Introduction to Artificial Intelligence

Lecture 9: Reinforcement Learning

Prof. Gilles Louppe
g.louppe@uliege.be

LIÈGE université

Today

How to make decisions under uncertainty, while learning about the environment?

- Reinforcement learning (RL)
- Passive RL
  - Model-based estimation
  - Model-free estimation
  - Direct utility estimation
  - Temporal-difference learning
- Active RL
  - Model-based learning
  - Q-Learning
  - Generalizing across states

—

Image credits: CS188, UC Berkeley.

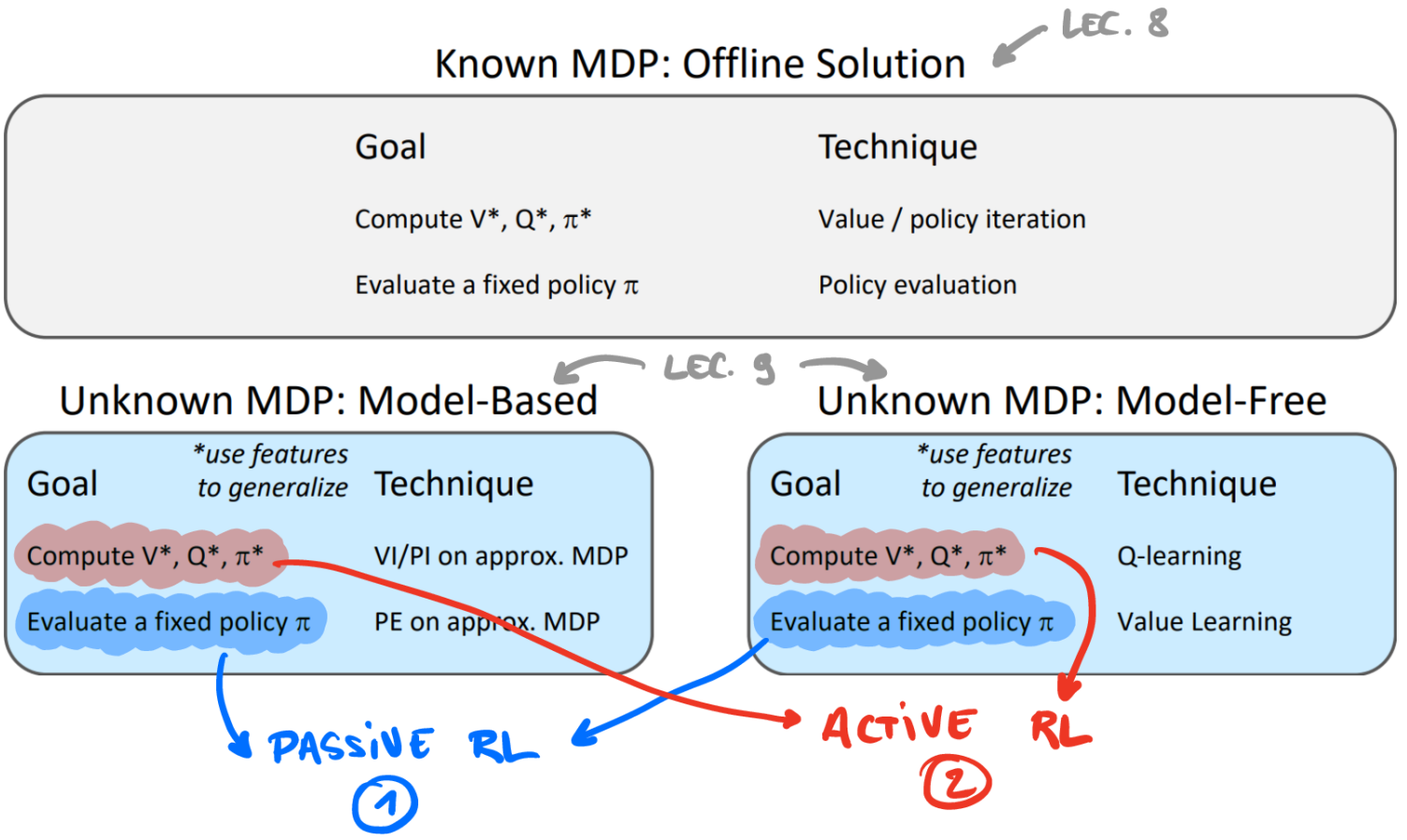


Image credits: CS188, UC Berkeley.

MDPs

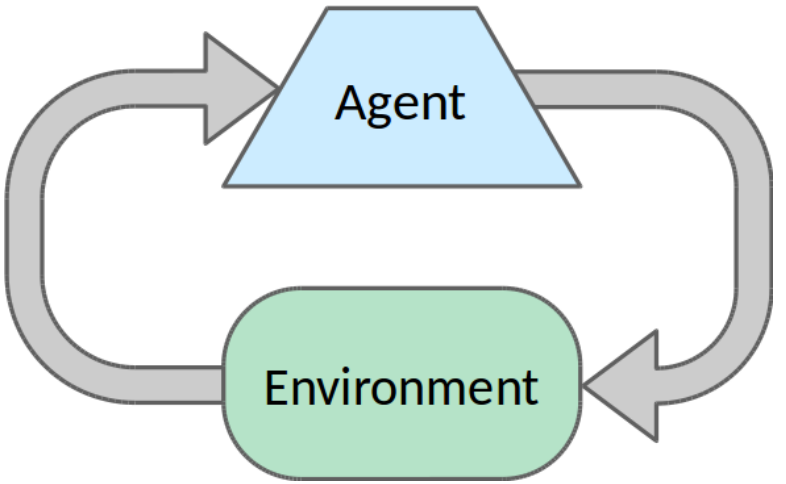
A short recap.

MDPs

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, P, R)$ such that:

- $\mathcal{S}$ is a set of states s;
- $\mathcal{A}$ is a set of actions a;
- P is a (stationary) transition model such that $P(s' \mid s, a)$ denotes the probability of reaching state s' if action a is done in state s;
- R is a reward function that maps immediate (finite) reward values $R(s)$ obtained in states s.
- ($0 < \gamma \leq 1$ is the discount factor.)

s



s'

$r' = R(s')$

a

$s' \sim P(s'|s, a)$

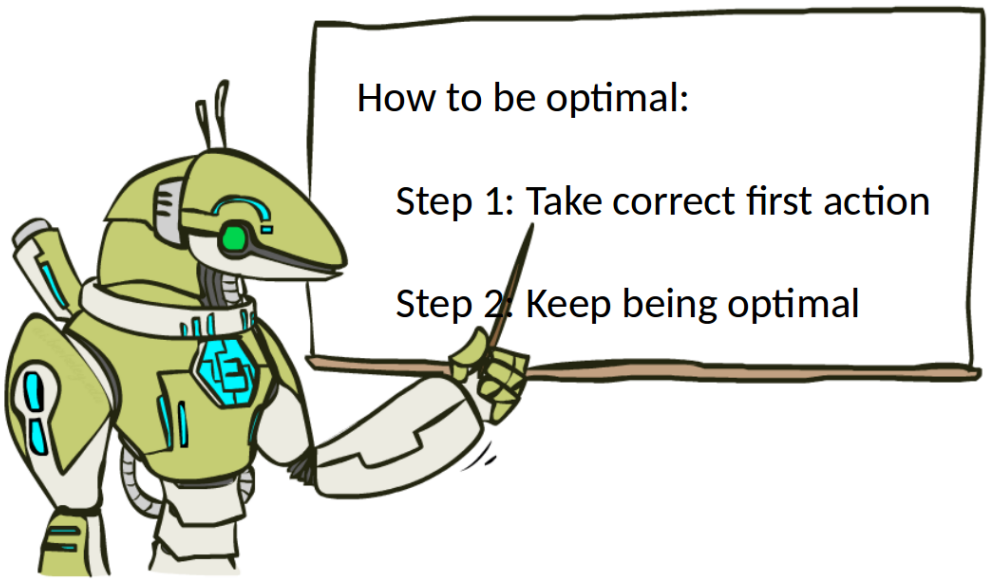
Remarks

- Although MDPs generalize to continuous state-action spaces, we assume in this lecture that both $\mathcal{S}$ and $\mathcal{A}$ are discrete and finite.
- The formalism we use to define MDPs is not unique. A quite well-established and equivalent variant is to define the reward function with respect to a transition (s, a, s'), i.e. $R(s, a, s')$. This results in new (but equivalent) formulations of the algorithms covered in Lecture 8.

