THE UNIVERSITY *of* EDINBURGH
informatics

Advanced Robotics

Introduction to Reinforcement Learning

Steve Tonneau – With some content borrowed from Stephane Caron
School of Informatics
University of Edinburgh

Recap from last time:

Problem presentation

$$\min_{X,U} \int_0^T l\big(x(t), u(t)\big)dt + l_T\big(x(T)\big)$$

$$\text{s.t.} \quad \dot{x}(t) = f(x(t),u(t))$$

Path cost

Terminal cost

- X and U are functions of t:

  $X: t \in \Re \rightarrow x(t) \in \Re^{\mathrm{nx}}$

  $U: t \in \Re \rightarrow u(t) \in \Re^{\mathrm{nu}}$

- The terminal time T is fixed

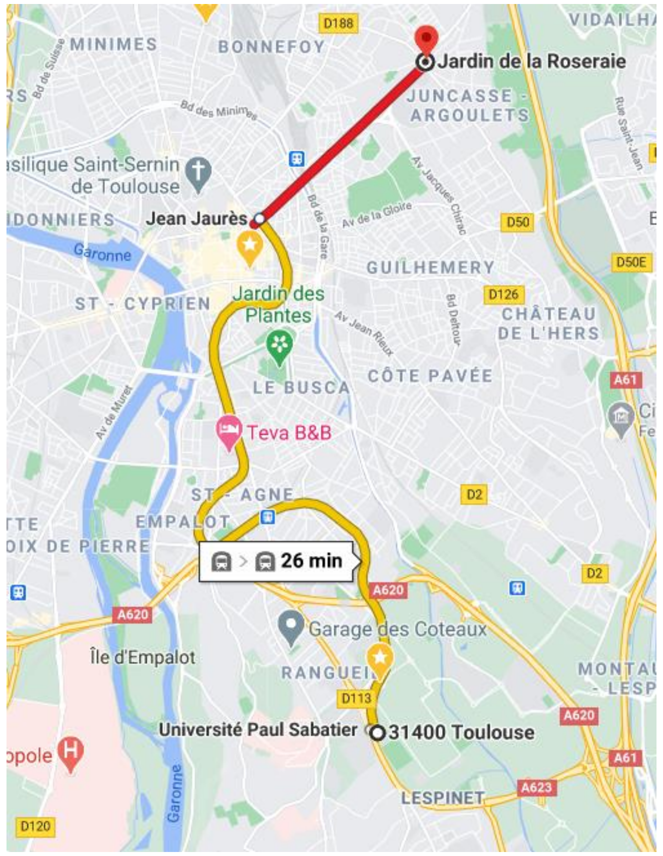
Recap from last time:

Principle of optimality

- ❑ How to find the optimal control?

- ❑ Principle of optimality:

  Subpath of optimal paths are also optimal for their own subproblem



Constructing an optimal policy

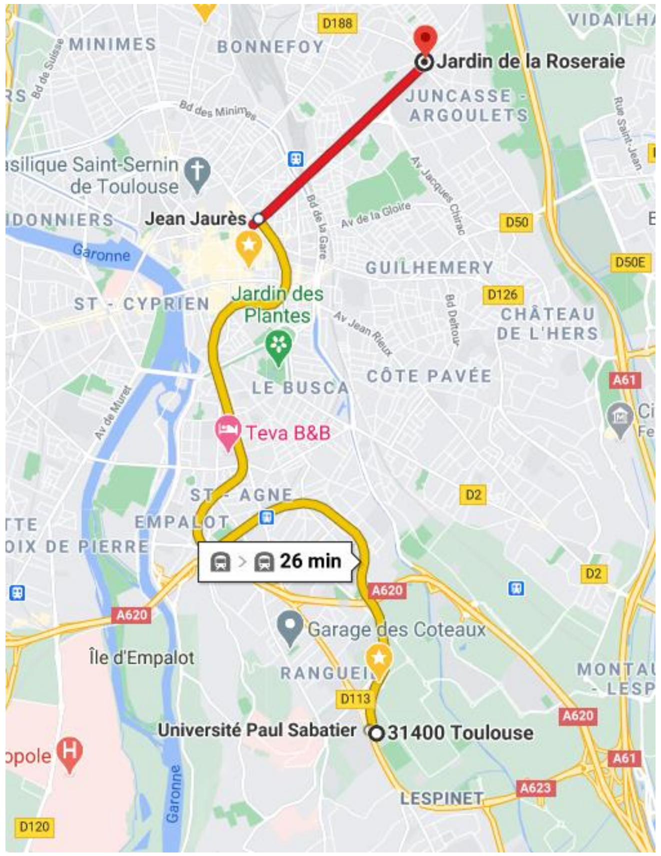
For every possible state we are in:

- ❑ There exists an optimal action towards the goal
  “Going to Jean Jaurès is optimal…”
- ❑ To know the action is optimal we need to know what next action will be optimal
  “… Because Jean Jaurès => Roseraie is optimal”

How is that helping?

If we know the cost of ALL actions from ALL states
at each state we can determine exactly what best action to take

A function that gives us an action to take for a given state is called a policy



Constructing an optimal policy

For every possible state we are in:

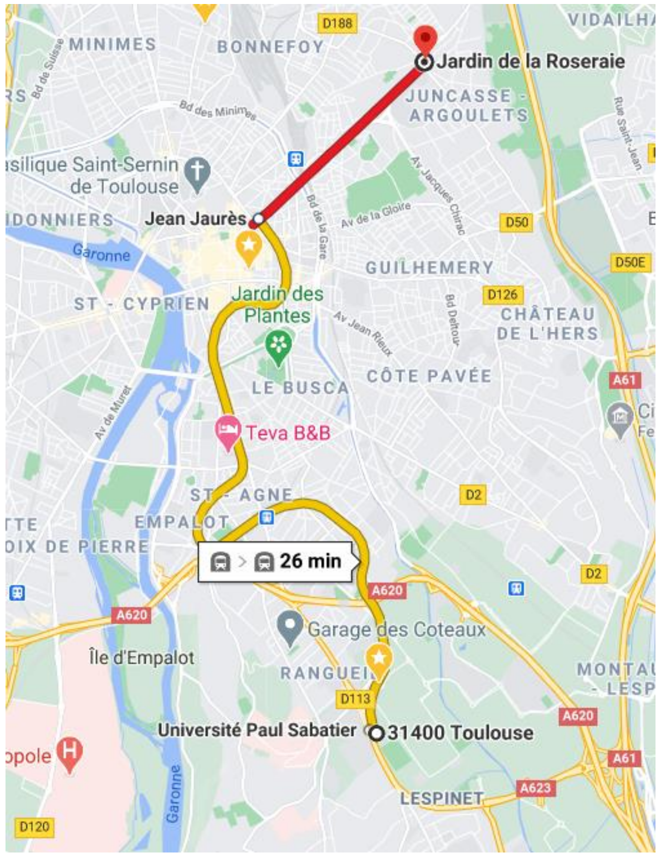
- ❑ There exists an optimal action towards the goal
  "Going to Jean Jaurès is optimal…"
- ❑ To know the action is optimal we need to know what next action will be optimal
  "… Because Jean Jaurès => Roseraie is optimal"

How is that helping?

