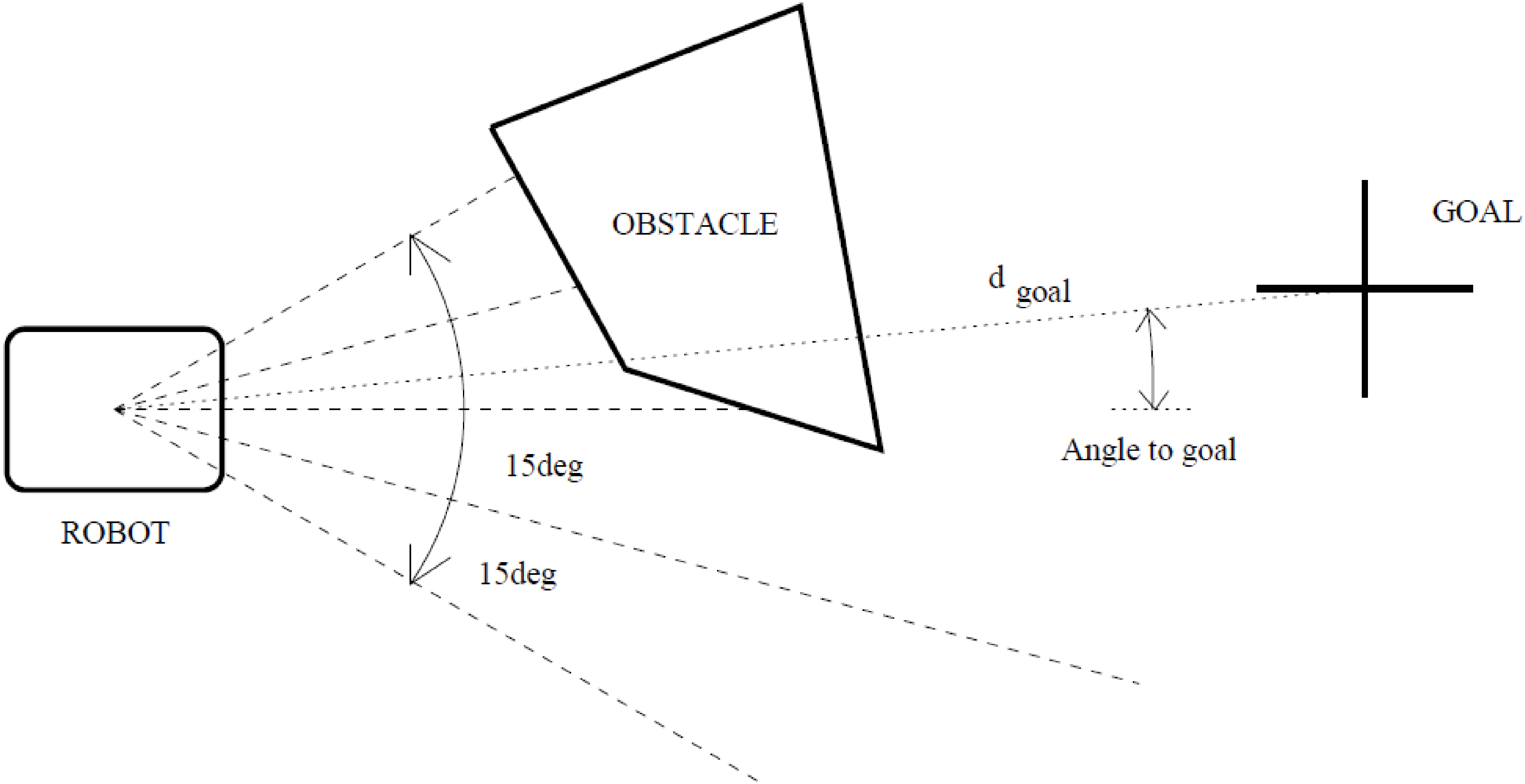
- Psychological research does show neuron activity associated with rewards
  - Really prediction error: actual – expected
  - Primarily in the striatum

