Reinforcement Learning

Unlocking the Power of AI Agents

Gianluca Aguzzi gianluca.aguzzi@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Talk @ Advanced School in Artificial Intelligence (ASAI)

26/07/2023

- PhD student in Computer Science and Engineering
- Research interests:
  - Multi-agent systems
  - Distributed Collective Intelligence
  - Deep Reinforcement Learning
  - Multi-agent Reinforcement Learning

Resources

- *An Introduction to Reinforcement Learning*, Sutton and Barto,1998
  - Available online at `http://incompleteideas.net/book/the-book-2nd.html`
- *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*, Laura Graesser and Wah Loon Keng, 2020
- *Deep Mind Lectures*:
  - **Introduction to Reinforcement Learning with David Silver**: `https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver`
  - **Reinforcement Learning Lecture Series**: `https://www.deepmind.com/learning-resources/reinforcement-learning-lecture-series-2021`

Contents

1. Introduction
2. Formalisation
3. Solving Reinforcement Learning problems
   - Tabular methods
   - Approximate methods – Deep Reinforcement Learning
4. Conclusion and open problems

What is Reinforcement Learning?

What is Reinforcement Learning?

It is related with the concept of **intelligence**

What is Intelligence?

What is Intelligence?

[..] otherwise called “good sense”, “practical sense”, “initiative”, the faculty of adapting one’s self to circumstances

What is Intelligence?

Goal-directed adaptive behavior

What is Intelligence?

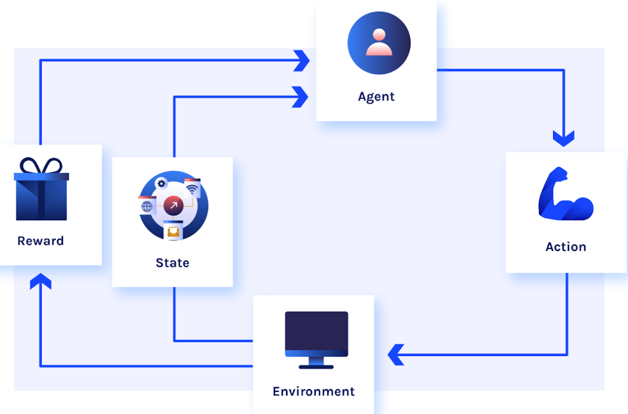
learning *to make* ***decisions*** *to achive* ***goals***

What is Reinforcement Learning?

- Animals learn by interacting with our environment
  - Babies learn how to communicate by interacting with parents
  - Dogs learn how to behave by following the owner's orders
  - Me learn how to surfing by falling from the surfboard
- Difference from supervised learning:
  - **active** learning (learn by doing)
  - **sequential** interaction
  - **delayed** feedback
- Learning guided by **goal** (goal-directed)
- Learning without examples ➔ guided by reward signal

Interaction loop

An **agent** interacts with the **environment** by perceive an **observation** and take an **action** accordingly which leads to a **reward**



Goal: maximize the **cumulative reward** over time

On problem expressiveness: reward hypothesis

Reward hypothesis

Any goal can be formalized as the outcome of maximizing a cumulative reward[a]

- Reinforcement learning is based on this hypothesis
- Ideally, we can formalize any problem as a reinforcement learning problem

[a] http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/rewardhypothesis.html

Stronger statement: reward is enough

intelligence, and its associated abilities, can be understood as subserving the maximisation of reward by an agent acting in its environment. [a]

- Really controversial

[a] David Silver et al. "Reward is enough". In: *Artificial Intelligence* 299 (Oct. 2021), p. 103535. doi: 10.1016/j.artint.2021.103535. url: https://doi.org/10.1016/j.artint.2021.103535

Examples of Reinforcement Learning problems

- Learning how to surf → **Reward:** surfing time on the wave
- Managing a portfolio of cryptocurrencies → **Reward:** money earned
- Controlling the battery of an electric car → **Reward:** battery level at the end of the day
- Playing chessboard → **Reward:** win the game
- Solving a cubic cube → **Reward:** solving time

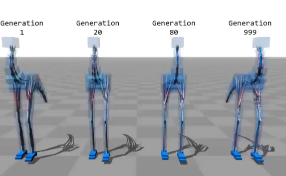
NB! if the goal is learn via environment interaction, then these are all reinforcement learning problems, regardless the algorithm involved

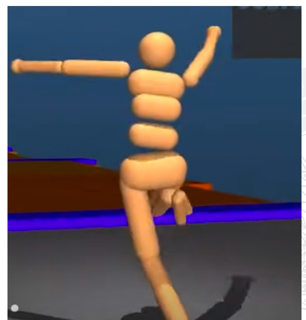
Again, What is Reinforcement Learning

- There are several reasons why we should learn:
  1. Find **solutions**
     - A robot that reaches a target
     - A program that plays chess (really well)
  2. Adapt **online** (dealing with unknowns)
     - A robot that learns how to walk in a new environment
     - A program that learns how to play a new game
- Reinforcement learning is used in both cases
- Episodic vs continuing tasks
- Adapting online is more challenging, and it is not just generalization (e.g. supervised learning)
- Is it planning? ➔ No, the *model* is not known

Motivating real world examples

Robotics, Games, Finance, Healthcare, . . .





