

Introduction to Reinforcement Learning

Dr. Andrea Santamaria Garcia

SUPERVISED LEARNING

Classification, prediction, forecasting
computer learns by example



- Spam detection
- Weather forecasting
- Housing prices prediction
- Stock market prediction

UNSUPERVISED LEARNING

Segmentation of data
computer learns without prior information about the data

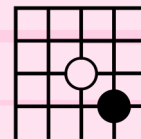


- Medical diagnosis
- Fraud (anomaly) detection
- Market segmentation
- Pattern recognition

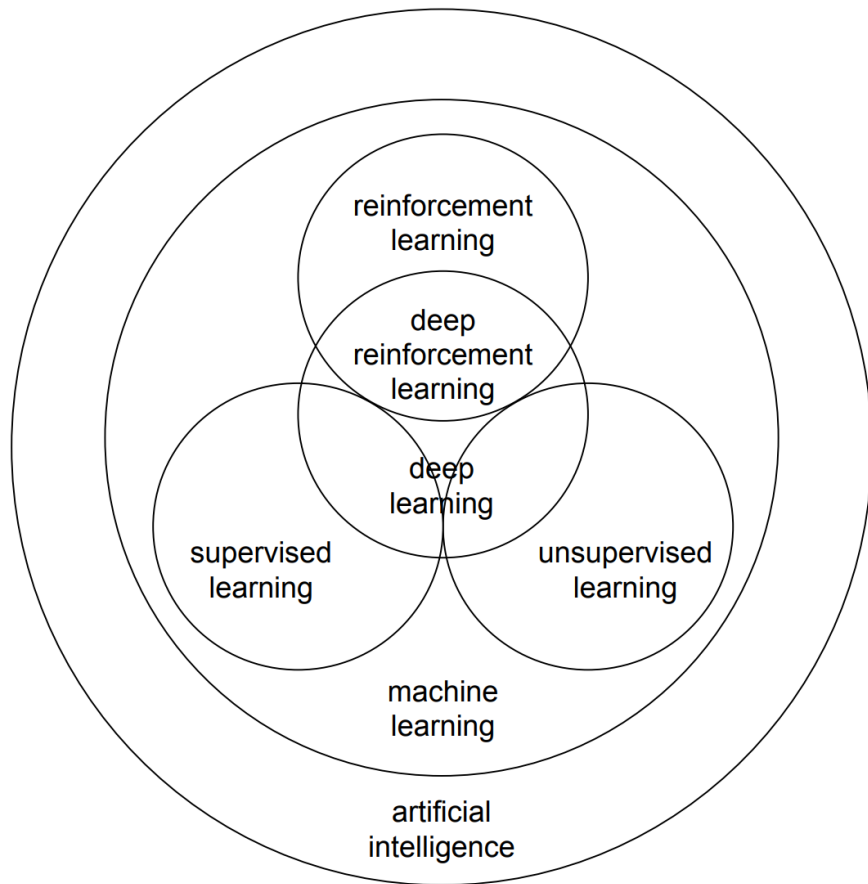
MACHINE LEARNING

REINFORCEMENT LEARNING

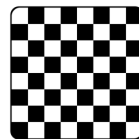
Real-time decisions
computer learns through trial and error



- Self-driving cars
- Make financial trades
- Gaming (AlphaGo)
- Robotics manipulation



Chess

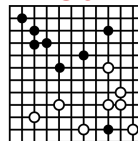


DeepBlue beat Kasparov (1997)

- 2e8 positions per second
- 700,000 grandmaster games
 - tabular type
- Alpha-beta pruning (search algorithm)

~10⁴⁰ possible moves

Go



AlphaGo beat Lee Sedol (2016)

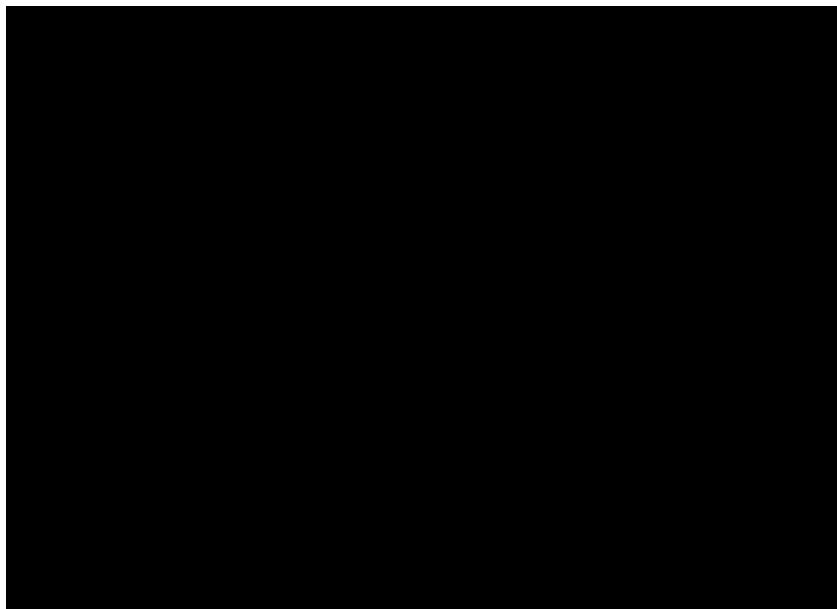
- Uses RL concepts:
 - Value network to evaluate positions
 - Policy network to select moves
 - Trained with human expert games and self-play with RL

~10¹⁷⁰ possible moves

Reinforcement learning

understanding how the human brain learns makes decisions

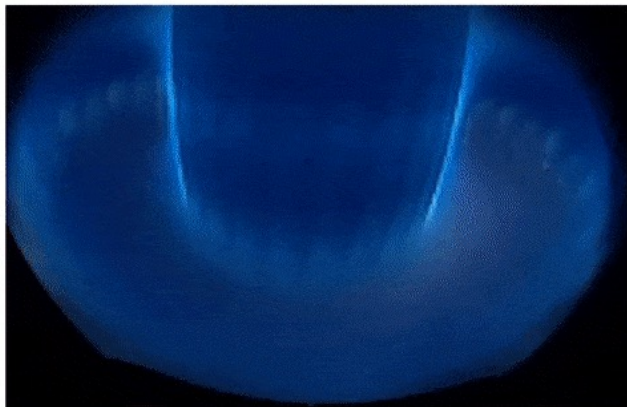
<https://arxiv.org/abs/1707.02286>



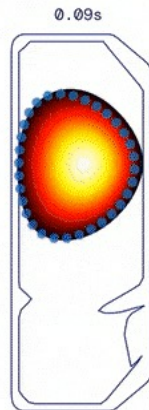
<https://www.deepmind.com/publications/playing-atari-with-deep-reinforcement-learning>