What People Do Better

- Parallelism
  - Separate systems for positive/negative errors
  - Multiple algorithms running simultaneously
- Use of RL in combination with other systems
  - Planning: Reasoning about why things do or don't work
  - Advice: Someone to imitate or correct us
  - Transfer: Knowledge about similar tasks
- More impulsivity
  - Is this necessarily better?
- The goal for machine learning: Take inspiration from humans without being limited by their shortcomings

Some examples and details

An example: directing robot in 2d plane



- G.A.Rummery: Problem Solving with Reinforcement Learning, 1995

Robot in 2d: the settings

- sensors:
  - five distance measures to nearest obstacle in 15 degree forward arc
  - always knows distance and angle to the goal
- payoff after the end of the trial (reaching goal, collision with an obstacle or time out)
- start, goal and obstacles are randomly changed after every trial
- robot has to learn a generalized reactive policy; how?

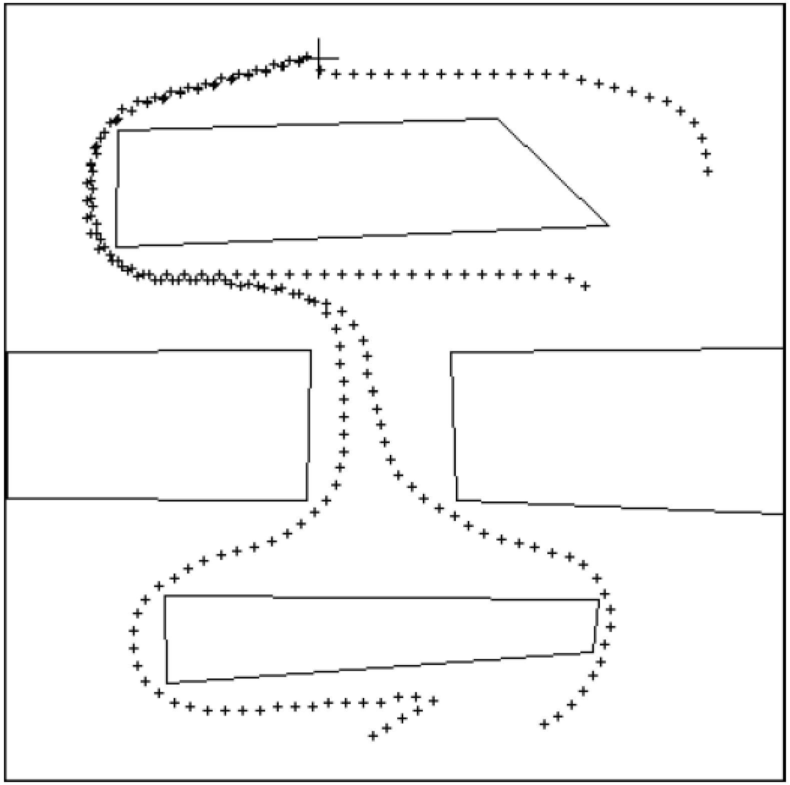
Robot in 2d: actions and rewards

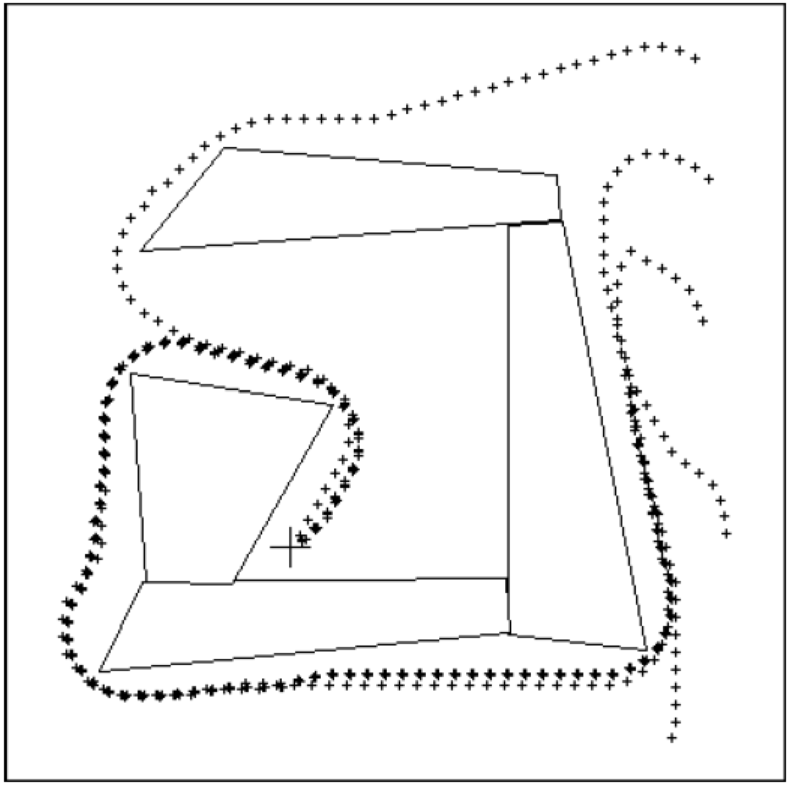
- 6 actions:
  - (turn left 15º, turn right 15º, stay in the same direction) x (move forward for a fixed distance d, do not move)
