Reinforcement Learning

mallozzi@chalmers.se

Piergiuseppe Mallozzi

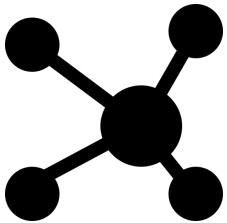
Types of Machine Learning

SUPERVISED

Task Driven
predict next value, classification

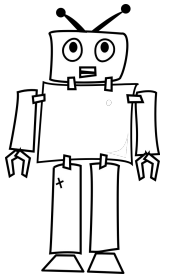
UNSUPERVISED

Data Driven
Identify clusters



REINFORCEMENT

Learn From Mistakes
Trial and Error



(Re)sources

These slides are inspired/taken *mainly* from

- **Dan Klein**, **Pieter Abbeel**, UC Berkeley CS188 Intro to AI:
  https://people.eecs.berkeley.edu/~russell/classes/cs188/f14/lecture_videos.html
  (Lecture MPD I, II and Reinforcement Learning I, II)
- **Joelle Pineau**, Reinforcement Learning Lecture
  http://videolectures.net/deeplearning2017_pineau_reinforcement_learning/
  David Silver, RL Course at UCL
  http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
- **Fei-Fei Li & Justin Johnson & Serena Yeung**, Reinforcement Learning, Stanford
  http://vision.stanford.edu/teaching/cs231n/slides/2019/cs231n_2019_lecture14.pdf
- **Barto** and **Sutton**, Reinforcement Learning: An Introduction
  http://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf

So far… Supervised Learning

Data:
(x, y) x is the *data*, y is the *label*

Goal:
learn a function to map x ⟶ y

Examples:
Regression, classification, semantic segmentation, image captioning etc..

Cat

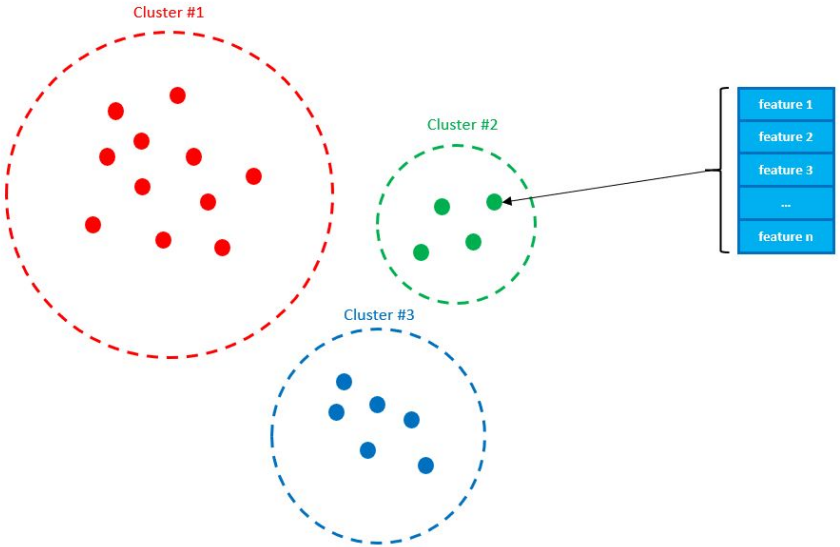
Classification

So far… Unsupervised Learning

Data:
Just *x*, the *data*, there are no labels!

Goal:
Learn some underlying hidden structure of the data

Examples:
Clustering, dimensionality reduction, feature learning, density estimation etc..

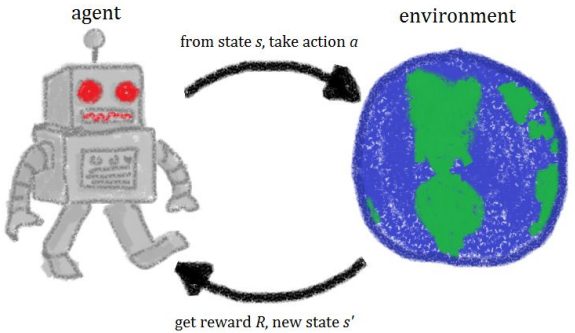


Today… Reinforcement Learning

Data:
Environment Observations and reward signal

Goal:
Learn how to take actions in order to maximize a reward

Examples:
Robotics, stock market, playing games

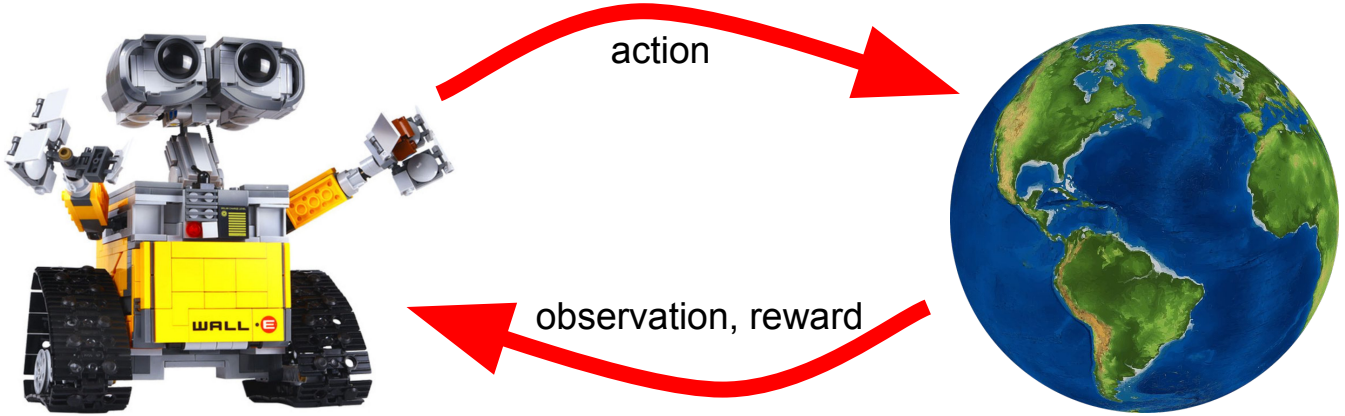


What is Reinforcement Learning?

Learning what to do

Reinforcement learning is learning how to map situations to action so as to maximize a numerical reward signal

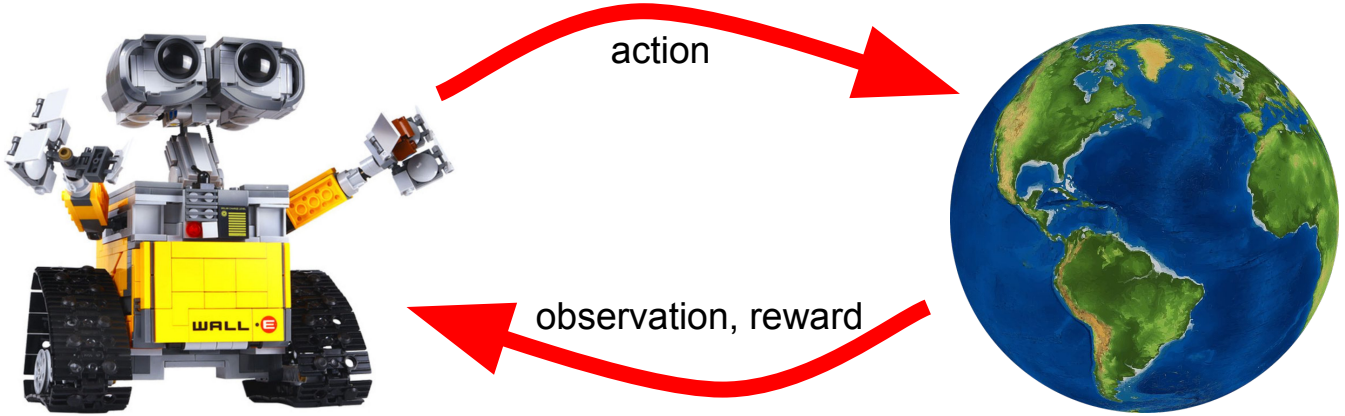
- Improves with experience
- Inspired by psychology



What is Reinforcement Learning?

Learning by trial-and-error

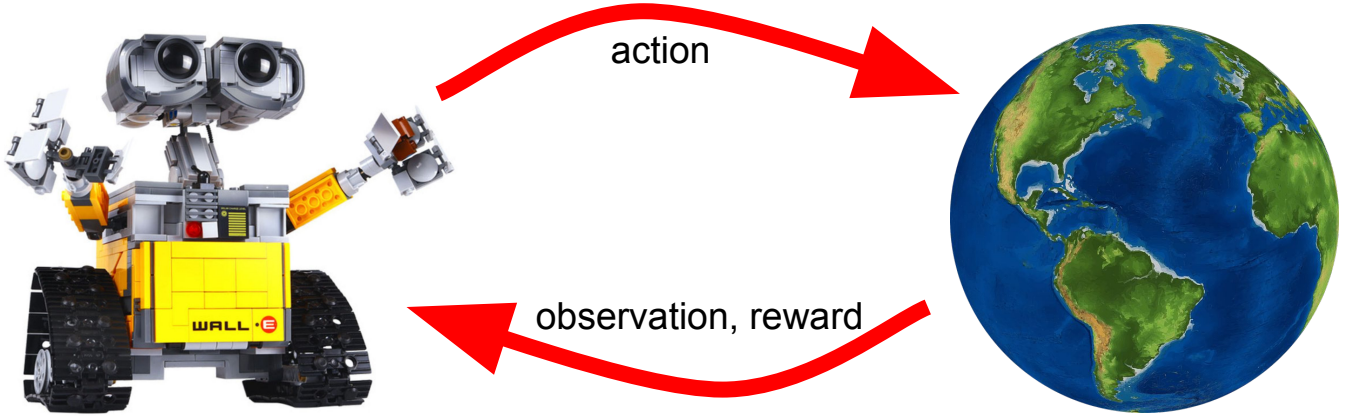
The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.



What is Reinforcement Learning?

Delayed Reward

Actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards.

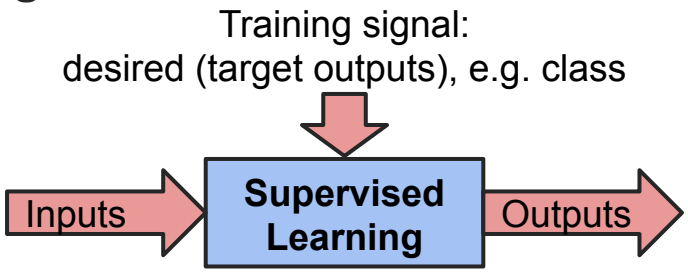


What is Reinforcement Learning?

