UNIVERSITY of York
Europe Campus
CITY College

# Reinforcement Learning

## Markov Decision Processes, Temporal Difference Learning, Q-Learning

Dr Ioanna Stamatopoulou

All material in these lecture notes is based on our textbook:
Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 3rd ed., O'Reilly, 2022

Additional recommended book on Reinforcement Learning:
Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: an Introduction*, 2nd ed., MIT Press, 2022

# Reinforcement Learning (RL)

## A bit of History

- RL has been around since the 1950s
  - Primarily for game-playing (e.g. Backgammon)
- Breakthrough: DeepMind (2013)
  - A system that learnt to play any Atari game from scratch
  - Without any prior knowledge of the rules!
- This resulted in AlphaGo
  - A system that beat two (human) world champions in the game of Go (2016-2017)

# Throwback to the Introduction

An AI **agent** is a software program designed to interact with its environment, perceive the data it receives, and take actions based on that data to achieve specific goals.

AI agents simulate intelligent behavior, and they can be as simple as rule-based systems or as complex as advanced machine learning models.

- A very different beast!
- The learning system is an **agent** that
  - Observes the environment
  - Selects an action using some **policy**
  - Acts!
  - Gets **reward** or **penalty**
  - **Updates policy** (learning step, i.e. the goal is to learn the policy)
- …and iterates the above until an optimal policy is found

# Examples

- The agent can be the program controlling a robot. The environment is the real world, the agent observes the environment through a set of sensors (cameras,  touch sensors), and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.
- The agent can be the program controlling Ms Pac-Man. The environment is a simulation of the Atari game, the actions are the possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.

# Examples

- The agent can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
- The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.
- The agent can be a robotic vacuum cleaner whose reward is the amount of dust/dirt it picks up in a period of 30 minutes.

# Policy

- **Policy** is the algorithm that the agent uses in order to determine its actions
- It can literally be any kind of algorithm!
  - Deterministic or stochastic (meaning involving randomness, e.g. the robot vacuum can move randomly in space)
  - May need to observe the environment or not (e.g. the robot vacuum does not need to observe the environment)

# Policy Parameters ⇒ Policy Search

- A possible stochastic policy for the robot vacuum might be:
  - The robot moves forward every second with probability ***p***
  - The robot rotates left or right in an angle between ***-r*** and ***r*** (with probability 1-***p***)
- This policy has two parameters: ***p*** and ***r***
- Training the robot means finding the best policy parameter values

  **⇒ Policy Search**

# Policy Search

## Finding the policy's parameters' values

Searching through the **policy space** to find the best values for the policy's parameters. Options include:

- Brute-Force
- Genetic Algorithms
- Optimisation Techniques
  - NNs - optimisation using Policy Gradients (gradient ascend)

