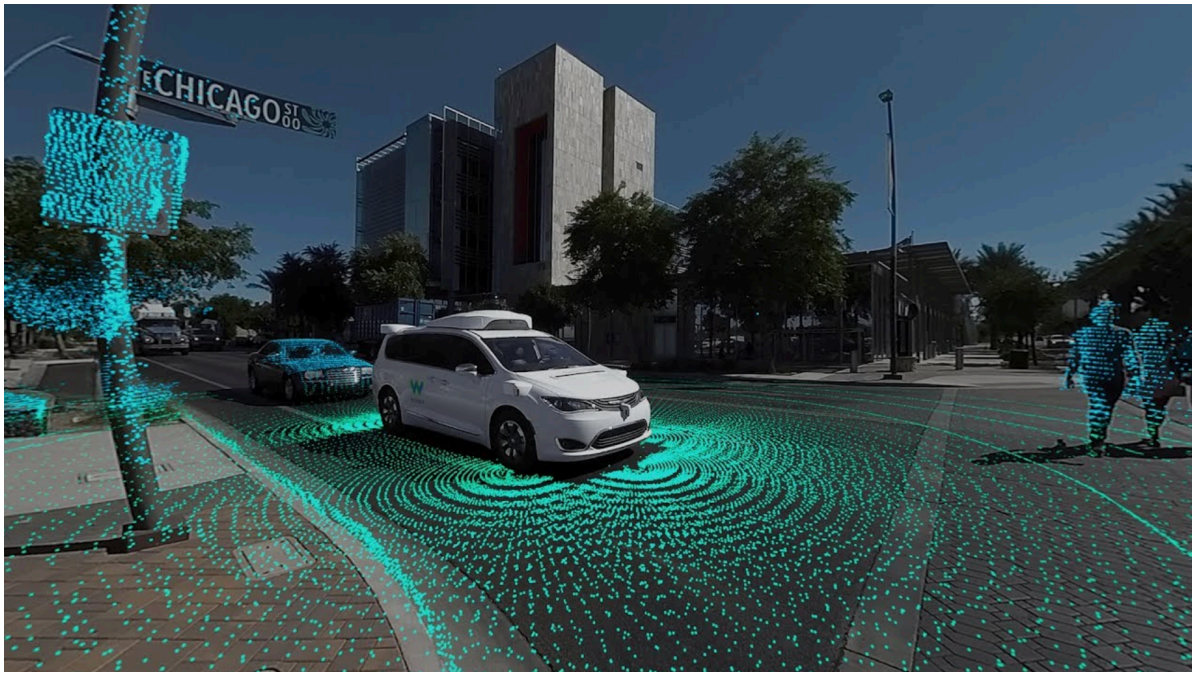
# Autonomous and Adaptive Systems

Alessia Crimaldi

2025-02-28

# Table of contents

# Preface




Official course page: Musolesi (2024-25).

## Overview

This course will provide the students with a solid understanding of the state of the art and the key conceptual and practical aspects of the design, implementation and evaluation of intelligent machines and autonomous systems that learn by interacting with their environment. The course also takes into consideration ethical, societal and philosophical aspects related to these technologies.

Strong focus on the state-of-the-art (and what's next): read papers from leading AI/ML conferences (but also classics from the ML field), try software, etc.

## Topics of the Module

- Introduction to the design of adaptive and autonomous systems: intelligent agents and intelligent machines, automatic vs autonomous decision-making.
- Introduction to Reinforcement Learning (RL): multi-armed bandits, Montecarlo methods, tabular methods, approximation function methods, and policy-based methods.
- Applications of RL to games, classic control theory problems and robotics.
- Introduction to algorithmic game theory for multi-agent learning systems: cooperation and coordination, social dilemmas, and Multi-Agent Reinforcement Learning.
- Bio-inspired adaptive systems.
- Intelligent machines that create: Generative Learning and AI creativity.
- The "brave new world": agentic AI systems, design of agents based on foundational models, training using Reinforcement Learning from human-feedback (RLHF) and Direct Preference Optimization (DPO).
- Open problems and the future: safety, value alignment, super-intelligence, controllability, and self-awareness.

## Interesting News and Readings

- The Dartmouth Summer Research Project on Artificial Intelligence: Solomonoff (06 May 2023).
- Andrew Barto and Richard Sutton Recognized as Pioneers of Reinforcement Learning: Ormond (2024).
- To catch up with Probability and Computing: Mitzenmacher (2017).