Reinforcement Learning vs **Supervised Learning**

In Supervised Learning:

- Learning from a training set of labeled examples provided by a knowledgeable external supervisor
- Objective: extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set
- Alone it is not adequate for learning from interaction

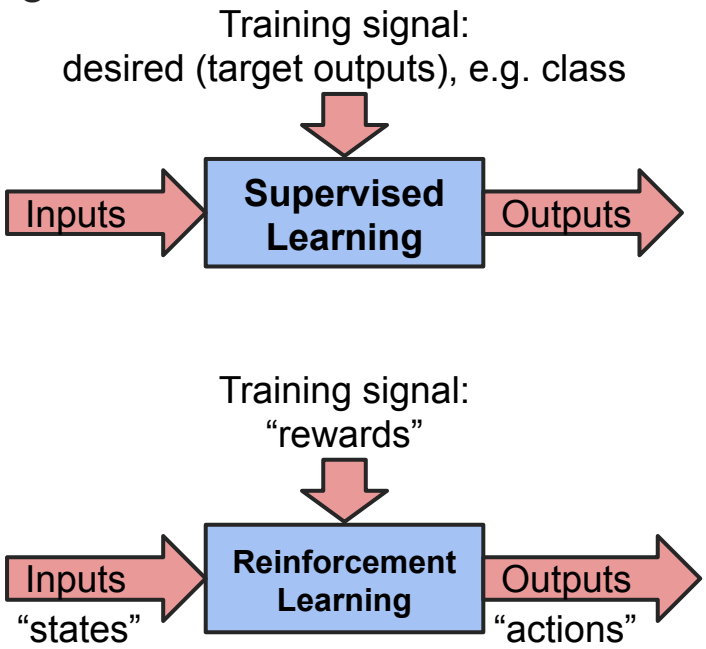


What is Reinforcement Learning?

Reinforcement Learning vs Supervised Learning

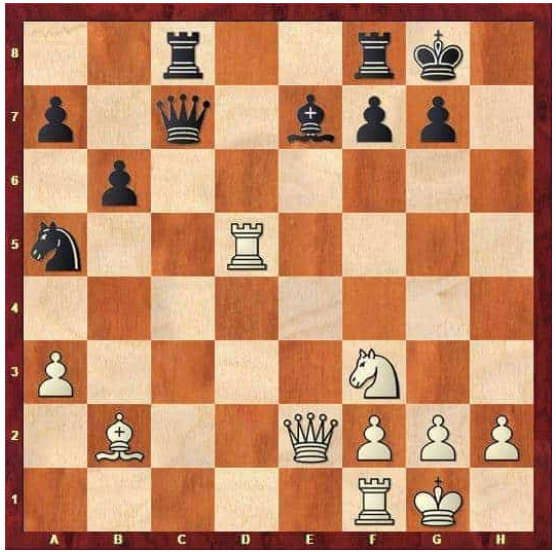
In Reinforcement Learning:

- There is no supervisor, only a *reward signal*
- Feedback is *delayed*, not instantaneous
- Time really matters (sequential, non i.i.d data)
  - What the agent sees at time *t* is going to be very correlated to what it sees at time *t+1*
- Agent's actions affect the subsequent data it receives
  - The agent is influencing the data it receives



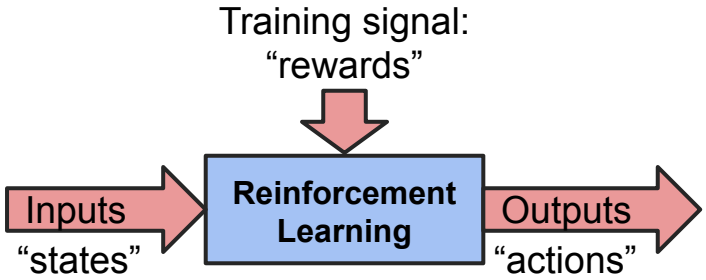
What is Reinforcement Learning?

Reinforcement Learning to approximate a **sequential decision task**



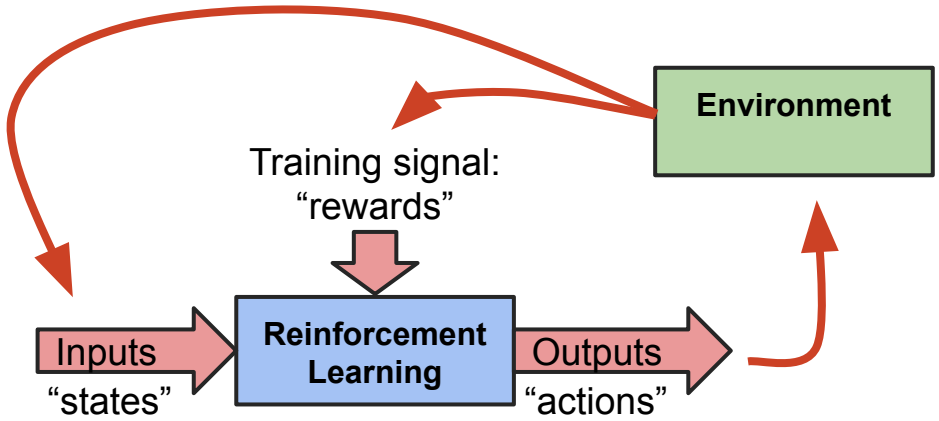
Rewards: +1 if win, -1 if loose

The computer makes a long sequence of moves before it finds out whether it wins or loses the game.



Reinforcement Learning Framework

Practical & Technical Challenges



1. Need access to the environment.
2. Jointly learning *AND* planning from correlated Samples.
3. Data distribution changes with action choice.

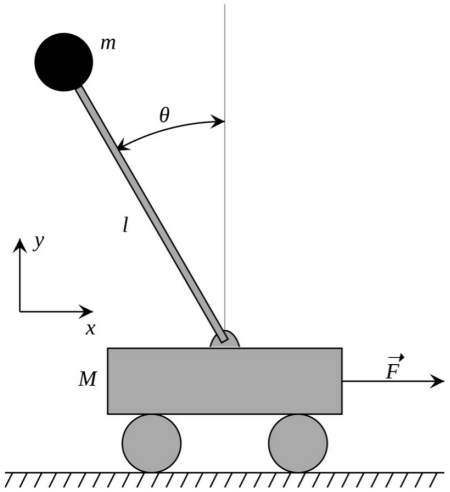
When to use Reinforcement Learning

- Data in the form of trajectories.
- Need to make a sequence of (related) decisions.
- Observe (partial, noisy) feedback to choice of actions.
- Tasks that require both *learning* and *planning*.

Reinforcement Learning Examples

Cart-Pole Problem

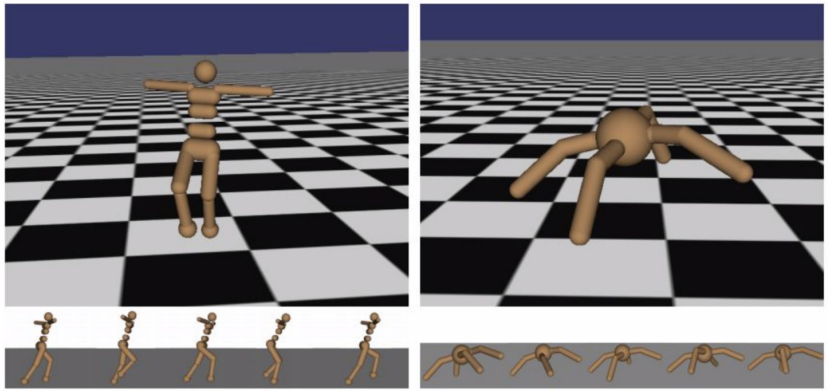
- **Objective**: Balance a pole on top of a movable cart
- **State**: angle, angular speed, position, horizontal velocity
- **Action**: horizontal force applied on the cart
- **Reward**: 1 at each time step if the pole is upright



Reinforcement Learning Examples

Robot Locomotion

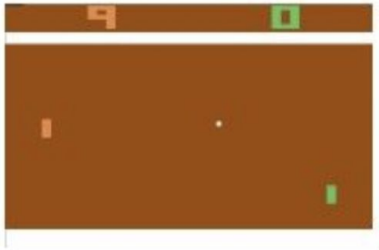
- **Objective**: Make the robot move forward
- **State**: Angle and position of the joints
- **Action**: Torques applied on joints
- **Reward**: 1 at each time step upright + forward movement



Reinforcement Learning Examples

Atari Games

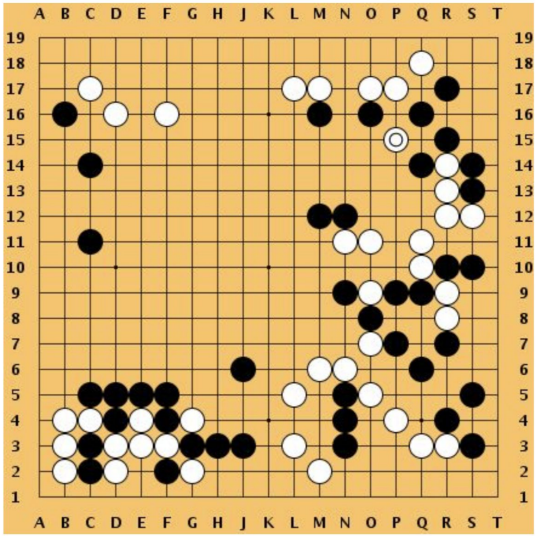
- **Objective**: Complete the game with the highest score
- **State**: Raw pixel inputs of the game state
- **Action**: Game controls e.g. Left, Right, Up, Down
- **Reward**: Score increase/decrease at each time step



Reinforcement Learning Examples

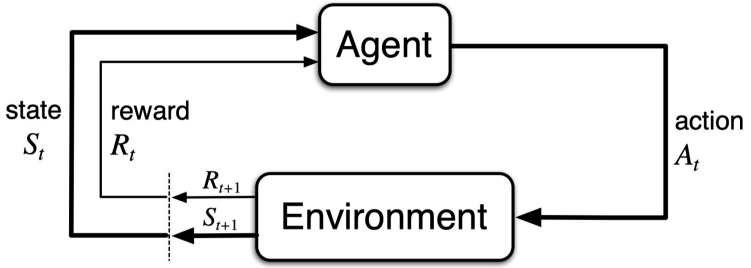
Go

- **Objective**: Win the game!
- **State**: Position of all pieces
- **Action**: Where to put the next piece down
- **Reward**: 1 if win at the end of the game, 0 otherwise



Markov Decision Process

Mathematical formulation of the RL problem



A **Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

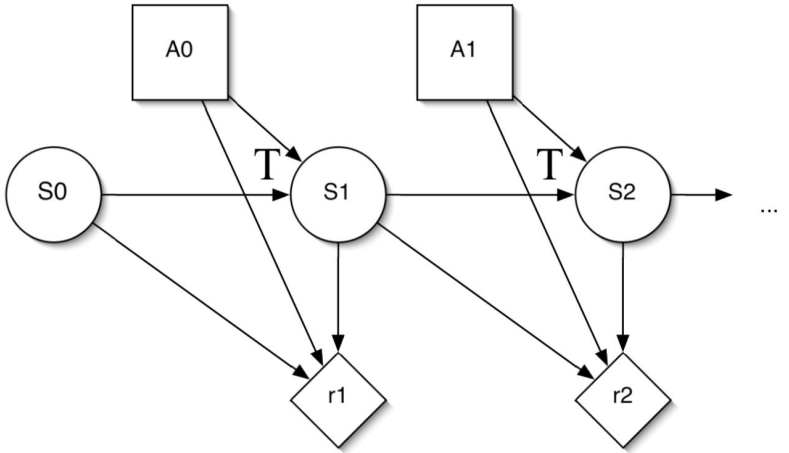
- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$