# Policy Search

## Policy Space

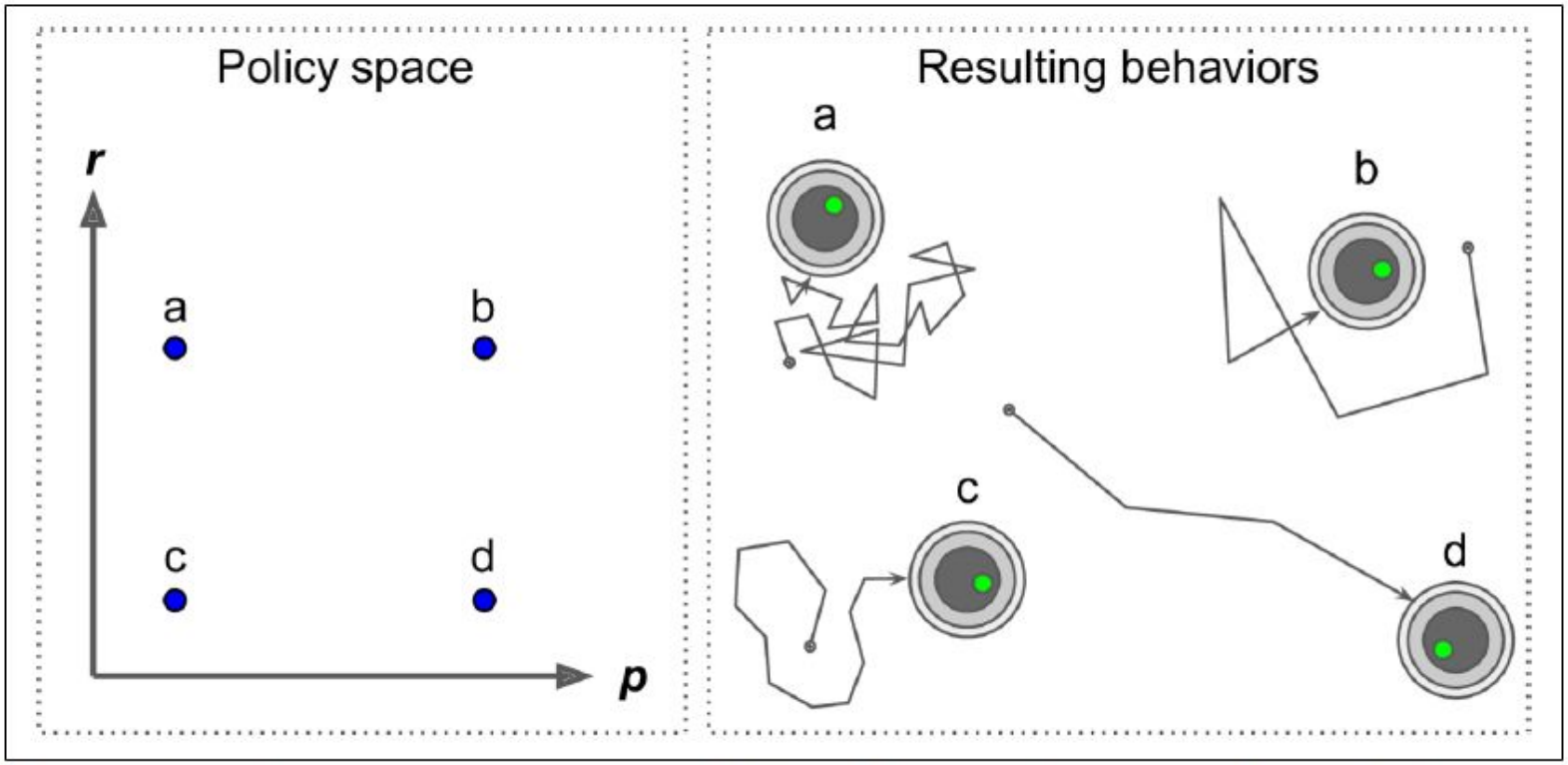


*Figure 18-3. Four points in policy space (left) and the agent's corresponding behavior (right)*

# Policy Search

## Brute-Force

- Try out many combinations of values for the parameters
- Pick the best!
- Highly inefficient, when the policy space is too large, which makes this approach literally impractical in most cases

# Policy Search

## Genetic Algorithms

Nature-Inspired Computing…

- Randomly create a first generation of policies (e.g., 100)
- Try them and then "kill" the worst policies (e.g., 80)
- Have the surviving policies produce offsprings
  - An offspring is a copy of the parent plus some random variation (asexual reproduction), OR
  - An offspring is a random combination of its parents (sexual reproduction)
  - The survivors together with their offsprings form the 2nd generation
- Continue until you reach a good policy

# Policy Search

## Optimisation Techniques

- Evaluate the gradients of the rewards wrt the policy parameters

  $\Rightarrow$ Gradient Ascent

*(outside the scope of this module but you will talk about this in Deep Learning)*

# Neural Network Policies

Deep Learning…

- Create a Neural Network that:
  - Receives an observation as input
  - Outputs a probability for each action
- **Pick an action randomly** based on these probabilities
- Issues to consider:
  - If each observation does not contain the environment's full state
  - If the observations are noisy

  you may need to take into account previous states as well, to better determine/estimate the current state

# The exploration/exploitation dilemma

- Why pick an action randomly and not just pick the one with the highest score?
- Strategy for balancing between:
  - **Exploiting** actions that we know work well
  - **Exploring** other actions that may turn out to be good/better
- Different context:
  consider YouTube's recommendations

# The Credit Assignment problem

- Rewards are typically sparse and delayed
  - We do not know whether an action was good immediately after doing it
  - e.g., when the robot vacuum collects a lot of dust at the end of the 30', how does it know which actions contributed positively/negatively to the result?