- rewards:
  - 0 in every step except the final
  - goal: if in a small fixed radius around the goal, +1
  - crash: based on a distance d from the goal e.g. $0.5 \exp(-\frac{2d_{goal}}{d_{max}})$, (note: maximum is 0.5)
  - time-out: as for crash + some small reward for not crashing, e.g., +0.3
- set γ = 0.99 to reward faster findings of the goal
- set probability of exploration/exploitation

Using NN for 2-d robot

- coarse coding the inputs (e.g., with several input sigmoids for each sensor)
- backpropagation with momentum term or eligibility traces
- batch and on-line training

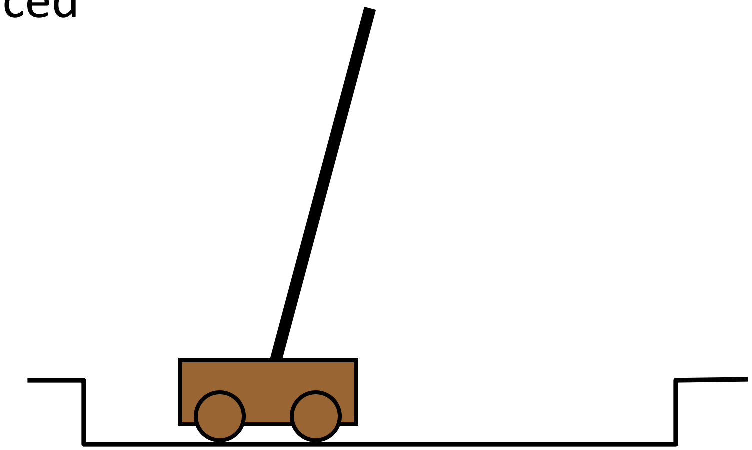
Some trajectories of trained robots





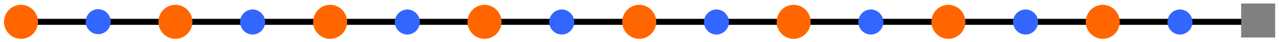
State representation

- pole-balancing
  - move car left/right to keep the pole balanced
- state representation
  - position and velocity of car
  - angle and angular velocity of pole
- what about *Markov property*?
  - would need more info
  - noise in sensors, temperature, bending of pole
- solution
  - coarse discretization of 4 state variables
    - left, center, right
  - totally non-Markov, but still works