The Bellman equation

The utility of a state is the immediate reward for that state, plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action:

$$V(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)V(s').$$



—

Image credits: CS188, UC Berkeley.

Value iteration

The value iteration algorithm provides a fixed-point iteration procedure for computing the state utilities $V(s)$:

- Let $V_i(s)$ be the estimated utility value for s at the i-th iteration step.
- The Bellman update consists in updating simultaneously all the estimates to make them locally consistent with the Bellman equation:

$$V_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s').$$

- Repeat until convergence.

Policy iteration

The policy iteration algorithm directly computes the policy (instead of state values). It alternates the following two steps:

- Policy evaluation: given π_i, calculate $V_i = V^{\pi_i}$, i.e. the utility of each state if π_i is executed:

$$V_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) V_i(s').$$

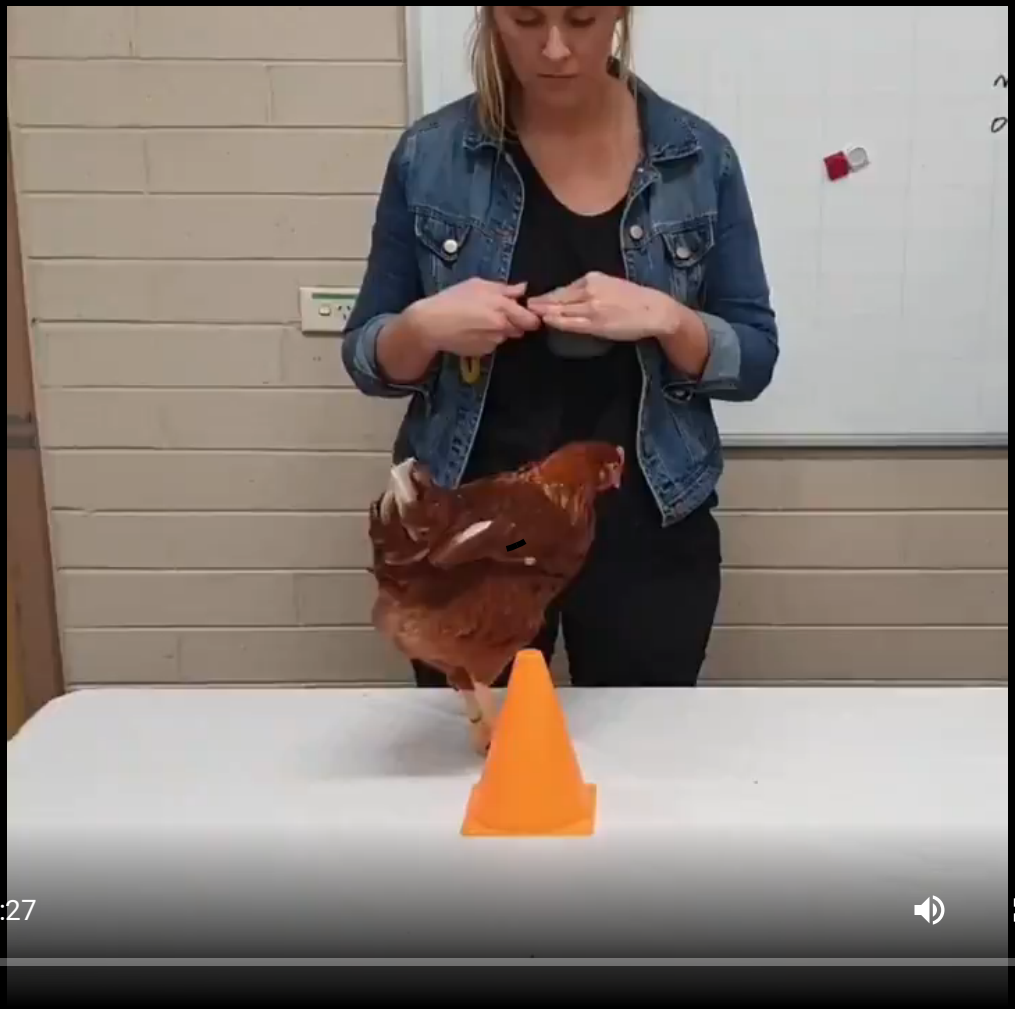
- Policy improvement: calculate a new policy π_{i+1} using one-step look-ahead based on V_i:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} P(s'|s, a) V_i(s').$$

Reinforcement learning

—

Video credits: Megan Hayes, @YAWScience, 2020.



—

Video credits: Megan Hayes, @YAWScience, 2020.

What just happened?

- This wasn't planning, it was reinforcement learning!
- There was an MDP, but the chicken couldn't solve it with just computation.
- The chicken needed to actually act to figure it out.

Important ideas in reinforcement learning that came up

- Exploration: you have to try unknown actions to get information.
- Exploitation: eventually, you have to use what you know.
- Regret: even if you learn intelligently, you make mistakes.
- Sampling: because of chance, you have to try things repeatedly.
- Difficult: learning can be much harder than solving a known MDP.

Reinforcement learning

We still assume a Markov decision process $(\mathcal{S}, \mathcal{A}, P, R)$ such that:

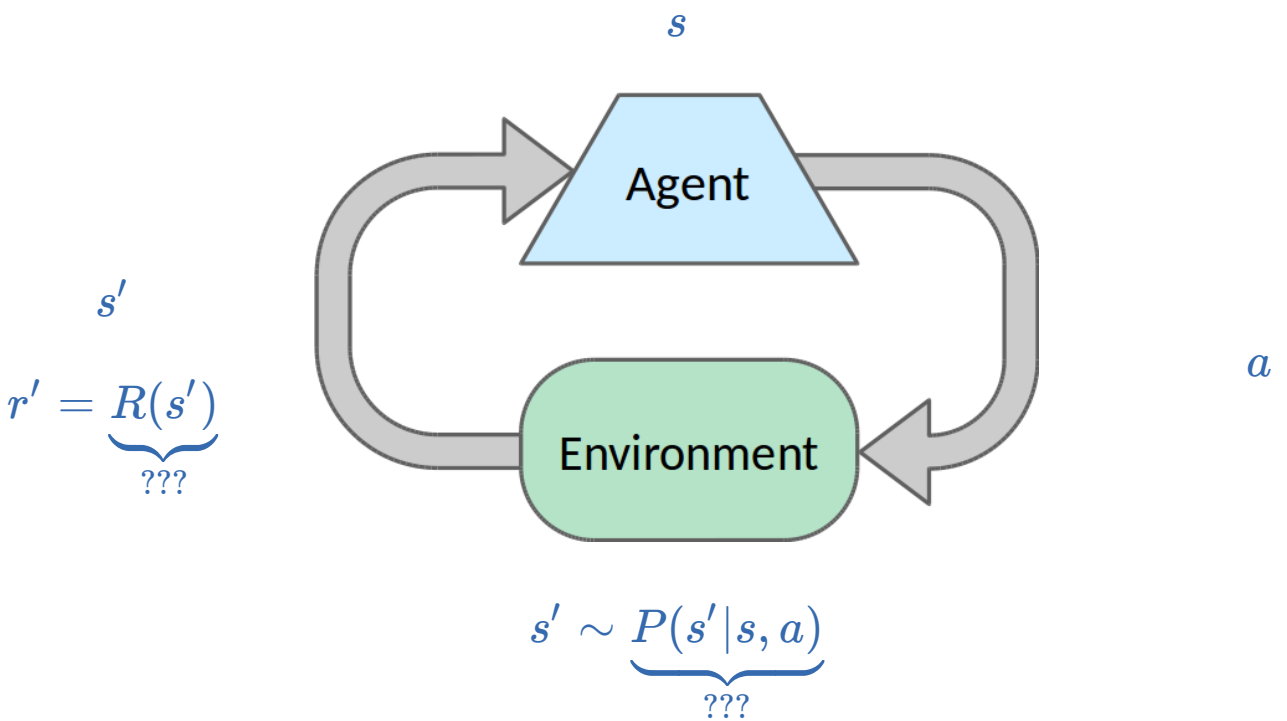
- $\mathcal{S}$ is a set of states s;
- $\mathcal{A}$ is a set of actions a;
- P is a (stationary) transition model such that $P(s' \mid s, a)$ denotes the probability of reaching state s' if action a is done in state s;
- R is a reward function that maps immediate (finite) reward values $R(s)$ obtained in states s.

Our goal is find the optimal policy $\pi^*(s)$.