Control the plasma in a tokamak fusion reactor



View from inside the tokamak



Plasma state reconstruction

ChatGPT: Optimizing Language Models for Dialogue

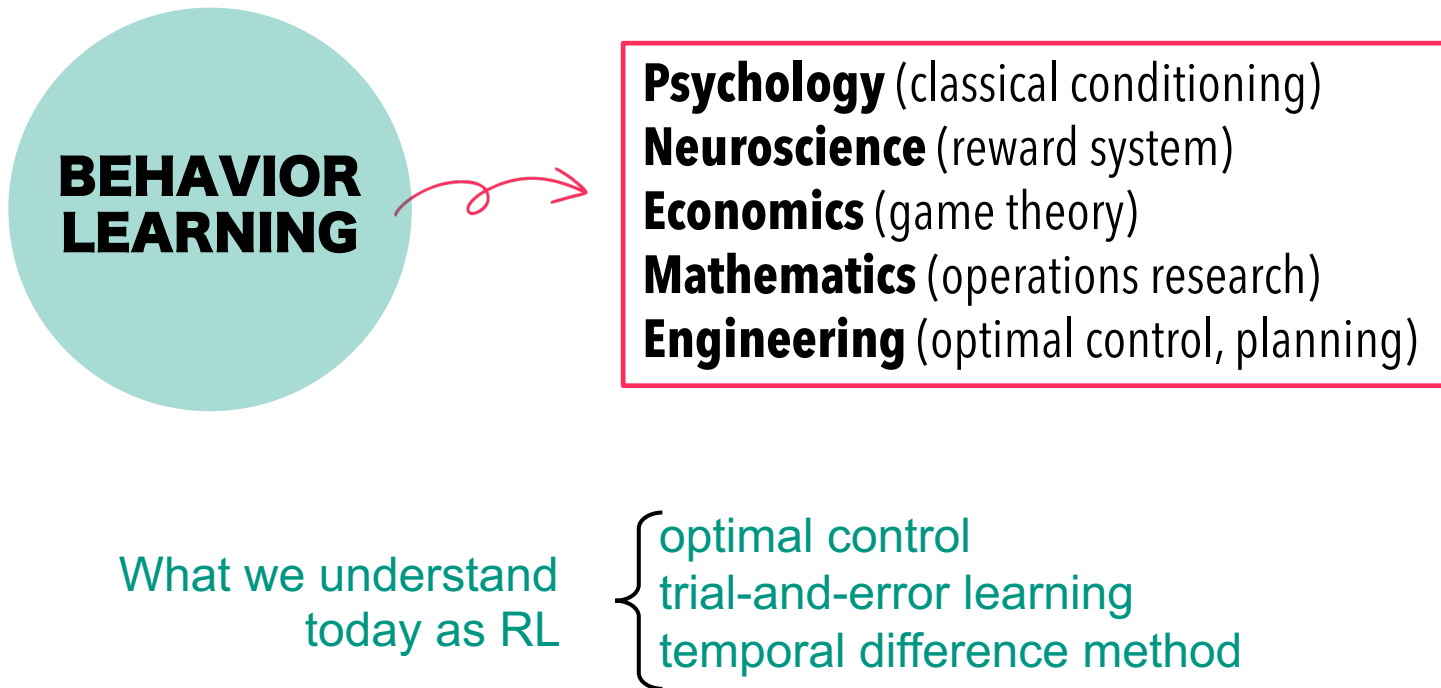
Methods

We trained this model using **Reinforcement Learning** from Human Feedback (RLHF), using the same methods as InstructGPT, but with slight differences in the data collection setup. We trained an initial model using supervised fine-tuning: human AI trainers provided conversations in which they played both sides—the user and an AI assistant. We gave the trainers access to model-written suggestions to help them compose their responses. We mixed this new dialogue dataset with the InstructGPT dataset, which we transformed into a dialogue format.

To create a reward model for **reinforcement learning**, we needed to collect comparison data, which consisted of two or more model responses ranked by quality. To collect this data, we took conversations that AI trainers had with the chatbot. We randomly selected a model-written message, sampled several alternative completions, and had AI trainers rank them. Using these reward models, we can fine-tune the model using **Proximal Policy Optimization**. We performed several iterations of this process.

Reinforcement learning

more than machine learning

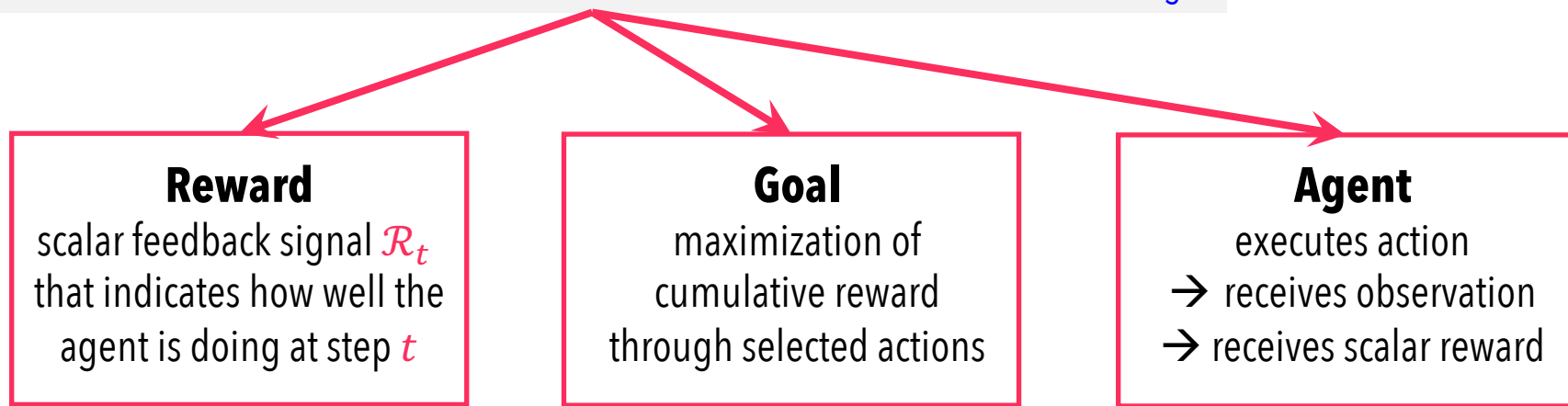


The RL problem

Reward hypothesis

all goals can be described by the maximization of expected cumulative sum of a received scalar signal

