Reinforcement learning for life-long microgrid control

An introduction

Bertrand Cornélusse
bertrand.cornelusse@uliege.be

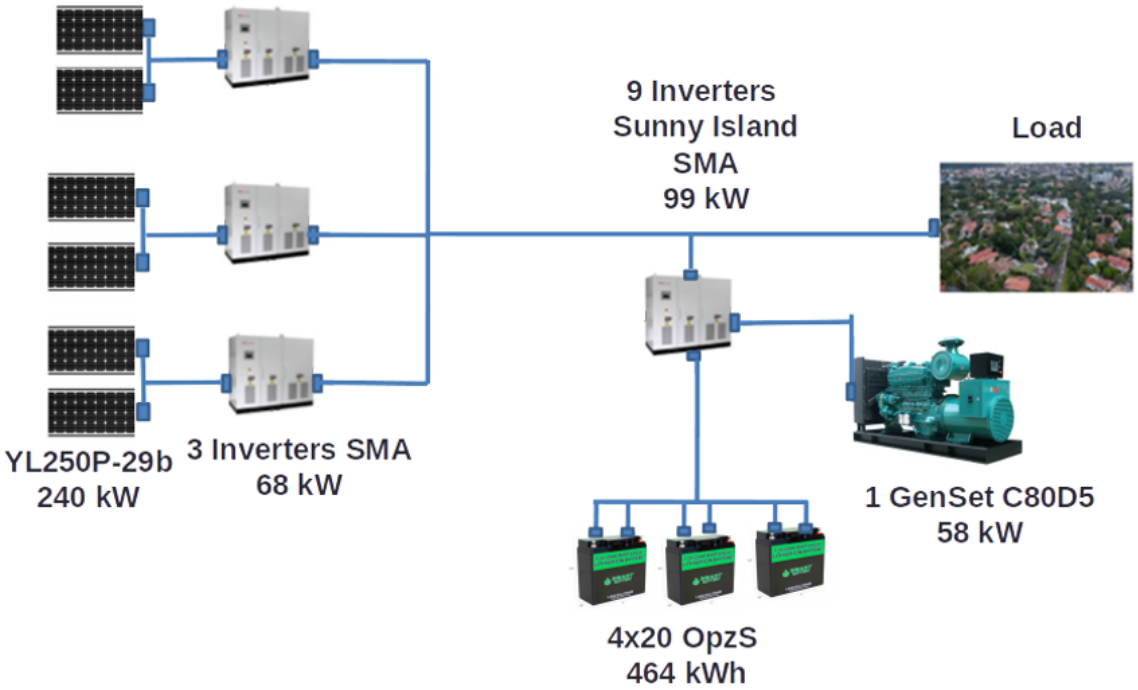
LIÈGE université

Lecture organization

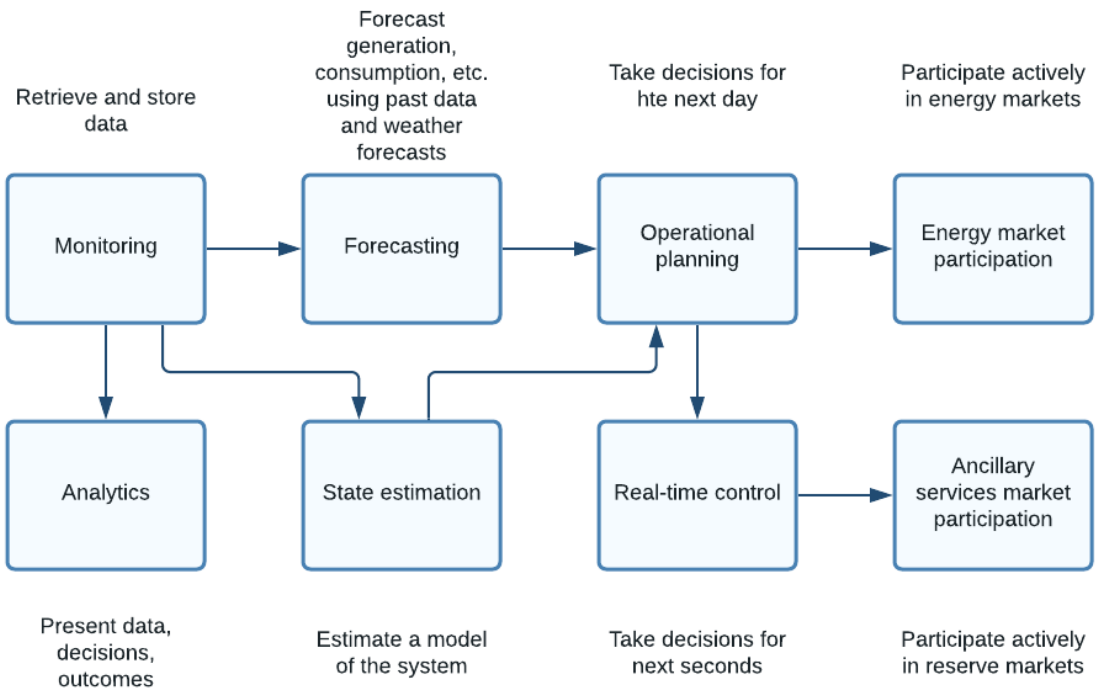
- Motivation
- Markov Decision Processes
- Reinforcement Learning
- Lifelong learning for microgrids control

Motivation

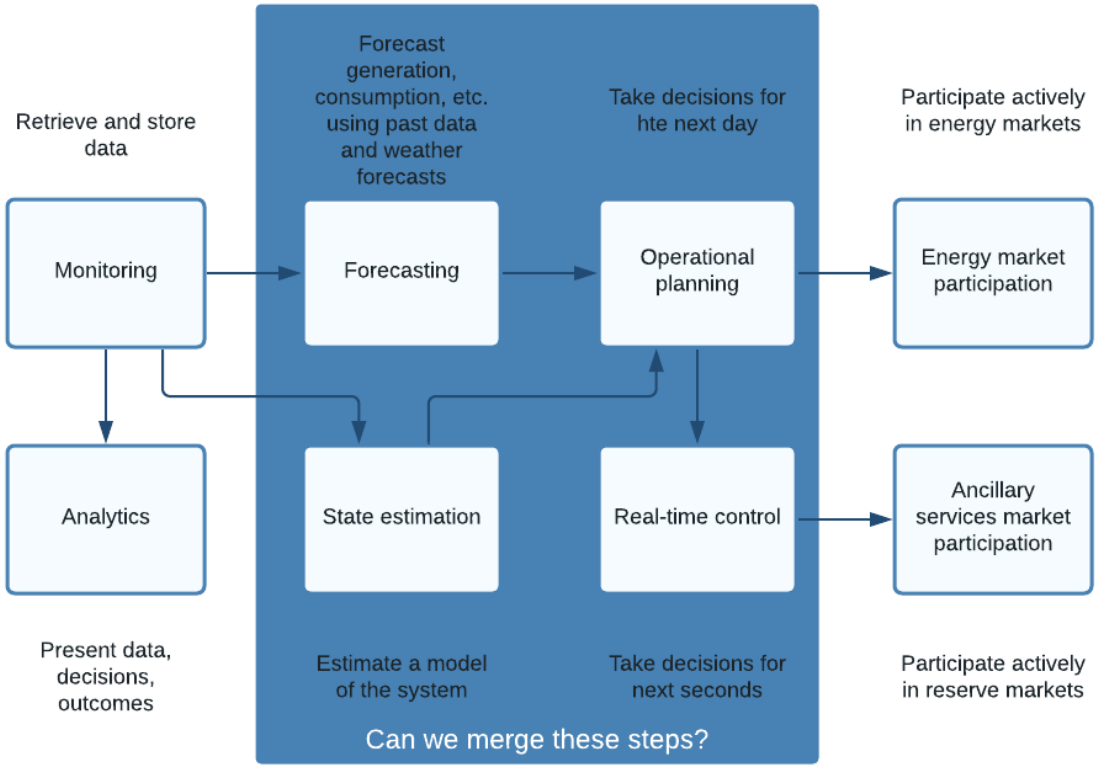
Microgrid (MG) example



Energy management principles



Let's bring in more AI



Why? To further improve automation, especially for small scale grids that evolve a lot with time.

Goal of this lecture

Assuming you remember notions of optimization and supervised learning, make you understand the concepts behind the ideas developed in the article

Totaro, S., Boukas, I., Jonsson, A., & Cornélusse, B. (2021). Lifelong Control of Off-grid Microgrid with Model-Based Reinforcement Learning. Energy, 121035.

- This article proposes a method to learn a model of the MG and its environment
  - PV conditions, demand evolution
- uses a simulator to generate data
  - to avoid relying solely on the actual system
  - freely available at https://github.com/bcornelusse/microgridRLsimulator
- uses the learned model to take meta actions
  - start-stop a generator, charge or discharge a battery
  - relying on simple rules to derive precise set-points.

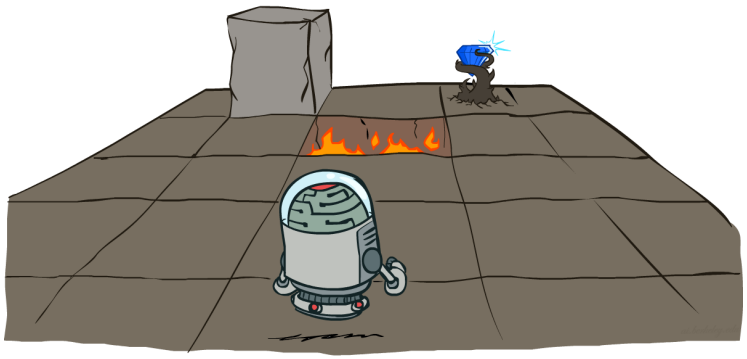
The learned policy is able to cope with abrupt and gradual changes of the MG and its environment.

To do this I have to introduce concepts of reinforcement learning, hence also of Markov decision processes.

Markov decision processes

This section is a subset of the lecture 8 of the AI course by my colleague Professor G. Louppe.

Content



Reasoning under uncertainty and taking decisions:

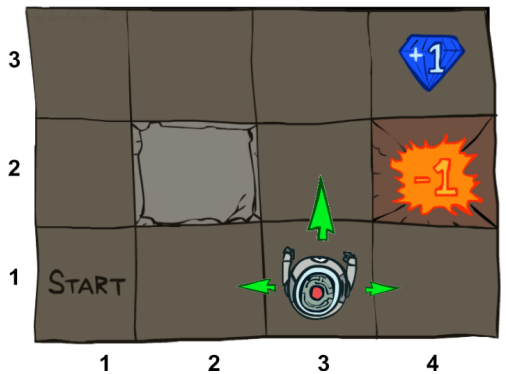
- Markov decision processes
  - MDPs
  - Bellman equation
  - Value iteration
  - Policy iteration

Image credits: CS188, UC Berkeley.

Grid world

Assume our agent lives in a 4×3 grid environment.