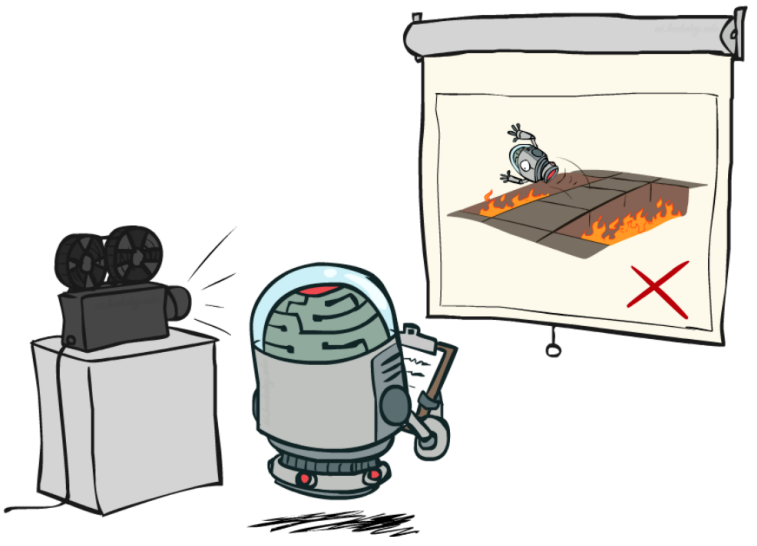
New twist

The transition model $P(s'|s,a)$ and the reward function $R(s)$ are unknown.

- We do not know which states are good nor what actions do!
- We must observe or interact with the environment in order to jointly learn these dynamics and act upon them.



Passive RL

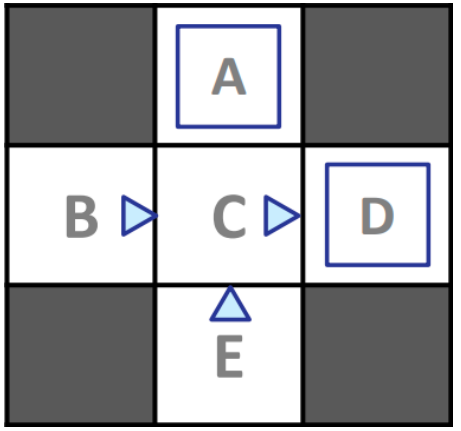


Goal: policy evaluation

- The agent's policy π is fixed.
- Its goal is to learn the utilities $V^{\pi}(s)$.
- The learner has no choice about what actions to take. It just executes the policy and learns from experience.

—

Image credits: CS188, UC Berkeley.



The agent executes a set of trials (or episodes) in the environment using policy π. Trial trajectories $(s, r, a, s'), (s', r', a', s''), \ldots$ might look like this:

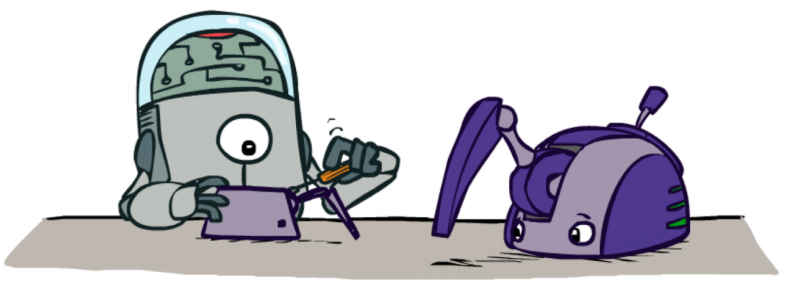
- Trial 1: $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- Trial 2: $(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- Trial 3: $(E, -1, \text{north}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
- Trial 4: $(E, -1, \text{north}, C), (C, -1, \text{east}, A), (A, -10, \text{exit}, \perp)$

Model-based estimation

A model-based agent estimates approximate transition and reward models $\hat{P}$ and $\hat{R}$ based on experiences and then evaluates the resulting empirical MDP.

- Step 1: Learn an empirical MDP.
  - Estimate $\hat{P}(s'|s,a)$ from empirical samples (s,a,s') or with supervised learning.
  - Discover each $\hat{R}(s)$ for each s.
- Step 2: Evaluate π using $\hat{P}$ and $\hat{R}$, e.g. as

$$V(s) = \hat{R}(s) + \gamma \sum_{s'} \hat{P}(s'|s,\pi(s))V(s').$$

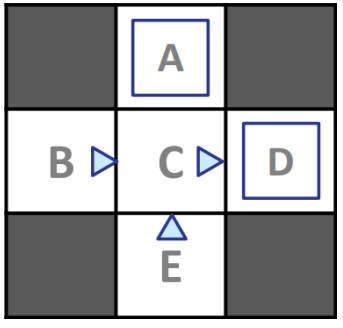


—

Image credits: CS188, UC Berkeley.

Example

Policy π:



Trajectories:

$$(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$$
$$(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$$
$$(E, -1, \text{north}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$$
$$(E, -1, \text{north}, C), (C, -1, \text{east}, A), (A, -10, \text{exit}, \perp)$$

Learned transition model $\hat{P}$:

$$\hat{P}(C|B, \text{east}) = 1$$
$$\hat{P}(D|C, \text{east}) = 0.75$$
$$\hat{P}(A|C, \text{east}) = 0.25$$
$$(...)$$

Learned reward $\hat{R}$:

$$\hat{R}(B) = -1$$
$$\hat{R}(C) = -1$$
$$\hat{R}(D) = +10$$
$$(...)$$

Model-free estimation

Can we learn V^{π} in a model-free fashion, without explicitly modeling the environment, i.e. without learning $\hat{P}$ and $\hat{R}$?

Direct utility estimation

(a.k.a. Monte Carlo evaluation)

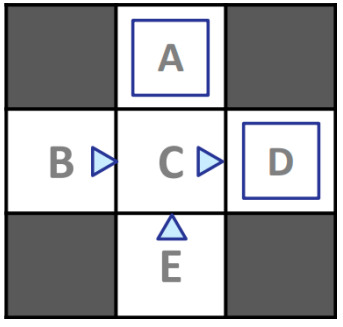
- The utility $V^\pi(s)$ of state s is the expected total reward from the state onward (called the expected reward-to-go)

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]\Bigg|_{s_0=s}$$

- Each trial provides a sample of this quantity for each state visited.
- Therefore, at the end of each sequence, one can update a sample average $\hat{V}^\pi(s)$ by:
  - computing the observed reward-to-go for each state;
  - updating the estimated utility for that state, by keeping a running average.
- In the limit of infinitely many trials, the sample average will converge to the true expectation.

Example ($\gamma = 1$)

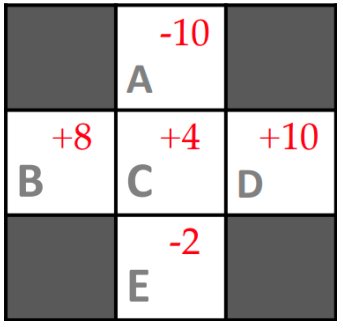
Policy π:



Trajectories:

$(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
$(B, -1, \text{east}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
$(E, -1, \text{north}, C), (C, -1, \text{east}, D), (D, +10, \text{exit}, \perp)$
$(E, -1, \text{north}, C), (C, -1, \text{east}, A), (A, -10, \text{exit}, \perp)$

Output values $\hat{V}^{\pi}(s)$:



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

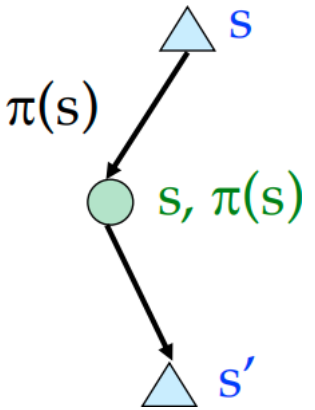
Unfortunately, direct utility estimation misses the fact that the state values $V^\pi(s)$ are not independent, since they obey the Bellman equations for a fixed policy:

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s').$$

Therefore, direct utility estimation misses opportunities for learning and takes a long time to learn.

Temporal-difference learning

Temporal-difference (TD) learning consists in updating $V^\pi(s)$ each time the agent experiences a transition $(s, r = R(s), a = \pi(s), s')$.



When a transition from s to s' occurs, the temporal-difference update steers $V^\pi(s)$ to better agree with the Bellman equations for a fixed policy, i.e.

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha \underbrace{(r + \gamma V^\pi(s') - V^\pi(s))}_{\text{temporal difference error}}$$

where α is the learning rate parameter.