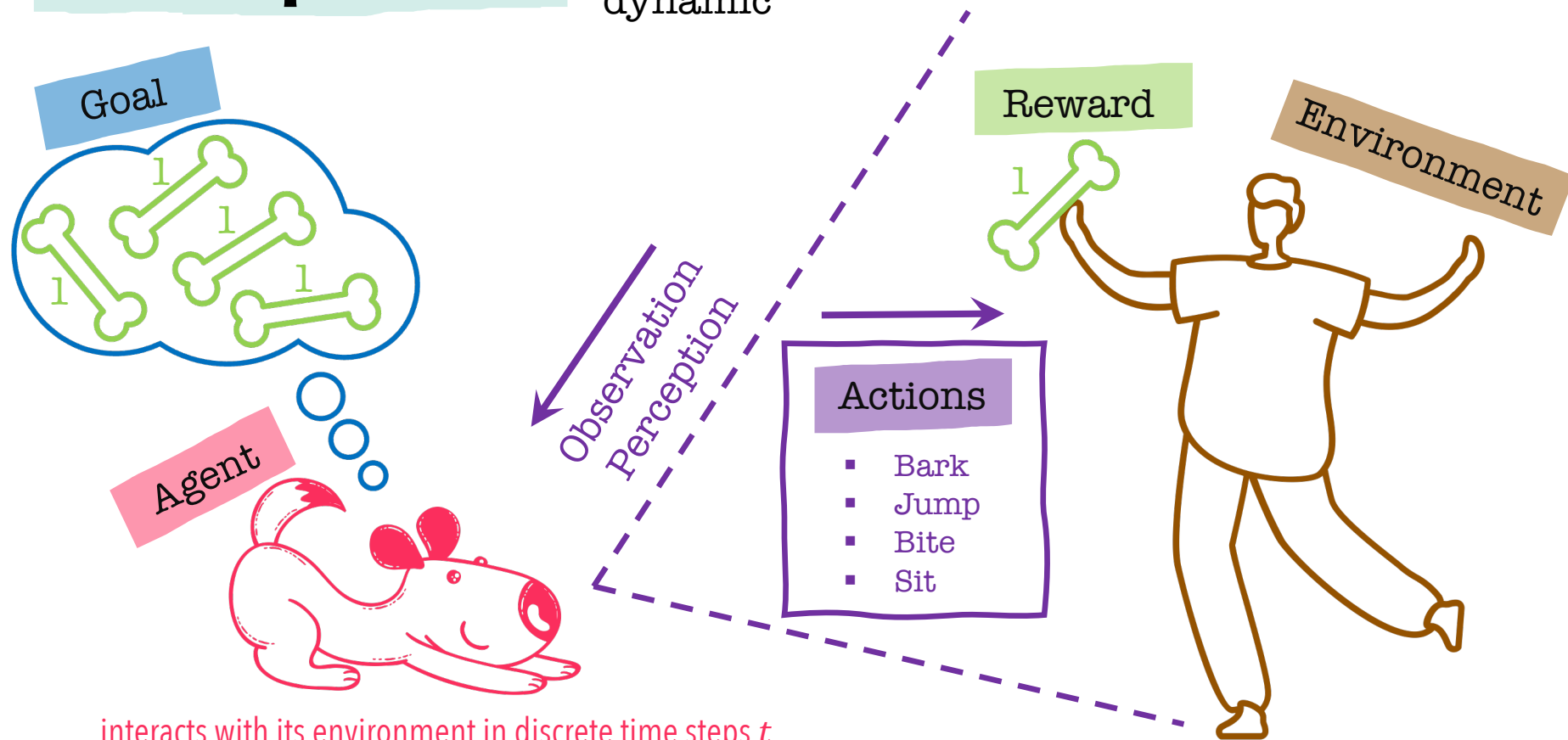
"Reward is enough"



an agent must learn through trial-and-error interactions with a dynamic environment