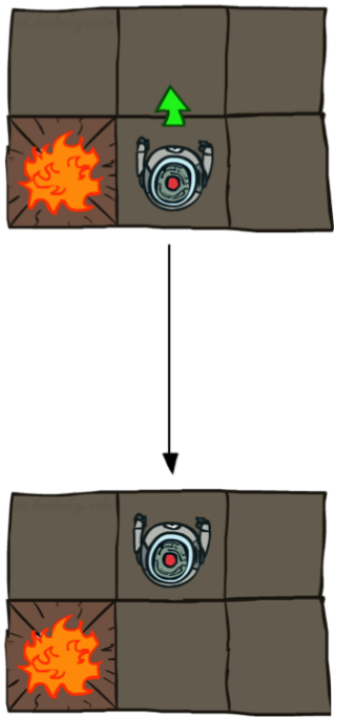
- Noisy movements: actions do not always go as planned.
  - Each action achieves the intended effect with probability 0.8.
  - The rest of the time, with probability 0.2, the action moves the agent at right angles to the intented direction.
  - If there is a wall in the direction the agent would have been taken, the agent stays put.
- The agent receives rewards at 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.

Image credits: CS188, UC Berkeley.

Deterministic actions



Stochastic actions

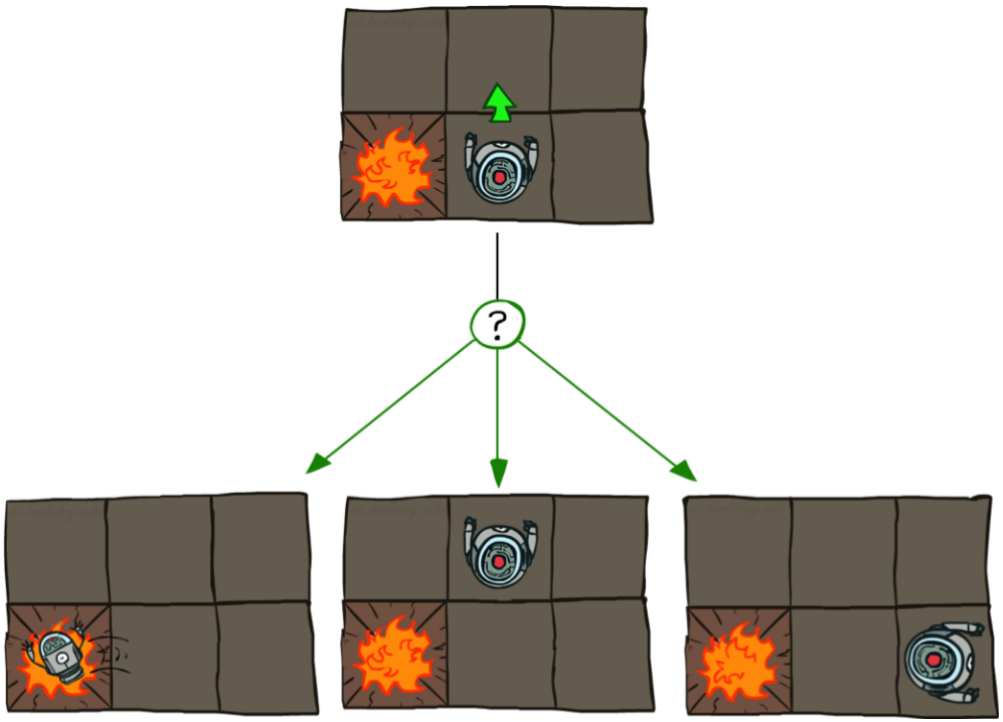


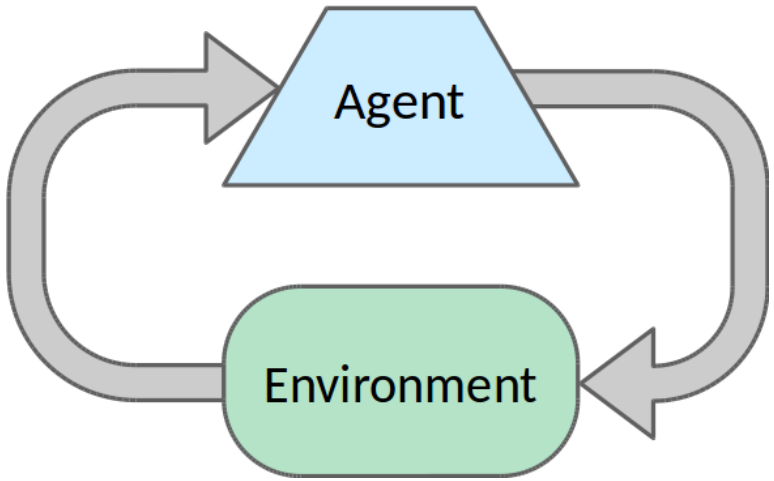
Image credits: CS188, UC Berkeley.

Markov decision processes

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.

s

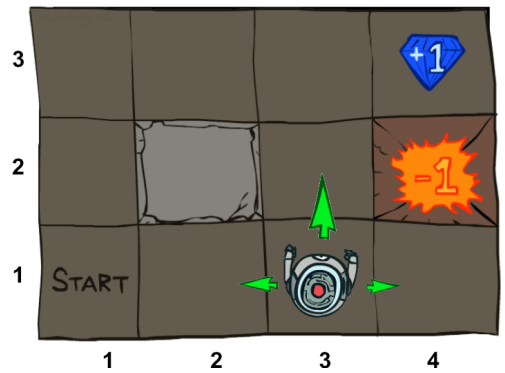


s'

$r' = R(s')$

a

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



Example

- $\mathcal{S}$: locations (i, j) on the grid.
- $\mathcal{A}$: $[\text{Up}, \text{Down}, \text{Right}, \text{Left}]$.
- Transition model: $P(s'|s, a)$
- Reward:

$$R(s) = \begin{cases} -0.3 & \text{for non-terminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

Image credits: CS188, UC Berkeley.

What is Markovian about MDPs?

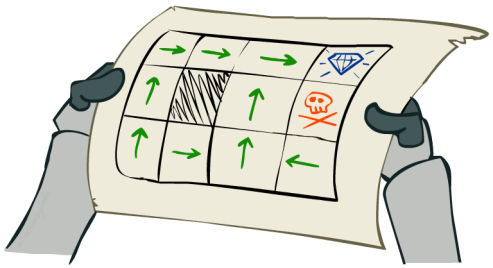
Given the present state, the future and the past are independent:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0) = P(s_{t+1}|s_t, a_t)$$

Andrey Markov

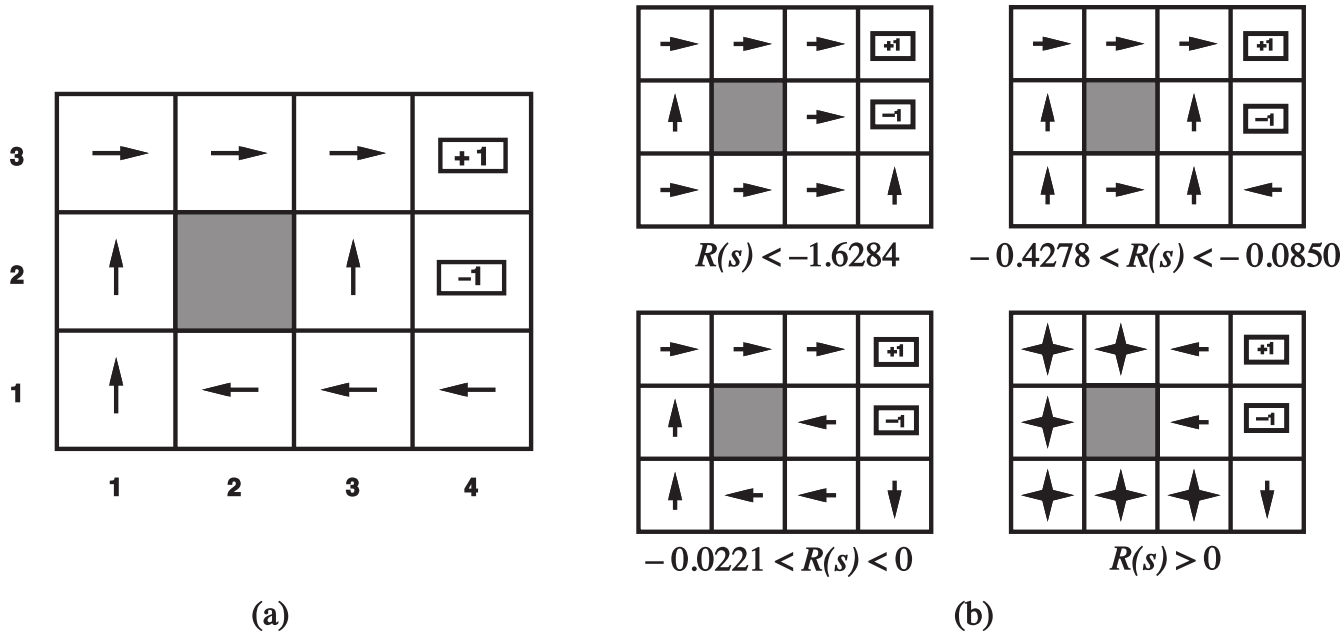
Policies

- We want to find an optimal policy $\pi^* : \mathcal{S} \to \mathcal{A}$.
  - A policy π maps states to actions.
  - An optimal policy is one that maximizes the expected utility, e.g. the expected sum of rewards.
  - An explicit policy defines a reflex agent.



Optimal policy when $R(s) = -0.3$ for all non-terminal states s.

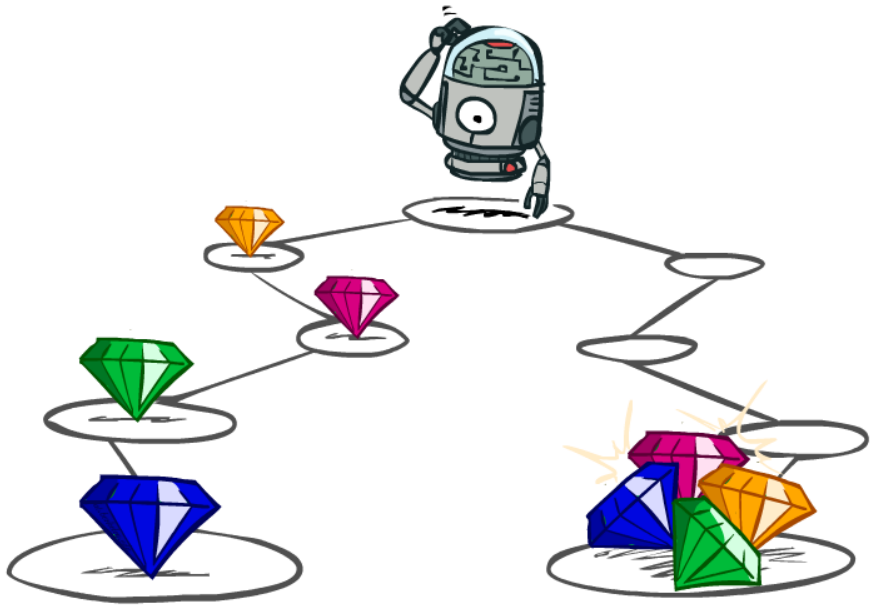
Image credits: CS188, UC Berkeley.