- R is a reward function $R(s,a,s')$ or $R(s)$ or $R(s')$

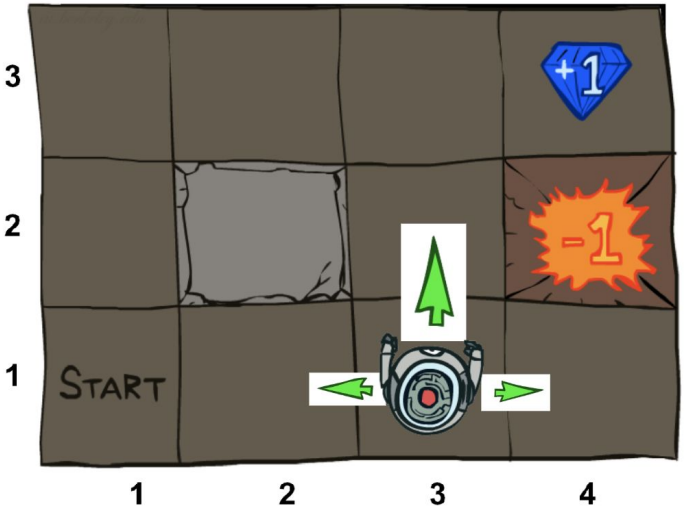
$$R(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

- γ is a discount factor $\gamma \in [0,1]$



MDP Example - Gridworld

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



Demo Gridworld

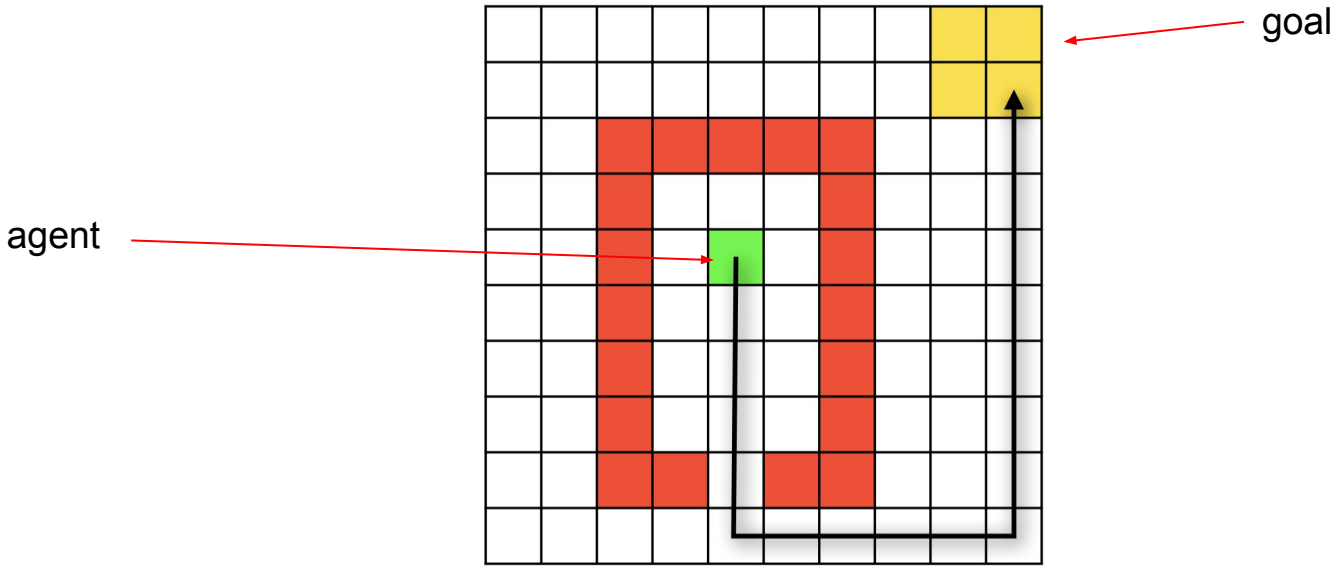
Manual control

```
python gridworld.py -n 0.2 -m
```

Noise level
How often action results
in an unintended direction

Markov Decision Process Example

Gridworld Example



Markov Decision Process Example

States

A **Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

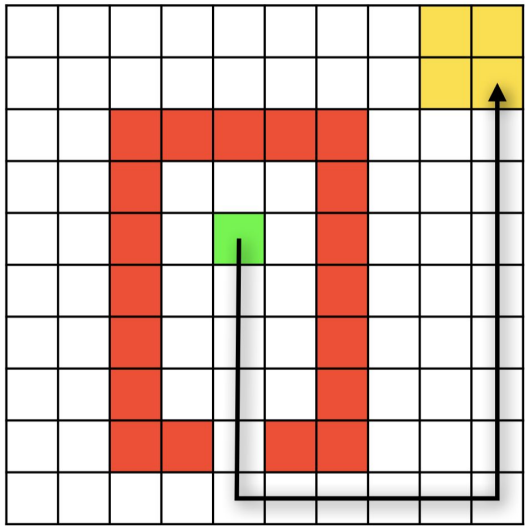
$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- R is a reward function $R(s,a,s')$ or $R(s)$ or $R(s')$

$$R(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

- γ is a discount factor $\gamma \in [0,1]$

position on the grid



Markov Decision Process Example

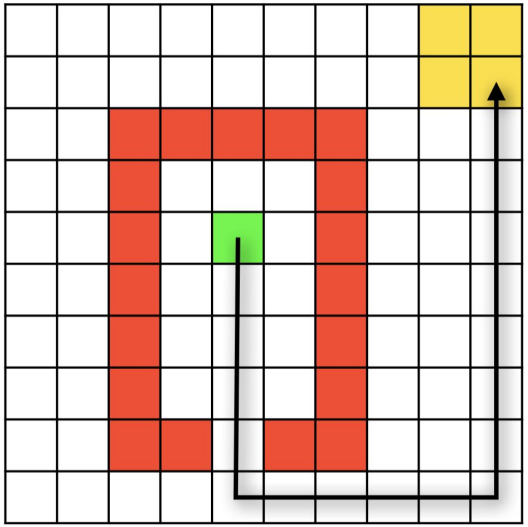
States

- A state captures 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, memories etc.
- A state should summarize past sensations so as to retain all "essential" information, i.e., it should have the **Markov Property:** *Current state completely characterises the state of the world*
- We should be able to throw away the history once state is known

$$\mathbb{P}[R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t] = \mathbb{P}[R_{t+1} = r, S_{t+1} = s' | S_t, A_t]$$

for all $s' \in \mathcal{S}, r \in \mathcal{R}$, and all histories

position on the grid



Markov Decision Process Example

Actions

A **Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- R is a reward function $R(s,a,s')$ or $R(s)$ or $R(s')$

$$R(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

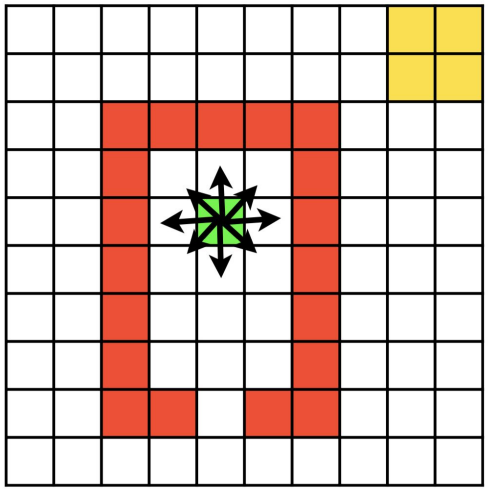
- γ is a discount factor $\gamma \in [0,1]$

actions = {

1. right
2. left
3. up
4. down

}

For now we consider *discrete actions*



Markov Decision Process Example

Transition probability function

A **Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

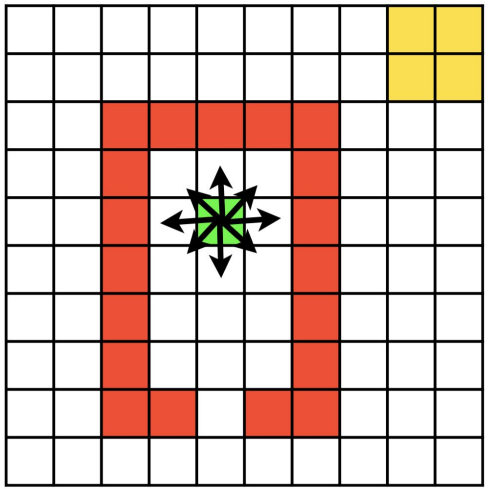
$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- R is a reward function $R(s,a,s')$ or $R(s)$ or $R(s')$

$$R(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

- γ is a discount factor $\gamma \in [0,1]$

State transition matrix, defines transition probabilities from all states *s* to all successor states *s'*

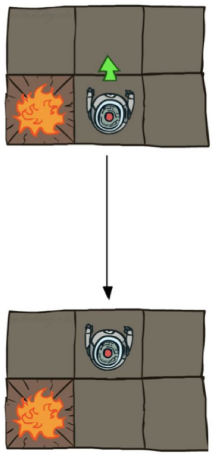


Define the *dynamics* of the environment

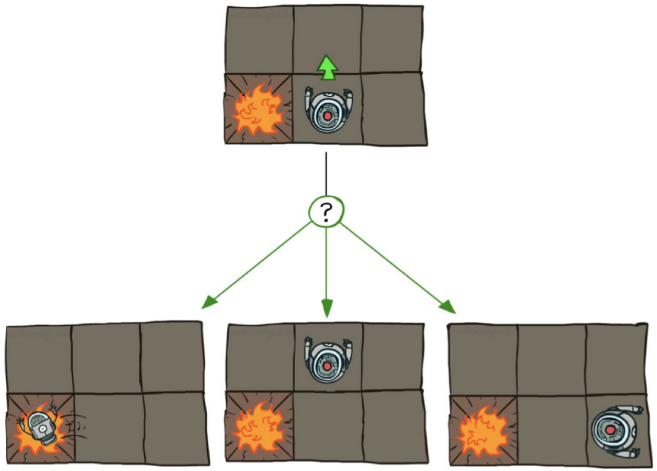
Markov Decision Process Example

Transition probability function

Deterministic Grid World



Stochastic Grid World



Markov Decision Process Example

The expectation is the sum of all its possible values weighted by their probability of being observed

$$\sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a).$$

Reward Function

A **Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

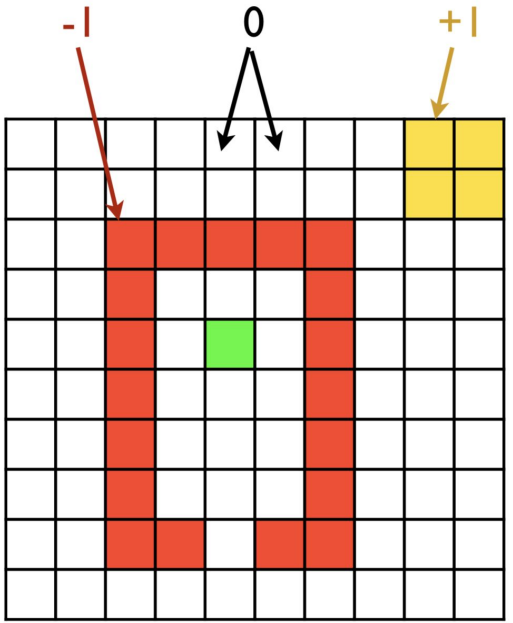
$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- R is a reward function $R(s,a,s')$ or $R(s)$ or $R(s')$