## Papers' sources

- Google Scholar (scholar.google.com)

# 1 Intelligent Agents and Machines

The Brave New World.

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \text{ gridworld}$$

# 2 Introduction to Reinforcement Learning

- **Key idea:** a natural way of thinking about learning is learning through interaction with the external world.
- Learning from interation is a foundational idea underlying nearly all theories of learning and intelligence.
- Reinforcement learning is learning what to do - how to map situations to actions - so as to maximise a numerical reward.
  - ***Goal-directed*** learning from interaction.
- The learner is not told which actions to take, but instead it must discover which actions yield the most reward by trying them.

## 2.1 Examples of Problems

## 2.2 Finite Markov Decision Processes

**Markov Decision Processes** (**MDPs**) are a mathematically idealised formulation of Reinforcement Learning for which precise theoretical statements can be made.

- Tension between breadth of applicability and mathematical tractability.
- MDPs provide a way for framing the problem of learning from experience, and, more specifically, from interacting with an environment.

Definitions

Reinforcement Learning, or RL for short, is a unique facet of machine learning where an agent learns to make decisions through trial and error.

Two main entities:

1. **Agent** = learner and decision-maker.
   - Interacts with the environment selecting ***actions***.
   - ***Observes*** and acts within the environment.

- Receives:
  - ***rewards*** for good decisions.
  - ***penalties*** for bad decisions.

2. **Environment** = everything else outside the agent.
   - Changes following actions of the agent.

- **Goal**: devise a strategy that maximises the total reward over time .

RL framework

- **Agent**: learner, decision-maker.
- **Environment**: challenges to be solved.
- **State**: environment snapshot at given time.
- **Action**: agent's choice in response to state.
- **Reward**: feedback for agent action (positive or negative).

RL interaction loop

- The agent and the environment interact at each discrete step of a sequence $t = 0, 1, 2, 3, ...$
- At each time step $t$, the agent receives some representation of the environment **state** $S_t \in \mathcal{S}$ where $\mathcal{S}$ is the set of the states.
- On that basis, an agent selects an **action** $A_t \in \mathcal{A}(S_t)$ where $\mathcal{A}(S_t)$ is the set of the actions that can be taken in state $S_t$.
- At time $t + 1$, as a consequence of its action, the agent receives a **reward** $R_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is the set of rewards (expressed as real numbers).

Let's demonstrate the agent-environment interaction using a generic code example. The process starts by creating an environment and retrieving the initial state. The agent then enters a loop where it selects an action based on the current state in each iteration. After executing the action, the environment provides feedback in the form of a new state and a reward. Finally, the agent updates its knowledge based on the state, action, and reward it received.

```
env = create_environment()
state = env_get_initial_state()

for i in range(n_iterations):
    action = choose_action(state)
    state, reward = env_execute(action)
    update_knowledge(state, action, reward)
```

## 2.3 Goals and Rewards

- The goal of the agent is formalised in terms of the reward it receives.
- At each time step, the reward is a simple number $R_t \in \mathbb{R}$.
- Informally, the agent's goal is to maximise the total amount it receives.
- The agent should not maximise the immediate reward, but the **cumulative reward**.

The Reward Hypothesis

We can formalise the goal of an agent by stating the "reward hypothesis":

> All of what we mean by goals and purposes can be well thought of as the maximisation of the exprected value of the cumulative sum of a received scalar signal (reward).

Expected Returns

In RL, actions carry long-term consequences, impacting both immediate and future rewards. The agent's goal goes beyond maximizing immediate gains; it receives a sequence of rewards and it strives to accumulate the highest *total reward over time*. This leads us to a key concept in RL: the **expected return**.

> The *expected return* $G_t$ is a function of the reward sequence $R_{t+1}, R_{t+2}, R_{t+3}, ...$
>
> > $G_t$ is the sum of all rewards the agent expects to accumulate throughout its journey.

Accordingly, the agent learns to anticipate the sequence of actions that will yield the highest possible return.

Episodic Tasks and Continuing Tasks

In RL, we encounter two types of tasks: *episodic* and *continuous*.

