IntroML
Reinforcement learning

Manel Rodríguez-Soto
Juan A. Rodríguez-Aguilar
ICMAT@Madrid
28.04.2023

Objectives and schedule

Introduce key concepts about reinforcement learning. Markov decision processes, dynamic programming, Q-learning, Deep reinforcement learning.

Contents

- Tabular reinforcement learning
- Deep reinforcement learning
- Multi-agent reinforcement learning (if possible)

Refs

Sutton, Barto (2018) RL: An intro https://www.csee.umbc.edu/courses/graduate/678/spring17/RL-3.pdf

Zai, Brown (2020) Deep RL in action

Brilliant summary in

- https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html

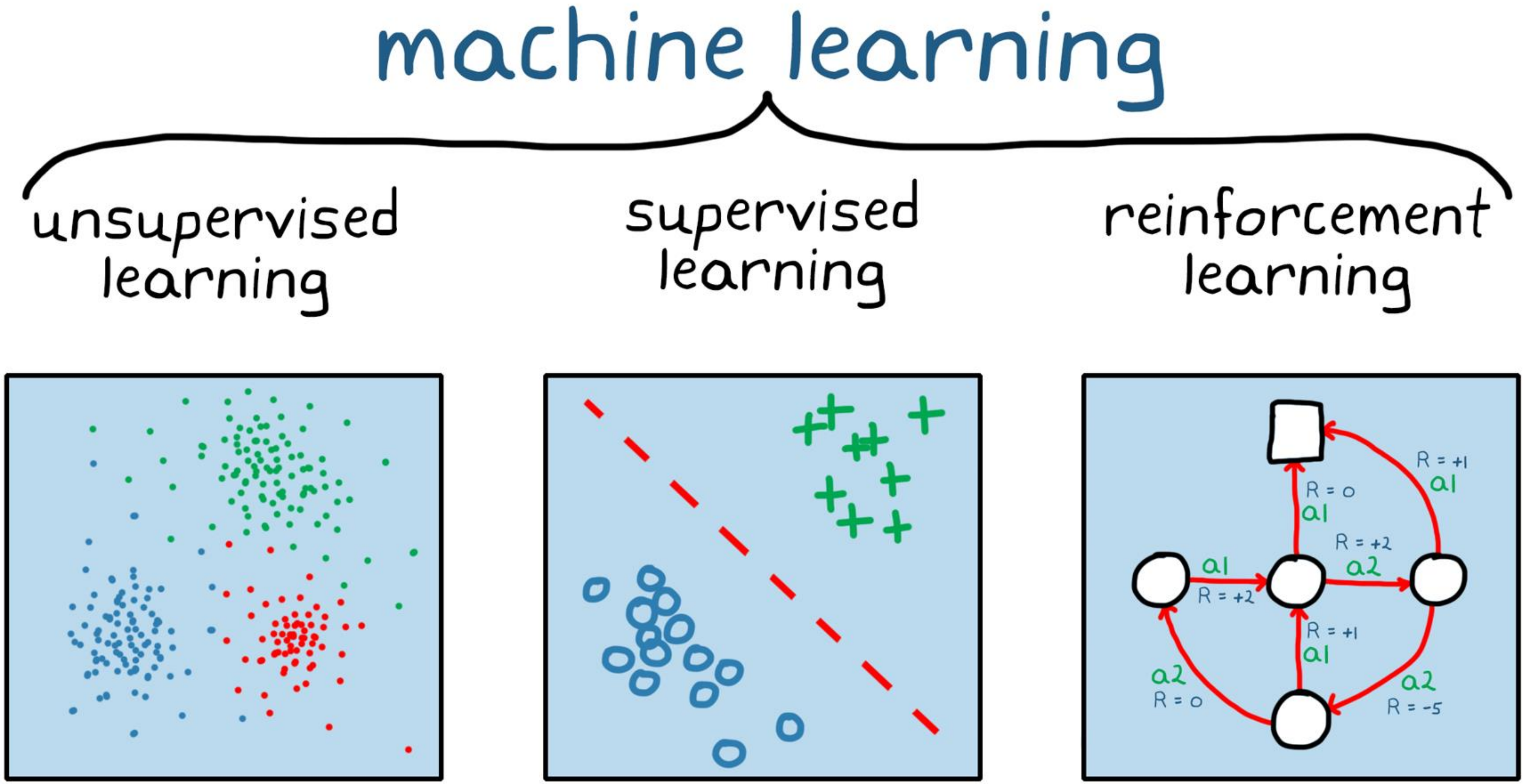
Videos

- https://www.youtube.com/watch?v=V1eYniJ0Rnk DeepMind DRL. Mnih et al
- https://www.youtube.com/watch?v=WXuK6gekU1Y Alphago
- https://www.youtube.com/watch?v=tCpf5wDr0UE AlphaZero

Section 1: Classic and tabular

Manel Rodríguez-Soto
Juan A. Rodríguez-Aguilar
ICMAT@Madrid
28.04.2023

Motivation

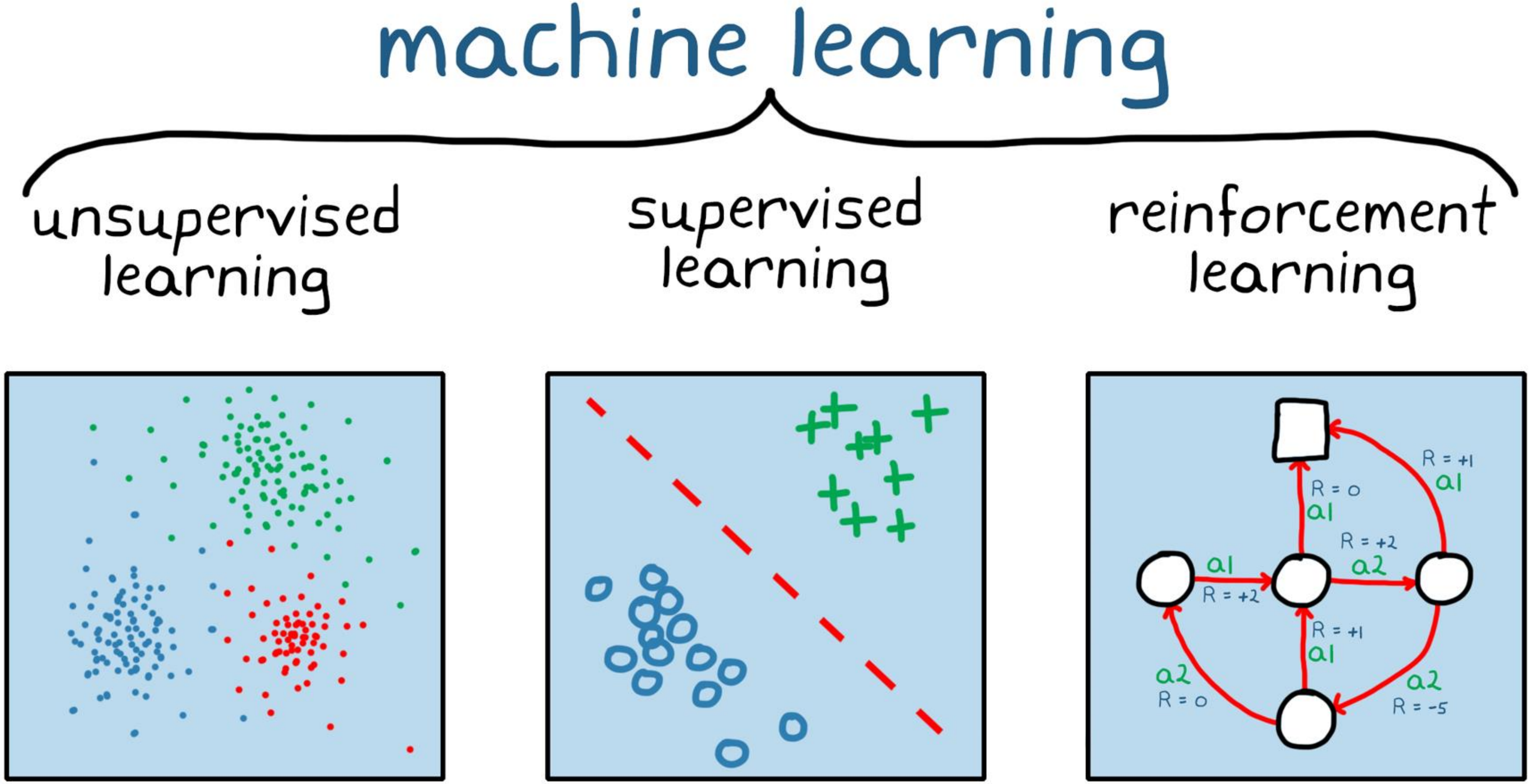


Clustering

Classification

Control

Manel Rodríguez-Soto & Juan A. Rodriguez-Aguilar, 2023



Unlabelled dataset

Labelled Dataset

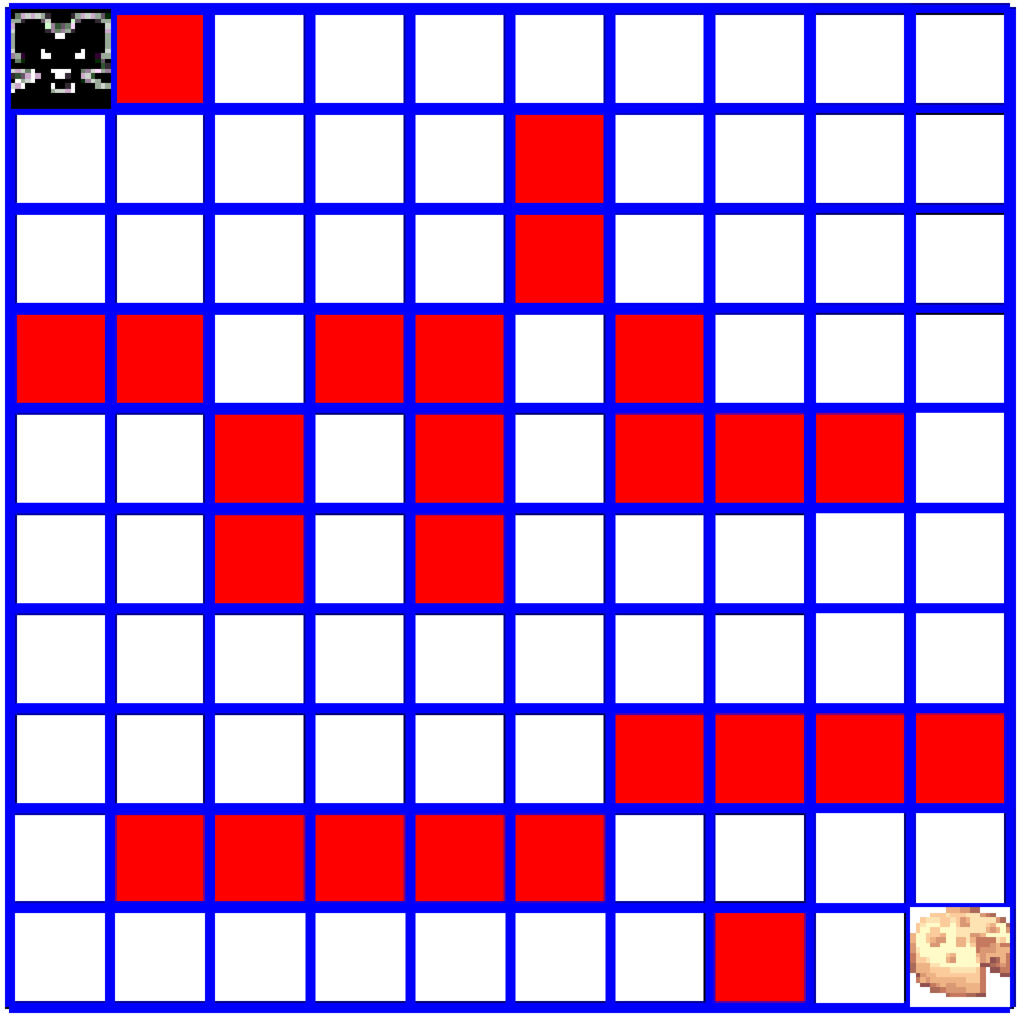
Environment (Rewards)

Reinforcement Learning Concept

Main characteristics of RL:

1. Goal: learning a behavior (policy)
2. Agent learns interacting with the environment
3. **Sequential** decision making

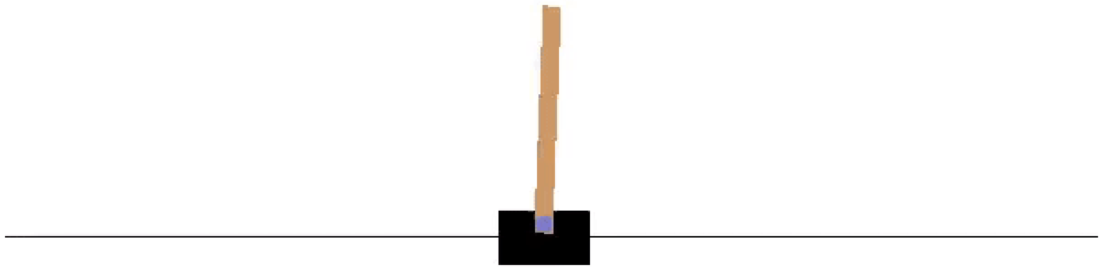
Examples



Maze

- +ive reward: get cheese

Examples



Cartpole

- +ive reward: stay up

Examples

GO

- +ive reward: win
- -ive reward: lose

Reinforcement Learning Concept

Main characteristics of RL:

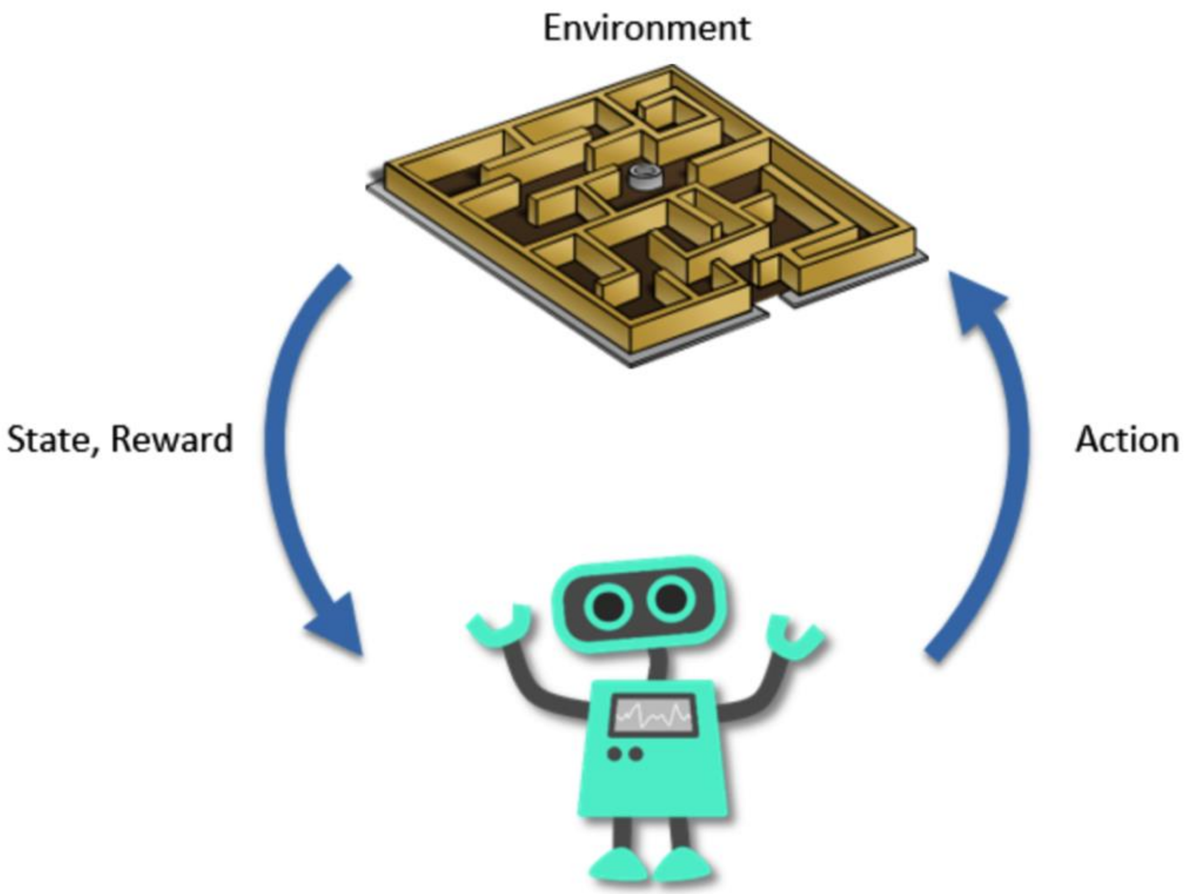
1. Goal: learning a behavior (policy)
2. Agent learns interacting with the environment
3. **Sequential** decision making

RL features

- Optimal control of decision making problems that are:
  - Sequential
  - Stochastic
  - Unknown probabilities

How??

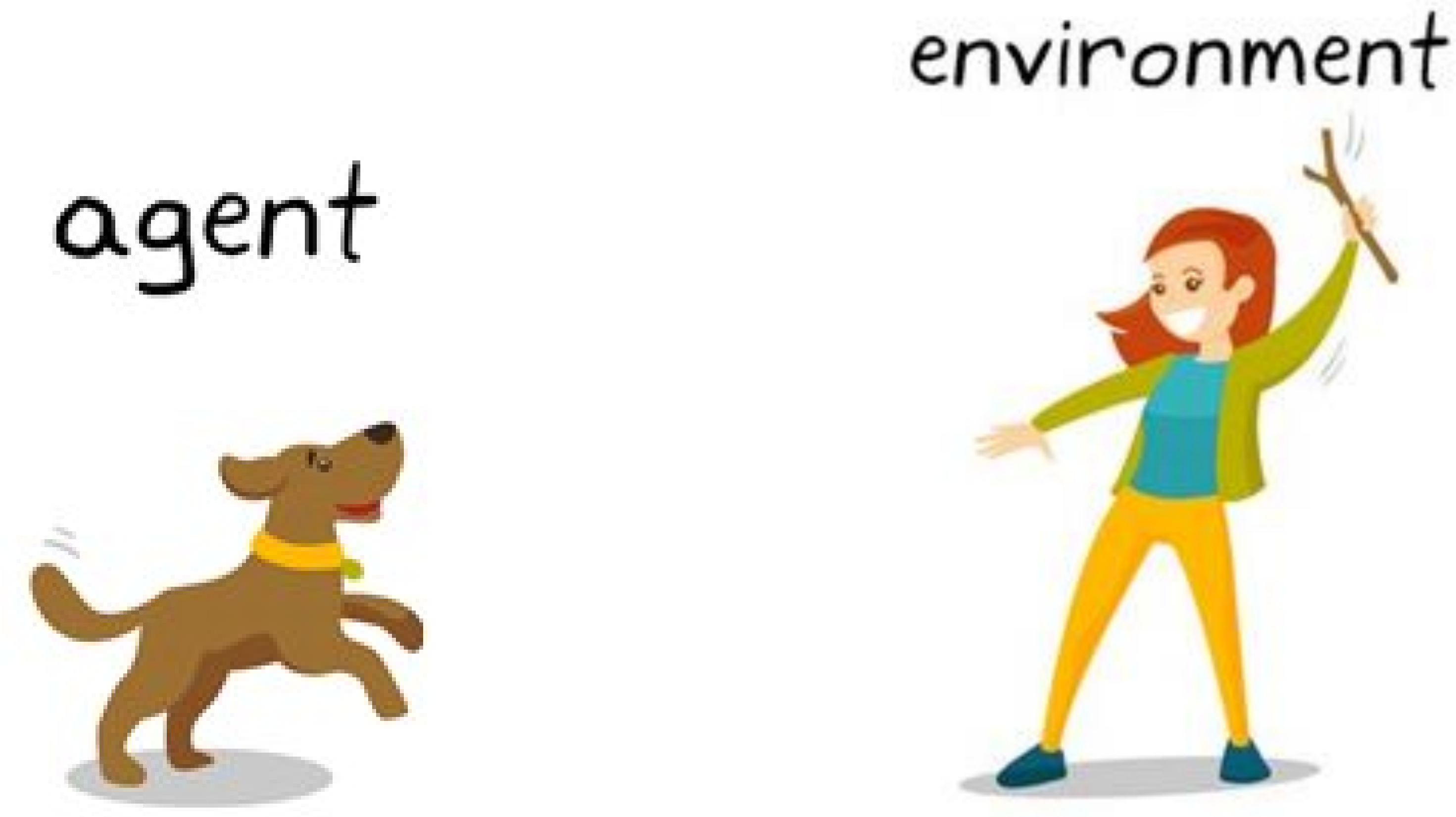
RL features



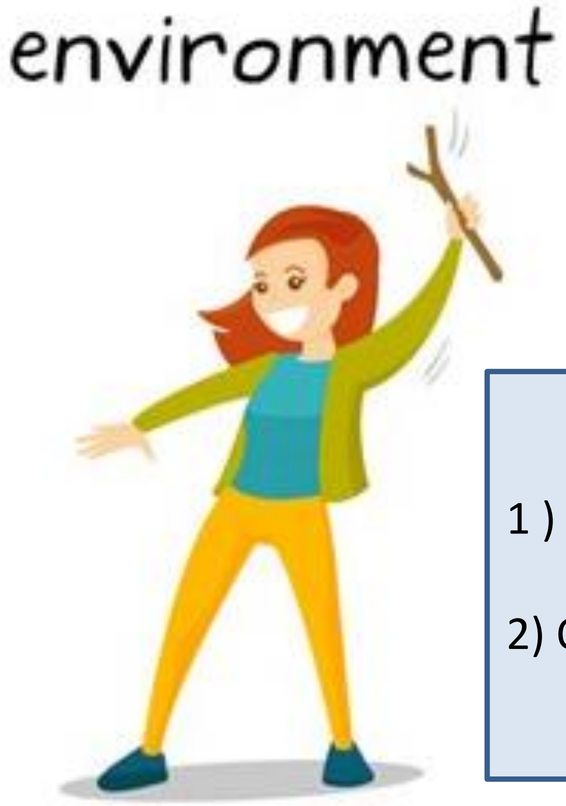
Agent

Reinforcement Learning Concept

Reinforcement Learning Concept



Reinforcement Learning Concept

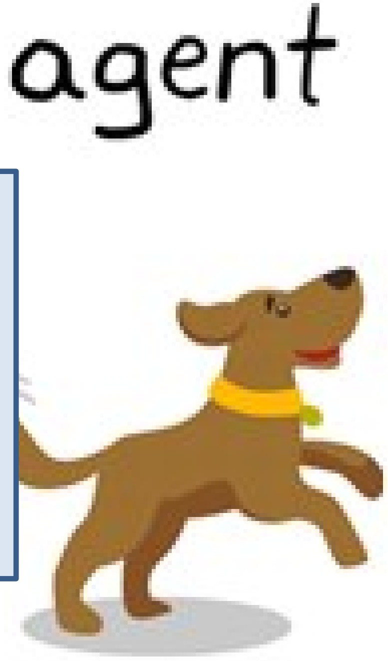


TWO POSSIBLE **STATES**:

1) OWNER HAS STICK

2) OWNER DOES NOT HAVE IT

Reinforcement Learning Concept



TWO POSSIBLE **ACTIONS**:

1) WAIT

2) FETCH

What happens after an action (I)

IN RL, the ***action*** *performed by the agent can* ***modify*** *the environment*

What happens after an action (I)

*IN RL, the **action** performed by the agent can **modify** the environment*

*Formally we say that the agent's **actions** can change the **state** of the environment*

State Transitions

STATE: OWNER DOES NOT HAVE STICK

ACTION: FETCH

NEW STATE: ????

State Transitions

STATE: OWNER DOES NOT HAVE STICK

ACTION: FETCH

NEW STATE: OWNER HAS STICK

What happens after an action (II)

IN RL, an agent receives a reward depending on the ***action*** *it performs...*

What happens after an action (II)

IN RL, an agent receives a reward depending on the ***action*** *it performs...*

...and the reward also depends on the ***state*** *you are in!*

Rewards

STATE: OWNER HAS STICK

ACTION: ???

REWARD: ????

Rewards

STATE: OWNER HAS STICK

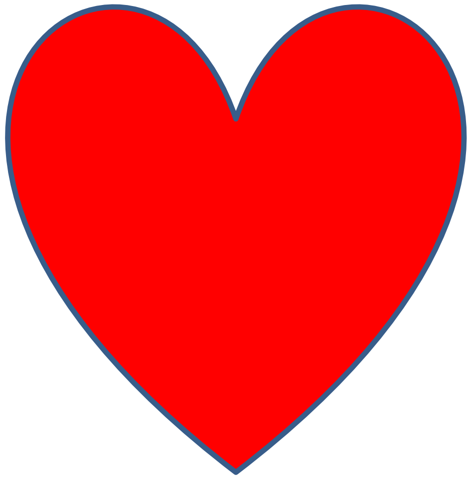
ACTION: WAIT

REWARD: ????

Rewards

STATE: OWNER HAS STICK

ACTION: WAIT



REWARD: POSITIVE

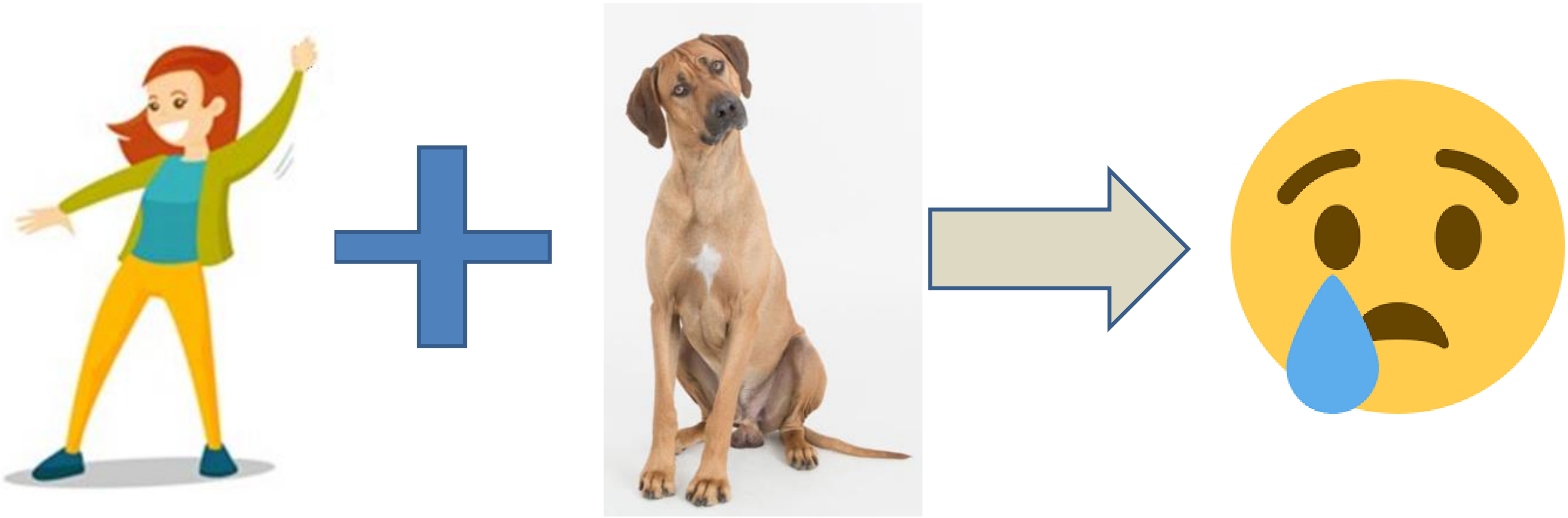
Rewards

STATE: OWNER DOES NOT HAVE STICK

ACTION: FETCH

REWARD: POSITIVE

Reinforcement Learning Concept



STATE: OWNER DOES NOT HAVE STICK

ACTION: WAIT

REWARD: NEGATIVE

Reinforcement Learning Concept

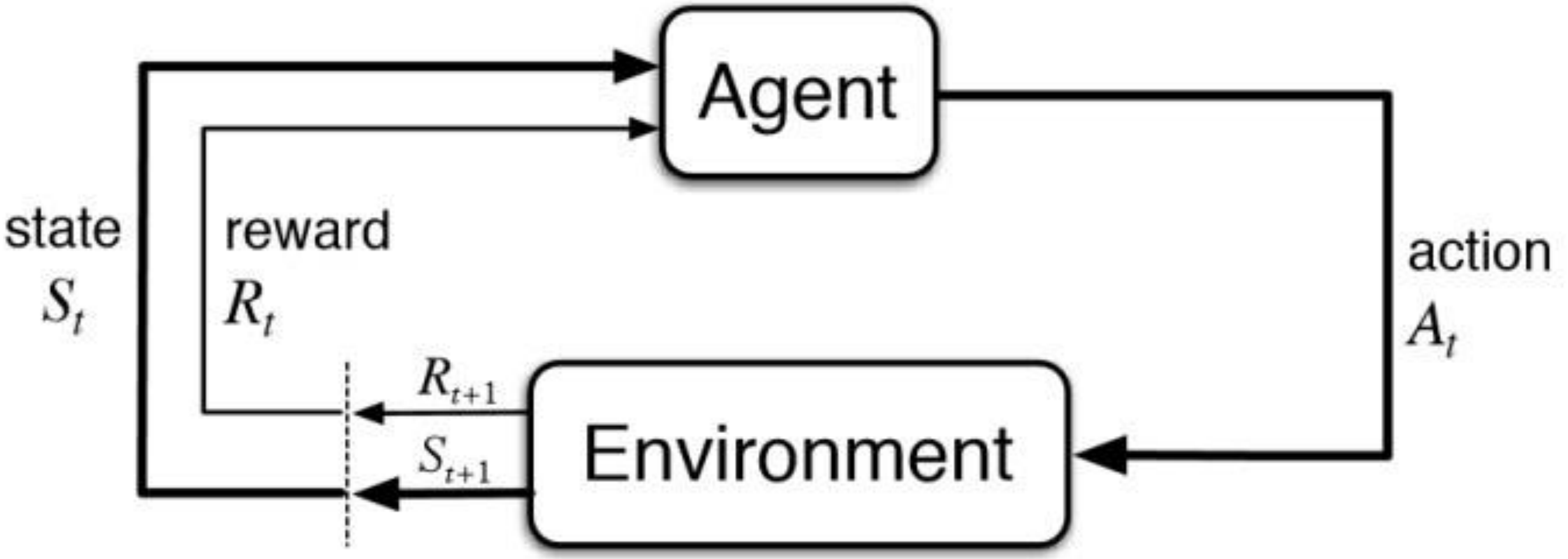
*At the beginning the agent **does not know** what will give it a good reward!*

Reinforcement Learning Concept

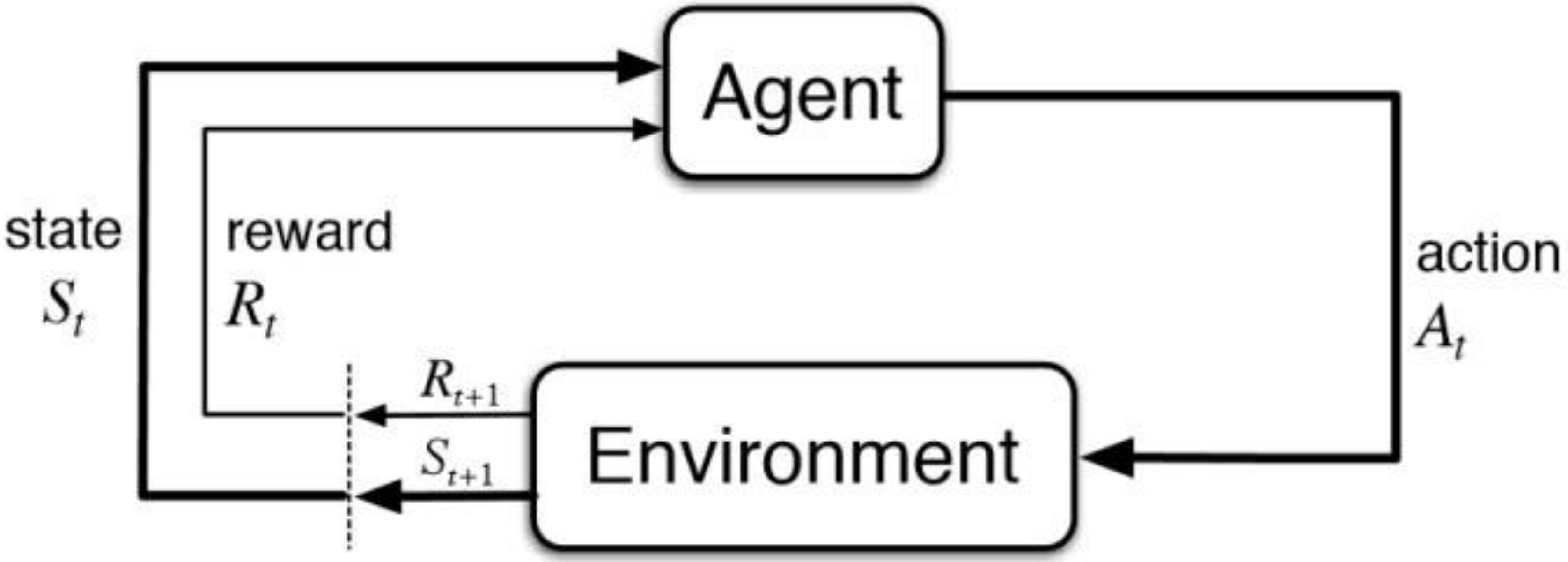
*At the beginning the agent **does not know** what will give it a good reward!*

*It can only learn via **trial and error***

Reinforcement Learning Concept



Reinforcement Learning Concept



MARKOV DECISION PROCESS

Markov Decision Processes

Markov Decission Process

$$\langle S, \mathcal{A}, T, R \rangle$$

STATES
ACTIONS

TRANSITION FUNCTION
REWARD FUNCTION

SETS

MODEL = FUNCTIONS

Transition Function (deterministic)

$$T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$$

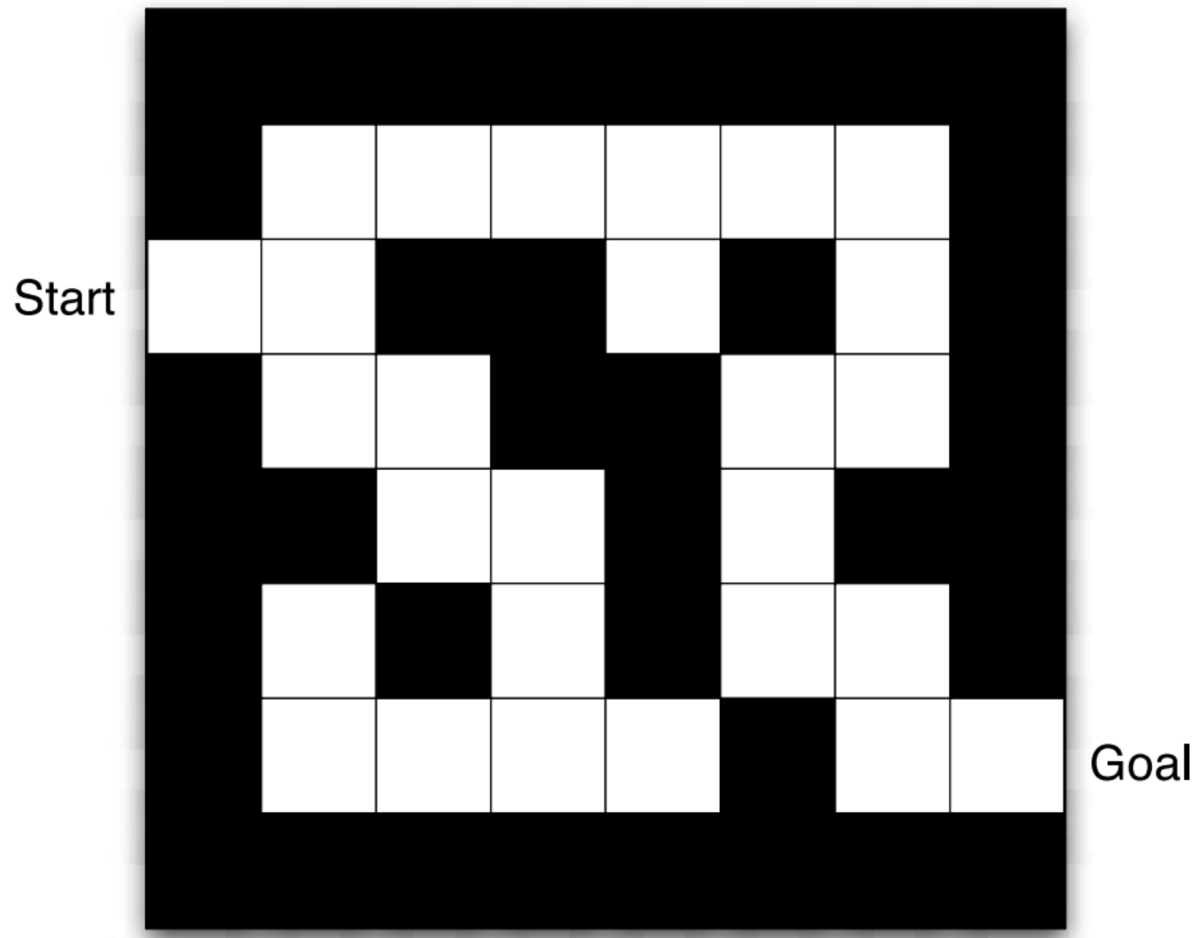
Transition Function (stochastic)

$$T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$$

Reward function

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

Example



- **Rewards**: -1 per time-step
- **Actions**: N, S, W, E
- **States**: Agent’s location
- **Transitions**: deterministic

Policies

We formalise the behaviour of an agent as a **policy**

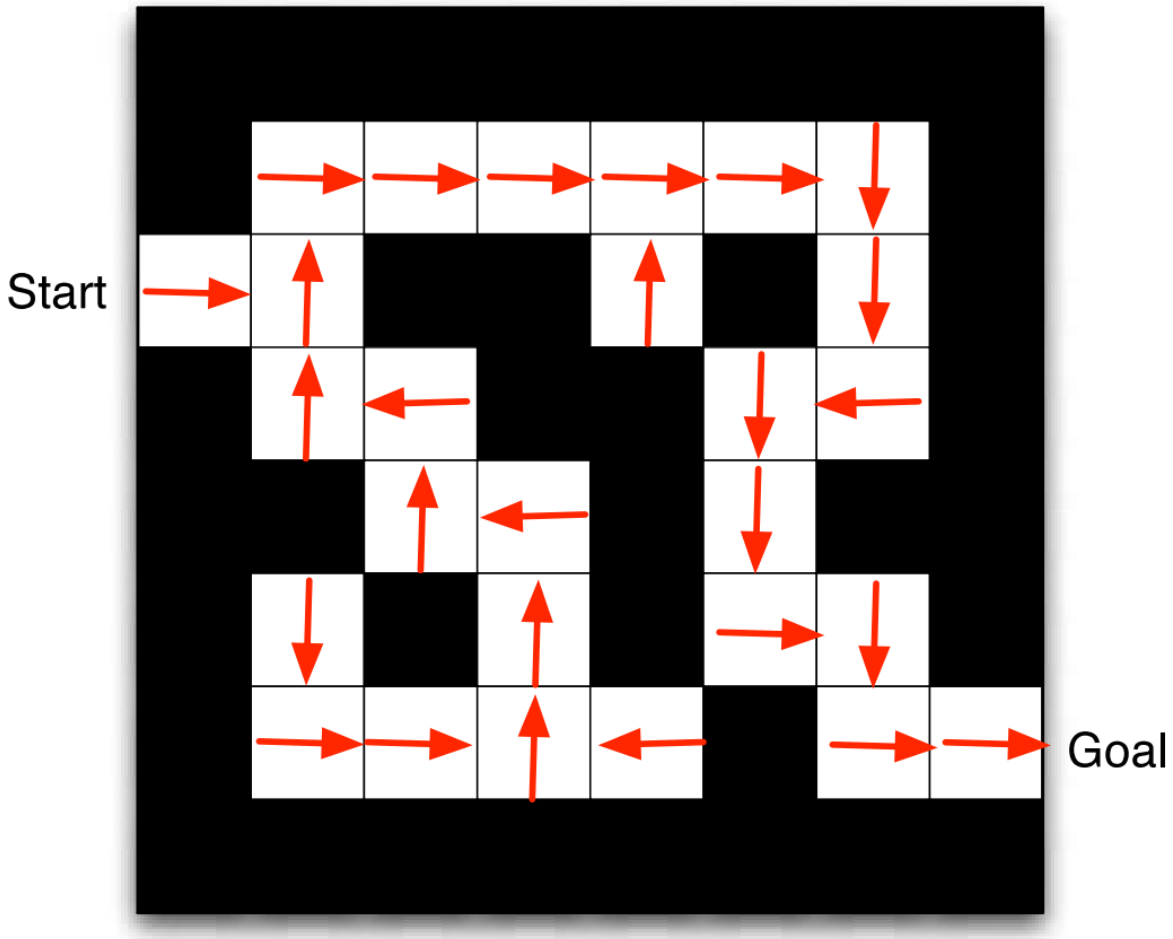
Policies

We formalise the behaviour of an agent as a **policy**

Generally speaking, a policy is a map from states to actions

$$\pi : \mathcal{S} \to \mathcal{A}$$

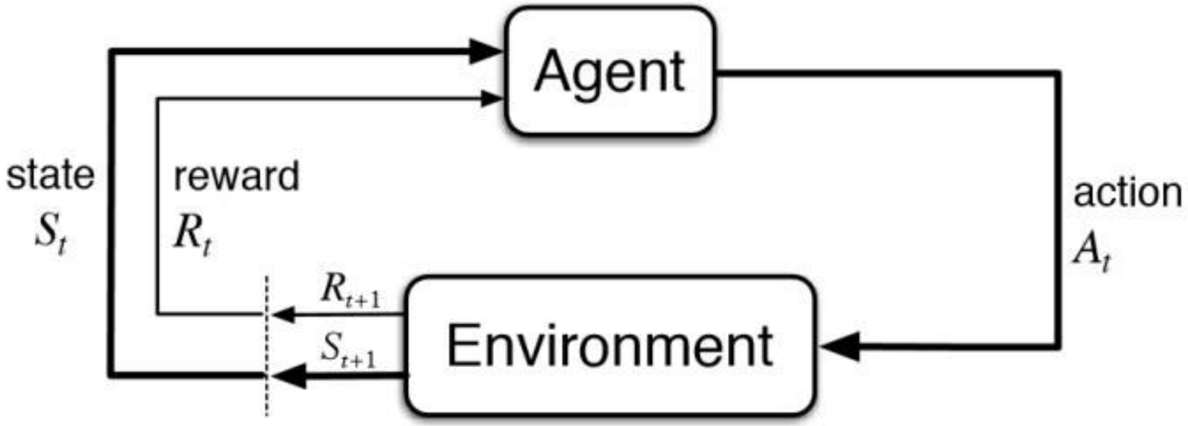
Policies



- **Rewards**: -1 per time-step
- **Actions**: N, S, W, E
- **States**: Agent's location
- **Transitions**: deterministic

Reinforcement Learning Goal

To answer this question, remember that we are at a loop of receiving rewards at each time step.