If we know the cost of ALL actions from ALL states
at each state we can determine exactly what action to take

A function that gives us an action to take for a given state is called a policy

⇨ (How) to compute optimal policies for robotics problems?



Obvious limitations

- ❑ Curse of dimensionality:
  How to solve for all possible states in high dimensions ?
  (Especially considering continuous state/action spaces in robotics?)

- ❑ Reinforcement Learning (RL) aims at finding a computationally tractable representation of the value function (and through that the optimal policy, sometimes directly)

RL for robotics – 2020[1]: quadrupedal locomotion

- ❑ Teacher / student
- ❑ Residual RL

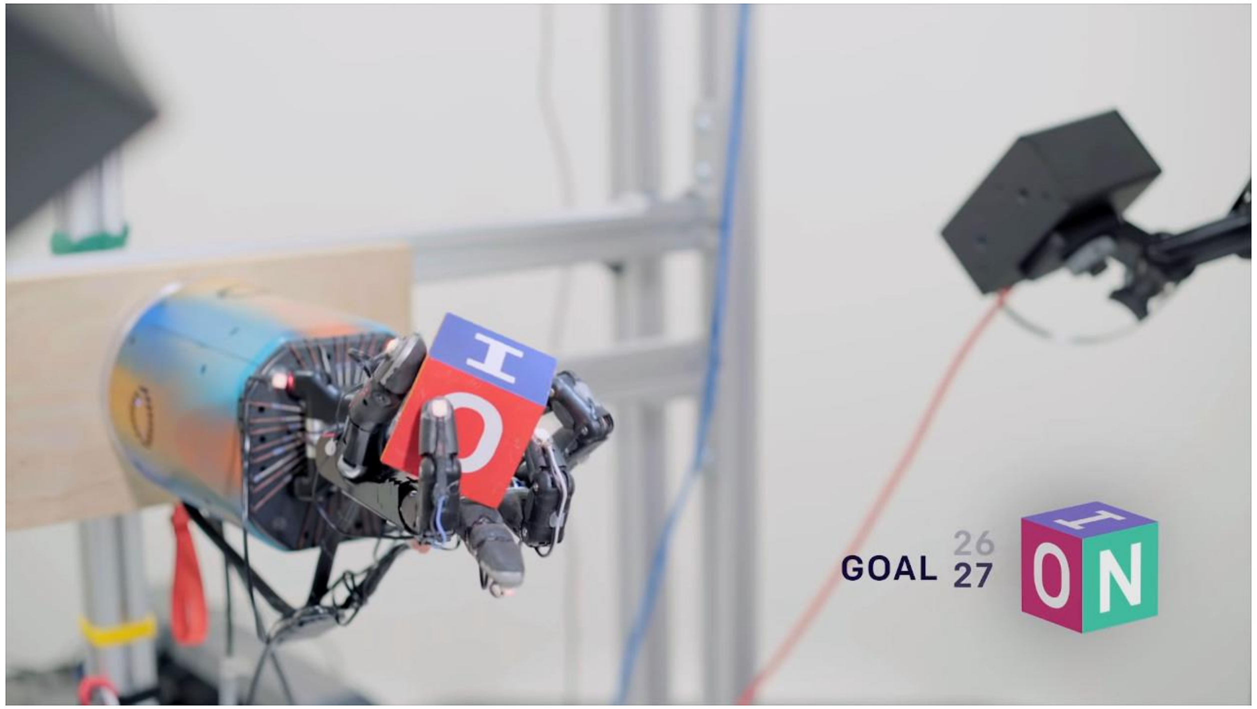
https://youtu.be/9j2a1oAHDL8?feature=shared&t=32

[1] Lee et al., Learning Quadrupedal Locomotion over Challenging Terrain, Science robotics 2020

RL for robotics – 2018: In-hand reorientation

- ❑ Domain randomization

- ❑ LSTM policy



https://openai.com/research/learning-dexterity

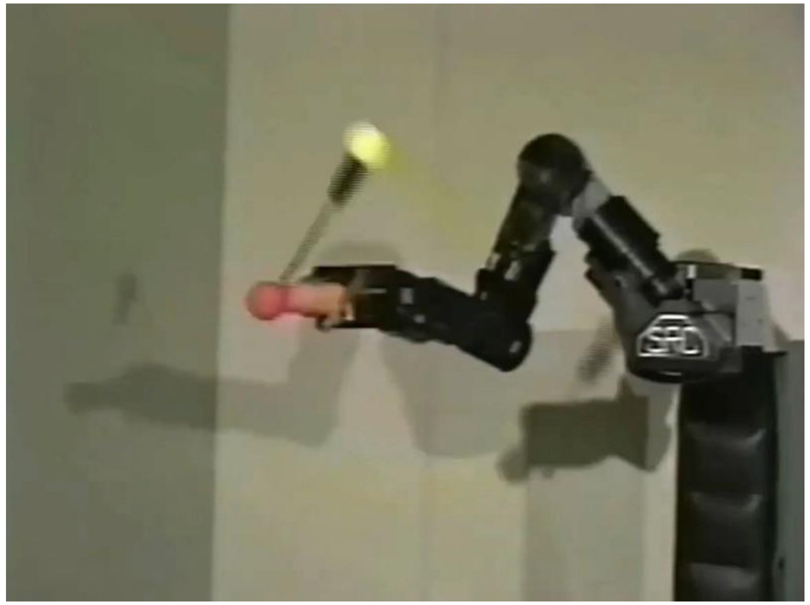
RL for robotics – 2010[2]: Helicopter stunts

- ❑ Supervised learning to learn a dynamic model
- ❑ Inverse reinforcement learning
  (learn value function from demonstrations)

https://youtu.be/M-QUkgk3HyE

[2] Abeel et al., Autonomous Helicopter Aerobatics through Apprenticeship Learning, IJRR 2010