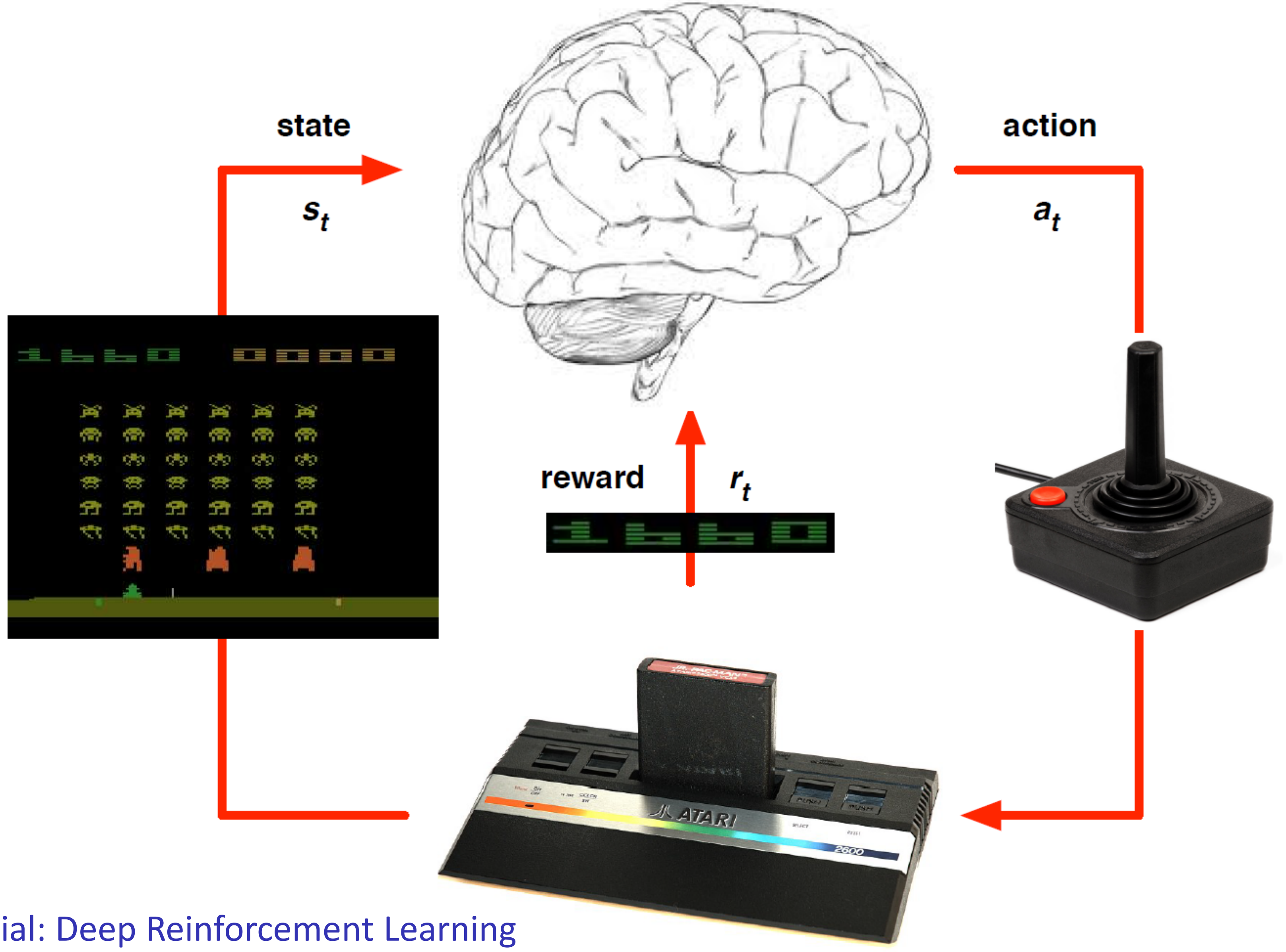
Designing rewards

- robot in a maze
  - episodic task, not discounted, +1 when out, 0 for each step
- chess
  - GOOD: +1 for winning, -1 losing
  - BAD: +0.25 for taking opponent's pieces
    - high reward even when lose
- rewards
  - rewards indicate what we want to accomplish
  - NOT how we want to accomplish it
- shaping