**⇒ credit assignment problem**

# The Credit Assignment problem

## Action Return

$$r_{t0} + \gamma \times r_{t1} + \gamma^2 \times r_{t3}$$

where $r_{t0}$, $r_{t1}$, $r_{t2}$ are the rewards at time instances $t_0$, $t_1$, $t_2$, etc.

- A solution is to evaluate the **action's return**:
  - The sum of all the rewards that are received after the action, after applying a **discount factor *γ***
  - ***γ*** is in the range [0,1]
  - **~0**: future rewards do not count as much
  - **~1**: rewards far into the future count almost as much as immediate rewards
  - Typically varies from 0.9 to 0.99
  - 0.95 means that rewards 13 steps into the future count half as much as immediate rewards

# The Credit Assignment problem

## Action Advantage

- We want to know how much better/worse an action is, compared to other actions, **on average**
- Run enough times, i.e. many **episodes**
- Normalise all the action returns
  - Subtract the mean and divide by the standard deviation

**⇒ action advantage**

# Markov Decision Processes

# Markov Chains

- Stochastic processes with no memory
- Fixed number of **states**
- **Transition** from a state ***s*** to another ***s'*** has a fixed **probability**
  - It only depends on the pair ***(s, s')***
  - That's why we say there is no memory

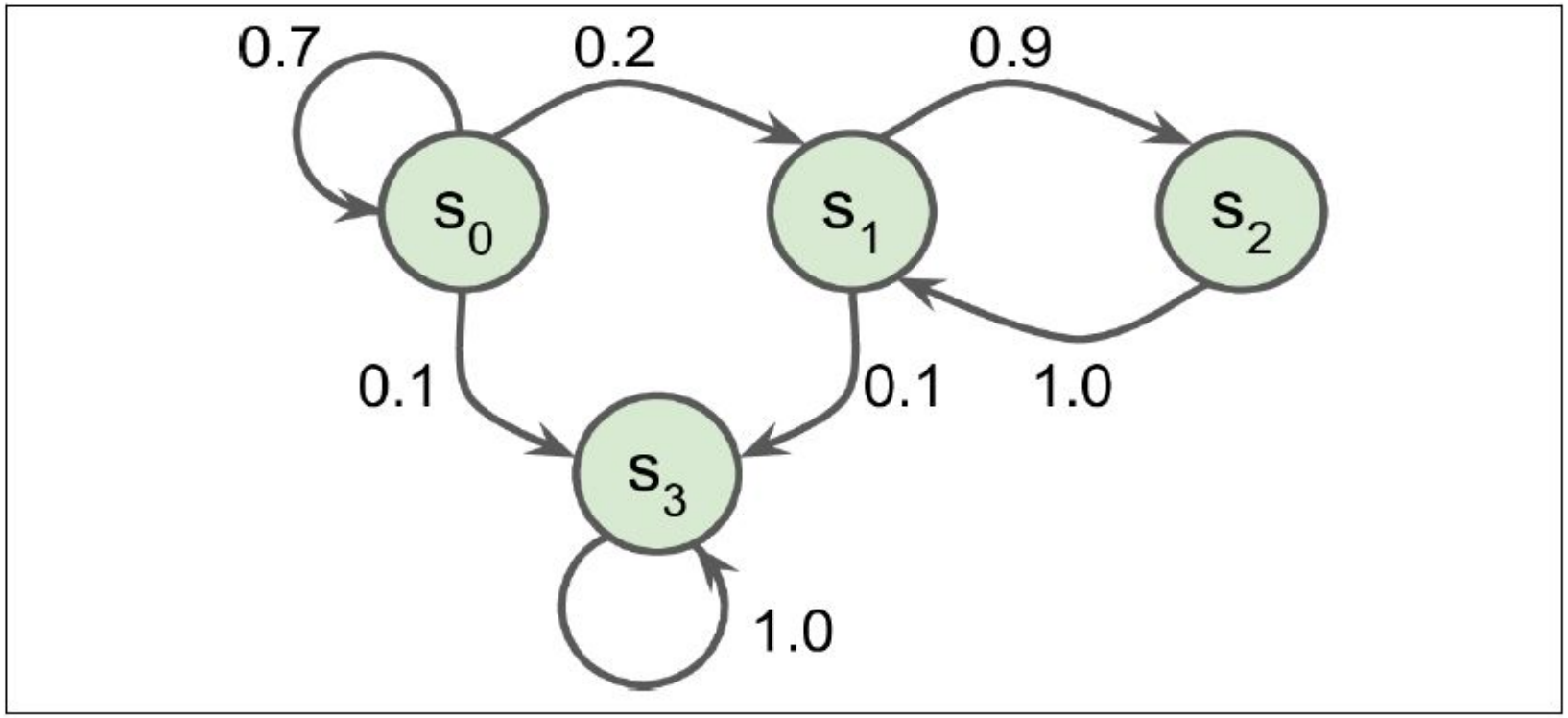


*Figure 18-7. Example of a Markov chain*

# Example of a Markov Chain

# Markov Decision Processes (MDPs)

- MDPs are based on Markov Chains but with the following additional features:
  - In any state ***s*** the agent may select an action ***a***
  - Transition probabilities depend on the the chosen action
  - Some transitions return a reward, positive or negative
- The goal of the agent is to find the policy that will maximise the reward over time

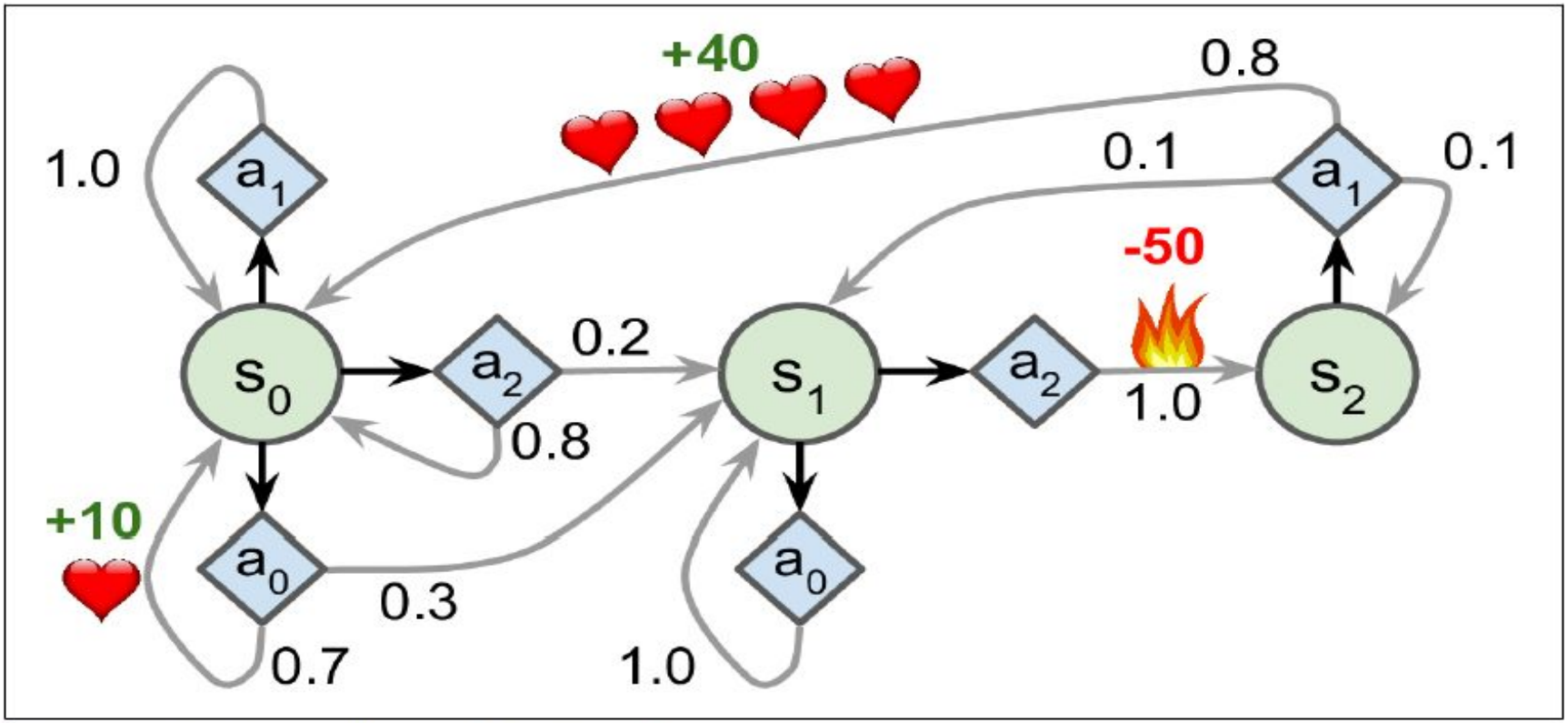


*Figure 18-8. Example of a Markov decision process*

# Example of a Markov Decision Processes

# Markov Decision Processes

## Optimal State Value

*Equation 18-1. Bellman Optimality Equation*

$$V^*(s) = \max_a \sum_s T(s,a,s')[R(s,a,s') + \gamma \cdot V^*(s')] \quad \text{for all } s$$