**Episodic tasks** are those in which we can identify a final step of the sequence of rewards, i.e. in which the interaction between the agent and the environment can be broken into subsequences that we call ***episodes*** (such as play of a game, rpeeated tasks, etc.). For example, in a chess game played by an agent, each game constitutes an episode; once a game concludes, the environment resets for the next one.

> An *episodic task* is divided into distinct episodes, each with a defined beginning and end.
>
> > Each *episode* ends in terminal state after $T$ steps, followed by a reset to a standard starting state or to a sample of a distribution of starting states.

The *next episode* is completely independent from the previous one.

On the other hand, **continuing tasks** involve ***ongoing interaction*** without distinct episodes (e.g. ongoing process control or robots with a long-lifespan). A typical example is an agent continuously adjusting traffic lights in a city to optimize flow.

A *continuing task* is one in which it is not possible to identify a final state.

Expected Return for Episodic Tasks and Continuing Tasks

In the case of ***episodic tasks*** the expected return associated to the selection of an action $A_t$ is the sum of rewards defined as follows:

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

In the case of ***continuing tasks*** the expected return associated to the selection of an action $A_t$ is defined as follows:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $\gamma$ is the discount rate, with $0 \leq \gamma \leq 1$.

Discounting Rewards

Immediate rewards are typically valued more than future ones, leading to the concept of **discounted return**. This concept prioritizes more recent rewards by multiplying each reward by a discount factor, gamma, raised to the power of its respective time step. For example, for expected rewards r1 through rn, the discounted return would be calculated as r1 + gamma * r2 + gamma^2 * r3, and so on.

The **discount factor** $\gamma$, ranging between 0 and 1, is crucial for balancing immediate and long-term rewards. A lower gamma value leads the agent to prioritize immediate gains, while a higher value emphasizes long-term benefits. At the extremes, a gamma of zero means the agent focuses solely on immediate rewards, while a gamma of one considers future rewards as equally important, applying no discount.

Numerical example:

In this example, we'll demonstrate how to calculate the discounted_return from an array of `expected_rewards`. We define a `discount_factor` of 0.9, then create an array of discounts, where each element corresponds to the discount factor raised to the power of the reward's position in the sequence.

```
import numpy as np
expected_rewards = np.array([1, 6, 3])
discount_factor = 0.9
discounts = np.array([discount_factor ** i for i in range(len(expected_rewards))])
print(f"Discounts: {discounts}")
```

```
Discounts: [1.   0.9  0.81]
```

As we can see, discounts decrease over time, giving less importance to future rewards. Next, we multiply each reward by its corresponding discount and sum the results to compute the `discounted_return`, which is 8.83 in this example.

```
discounted_return = np.sum(expected_rewards * discounts)
print(f"The discounted return is {discounted_return}")
```

```
The discounted return is 8.83
```

Relation between Returns at Successive Time Steps

Returns at successive time steps are related to each others as follows:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots \quad = R_{t+1} + \gamma(R_{t+1} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \ldots) \quad = R_{t+1} + \gamma G_{t+1}$$

## 2.4 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating value functions, i.e., functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state).

A policy is used to model the behaviour of the agent based on the previous experience and the rewards (and consequently the expected returns) an agent received in the past.

Policy

Formally, a *policy* is a mapping from states to probabilities of each possible action, i.e. policy $\pi$ is a *probability distribution*.

If the agent is following policy $\pi$ at time $t$, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

State-Value Function

The value function of a state $s$ under a policy $\pi$, denoted $v_\pi(s)$, is the expected return when starting in $s$ and following $\pi$ thereafter.

For MDPs, we can define the *state-value function* $v_\pi$ for policy $\pi$ formally as:

$$v_s \doteq E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$$

for all $s \in \mathcal{S}$.

where $E_\pi[.]$ denotes the expected value of a random variable, given that the agent follows $\pi$ and $t$ is any time step. The value of the terminal state is 0.

```
choose = rand(0.1)
if choose >= 0.7:
    move left
else:
    move right
```

Action-Value Function

## 2.5 Choosing the Rewards

Marvin Minsky. Steps Toward Artificial Intelligence. Proceedings of the IRE. Volume 49. Issue 1. January 1961.

Examples of Rewards

## 2.6 Estimating Value Functions

Monte-Carlo Methods

Using function approximators (neural networks and deep RL)

## 2.7 Optimal Policies and Optimal Value Functions

Definition of Optimal Policy

Optimal Value Functions

## 2.8 Difference between RL and other ML sub-domains

RL differs significantly from other types of machine learning, such as supervised and unsupervised learning.