  - positive reward often very "far away"
  - rewards for achieving subgoals (domain knowledge)
  - also: adjust initial policy or initial value function

Reinforcement Learning in Atari



state

s_t

action

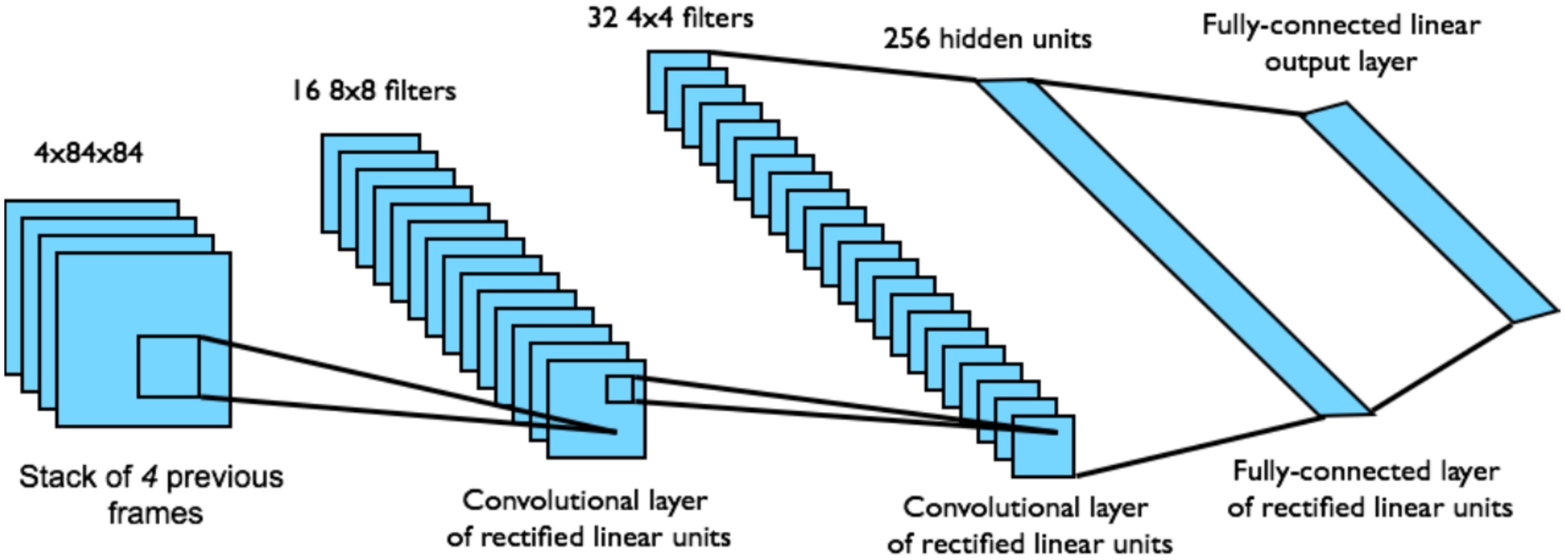
a_t

reward

r_t

DQN in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step


32 4x4 filters
256 hidden units
Fully-connected linear output layer
16 8x8 filters
4x84x84
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Network architecture and hyperparameters fixed across all games

[Mnih et al.]