The **optimal state value *V*** of any state ***s***, ***V*(s)***, is the sum of all discounted rewards the agent can expect on average, assuming it acts optimally

- ***T(s, a, s′)***: the transition probability from state ***s*** to state ***s′***, given that the agent chose action ***a***
- ***R(s, a, s′)***: the reward that the agent gets going from state ***s*** to state ***s′***, given that the agent chose action ***a***
- ***γ*** is the discount factor

# Markov Decision Processes

## Value Iteration algorithm

*Equation 18-2. Value Iteration algorithm*

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot V_k(s')\right] \quad \text{for all } s$$

The **value iteration algorithm** is used to calculate the **optimal state values**

- Initialise all optimal state values to zero
- Iteratively update them using the above equation
  - $V_k(s)$: the state value at iteration $k$
- Given enough iterations, the algorithm is guaranteed to converge to the optimal state values

# Markov Decision Processes

## Quality values

*Equation 18-3. Q-Value Iteration algorithm*

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s'a)$$

- **Knowing the state values does not constitute a policy because it does not help the agent identify the action to be selected**; it only indicates if a state is "good" or "bad"
- The **optimal Q-Value** , ***Q*(s, a)***, of a state-action pair ***(s, a)*** is the sum of discounted future rewards the agent can expect on average after it reaches the state ***s*** and chooses action ***a***
- A Q-Value Iteration algorithm is used to determine the Q-Values
- The policy ***π*** then is: when in a state ***s***, select the action ***a*** with the highest Q-Value for that state

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$$

# Temporal Difference Learning (TDL)

- **Temporal Difference Learning** expresses the idea that the agent does not need to wait until the end of an episode in order to be able to update the value estimates and policies
  - Step-by-step updates
  - Monte Carlo methods are episode-by-episode (outside the scope of this module)
- TDL is particularly useful when the agent initially:
  - Knows only the states and the actions
  - Does not know the transition probabilities ***T(s, a, s')***
  - Does not know the rewards ***R(s, a, s')***

# Markov Decision Processes

## TDL for finding the state values

- **TDL** techniques can be used to **adapt the Value Iteration algorithm**
- The algorithm uses an exploration policy (e.g. random) to explore the MDP
- For each state ***s***, it keeps a running average of the immediate and future-expected rewards

# Q-Learning

TDL for finding the (state-action) Q-Values

# Q-Learning

- Same way TDL can be used as to adapt the Value Iteration algorithm for when the transition probabilities and the rewards are unknown, it can be used also for the Q-Value Iteration algorithm
- **Q-Learning** is an **adaptation of the Q-Value Iteration algorithm** for when the transition probabilities and the rewards are unknown

# Q-Learning (cont'd)

- The algorithm learns by watching the agent play randomly, gradually improving the estimates of the Q-Values

- Note: Q-Learning (like Stochastic Gradient Descent) uses a learning rate $\alpha$ that needs to be gradually reduced for the algorithm to converge