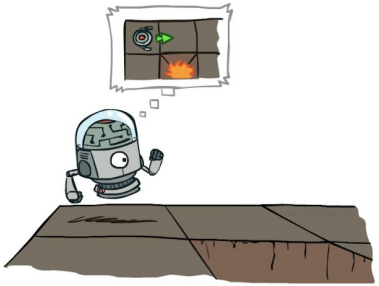
$$R(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

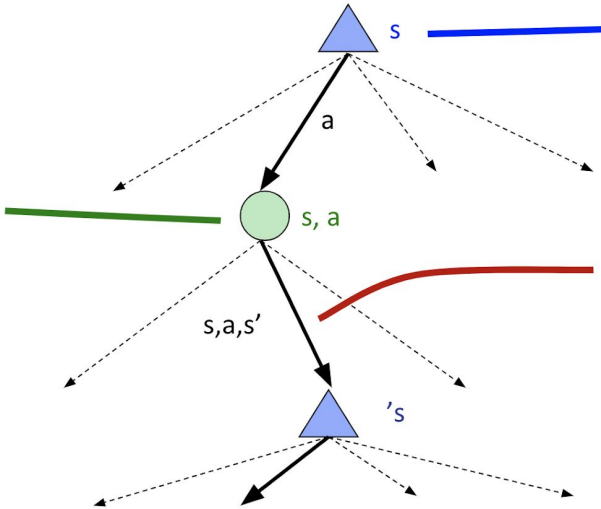
- γ is a discount factor $\gamma \in [0,1]$

$$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$

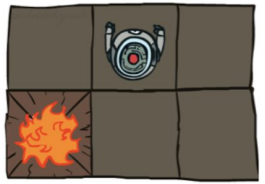


MDP Search Trees





s is a *state*



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

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

$R(s,a,s')$

Markov Decision Process Example

Discount Factor

A **Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

$$T(s'|s,a) = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- R is a reward function $R(s,a,s')$ or $R(s)$ or $R(s')$

$$R(s,a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

- γ is a discount factor $\gamma \in [0,1]$

Intuition:

- Receiving \$80 today is worth the same as \$100 tomorrow (assuming a discount factor of factor of γ = 0.8).
- At each time step, there is a 1-γ chance that the agent dies, and does not receive rewards afterwards.

Returns and Episodes

RL Goal: the agent has to maximize the cumulative reward it receives in the long run

Maximize the expected **return:**

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

Final time step

Episodic task: consider return over *finite horizon* (e.g. games, maze).

Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state

Returns and Episodes

RL Goal: the agent has to maximize the cumulative reward it receives in the long run

Maximize the expected **return:**

discount factor (or discount rate)

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Continuing task: consider return over *infinite horizon* (e.g. juggling, balancing)

If γ = 0, the agent is *myopic* in being concerned only with maximizing immediate rewards.
As γ approaches 1, the return objective takes future rewards into account more strongly, the agent becomes more *farsighted*.

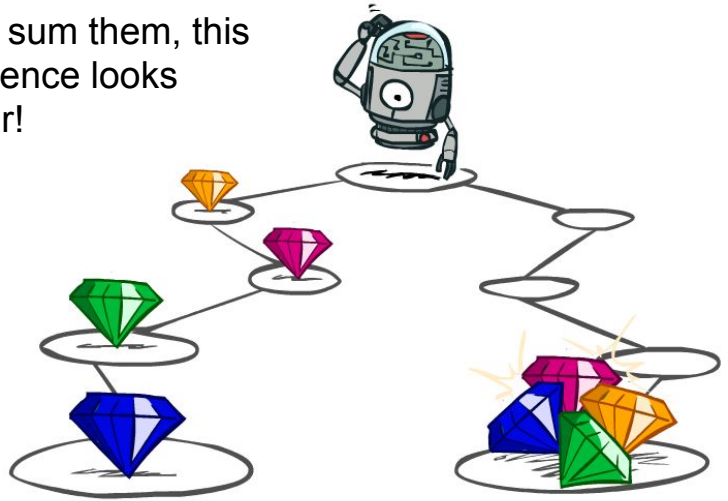
Discounting

The rewards come step-by-step

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]

If we sum them, this sequence looks better!

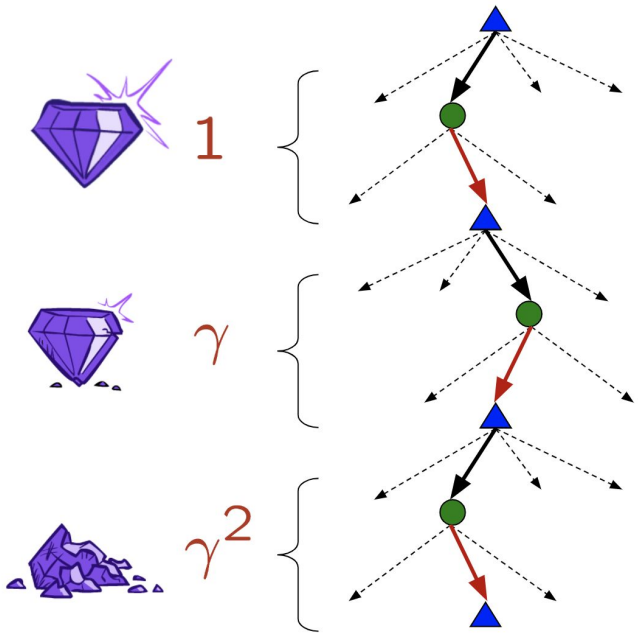


Discounting

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

Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge
- Example: discount of 0.5
  - G([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - G([1,2,3]) < G([3,2,1])

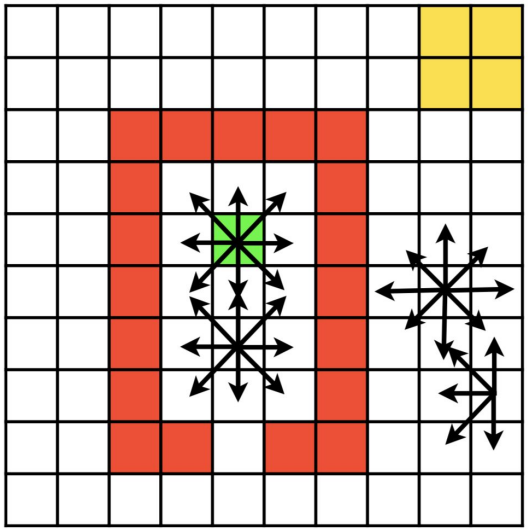


Policy

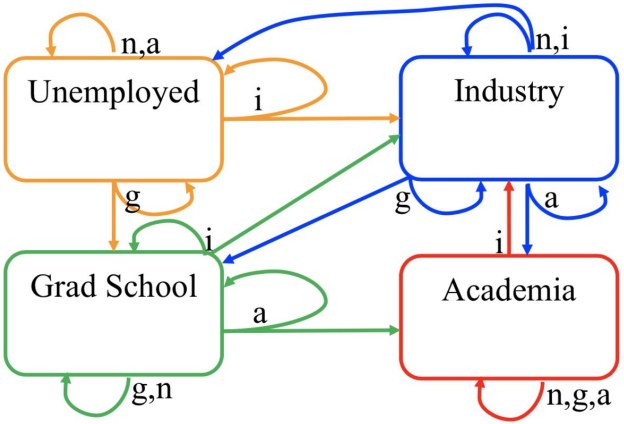
Mapping from states to probabilities of selecting each possible action.

$$\pi(a|s) \quad = \quad p(a|s)$$

If the agent is following policy π at time *t*, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$



Example: Career Options

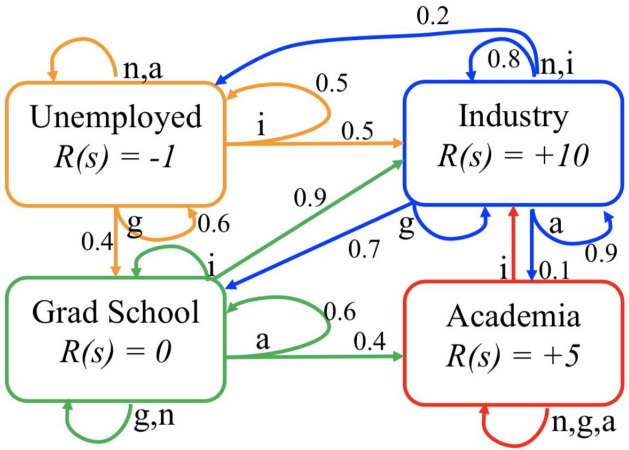


Actions:

n=Do Nothing
i = Apply to industry
g = Apply to grad school
a = Apply to academia

What is the best policy?

Example: Career Options



Actions:

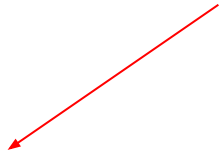
n=Do Nothing
i = Apply to industry
g = Apply to grad school
a = Apply to academia

What is the best policy?

Value functions

Expected return of a policy (for every state)

Very hard in practice



Simple strategy to find the best policy:

1. Enumerate the space of all possible policies.
2. Estimate the expected return of each one.
3. Keep the policy that has maximum expected return.

Confused with terminology?

- **Reward**: 1 step numerical feedback
- **Return**: Sum of rewards over the agent's trajectory.
- **Value**: Expected sum of rewards over the agent's trajectory.

Rewards are instantaneous, Values are accumulative

Value Function

How good is to be in a given state

Defined in terms of future rewards that can be expected, or, to be precise, in terms of expected **return**. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to **policies**.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}$$

Also called *state-value* function

Action-value function

How good is to take an action in a given state

Value of taking action *a* in state *s* under a policy π, denoted $q_\pi(s,a)$ as the expected return starting from s, taking the action a, and thereafter following policy π

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Optimal Policy

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run.

Optimal state-value function is the highest value that can be achieved for each state:

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

Any policy that achieves v_* is called **optimal policy**

Optimal Policy

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run.