(a) Optimal policy when $R(s) = -0.04$ for all non-terminal states s. (b) Optimal policies for four different ranges of $R(s)$.

Depending on $R(s)$, the balance between risk and reward changes from risk-taking to very conservative.

Utilities over time



What preferences should an agent have over state or reward sequences?

- More or less? $[2, 3, 4]$ or $[1, 2, 2]$?
- Now or later? $[1, 0, 0]$ or $[0, 0, 1]$?

Image credits: CS188, UC Berkeley.

Theorem

If we assume stationary preferences over reward sequences, i.e. such that

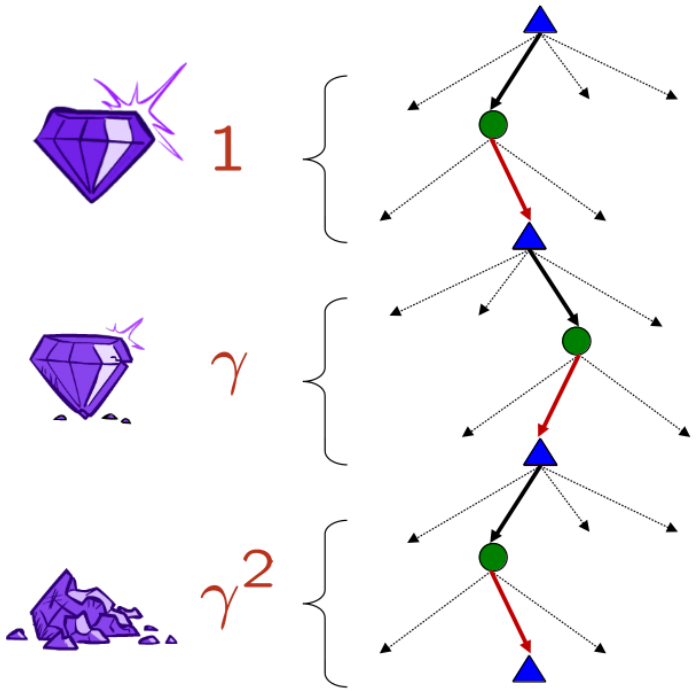
$$[r_0, r_1, r_2, ...] \succ [r_0, r'_1, r'_2, ...] \Rightarrow [r_1, r_2, ...] \succ [r'_1, r'_2, ...],$$

then there are only two coherent ways to assign utilities to sequences:

Additive utility:	$V([r_0, r_1, r_2, ...]) = r_0 + r_1 + r_2 + ...$
Discounted utility: ($0 < \gamma < 1$)	$V([r_0, r_1, r_2, ...]) = r_0 + \gamma r_1 + \gamma^2 r_2 r + ...$

Discounting

- Each time we transition to the next state, we multiply in the discount once.
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards.
  - Will help our algorithms converge.



Example: discount $\gamma = 0.5$

- $V([1,2,3]) = 1 + 0.5 \times 2 + 0.25 \times 3$
- $V([1,2,3]) < V([3,2,1])$

Image credits: CS188, UC Berkeley.

Infinite sequences

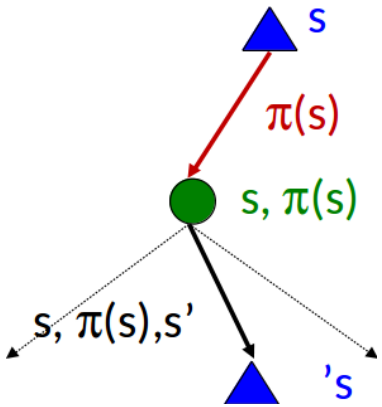
What if the agent lives forever? Do we get infinite rewards? Comparing reward sequences with $+\infty$ utility is problematic.

Solutions:

- Finite horizon: (similar to depth-limited search)
  - Terminate episodes after a fixed number of steps T.
  - Results in non-stationary policies (π depends on time left).
- Discounting (with $0 < \gamma < 1$ and rewards bounded by $\pm R_{\max}$):

$$V([r_0, r_1, ..., r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq \frac{R_{\max}}{1-\gamma}$$

  Smaller γ results in a shorter horizon.
- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached.



Policy evaluation

The expected utility obtained by executing π starting in s is given by

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

where the expectation is with respect to the probability distribution over state sequences determined by s and π.

Image credits: CS188, UC Berkeley.

Optimal policies

Among all policies the agent could execute, the optimal policy is the policy π_s^* that maximizes the expected utility:

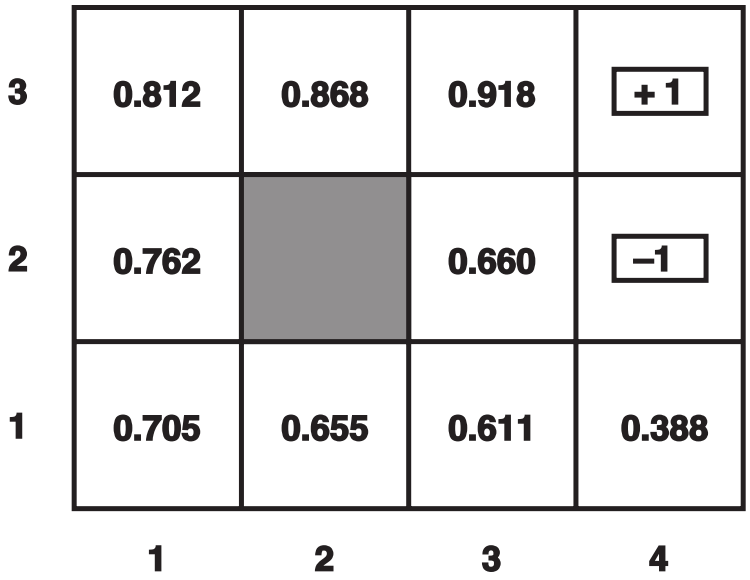
$$\pi_s^* = \arg\max_\pi V^\pi(s)$$

Because of discounted utilities, the optimal policy is independent of the starting state s. Therefore we simply write π^*.

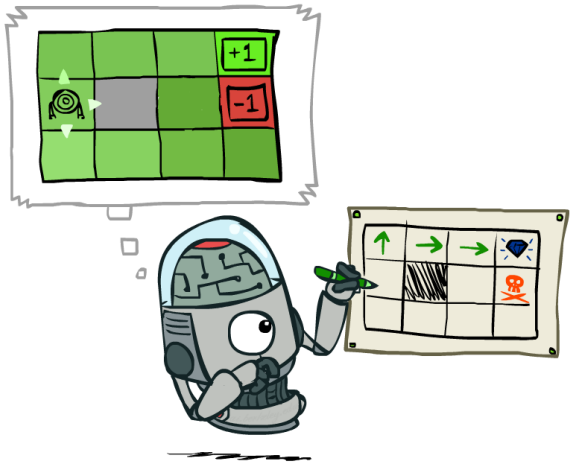
Values of states

The utility, or value, $V(s)$ of a state is now simply defined as $V^{\pi^*}(s)$.

- That is, the expected (discounted) reward if the agent executes an optimal policy starting from s.
- Notice that $R(s)$ and $V(s)$ are quite different quantities:
  - $R(s)$ is the short term reward for having reached s.
  - $V(s)$ is the long term total reward from s onward.



Utilities of the states in Grid World, calculated with $\gamma = 1$ and $R(s) = -0.04$ for non-terminal states.



Policy extraction

Using the principle of maximum expected utility, the optimal action maximizes the expected utility of the subsequent state. That is,

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

Therefore, we can extract the optimal policy provided we can estimate the utilities of states.

Image credits: CS188, UC Berkeley.

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

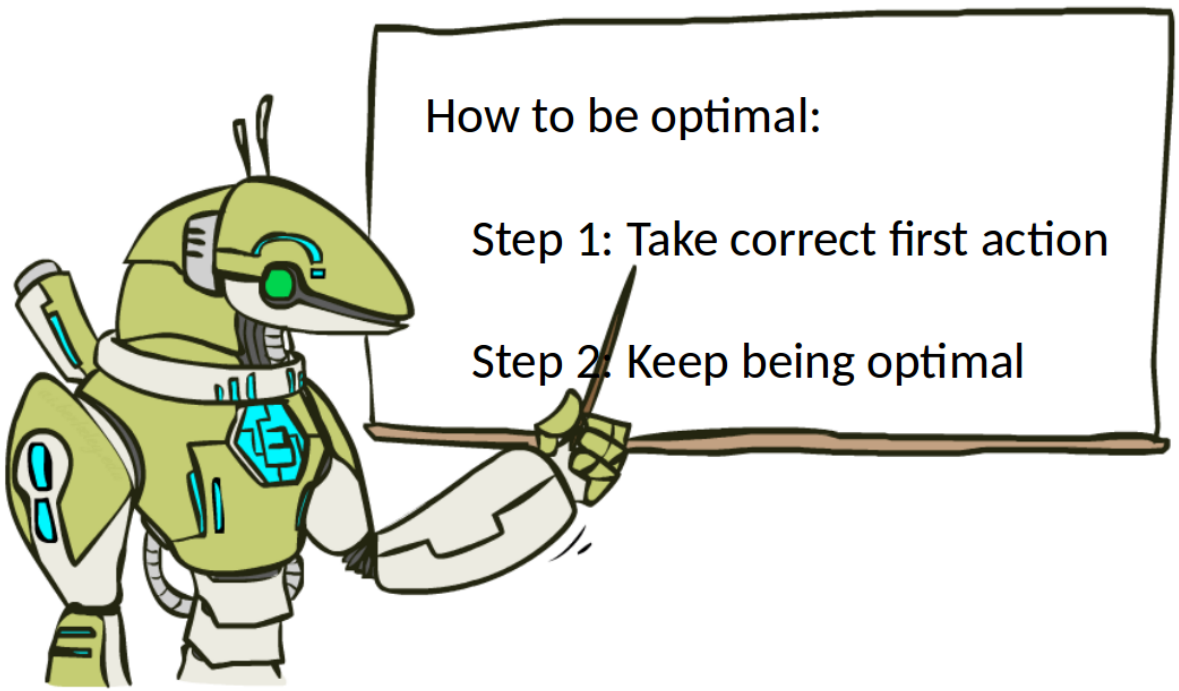


Image credits: CS188, UC Berkeley.

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:

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

- These equations are called the Bellman equations. They form a system of $n = |\mathcal{S}|$ non-linear equations with as many unknowns.
- The utilities of states, defined as the expected utility of subsequent state sequences, are solutions of the set of Bellman equations.