# Q-Value iteration vs Q-Learning

## Time needed to converge

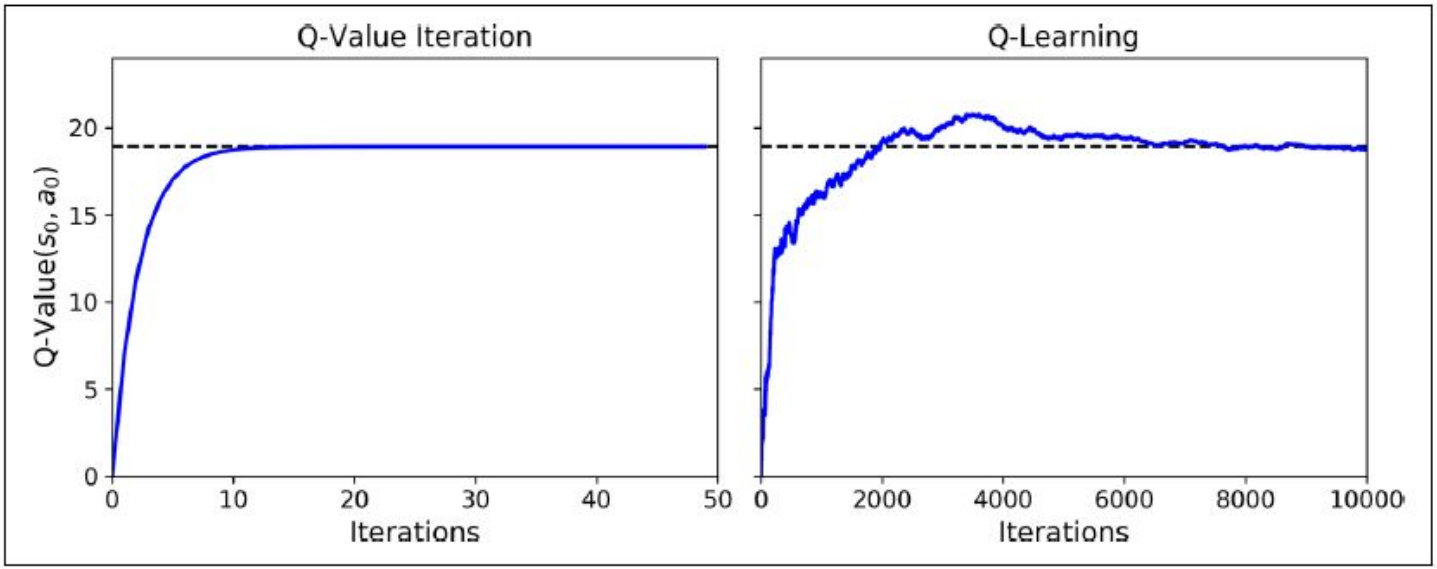


*Figure 18-9. The Q-Value Iteration algorithm (left) versus the Q-Learning algorithm (right)*

Understandably, knowing the transition probabilities and the rewards beforehand makes a huge difference…

# Off-policy vs on-policy learning algorithms

- Q-Learning is an **off-policy** algorithm:
- The policy being trained for the agent is not the one being used during exploration/learning (i.e. random)
  - After training (once the it has accurate Q-Value estimates) the best policy for the agent is the **greedy policy**: select the action with the highest Q-Value
- **On-policy** algorithms explore the world using the policy being trained aiming to improve it (e.g. Value Iteration)

# Advantages of off-policy algorithms

- Continuous exploration while learning the optimal policy
  - On-policy may lead to a suboptimal policy
- Learning from Demonstration
  - Learns by watching the agent act randomly

# Other Exploration Policies

## ε-greedy: Exploration vs Exploitation

- Instead of exploring totally randomly, a better policy is to follow an ***ε*-greedy policy**
- At each step the agent acts:
  - Randomly with probability ***ε***, or
  - Greedily with probability **1-*ε***
  - Exploration/Exploitation balance…
- Start with a high value for ***ε*** (favouring exploration)
- As Q-Values become more accurate, gradually reduce ***ε*** (favouring exploitation)

# Other Exploration Policies

## Exploration function

- Another approach is to encourage the exploration policy to try actions that it has not tried much before
- Q-Learning incorporates an exploration function that uses:
  - The number of times an action ***a*** has been chosen in state ***s*** in the past
  - A curiosity hyperparameter that defines how much the agent is attracted to the unknown (i.e. how likely it is to choose actions less chosen in the past)

# Coming up next:
# What does an ML Project typically involve?

(abstractly…)

- Study the data
- Select an appropriate ML model/algorithm
- Train the model on the data
- Apply the model to make predictions on new cases

  ⇒ Inference!

# **Coming up next:**
# What does an ML Project typically involve?

(more refined…)

1. Look at the big picture
2. Get the data
3. Explore and visualize the data to gain insights
4. Prepare the data for ML algorithms
5. Select a model and train it
6. Fine-tune your model
7. Launch, monitor, and maintain your system

Chapter 2 of our textbook takes you through these stages in detail for a particular Case Study