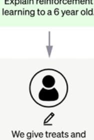
ChatGPT [1]

Step 1

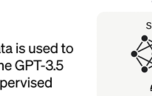
Collect demonstration data and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.



This data is used to fine-tune GPT-3.5 with supervised learning.



Step 2

Collect comparison data and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.

Step 3

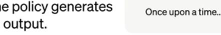
Optimize a policy against the reward model using the PPO reinforcement learning algorithm.

A new prompt is sampled from the dataset.

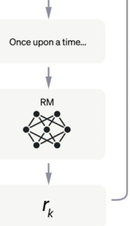
The PPO model is initialized from the supervised policy.

The policy generates an output.

The reward model calculates a reward for the output.

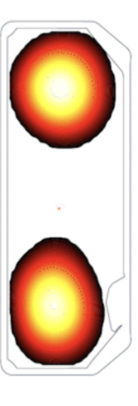


The reward is used to update the policy using PPO.

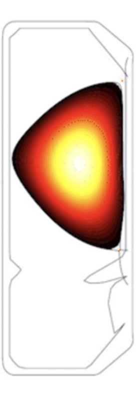
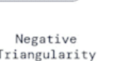


[1] https://openai.com/blog/chatgpt

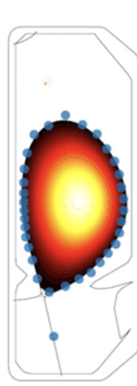
Learning Plasma Control for Fusion Science [2]



Droplets

Negative Triangularity



ITER-like shape

[2] Jonas Degrave et al. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *Nature* 602.7897 (2022), pp. 414–419

Contents

1. Introduction
2. Formalisation
3. Solving Reinforcement Learning problems
   - Tabular methods
   - Approximate methods – Deep Reinforcement Learning
4. Conclusion and open problems

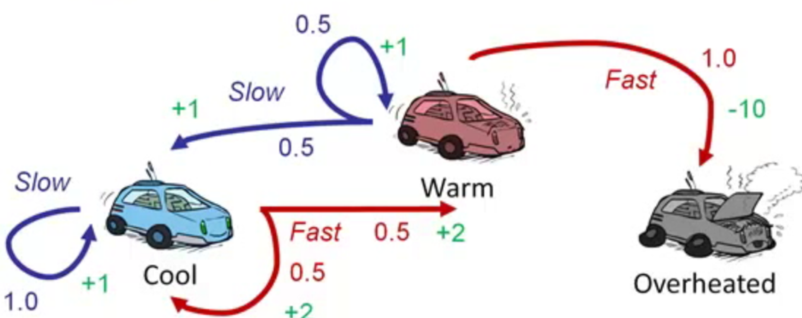
Reinforcement Learning: core concept

Reinforcement learning formalism includes:

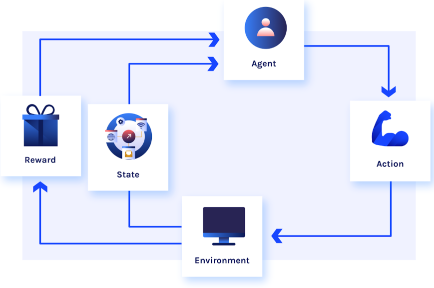
- **Environment**:
    - Typically *stochastic* and *unknow* but *stationary* (**Markov Decision Process**)
    - Environment dynamics (i.e., the *model*) expressed as: $p(s', r|s, a)$ ➔ not known by the agent
- **Reward signal**
    - Identifies what is *good* in the environment (the goal)
- **Agent**, which contain:
    - State
    - Policy
    - *Value* function estimation?

Environment: Stochasticity

- The environment is **stochastic** if the next state is *not* fully determined by the current state and action
- But the environment is **stationary** if the probability distribution of the next state is the same for all time steps



Agent and environment

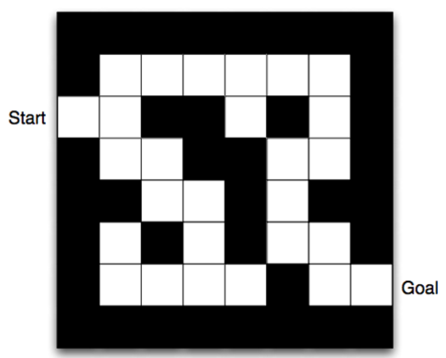


- Each time step t:
  - The agent receives an observation $s_t \in \mathcal{S}$ (and a reward $r_t \in \mathbb{R}$)
  - Executes an action $a_t \in \mathcal{A}$
- The environment
  - Receives the action a_t
  - Emits the observation s_{t+1} and the reward r_{t+1}
- The agent-environment interaction can be (i.e., the **task**):
  - *Episodic*: the agent-environment interaction breaks down into episodes (e.g., chess)
    - A sequence of actions that terminates in a terminal state
  - *Continuing*: the agent-environment interaction continues without limit (e.g., robot)