Example

$$\begin{aligned} V(1,1) = -0.04 + \gamma \max[&0.8V(1,2) + 0.1V(2,1) + 0.1V(1,1), \\ &0.9V(1,1) + 0.1V(1,2), \\ &0.9V(1,1) + 0.1V(2,1), \\ &0.8V(2,1) + 0.1V(1,2) + 0.1V(1,1)] \end{aligned}$$

Value iteration

Because of the $\max$ operator, the Bellman equations are non-linear and solving the system is problematic.

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.

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
rewards $R(s)$, discount γ
ϵ, the maximum error allowed in the utility of any state
local variables: U, U', vectors of utilities for states in S, initially zero
δ, the maximum change in the utility of any state in an iteration

repeat
$U \leftarrow U'$; $\delta \leftarrow 0$
for each state s **in** S **do**
$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
until $\delta < \epsilon(1-\gamma)/\gamma$
return U

Convergence

Let V_i and V_{i+1} be successive approximations to the true utility V.

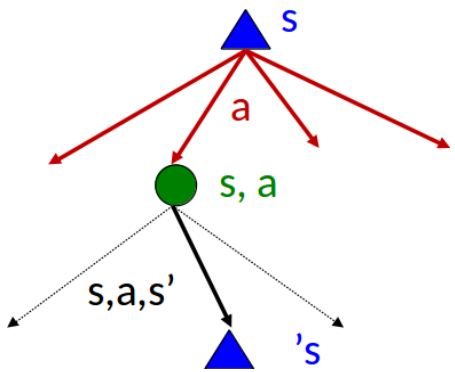
Theorem. For any two approximations V_i and V'_i,

$$||V_{i+1} - V'_{i+1}||_\infty \leq \gamma ||V_i - V'_i||_\infty.$$

- The Bellman update is a contraction by γ on the space of utility vector.
- Therefore, any two approximations must get closer to each other, and in particular any approximation must get closer to the true V.

$\Rightarrow$ Value iteration always converges to a unique solution of the Bellman equations whenever $\gamma < 1$.

Problems with value iteration



Value iteration converges exponentially fast since $||V_{i+1} - V||_\infty \leq \gamma ||V_i - V||_\infty$, i.e. the error is reduced by a factor of at least γ at each iteration, but it repeats the Bellman updates:

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

- Problem 1: it is slow – $O(|\mathcal{S}|^2 |\mathcal{A}|)$ per iteration.
- Problem 2: the $\max$ at each state rarely changes.
- Problem 3: the policy π_i extracted from the estimate V_i might be optimal even if V_i is inaccurate!

Policy iteration

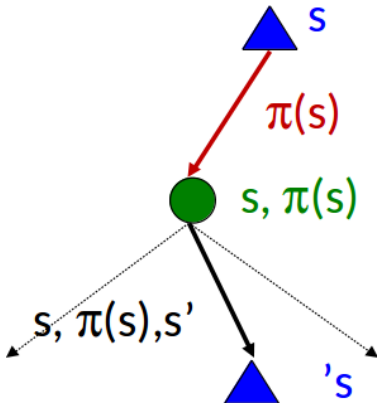
The policy iteration algorithm instead directly computes the policy. μ

It alternates the following two steps:

1. Policy evaluation: given π_i, calculate $V_i = V^{\pi_i}$, i.e. the utility of each state if π_i is executed.
2. Policy improvement: calculate a new policy π_{i+1} using one-step look-ahead based on V_i:

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

This algorithm is still optimal, and might converge (much) faster under some conditions.



Policy evaluation

At the i-th iteration we have a simplified version of the Bellman equations that relate the utility of s to the utilities of its neighbors:

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

These equations are now linear because the $\max$ operator has been removed.

- for n states, we have n equations with n unknowns;
- this can be solved exactly in $O(n^3)$ by standard linear algebra methods.

In some cases $O(n^3)$ is too prohibitive. Fortunately, it is not necessary to perform exact policy evaluation. An approximate solution is sufficient.

One way is to run k iterations of simplified Bell updates:

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

This hybrid algorithm is called modified policy iteration.

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

Reinforcement learning

This section is a subset of the lecture 9 of the AI course by my colleague Professor G. Louppe.

Content

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.

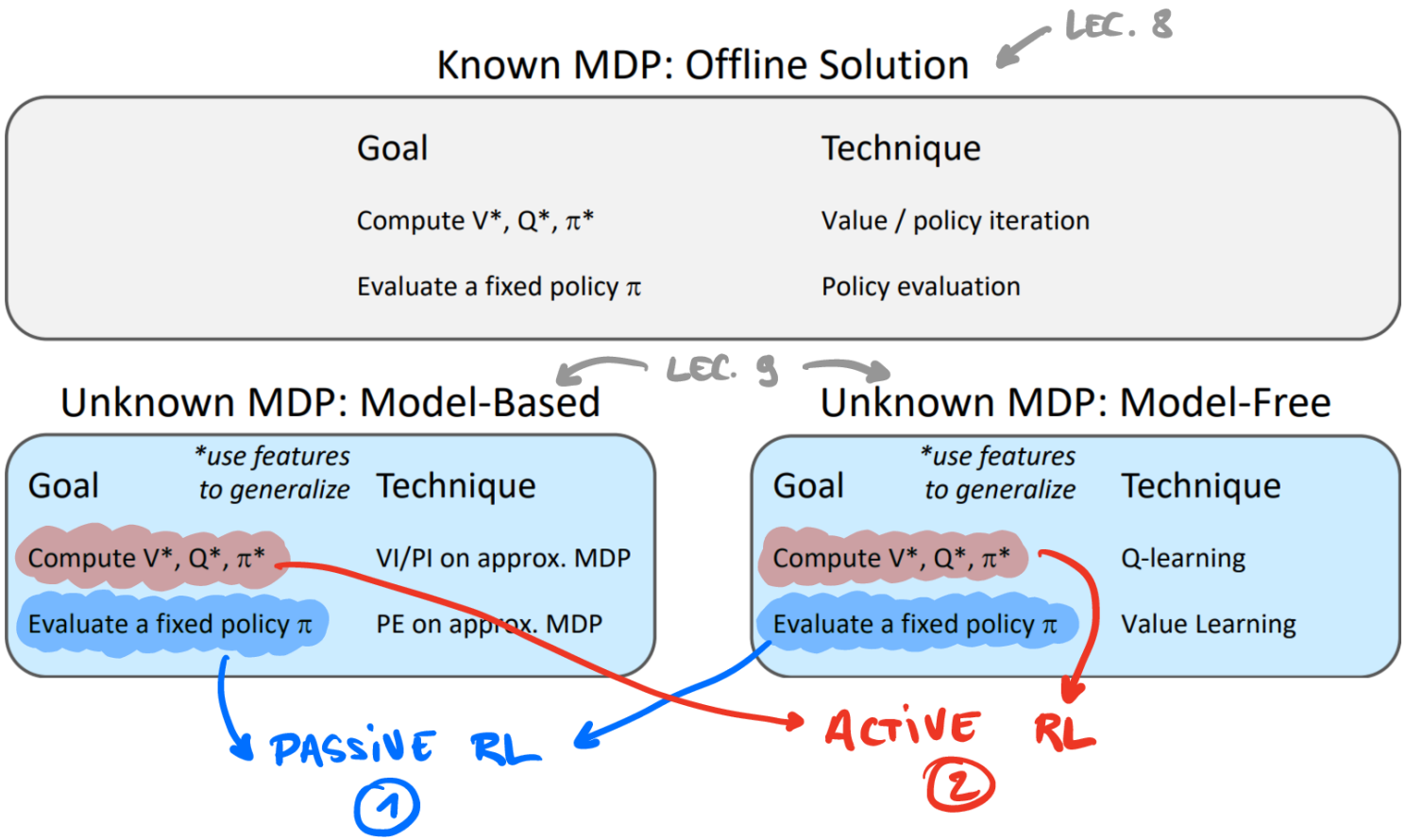
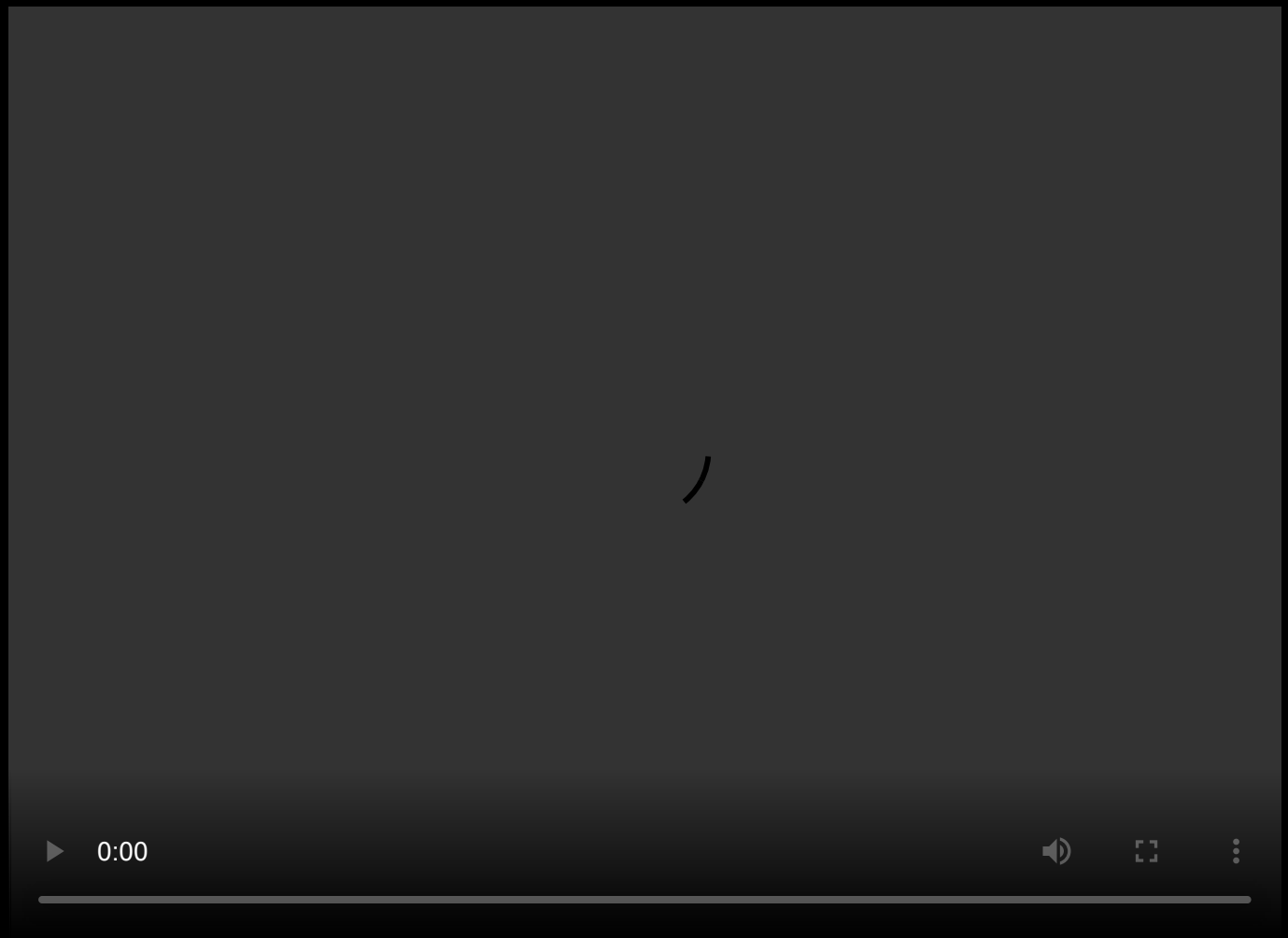


Image credits: CS188, UC Berkeley.