In ***supervised learning***, models are trained using labeled data, learning to predict outcomes based on examples. It is suitable for solving problems like classification and regression.

***Unsupervised learning***, on the other hand, involves learning to identify patterns or structures from unlabeled data. It is suitable for solving problems like clustering or association analysis.

***Reinforcement Learning***, distinct from both, does not use any training data, and learns through trial and error to perform actions that maximize the reward, making it ideal for decision-making tasks.

RL vs. Supervised Learning

*Supervised learning* is learning from a set of labeled examples and, in interactive problems, it is hard to obtain labels in the first place. Therefore, in "unknown" situations, agents have to learn from their experience. In these situations *Reinforcement learning* is most beneficial.

RL vs. Unsupervised Learning

*Unsupervised learning* is learning from datasets containing unlabelled data. Since RL does not rely on examples (labels) of correct behaviour and instead explored and learns it, we may think that RL is a type of unsupervised learning. However, this is not the case because in *Reinforcement Learning* the goal is to maximise a reward signal instead of trying to find a hidden structure.

For this reason, *Reinforcement Learning* is usually considred a third paradigm in addition to supervised and unsupervised learning.

## 2.9 When to use RL

In particular, RL is well-suited for scenarios that require training a model to make sequential decisions where each decision influences future observations. In this setting, the agent learns through rewards and penalties. These guide it towards developing more effective strategies without any kind of direct supervision.

- Sequential decision-making
  - Decisions influence future observations
- Learning through rewards and penalties

- No direct supervision

Appropriate vs. Inappropriate for RL

An appropriate example for RL is **playing video games**, where the player needs to make sequential decisions such as jumping over obstacles or avoiding enemies. The player learns and improves by trial and error, receiving points for successful actions and losing lives for mistakes. The goal is to maximize the score by learning the best strategies to overcome the game's challenges.

- Player makes sequential decisions.
- Receives points and loses lives depending on actions.

Conversely, RL is unsuitable for tasks such as **in-game object recognition**, where the objective is to identify and classify elements like characters or items in a video frame. This task does not involve sequential decision-making or interaction with an environment. Instead, supervised learning, which employs labeled data to train models in recognizing and categorizing objects, proves more effective for this purpose.

- No sequential decision-making.
- No interaction with an environment.

## 2.10 RL Applications

Beyond its well-known use in gaming, RL has a myriad of applications across various sectors.

1. **Robotics.** In robotics, RL is pivotal for teaching robots tasks through trial and error, like walking or object manipulation.

    - Robot walking
    - Object manipulation

2. **Finance.** The finance industry leverages RL for optimizing trading and investment strategies to maximize profit.

    - Optimizing trading and investment
    - Maximise profit

3. **Autonomous Vehicles.** RL is also instrumental in advancing autonomous vehicle technology, enhancing the safety and efficiency of self-driving cars, and minimizing accident risks.

    - Enhancing safety and efficiency
    - Minimising accident risks

4. **Chatbot development.** Additionally, RL is revolutionizing the way chatbots learn, enhancing their conversational skills. This leads to more accurate responses over time, thereby improving user experiences.

    - Enhancing conversational skills
    - Improving user experiences

## 2.11 References

The notation and definitions are taken (with small variations) from:

- Richard S. Sutton (2018)

# 3 Multi-armed Bandits

- In the case of *multi-armed bandits*, no state is used for the selection of the next actions.
- It exists an "intermediate case" where the state is used, but the state itself does not depend on the previous actions, called *contextual bandits*.
- Multi-armed bandits are a simplified version of *reinforcement learning*.

| Multi-Armed Bandits | Contextual Bandits | Reinforcement Learning |
|---|---|---|
| No state is used | State is usedState does not depend on previous actions. The action that is selected does not affect the next state/situation. | State is usedState depends on previous actions. The action that is selected affects the next state/situation. |

## 3.1 K-armed Bandit Problem

Problem Definition

- We can model a k-armed problem as follows:
  - $k$ different actions;
  - after each choice we receive a numerical value that depends only on the action we selected;
  - the goal is to maximise the expected reward over a certain number of time steps.