Alternatively, the TD-update can be viewed as a single gradient descent step on the squared error between the target $r + \gamma V^{\pi}(s')$ and the prediction $V^{\pi}(s)$. (More later.)

Exponential moving average

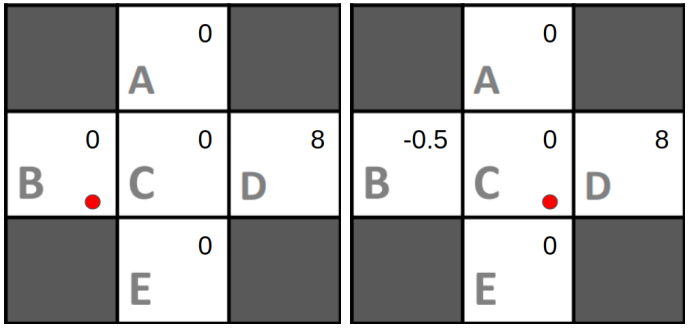
The TD-update can equivalently be expressed as the exponential moving average

$$V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + \alpha(r + \gamma V^\pi(s')).$$

Intuitively,

- this makes recent samples more important;
- this forgets about the past (distant past values were wrong anyway).

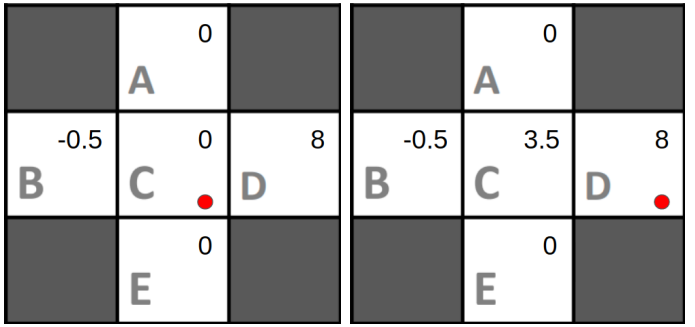
Example ($\gamma = 1, \alpha = 0.5$)



Transition: $(B, -1, \text{east}, C)$

TD-update:

$$
\begin{aligned}
V^\pi(B) &\leftarrow V^\pi(B) + \alpha(R(B) + \gamma V^\pi(C) - V^\pi(B)) \\
&\leftarrow 0 + 0.5(-1 + 0 - 0) \\
&\leftarrow -0.5
\end{aligned}
$$



Transition: $(C, -1, \text{east}, D)$

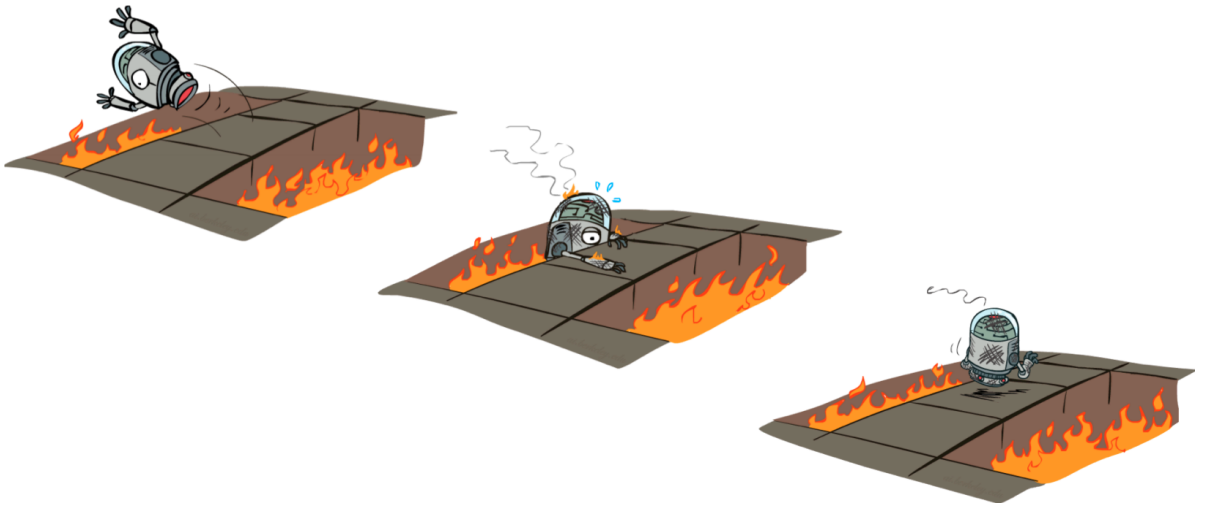
TD-update:

$$
\begin{aligned}
V^\pi(C) &\leftarrow V^\pi(C) + \alpha(R(C) + \gamma V^\pi(D) - V^\pi(C)) \\
&\leftarrow 0 + 0.5(-1 + 8 - 0) \\
&\leftarrow 3.5
\end{aligned}
$$

Convergence

- Notice that the TD-update involves only the observed successor s', whereas the actual Bellman equations for a fixed policy involves all possible next states. Nevertheless, the average value of $V^\pi(s)$ will converge to the correct value.
- If we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $V^\pi(s)$ will itself converge to the correct value.

Active RL



Goal: learn an optimal policy

- The agent's policy is not fixed anymore.
- Its goal is to learn the optimal policy π^* or the state values $V(s)$.
- The learner makes choices!
- Fundamental trade-off: exploration vs. exploitation.

—

Image credits: CS188, UC Berkeley.

Model-based learning

The passive model-based agent can be made active by instead finding the optimal policy π^* for the empirical MDP.

For example, having obtained a utility function V that is optimal for the learned model (e.g., with Value Iteration), the optimal action by one-step look-ahead to maximize the expected utility is

$$\pi^*(s) = \arg\max_a \sum_{s'} \hat{P}(s'|s,a)V(s').$$

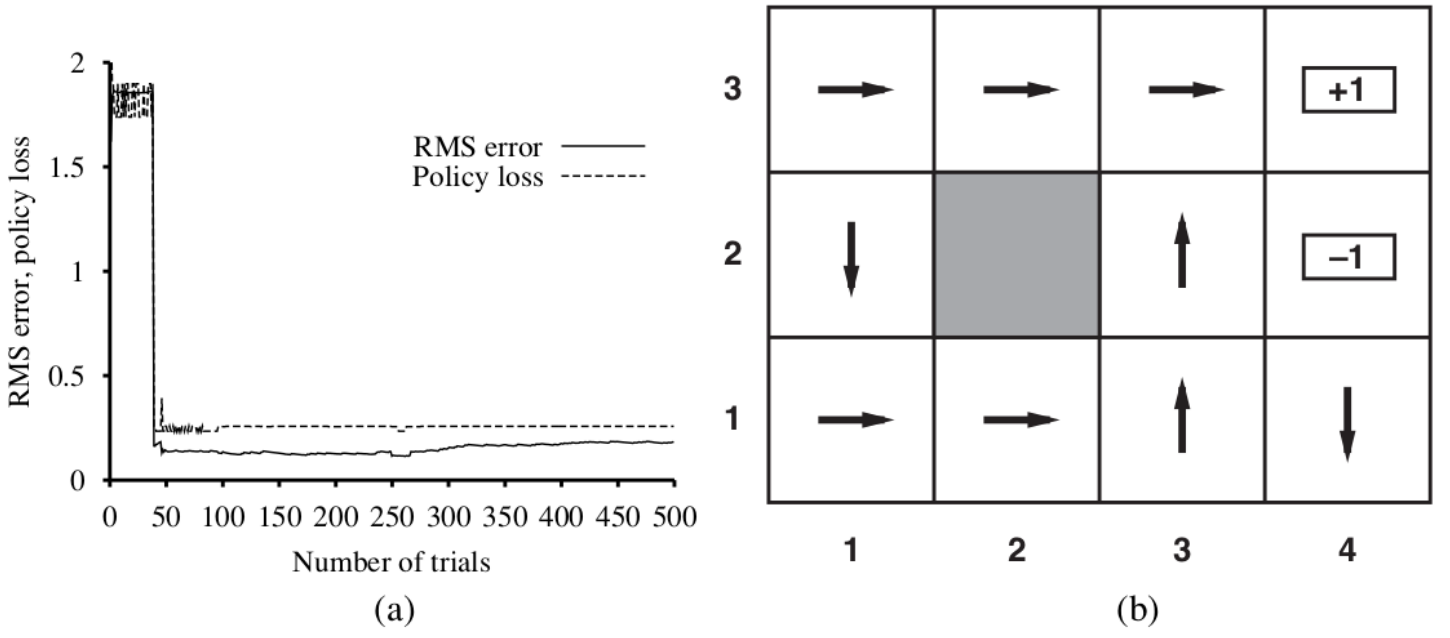


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

The agent does not learn the true utilities or the true optimal policy!