Agent state

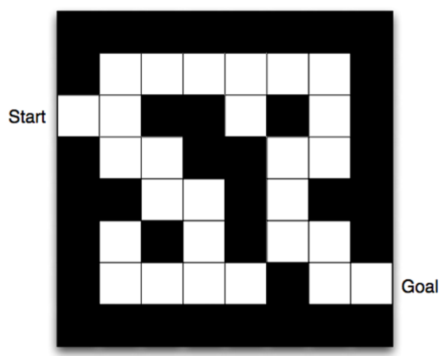
- A full episode is a sequence of state-action-reward tuples (called **trajectory**)
- Example: $\mathcal{H}_T = \{(s_0, a_0, r_1), (s_1, a_1, r_2), \ldots, (s_{T-1}, a_{T-1}, r_T)\}$
- Markovian property: the future is independent of the past given the present
  - Formula: $p(s_{t+1}|s_t, a_t) = p(s_{t+1}|\mathcal{H}_t, a_t)$
  - **NB!**: this means that the state s_t is a **sufficient statistic** of the future
- The environment state can be either:
  - **Fully observable**: the agent knows the full environment state
  - **Partially observable**: the agent partially observes environment state
- Today we will assume that the state is **fully observable** and **Markovian**
- Real case scenario: **partially observable** and **non-Markovian**
  - Also in that situation, reinforcement learning algorithms can be used (particularly the ones based on *deep learning*)

Example: Maze



- **Action**: move in one of the four directions (up, down, left, right)
- **State**: *???*
- **Reward**: *???*
- **Policy**: *???*

Example: Maze



- **Action**: move in one of the four directions (up, down, left, right)
- **State**: *position in the maze*
- **Reward**: *???*
- **Policy**: *???*

Rewards

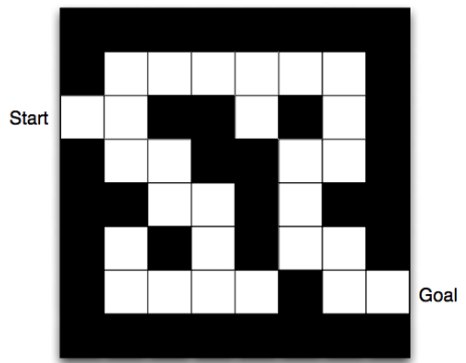
- A **reward** r_t is a scalar feedback signal
  - In chess, $r_t = 1$ if the agent wins, $r_t = 0$ otherwise
  - In a robot, $r_t = 1$ if the robot reaches the target, $r_t = 0$ otherwise
  - For a portfolio, r_t is the profit
- Describes how well the agent is doing at step t (*define the goal*)
- The agent's sole objective is to maximize the discounted cumulative reward (**return**)

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned}$$

Why discounted?

- Immediate rewards can be more important than future rewards
- $\gamma \in [0, 1]$ is the discount factor
- $\gamma = 0$ ➔ myopic agent
- $\gamma = 1$ ➔ far-sighted agent

Example: Maze

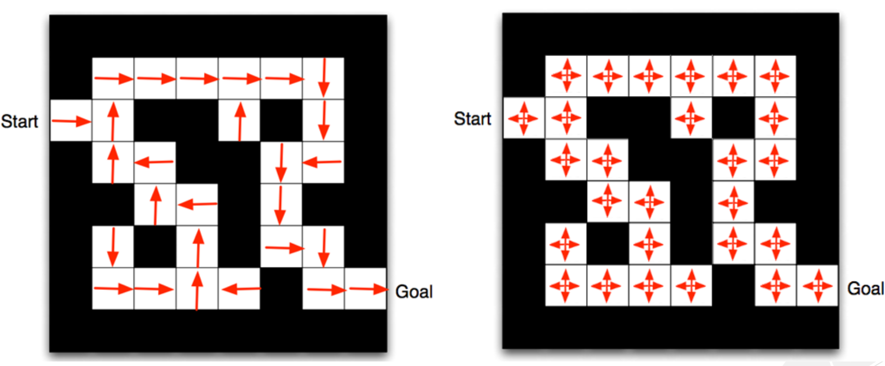


- **Action**: move in one of the four directions (up, down, left, right)
- **State**: *position in the maze*
- **Reward**:
  - r_t = *-1 for each step,* r_t = *0 if the agent reaches the target*
  - $r_t = 1$ *if the agent reaches the target,* $r_t = 0$ *otherwise*
- **Policy**: *???*

Agent Policy

- The agent's behaviour is determined by a **policy** π
- A policy is a mapping from state to action
- An action is something that affects the state
  - the action can be either:
    - **Discrete**: $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ (e.g., chess)
    - **Continuous**: $\mathcal{A} = \mathbb{R}^n$ (e.g., the torque applied to each joint of a robot)
- The policy can be either:
  - **Deterministic**: $\pi : \mathcal{S} \to \mathcal{A}$
  - **Stochastic**: $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$
- The policy is typically represented as a **lookup table** or a **neural network**
- The policy can be based on an **estimation** of the *value* function

Example: Maze



- **Action**: move in one of the four directions (up, down, left, right)
- **State**: *position in the maze*
- **Reward**:
  - r_t = -1 *for each step,* r_t = 0 *if the agent reaches the target*
  - $r_t = 1$ *if the agent reaches the target,* $r_t = 0$ *otherwise*
- **Policy**:
  - *shortest path to the target*
  - *random walk*