This is the "classic" k-armed bandit problem, which is named by analogy to a **slot machine** or ***"one-armed" bandit***, except it has $k$ levers instead than one:

like in a slot machine, we need to learn which lever provides us with the highest reward.

The problem is the same, maximising the expected reward:

$$
\begin{aligned}
&E[G_t|S_t = s, A_t = a] \\
&\quad \doteq E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... |S_t = s, A_t = a] \\
&\quad = E[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]
\end{aligned}
$$

In the case of the k-armed bandit problem, the state is always the same (or, in other words, it does not matter). We can think about having $s = \overline{s}$ with $\overline{s}$ constant.

Considering a constant state $\overline{s}$, the problem is equal to maximise:

$$
\begin{aligned}
&E[G_t|S_t = \overline{s}, A_t = a] \\
&\quad \doteq E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... |S_t = \overline{s}, A_t = a] \\
&\quad = E[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = \overline{s}, A_t = a]
\end{aligned}
$$

with $a$ one of the $k$ actions.

Action Selection and Estimated Value

In a k-armed bandit problem, each of the $k$ actions has a value, equal to the expected or mean reward given that that action is selected.

- As in the general RL case, we denote the action selected on time step $t$ as $A_t$ and the corresponding reward as $R_t$.

  The ***value of an arbitrary action*** $a$ is the expected reward given that $a$ is selected:

  $$q_*(a) \doteq E[R_t|A_t = a]$$

In the trivial case, since we know the value of each action, solving the k-armed bandit problem is very easy: it is sufficient to always select the action with the highest value!

- However, in general, we do not know the action values with certainty, we only know ***estimates***.

  The ***estimated value of action*** $a$ ***at time step*** $t$:

  $$Q_t(a)$$

  Ideally, we would like to have that the value of $Q_t(a)$ would be very close to $q_*(a)$.

## 3.2 Exploration vs. Exploitation

We maintain the estimates of each action value.

***Exploitation***:

- At any step there is at least one action whose estimated value is the greatest $\rightarrow$ we refer to this action as ***greedy action***.
- When we select one of these actions, we are *exploiting* our current knowledge of the values of the actions.

***Exploration***:

- If we select non-greedy actions, we say that we are *exploring* $\rightarrow$ ***alternative actions***.
- By doing so, we can improve our estimation of the value functions of the non-greedy actions.

Balancing exploration and exploitation in a smart way is the key aspect: Several theoretical resuslts in terms of bounds, etc. given specific assumptions.

## 3.3 Evaluating Action-Value Methods

- Multi-armed bandits are a very good way of approaching the general problem of Reinforcement Learning.

  Simplification: the state does not change, which means:

  - Our actions *do not* modify the state;
  - Since the state does not change, the agent's actions *will not* depend on the previous actions (the two things are strictly linked).

  We will consider later the "full" reinforcement learning problem where the agent's actions *do* modify the state and the agent's action *do* depend on the previous actions.

- Recall: the *true value* of an action is the mean reward when that action is selected.

  A possible way to estimate this is by averaging the rewards actually received:

$$Q_t(a) \doteq \frac{\text{sum of rewards when an action } a \text{ is taken prior time } t}{\text{number of times an action } a \text{ is taken prior time } t}$$

$$= \frac{\sum_{i=1}^{t-1} R_i 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$$

where 1 denotes the random variable that is 1 if the predicate $A_i = a$ is true and 0 if not.

If the denominator is 0, we set $Q_t(a)$ to some default value (e.g., 0).

As the denominator goes to infinity, by the *law of large numbers*, $Q_t(a)$ converges to $q_*(a)$.

This is called the ***sample-average method*** because each estimate is the average of the sample of the rewards.

## 3.4 Greedy Action Selection Rule

## 3.5 $\epsilon$-Greedy Selection Rule

## 3.6 Incremental Implementation

Tracking a Stationary Problem

Let's now discuss how to implement these methods in practice. We consider a single action $a$ to simplify the notation.

- Let $R_i(a)$ the reward received after the $i$-th selection of the action $a$.
- Let $Q_n(a)$ denote the estimate of its action value after it has been selected $n-1$ times.

  We can write:

  $Q_n(a) \doteq \frac{R_1(a)+R_2(a)+\cdots+R_{n-1}(a)}{n-1}$

Trivial impementation

The trivial implementation would be to maintain a record of all the rewards and then execute the formula when that value would be needed. However, if this is done, the computational requirements would grow over time. A possible alternative is an *incremental implementation*.