The resulting policy is greedy and suboptimal:

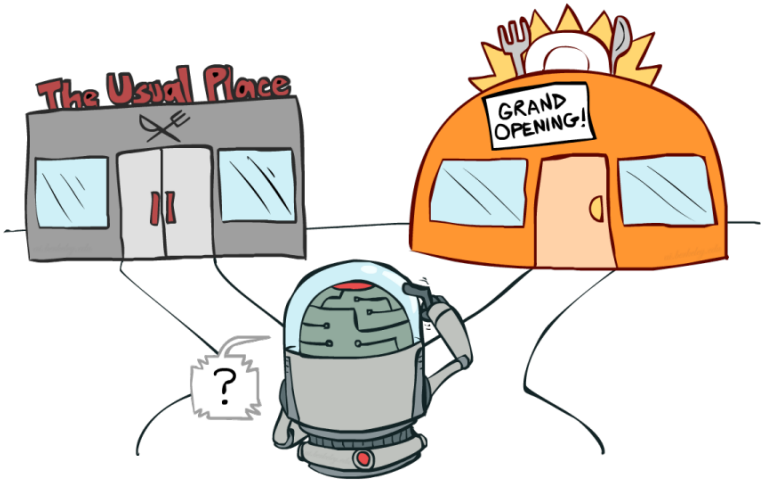
- The learned transition and reward models $\hat{P}$ and $\hat{R}$ are not the same as the true environment since they are based on the samples obtained by the agent's policy, which biases the learning.
- Therefore, what is optimal in the learned model can be suboptimal in the true environment.

Exploration

Actions do more than provide rewards according to the current learned model. They also contribute to learning the true environment.

This is the exploitation-exploration trade-off:

- Exploitation: follow actions that maximize the rewards, under the current learned model;
- Exploration: follow actions to explore and learn about the true environment.



—

Image credits: CS188, UC Berkeley.

How to explore?

Simplest approach for forcing exploration: random actions (ϵ-greedy).

- With a (small) probability ϵ, act randomly.
- With a (large) probability $(1 - \epsilon)$, follow the current policy.

ϵ-greedy does eventually explore the space, but keeps trashing around once learning is done.

When to explore?

Better idea: explore areas whose badness is not (yet) established, then stop exploring.

Formally, let $V^+(s)$ denote an optimistic estimate of the utility of state s and let $N(s,a)$ be the number of times actions a has been tried in s.

For Value Iteration, the update equation becomes

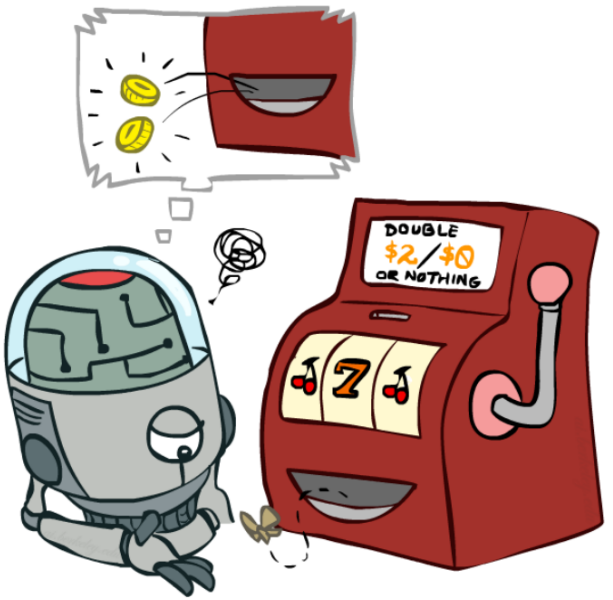
$$V^+_{i+1}(s) = R(s) + \gamma \max_a f(\sum_{s'} P(s'|s,a)V^+_i(s'), N(s,a)),$$

where $f(v,n)$ is called the exploration function.

The function $f(v,n)$ should be increasing in v and decreasing in n. A simple choice is $f(v,n) = v + K/n$.

Model-free learning

Although temporal difference learning provides a way to estimate V^{π} in a model-free fashion, we would still have to learn a model $P(s'|s,a)$ to choose an action based on a one-step look-ahead.

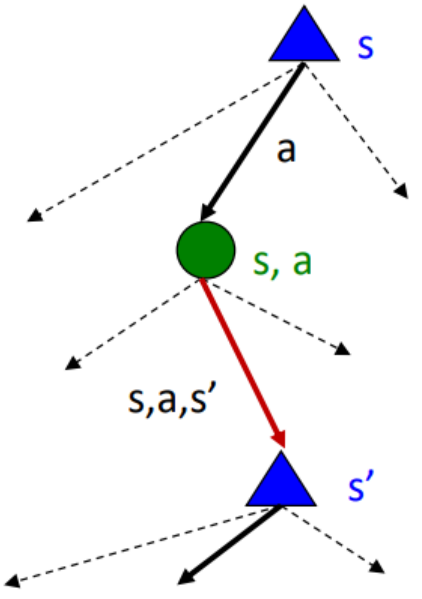


—

Image credits: CS188, UC Berkeley.

Détour: Q-values

- The state-value $V(s)$ of the state s is the expected utility starting in s and acting optimally.
- The state-action-value $Q(s, a)$ of the q-state (s, a) is the expected utility starting out having taken action a from s and thereafter acting optimally.



s is a *state*

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

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

Optimal policy

The optimal policy $\pi^*(s)$ can be defined in terms of either $V(s)$ or $Q(s, a)$:

$$\begin{aligned}\pi^*(s) &= \arg\max_a \sum_{s'} P(s'|s, a)V(s') \\ &= \arg\max_a Q(s, a)\end{aligned}$$

—

Image credits: CS188, UC Berkeley.

Bellman equations for Q

Since $V(s) = \max_a Q(s,a)$, the Q-values $Q(s,a)$ are recursively defined as

$$\begin{aligned} Q(s,a) &= R(s) + \gamma \sum_{s'} P(s'|s,a) V(s') \\ &= R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a'). \end{aligned}$$

As for value iteration, the last equation can be used as an update equation for a fixed-point iteration procedure that calculates the Q-values $Q(s,a)$. However, it still requires knowing $P(s'|s,a)$!

Q-Learning

The state-action-values $Q(s, a)$ can be learned in a model-free fashion using a temporal-difference method known as Q-Learning.

Q-Learning consists in updating $Q(s, a)$ each time the agent experiences a transition $(s, r = R(s), a, s')$.

The update equation for TD Q-Learning is

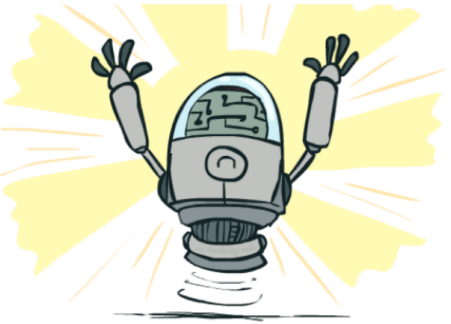
$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)).$$

Since $\pi^*(s) = \arg\max_a Q(s, a)$, a TD agent that learns Q-values does not need a model of the form $P(s'|s, a)$, neither for learning nor for action selection!

function Q-LEARNING-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$
  return a

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.



Convergence

Q-Learning converges to an optimal policy, even when acting suboptimally.

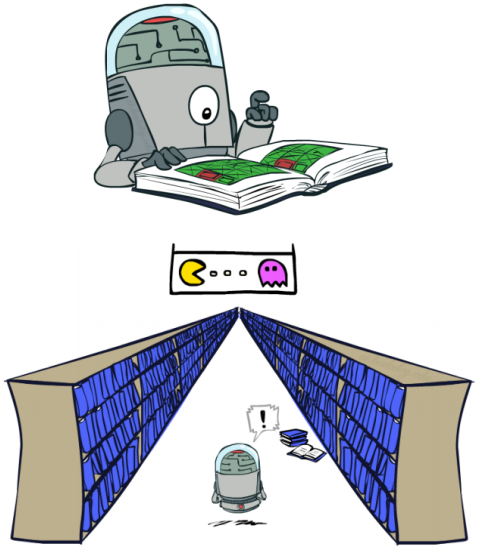
- This is called off-policy learning.
- Technical caveats:
  - You have to explore enough.
  - The learning rate must eventually become small enough.
  - ... but it shouldn't decrease too quickly.