Remark

- Although MDPs generalize to continuous state-action spaces, we assume in this lecture that both $\mathcal{S}$ and $\mathcal{A}$ are discrete and finite.



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:

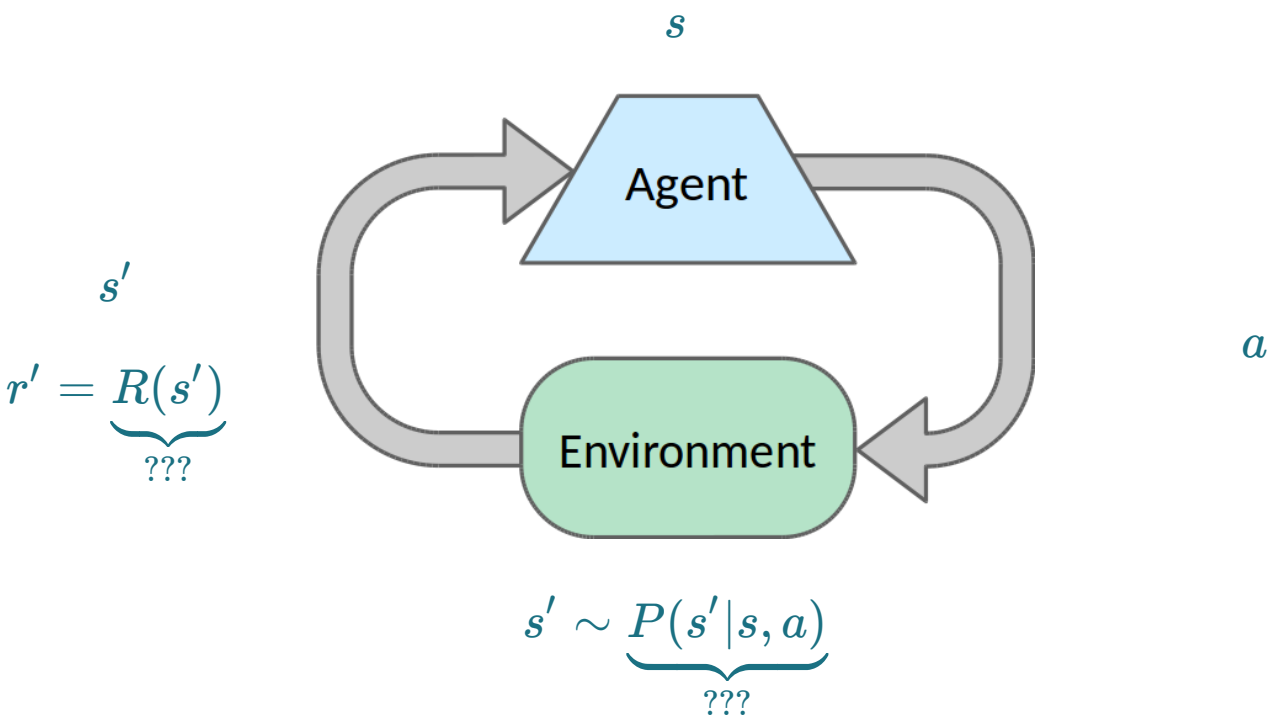
- $\mathcal{S}$ is a set of states s;
- $\mathcal{A}$ is a set of actions a;
- P is a (stationary) transition model such that $P(s' \mid s, a)$ denotes the probability of reaching state s' if action a is done in state s;
- R is a reward function that maps immediate (finite) reward values $R(s)$ obtained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

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

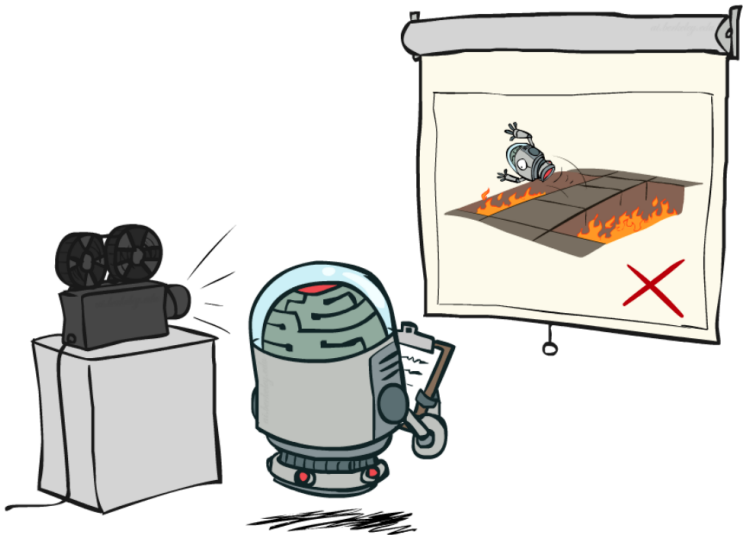
New twist

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.



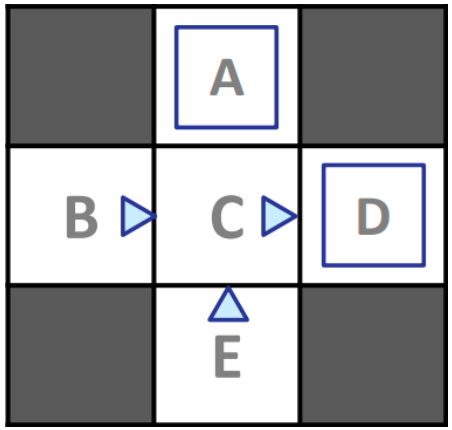
Passive RL



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') (e.g with supervised learning).
  - Discover each $\hat{R}(s)$ for each s.
- Step 2: Evaluate π using $\hat{P}$ and $\hat{R}$, e.g. as

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

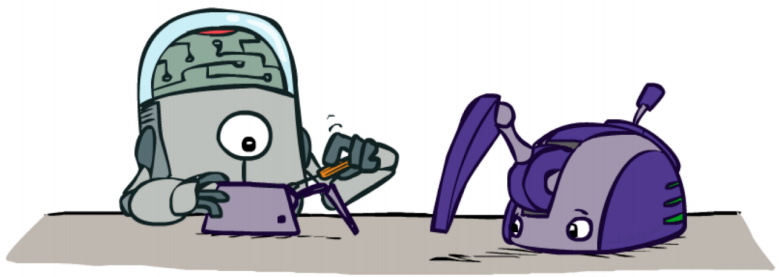
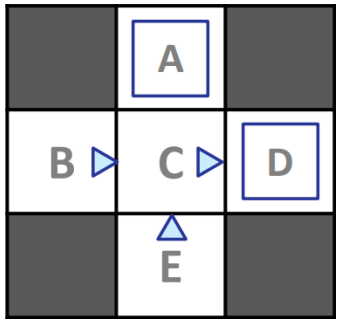


Image credits: CS188, UC Berkeley.

Example

Policy π:



Trajectories:

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

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)

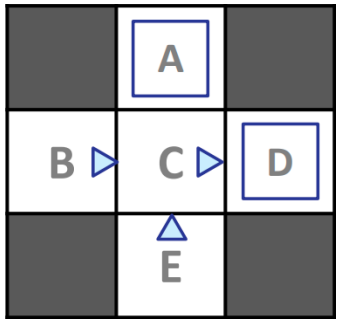
- The utility $V^\pi(s)$ of state s is the expected total reward from the state onward (called the expected reward-to-go)

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

- 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 ($\gamma = 1$)

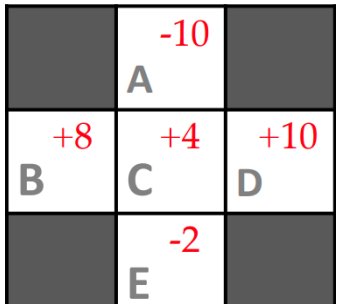
Policy π:



Trajectories:

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

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



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

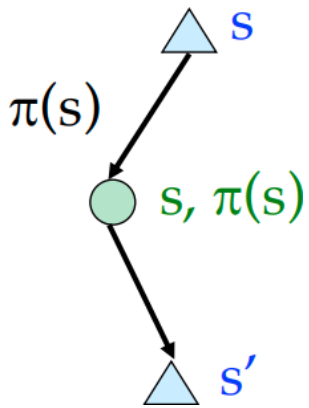
Unfortunately, direct utility estimation misses the fact that the state values $V^\pi(s)$ are not independent, since they obey the Bellman equations for a fixed policy:

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

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

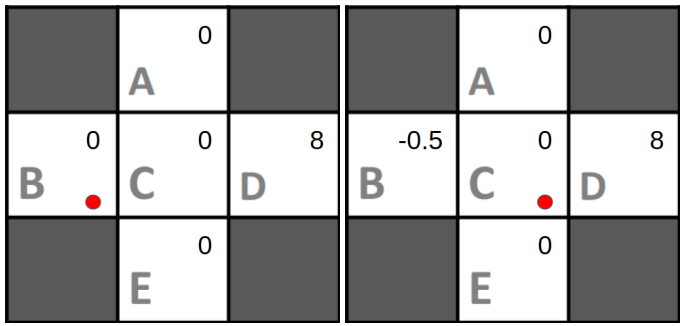
The TD-update can equivalently be expressed as the exponential moving average

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

Intuitively,

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

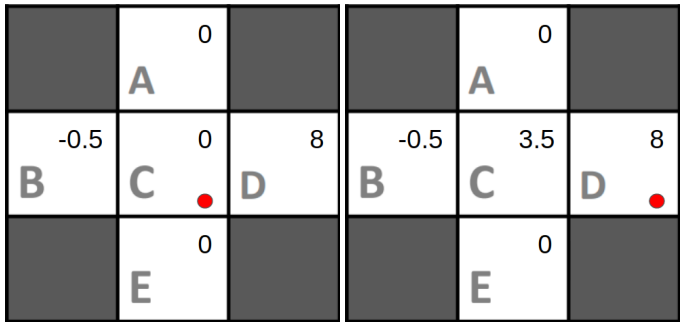
Example ($\gamma = 1, \alpha = 0.5$)