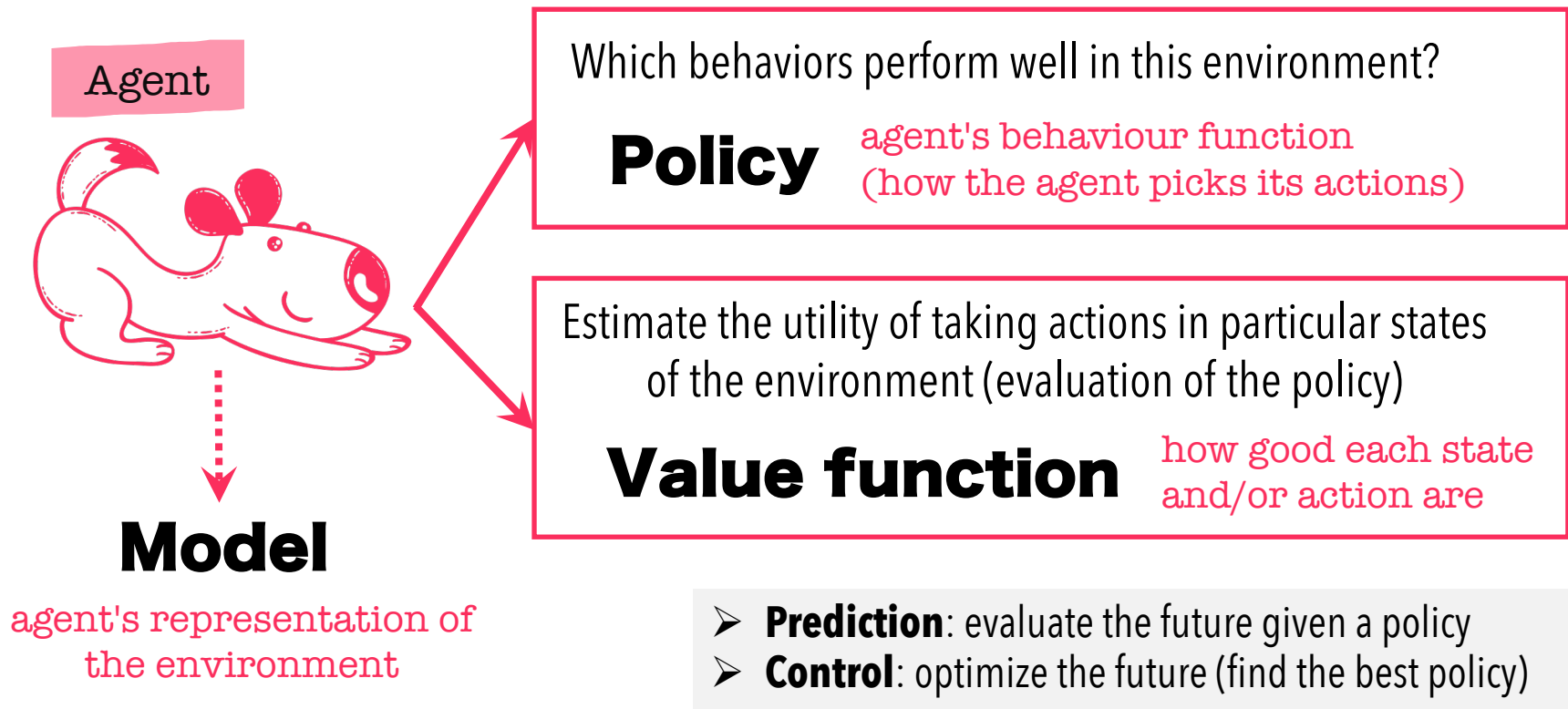
The RL problem

interactive
dynamic



interacts with its environment in discrete time steps t

How to cumulate reward?



Challenges in RL

Trade-off between exploitation and exploration

- Actions may have long-term consequences
- Reward might be delayed (does not happen immediately)

➡ should the agent sacrifice immediate reward to gain more long term reward?

The agent needs to:

- ✓ **Exploit** what it has already experienced in order to obtain reward now
- ✓ **Explore** the environment to select better actions in the future by sacrificing known reward now

...and both cannot be pursued exclusively without failing at the task

The agent

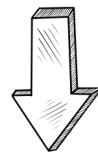
Must:

- Be able to **sense the state** of its environment to some extent
- Be able to **take actions** that affect that state
- **Have a goal** or goals relating to the state of the environment

Sensation

“Free-will”

Motivation



Markov Decision Processes

Include this 3 elements without trivializing any of them

Markov Decision Process (MDP)

Mathematical framework for modelling sequential decision making

A Markov Decision Process is a 5-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}_{ss'}^a, \mathcal{R}_s^a, \gamma)$ \mathcal{S} = finite set of states

State

information used to determine what happens next

A state transition can be:

- **Deterministic** $s_{t+1} = f(\mathcal{H}_t)$
- **Stochastic** $s_{t+1} \sim \mathbb{P}(s_{t+1} | \tau_t)$

Trajectory

sequence of states and actions until time t

$$\tau = (s_0, a_0, s_1, a_1, s_2, a_2, \dots)$$

Environment state (\mathcal{S}^e): environment's internal representation, usually not visible to the agent

Agent state (\mathcal{S}^a): agent's internal representation, used by the RL algorithm to pick the next action

Observation (\mathcal{O}): partial description of a state, which may omit information

Markov Decision Process (MDP)

Mathematical framework for modelling sequential decision making

A Markov Decision Process is a 5-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}_{ss'}^a, \mathcal{R}_s^a, \gamma)$ \mathcal{S} = finite set of states

State

information used to determine what happens next

A state transition can be:

- **Deterministic** $s_{t+1} = f(\mathcal{H}_t)$
- **Stochastic** $s_{t+1} \sim \mathbb{P}(s_{t+1} | \tau_t)$

Trajectory

sequence of states and actions until time t

$$\tau = (s_0, a_0, s_1, a_1, s_2, a_2, \dots)$$

Markov state / property

A state is Markov if and only if:

$$\mathbb{P}[s_{t+1} | s_t] = \mathbb{P}[s_{t+1} | s_1, \dots, t]$$

- The state is a sufficient statistic of the future
- The future is independent of the past, given the present
- Once the state is known, the history may be discarded

state transitions of an MDP satisfy the Markov property



Fully observable environments

$$\mathcal{O}_t = \mathcal{S}_t^a = \mathcal{S}_t^e$$

- Agent directly observes environment state
- Necessary condition to formalize an RL problem with an MDP

Partially observable environments

$$\mathcal{S}_t^a \neq \mathcal{S}_t^e$$

Agent constructs its own state representation:

- Complete trajectory: $\mathcal{S}_t^a = \tau_t$
- Beliefs of environment state: $\mathcal{S}_t^a = (\mathbb{P}[\mathcal{S}_t^e = s_1], \dots, \mathbb{P}[\mathcal{S}_t^e = s_n])$
- Recurrent neural networks: $\mathcal{S}_t^a = \sigma(w_0 \mathcal{O}_t + w_s \mathcal{S}_{t-1}^a)$

→ Partially observable MDP

Markov Decision Process (MDP)

Mathematical framework for modelling sequential decision making

A Markov Decision Process is a 5-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}_{ss'}^a, \mathcal{R}_s^a, \gamma)$

State transition model / probability

Predicts the next state
(dynamics of the environment)

$\mathcal{P}_{ss'}^a = \mathbb{P}[\mathcal{S}_{t+1} = s' | \mathcal{S}_t = s, \mathcal{A} = a]$ Probability of ending in state s' after taking action a while being in state s

$$\mathcal{P} = \begin{pmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{pmatrix} \Sigma=1$$