—

Image credits: CS188, UC Berkeley.

Generalizing across states

- Basic Q-Learning keeps a table for all Q-values $Q(s, a)$.
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training.
  - Too many states to hold the Q-table in memory.
- We want to generalize:
  - Learn about some small number of training states from experience.
  - Generalize that experience to new, similar situations.
  - This is supervised machine learning again!

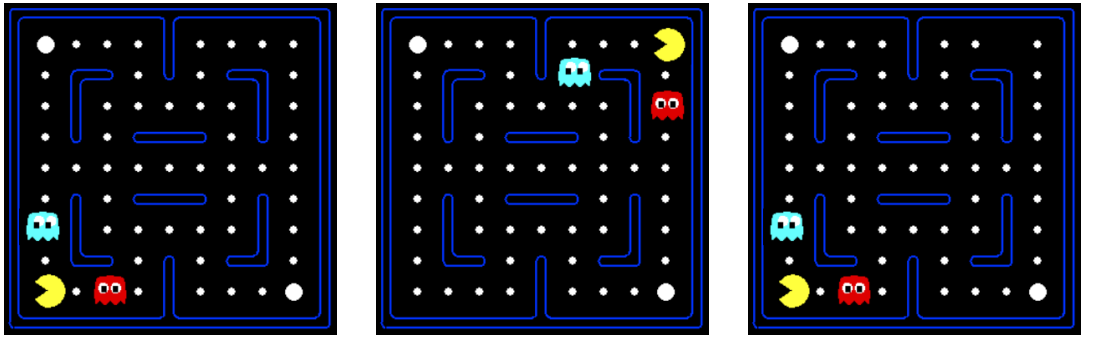


—

Image credits: CS188, UC Berkeley.

Example: Pacman

(a) (b) (c)

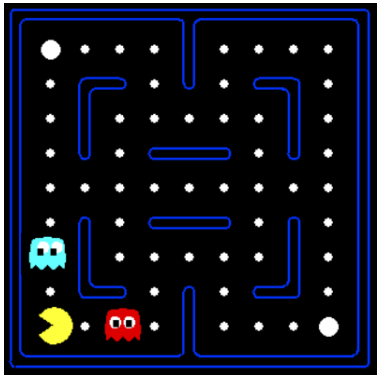


If we discover by experience that (a) is bad, then in naive Q-Learning, we know nothing about (b) nor (c)!

—

Image credits: CS188, UC Berkeley.

Feature-based representations

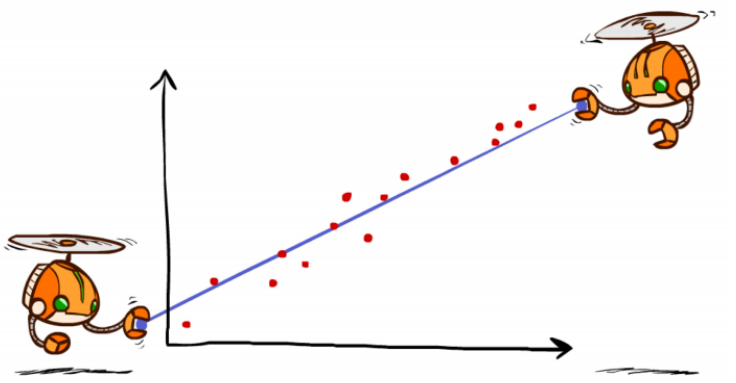


Solution: describe a state s using a vector $\mathbf{x} = [f_1(s), ..., f_d(s)] \in \mathbb{R}^d$ of features.

- Features are functions f_k from states to real numbers that capture important properties of the state.
- Example features:
  - Distance to closest ghost
  - Distance to closest dot
  - Number of ghosts
  - ...
- Can similarly describe a q-state (s, a) with features $f_k(s, a)$.

—

Image credits: CS188, UC Berkeley.



Approximate Q-Learning

Using a feature-based representation, the Q-table can now be replaced with a function approximator, such as a linear model

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_d f_d(s,a).$$

Upon the transition (s, r, a, s'), the update becomes

$$w_k \leftarrow w_k + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s,a)) f_k(s,a),$$

for all w_k.

—

Image credits: CS188, UC Berkeley.

In linear regression, imagine we had only one point $\mathbf{x}$ with features $[f_1, \ldots, f_d]$. Then,

$$\ell(\mathbf{w}) = \frac{1}{2}\left(y - \sum_k w_k f_k\right)^2$$

$$\frac{\partial \ell}{\partial w_k} = -\left(y - \sum_k w_k f_k\right) f_k$$

$$w_k \leftarrow w_k + \alpha \left(y - \sum_k w_k f_k\right) f_k,$$

hence the Q-update

$$w_k \leftarrow w_k + \alpha \left(\underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{target } y} - \underbrace{Q(s, a)}_{\text{prediction}} \right) f_k(s, a).$$

DQN

Similarly, the Q-table can be replaced with a neural network as function approximator, resulting in the DQN algorithm.

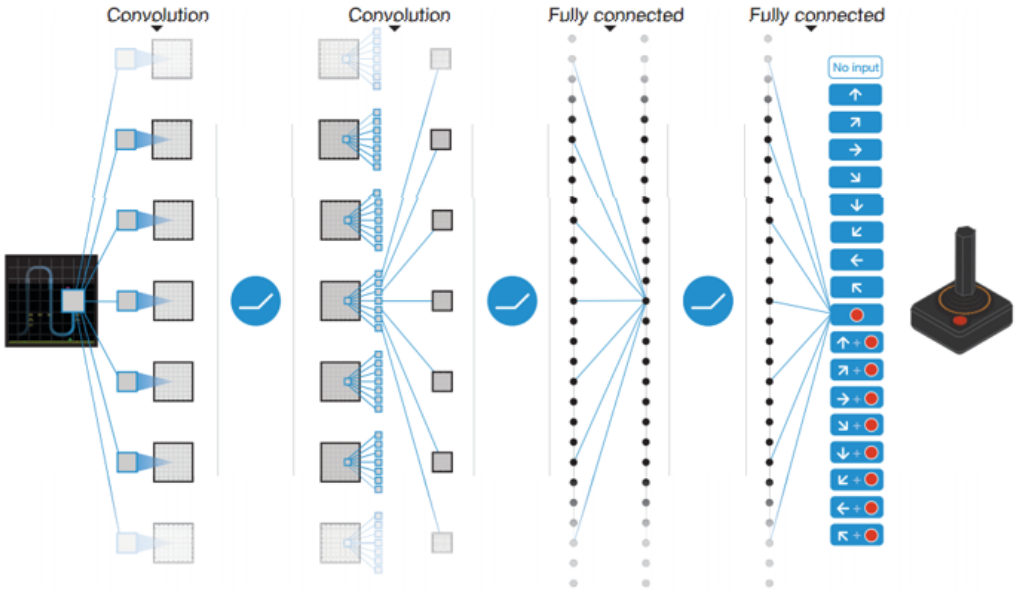
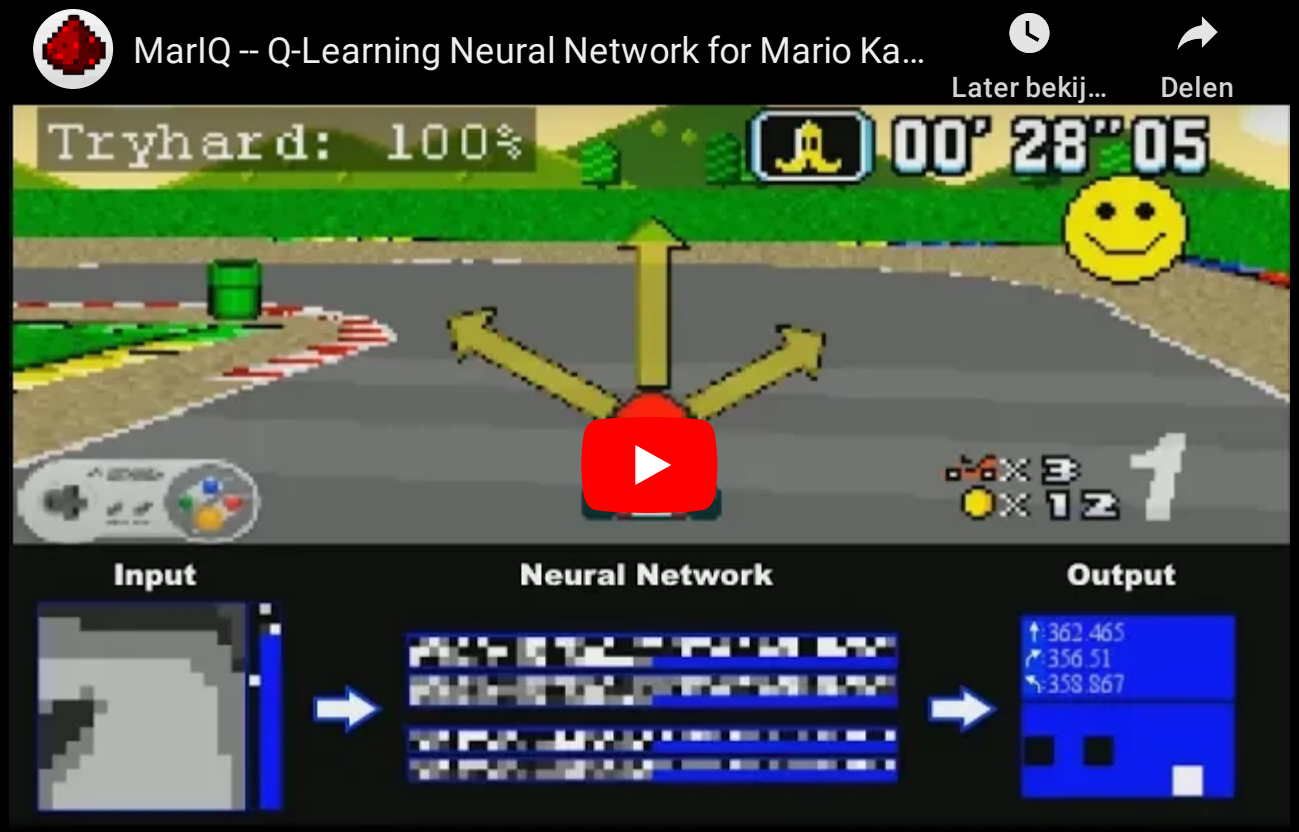