Optimal policies also share the same **optimal action-value function**, denoted q_*, and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a)$$

For the state-action pair (s, a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

Optimal Quantities

- The value of a state s:

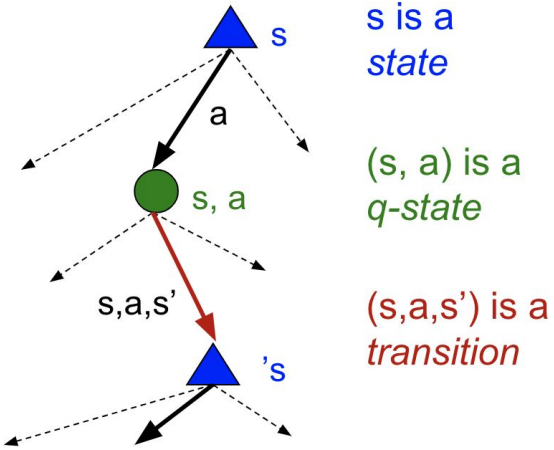
  $V^*(s)$ = expected return starting in s and acting optimally

- The value of a q-state (s,a):

  $Q^*(s,a)$ = expected return starting out having taken action a from state s and (thereafter) acting optimally

- The optimal policy:

  $\pi^*(s)$ = optimal action from state s



Demo

Agent type

Runs of the MDP

```
python gridworld.py -n 0.2 -a value -d 0.9 -k 0 -i 100
```

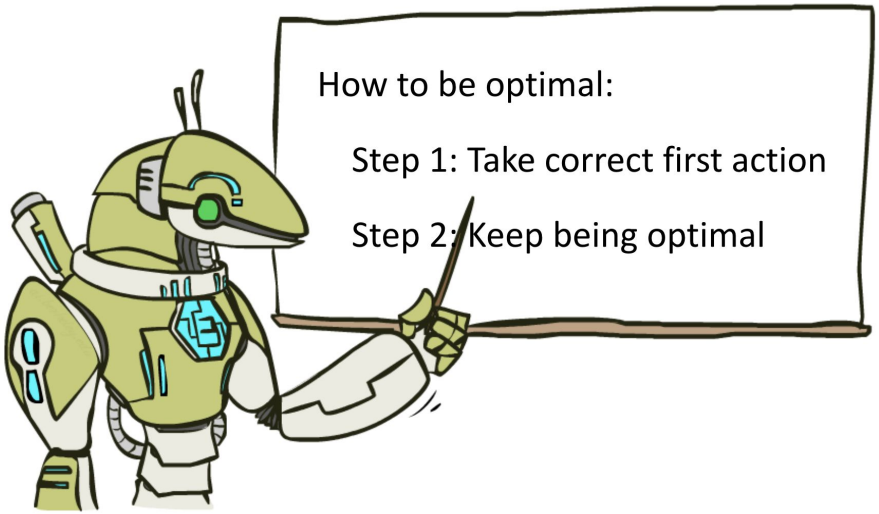
How often action results in an unintended direction

Discount factor

Iterations

The Bellman Equations

They specify what it means to be optimal in a recursive way



The Bellman Equations

Simple one-step lookahead relationship amongst optimal utility values

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

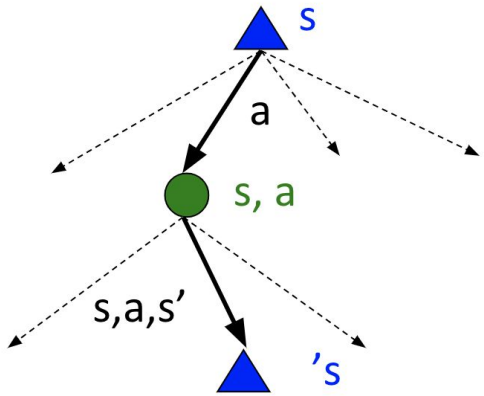
look at all the action and maximize over the values of the children nodes

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

weighted average for all outcomes s'

instantaneous reward for this transition

value in the landing state, discounted



putting them together:

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

Solving the MDP

Dynamic Programming Algorithms

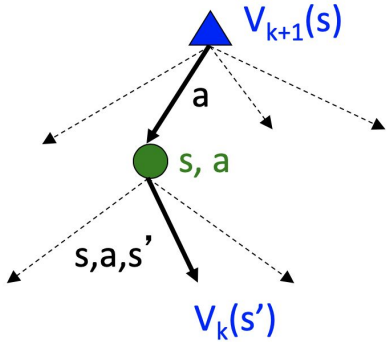
Collection of algorithms that can be used to compute optimal policies *given a perfect model of the environment* as a Markov decision process (MDP)

Use of *value functions* to organize and structure the search for good policies.

- Value Iteration
- Policy Iteration

Value Iteration

One-step lookahead, s->s'



- Bellman equations characterize the optimal values:

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

- Value iteration computes them:

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

Max value across all the actions I can choose

Average of all outcomes, weighted by their probability of occurring *T*

Instantaneous reward of the landing state s'

Discounted future value of the landing state s'

Value Iteration

1. Start with an arbitrary initial approximation of V(s)
2. On each iteration, update the value function estimate:

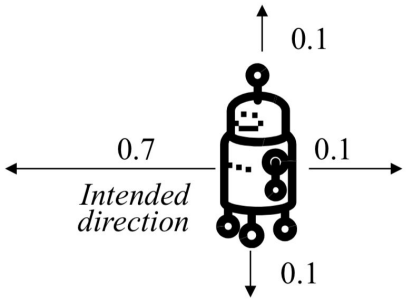
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

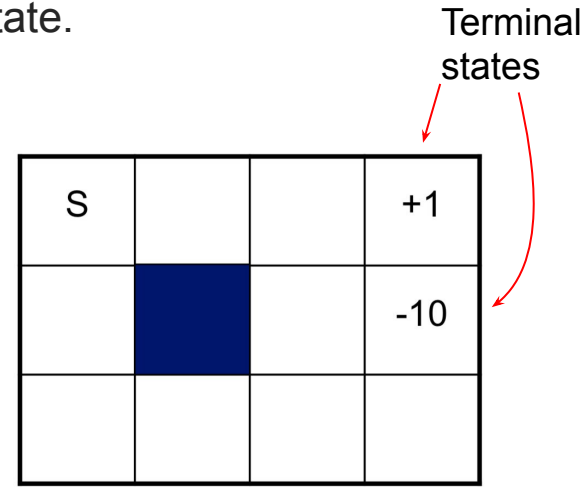
3. Stop when max value change between iteration is below a threshold.

The algorithm converges (in the limit) to the true optimal value function V*

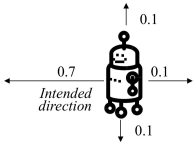
Value Iteration Example

- 11 discrete states, 4 motion actions (N, S, E, W) in each state.
- Transitions are mildly stochastic.
- Reward is +1 in top right state, -10 in state directly below, -0 elsewhere.
- Episode terminates when the agent reaches +1 or -10 state.
- Discount factor g = 0.99.





Value Iteration (1)



Values initializations

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

For each state:
Look at the actions you can take, for each action let's see where I can end up and what's the value of there.

0	0	0	+1
0		0	-10
0	0	0	0

We initialize the state-values to be equal to the reward function

Value Iteration (2)

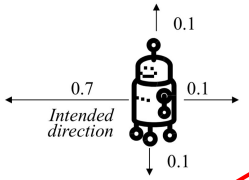
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

Best value if going *right* = ***a***

reward for choosing action *right*

After one round

For each state:
Look at the actions you can take, for each action let's see where I can end up and what's the value of there. Then choose the highest value.



0 +

0.99(0.7*1+0.1*0+0.1*0+0.1*0) = **0.69**

discount factor | intended direction (right) | left | down | up

By going up, there is a wall so it will stay where it is, so V(s')=0

0	0	0	+1
0		0	-10
0	0	0	0

Stay zero cause the neighbours are also zero

0	0	0.69	+1
0		-0.99	-10
0	0	0	-0.99

We should compute this for every action and then choose the highest (we only seen the value for choosing action=right, cause we know it was already the best choice)

Value Iteration (20)

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

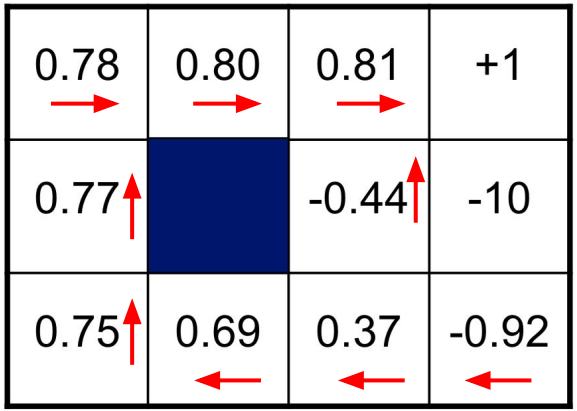
After 20 iterations

0.78	0.80	0.81	+1
0.77		-0.44	-10
0.75	0.69	0.37	-0.92

*Now it's easy to compute the **optimal policy** for each state*

Value Iteration (20)

After 20 iterations



*Now it's easy to compute the **optimal policy** for each state*

In each state, choose the action where the value increase in the next state

Demo

```
python gridworld.py -n 0.2 -a value -d 0.9 -v -i 100
```

Reinforcement *Learning*

So far we have only done ***planning*** on a *known* MDP

Real world problems don't have perfect information about the environments and the rewards..

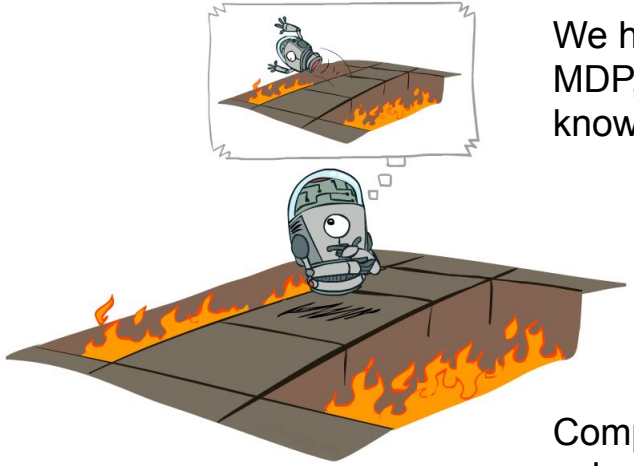
What is reinforcement ***learning***?

Reinforcement *Learning*

- Still assume a Markov decision process (MDP):
  - A set of states s ∈ 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 π(s)

- 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 out to learn

Offline (MDPs) vs Online (RL)



We have an MDP, so we know *T* and *R*

Compute the values for each state and plan

Offline Solution

Planning

We don't know the underlying MDP, but we can interact with the environment and collect ***samples***

Current state

Action

Experiences: (s, a, s', r)

Next state

Immediate reward

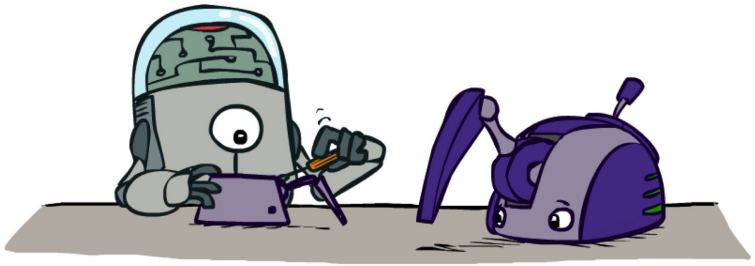
Online Learning

Learning and Planning

Online Learning (based on samples)

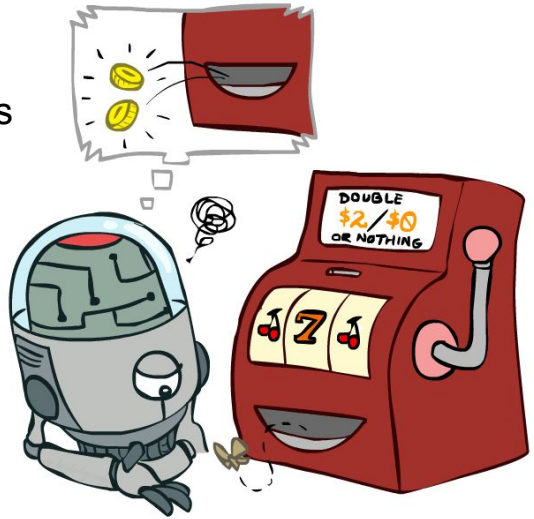
Model-Based

Learn an approximate model of the MDP and then plan on it using for example Value Iteration