Value function

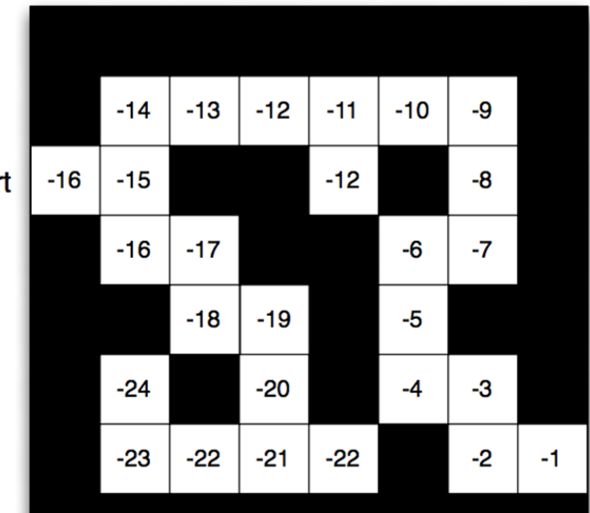
- The value function $v_\pi(s)$ gives the **long-term value** of state s under policy π
- The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state
- Formally, it can be expressed as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s\right]$$

- the state-action value function $q_\pi(s, a)$ gives the **long-term value** of state-action pair (s, a) under policy π
  - Formally, it can be expressed as:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

Example: Maze (value)



Bellman equation

- The return G_t can be computed recursively:

$$
\begin{aligned}
G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\
&= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \\
&= r_{t+1} + \gamma G_{t+1}
\end{aligned}
$$

- The value itself can be formulated recursively:
- This idea can be used for computing the optimal value function $v_*(s)$ (Bellman equation)

$$v^*(s) = max_a \mathbb{E}[r_{t+1} + \gamma * v_*(s_{t+1}) | S_t = s, A_t = a]$$

- Can be also defined for the state-action value function $q^*(s, a)$

$$q^*(s, a) = \mathbb{E}[r_{t+1} + \gamma * max_{a'} q^*(s_{t+1}, a') | S_t = s, A_t = a]$$

Optimal policy

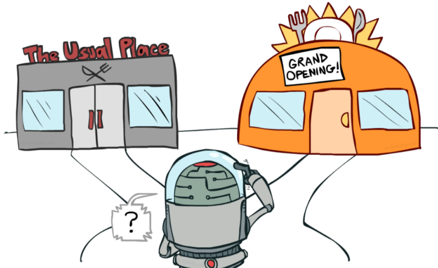
- The policy function can be defined on top of the value function (or the state-action value function)
- Greedy policy: $\pi_*(s) = argmax_a q_\pi(s, a)$
- Epsilon greedy policy: $\pi_*(s) = argmax_a q_\pi(s, a)$ with probability $1 - \epsilon$, otherwise a random action is selected
- How to compare policies??
  - A policy π is better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states
- The *optimal* policy is the one that maximizes the expected return
- Formally, it can be expressed as:

$$\pi_*(s) = argmax_a q_*(s, a)$$

- In RL, the policy is essential to explore the environment (*exploration*) while maximizing the reward (*(*exploitation))

Exploration vs Exploitation

- **Exploration**: finds more information about the environment
- **Exploitation**: exploits known information to maximize the reward
- In order to find the optimal policy, the agent must explore the environment as well as exploit the knowledge it has already acquired
- This is the **exploration-exploitation dilemma**
- The agent must find a good trade-off between exploration and exploitation (e.g., ϵ-greedy)



Recap – modelling perspective

Encode your application as a RL problem

- Identify the environment (i.e., the state space $\mathcal{S}$ of a given application)
- Identify the action space $\mathcal{A}$ of a given application (decisions that affect the state)
- Identify the task type
- Identify the reward function $r(s, a)$ (i.e., the goal of the application)

Recap – modelling perspective

Encode your application as a RL problem

- Identify the environment (i.e., the state space $\mathcal{S}$ of a given application)
- Identify the action space $\mathcal{A}$ of a given application (decisions that affect the state)
- Identify the task type
- Identify the reward function $r(s, a)$ (i.e., the goal of the application)

Find the optimal policy

- Find the optimal policy π_* that maximizes the expected return
- The optimal policy can be found by solving the Bellman equations ...
- ... but in practice it is not fleasible
    - The model is not known
    - Is computationally expensive

Contents

1 Introduction

2 Formalisation

3 Solving Reinforcement Learning problems
- Tabular methods
- Approximate methods – Deep Reinforcement Learning

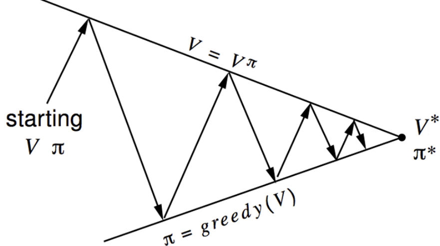
4 Conclusion and open problems

Contents

1. Introduction
2. Formalisation
3. Solving Reinforcement Learning problems
   - Tabular methods
   - Approximate methods – Deep Reinforcement Learning
4. Conclusion and open problems