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

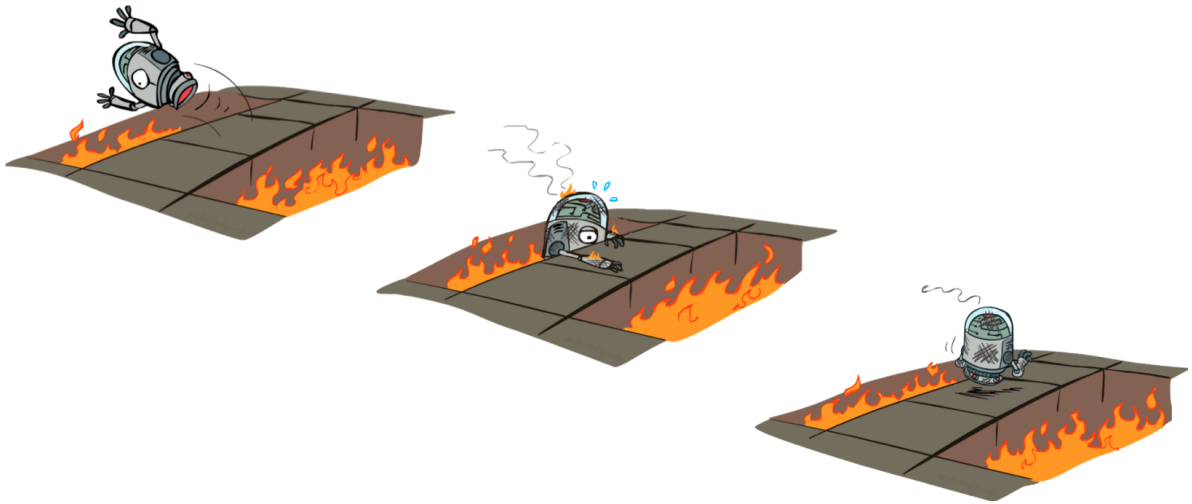
TD-update:

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

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

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

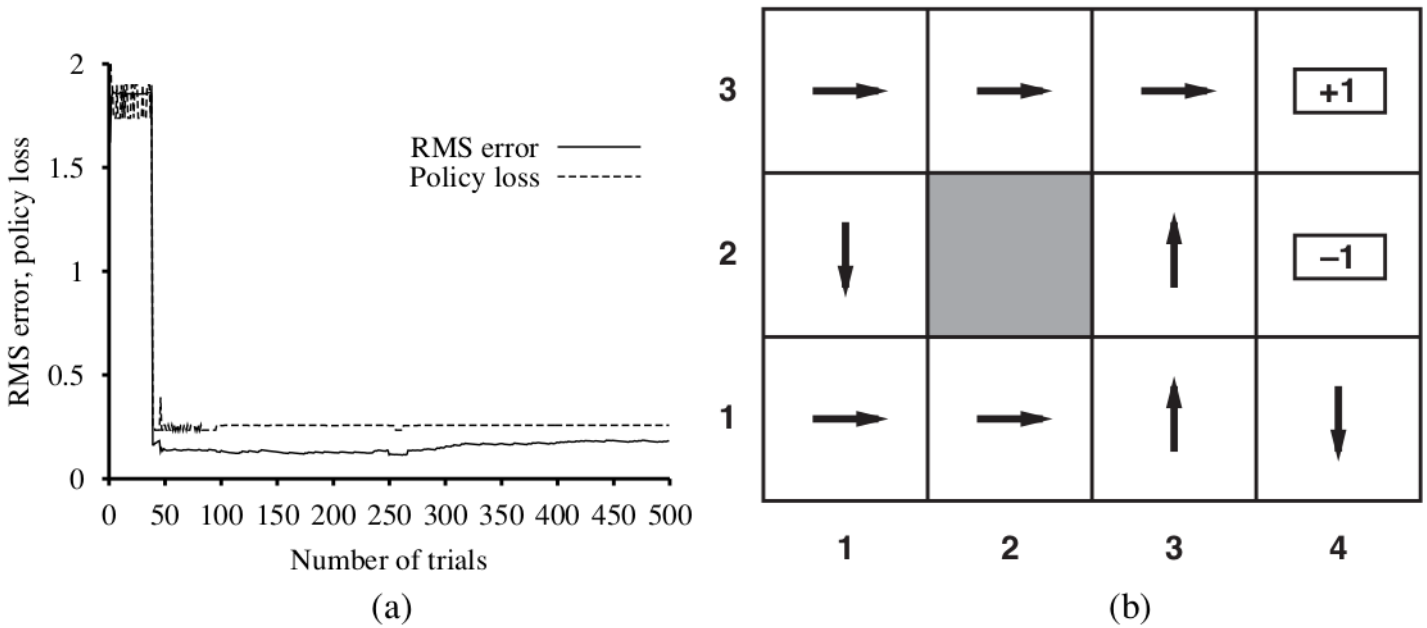


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence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

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

The resulting is greedy and suboptimal:

- The learned transition and reward models $\hat{P}$ and $\hat{R}$ are not the same as the true environment.
- 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.

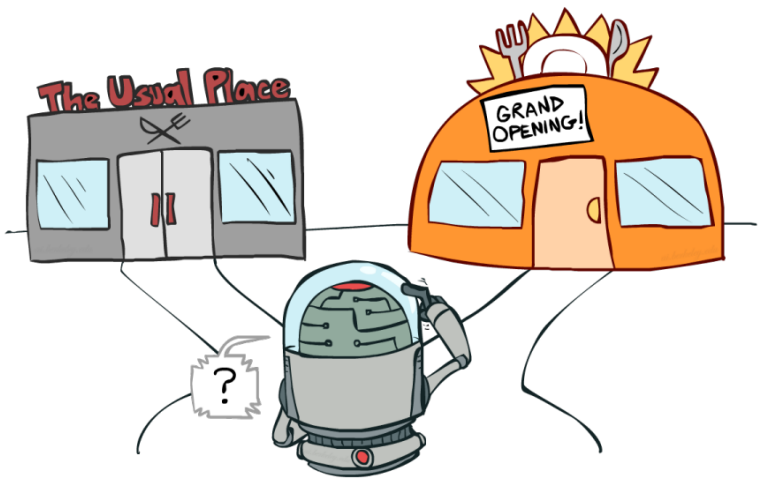


Image credits: CS188, UC Berkeley.

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.

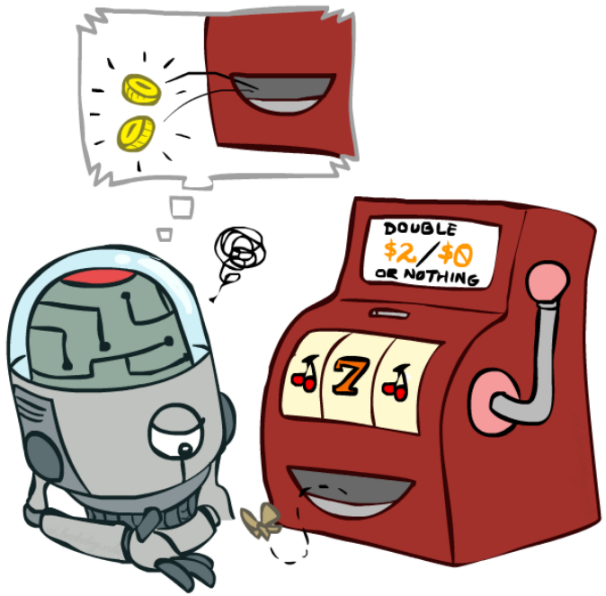
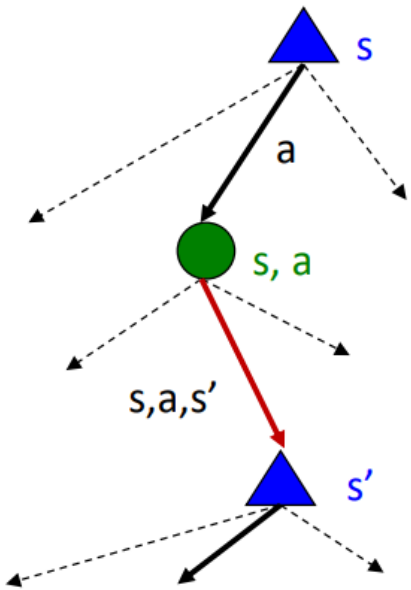


Image credits: CS188, UC Berkeley.

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 $Q(s,a)$:

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

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

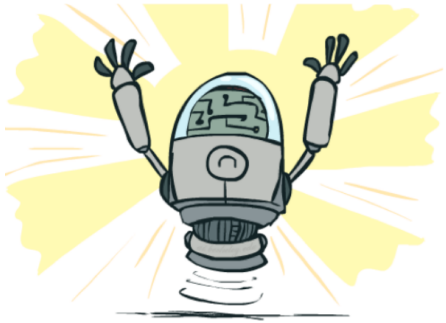
$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)).$$

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!

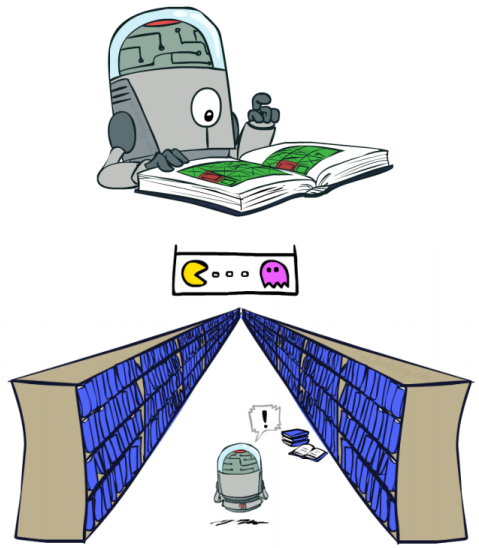
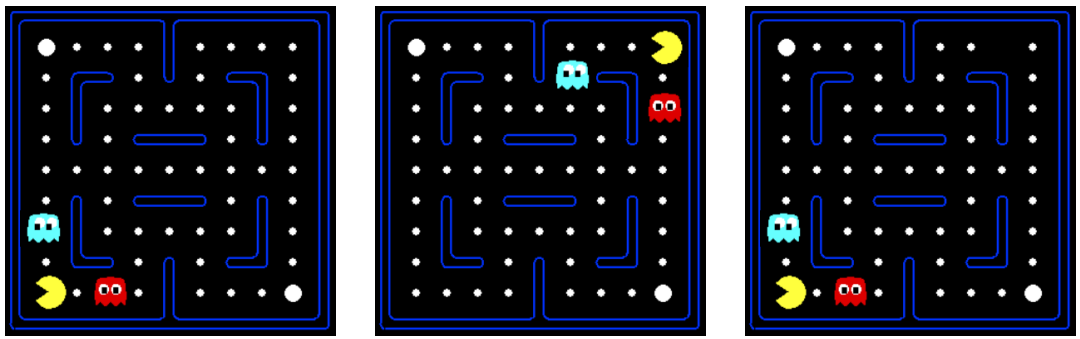


Image credits: CS188, UC Berkeley.