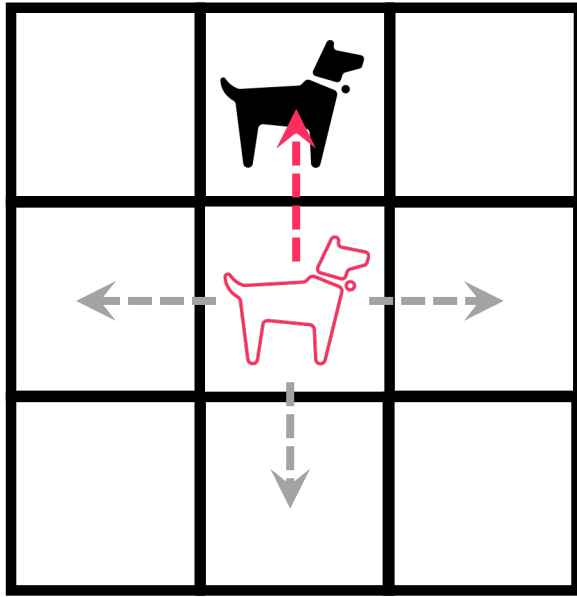
If probabilities change overtime
= **non-stationary Markov process**

Transition probabilities from all states and successor states

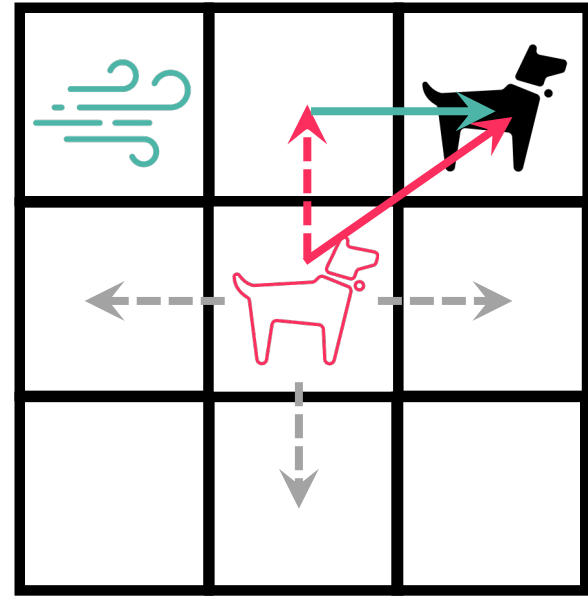
Non-deterministic environment

Taking the same action in the same state on two different occasions may result in different next states

$t = t_0$



$t = t_0 + \tau$



Markov Decision Process (MDP)

Mathematical framework for modelling sequential decision making

A Markov Decision Process is a 5-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}_{ss'}^a, \mathcal{R}_s^a, \gamma)$

Return

Total discounted reward
from time step t

$$\begin{aligned} G_t &= \mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \dots \\ &= \sum_{t=0}^{\infty} \gamma^t \mathcal{R}_{t+1} \end{aligned}$$

“infinite-horizon discounted return”

The goal is to maximize the return

- The discount factor $\gamma \in [0, 1)$ avoids infinite returns (sum converges)
- It values immediate reward over delayed reward (human-like)
- It deals with uncertainty about the future (no perfect model of env.)

Side notes:

- There are also undiscounted Markov processes if all sequences terminate (episodic)
- Model-based: there is an expectation of a reward (but not in model-free)

Policy

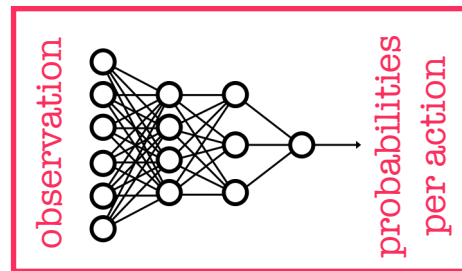
Map from state
to action

- Policy π completely defines how the agent will behave
- It's a distribution over actions given a certain state

Deterministic: $a = \pi(s)$

Stochastic: $\pi(a|s) = \mathbb{P}[\mathcal{A}_t = a | \mathcal{S}_t = s]$

Probability of taking a specific
action by being in a specific state



Categorical (discrete action spaces)
Gaussian (continuous action spaces)

Given an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π :

$$\mathcal{P}_{s,s'}^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{s,s'}^a \quad \mathcal{R}_s^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a$$

Value function

Estimation of expected future reward

A way to compare policies

- Used to choose between states depending on how much reward we expect to get
- Depends on the agent's behavior (policy)

State-value function

Expected return starting from state s and following policy π (evaluates the policy)

$$v_{\pi}(s) = \mathbb{E}_{\pi}[\mathcal{G}_t \mid \mathcal{S}_t = s]$$

given policy

Action-value function

Expected return starting from state s , taking action a , and following policy π

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}[\mathcal{G}_t \mid \mathcal{S}_t = s, \mathcal{A}_t = a]$$

"Q function"

Bellman optimality equation

The state-value function can be decomposed into:

- **immediate reward** \mathcal{R}_{t+1}
- **discounted value of next state** $\gamma v(\mathcal{S}_{t+1})$

$$\mathcal{V}(s) = \mathbb{E}[\mathcal{G}_t \mid \mathcal{S}_t = s]$$

$$= \mathbb{E}[\mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} \dots \mid \mathcal{S}_t = s]$$

$$= \mathbb{E}[\mathcal{R}_{t+1} + \gamma (\mathcal{R}_{t+2} + \gamma \mathcal{R}_{t+3} \dots) \mid \mathcal{S}_t = s]$$

$$= \mathbb{E}[\mathcal{R}_{t+1} + \gamma \mathcal{G}_{t+1} \mid \mathcal{S}_t = s] \quad \mathbb{E}(f) = \mathbb{E}(\mathbb{E}(f))$$

$$= \mathbb{E}[\mathcal{R}_{t+1} + \gamma \mathcal{V}(\mathcal{S}_{t+1}) \mid \mathcal{S}_t = s]$$



$$\mathcal{V}(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \mathcal{V}(s')$$