Optimal policy – model based approach

- Simpliest approach: **policy iteration** (based on dynamic programming)
- Given a policy π:
  - at each iteration, $k+1$
  - For all states $s \in \mathcal{S}$:
  - Evaluate the policy π (i.e., compute $v_\pi(s) = \mathbb{E}[r_{t+1} + \gamma * t + 1 + ...|s_t = s]$)
  - Improve the policy π (i.e., compute $\pi'(s) = greedy(v_\pi)$)
- For each iteration, it is proven that the policy π' is better than or equal to the previous one
- it needs to be repeated until convergence
- This process *always* converges to the optimal policy (in Markov decisions processes)
- Dynamic programming algorithms are based on this idea



Model free approach – families

- **Value based**:
  - Compute the value function and then derive the optimal policy (Q-learning, SARSA, DQN)
- **Policy based**:
  - Compute directly the optimal policy (REINFORCE)
- *Actor critic*:
  - Have both a value function and a policy (A3C, PPO)

Model-free approach – Monte carlo

- True essence of reinforcement learning: **trial and error**
- Simulated experience is used to solve the problem
- **Monte Carlo** methods are based on averaging sample returns
- How to guarantee that all states are visited?

Monte carlo with exploring start (ES)

- Start an arbitrary π and q and repeat forever the following steps:
  - choose s_0 and a_0 such that all pairs have probability > 0
  - generate an episode following π (e.g., a simulation run)
  - for each pair s_t, a_t in the episode:
    - $G \leftarrow$ return following the first occurence of s_t, a_t
    - $q(s_t, a_t) \leftarrow average(G, q(s_t, a_t))$
    - $\pi(s_t) \leftarrow argmax_a q(s_t, a)$

Model-free approach – Monte carlo

- True essence of reinforcement learning: **trial and error**
- Simulated experience is used to solve the problem
- **Monte Carlo** methods are based on averaging sample returns
- How to guarantee that all states are visited?

Monte carlo with exploring start (ES)

- Start an arbitrary π and q and repeat forever the following steps:
  - choose s_0 and a_0 such that all pairs have probability > 0
  - generate an episode following π (e.g., a simulation run)
  - for each pair s_t, a_t in the episode:
    - $G \leftarrow$ return following the first occurence of s_t, a_t
    - $q(s_t, a_t) \leftarrow average(G, q(s_t, a_t))$
    - $\pi(s_t) \leftarrow argmax_a q(s_t, a)$

Monte carlo with without ES

- stick to an initial state s_0
- but be sure that all states will eventually be visited
- π should never give less than $\epsilon > 0$ probability of being selected
- e.g., can super-impose current π with a non-deterministic policy

Temporal difference (TD)

- A combination of Monte Carlo ideas and dynamic programming ideas
- Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics
- Like dynamic programming methods, TD methods update estimates based in part on other learned estimates, without waiting for an outcome (they *bootstrap*)
  - bootstrap: the value of a state is updated based on the estimated value of the next state

Example methods

- Q-Learning: updates q using next state and $\epsilon -$ *greedy* policy for current q (*off-policy*)
- SARSA: updates q using next state and $\epsilon -$ *greedy* policy for next q (*on-policy*)

Q-learning

- Generally considered a flexible, simple and effective method: typically the starting point

Q-learning

Algorithm core: Q-update

- $q(s_t, a_t) = (1 - \alpha) * q(s_t, a_t) + \alpha[r_{t+1} + \gamma * max_a \, q(s_{t+1}, a)]$
- α is the *learning rate* ($\alpha \in [0, 1]$)
- it is proven that this converges to the optimal q function (if α is sufficiently small and all state-action pairs are visited infinitely often)

Q-learning

Algorithm core: Q-update

- $q(s_t, a_t) = (1 - \alpha) * q(s_t, a_t) + \alpha[r_{t+\mathbf{1}} + \gamma * max_a\, q(s_{t+\mathbf{1}}, a)]$
- α is the *learning rate* ($\alpha \in [0, 1]$)
- it is proven that this converges to the optimal q function (if α is sufficiently small and all state-action pairs are visited infinitely often)

Full algorithm (episodic task)

- initialize Q arbitrarily and put 0 on terminal states
- for each episode:
- initialize s_0 (e.g., randomly) and $t = 0$
  - for each step t:
    1. choose a_t from s_t using a policy from q (e.g., $\epsilon - greedy$)
    2. take the action a_t and observe r_t, $s_{t+\mathbf{1}}$ (interaction with the environment)
    3. perform the update $q(s_t, a_t) = (1 - \alpha) * q(s_t, a_t) + \alpha[r_{t+\mathbf{1}} + \gamma * max_a\, q(s_{t+\mathbf{1}}, a)]$
    4. increase t
    5. end if $s_{t+\mathbf{1}}$ is terminal (or $t > T$)