Example: Pacman

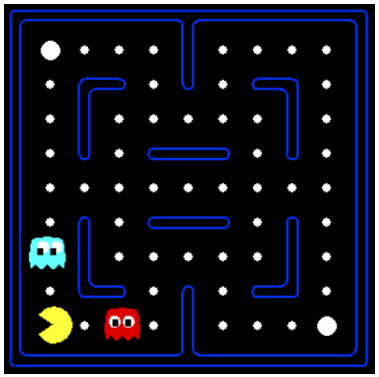
(a) (b) (c)



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.

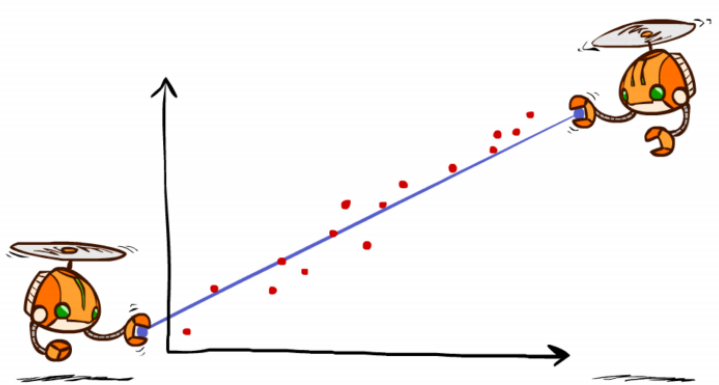
Feature-based representations



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.

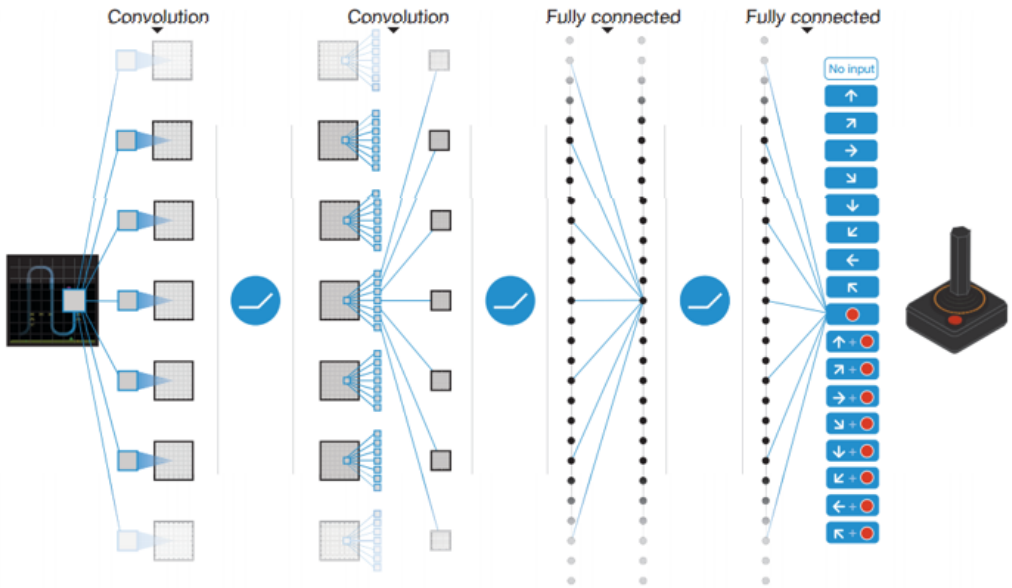


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking 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

Lifelong learning

Problem statement

We consider an off-grid microgrid with a storage system, PV generation, a diesel generator, and some consumption.

Changes that can occur are

- either gradual: PV degradation, demand growth
- or abrupt: device not responding.

A simulator of the system is used to train a control policy off-line.

Time steps of one-hour, several months of data available (PV and load).

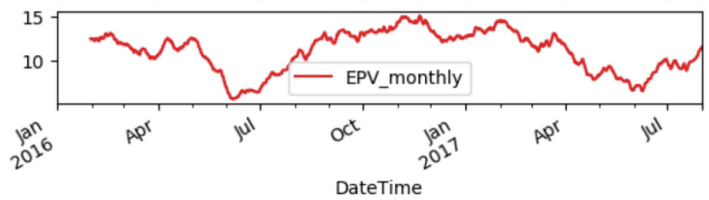


FIGURE 4.2: PV production and its daily, weekly and monthly rolling average.

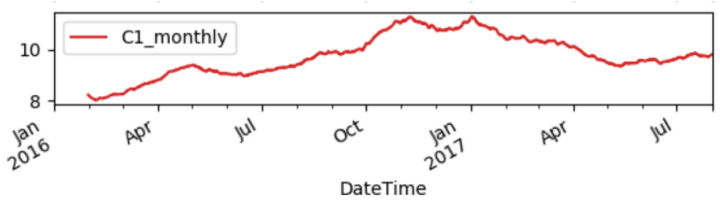


FIGURE 4.3: Electrical load and its daily, weekly and monthly rolling average.

Microgrid MDP

Consumption:

$$C_t \sim P_t^C(C_{t-1},...,C_{t-h}).$$

Renewable generation:

$$P_t^{\text{res}} \sim P_t^{P^{\text{res}}}(P_{t-1}^{\text{res}},...,P_{t-h}^{\text{res}}).$$

Stochastic variable:

$$\bar{s}_t = \left((C_t,...,C_{t-h}),\left(P_t^{\text{res}},...,P_{t-h}^{\text{res}}\right)\right) \in \bar{S}$$

Controllable state:

$$\underline{s}_t = (SoC_t) \in \underline{S}$$

State:

$$s_t = (\underline{s}_t, \bar{s}_t) \in S$$

Action:

$$a_t = \left(P_t^{\text{ch}}, P_t^{\text{dis}}, P_t^{\text{gen}}\right) \in A,$$

Deterministic transition:

$$SoC_{t+1} = SoC_t + \Delta t \cdot (\eta^{\text{ch}} P_t^{\text{ch}} - \frac{P_t^{\text{dis}}}{\eta^{\text{dis}}}),$$

$$\underline{s}_{t+1} = f_t\left(s_t, a_t\right),$$

Stochastic transition:

$$\bar{s}_{t+1} \sim \bar{P}_t\left(\bar{s}_t\right),$$

Reward function:

$$P_t^{\text{res}} + P_t^{\text{gen}} + P_t^{\text{dis}} + P_t^{\text{shed}} = P_t^{\text{ch}} + P_t^{\text{curt}} + C_t,$$

$$c_t^{\text{curt}} = P_t^{\text{curt}} \cdot \pi^{\text{curt}}$$

$$c_t^{\text{shed}} = P_t^{\text{shed}} \cdot \pi^{\text{shed}}$$

$$c_t^{\text{fuel}} = F_t \cdot \pi^{\text{fuel}}.$$

$$r_t = r(s_t, a_t) = -(c_t^{\text{fuel}} + c_t^{\text{curt}} + c_t^{\text{shed}}).$$

Objective function:

$$\pi^* = arg\max_{\pi} \mathbb{E} \sum_{t \in T} \gamma^t \cdot r(s_t, \pi(s_t)))$$

The proposed algorithm D-Dyna

Algorithm 4 DYNA

Inputs: MDP M, integers T, B, N
initialize policy π_θ, model M_ψ
for $t = 0$ **to** $T-1$ **do**
  $s \sim d_0$
  $a \sim \pi_\theta(\cdot|s)$
  $s', r \sim M(s,a)$
  $\pi_\theta = \text{UPDATEPOLICY}(s,a,r,s')$
  $M_\psi = \text{UPDATEMODEL}(s,a,r,s')$
  if $t \geq B$ **then**
    for $n = 0$ **to** $N-1$ **do**
      $s \sim d_0$
      $a \sim \pi_\theta(\cdot|s)$
      $\hat{s}', \hat{r} \sim M_\psi(s,a)$
      $\pi_\theta = \text{UPDATEPOLICY}(s,a,\hat{r},\hat{s}')$
    end for
  end if
end for

- Standard RL optimization where the policy is updated using samples collected from interaction with the real environment.
- Plus a loop where the policy is updated by samples collected by a model of the environment.

Model and policy updates

Algorithm 5 UPDATEPOLICY

1: **Input**: transition (s, a, r, s')
2: $V_\phi = \arg\min_{V_\varphi} L^V(V_\varphi)$
3: $\pi_\theta = \arg\max_{\pi_\varphi} \eta(\pi_\varphi)$

Algorithm 6 UPDATEMODEL

1: **Input**: transition (s, a, r, s')
2: $P_\psi = \arg\min_{P_\varphi} L^P(P_\varphi)$
3: $r_\psi = \arg\min_{r_\varphi} L^r(r_\varphi)$

Algo 5: The policy update was performed with the proximal policy optimization PPO algorithm.

Algo 6: The model of the environment is fitted with a regressor using states and reward samples collected from the real environment. A quantile regressor was used as a model and was trained with distributional losses.

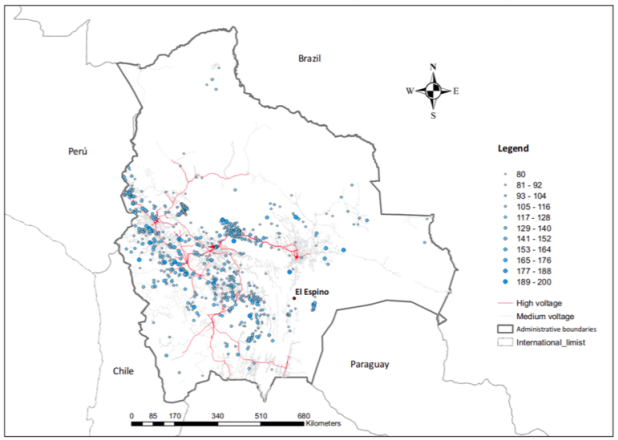
Implementation tricks

Although there has been some progress in RL algorithms and function approximators, this problem is still challenging.

The problem is further simplified to consider meta-actions:

- start-stop a generator, charge or discharge a battery
- relying on simple rules to derive precise set-points.