Model-Free

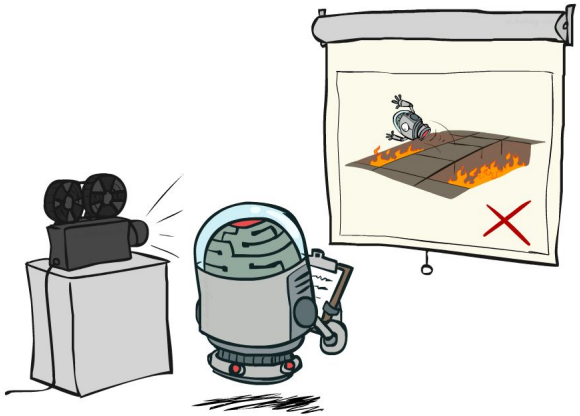
Estimate directly the values of the states from the experiences without building the MDP



Model-Free Learning

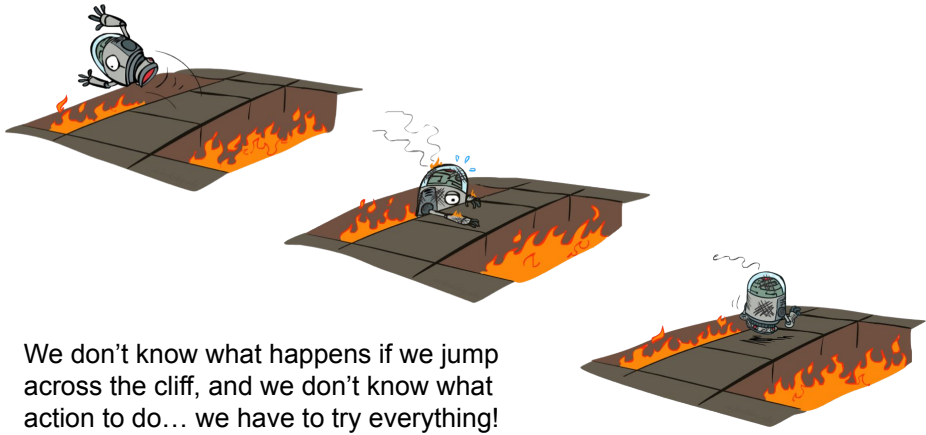
Passive Learning

We have a *fixed policy* and we want to evaluate the value-function for each state by running it and collecting samples



Active Learning

I don't have a policy, we try action and learn by trial-and-error. We approximate directly the action-value function for each state.



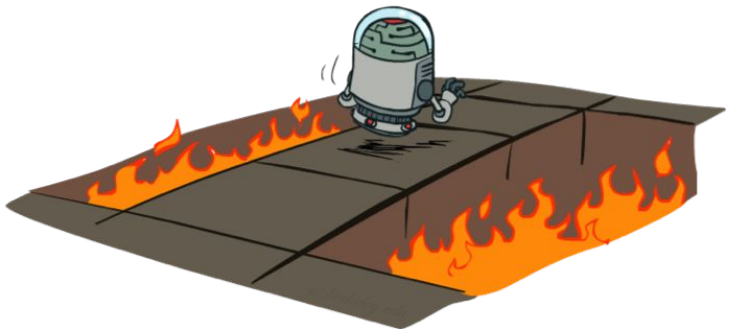
We don't know what happens if we jump across the cliff, and we don't know what action to do… we have to try everything!

Reinforcement Learning Summary

- Offline (we know the MDP) = planning
  - **Value Iteration**
  - Policy Iteration
- Online (we don't know the MDP) = collecting *samples* by interacting with the environment
  - Model-Based = we **build an estimate of the MDP** and then do planning on it
  - Model-Free = we **estimate directly the values** of the states
    - Passive RL = we know the policy, we learn the *value function*
      - TD-learning
    - Active RL = we don't know the policy, we learn the *action-value function (trial-and-error)*
      - **Q-learning**

Active Reinforcement Learning

- Full reinforcement learning: optimal policies (like value 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!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens…

Q-Learning

We can't compute this update without knowing *T*, R

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: $Q(s,a)$
  - The sample suggests that:

$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

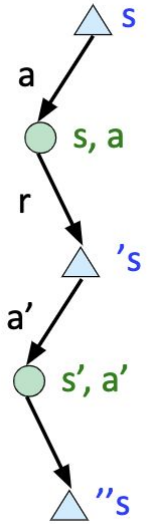
  - Incorporate the information received (***r***) into my old estimate

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha) \left[r + \gamma \max_{a'} Q(s',a') \right]$$

The *learning rate (or step-size)* determines to what extent newly acquired information overrides old information

Q-Learning

Model-free learning



- Experience world through episodes

$$(s, a, r, s', a', r', s'', a'', r'', s'''' \ldots)$$

- We don't build a model of T and R, instead we update the Q-values directly after we experience a transition: (s, a, r, s')
- How? We store the Q-values (e.g. in a table) and update them at each transition:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

current estimate of Q

Step-size / learning rate

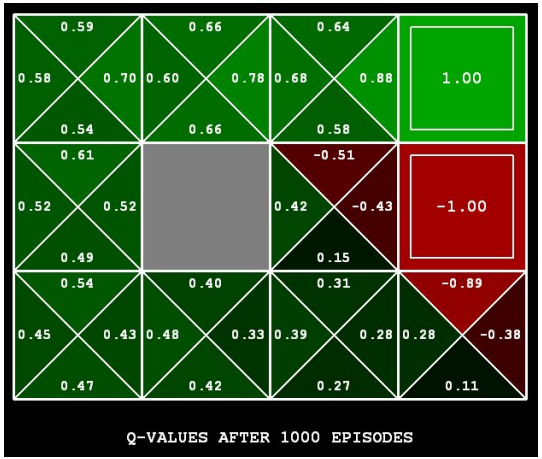
reward we experience

discount factor

estimate q-value of the next state

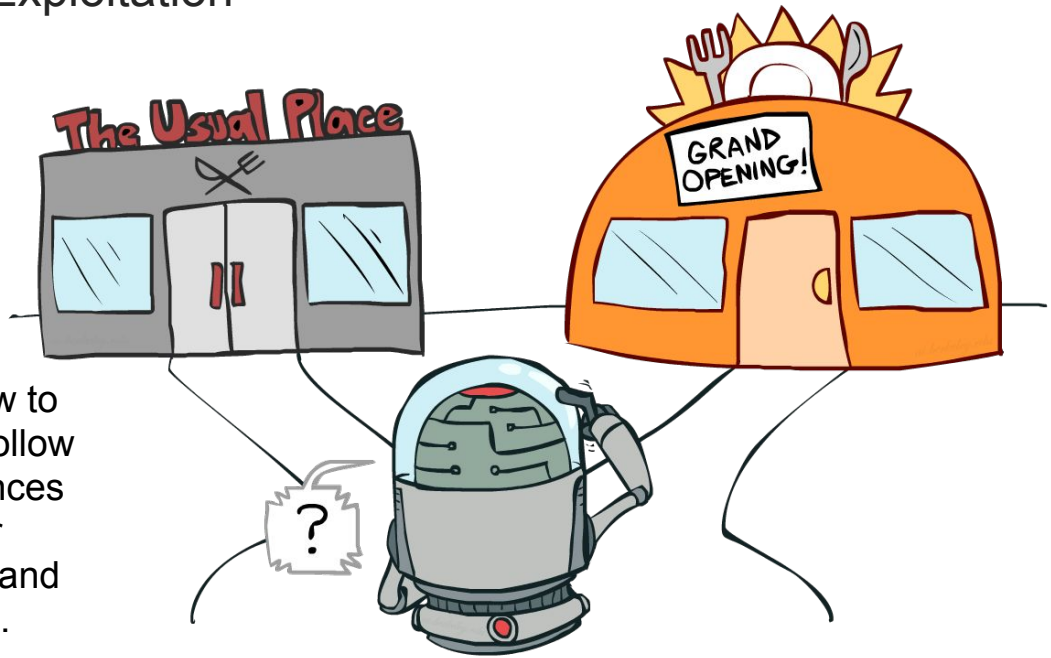
DEMO

```
python gridworld.py -a q -d 0.9 -i 100 -n 0.2 -m -k 100
```



We know how to learn, how do we act?

Exploration vs Exploitation



The Q value tells us how to act, we can choose to follow it based on the experiences we have sampled so far (**exploitation**), or don't and **exploring** other actions.

How to explore?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a 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

Q-Learning Algorithm

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Same as we have seen before, just different notation

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha)\left[r + \gamma \max_{a'} Q(s', a')\right]$$

Q-Learning

- 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 (!)

You will find the optimal policy (optimal Q-values) also if you never act according to it!

Summary

What we have seen in this lecture:

- Supervised/Unsupervised Learning vs Reinforcement Learning
- Reinforcement Learning Framework
- RL Examples
- Markov Decision Process
- Gridworld Example
- Returns and Episodes
- Discount Factor
- Value Functions
- Policy
- Optimal Policy, value and q
- Bellman equations
- Solving the MDP
- Value Iteration
- Online vs Offline learning
- Model-based vs Model-free learning
- Q-learning

Extra Material

(will not be in the exam, but it’s good if you want to know more about it..)

Solving the MDP

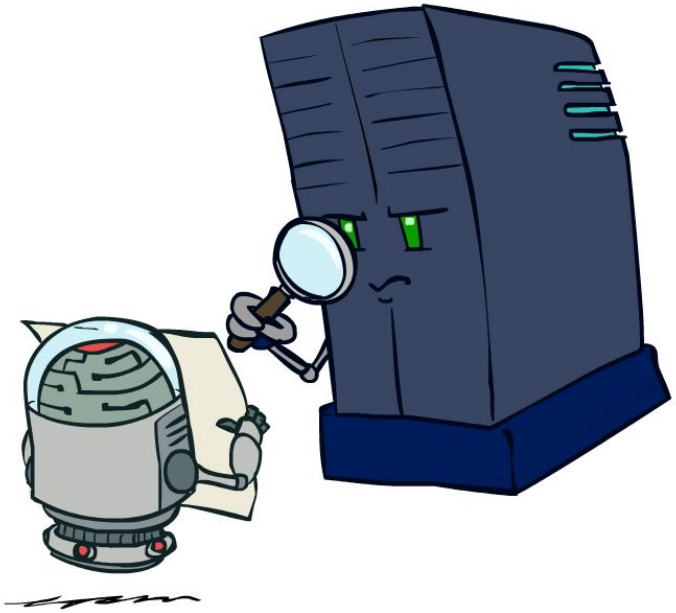
Dynamic Programming Algorithms

Collection of algorithms that can be used to compute optimal policies *given a perfect model of the environment* as a Markov decision process (MDP)

Use of *value functions* to organize and structure the search for good policies.