Reinforcement Learning Goal

At each **time step** we receive a new reward

$$R_{t+1} \quad , \quad R_{t+2} \quad , \quad \cdots$$

Reinforcement Learning Goal

At each **time step** we receive a new reward

Objective: Maximise the **accumulation** of rewards

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

Reinforcement Learning Goal

At each **time step** we receive a new reward

Objective: Maximise the **accumulation** of rewards

$$G_t \doteq \quad R_{t+1} + \lambda R_{t+2} + \lambda^2 R_{t+3} + \cdots$$

$0 < \lambda < 1$ is the discount factor

Reinforcement Learning Goal

Now we have a **criterion**!

But...

Reinforcement Learning Goal

Now we have a **criterion**!

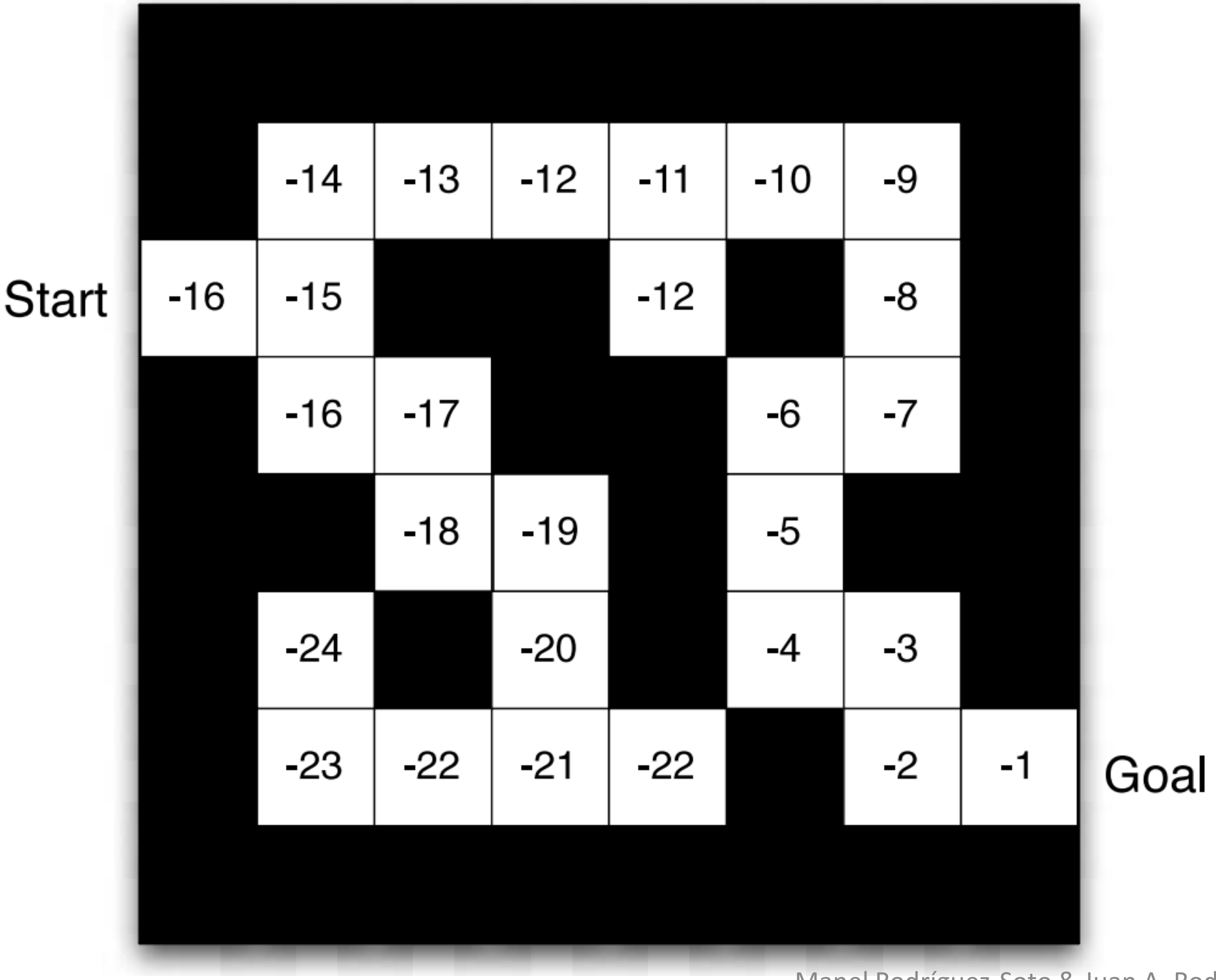
But **how** do we know what is the best policy?

State Value Function

The V-function tells you how good is your **policy** at each **state**

$$V_\pi(s) \doteq \mathbb{E}[G_t \mid \pi, S_t = s]$$

State Value function



Rewards: -1 per time-step
Actions: N, S, W, E
States: Agent’s location
Transitions: deterministic

State Value Function

The V-function tells you how good is your **policy** at each **state**

$$V_\pi(s) \doteq \mathbb{E}[G_t \mid \pi, S_t = s]$$

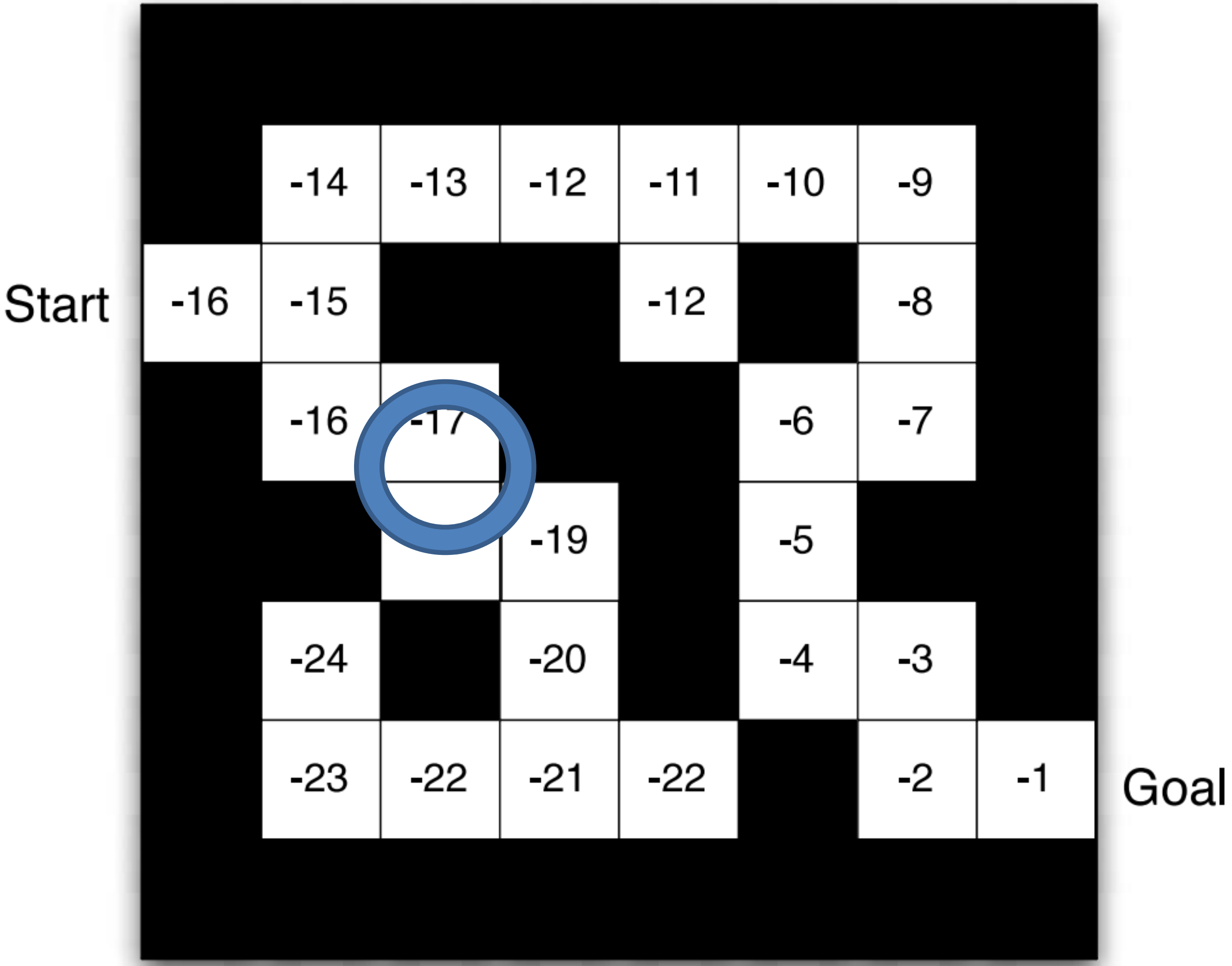
But what if I also want to know how good it is at each state-action pair?

Action-State Value Function

The Q-function tells you how good is your **policy** at each **state-action** pair

$$Q_\pi(s, a) \doteq \mathbb{E}[\, G_t \mid \pi, S_t = s, A_t = a]$$

Action-State Value function



- Q(s, W) = -17
- Q(s, N) = -18
- Q(s, E) = -18
- Q(s, S) = -19

Optimal Value Functions

Since these functions tell you how good is your policy

Optimal Value Functions

Since these functions tell you how good is your policy

$$V_* = \max_\pi V_\pi$$
$$Q_* = \max_\pi Q_\pi$$

Our goal is to **maximise** them (it doesn't matter which one)

Optimal Value Functions

Our goal is to **maximise** them (it doesn’t matter which one)

Optimal Value Functions

Our goal is to **maximise** them (it doesn't matter which one)

In other words:

Our goal is to find the **policy that maximises** them

Optimal Policy

The objective in RL is to find an **optimal policy**, i.e.:
A policy with an **optimal value function,** such that for every state **s**:

$$\pi_* = \arg\max_{\pi} V_{\pi}(s), \pi_* = \arg\max_{\pi} Q_{\pi}(s, a)$$

Optimal Policy

The objective in RL is to find an **optimal policy**, i.e.:
A policy with an **optimal value function,** such that for every state **s**:

$$\pi_* = \arg\max_{\pi} V_\pi(s), \pi_* = \arg\max_{\pi} Q_\pi(s, a)$$

THIS IS THE EQUATION THAT EVERY (*TABULAR*) REINFORCEMENT LEARNING ALGORITHM TRIES TO SOLVE

Bellman equations

Decomposing value function into immediate reward and future value

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots)|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|S_t = s] \end{aligned}$$

Bellman equations

Decomposing value function into immediate reward and future value

$$
\begin{aligned}
Q(s,a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a \sim \pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a]
\end{aligned}
$$

Bellman optimality equations

$$V_*(s) = \max_{a \in \mathcal{A}} Q_*(s, a)$$

Bellman optimality equations

$$V_*(s) = \max_{a \in \mathcal{A}} Q_*(s, a)$$

$$\begin{aligned} Q(s, a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a \sim \pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a] \end{aligned}$$

Bellman optimality equations

$$V_*(s) = \max_{a \in \mathcal{A}} Q_*(s, a)$$

$$\begin{aligned} Q(s, a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a \sim \pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a] \end{aligned}$$

$$Q_*(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} Q_*(s', a')$$

Bellman optimality equations

$$V_*(s) = \max_{a \in \mathcal{A}} Q_*(s, a)$$

$$\begin{aligned} Q(s, a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a \sim \pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a] \end{aligned}$$

$$Q_*(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^{a} \max_{a' \in \mathcal{A}} Q_*(s', a')$$

If complete info available (R and P), dynamic programming

If not, can't apply these, but guides solution!!!

DYNAMIC PROGRAMMING:

Value Iteration

Value Iteration

If Model fully known. Apply Bellman equations iteratively

Value Iteration

If Model fully known. Apply Bellman equations iteratively

Sweep through each state and action,

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_{t-1}(s')$$

Value Iteration

If Model fully known. Apply Bellman equations iteratively

Sweep through each state and action,

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_{t-1}(s')$$

Update V_t with the updated Q. **Repeat until convergence of Q.**

Value Iteration

Pros:

- No hyperparameters: very easy to use.
- It is exact: wait enough iterations and you obtain the optimal policy.

Value Iteration

Pros:

 - No hyperparameters: very easy to use.

 - It is exact: wait enough iterations and you obtain the optimal policy.

Cons:

 - Requires knowing the complete model of the MDP.

 - Very slow.

MODEL-FREE REINFORCEMENT LEARNING

Q-learning & SARSA

Q-learning

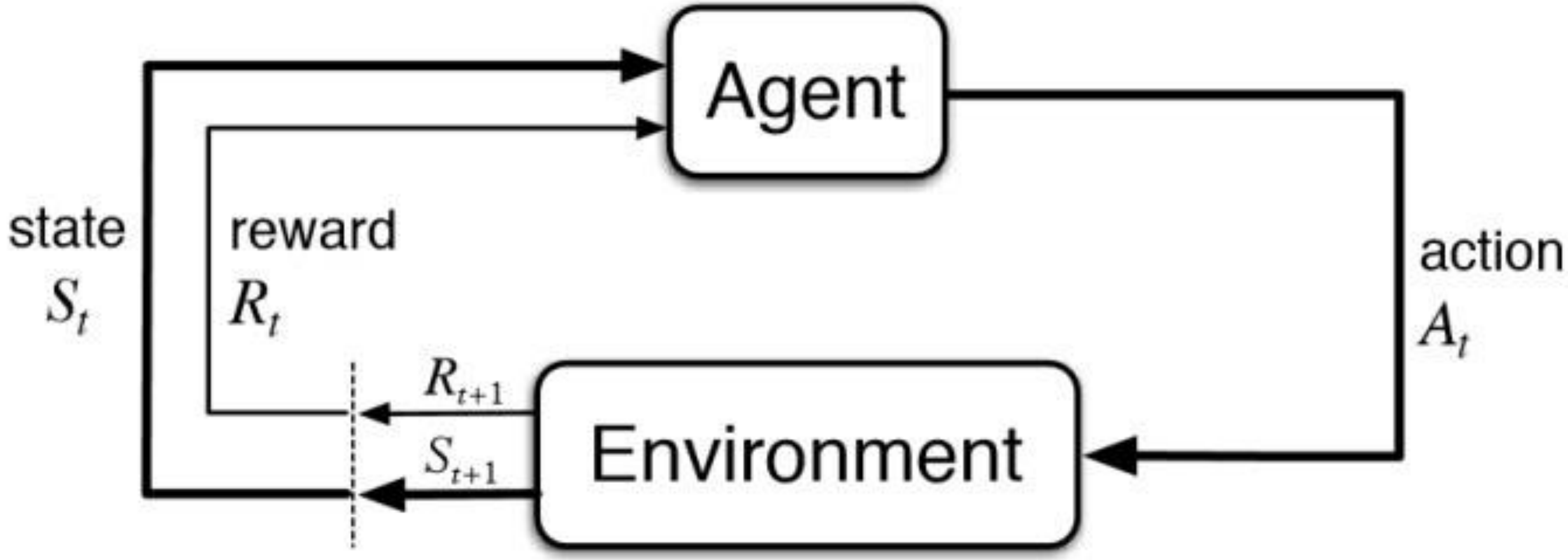
What to do if no model available?

Q-learning

What to do if no model available?

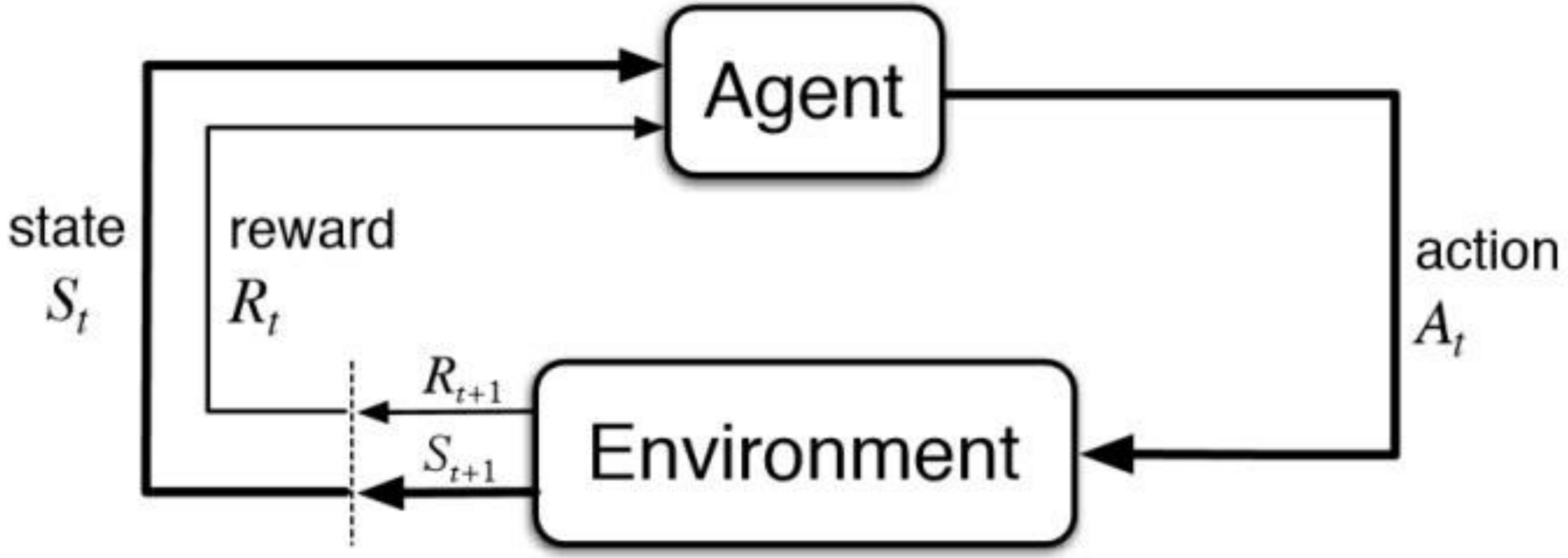
Trial-and-error!!

Q-learning



Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a\in\mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)).$$



Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)).$$

The policy used to learn can be chosen randomly. We only care that each state is visited enough times.

Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)).$$

We update the **Q-table** of our policy using **max Q,** which is the Q-table of the greedy policy.

However, we do not use the greedy policy for exploring!

Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)).$$

However, we do not use the greedy policy for exploring!

For that reason, Q-learning is an **OFF-POLICY** algorithm.