El Espino microgrid



- Guarani village
- Electrified village with a hybrid system PV/Diesel and storage
- 125 households
- 1 school
- 1 hospital

$\overline{S}$	120	kWh
$\overline{P}$, $\underline{P}$	100	kW
η^{ch}, η^{dis}	75^2 %	
π^{fuel}	1	€/kWh
π^{curt}	1.5	€/kWh
π^{shed}	10	€/kWh
Δ_t	1	h
$\overline{P^{\text{res}}}$	120	kW
$\overline{P^{\text{gen}}}$	9	kW
$\underline{P^{\text{gen}}}$	0	kW

Benchmarks

The policy learned with D-Dyna is compared to

1. Heuristic: a myopic rule based controller (lower bound)
2. MPC-1h: an optimization-based controller with perfect foresight over 1 hour
3. MPC-24h: an optimization-based controller with perfect foresight over 24 hours (upper bound)
4. "vanilla" PPO: model-free RL, to see the benefits from using a model

Test 1: Generalization

- We use the first year 2016 of the dataset for training and we evaluate on the second year 2017

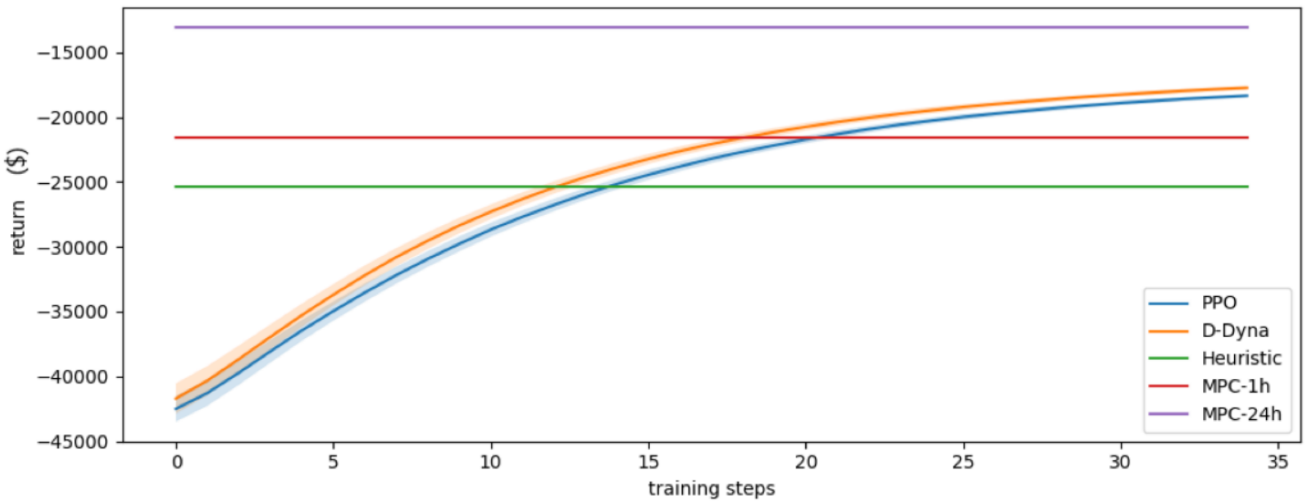


FIGURE 4.4: Cumulative returns (cost) on the test set as a function of the training progress of the RL algorithms.

- 25% cost reduction w.r.t. heurisitc controller
- Comparable performance to MPC-24h

Test 2: Robustness

- Abrupt failure of the storage system (not known by any model!)

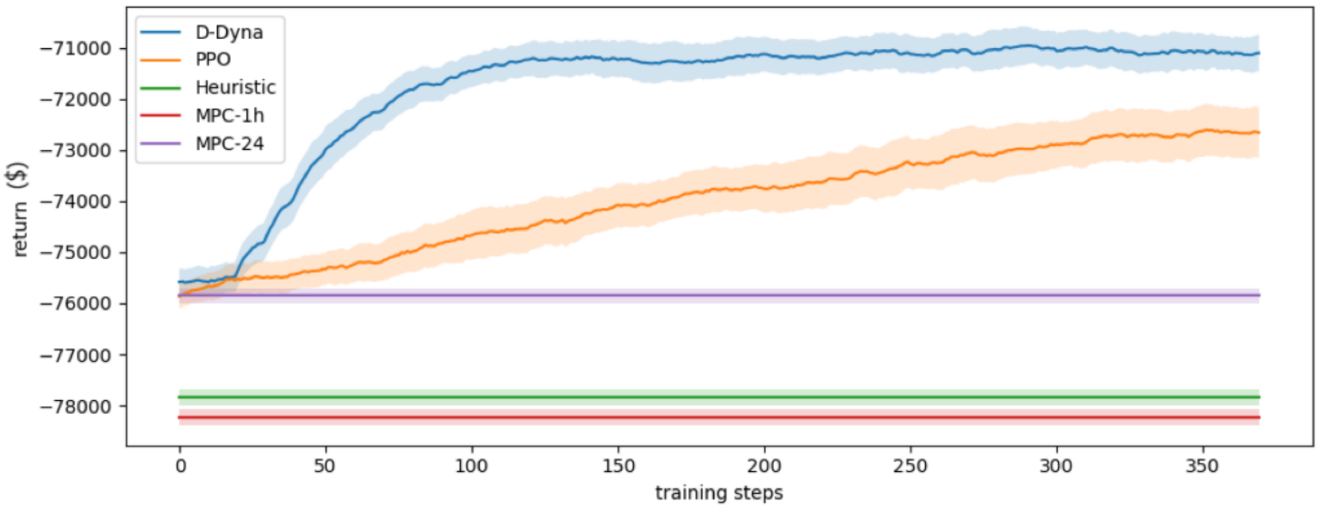


FIGURE 4.5: Cumulative returns (costs) when the battery is excluded as a function of the training progress of the RL algorithms.

- D-Dyna can detect change since the model has been exposed to similar incidents during training.
- Heuristic, MPC-1h, MPC-24h cannot adapt since no mechanism to detect or handle failure.

Test 3: Transfer

Transfer learning is the ability of speeding up learning on new MDPs by reusing past experiences between similar MDPs

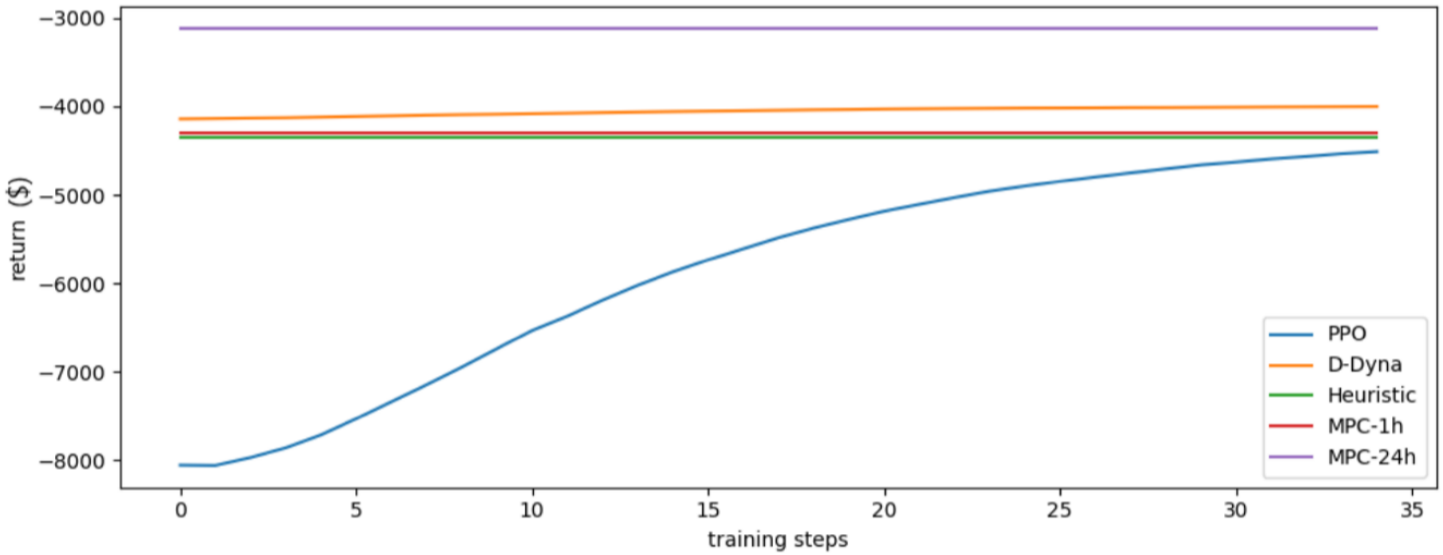


FIGURE 4.6: Cumulative returns (cost) during February as a function of the training progress of the RL algorithms.

- Jan. 2016 to pre-train the algorithms. Then we initiate the training process for Feb. and Aug. 2016 using the pre-trained model.
- Better performance than learning from scratch (compare D-Dyna to PPO)

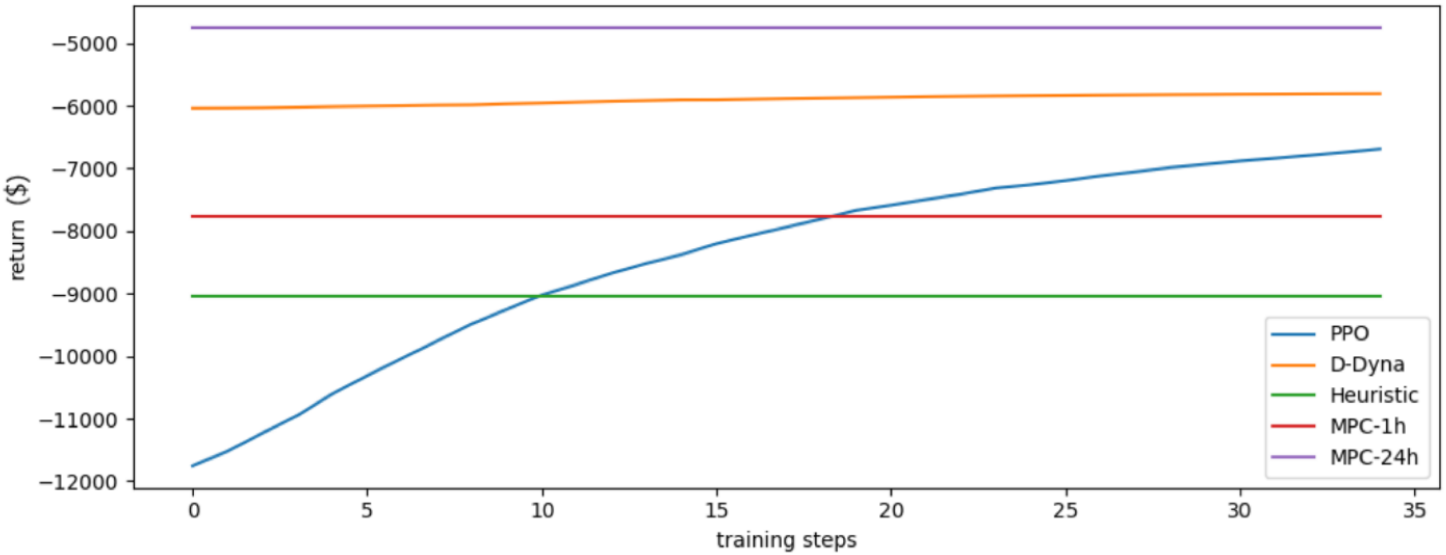


FIGURE 4.7: Cumulative returns (cost) during August as a function of the training progress of the RL algorithms.

- In contrast to February where the two policies perform similarly, in August the solar irraditation is limited (south-hemisphere).
- This amplifies the difference between a naïve controller and a good look-ahead policy

Conclusion

- Learned explicitely a control policy that can be transferred to a new setup with mild adaptations
- robust to changes in the system
- Learning a policy with continuous actions as future work
- There is still a lot to be done to apply to larger systems

The end.