RL for robotics – 1997[3]: Pendulum swing up



https://youtu.be/g3I2VjeSQUM?t=412

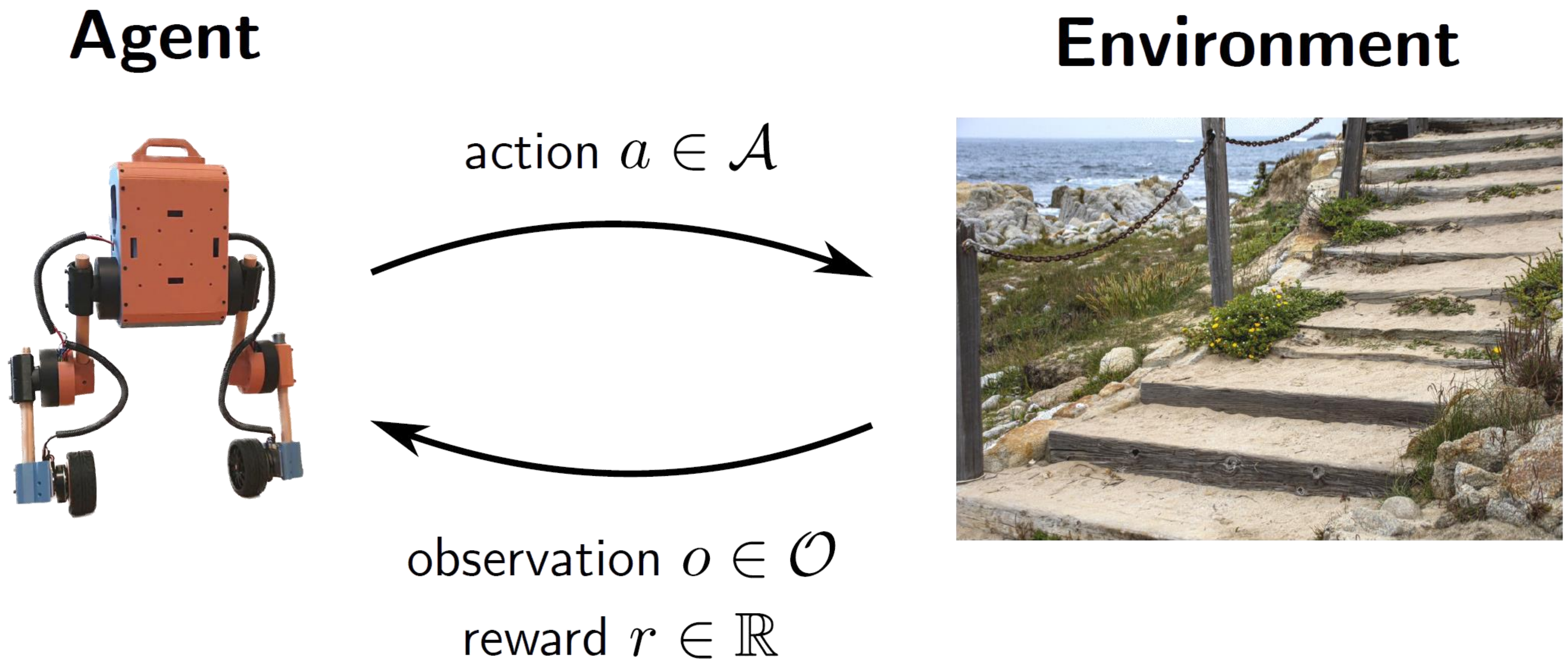
[3] Atkeson and Schaal, Robot learning from demonstration, ICML. 1997

Introduction to Concepts in Reinforcement Learning

A familiar formalism

Reinforcement learning	**Control theory**
State: s_t	State: x_t
Observation: o_t	Observation: y_t
Action: a_t	Input: u_t
Reward: r_t	Stage cost: $\ell(x_t, u_t)$
Episode: $\tau = (s_0, a_0, \dots)$	Trajectory: $\tau = (x_0, u_0, \dots)$
Return: $\max R(\tau) = \sum_{t \in \tau} r_t$	Cost: $\min J(\tau) = \sum_{t \in \tau} \ell(x_t, u_t)$
Model: $s_{t+1} = f(s_t, a_t)$	Dynamics: $x_{t+1} = f(x_t, u_t)$

Scope


Agent
Environment
action $a \in \mathcal{A}$
observation $o \in \mathcal{O}$
reward $r \in \mathbb{R}$

Reward

Image credit: L. M. Tenkes, source: https://araffin.github.io/post/sb3/

Model of the environment

- **State:** s_t, ground truth of the environment
- **Action:** a_t, decision of the agent
- **Observation:** o_t, *partial* estimation of the state from sensors
- **Reward:** $r_t \in \mathbb{R}$, scalar feedback, often $r_t = r(s_t, a_t)$ or $r(s_t, a_t, s_{t+1})$

	Deterministic	*Stochastic*	
Model:	$s_{t+1} = f(s_t, a_t)$	$s_{t+1} \sim p(\cdot \mid s_t, a_t)$	how the environment evolves
Initial state:	s_0	$s_0 \sim \rho_0(\cdot)$	where we start from
Observation:	$o_t = h(s_t)$	$o_t \sim z(\cdot \mid s_t)$	how sensors measure the world

Altogether: partially-observable Markov decision process (POMPD).

Model of the agent

A couple more definitions:

- **Episode:** $\tau = (s_0, a_0, s_1, a_1, \ldots)$
- **Return:** $R(\tau) = \sum_{t \in \tau} r_t$ or with discount $\gamma \in]0,1[$: $R(\tau) = \sum_{t \in \tau} \gamma^t r_t$

Finally, our model of the agent:

- **Policy:** agent decisions, deterministic $a_t = \pi(s_t)$ or stochastic $a_t \sim \pi(\cdot|s_t)$

Goal

The goal of reinforcement learning is to find a *policy* that *maximizes* return:

$$
\begin{aligned}
\max_{\pi}\ & \mathbb{E}_{\tau}[R(\tau)] \\
\text{s.t. } & \tau = (s_0, a_0, s_1, a_1, \ldots) \\
& s_0 \sim \rho_0(\cdot) \\
& a_0 \sim \pi(\cdot|s_0) \\
& s_1 \sim f(\cdot|s_0, a_0) \\
& \vdots
\end{aligned}
$$

Shorthand: $\max_{\pi} \mathbb{E}_{\tau \sim \pi}[R(\tau)]$.

Value functions

State value functions:

- **On-policy value function:** return we can expect from a given policy: $V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}(R(\tau)|s_0 = s)$
- **Optimal value function:** best return we can expect from a state: $V^{*}(s) = \max_{\pi} \mathbb{E}_{\tau \sim \pi}(R(\tau)|s_0 = s)$

There are also state-action value functions $Q^{\pi}(s, a)$ and $Q^{*}(s, a)$.

Finding the optimal policy / finding the optimal value function are dual problems