SARSA. On-policy TD learning

$$S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)).$$

If we use the same policy for exploring and updating:

The algorithm now is called SARSA. SARSA is an **ON-POLICY** algorithm.

Model-free reinforcement learning

Pros:

- Very easy to code as no model is needed.
- Much faster than VI.

Model-free reinforcement learning

Pros:

- Very easy to code as no model is needed.
- Much faster than VI.

Cons:

- Two hyperparameters now (learning rate and learning policy).
- Not all hyperparameters are valid!

Q-learning vs SARSA

Pros of Q-learning:

- The hyperparameter policy is not so important in Q-learning: one problem less!

Pros of SARSA:

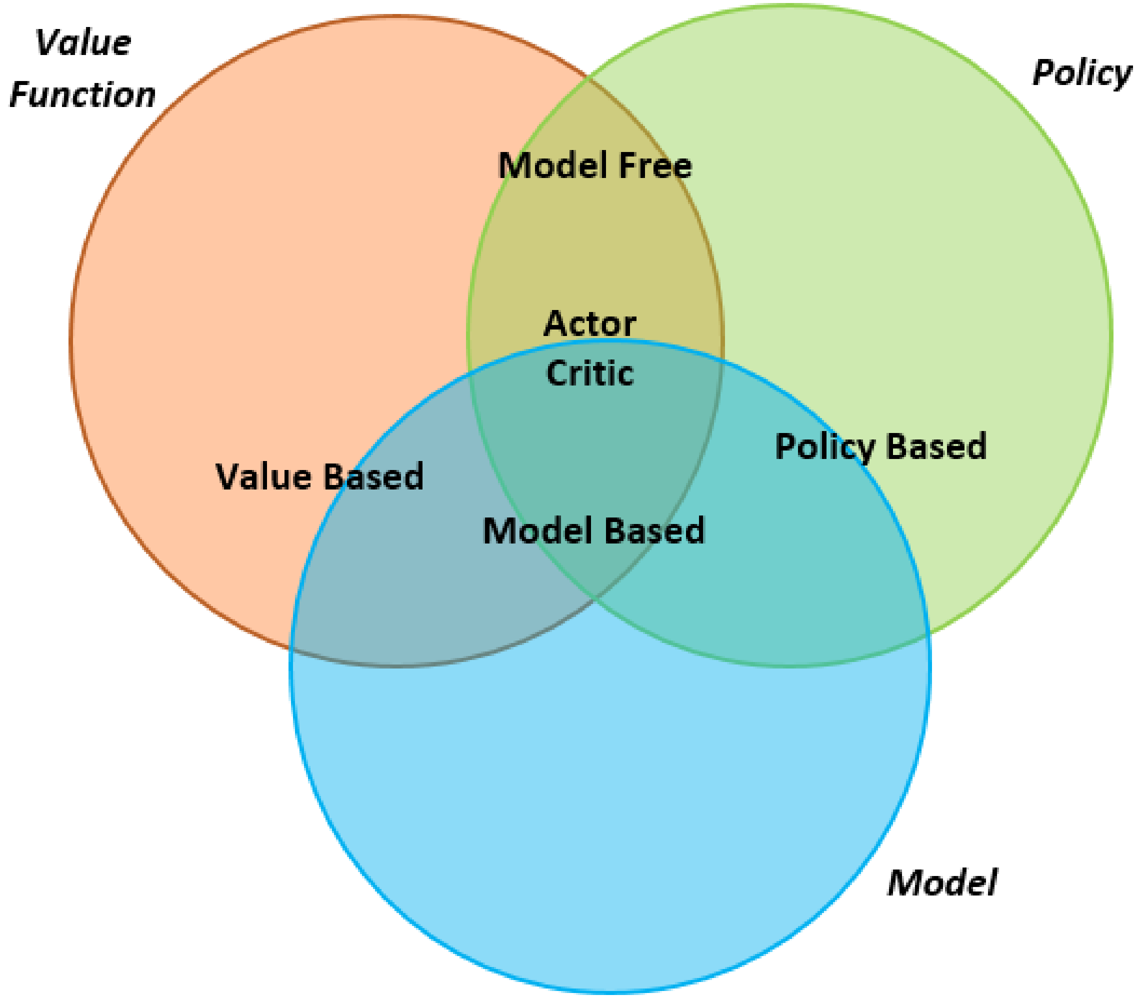
- But because of this, Q-learning typically converges slower!

RL elements: Summary

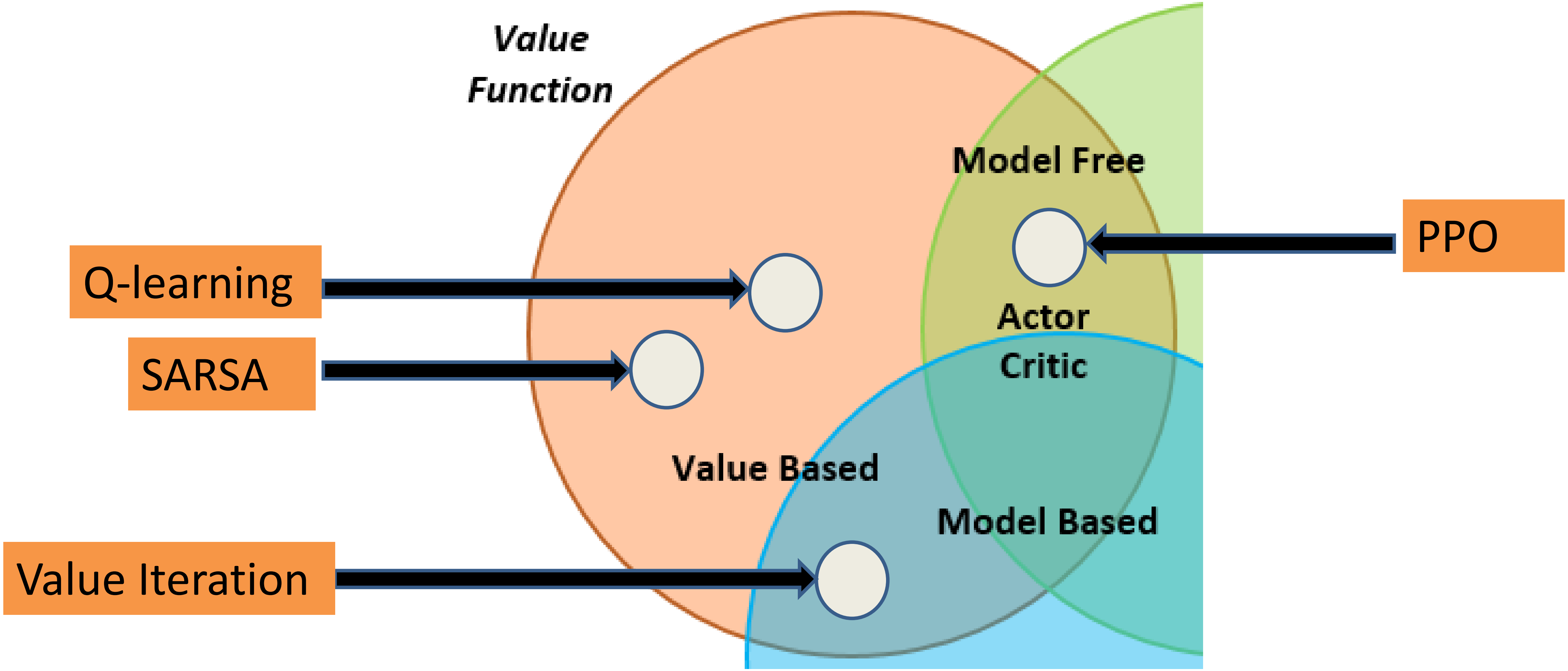
When complete info not available

- **Model-based.** Model (of environment) available and used.
- **Model-free.** No model (of environment) not used/not available.
- **On-policy.** The policy for exploring is the same as the policy for updating estimates.
- **Off-policy**. The policy for exploring is different to the policy for updating estimates.

RL algorithms: taxonomy



RL algorithms: taxonomy



RL criticisms

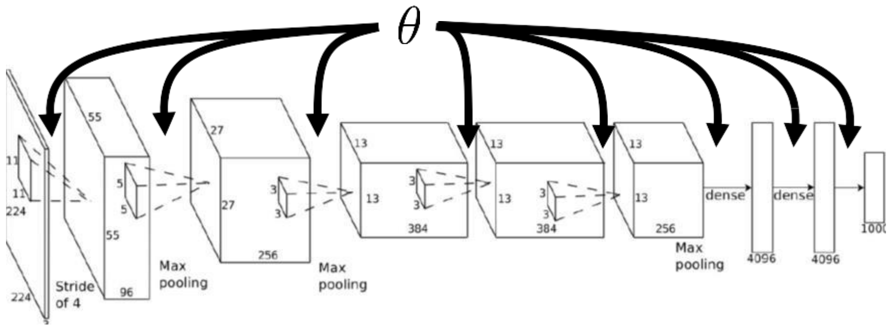
- High sensitivity to hyper-parameter tuning
- High sensitivity to initialisation
- Sparsity of rewards
- Difficult stability during training because, unlike supervised learning, the data distribution of observations and rewards change as the agent learns

Section 2: Going deep

Manel Rodríguez-Soto
Juan A. Rodríguez-Aguilar
ICMAT@Madrid
28.04.2023

Five cents about deep learning

- Before diving into DRL we need to recall fundamental concepts in deep learning
  - **<u>Deep network</u> with hyperparameters $\boldsymbol{\theta}$**

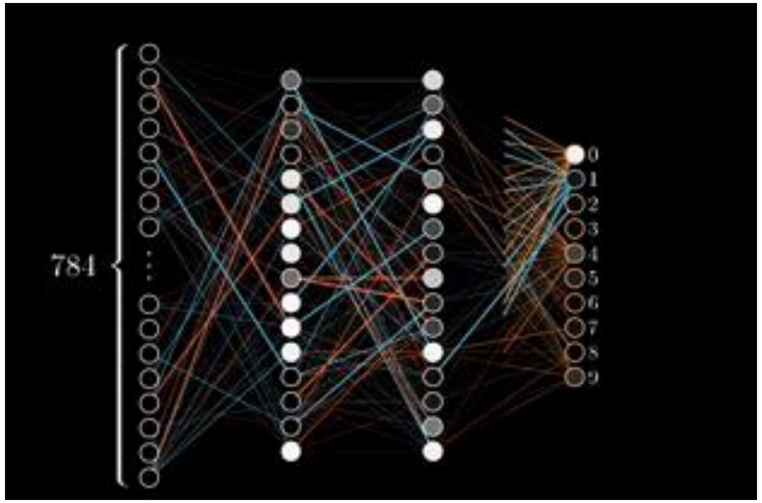


Five cents about deep learning

- Before diving into DRL we need to recall fundamental concepts in deep learning
  - **Deep network with hyperparameters $\boldsymbol{\theta}$**
  - **Prediction**

Five cents about deep learning

- Before diving into DRL we need to recall fundamental concepts in deep learning
  - **<u>Deep network</u> with hyperparameters $\boldsymbol{\theta}$**
  - **<u>Prediction</u>**
  - **<u>Learning</u> with a <u>loss function</u> changes values of $\boldsymbol{\theta}$**

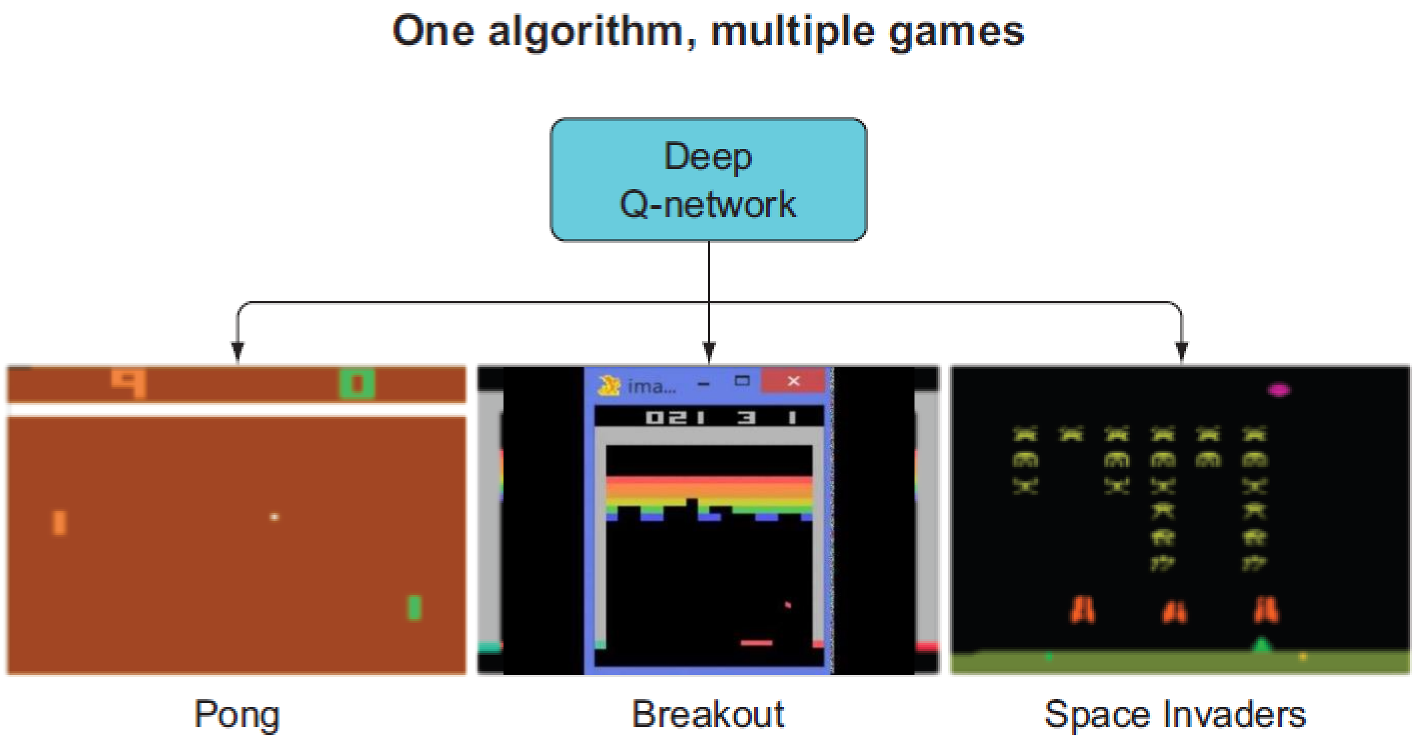


Deep Reinforcement Learning (DRL)

- DRL is a **subfield of machine learning** that utilizes deep learning models (i.e., neural networks) in RL tasks
- Why DRL?
  - **Classic RL cannot cope with large state spaces** (<u>curse of dimensionality</u>: when we have a state space with many dimensions, the number of entries that you need in a table for tabular RL is exponential on the number of dimensions)
- Why is DRL successful?
  - Because of the general boost by **deep learning** that allowed the training of much larger networks
  - Because of the specific **novel features** implemented to address some of the issues that RL algorithms struggled with
- Breakthrough in RL comes with the application of DRL to **large RL problems** (e.g. games, robotics, autonomous vehicles)

Examples

- In 2013 DeepMind’s DQN algorithm successfully learned to play seven Atari games **at superhuman levels** with only the raw pixels as input and the player’s score as the objective to maximize.

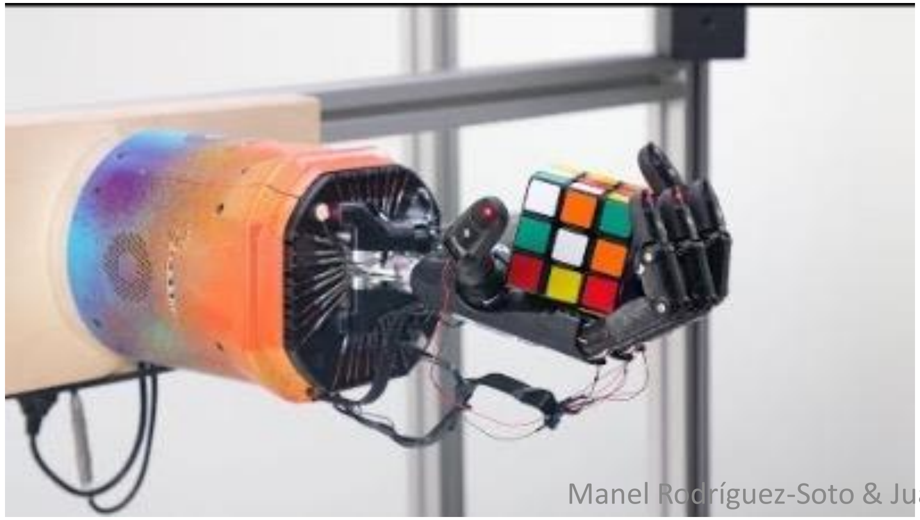


Examples

- Sony's Gran Turismo Sophy, a superhuman racing agent : https://www.gran-turismo.com/us/gran-turismo-sophy/technology/
- RL is particularly well suited to developing game AI agents because RL agents consider the long-term repercussions of their actions and can independently collect their own data during learning, avoiding the need for complex, hand-coded behaviour rules

Examples

- Robotics is a very important application domain of DRL.
- OpenAI trained robot hands with the PPO algorithm to learn to solve the Rubik's cube and to re-orientate physical objects.
- Training 100% in simulation using *Automatic Domain Randomization*.
- The learned policies were later deployed in the real world.



But…

- Major DRL challenge
  - One added complexity of tackling RL with deep learning is the additional dimension of time: <u>training is dynamic</u>

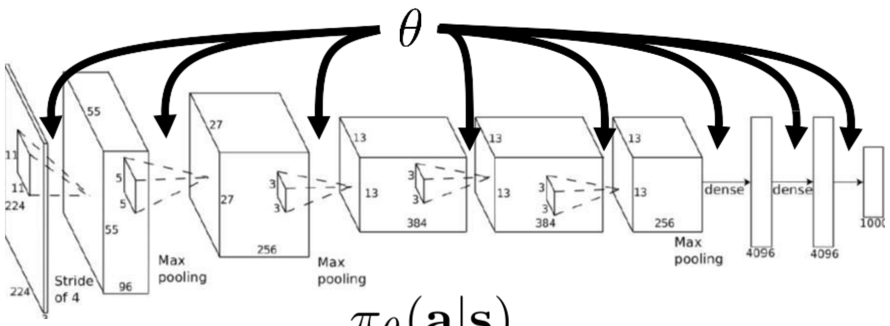
But…

- Major DRL challenge
  - One added complexity of tackling RL with deep learning is the additional dimension of time: training is dynamic
- Bad news:
  - We lose convergence guarantees
  - DRL strongly relies on approximations
  - Using deep architectures for RL is an intricate engineering exercise (as it is for deep learning in general!)
  - Difficult to achieve stability in the learning process

Our agenda

- **Deep Q learning (DQN)**
- Proximal policy optimization (PPO): a state-of-the-art algorithm
- Multi-agent reinforcement learning

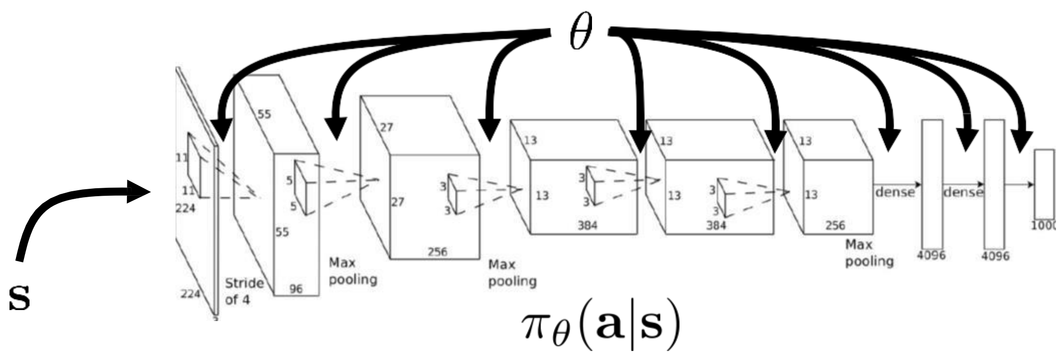
The goal of reinforcement learning



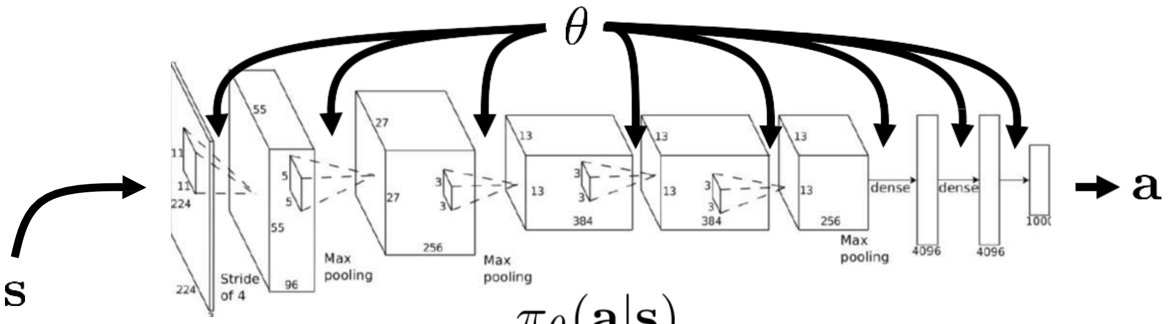
$\pi_\theta(\mathbf{a}|\mathbf{s})$

- We have a deep network parametrized by θ
- $\pi_\theta(a|s)$ is the probability of selecting action a at state s
- The policy will depend on the hyperparameters θ of the deep network