Q-learning – pratical tips

Exploration-exploitation trade-off

- $\epsilon - greedy$ policy is a simple way to balance exploration and exploitation
- Typically, is better to start with a high ϵ and then decrease it over time
- This is called *ϵ-decay* and can be done in different ways
- Example (linear decay): $\epsilon = max(\epsilon_{min}, \epsilon_{max} - \frac{t}{\epsilon_{decay}})$
- Example (exponential decay): $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda * t}$

Q-learning – pratical tips

Exploration-exploitation trade-off

- $\epsilon - greedy$ policy is a simple way to balance exploration and exploitation
- Typically, is better to start with a high ϵ and then decrease it over time
- This is called *ϵ-decay* and can be done in different ways
- Example (linear decay): $\epsilon = max(\epsilon_{min}, \epsilon_{max} - \frac{t}{\epsilon_{decay}})$
- Example (exponential decay): $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda * t}$

Learning rate

- Learning rate α is typically set to a small value (e.g., 0.1)
- However, it can be useful to decrease it over time
- **Motivation**: the agent can learn faster in the beginning and then slow down the learning rate
- Example (linear decay): $\alpha = max(\alpha_{min}, \alpha_{max} - \frac{t}{\alpha_{decay}})$

Q-learning – Offline and online applications

Offline

- create a simulation of the selected environment (e.g., a trading simulator)
- simulates the agent-environment interaction learning the Q-function
- in production uses the learned Q-function to take decisions
- **cons**:
  - no adaptation to the real environment
  - The simulation must be a good approximation of the real environment

Q-learning – Offline and online applications

Offline

- create a simulation of the selected environment (e.g., a trading simulator)
- simulates the agent-environment interaction learning the Q-function
- in production uses the learned Q-function to take decisions
- **cons**:
  - no adaptation to the real environment
  - The simulation must be a good approximation of the real environment

Online

- learn the Q-function while interacting with the real environment
- implement your agent and let it learn while taking decisions
- pratical solution: initially learn fast and then slow down the learning rate
- **cons**:
  - the agent can take bad decisions while learning (e.g., the robot can fall)
  - the agent can take a lot of time to learn

Q-learning – programming perspective

- What is the best way to implement Q-learning? (or in general RL algorithms)
- Separation of concerns:
  - **Environment**: the environment is a black box that can be interacted with (e.g., a trading simulator)
    - Role: provide a clear interface to the agent in order to interact with the environment
    - Reference example: gymnasium: `https://gymnasium.farama.org/`
  - **Policy**: the policy is a function that maps states to actions
    - It could be a lookup table or a neural network
    - In the case of a neural network, there is a need of an autodifferentiation library
    - State-of-the-art libraries: **PyTorch** (`https://pytorch.org/`), **Tensorflow** (`https://www.tensorflow.org/`), and **TorchRL** (`https://github.com/pytorch/rl`)
  - **Learning algorithm**: the learning algorithm is responsible for learning the policy
    - It could be a simple algorithm (e.g., Q-learning) or a complex one (e.g., DQN)
    - Some libraries provide a set of algorithms: Stable Baselines (`https://stable-baselines3.readthedocs.io/en/master/`)

Hands-on at

`https://github.com/cric96/intro-reinforcement-learning-python`

Contents

1 Introduction

2 Formalisation

3 Solving Reinforcement Learning problems
- Tabular methods
- Approximate methods – Deep Reinforcement Learning

4 Conclusion and open problems

Reinforcement Learning Pitfalls: Large state space

Problem

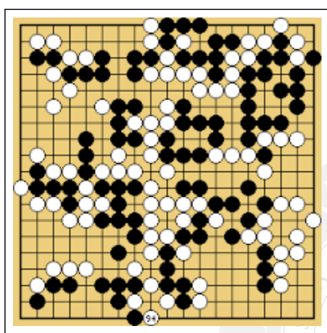
- **State space ➔** set of all possible states
- **State space explosion ➔** the number of states is too large to be stored in memory

Example (Go)

- 10^{170} possible states **(!!!!)**
- 10^{80} atoms in the universe
- 10^{16} seconds since the Big Bang

Example (Chess)

- 10^{46} possible states
- total space required $\sim 10^{35}$ terabytes



Question

How to deal with large state space?

Reinforcement Learning Pitfalls: Continous Action Space

Problem

- **Action space** ➔ set of all possible actions
- **Continous action space** ➔ the actions are real numbers (e.g. $[0, 1]$) ➔infinite number of actions

Example (Robotics)

- **Action space** ➔ the set of all possible joint angles
- **Continous action space** ➔ the set of all possible real joint angles

Question

How to deal with continuous action space?

Reinforcement Learning Pitfalls: Generalization

Problem

- **Generalization** ➔ the ability to perform well on previously unseen environments
- Can be also seen as **transfer learning** ➔ the ability to transfer knowledge from one environment to another
- **Generalization gap** ➔ the difference between the performance in the training environments and the performance in the test environments

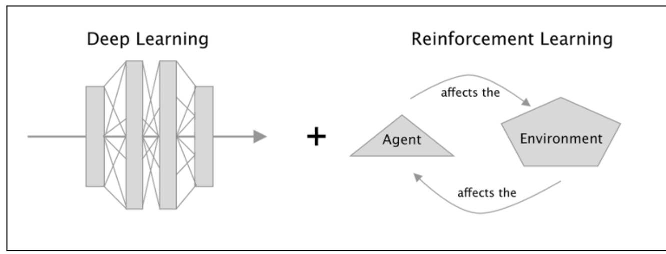
Example (Go)

- **Generalization** ➔ the ability to play well with different opponents
- **Generalization gap** ➔ the difference between the performance on the training set and the performance on the test set

Question

How to deal with generalization?