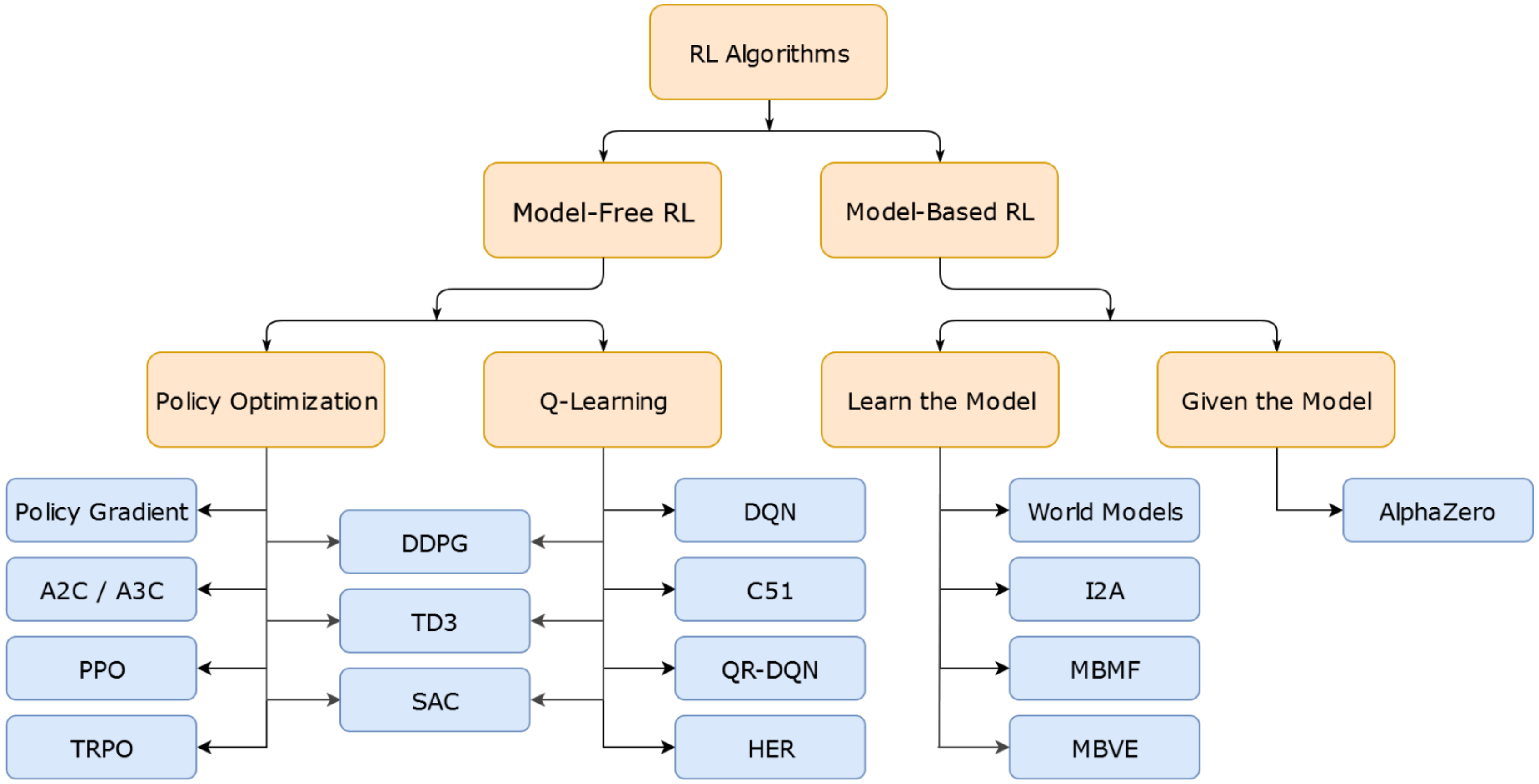
Components of an RL algorithm

A reinforcement-learning algorithm may include any of the following:

- **Policy:** behavior function
- **Value function:** evaluation of states
- **Model:** representation of the environment

An algorithm with a policy (actor) and a value function (critic) is called *actor-critic*. If it doesn't use a model, it is *model-free*. If it has one (e.g. chess, go), it is *model-based*.

A partial taxonomy of RL algortihms



There are many partial taxonomies, this one is from Josh Achiam, Spinning Up in Deep Reinforcement Learning.
https://spinningup.openai.com

What now?

- We'll briefly discuss the structure of a few major algorithms

Value-based reinforcement learning

2D grid world (taken from https://edin.ac/48JKP5X)

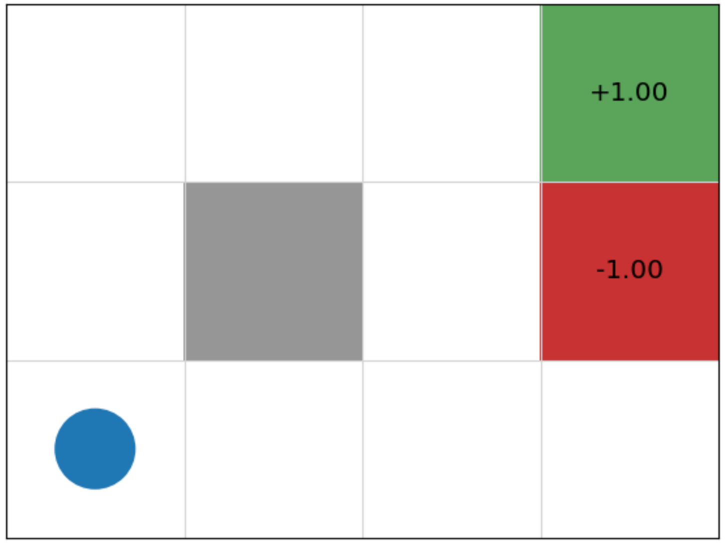
- State s_t: (x,y) coordinate

- Action: Move up / down / left / right / terminate*
  (0,1) / (0,-1) / (-1,0) / (1,0) / (0,0)

- Reward r(s,a,s'): reward from transitioning from s to s' through action a

  +1 in green cell / -1 in red cell

- Model $s_{t+1} = s_t + a_t$ (bounded by walls / holes)

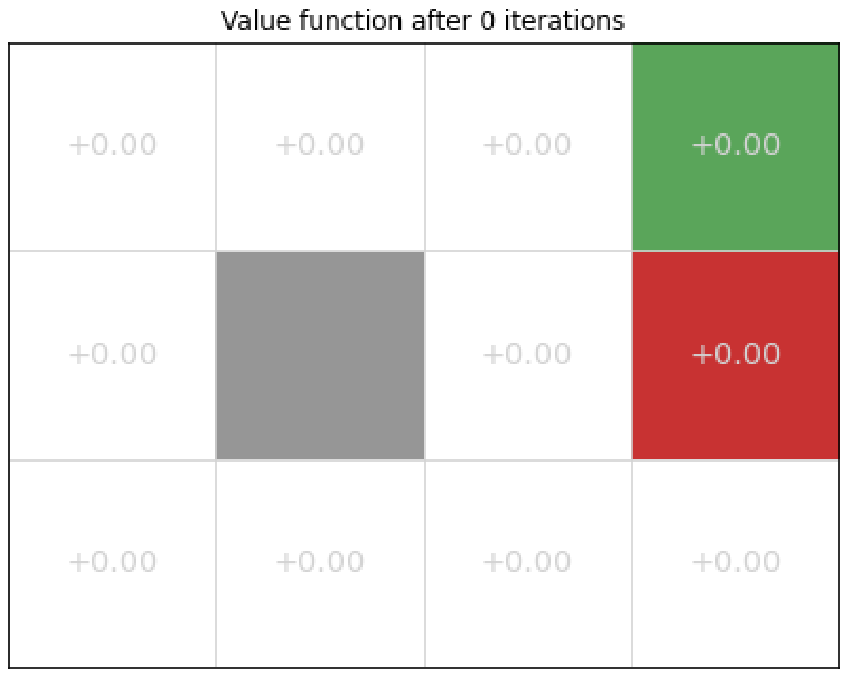


* Only action possible at red and green states

Value iteration / Dynamic programming

- ❑ Iterative algorithm to learn Value function
- ❑ Start: initial value function (typically 0 everywhere)