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0,x)$).

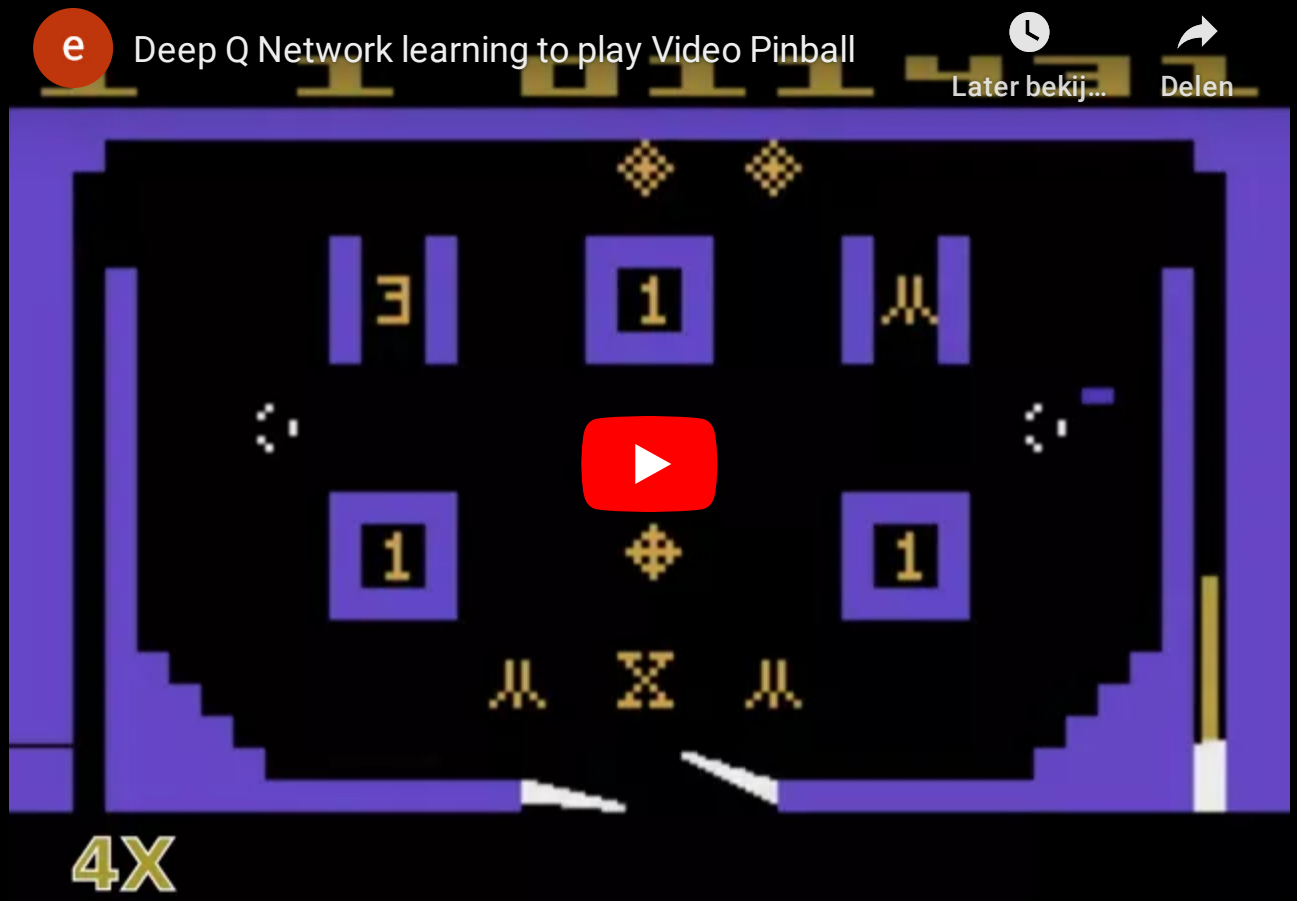
DQN Network Architecture

(demo)

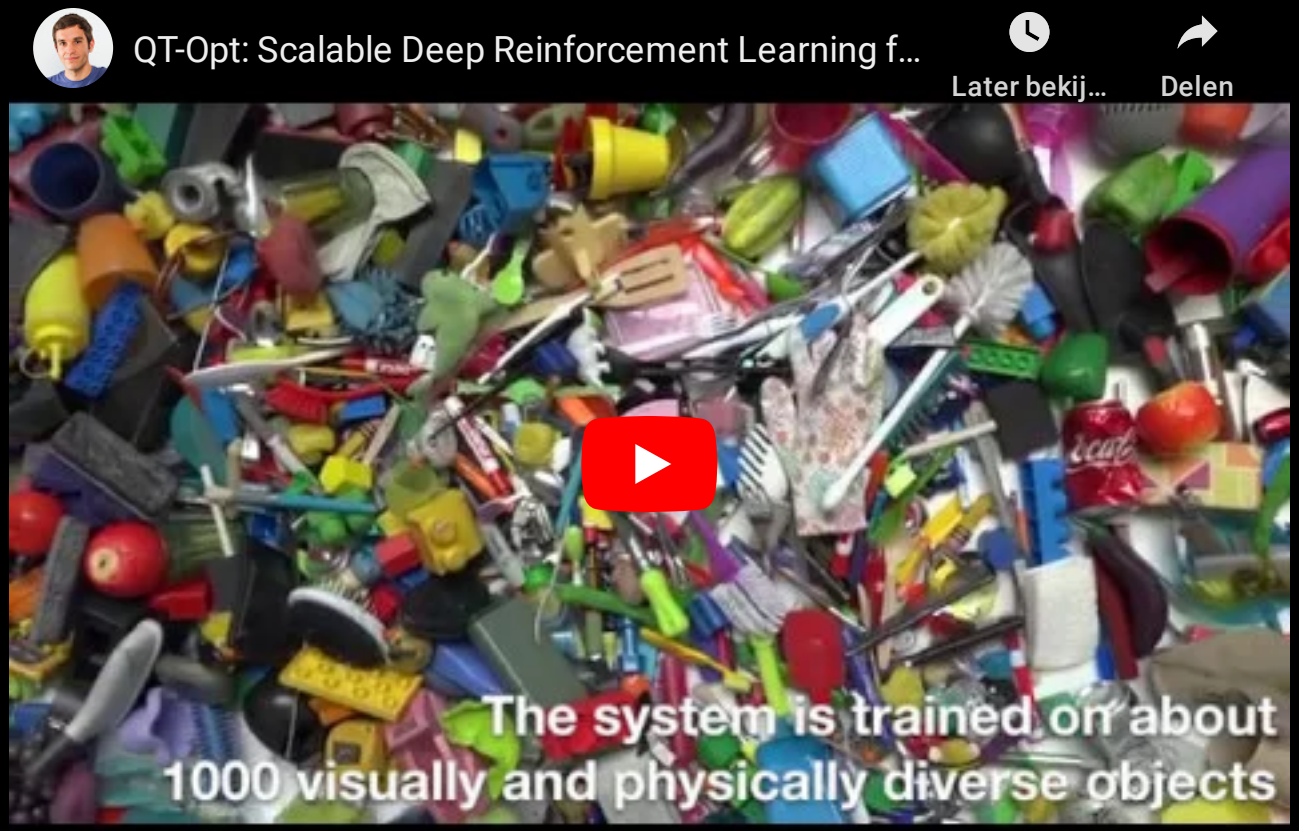
Applications


MarIQ -- Q-Learning Neural Network for Mario Ka...
Later bekij...
Delen
Tryhard: 100%
00' 28"05
Input
Neural Network
Output
362.465
356.51
358.867