Deep Reinforcement Learning



Overview

- **Deep Reinforcement Learning (DRL)** ➔ the use of deep neural networks to approximate the value function/policy

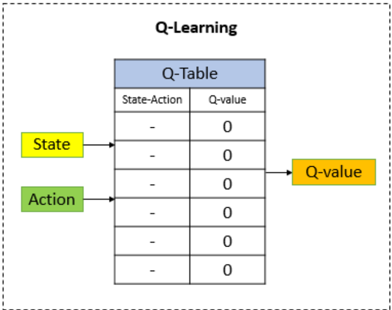
Key features

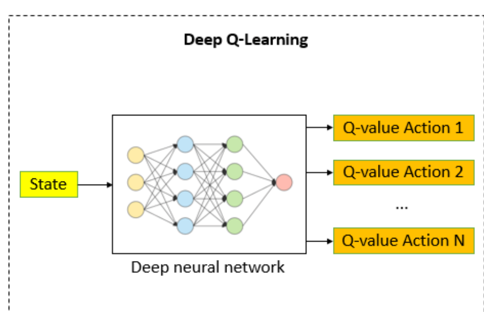
- **value function approximation** (instead of table) ➔ **handle large state space**
- **policy gradient** (instead of Q-Learning) ➔ **handle continous action space**
- **deep neural networks** ➔ **handle generalization** (Representation learning)

Deep Q-Learning

Q-Learning but q-function is approximated by a neural network

$$Q(s, a, \theta) \sim Q(s, a)$$





Deep Q-Learning

Loss function

- Bellman equation: $Q(s, a) = (r + \gamma \max_{a'} Q(s', a'))$
- Treating $r + \gamma \max_{a'} Q(s', a')$ as a target value
- Regression problem: $L(\theta) = (r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta))^2$

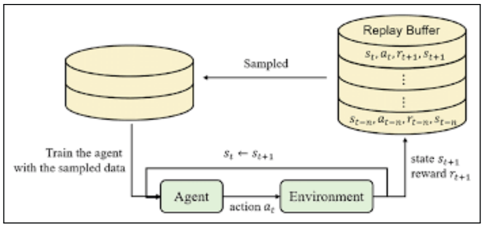
Issues

- **Correlation** ➔ the samples are not independent
- **Non-stationary** ➔ the target value changes over time

Solutions

- **Replay Buffer** ➔ store the transitions (s, a, r, s') and sample them randomly
- **Target Network** ➔ used to compute the target value

Deep Q Learning: Replay Buffer



How

- Store the transitions (s, a, r, s') in $\mathcal{D}$ of prior experience
- During Backpropagation, sample a batch of transitions (s, a, r, s')

Loss computation

- Sample a random batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute the target value $y = r + \gamma \max_{a'} Q(s', a', \theta)$
- Use the target value to compute the loss $L(\theta) = \mathbb{E}[(y - Q(s, a, \theta))^2]$

Deep Q Learning: Fixed Target Network

How

- Use a separate network to compute the target value
- The target network is updated every C step

Loss computation

- Let θ^- be the parameters of the target network
- Sample a random batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute the target value $y = r + \gamma \max_{a'} Q(s', a', \theta^-)$
- Use the target value to compute the loss $L(\theta) = \mathbb{E}[(y - Q(s, a, \theta))^2]$
- After C steps, update the target network parameters $\theta^- \leftarrow \theta$

Benefits

- **Stable** ➔ the target value is fixed for C steps, avoiding the non-stationary issue (dipendece on target and prediction cause)

Deep Q Learning: Epsilon decay

How

- ϵ is the probability of selecting a random action
- ϵ is decayed over time (or steps or episodes)
- **(!!!)** Off-policy nature of DQL ➔ the agent can learn from random actions

Why

- **Exploration vs Exploitation** ➔ the agent needs to explore the environment to learn the optimal policy
- **Exploitation** ➔ the agent needs to exploit the learned policy to maximize the reward

Deep Q Learning: Algorithm

Algorithm

- Initialize the replay buffer $\mathcal{D}$
- Initialize the target network parameters θ^-
- Initialize the Q-network parameters θ
- **for** episode = 1, M **do**
  - Initialize the initial state s_1
  - **for** t = 1, T **do**
    - With probability ϵ select a random action a_t
    - otherwise select $a_t = argmax_a Q(s_t, a, \theta)$
    - Execute action a_t in the environment and observe reward r_t and next state s_{t+1}
    - Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    - Sample a random minibatch of transitions (s, a, r, s') from $\mathcal{D}$
    - Set $y_i = r + \gamma \max_{a'} Q(s', a', \theta^-)$
    - Perform a gradient descent step on $(y_i - Q(s, a, \theta))^2$ with respect to the network parameters θ
    - Every C steps reset $\theta^- \leftarrow \theta$