Efficient computation of estimates

***Incremental implementation***:

$$
\begin{aligned}
Q_{n+1} &= \tfrac{1}{n}\sum_{i=1}^{n} R_i \\
&= \tfrac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i) \\
&= \tfrac{1}{n}(R_n + (n-1)\tfrac{1}{n-1}\sum_{i=1}^{n-1} R_i) \\
&= \tfrac{1}{n}(R_n + (n-1)Q_n) \\
&= \tfrac{1}{n}(R_n + nQ_n - Q_n) \\
&= Q_n + \tfrac{1}{n}(R_n - Q_n)
\end{aligned}
$$

This form of **update rule** is quite common in reinforcement learning.

The ***general form*** is:

NewEstimate $\leftarrow$ OldEstimate + StepSize *(Target – OldEstimate)*

where the expression *(Target – OldEstimate)* is usually defined as *error* in the estimate.

Tracking a Nonstationary Problem

The averaging method discussed before is appropriate for *stationary* bandit problems.

However, many problems are *non-stationary*. In these cases, it makes sense to give more weight to recent rewards rather than long-past rewards.

One of the most popular way of doing this is to use constant step-size parameter (in the example above was $\frac{1}{n}$).

The ***incremental update rule*** for updating an average $Q_n$ of the $n-1$ past rewards is modified to be:

$$Q_{n+1} \doteq Q_n + \alpha(R_n - Q_n)$$

where the step-size parameter $\alpha \in (0, 1]$ is constant.

This results in $Q_{n+1}$ being a weighted average of past rewards and the initial estimate.

More formally:

$$
\begin{aligned}
Q_{n+1} &\doteq Q_n + \alpha(R_n + Q_n) \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{(n-i)} R_i
\end{aligned}
$$

- This is called a ***weighted average*** since:

$$(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{(n-i)} = 1$$

- The quantity $1-\alpha$ is less than 1 and the weight given to $R_i$ decreases as the number of rewards increases: actually it's exponential, and for this reason it is sometimes called ***exponential-recency-weighted average***.

## 3.7 Optimistic Initial Values

Example

## 3.8 Upper-Confidence-Bound (UCB) Action Selection

## 3.9 Multi-armed Bandits in practice

```
import numpy as np
import random

class MultiArmedBandit:
    '''Create a k-armed bandit.

    Each bandit arm must have its reward distribution.

    The bandit has to receive the action: to select one of the k bandits,
    with possible value from 1 to 10.

    The bandit must have k random generators of two values i,j (with i < j):
    the min and max of the reward distribution.
    e.g.:
        1 3 -> 1.5, ...
        2 6 -> 2.1, 3.3, ...
        4 8 -> 4.2, 5.3, 7.8
    '''
    def __init__(self, k):
        self.arms = [self._create_generator() for i in range(k)]
        for i, arm in enumerate(self.arms):
            print(f"arm {i} range: {arm}")
```

```
    def _create_generator(self):
        lbound, ubound = 1, 10

        a = random.randint(lbound, ubound)
        b = random.randint(a, ubound)

        return a, b

    def get_reward(self, a: int):
        assert a >= 0 and a < len(self.arms)
        return random.uniform(*self.arms[a])

class ArmChooser:
    def __init__(self, k: int, eps: float, initial_value: float):
        self.Q, self.N = [initial_value] * k, [0.] * k
        self.eps = eps

    def _epsilon_greedy(self):
        if random.random() < 1 - self.eps:
            return np.argmax(self.Q)
        else:
            return random.randint(0, len(self.Q) - 1)

    def choose(self):
        a = self._epsilon_greedy()
        R = bandit.get_reward(a)
        print(f"R: {R}")
        self.N[a] += 1
        self.Q[a] += 1/self.N[a] * (R - self.Q[a])
        print(f"Q: {self.Q}")
        print (f"N: {self.N}")

# Example usage
k, T, eps, initial_value = 4, 10, 0.1, 10.
bandit = MultiArmedBandit(k)
chooser = ArmChooser(k, eps, initial_value)

for t in range(T):
    print(f"step {t}")
    chooser.choose()
```

```
arm 0 range: (10, 10)
arm 1 range: (3, 6)
arm 2 range: (7, 8)
arm 3 range: (10, 10)
step 0
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [0.0, 0.0, 0.0, 1.0]
step 1
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [1.0, 0.0, 0.0, 1.0]
step 2
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [2.0, 0.0, 0.0, 1.0]
step 3
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [3.0, 0.0, 0.0, 1.0]
step 4
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [4.0, 0.0, 0.0, 1.0]
step 5
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [5.0, 0.0, 0.0, 1.0]
step 6
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [6.0, 0.0, 0.0, 1.0]
step 7
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [7.0, 0.0, 0.0, 1.0]
step 8
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
N: [8.0, 0.0, 0.0, 1.0]
step 9
R: 10.0
Q: [10.0, 10.0, 10.0, 10.0]
```

```
N: [9.0, 0.0, 0.0, 1.0]
```