- **Value Iteration** ← seen so far
- Policy Iteration
  - Policy Evaluation +
  - Policy Improvement

Policy Evaluation



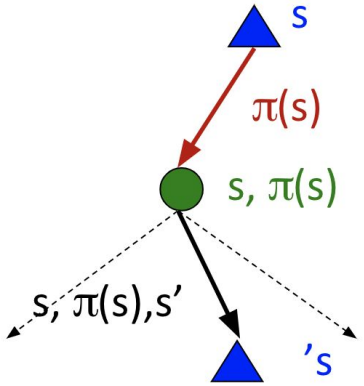
Value Function for a Fixed Policy

- Compute the value of a state s under a fixed (generally non-optimal) policy

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

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

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



- Recursive relation (one-step look-ahead / Bellman equation):

Policy Evaluation

How to compute the state-value function v_π for an arbitrary policy π

1. Start with some initial guess of the value function for each state (e.g. 0 or r(s))

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

2. During every iteration *k,* update the value function for all states

   Immediate reward *value function of next state*

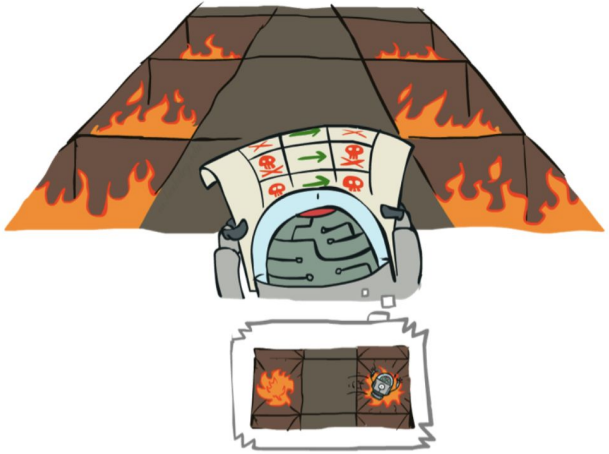
   $$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

3. Stop when the maximum changes between two iterations is smaller than a desired threshold (the values stop changing)

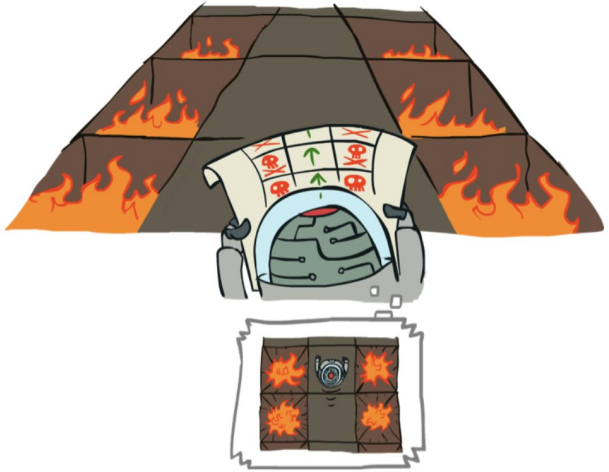
Guaranteed to converge!

Policy Evaluation Example

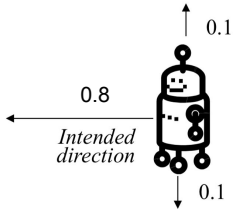
Always Go Right



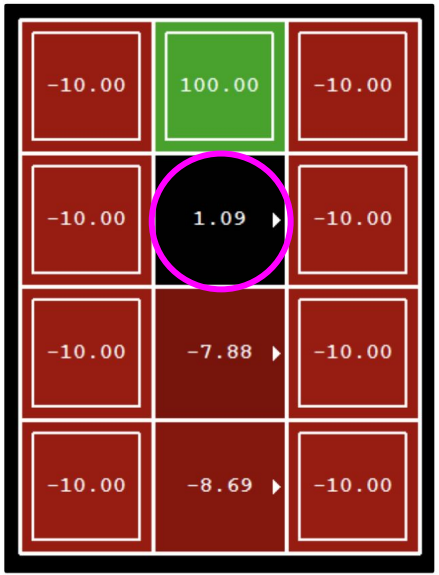
Always Go Forward



Policy Evaluation Example



Always Go Right



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

0 + 0.9*(0.8*(-10) + 0.1*(100) + 0.1*(-7.88)) = **1.09**

Immediate reward

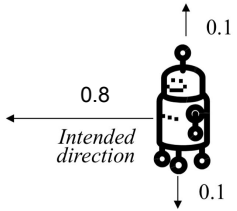
Discount factor

Value for Intended action

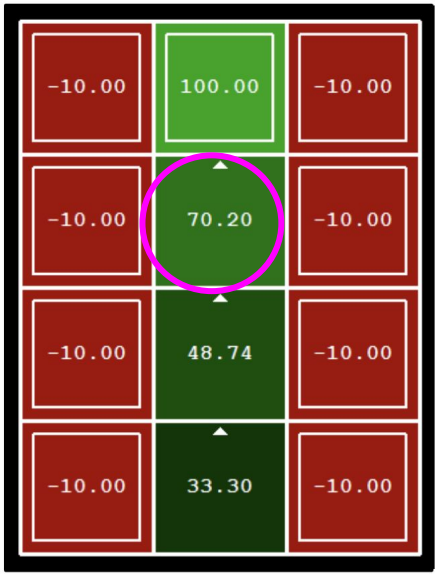
Value left

Value right

Policy Evaluation Example



Always Go Forward



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

0 + 0.9*(0.8*(100) + 0.1*(-10) + 0.1*(-10)) = **70.2**

Immediate reward

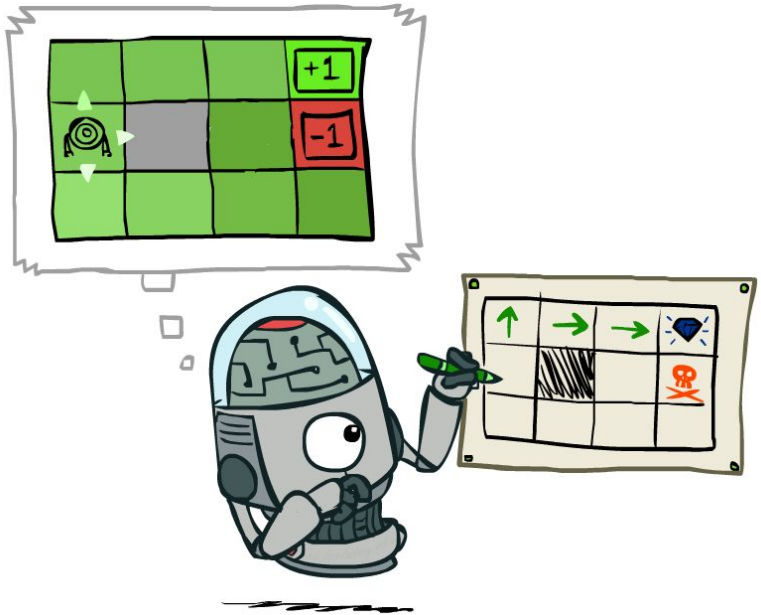
Discount factor

Value for Intended action

Value left

Value right

Policy Improvement



Policy Improvement

We compute the value function for a policy so that we can find better policies.

How good it is to follow the current policy from **s** - that is $v_\pi(s)$- but would it be better or worse to change to the new policy?

Let's consider the value of selecting **a** in **s** following the existing policy π

$$q_\pi(s,a) \doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a]$$

Is it greater than or less than $v_\pi(s)$?

Policy Improvement

Select at each state the action that appears best according to $q_\pi(s, a)$

The *greedy policy* takes the action that looks best in the short term - after one step of lookahead - according to the value function of the original policy v_π

$$\begin{aligned}
\pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\
&= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma v_\pi(s') \right]
\end{aligned}$$

V^π equal

Look at all the actions, choose the action at state **s** that maximise the return

Weighted return of executing action **a** followed by normally executing π

Policy improvement gives us a strictly better policy except when the original policy is already optimal.

Policy Iteration

Once a policy, π has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

1. Start with an initial policy π (e.g. radom)

2. Repeat:
   a. Compute v_π using iterative policy evaluation
   b. Compute a new policy π' using policy improvement (i.e. greedy with respect to v_π)

3. Terminate when $\pi = \pi'$

Iterative policy evaluation on a gridworld

The goal is to make it to one of the two gray corners, ending the game.
To encourage this, *every move will have a reward of -1.*

Instead of sticking to the same policy every time the values are iterated, the policy acts greedily towards the best expected result.

Immediate reward *value function of next state*

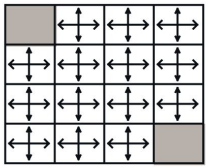
$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

v_k for the random policy

greedy policy w.r.t. v_k

$k = 0$

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

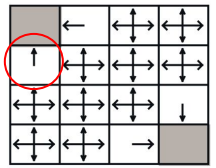


random policy

(0.25*4)*(-1) + (0.25*4)*(0)

$k = 1$

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



Immediate reward following the policy (up)

(1)*(-1) +

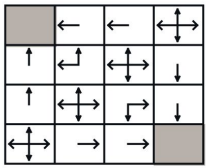
(0.25)*(0) + (0.25)*(-1) + (0.25)*(-1) + (0.25)*(-1)

up down right left (stay)

Value of the neighbouring states weighted on the probabilities of going there (equal in this case)

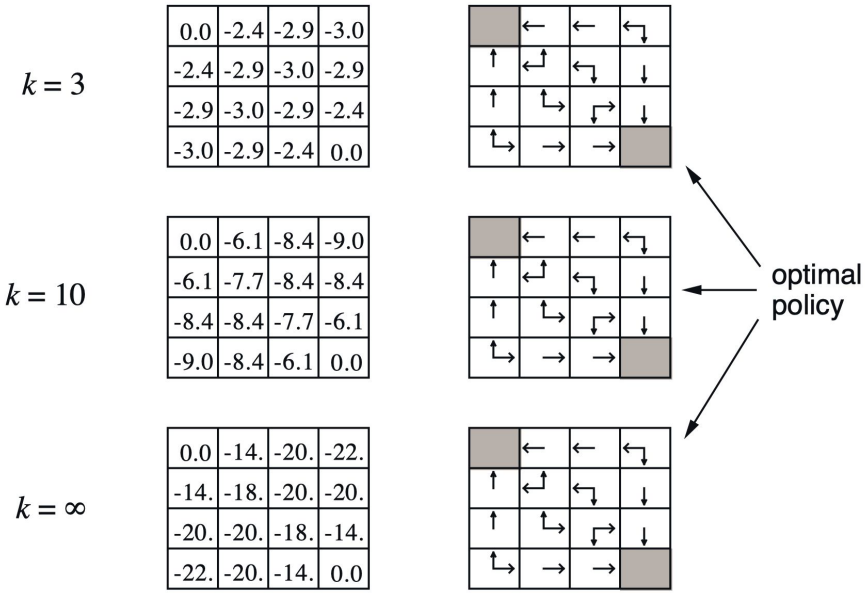
$k = 2$

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



Iterative policy evaluation on a gridworld

Eventually, the policy would reach a point where continuing to iterate would no longer change anything. That final policy would therefore be the optimal policy for that environment.



Three related algorithms

1. **Policy evaluation:** Fix the policy, estimate its value.