Reward you expect
to get from being in
your current state

Expected value of
wherever state
you land next

Bellman expectation equation

Considering the policy π we get:

$$v(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'}^a v(s') \right)$$

Direct solution only for small MDPs

- System of \mathcal{S} simultaneous linear equations with \mathcal{S} unknowns

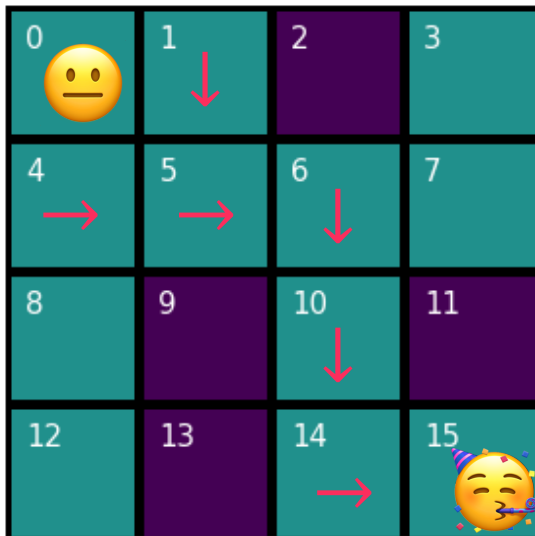
Other ways of solving it:









- Iteratively (dynamic programming)
- Sampling (Monte-Carlo evaluation)
- Approximation (temporal-difference learning)

Example: gridworld

The agent needs to get from state **0** to state **15** to get out of the maze

States

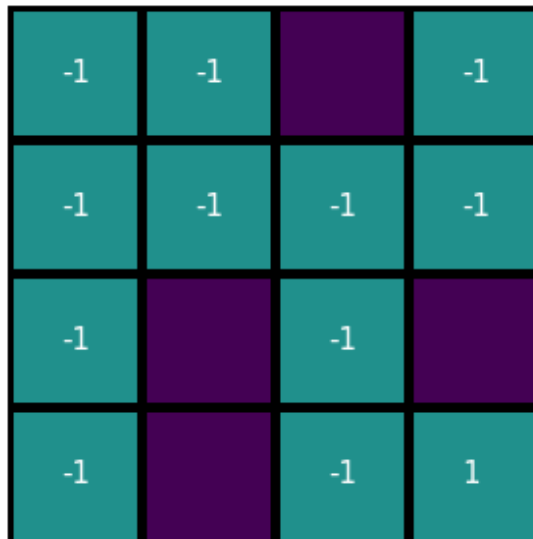


0 	1 	2	3
4 	5 	6 	7
8	9	10 	11
12	13	14 	15 

Actions $\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$

Deterministic env: $\mathcal{P}_{s,s'}^a = 1$

Rewards no discount γ



-1	-1		-1
-1	-1	-1	-1
-1		-1	
-1		-1	1

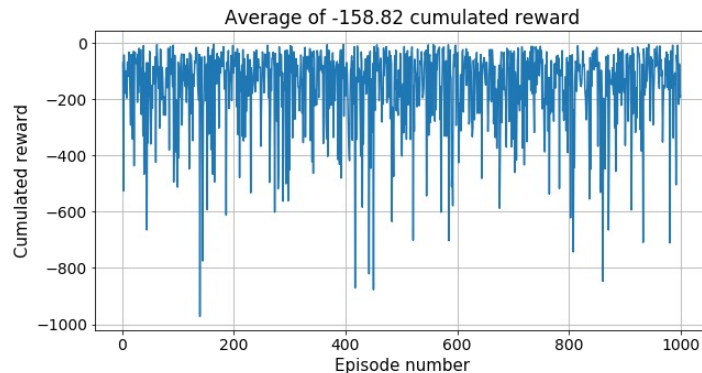
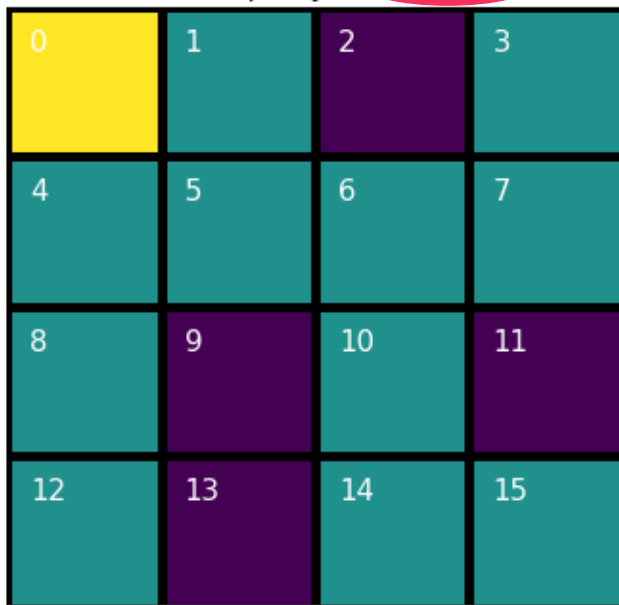
Example: gridworld

Policy $\pi(a|s) = \mathbb{P}[\mathcal{A}_t = a | \mathcal{S}_t = s] \rightarrow \pi(a|s) = \mathbb{P}[\uparrow, \downarrow, \leftarrow, \rightarrow | \mathcal{S}_t] = 0.25$

random policy

Random policy

Steps=1



Example: gridworld

Value function

$$v(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'}^a v(s') \right)$$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Solving simultaneously linear set of equations:

- environment's dynamics are completely known

$0.5*v_0 - 0.25*v_1 - 0.25*v_4 + 1.0 = 0$
 $-0.25*v_0 + 0.5*v_1 - 0.25*v_5 + 1.0 = 0$
 $0.25*v_3 - 0.25*v_7 + 1.0 = 0$
 $-0.25*v_0 + 0.75*v_4 - 0.25*v_5 - 0.25*v_8 + 1.0 = 0$
 $-0.25*v_1 - 0.25*v_4 + 0.75*v_5 - 0.25*v_6 + 1.0 = 0$
 $-0.25*v_{10} - 0.25*v_5 + 0.75*v_6 - 0.25*v_7 + 1.0 = 0$
 $-0.25*v_3 - 0.25*v_6 + 0.5*v_7 + 1.0 = 0$
 $-0.25*v_{12} - 0.25*v_4 + 0.5*v_8 + 1.0 = 0$
 $0.5*v_{10} - 0.25*v_{14} - 0.25*v_6 + 1.0 = 0$
 $0.25*v_{12} - 0.25*v_8 + 1.0 = 0$
 $-0.25*v_{10} + 0.5*v_{14} + 0.5 = 0$

11 variables, 11 equations



-154.0	-150.0		-130.0
-154.0	-142.0	-118.0	-126.0
-162.0		-82.0	
-166.0		-42.0	0.0

$\pi \rightarrow \mathcal{V}_\pi =$ policy evaluation

how much value this policy has?