Deep Q Learning: Extensions and Limits

Limits

- Works only for discrete action spaces
- Sample inefficiencient
- Overestimation of the action value due to the max operator

Extensions

- **Double DQN** → use two separate networks to select and evaluate the action
  - Pro: avoid overestimation of the action value
- **Prioritized Experience Replay** → sample the transitions from the replay buffer according to their TD-error
  - Pro: better exploration of the state space
- **Raindow DQN** → combination of the previous extensions

Policy gradient methods – REINFORCE

- **Policy gradient** ➔ the policy is directly optimized
  - **Pro**: can handle continous action space
  - **Pro**: can learn stochastic policies
  - **Pro**: sometimes policies are easier to learn than value functions
- **REINFORCE** is an policy gradient algorithm for maximizing the expected return $G = \sum_{t=0}^{T} \gamma^t r_t$
- *intuition*: trial and error
  - Sample a trajectory τ from the policy $\pi(\theta)$. If the trajectory is good, increase the probability of the actions. Otherwise, decrease the probability of the actions
  - It can be seen as stochasti gradient ascent on $G(H_T)$
- we want to train the policy in a way theta:

$$\theta_{n+1} = \theta_n + \alpha \nabla J(\theta_n)$$

- where $J(\theta) = \mathbb{E}_{\pi(\theta)}[G]$

Policy gradient methods – REINFORCE (cont.)

- The gradient can be estimated using the sample return G_t (came from the policy theorem):

$$\theta_{n+1} = \theta_n + \alpha \nabla log \pi(a|s) G_t$$

- The sample return G_t can be only computed at the end of the episode
- Therefore, this is a *episodic* algorithm with Monte Carlo updates

REINFORCE

- Initialize the policy parameters θ
- **for** episode = 1, M **do**
  - Generate an episode following $\pi(\theta)$: $s_1, a_1, r_1, ..., s_T, a_T, r_T$
  - $G = 0$
  - **for** $t = 1, T$ **do**
    - $G \leftarrow r_t + \gamma * G$
    - $\theta \leftarrow \theta + \alpha \nabla log \pi(a_t|s_t) G$

REINFORCE – Extension and Limitations

Limitations

- **High variance** ➔ the gradient is computed using the sample return G_t
- **Sample inefficiency** ➔ the policy is updated only at the end of the episode

Extension

- **Actor-critic** ➔ use a critic to estimate the value function
  - Pro: reduce the variance of the gradient
  - Pro: reduce the sample inefficiency
  - Cons: you need to train two networks (one for the actor and one for the critic)
- **Proximal Policy Optimization** ➔ use a surrogate objective function to avoid too large policy updates

Contents

1 Introduction

2 Formalisation

3 Solving Reinforcement Learning problems
- Tabular methods
- Approximate methods – Deep Reinforcement Learning

4 Conclusion and open problems

Open problems I

- Reinforcement Learning is a powerful tool to solve sequential decision-making problems . . .
- . . . but still there are some open-problems

Transfer Learning

- **Problem**: the agent needs to learn a new task from scratch
- **Solution**: transfer knowledge from a previous task
- Unfortunately, it is not easy to transfer knowledge from one task to another
- **Open problem**: how to transfer knowledge from one task to another?

Sample efficency

- **Problem**: the agent needs a lot of samples to learn a good policy
- The human brain can learn a new task with few samples, why RL agents cannot?
- **Open problem**: how to explore the environment efficiently?
- Ideas: curiosity, intrinsic motivation, long-time versus short-time learning learning

Open problems II

Safe exploration

- **Problem**: the agent needs to explore the environment to learn a good policy without taking bad decisions
- **Open problem**: how to explore the environment safely?

Multi-agent RL

- **Problem**: the agent needs to learn in a multi-agent environment
- **Open problem**: how to learn in a multi-agent environment?
- Some issues are: credit assignment, non-stationarity, exploration
- No foundational theory for multi-agent RL (even worst for many agents)

Continous adaptation

- **Problem**: the agent needs to adapt to a changing environment
- **Open problem**: how to adapt to a changing environment?
- Ideas: meta-learning, continual learning, online learning

Conclusion

- Reinforcement Learning is a tool really used in practice
  - **AlphaGo** ➔ DeepMind (2016)
  - **ChatGPT** ➔ OpenAI (2023)
- In this lecture we have seen:
  - Simple formulation of the RL problem
  - Tabular methods (e.g., Q-learning)
  - Approximate methods (e.g., DQL)
  - Policy gradient methods (e.g., REINFORCE)
  - hands-on with gymnasium and PyTorch
- This is only the tip of the iceberg
  - Actor-Critic methods ➔ A3C, A2C, PPO
  - Multi-agent RL ➔ MADDPG, QMIX, MAPPO
  - Hierarchical RL ➔ Healthcare

Reinforcement Learning

Unlocking the Power of AI Agents

Gianluca Aguzzi gianluca.aguzzi@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Talk @ Advanced School in Artificial Intelligence (ASAI)

26/07/2023