RL with human feedback (RLHF)

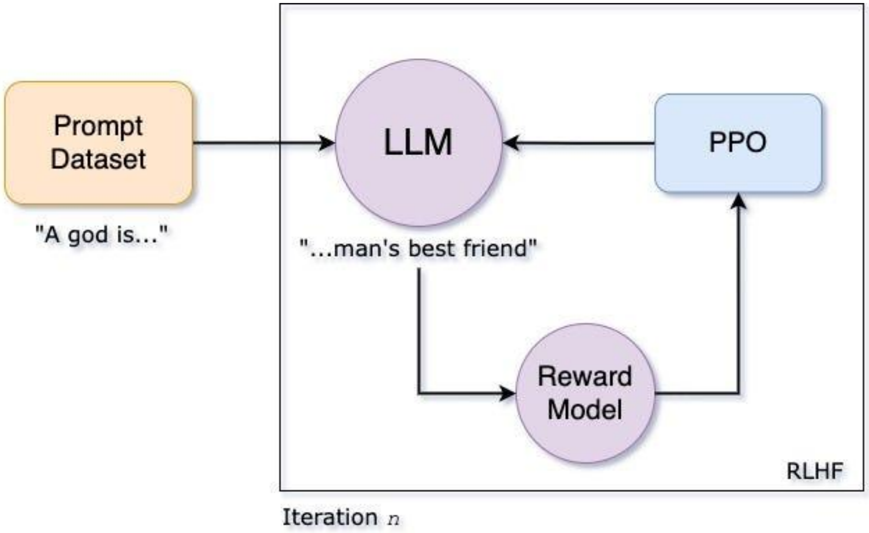
- RLHF for LLM preference alignment
- **RLHF** is the classic pipeline used to **align LLMs** with human preferences such as helpfulness, harmlessness, and honesty. It combines **supervised learning**, **human preference modeling**, and **RL**
- RLHF teaches an LLM what humans prefer by first learning a reward model from human judgments, then optimizing the LLM to maximize that reward while staying close to its original behavior.
- RLHF does **not** align truth or correctness directly. It aligns the model to **human preferences**, such as helpfulness, politeness, refusal behavior, non-toxicity, style and tone
- This can lead to known issues, such as reward hacking, over-verbosity, preference bias, and misalignment with ground truth

Three stages of RLHF

1. Supervised Fine-Tuning of the initial LLM to produce reasonable answers

2. Reward model on e.g., pairs of human responses (x,y+,y−) i.e., which score is better

3. RL to adjust LLM weights, typically using PPO (proximal policy optimization)



Proximal policy optimization (PPO)

In LLM alignment, PPO objective (simplified) is set as:

$$\max_{\theta} \quad \mathbb{E}_{y\sim\pi_\theta}[r_\phi(x,y) - \beta\,\mathrm{KL}\left(\pi_\theta \parallel \pi_{ref}\right)]$$

- Where:
- π_{ref} =frozen SFT model
- KL penalty prevents the model from drifting too far
- PPO ensures stable, clipped updates

Direct Preference Optimization (DPO)

- A simple and stable algorithm for aligning large language models (LLMs) using preference data — without reinforcement learning.
- DPO removes the reward model and RL
- Instead of learning a reward function $r(x, y)$, DPO directly optimizes the policy so that:

$$\pi_\theta(y^+ \mid x) \;>\; \pi_\theta(y^- \mid x)$$

- but relative to a reference model π_{ref} i.e. usually the pretrained SFT model).

Training Procedure for DPO

- 1. Start from pretrained/SFT model
- 2. Clone reference model (frozen)
- 3. Compute log-probs for y+ and y-
- 4. Apply DPO loss
- 5. Standard backpropagation

- No rollouts, no rewards, no RL.

Summary of RL

- Reinforcement learning
  - use when need to make decisions in uncertain environment
  - actions have delayed effect

- solution methods
  - dynamic programming
    - need complete model

  - Monte Carlo
  - time difference learning (Sarsa, Q-learning)

- simple algorithms
- most work
  - designing features, state representation, rewards