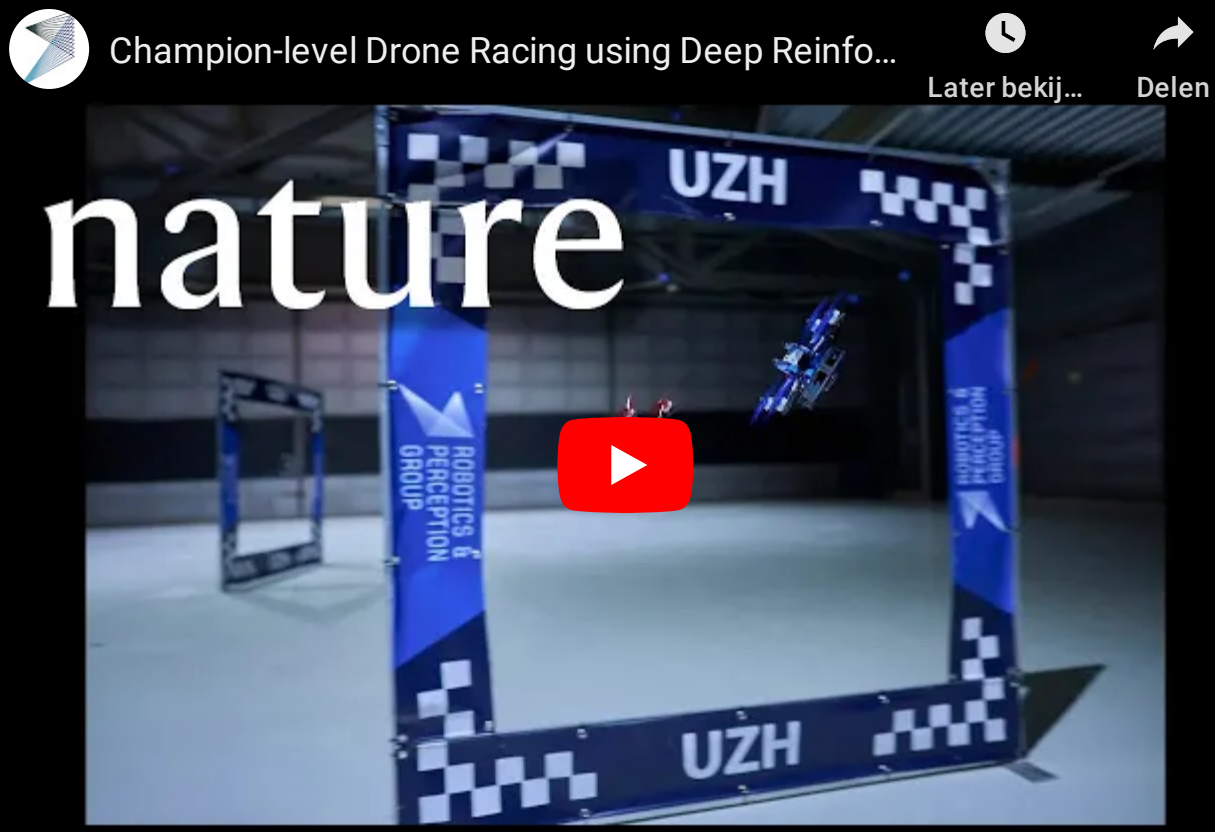
MarIQ



Playing Atari Games (Pinball)



Robotic manipulation



Drone racing

Summary

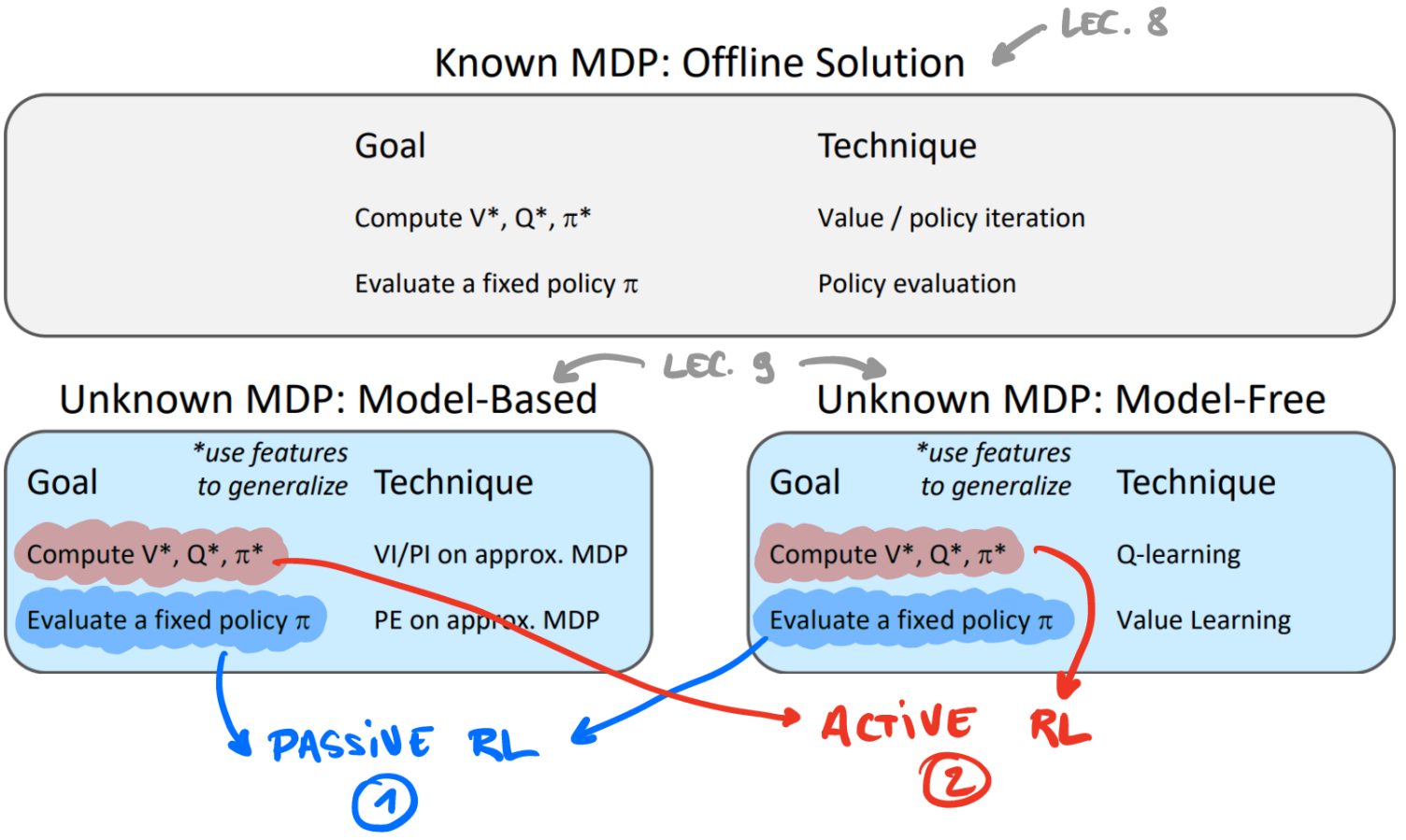


Image credits: CS188, UC Berkeley.

My mission ✔

By the end of this course, you will have built autonomous agents that efficiently make decisions in fully informed, partially observable and adversarial settings. Your agents will draw inferences in uncertain and unknown environments and optimize actions for arbitrary reward structures.

The end.