$$V^0(s) = 0, \forall s$$



Value iteration / Dynamic programming

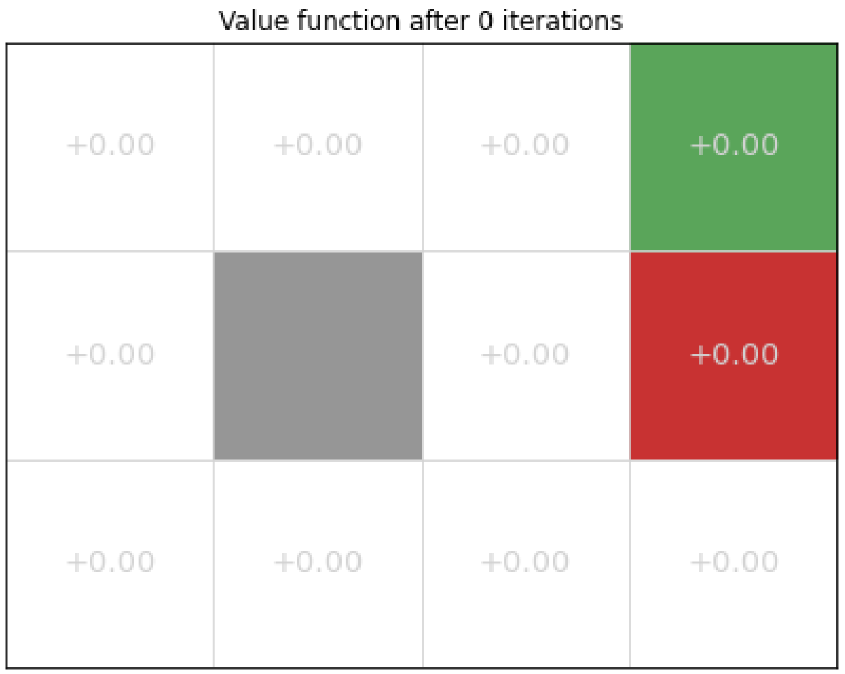
- Iterative algorithm to learn Value function
- Start: initial value function (typically 0 everywhere)

$$V^0(s) = 0, \forall s$$

- At step i:
  - Assume value of each neighbour state is optimal value
  - Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$



Value iteration / Dynamic programming

- Iterative algorithm to learn Value function
- Start: initial value function (typically 0 everywhere)

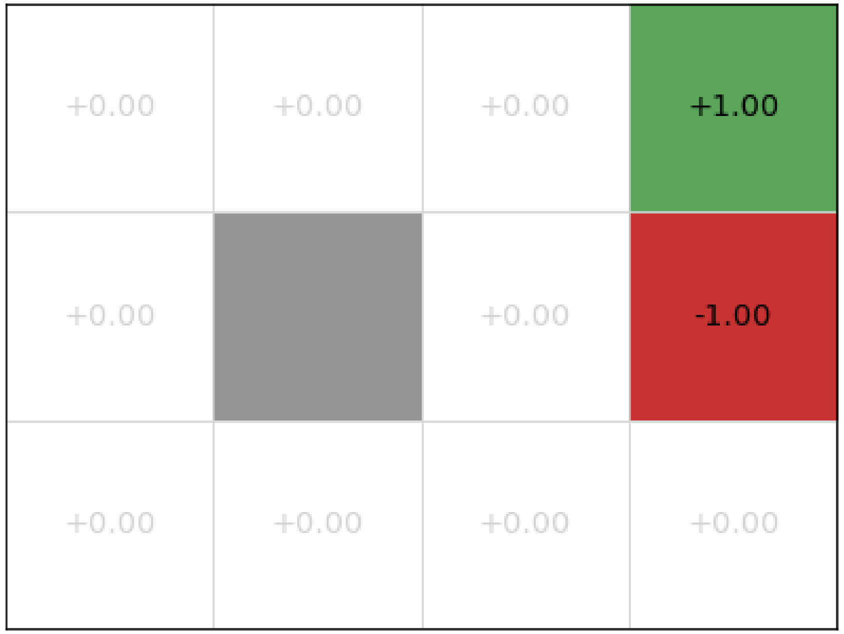
$$V^0(s) = 0, \forall s$$

- At step i:
  - Assume value of each neighbour state is optimal value
  - Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$

Value function after 1 iterations



Value iteration / Dynamic programming

- Iterative algorithm to learn Value function
- Start: initial value function (typically 0 everywhere)

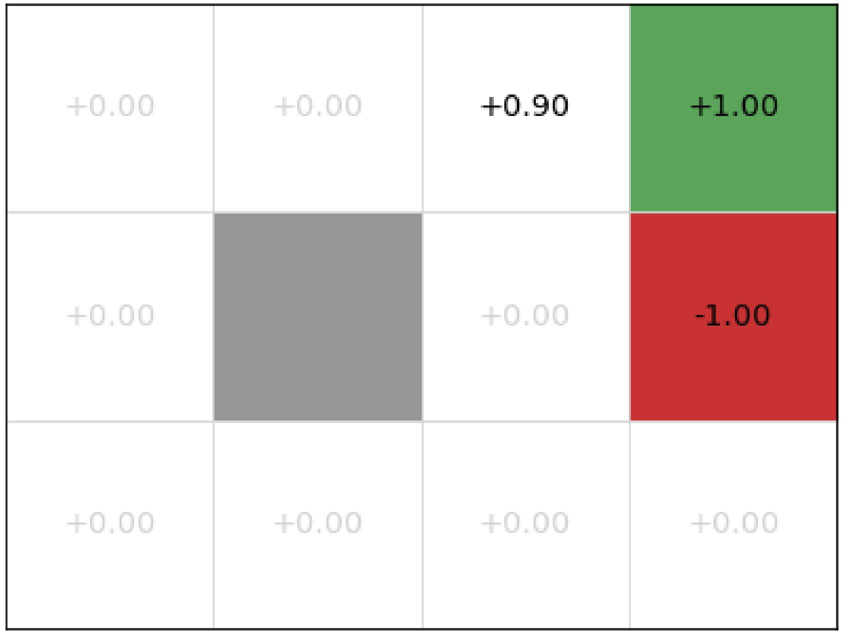
$$V^0(s) = 0, \forall s$$

- At step i:
  - Assume value of each neighbour state is optimal value
  - Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$

Value function after 2 iterations $\gamma = 0,9$



+0.00	+0.00	+0.90	+1.00
+0.00		+0.00	-1.00
+0.00	+0.00	+0.00	+0.00

Value iteration / Dynamic programming

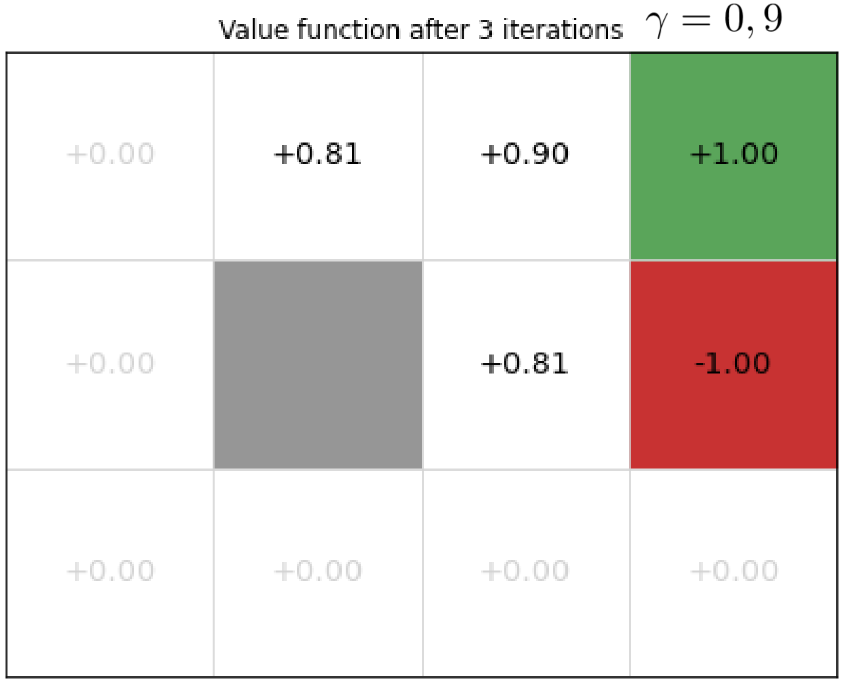
- Iterative algorithm to learn Value function
- Start: initial value function (typically 0 everywhere)

$$V^0(s) = 0, \forall s$$

- At step i:
  - Assume value of each neighbour state is optimal value
  - Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$