The goal of reinforcement learning

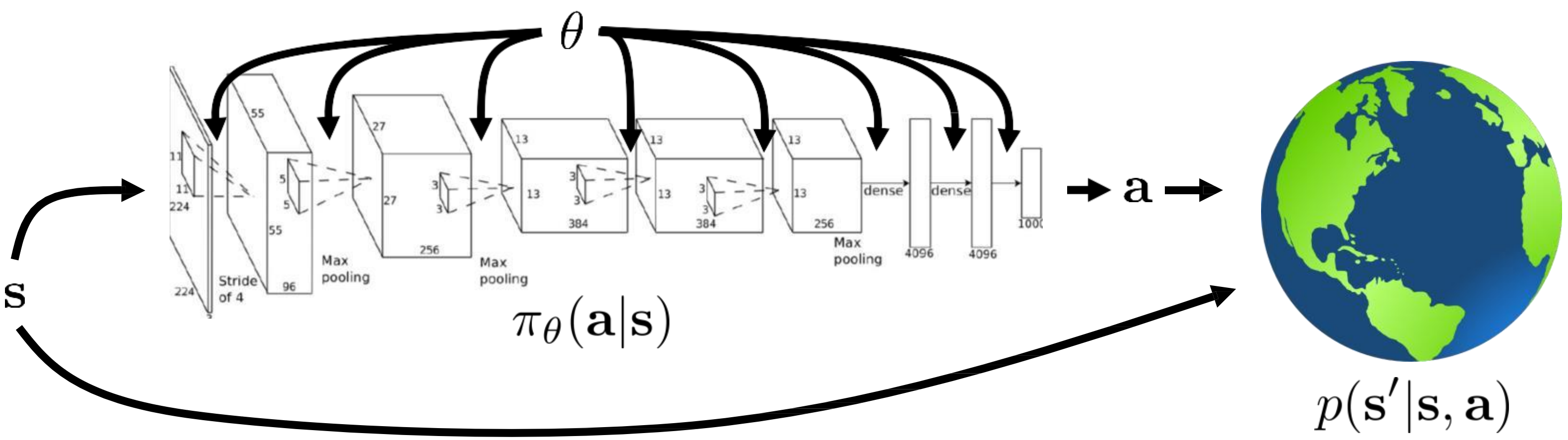


The goal of reinforcement learning

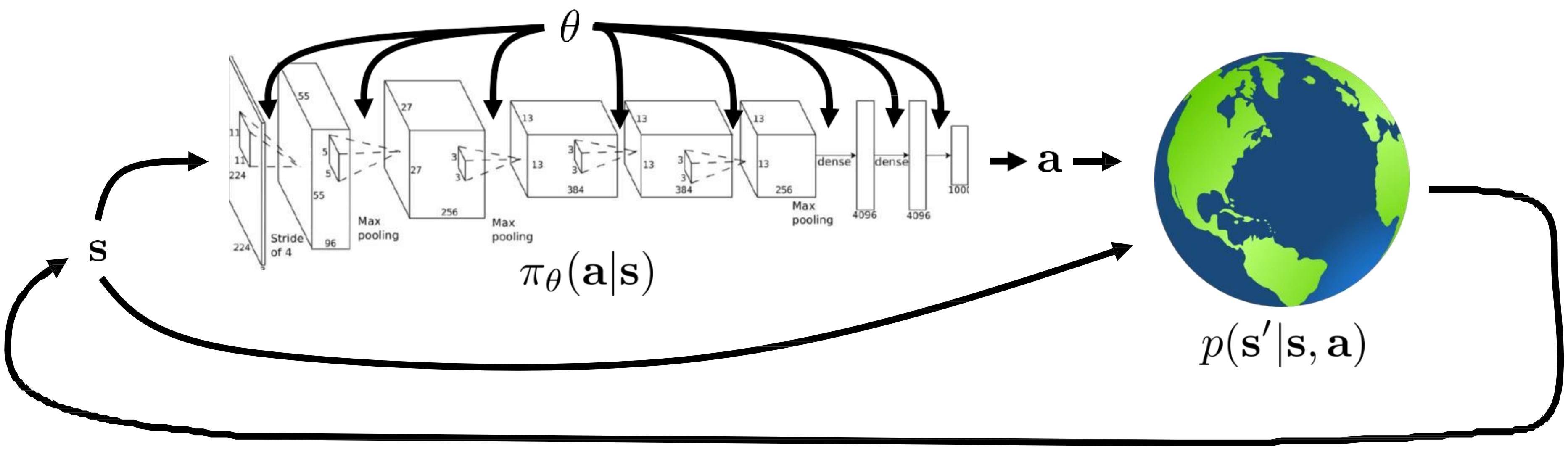


$\pi_\theta(\mathbf{a}|\mathbf{s})$

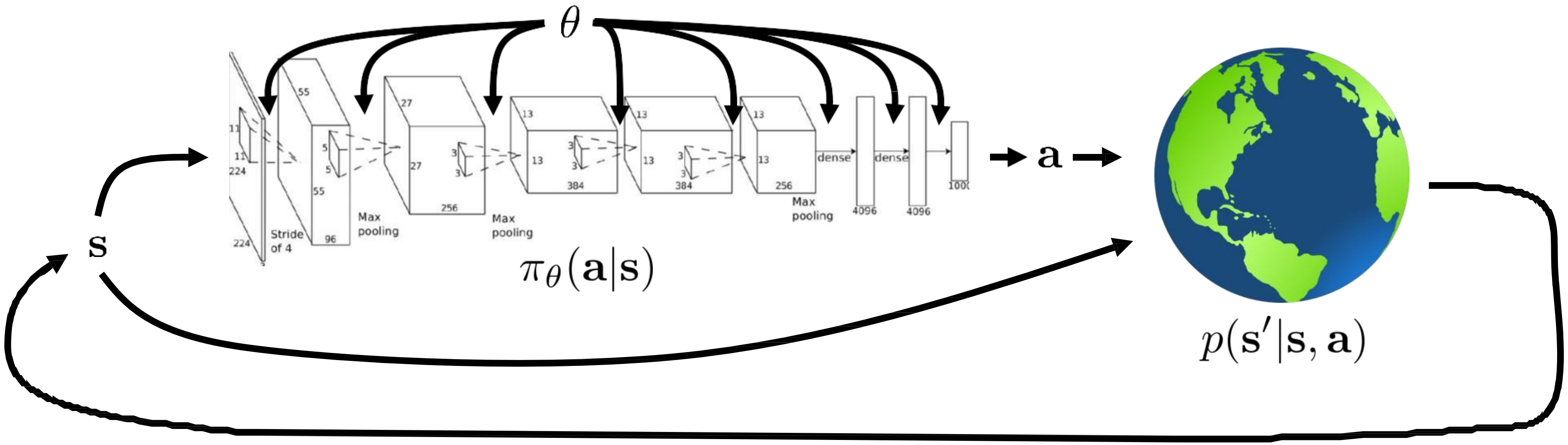
The goal of reinforcement learning



The goal of reinforcement learning



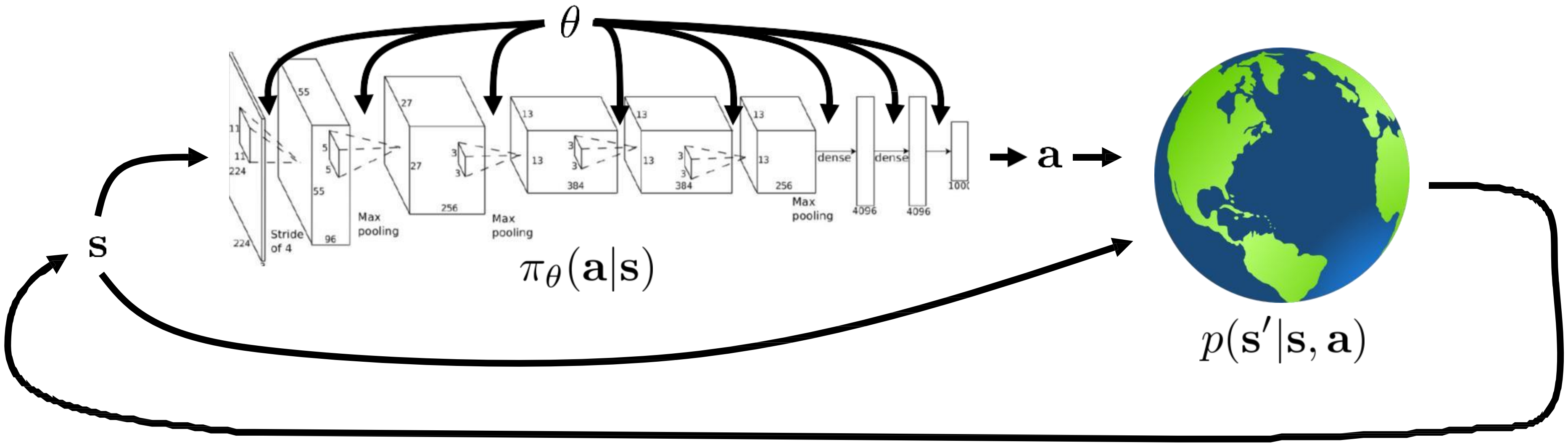
The goal of reinforcement learning



$$\underbrace{p_\theta(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)}_{p_\theta(\boxed{\tau})} = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

TRAJECTORY

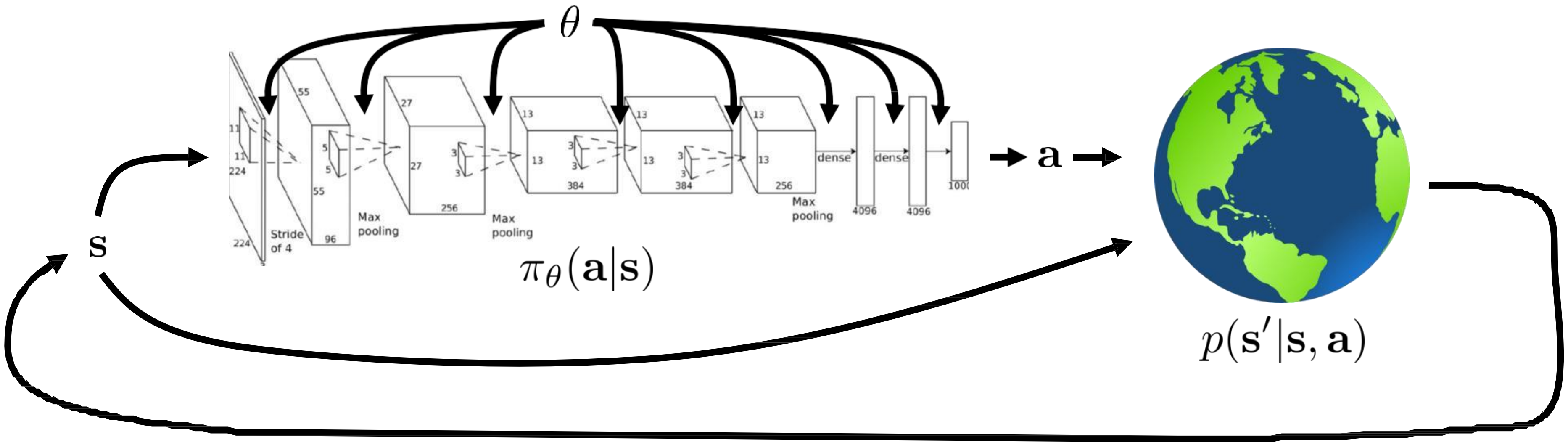
The goal of reinforcement learning



We face an optimization problem: to find the hyperparameters θ that on expectation maximise the rewards over trajectories (τ)

$$\theta^\star = \arg\max_\theta E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t,\mathbf{a}_t)\right]$$

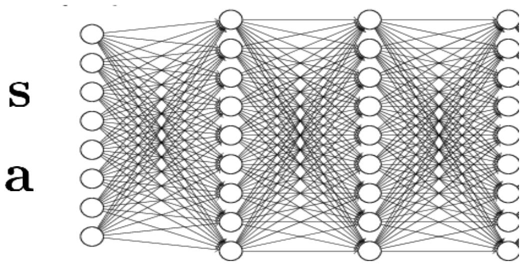
The goal of reinforcement learning



We face an optimization problem: to find the hyperparameters θ that on expectation maximise the rewards over trajectories (τ)

$E_{\mathbf{s}_1 \sim p(\mathbf{s}_1)}[V^\pi(\mathbf{s}_1)]$ is the RL objective!

Q learning on a deep network (fitted Q-iteration)



$Q_\phi(\mathbf{s}, \mathbf{a})$
parameters ϕ

In a purely value-base method we do not represent the policy in a neural net, we represent it **implicitly**: we would need to do **epsilon greedy** or **softmax** to select the action from Q values.

Q learning on a deep network (fitted Q-iteration)

full fitted Q-iteration algorithm:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy

$K\times$

2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

parameters

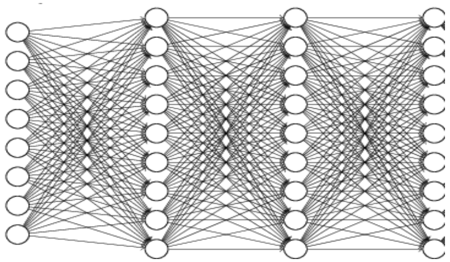
dataset size N, collection policy

iterations K

gradient steps S

$\mathbf{s}$

$\mathbf{a}$



$Q_\phi(\mathbf{s}, \mathbf{a})$

parameters ϕ

Q learning on a deep network (fitted Q-iteration)

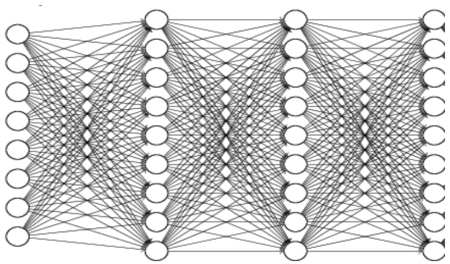
full fitted Q-iteration algorithm:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using $\boxed{\text{some policy}}$
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

$K\times$

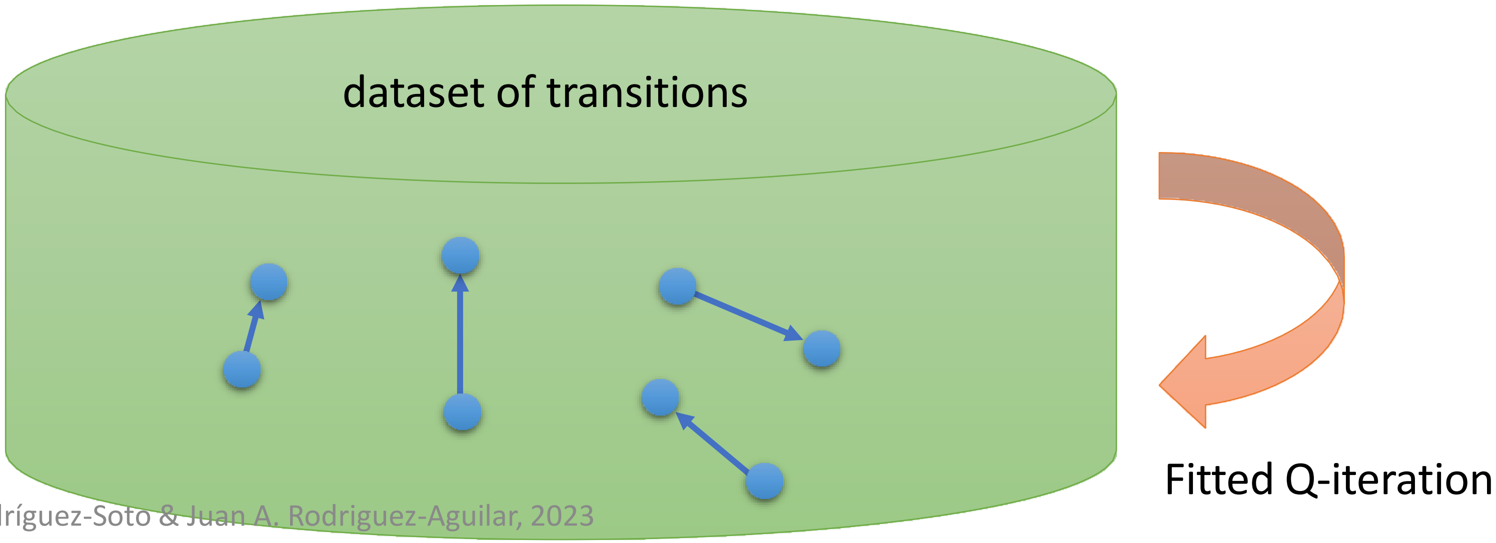
$\mathbf{s}$

$\mathbf{a}$



$Q_\phi(\mathbf{s}, \mathbf{a})$

parameters ϕ



Q learning on a deep network (fitted Q-iteration)

full fitted Q-iteration algorithm:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|\underbrace{Q_\phi(\mathbf{s}_i, \mathbf{a}_i)}_{\text{Current value predicted by the network}} - \underbrace{\mathbf{y}_i}_{\text{Target value to learn}}\|^2$

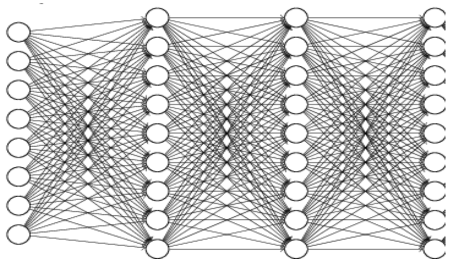
$K \times$

Current value predicted by the network

Target value to learn

$\mathbf{s}$

$\mathbf{a}$



$Q_\phi(\mathbf{s}, \mathbf{a})$

parameters ϕ

Q learning on a deep network (fitted Q-iteration)

full fitted Q-iteration algorithm:

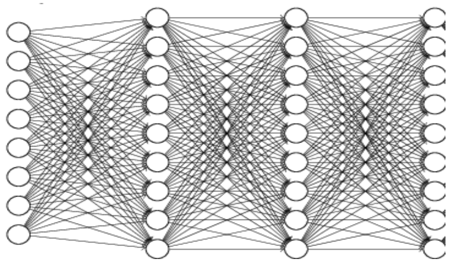
1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \underline{\|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|}^2$

$K \times$

TEMPORAL DIFFERENCE ERROR
difference between consecutive temporal predictions

$\mathbf{s}$

$\mathbf{a}$



$Q_\phi(\mathbf{s}, \mathbf{a})$
parameters ϕ

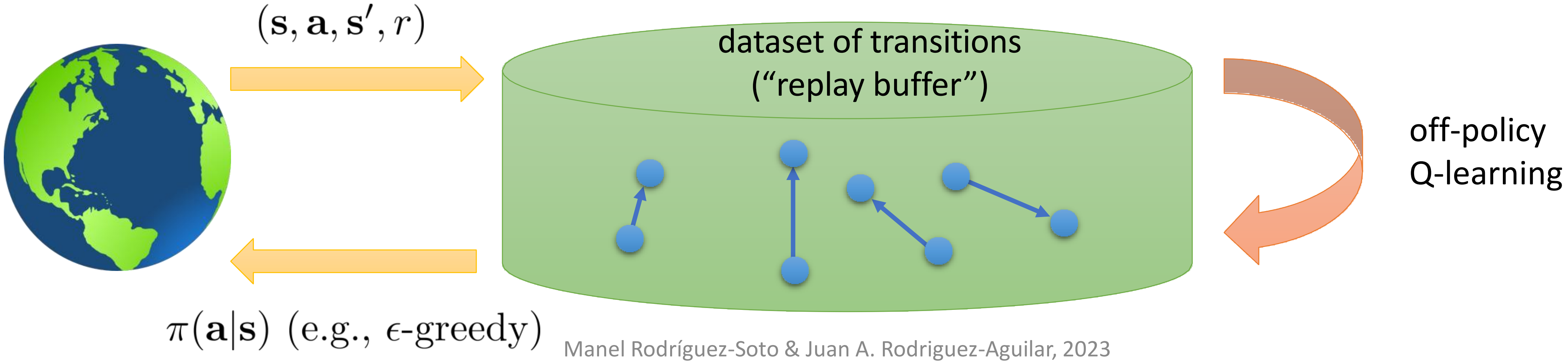
Data-driven Q-learning: use replay buffer

but where does the data come from?