## 3.10 Contextual bandits

Policy-Based Action Selection

Associative Search

Exercise:

Website serving ads with contextual bandits.

> Design a system serving ads on a website (e.g. Google Ads) using contextual bandits.

Additional Readings

Lihong Li, Wei Chu, John Langford, Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. Proceedings of the 19th International Conference on World Wide Web (WWW 2010): 661–670.

## 3.11 Exercise

Website serving ads with bandits.

> Design a system serving ads on a website (e.g. Google Ads) only using bandits.

```python
import random

def shuffle_array(arr):
    random.shuffle(arr)

def get_random_int(min_val, max_val):
    return random.randint(min_val, max_val - 1)

def get_favorite_category(Q):
    return max(Q, key=Q.get)

def choose_next_ad(Q, ads, eps):
    if random.random() > eps:
        favorite_category = get_favorite_category(Q)
        category_ads = [ad for ad in ads if ad[1] == favorite_category]
    else:
```

```
        random_category = random.choice(list(Q.keys()))
        category_ads = [ad for ad in ads if ad[1] == random_category]

    return random.choice(category_ads) if category_ads else None

def display_ads(ad):
    print(f"Showing Ad: {ad[0]} (Category: {ad[1]})")
```

```
ads = [
    ("Soccer ", "Sport"), ("Basket ", "Sport"), ("Volleyball ", "Sport"),
    ("Banana ", "Fruit"), ("Apple ", "Fruit"), ("Orange ", "Fruit"),
    ("Cat ", "Pet"), ("Dog ", "Pet"), ("Parrot ", "Pet"),
    ("Sneakers ", "Fashion"), ("Sunglasses ", "Fashion"), ("Dress ", "Fashion"),
    ("Paris ", "Travel"), ("Tokyo ", "Travel"), ("London ", "Travel")
]

shuffle_array(ads)

Q = {category: 0 for _, category in ads}

eps = 0.2

display_ads(ads[0])
while True:
    user_input = input("Did you like this ad? (y/skip): ").strip().lower()
    if user_input == 'y':
        Q[ads[0][1]] += 1
    elif user_input == 'skip':
        Q[ads[0][1]] -= 1

    next_ad = choose_next_ad(Q, ads, eps)
    if next_ad:
        ads[0] = next_ad
        display_ads(next_ad)
    else:
        print("No more ads to display.")
        break
```

## 3.12 References

Books for the bandit:

- **Chapter 2** of Richard S. Sutton (2018)
- Tor Lattimore (2020)

# 4 Temporal Difference Methods

## 4.1 Temporal-Difference Learning

- **Temporal-difference (TD) methods**:
  - Like Monte Carlo methods, TD methods *can learn directly from experience*.
  - Unlike Monte Carlo methods, TD methods *update estimates based in part on other learned estimates*, without waiting for the final outcome (we say that "they *bootstrap*").
- **Prediction vs. Control Problems in TD learning**:
  1. We will consider the problem of prediction (**TD prediction**) first (i.e., we fix a policy $\pi$ and we try to estimate the value $v_\pi$ for that given policy).
  2. Then we will consider the problem of finding an optimal policy (**TD control**).