Value iteration / Dynamic programming

- ❑ Iterative algorithm to learn Value function
- ❑ Start: initial value function (typically 0 everywhere)

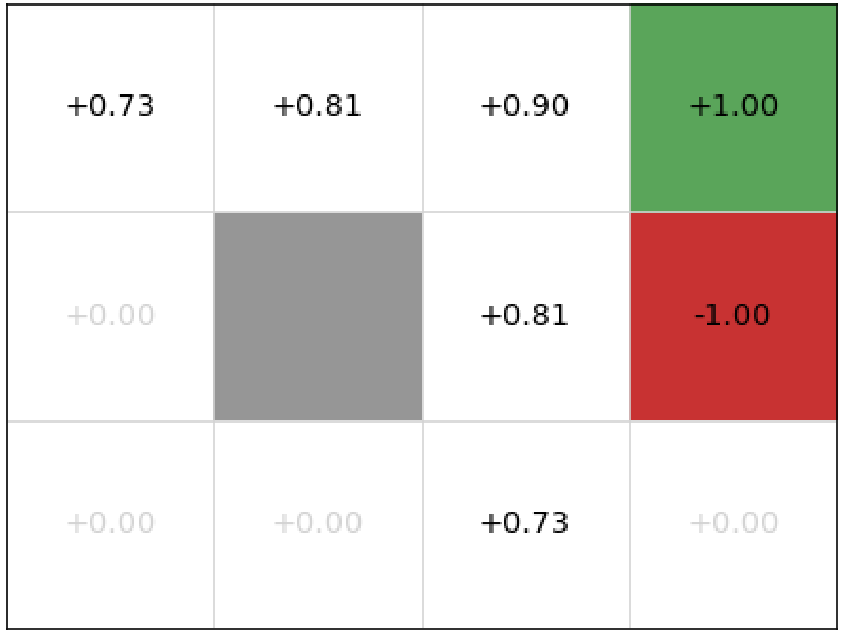
$$V^0(s) = 0, \forall s$$

- ❑ At step i:
  - ❑ Assume value of each neighbour state is optimal value
  - ❑ Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$

Value function after 4 iterations $\gamma = 0,9$



+0.73	+0.81	+0.90	+1.00
+0.00		+0.81	-1.00
+0.00	+0.00	+0.73	+0.00

Value iteration / Dynamic programming

- ❑ Iterative algorithm to learn Value function
- ❑ Start: initial value function (typically 0 everywhere)

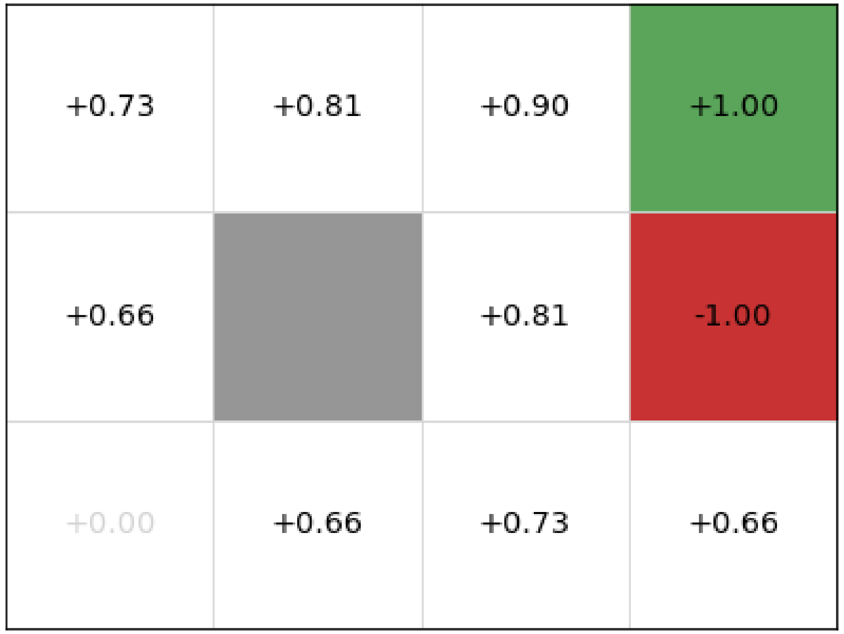
$$V^0(s) = 0, \forall s$$

- ❑ At step i:
  - ❑ Assume value of each neighbour state is optimal value
  - ❑ Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$

Value function after 5 iterations $\gamma = 0,9$



+0.73	+0.81	+0.90	+1.00
+0.66		+0.81	-1.00
+0.00	+0.66	+0.73	+0.66

Value iteration / Dynamic programming

- ❑ Iterative algorithm to learn Value function
- ❑ Start: initial value function (typically 0 everywhere)

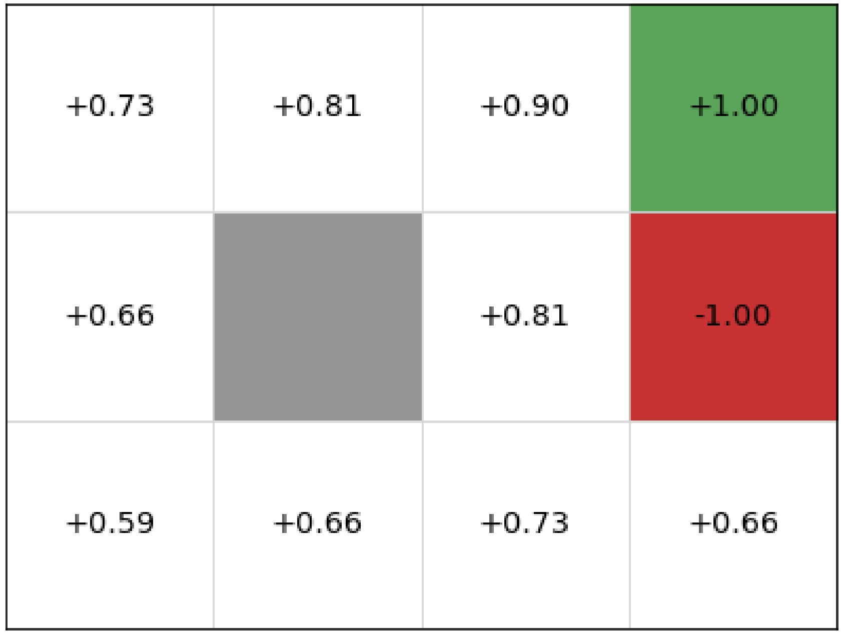
$$V^0(s) = 0, \forall s$$

- ❑ At step i:
  - ❑ Assume value of each neighbour state is optimal value
  - ❑ Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$

Value function after 6 iterations $\gamma = 0,9$



+0.73	+0.81	+0.90	+1.00
+0.66	▨	+0.81	-1.00
+0.59	+0.66	+0.73	+0.66

Value iteration / Dynamic programming

- ❑ Iterative algorithm to learn Value function

- ❑ Start: initial value function (typically 0 everywhere)