a replay buffer help us cope with catastrophic forgetting by avoiding the use of correlated samples

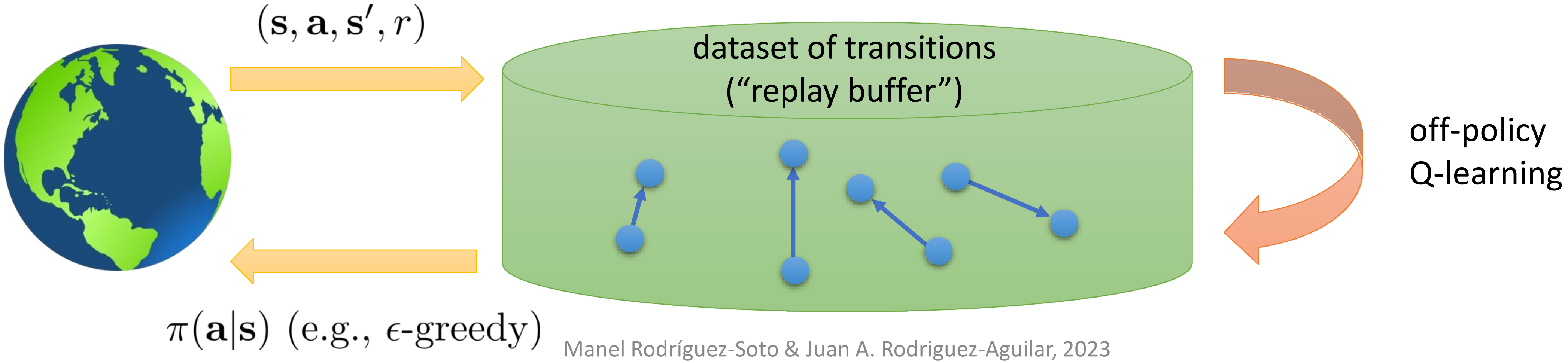
need to periodically feed the replay buffer by using our latest policy to collect data that does better coverage of transitions

we don't care so much about where these transitions came from…as long as they cover the space of transitions pretty well



Classic deep Q learning with replay buffer

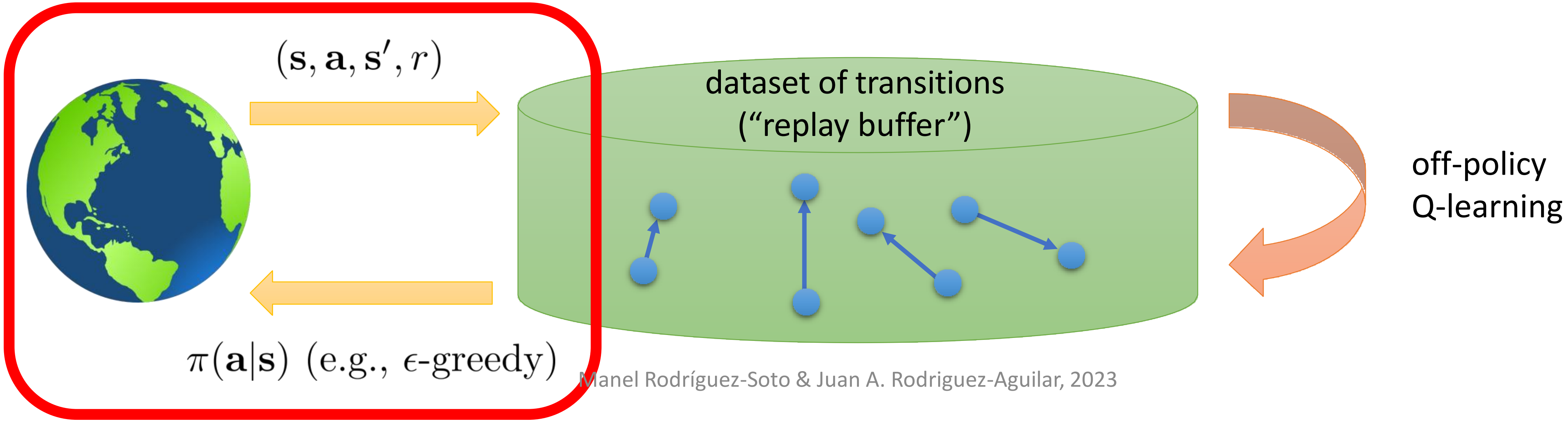
full Q-learning with replay buffer:



Classic deep Q learning with replay buffer

full Q-learning with replay buffer:

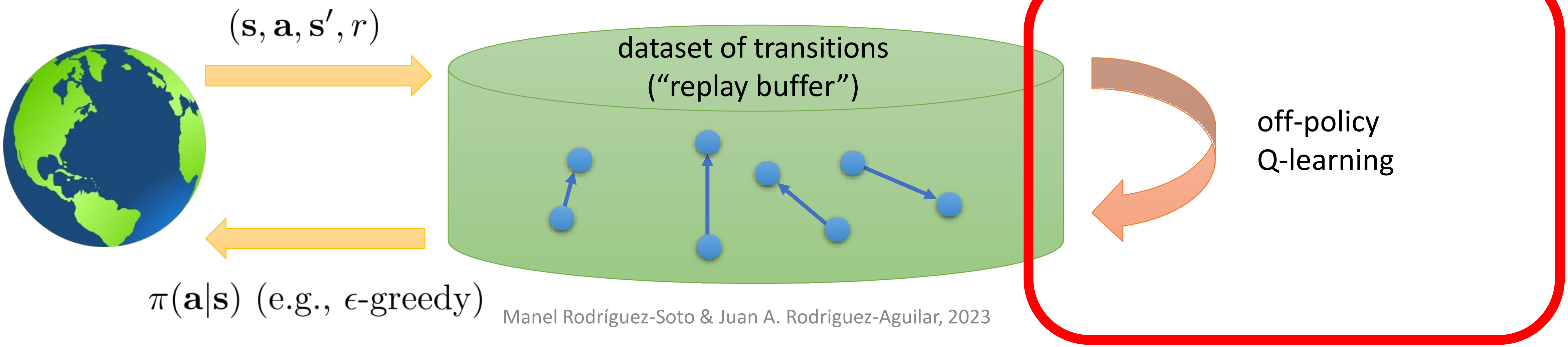
1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$



Classic deep Q learning with replay buffer

full Q-learning with replay buffer:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
   2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
   3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(\underbrace{Q_\phi(\mathbf{s}_i, \mathbf{a}_i)}_{\text{Current value predicted by the network}} - \underbrace{[r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)]}_{\text{Target value to learn}})$



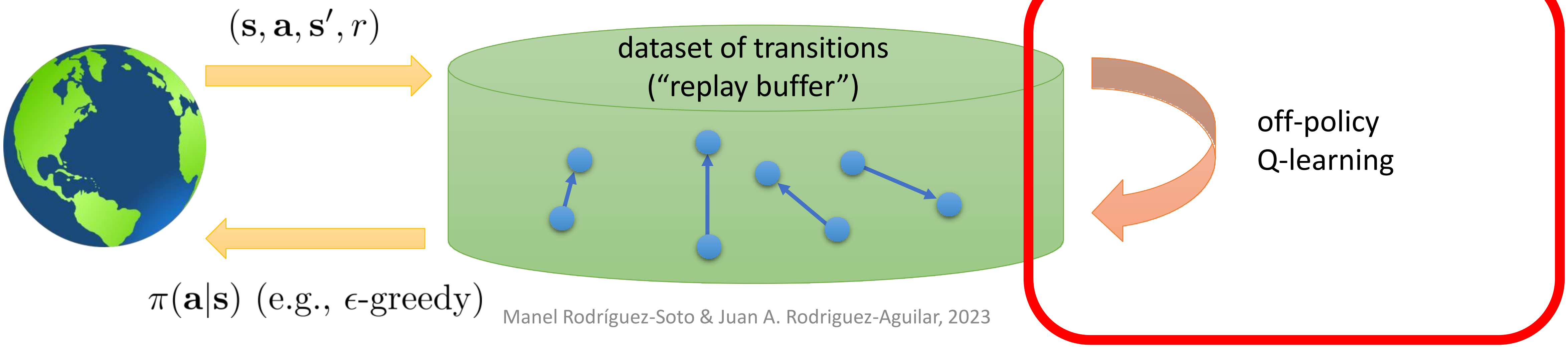
Classic deep Q learning with replay buffer

full Q-learning with replay buffer:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$



The moving target problem

full Q-learning with replay buffer:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \underline{[r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)]})$

This learning target continuously moves
because we use the Q network to predict Q values
and the learning changes the weights of the network,
and hence the Q values and target values

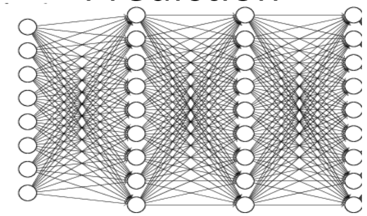
“Classic” deep Q-learning algorithm (DQN)

DQN uses two Q networks for stability:

- prediction network: to predict Q values
- target network: to provide **stable** target Q values for supervised learning

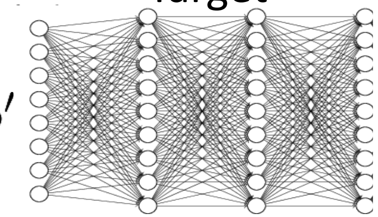
Prediction

ϕ



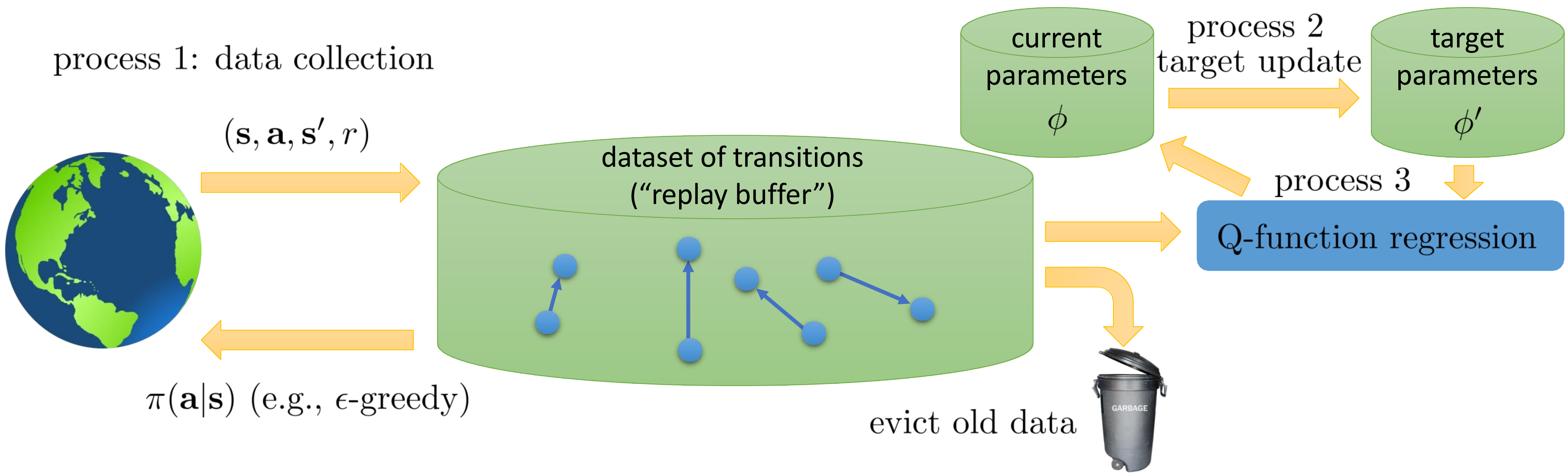
Target

ϕ'



Mnih et al. ’13. Human-level control through deep reinforcement learning: Q- learning with convolutional networks for playing Atari.

A general view of Q learning



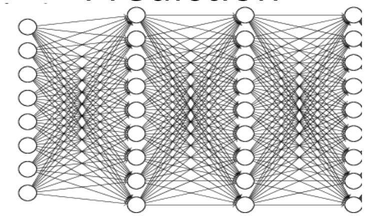
“Classic” deep Q-learning algorithm (DQN)

DQN uses two Q networks for stability:

- prediction network: to predict Q values
- target network: to provide target Q values for supervised learning

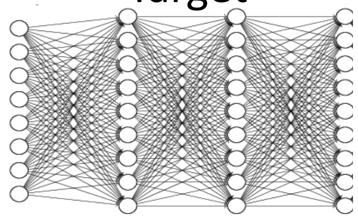
Prediction

ϕ



Target

ϕ'



“classic” deep Q-learning algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

supervised regression of Q function

targets don’t change!

Mnih et al. ‘13

Review

- Q-learning in practice
  - Replay buffers
  - Target networks
- Multiple variations and extensions of DQN, e.g.:
  - Double DQN (DDQN) (Hado van Hasselt et al. 2015)
  - DDQN with Prioritised Experience Replay (Schaul et al. 2016)
  - Dueling DDQN (Wang et al. 2016)
- Deep Q-learning with continuous actions
  - Random sampling
  - Analytic optimization
  - Second “actor” network

Q-learning suggested readings

- Classic papers
  - Watkins. (1989). Learning from delayed rewards: introduces Q-learning
  - Riedmiller. (2005). Neural fitted Q-iteration: batch-mode Q-learning with neural networks
- Deep reinforcement learning Q-learning papers
  - Lange, Riedmiller. (2010). Deep auto-encoder neural networks in reinforcement learning: early image-based Q-learning method using autoencoders to construct embeddings
  - Mnih et al. (2013). Human-level control through deep reinforcement learning: Q-learning with convolutional networks for playing Atari.
  - Van Hasselt, Guez, Silver. (2015). Deep reinforcement learning with double Q-learning: a very effective trick to improve performance of deep Q-learning.
  - Lillicrap et al. (2016). Continuous control with deep reinforcement learning: continuous Q-learning with actor network for approximate maximization.
  - Gu, Lillicrap, Stuskever, L. (2016). Continuous deep Q-learning with model-based acceleration: continuous Q-learning with action-quadratic value functions.
  - Wang, Schaul, Hessel, van Hasselt, Lanctot, de Freitas (2016). Dueling network architectures for deep reinforcement learning: separates value and advantage estimation in Q-function.

Pitfalls of DQN

- DQN are very data efficient but rather unstable
- Q learning fails in many simple problems
- DQN works well on game environments like the Arcade Learning Environment [Bel+15] with discrete action spaces
- It <u>has not been demonstrated to perform well on continuous control benchmarks</u> such as those in OpenAI Gym [Bro+16] and described by Duan et al. [Dua+16]

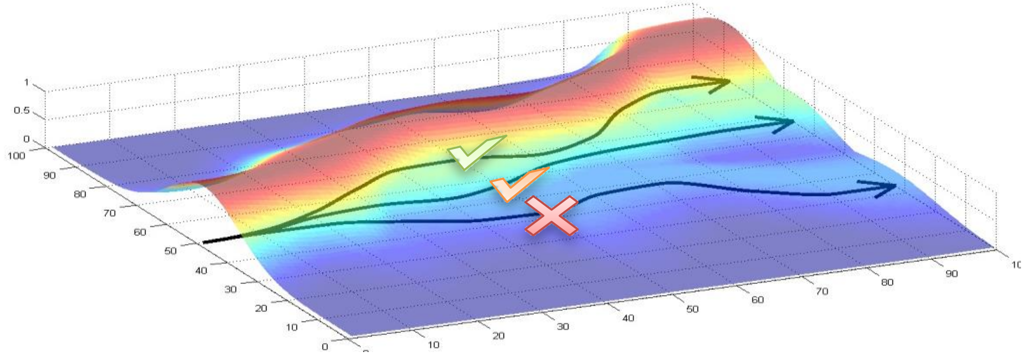
[Bel+15] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. "The arcade learning environment: An evaluation platform for general agents". In: Twenty-Fourth International Joint Conference on Artificial Intelligence. 2015.
[Bro+16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. "OpenAI Gym". In: arXiv preprint arXiv:1606.01540 (2016).
[Dua+16] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. "Benchmarking Deep Reinforcement Learning for Continuous Control". In: arXiv preprint arXiv:1604.06778 (2016).

Our agenda

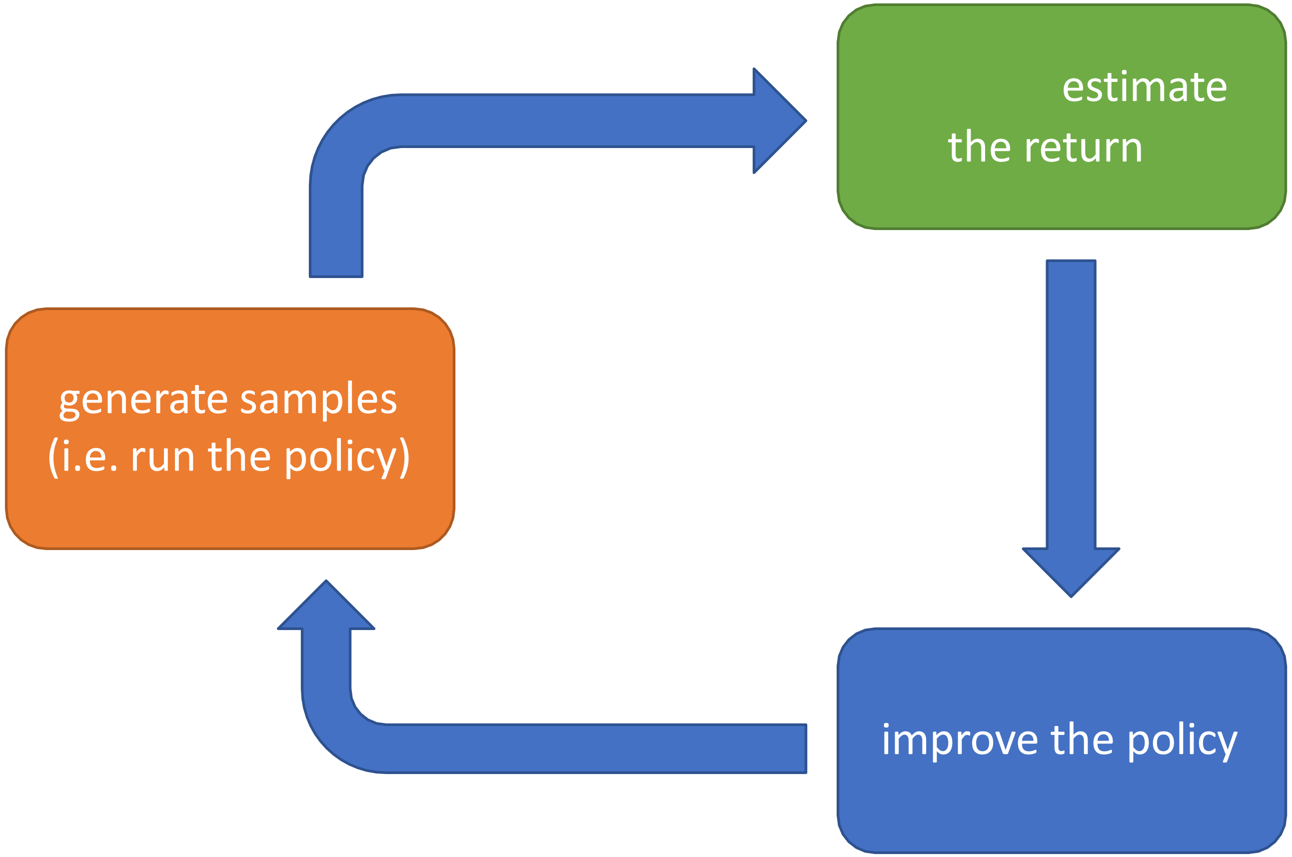
- Deep Q learning (DQN)
- **Proximal policy optimization (PPO): a state-of-the-art algorithm**
- Multi-agent reinforcement learning