## 4.2 Review/Preliminaries

## 4.3 TD Prediction

Monte Carlo vs. TD for Prediction

TD(0) Algorithm and Updates

TD Error and Interpretation

Advantages of TD Methods

## 4.4 Theoretical Basis of TD(0)

Convergence Properties

Comparison with Dynamic Programming and Monte Carlo

## 4.5 On-Policy and Off-Policy Control

## 4.6 Sarsa: On-Policy TD Control

Transition from State-Action Pairs

Sarsa Algortihm and Updates

Online TD(0) Control

## 4.7 Q-Learning: Off-Policy TD Control

Q-Learning Algorithm and Updates

Convergence Properties

## 4.8 Summary

Recap of TD Methods

Function Approximation in RL

## 4.9 Exercise

Gridworld.

> Solve the gridworld problem with Sarsa.
>
> REINFROCEjs - GridWorld: TD

```
import numpy as np
import random
import copy

class GridWorld:
    def __init__(self, size):
        self.size = size
        self.agentPos = [0, 0]

        grid = np.full((size, size), '0', dtype=str)
```

```
        grid[0, 0] = ' '
        grid[size-1, size-1] = " "
        self.grid = grid
        self.print_grid()

    def print_grid(self):
        for i, cell in enumerate(self.grid.flatten()):
            if i % self.size == 0:
                print()
            if cell == ' ':
                print(f'{cell} ', end = '')
            else:
                print(f'{cell}  ', end = '')
        print()

    def isInsideBoundary(self, pos):
        return 0 <= pos[0] < self.size and 0 <= pos[1] < self.size

    def nextStep(self, a):
        nextPos = copy.deepcopy(self.agentPos)

        match a:
            case 'u':
                nextPos[1] -= 1
            case 'd':
                nextPos[1] += 1
            case 'r':
                nextPos[0] += 1
            case 'l':
                nextPos[0] -= 1
            case _:
                return

        if self.isInsideBoundary(nextPos):
            self.grid[self.agentPos[1], self.agentPos[0]] = '0'
            self.grid[nextPos[1], nextPos[0]] = ' '
            self.agentPos = nextPos

        self.print_grid()


class Agent:
```

```
    def __init__(self, size, nActions, eps):
        self.Q = np.zeros((size, nActions))
        self.eps = eps

    def _epsilon_greedy(self):
        if random.random() < 1 - self.eps:
            return np.argmax(self.Q)
        else:
            return random.randint(0, len(self.Q) - 1)


# Example usage
size, nActions = 10, 4
world = GridWorld(size)
agent = Agent(size, nActions, 0.1)
```

```
  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0
```

## 4.10 References

- **Sections 6.2 and 9.4** of Richard S. Sutton (2018)

# 5 Deep Learning and Neural Architectures

# 6 Value Approximation Methods

## 6.1 Approximation Methods

We will write $\hat{v}(s, \mathrm{w}) \approx v_\pi(s)$

w, but more generally it can be a non-linear function.

## 6.2 Value Function Approximation

Supervised learning, aquiring the values through exploration.

We have:

- the *estimation of the value* of a state (we're in state $S_t$)
- the *actual value* of the state (we get the reward and we are in state $S_{t+1}$).

We'll use that state $S_t$ as the actual correct value of the output of our Neural Network. We try to improve our estimation, reducing the error.

– explanation NN –

# 7 Monte Carlo Methods

# 8 Policy Gradient Methods

# 9 Generative Learning

# 10 Multi-agent Systems

# References


Mitzenmacher, Michael. 2017. *Probability and Computing.* Second edition. Cambridge University Press. https://www.cambridge.org/jp/universitypress/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/probability-and-computing-randomization-and-probabilistic-techniques-algorithms-and-data-analysis-2nd-edition.

Musolesi, Mirco. 2024-25. "Autonomous and Adaptive Systems." https://www.mircomusolesi.org/courses/AAS24-25/AAS24-25-main/.

Ormond, Jim. 2024. "2024 Turing Award." ACM. https://awards.acm.org/about/2024-turing.

Richard S. Sutton, Andrew G. Barto. 2018. *Reinforcement Learning. An Introduction.* Second edition. MIT Press. https://github.com/MrinmoiHossain/Reinforcement-Learning-Specialization-Coursera/tree/master.

Solomonoff, Grace. 06 May 2023. "The Meeting of the Minds That Launched AI." IEEE Spectrum. https://spectrum.ieee.org/dartmouth-ai-workshop.

Tor Lattimore, Csaba Szepesvari. 2020. *Bandit Algorithms.* Cambridge University Press. https://www.cambridge.org/core/books/bandit-algorithms/8E39FD004E6CE036680F90DD0C6F09FC.