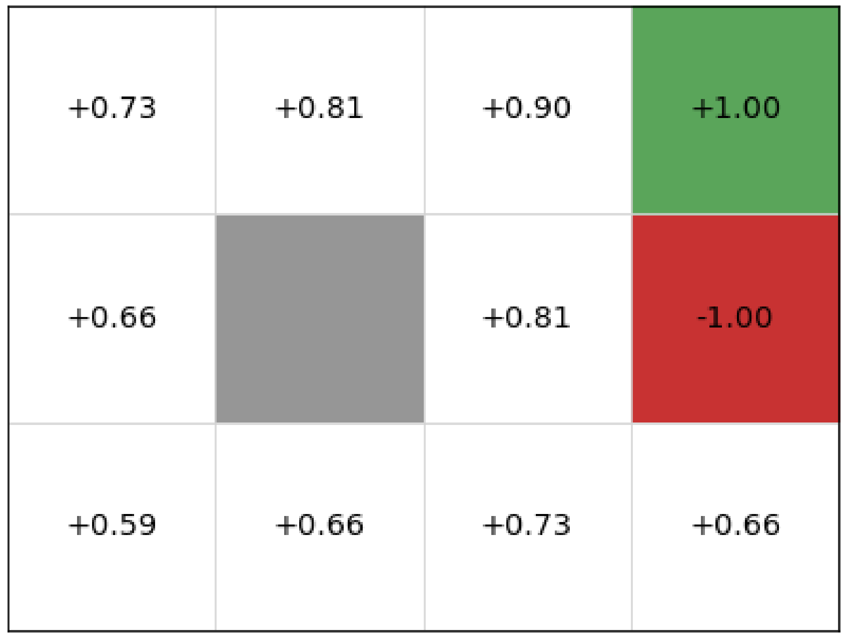
$$V^0(s) = 0, \forall s$$

- ❑ At step i:
  - ❑ Assume value of each neighbour state is optimal value
  - ❑ Update current value as state value + maximum value of all states reachable from any action from current state times discount factor:

$$V^{i+1}(s) = \max_a Q(s,a)$$

$$Q(s,a) = r(s,a,s') + \gamma V^i(s')$$

Value function after 7 iterations $\gamma = 0,9$



+0.73	+0.81	+0.90	+1.00
+0.66		+0.81	-1.00
+0.59	+0.66	+0.73	+0.66

Stop when value function is no longer significantly updated

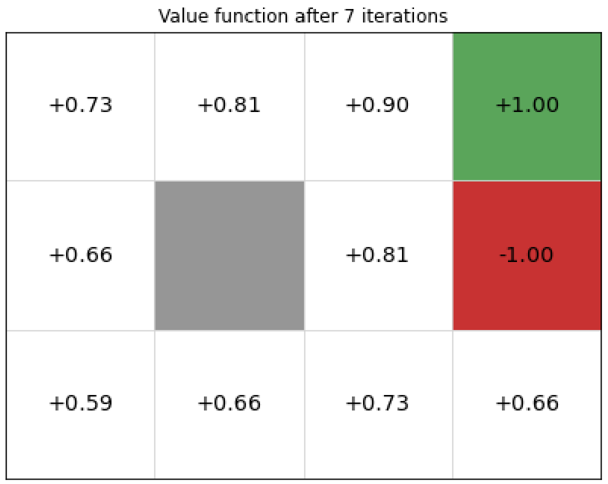
Policy extraction

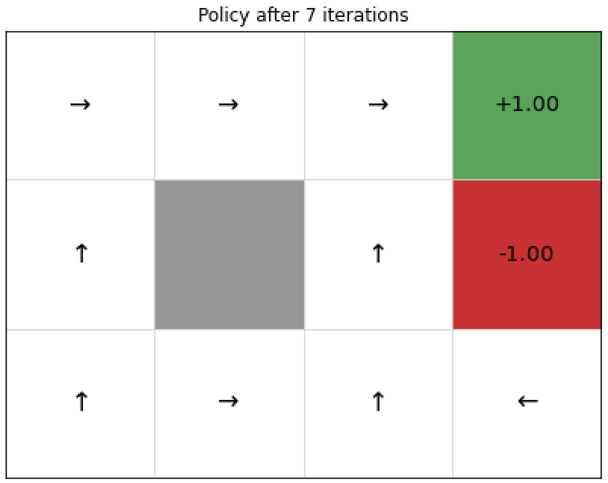
- Optimal policy obtained by selecting action that maximises reward

$$\pi(s) = \text{argmax}_{a \in A(s)} Q(s, a)$$

- With

$$Q(s, a) = r(s, a, s') + \gamma V^i(s')$$





Stochastic version

- Extension is straightforward:

$Q(s, a)$ becomes the sum of prob. to reach all states when taking action a:

$$Q(s,a) = \sum_{s' \in S} P_a(s' \mid s) \, [r(s,a,s') + \gamma \, V(s')]$$

The rest of the algorithm is exactly the same

Value iteration summary

- ❑ Requires complete knowledge of the MDP

  including transition probabilities and rewards

- ❑ Requires tractable problem size

- ❑ What if model is not known / too complex?

Q-learning – model-free learning

- Idea: approximate $Q(s,a)$ by running episodes and value iteration update
  - Select sequence of actions until terminal state is reached $\tau = (s_0, a_0, \dots)$
- Stochastic model: same action from same state can lead to different results

  Introducing learning rate $\alpha \in [0,1]$: desire to change opinion when facing new outcomes
- As for value iteration, arbitrary initialisation (or guided by initial guess)
- A time t, from s_t choose a_t , compute r_t and update as follows (img src: Wikipedia)

$$Q^{new}(s_t,a_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(s_t,a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \bigg)}^{}_{\text{new value (temporal difference target)}}$$

Q-learning - shortcomings

- ❑ Q-learning still assumes discrete environments (store Q in tables)
- ❑ In most continuous settings, a same state might never be visited twice

  We also do not want to ignore similar states
- ❑ We can combine Q-learning with function approximation for larger problems

  For instance using a neural network as a function approximator to replace the Q-table

  Not covered here, though an example is given in tutorial notebook

Policy-based Reinforcement learning

Policy-based learning

- ❑ Idea: instead of learning the value function, directly learn a policy (the dual problem)

  Useful when policy is simpler to describe than the value function (e.g. simpler control law than full dynamics model) or policy ‘extraction’ is harder (e.g. action space is too large or infinite)