Policy gradient algorithms: intuition

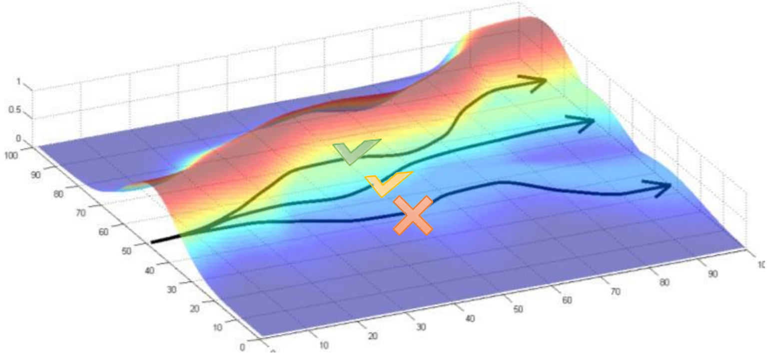
- In short:
  - To boost policies that obtain above average returns
  - To bring down policies that obtain below average returns



Policy gradient algorithms



evaluate returns $R_\tau = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$



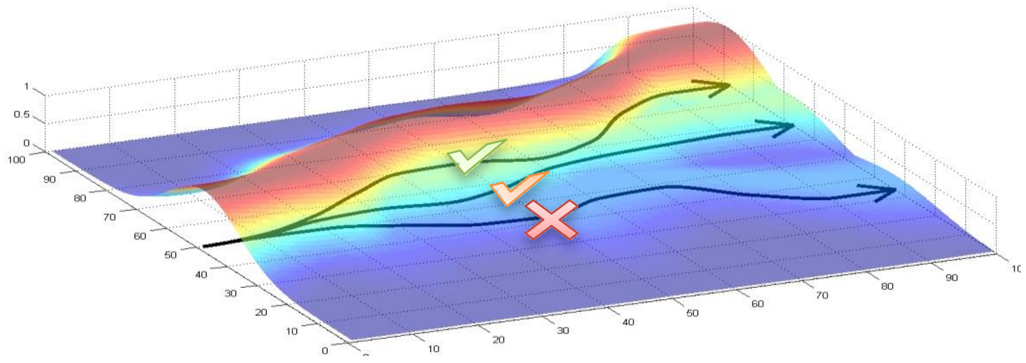
$$\theta \leftarrow \theta + \alpha \boxed{\nabla_\theta E[\textstyle\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]}$$

POLICY
GRADIENT

- Policy gradient methods work by computing **an estimator of the policy gradient** and plugging it into a stochastic gradient **ascent** algorithm
- Policy gradient algorithms alternate between sampling and optimisation

Policy gradient: intuition

- Gradient tries to:
  - **Increase probability of trajectories** with above average (**positive**) returns
  - **Decrease probability of trajectories** with below average (**negative**) returns



! It changes probabilities of experienced trajectories, does not try to change the trajectories

Computing the policy gradient

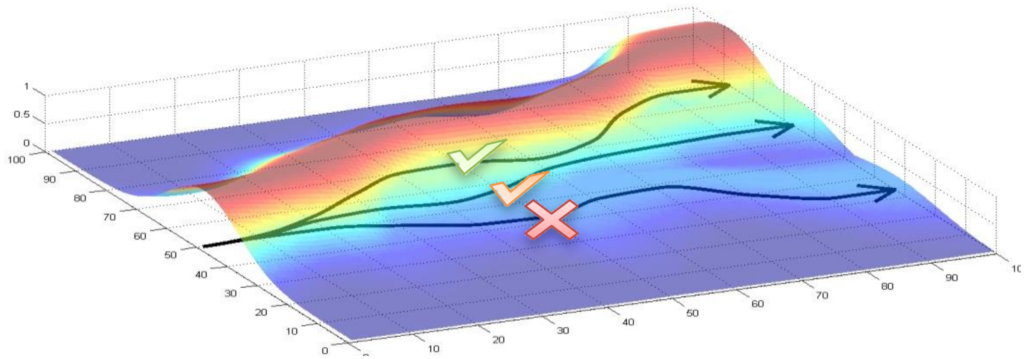
- Although the policy needs to be differentiable and gradients can be computed using calculus, manually computing partial derivatives is rather cumbersome.
- When the policy is a deep neural network (with θ representing network weights), we typically rely on **automated gradient calculations => Our deep learning library (e.g. Pytorch, Keras, Tensorflow) computes the gradient!**
- With automated gradients, we simply define a **loss function** and let the deep learning library solve all the derivations.
- The loss function effectively represents the update signal. We add a minus sign (as training relies on gradient *descent* rather than *ascent*).

Proximal Policy Optimization (PPO)

- State of the art policy gradient (and DRL) algorithm.
- Can be used for a wide range of RL tasks (not only robotics, tested well for games as well).
- On-policy (vs off-policy DQN), meaning that experience is discarded after learning update (updating parameters in the network).
- If data efficiency is not an issue (because data is e.g generated in a simulator), on-policy methods can be more effective in terms of training time.
- Very competitive algorithm for continuous control tasks.

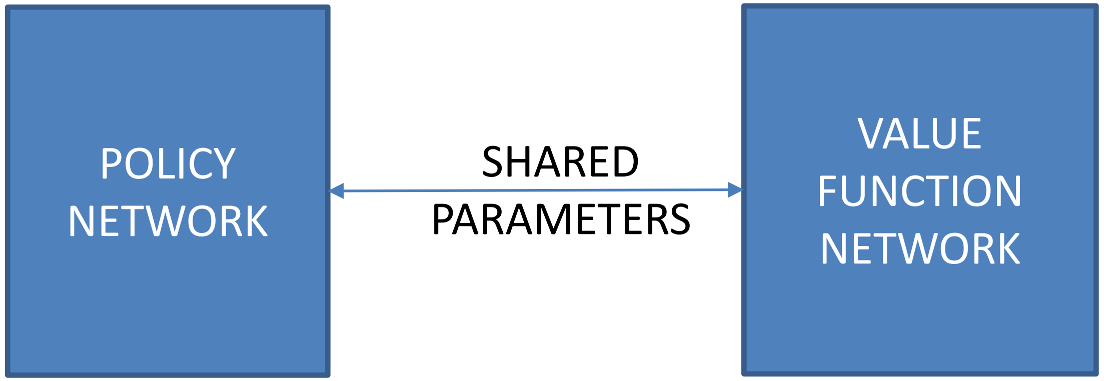
PPO: intuition

- In short:
  - To boost policies that obtain above average returns
  - To bring down policies that obtain below average returns
  - **Don’t move too far away from the old policy when updating the policy (updating the policy network)**



The PPO deep network

- PPO uses two deep networks
  - policy network: to predict the probability of selecting an action at a state
  - value function network: to estimate the value function
  - both networks share information
- Remember DQN? It uses value network plus target value network



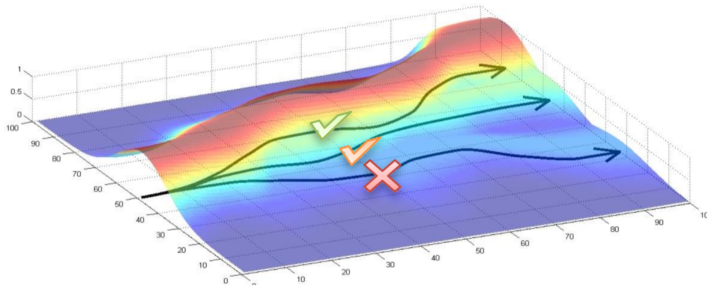
How good is an action within a trajectory?

- Given a trajectory, we need to compute how much better was the action **a** that an agent took at state **s** compared with the expectation of what normally happens from state **s**
- If so, we can determine whether the action **a** that the agent took was better than expected or worse
- Recall that the **advantage** $A^{\pi}(st\ , at)$ tells us how much better than the average action a_t is according to π

How to estimate advantages (relative value of actions)

- Given π (policy network), say that a trajectory does a_t at state s_t

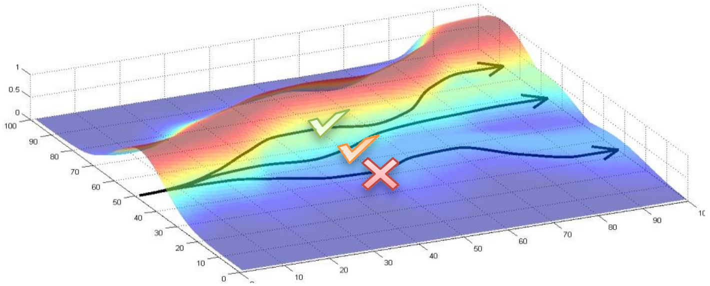
$\hat{A}_t$ $=$ Return (discounted rewards) (<u>observed</u> from **experience**) $-$ Baseline estimate



How to estimate advantages (relative values)

- Given π (policy network), say that a trajectory does a_t at state s_t

$\hat{A}_t$ = Return (discounted rewards) (observed from **experience**) − Value estimate (predicted from **value network**)



We know what happened

What did we expect would happen?

How to estimate advantages (relative values)

- Given π (policy network), say that a trajectory does a_t at state s_t

$\hat{A}_t$ = Return (discounted rewards) (<u>observed</u> from **experience**) − Value estimate (<u>predicted</u> from **value network**)

Observed environment state **s** → Value function network → V(s) = Estimate of the **return** from time t onwards

Policy gradient

- Given π (policy network), say that a trajectory does a_t at state s_t
- Policy gradient depends on the advantage estimates $\hat{A}_t$

Positive advantage?
Better than average return

Policy gradient loss

- Given π (policy network), say that a trajectory does a_t at state s_t
- Policy gradient depends on the advantage estimates $\hat{A}_t$

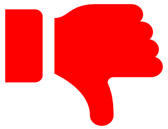
Positive advantage?

Gradient is positive!

Increase the probability of selecting the action in the future when we encounter the same state

Policy gradient

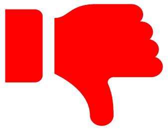
- Given π (policy network), say that a trajectory does a_t at state s_t
- Policy gradient depends on the advantage estimates $\hat{A}_t$



Negative advantage?
Worse than average

Policy gradient

- Given π (policy network), say that a trajectory does $\underline{a_t}$ at state s_t
- Policy gradient depends on the advantage estimates $\hat{A}_t$



Negative advantage?

Gradient is negative!

Decrease the probability of selecting the <u>action</u> in the future when we encounter the same state

Main objective function in PPO

- PPO's goal is not to move too far away from the old policy when doing policy updates.
- PPO penalises changes to the policy that move $r_t(\theta)$ away from 1.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

Main objective function in PPO

- PPO's goal is not to move too far away from the old policy when doing policy updates.
- PPO penalises changes to the policy that move $r_t(\theta)$ away from 1.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big]$$

We will compute this objective function over batches of trajectories

Main objective function in PPO

- PPO's goal is not to move too far away from the old policy when doing policy updates.
- PPO penalises changes to the policy that move $r_t(\theta)$ away from 1.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(\underbrace{r_t(\theta)\hat{A}_t}_{\text{Policy gradient objective}}, \underbrace{\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t}_{\text{Clipped version of policy gradient objective}})\right]$$

Main objective function in PPO

- PPO's goal is not to move too far away from the old policy when doing policy updates.
- PPO penalises changes to the policy that move $r_t(\theta)$ away from 1.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \underbrace{\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t}_{\text{This clipping avoids that the updated policy moves too far from the old policy}})\right]$$

Main objective function in PPO

- PPO's goal is not to move too far away from the old policy when doing policy updates.
- PPO penalises changes to the policy that move $r_t(\theta)$ away from 1.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\underline{\min}(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

The final objective is pessimistic

Final training objective (loss function) in PPO

- Function to maximise

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

Return component

POLICY
NETWORK

Final training objective (loss function) in PPO

- Function to maximise

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

Value estimation component
Square-error loss of value estimation

VALUE NETWORK

Final training objective (loss function) in PPO

- Function to maximise

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + \underbrace{c_2 S[\pi_\theta](s_t)}_{\text{Entropy component}}\right]$$

Entropy component

- to ensure sufficient exploration
- pushes the policy to behave more randomly until the other parts of the objective start dominating

The PPO algorithm

- To optimize policies, PPO alternates between sampling data from the policy and performing several epochs of optimization on the sampled data

[Schulman et al, 2017] Proximal policy optimization algorithms

The PPO algorithm

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

COLLECT EXPERIENCE

Current policy from the policy network interacts with the environment generating trajectories