2. **Policy iteration:** Find the best policy at each state.
   a. Policy evaluation + greedy improvement

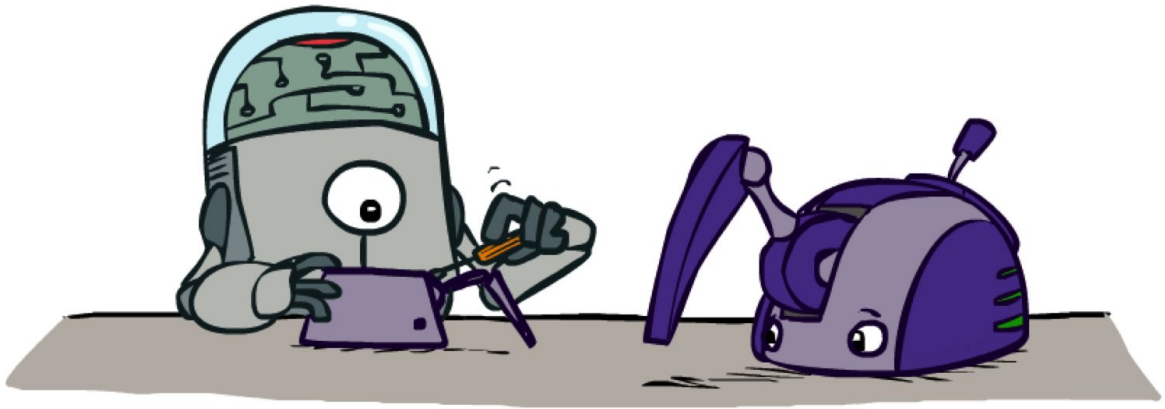
3. **Value Iteration:** Find the optimal value function

Comparison Value vs Policy Iteration

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

Model-Based Learning

I don’t know the MDP but I can learn it and the run Value Iteration to solve it

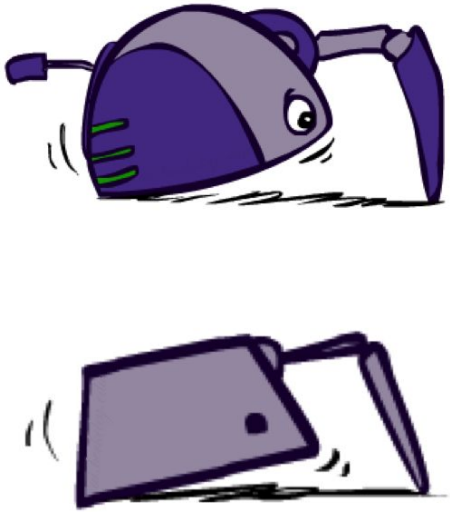


Model-Based Learning

- Model-Based Idea:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

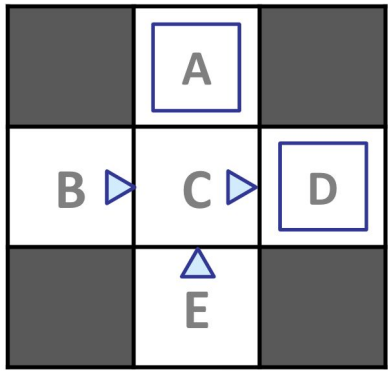
- Step 1: Learn empirical MDP model
  - Count outcomes s' for each s, a
  - Give an estimate of $\hat{T}(s, a, s')$
  - Discover each $\hat{R}(s, a, s')$ when we experience (s, a, s')

- Step 2: Solve the learned MDP
  - For example, use value iteration, as before



Model-Based Learning Example

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

Learned Model

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

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

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

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

We can apply Value Iteration now

Compute expected age of students here

Known P(A)

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

Without P(A), instead collect samples [a_1, a_2, … a_N]

Unknown P(A): "Model Based"

$$\hat{P}(a) = \frac{\text{num}(a)}{N}$$

$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

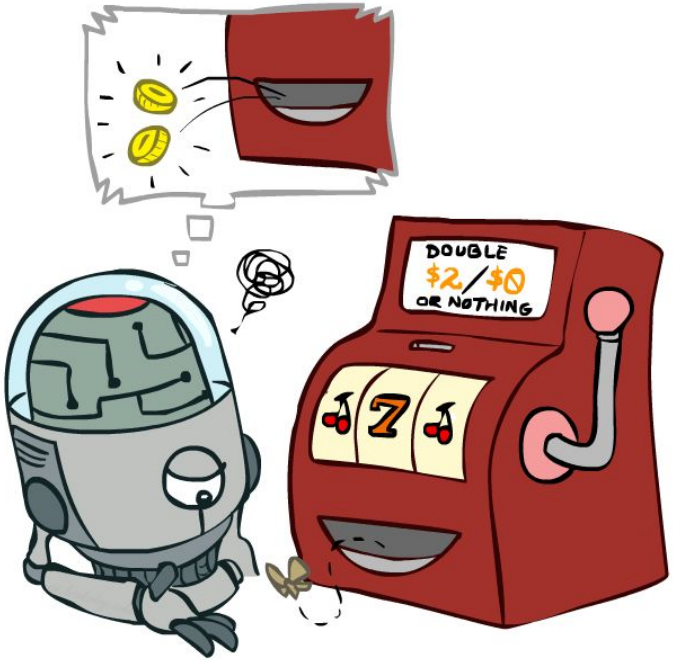
Why does this work? Because eventually you learn the right model.

Unknown P(A): "Model Free"

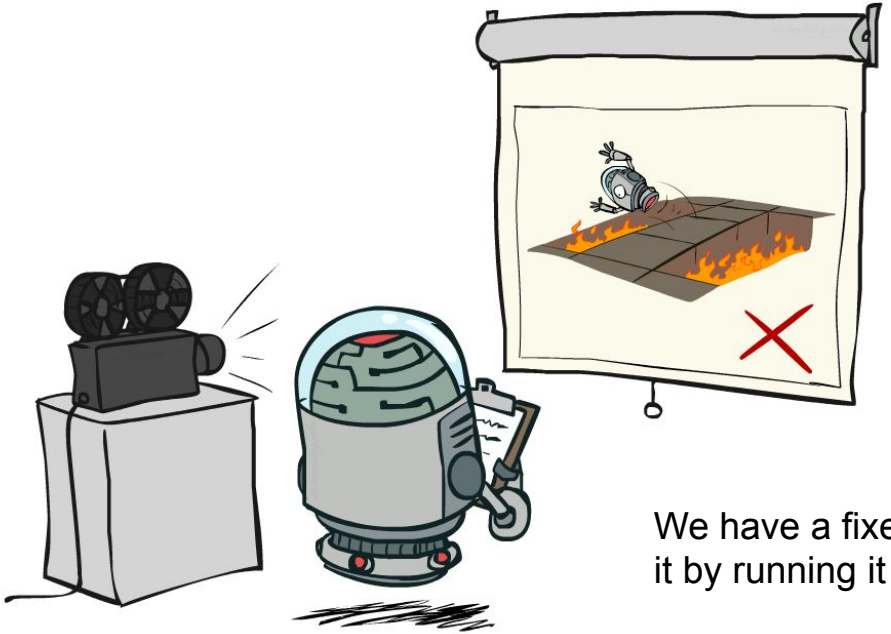
$$E[A] \approx \frac{1}{N} \sum_i a_i$$

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

Model-Free Learning



Passive Reinforcement Learning

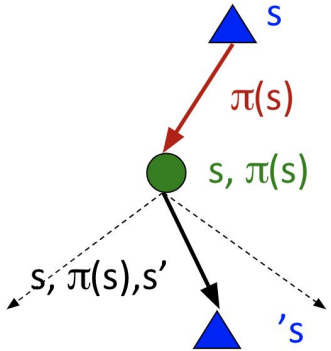


We have a fixed policy and we want to evaluate it by running it and collecting ***samples***

Temporal Difference Learning

Learning from experience

Temporal Difference: Nudge the values towards *recent* samples



- Act according a fixed policy π
- Update V(s) each time we experience a transition (s, a, s', r)
- Likely outcomes s' will contribute updates more often
- Move values toward value of whatever successor occur: running average

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

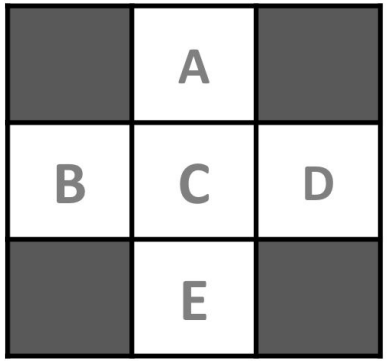
Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$

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

Learning rate: parameter between 0 and 1 that regulates how the new information overwrites the old one, forgetting about the past

Temporal Difference Learning Example

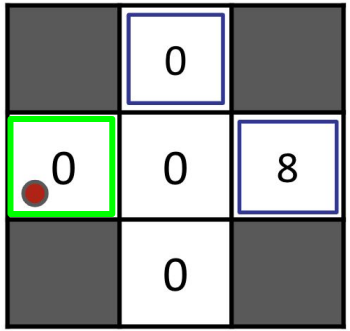
States

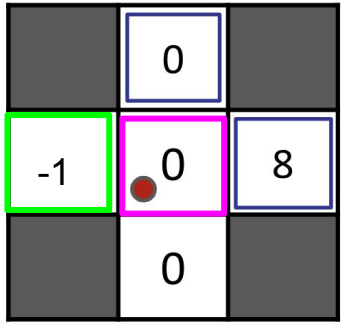


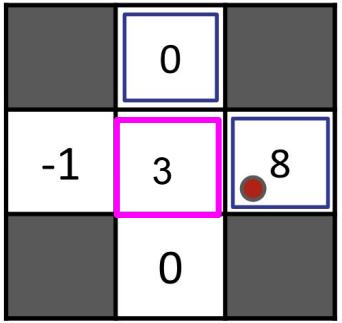
Assume: $\gamma = 1$, $\alpha = 1/2$

B, east, C, -2

C, east, D, -2







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

1/2 0 1/2 -2 1 0

1/2 0 1/2 -2 1 8

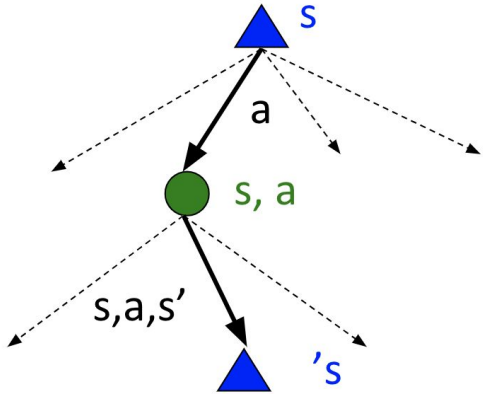
Temporal Difference Learning

- TD value leaning 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

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

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

- Idea: learn Q-values (action-value), instead of state-values
- Makes action selection model-free too!



Q-Value Iteration

- Remember - Value iteration: find successive values
  - Start with $V_0(s) = 0$
  - Given V_k, calculate the k+1 values for all states:

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

Hard to do the max from samples

- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s,a) = 0$
  - Given Q_k, calculate the k+1 q-values for all q-states:

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

We can do average from samples