- ❑ As with value iteration, policy iteration methods also exist

- ❑ This leads us to a technique called the policy gradient method

Policy gradient methods

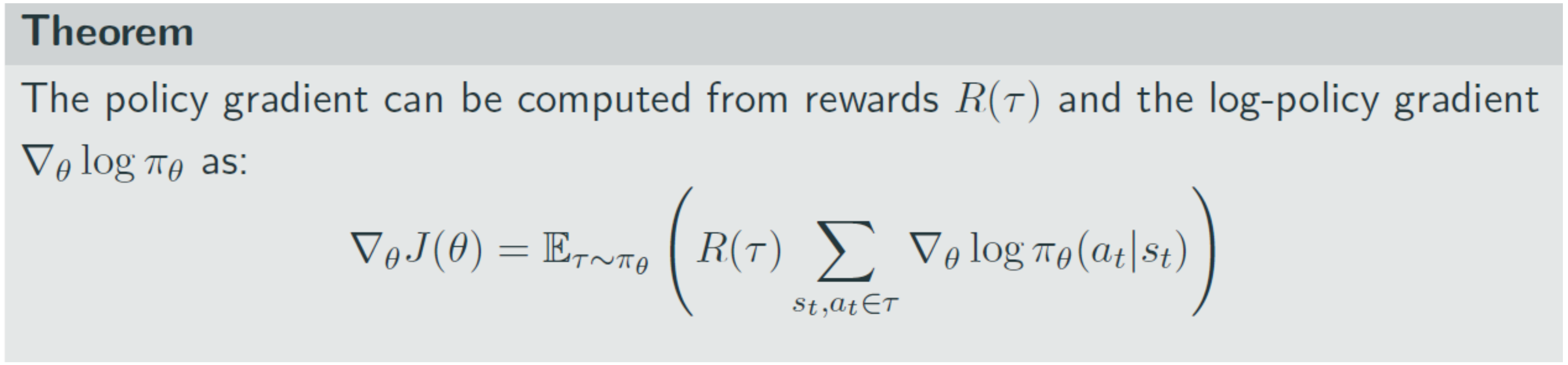
- Assume policy π_θ is differentiable and parametrised by vector $\theta \in \mathbb{R}^n$
- Estimate the reward R returned by the policy (we'll come to this)
- For minimising cost in optimisation, we typically perform gradient descent
- To maximise the reward, we will perform gradient **ascent:**

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta)$$

  - With α being the step size
  - $J(\theta) = \max_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$ maximises the expected return
  - $\nabla_\theta J(\theta)$ being the policy gradient

Computing $\nabla_\theta J(\theta)$

❑ One can rigorously prove the following statement:



Theorem

The policy gradient can be computed from rewards $R(\tau)$ and the log-policy gradient $\nabla_\theta \log \pi_\theta$ as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left(R(\tau) \sum_{s_t, a_t \in \tau} \nabla_\theta \log \pi_\theta(a_t | s_t) \right)$$

❑ We will simply accept this for the purposes of this lecture

Policy gradient algorithms

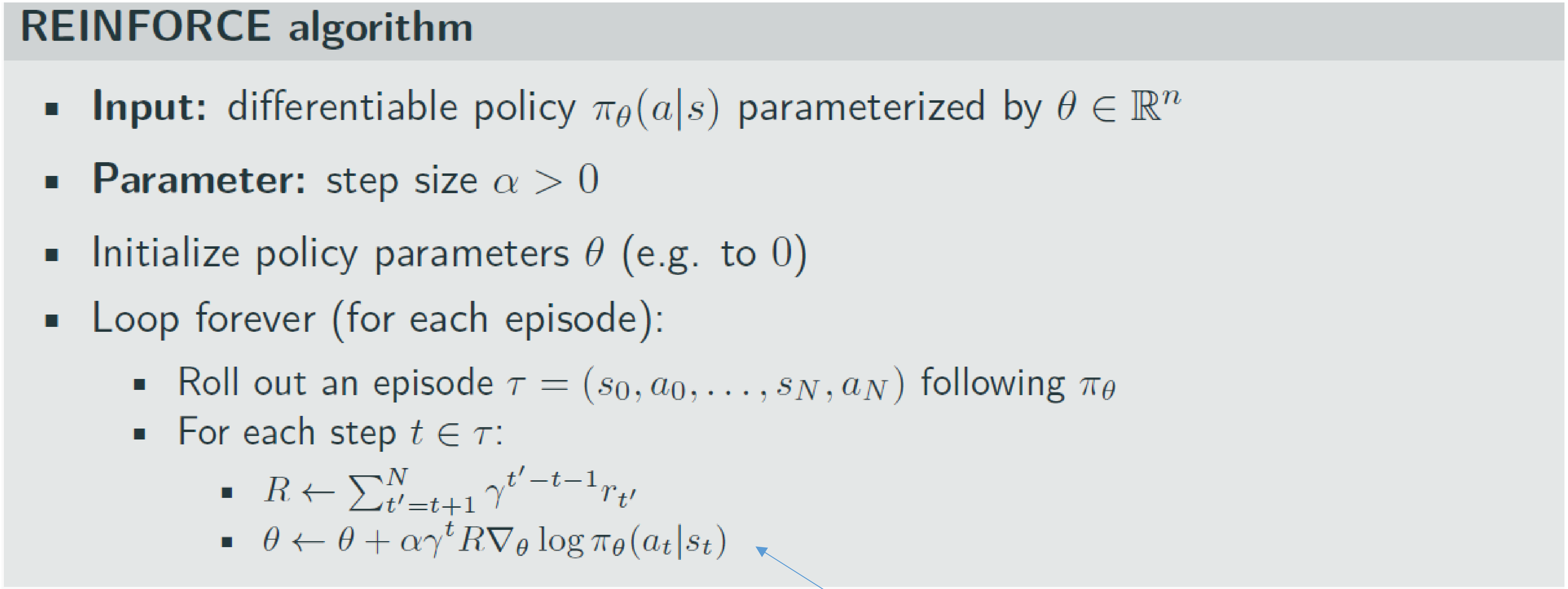
Policy-based algorithms update a policy iteratively. At each iteration k:

- Collect trajectories $\mathcal{T}_k = \{\tau\}$
- Update the policy $\pi_{k+1} = update(\pi_k, \mathcal{T}_k)$

Two ways to collect trajectories:

- **On-policy:** trajectories $\mathcal{T}_k$ are collected with the latest policy π_k
- **Off-policy:** trajectories $\mathcal{T}_k$ are collected with any policy

The REINFORCE algorithm



REINFORCE algorithm

- **Input:** differentiable policy $\pi_\theta(a|s)$ parameterized by $\theta \in \mathbb{R}^n$
- **Parameter:** step size $\alpha > 0$
- Initialize policy parameters θ (e.g. to 0)
- Loop forever (for each episode):
  - Roll out an episode $\tau = (s_0, a_0, \ldots, s_N, a_N)$ following π_θ
  - For each step $t \in \tau$:
    - $R \leftarrow \sum_{t'=t+1}^{N} \gamma^{t'-t-1} r_{t'}$
    - $\theta \leftarrow \theta + \alpha \gamma^t R \nabla_\theta \log \pi_\theta(a_t|s_t)$

Dropping the expectation

Vanilla policy gradient [Ach18]

```
Data: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, \ldots$ do
    Collect trajectories $\mathcal{D}_k = \{\tau_i\}$ by running $\pi_\theta = \pi(\theta_k)$;
    Compute rewards-to-go $\hat{R}_t$ and advantage estimates $\hat{A}_t$ based on $V_{\phi_k}$;
    Estimate the policy gradient as
        $$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$$
    Update policy parameters by e.g. gradient ascent, $\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k$;
    Fit value function by regression on mean-square error:
        $$\phi_{k+1} = \arg\min_\phi \frac{1}{T|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$$
end
```

$$\hat{A}_t = \hat{R}_t - V_{\phi_k}(s_t)$$

[Ach18] : Josh Achiam, Spinning Up in Deep Reinforcement Learning. https://spinningup.openai.com