[Schulman et al, 2017] Proximal policy optimization algorithms

The PPO algorithm

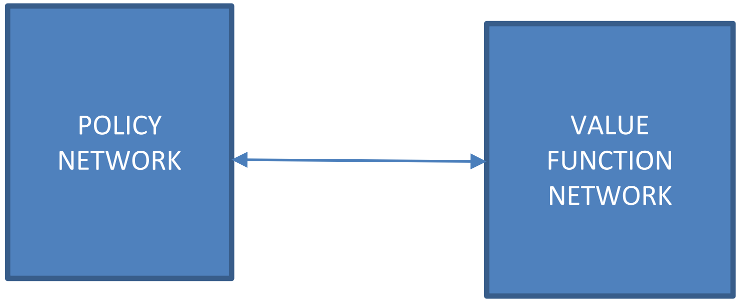
Algorithm 1 PPO, Actor-Critic Style

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., N do
        Run policy π_θold in environment for T timesteps
        Compute advantage estimates Â_1, ..., Â_T
    end for
    Optimize surrogate L wrt θ, with K epochs and minibatch size M ≤ NT
    θ_old ← θ
end for
```

ESTIMATE HOW GOOD TRAJECTORIES ARE
Estimate advantages from the collected returns from experience and the estimated returns from the value function network

[Schulman et al, 2017]

The PPO algorithm



Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

UPDATE POLICY AND VALUE FUNCTION NETWORK

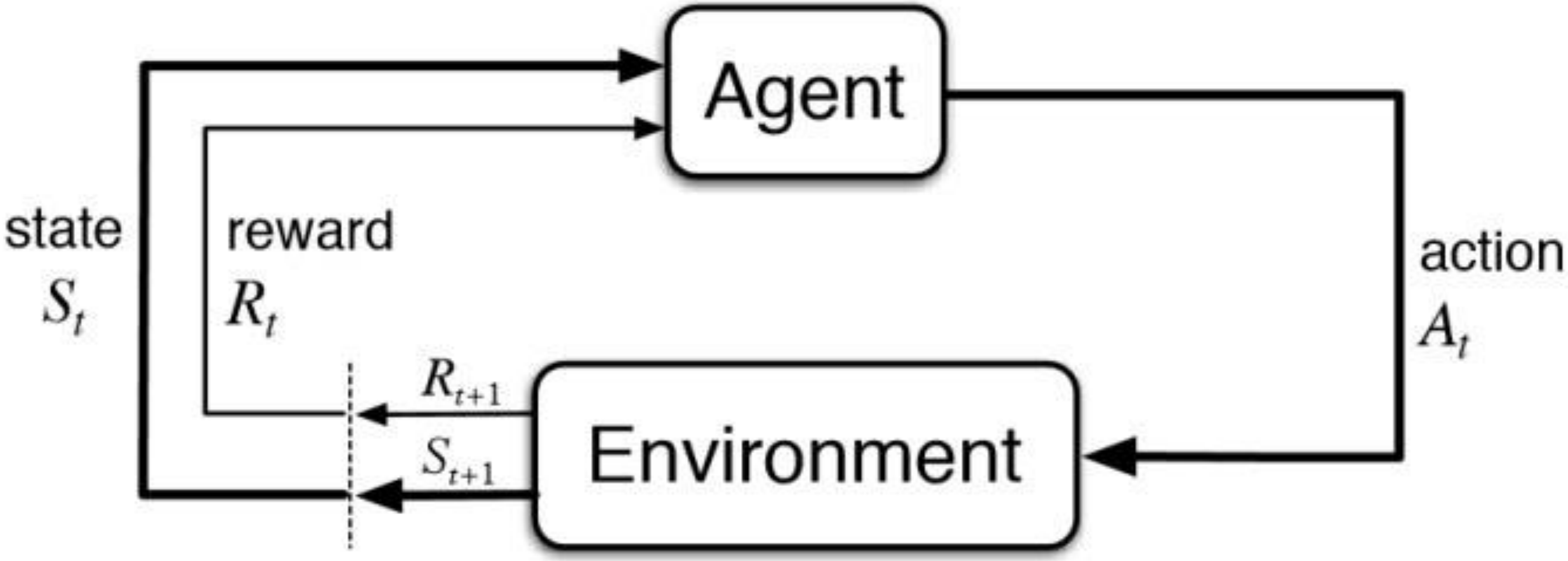
- PPO runs several epochs of optimization on the sampled data
- Runs gradient descent on the policy network
- Does supervised learning on the value function network by using the collected experiences (sampled data)

[Schulman et al, 2017] Proximal policy optimization algorithms

Section 3: Now with multiple agents

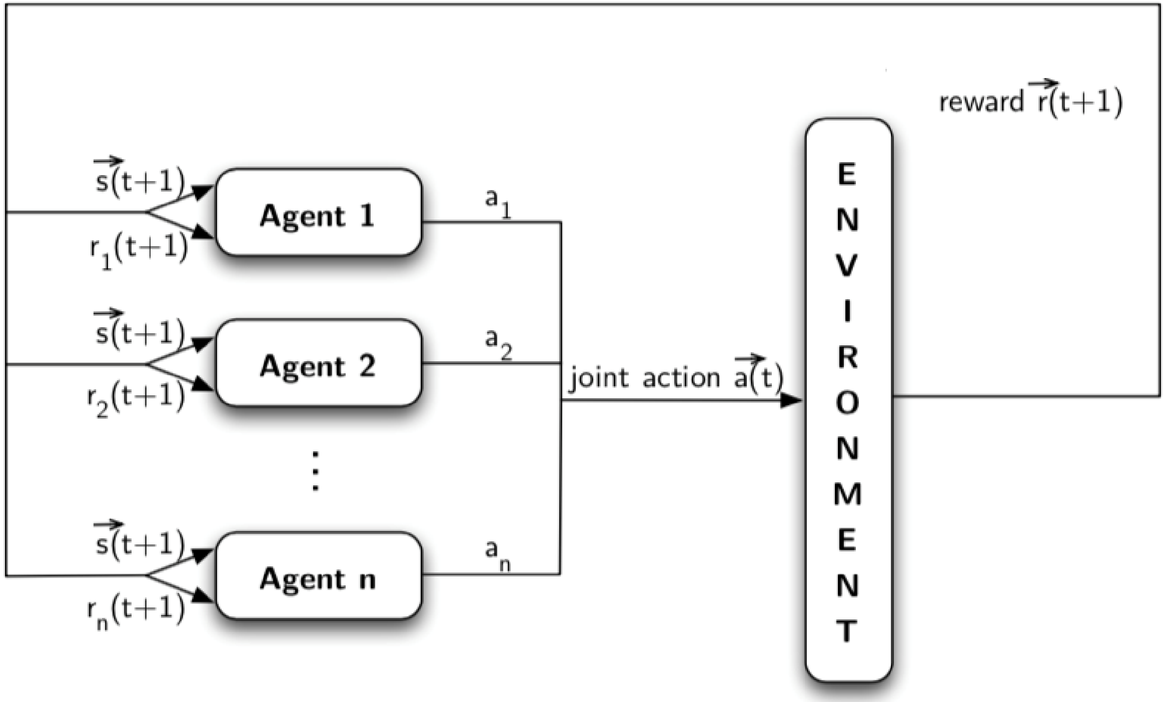
Manel Rodríguez-Soto
Juan A. Rodríguez-Aguilar
ICMAT@Madrid
28.04.2023

From MDP's to Markov games



MARKOV DECISION PROCESS

From MDP's to Markov games



MARKOV GAME

Markov Games

- The generalisation of Markov decision processes. The differences are: $\mathcal{A}^i$
- Each agent has its set of actions:

Markov Games

- The generalisation of Markov decision processes. The differences are:
- Each agent has its set of actions: $\mathcal{A}^i$
- Transitions depend on the actions of all agents:

$$T : \mathcal{S} \times \mathcal{A}^1 \times \cdots \times \mathcal{A}^m \to \mathcal{S}$$

Markov Games

- The generalisation of Markov decision processes. The differences are: $\mathcal{A}^i$
- Each agent has its set of actions:
- Transitions depend on the actions of all agents:

$$T : \mathcal{S} \times \mathcal{A}^1 \times \cdots \times \mathcal{A}^m \rightarrow \mathcal{S}$$

- Each agent has a different reward function:

$$R^i : \mathcal{S} \times \mathcal{A}^1 \times \cdots \times \mathcal{A}^m \rightarrow \mathbb{R}$$

Types of Markov games

- Pursuing an individual best-response (consequently creating a Nash Equlibrium) is only a good idea in very **particular** situations.

- We divide Markov games in three kinds depending on the objective of the game:

- 1. Fully cooperative games
- 2. Fully competitive games
- 3. Mixed games

I. Fully Cooperative games

- All the agents have the same reward function.

$$R^1 = R^2 = R^3 = \cdots = R^n$$

- The goal is to maximize the reward, no ne
  responses.

I. Fully Cooperative games

- All the agents have the same reward function.

$$R^1 = R^2 = R^3 = \cdots = R^n$$

- The goal is to maximize the reward, no need for best responses.

- If a centralized controller, can be simplified as an MDP.

I. Fully Cooperative games

- All the agents have the same rewards

$$R^1 = R^2 = R^3 = \cdots = R^n$$

- Currently there are new approximations in which agents rewards can be different but the goal is to maximise the mean reward.
- This setting facilitates decentralysed algorithms.

II. Fully Competitive games

- Each agent has its own reward in conflict with the others. This is also called a zero-sum game:

$$R^1 + R^2 + R^3 + \cdots + R^n = 0$$

- Necessity to calculate best responses.
- Impossible to centralise.

II. Fully Competitive games

- Each agent has its own reward in conflict with the others. This is also called a zero-sum game:

$$R^1 + R^2 + R^3 + \cdots + R^n = 0$$

- Necessity to calculate best responses.
- Impossible to centralise.
- This setting is also used as a model for **robust** learning a safe strategy that always obtains a minimum reward (minimax strategy).

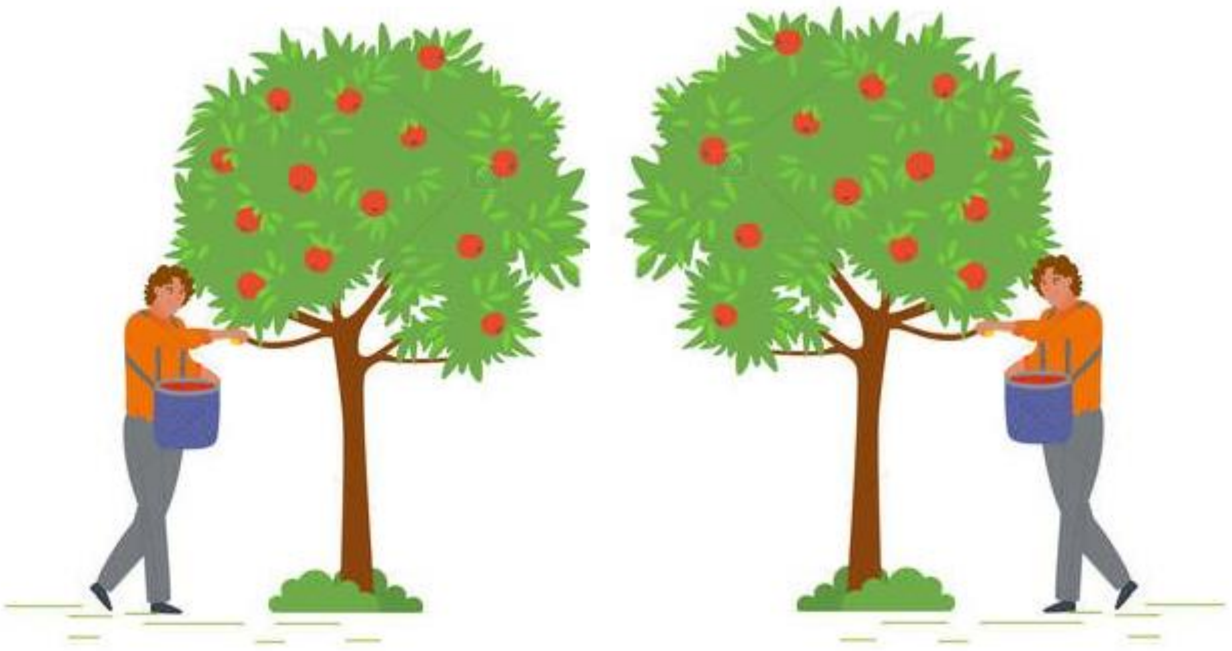
III. Mixed games

- Rewards not in conflict but not shared neither.
- Not clear how to answer them (problem dependant).
- *Typical solution:* provide each agent a single-agent RL algo. and hope everything works.

Example: Ethical Gathering Game

Example: Gathering Game

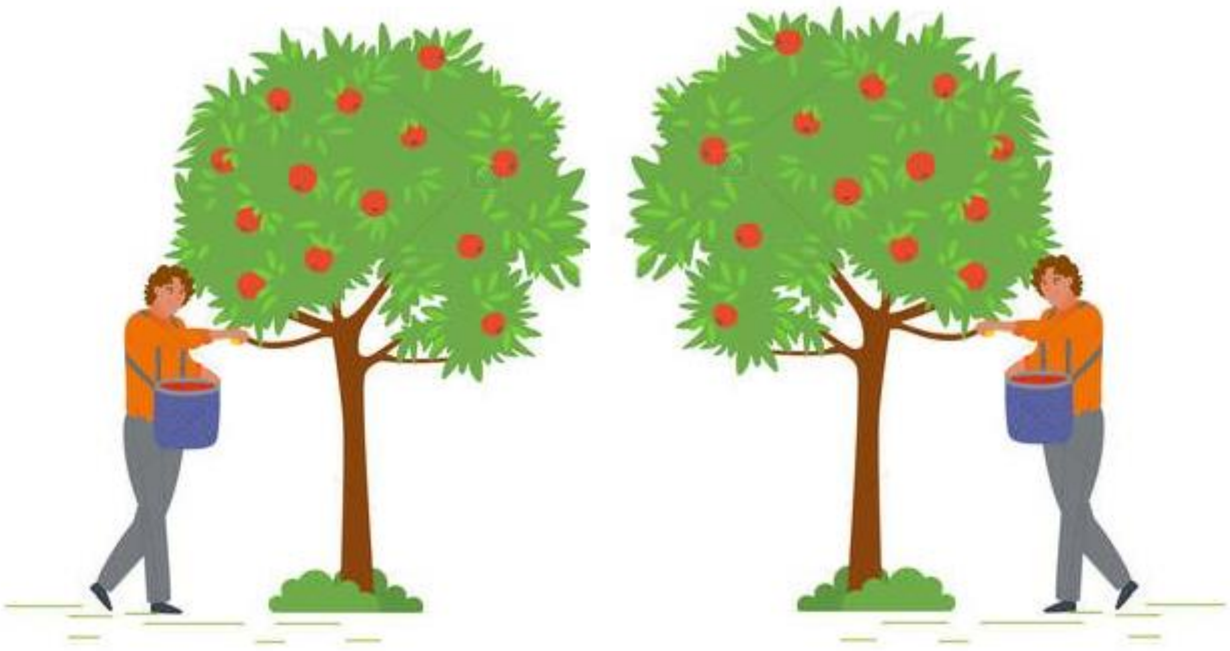
- Imagine a field with apples, and several agents collecting them for eating, each working independently.



From: Leibo et al. (2017): *Multi-agent reinforcement learning in sequential social dilemmas*, AAMAS’17.

Example: Gathering Game

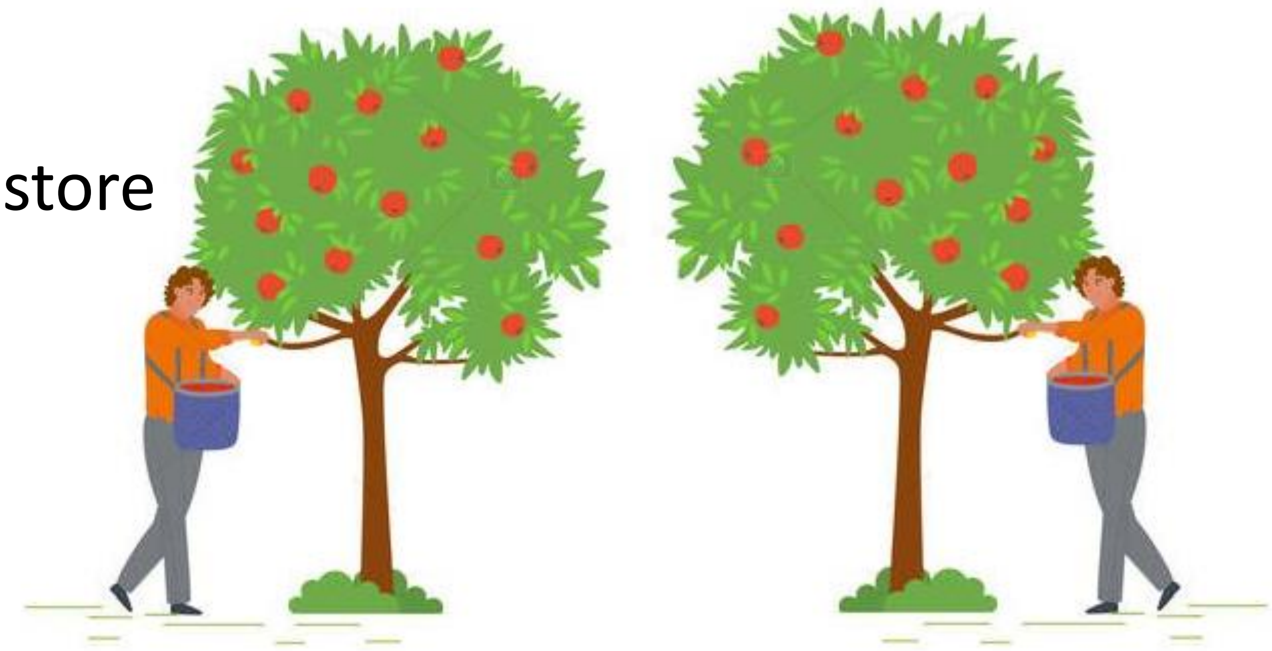
- **Individual objective:** collect as many apples as possible.
- **Ethical objective:** guarantee that everyone has enough apples.



Example: Gathering Game

- **Individual objective:** collect as many apples as possible.
- **Ethical objective:** guarantee that everyone has enough apples.

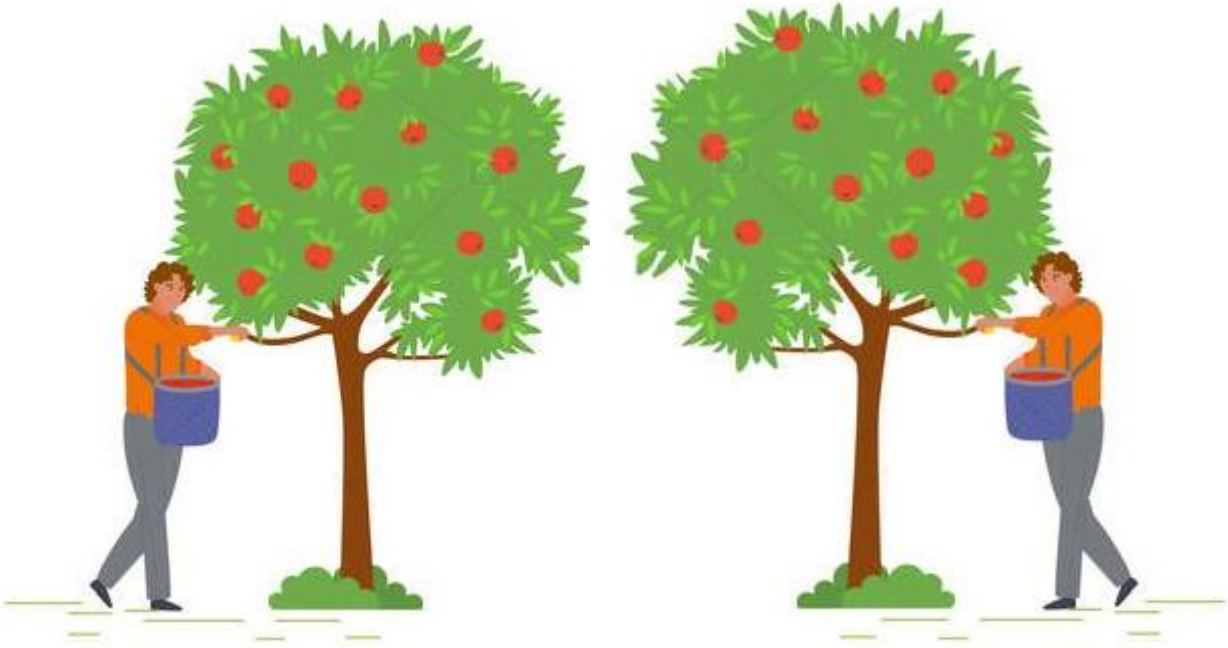
- Each agent has its own store,
- but they can donate to a common store



Example: Gathering Game

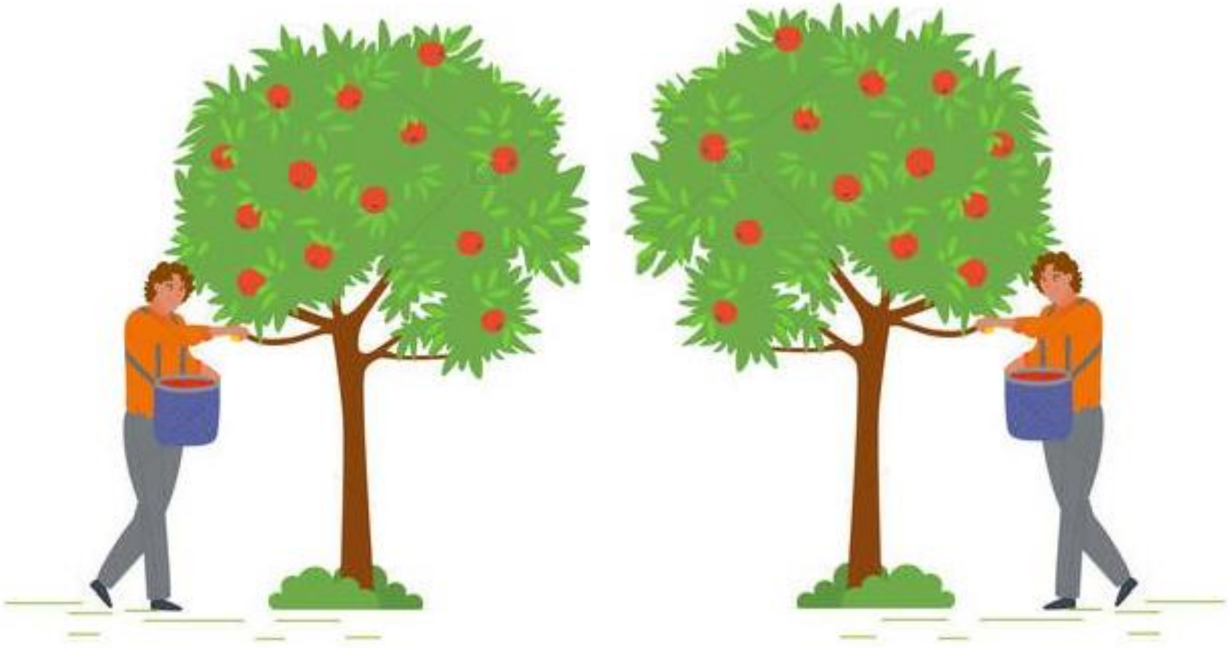
- **+ve reward:** each time an agent collects an appl
- **+ve reward:** each time an agent donates an apple

(and already has many apples).



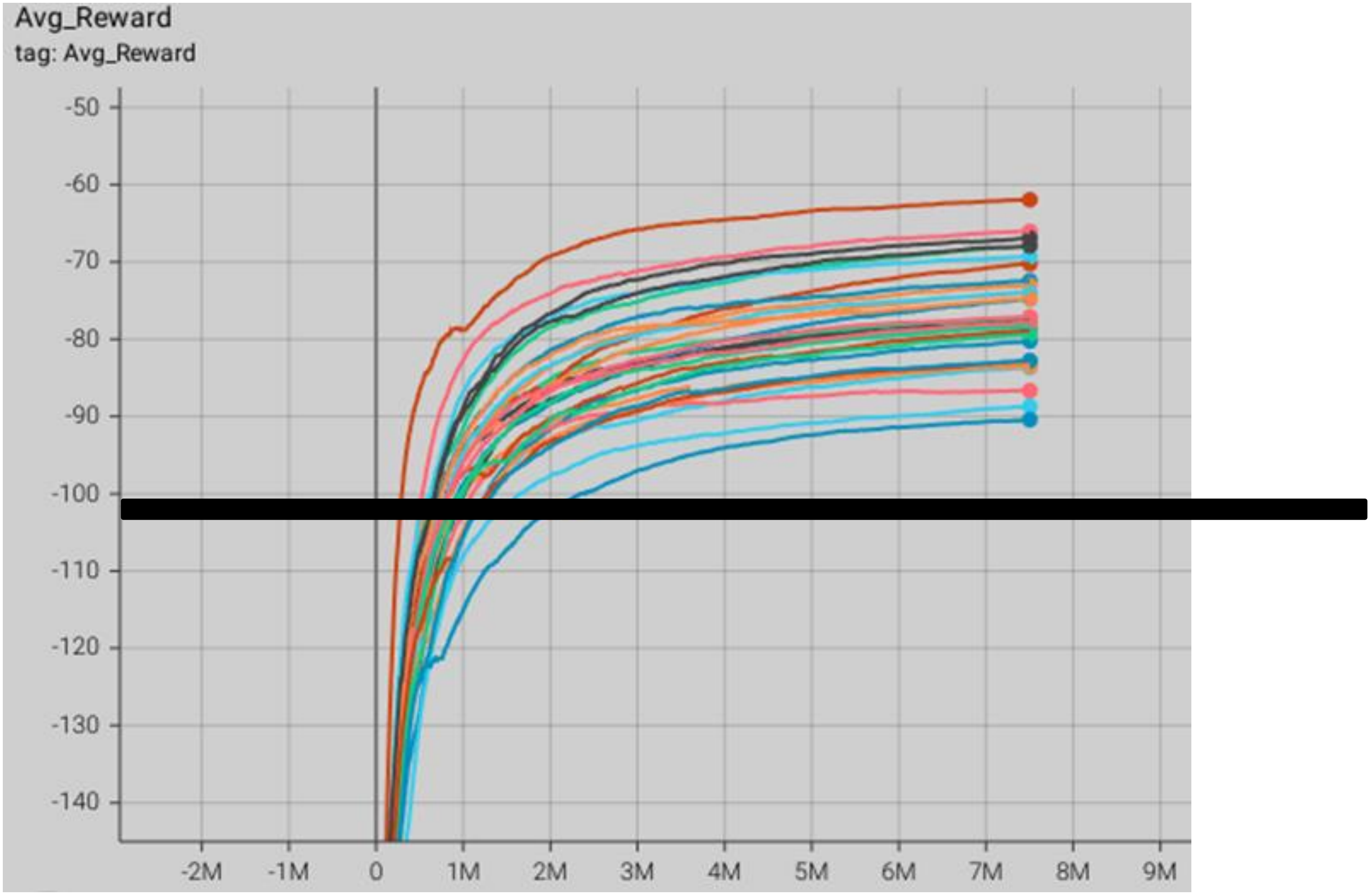
Our solution: Independent Learns

- ***Algorithm: Independent PPO*** (i.e., each agent uses PPO independently).



Results

Both agents survive thanks to donations

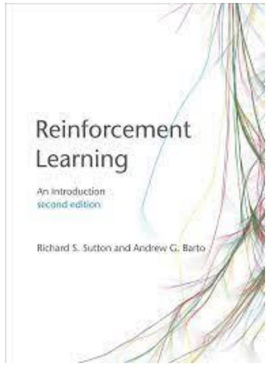


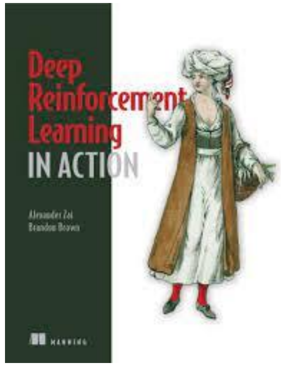
SURVIVAL THRESHOLD

To learn and implement

Resources to explore further

- Excellent course on RL
  - David Silver (UCL and Deepmind): https://www.davidsilver.uk/teaching/
- Excellent on-line courses on DRL
  - Sergey Levine (UC Berkeley): https://rail.eecs.berkeley.edu/deeprlcourse/
  - Pieter Abbeel (Open AI, UC Berkeley, Gradescope): https://www.youtube.com/@PieterAbbeel
  - Deep RL Bootcamp: https://www.youtube.com/watch?v=qaMdN6LS9rA&list=PLXoDfcPNqdnkdhRCrCCdVUOtKOwuBhJdF
- Books
  - Zai & Brown's "DRL in Action"
  - Sutton & Barto's RL "An introduction"





Resources to implement algorithms

- Gym
- Mujoco
- OpenAI DRL library: https://github.com/openai/baselines
- DRL Policy gradients: https://github.com/higgsfield/RL-Adventure-2