Example: gridworld

Value function

$$v(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'}^a v(s') \right)$$

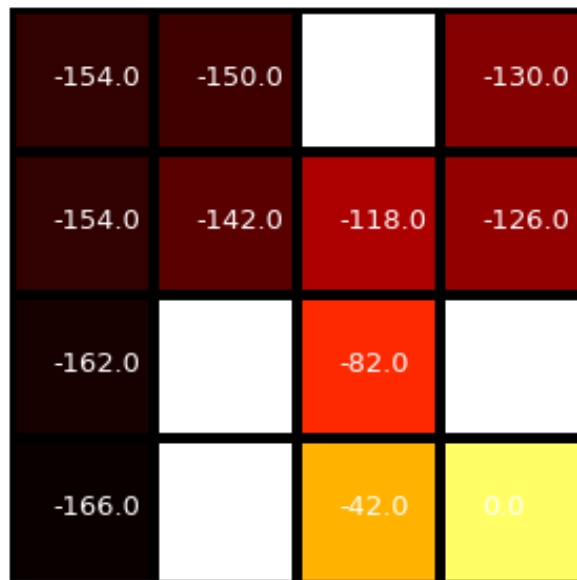
Solving iteratively:

- Bellman equation becomes an update rule

$$v_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'}^a v_k(s') \right)$$

```
Iterative Policy Evaluation, for estimating  $V \approx v_\pi$ 
Input  $\pi$ , the policy to be evaluated
 $V \leftarrow \vec{0}, V' \leftarrow \vec{0}$ 
Loop:
   $\Delta \leftarrow 0$ 
  Loop for each  $s \in \mathcal{S}$ :
     $V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$ 
     $\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$ 
   $V \leftarrow V'$ 
until  $\Delta < \theta$  (a small positive number)
Output  $V \approx v_\pi$ 
```

Coursera



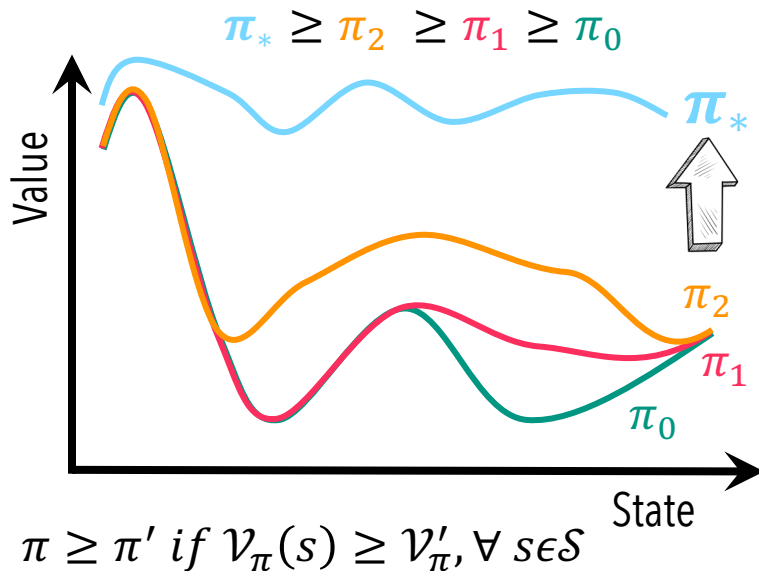
$\pi \rightarrow v_\pi = \text{policy evaluation}$
how much value this policy has?

Dynamic programming algorithms

turn the Bellman eq.
into update rules

✓ **Prediction:** what's the value for a specific policy?

- **Control:** which policy gives as much reward as possible?
→ the policy with more value!



For any MDP:

- There exists an optimal policy π_* that is better or equal to all other policies $\pi_* \geq \pi \forall \pi$
- All optimal policies achieve the optimal value function $\mathcal{V}_{\pi_*} = \mathcal{V}_*(s)$ and $Q_{\pi_*} = Q_*(s, a)$

So...do I have to calculate the value of every policy and compare them?

$|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies in an MDP

$4^{11} \approx 4$ million policies for simple gridworld example

Bellman optimality equations

$$\mathcal{V}_{\pi^*}(s) = \mathbb{E}_{\pi^*}[\mathcal{G}_t \mid \mathcal{S}_t = s] = \max_{\pi} \mathcal{V}_{\pi}(s) \quad \forall s \in \mathcal{S}$$

$$\mathcal{Q}_{\pi^*}(s) = \max_{\pi} \mathcal{Q}_{\pi}(s) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

Optimal value functions

By replacing the optimal policy on the Bellman equations we get:

$$\mathcal{V}_*(s) = \max_a \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \mathcal{V}_*(s') \right)$$

maximum value over every next possible state

$$\mathcal{Q}_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'}^a \max_{a'} \mathcal{Q}_*(s', a')$$

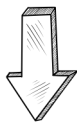
π_* assigns probability 1 to the action that receives the highest value

- Nonlinear (max), no closed-form solution
- Dynamic programming solutions only applicable if the dynamics of the system \mathcal{P} are known

Determining an optimal policy

$$v_*(s) = \max_a \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} v_*(s') \right)$$

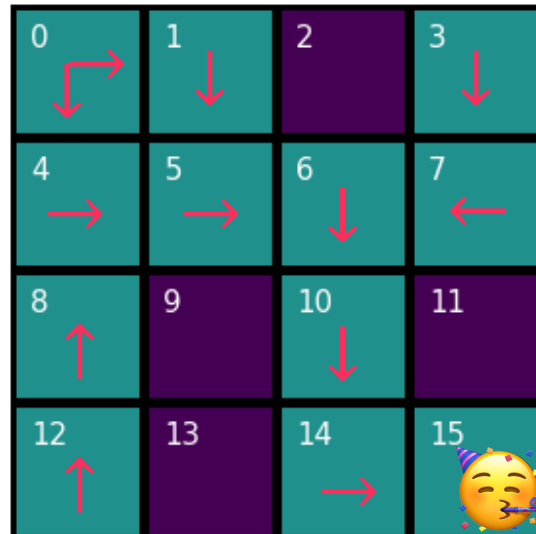
maximum over all actions



For any state we look at each available action and take the one that maximizes the argument

$$\pi_*(s) = \operatorname{argmax}_a \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} v_*(s') \right)$$

particular action that
achieves that maximum
(greedy action)



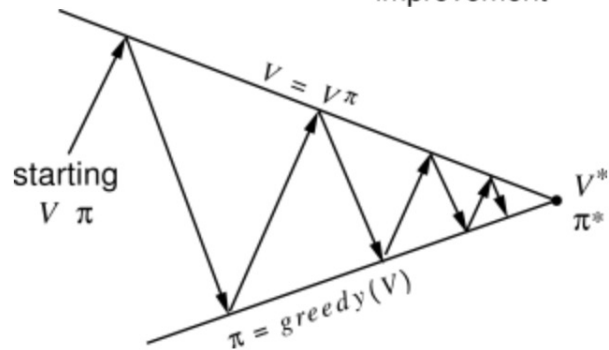
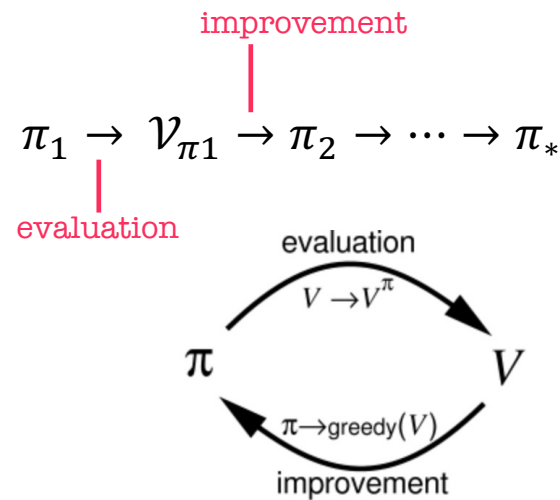
$$\pi_*(s) = \operatorname{argmax}_a Q_*$$

Policy improvement & iteration

Let's consider a value function \mathcal{V}_π that is non-optimal, and we select an action that is greedy with respect to it:

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \mathcal{V}_\pi(s') \right)$$

- If the action has a higher value, the policy is better
- \mathcal{V}_* is the unique solution to the Bellman optimality eq.
- If this greedy operation does not change \mathcal{V} , then it converged to the optimal policy because it satisfies the Bellman optimality eq.



Dynamic programming algorithms

turn the Bellman eq.
into update rules

Problem	Bellman equation	Algorithm	Sample-based version
Prediction	Expectation equation	Iterative policy evaluation	Temporal difference
Control	Expectation equation + greedy policy	Policy iteration	Sarsa
Control	Optimality equation	Value iteration	Q-learning

when we don't know \mathcal{P}

Off-policy learning

On-policy: improve and evaluate the policy being used to select actions

Off-policy: improve and evaluate a different policy from the one used to select actions

- Learn a **target policy** π (optimal policy) while...
- ...selecting actions from **behavior policy** b (exploratory policy)

Provides another strategy for continuous exploration (experiences a larger # of states)

Temporal difference learning

Learning method specialized for multi-step **prediction learning**

- TD learning is learning a prediction from another, later learned prediction
 - learning a guess from a guess (you don't know the true \mathcal{V})

$$\mathcal{V}(s) \leftarrow \mathcal{V}(s) + \alpha[\mathcal{R} + \gamma\mathcal{V}(s') - \mathcal{V}(s)]$$

- Difference between both predictions = temporal difference
- No \mathcal{P} model needed (unlike in dynamic programming)



- Allows you to estimate the value function before the episode is finished
- Making long-term predictions is exponentially complex
 - Memory scales with the #steps of the prediction
- TD model = standard model of reward systems in the brain

Q-learning

Off-policy TD control

$$Q(s, a) \leftarrow Q(s, a) + \alpha[\mathcal{R} + \gamma \max_a Q(s', a) - Q(s, a)]$$

Converges to the optimal value function as long as the agent continues to explore sampling the state-action space

Overview of RL methods

Tabular solution methods

- Iterative (dynamic programming)
- Sample-based (Monte-Carlo evaluation)
- Temporal-difference learning

- Used to solve finite MDPs
- Value functions are stored as arrays (tables)
- Methods can often find exact solutions

In real-life situations, we cannot store the values of each possible state in an array, especially in continuous problems

- Autonomous driving: array per possible image the camera sees?

Approximate solution methods

- Value-based
- Policy gradient
- Policy-based
- Actor-critic

- Approximate value by function parametrized by a weight vector
--> **neural networks (learning!)**
- Applicable to partially observable problems

Approximate solution methods

Value-based

contains a value function,
policy is implicit

- Sample efficient
- Computationally fast
- Unstable (bias, don't know true \mathcal{V})

DQN, NAF

Policy-based

does not store the value
function, only the policy

Policy gradient

optimizes parametrized
policies with gradient descent

- Convergence guarantees
- Sensitive to stepsize choice
- Poor sample efficiency
- Large variance

PPO, TD3,
DDPG

Actor-critic

stores both the policy
and value function

ACER, A2C/A3C, SAC

	Description	Policy	Action space	State space	Operator
DQN	Deep Q Network	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	On-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor Critic	Off-policy	Continuous	Continuous	Advantage

Thank you for your attention!

What questions do you have for me?

- [Sutton & Barto book](#)
- <https://arxiv.org/pdf/cs/9605103.pdf>
- [Reinforcement learning lectures by David Silver](#)
- <https://spinningup.openai.com/en/latest/>
- [Coursera RL specialization](#)
- <https://arxiv.org/pdf/1810.06339.pdf>

Let's connect! [@andrea.santamaria@kit.edu](mailto:andrea.santamaria@kit.edu) / [@ansantam](#)