Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Matematica -
curriculum Advanced Mathematics for Applications

REINFORCEMENT LEARNING FOR NON-DETERMINISITIC PROBLEMS: EXPLORATION OF CHANCE NODES

Relatore:
Chiar.mo Prof.
GIOVANNI PAOLINI

A cura di:
LORENZO CECCHI

Sessione II
Anno Accademico 2024/2025

Contents

Introduction

The quest to teach computers how to make decisions in complex and chaotic environments has been a central point of research since their invention. Among the frameworks that have been proposed regarding this topic, Reinforcement Learning (RL) stands out. It provides a general approach to sequential decision problems, based on Markov decision processes and stochastic control theory. Reinforcement Learning algorithms are often characterized by the need to balance exploration (trying new actions to gather more information) and exploitation (using known information to gain reward); this is a crucial and recurrent topic in decision-making algorithms.

A simplified but fundamental instance of this dilemma is the multi-armed bandit (MAB) problem. It captures the core of the exploration-exploitation trade-off in a minimal form and serves both as a theoretical tool and a building block for more complex models. Over the years, a rich theory has been developed around the multi-armed bandit problem, including performance bounds and optimal strategies under various assumptions. Classical algorithms such as the ε-greedy strategy and Upper Confidence Bounds (UCB) have been widely studied and applied.

However, in many realistic settings, some form of prior knowledge or initial preference is available regarding possible actions. This leads to extensions of the classical bandit model that incorporate Bayesian priors, like the Thompson sampling algorithm, or externally provided preferences, such as those employed in PUCT: a key component of the Monte Carlo Tree Search (MCTS) algorithm used in AlphaZero.

The aim of this thesis is to investigate the relations between bandit strategies and tree search methods for decision-making, especially in the presence of stochasticity. We begin by reviewing the foundations of Reinforcement Learning (Chapter 1) and formalizing the standard multi-armed bandit problem and its classical solutions (Chapter 2). We then extend the discussion to bandit models that incorporate prior information and analyze the behavior of algorithms like Thompson sampling and PUCT.

Chapter 3 introduces the Monte Carlo Tree Search framework and its integration with bandit methods via UCT and PUCT. In Chapter 4, we shift focus to decision problems with stochastic transitions, which introduce additional complexity through the presence of Chance Nodes. We explore different strategies for dealing with such uncertainty, including random sampling, fixed number sampling and progressive widening.

Finally, Chapter 5 presents a series of numerical experiments designed to compare the performance of the various strategies presented throughout the dissertation, thought to study the suitability of the methods under different types of environments.

1 Reinforcement Learning

Reinforcement Learning is a branch of Machine Learning, which in turn is a branch of Artificial Intelligence. Artificial Intelligence (AI) is a discipline that aims to equip non-human systems with capabilities typically associated with human intelligence, such as reasoning, learning, and planning. These capabilities, once considered uniquely human, are now increasingly being replicated in artificial systems through a variety of computational methods. AI encompasses both symbolic approaches (based on formal logic and knowledge representation) and data-driven techniques, which rely on extracting patterns from large datasets to support autonomous decision-making.

The skill Machine Learning (ML) wants to master is pattern recognition: extrapolate information from known data to better understand new data. This generalization ability is crucial in a wide range of applications, including computer vision, speech recognition, and medical diagnosis. ML models are trained on historical data and are expected to adapt their behavior when encountering new, previously unseen inputs. As such, ML serves as a core mechanism through which AI systems are able to operate effectively in real-world, data-rich environments.

Among all types of ML methods, the one that takes inspiration the most from human learning is Reinforcement Learning (RL): its main idea is to give feedback to an agent in the form of rewards or penalties depending on the choices it makes. This learning paradigm mirrors the trial-and-error process by which humans and animals learn to interact with their environment. An RL agent is not explicitly told which actions to take, but rather must discover optimal behavior through repeated experimentation, guided by the outcomes it experiences.

This idea was born in the middle of the 20th century, inspired by the behavioristic studies of psychologist B.F. Skinner (1904–1990), who introduced the concept of reinforcement as the principal mechanism shaping and regulating behavior. Skinner's experiments demonstrated how organisms could be conditioned to perform specific actions based on the administration of rewards or punishments. These findings laid the conceptual groundwork for computational models of learning based on interaction and feedback.

The Dynamic Programming algorithm [5] is the first instance of a reward associated to a state in order to infer an optimal choice. Bellman's formulation introduced the notion of recursively solving subproblems to construct an optimal solution, a principle that remains foundational in RL. In the following years, the Markov Decision Process (MDP) became the standard formalism to model RL problems. MDP provide a rigorous mathematical framework to represent environments in which outcomes depend both on the agent's decisions and on probabilistic dynamics, making them well-suited to capture the structure of sequential decision-making tasks.

The first concrete achievement came with TD-Gammon [16]: a Backgammon program able to play and beat the strongest human players, thanks to the use of a neural network to approximate rewards. This was one of the earliest examples of combining RL with function approximation via neural networks, enabling learning in high-dimensional spaces where tabular methods were no longer feasible. TD-Gammon demonstrated that reinforcement learning could surpass human expertise in complex strategic domains, thus attracting widespread attention from both academia and industry.

From 2006, the introduction of Monte Carlo Tree Search (MCTS) [11] further improved RL algorithms, which have since surpassed human-level performance in a range of applications, including games [15], recommendation systems [1], financial trading [8], and autonomous control [10]. MCTS enables more efficient exploration of decision spaces by simulating multiple future action trajectories, balancing exploitation of known good strategies with the exploration of less certain alternatives. Its effectiveness was famously demonstrated by AlphaGo and its successors, which achieved superhuman performance in the ancient board game Go. Today, reinforcement learning continues to evolve rapidly, playing a central role in areas such as robotics, healthcare, industrial automation, and adaptive resource management.

1.1 Formalization

Let us take a closer look at Reinforcement Learning, in particular at how we can formally describe its strategy.

The interactions of an agent with the environment occur in discrete time steps and are modeled as a Markov decision process (MDP). The MDP is defined by:

- The set of states S: it contains every possible configuration that the environment can assume at a given time step. Every element of S is called a state and contains only information available to the agent (e.g. in a poker game, a state does not include opponents' hands). Depending on the application, a state can encode the position of a robot, the configuration of a game board, or the current user behavior in a suggestions system.

- The set of actions A possible in each state s. The actions represent the different ways the agent can act in the environment. Each action leads the agent to interact with the environment and cause changes in its state. The action space can also be discrete or continuous and may vary depending on the current state (e.g., some actions may only be available in certain contexts). Choosing the right action is central to the decision-making process.

- $P(s' \mid s, a)$: probability of transition from state s to state s' given action a, that is: P gives the probability of ending in state s' in the following time step, after picking action a in state s and it follows the Markov property, implying that (the probability of) future states of the process depend only upon the present state, not on the sequence of events that preceded it. This assumption greatly simplifies modeling and computation. In practical applications, however, real environments may not strictly satisfy the Markov property, and additional techniques may be required to approximate this setting, such as using history-based features or memory-augmented models.

- $R(s'|s, a)$: reward function on transition from s to s' given a. When the agent in state s picks an action a and moves to state s' in the following time step, it receives an immediate reward from the environment. This reward serves as a signal to guide learning, helping the agent distinguish between good and bad actions based on their outcomes.

In this framework, the aim of an agent is to discover a policy π. A policy is a mapping from states to actions (deterministic) or to action probabilities (stochastic). The optimal policy maximizes the expected cumulative reward over time. Learning an optimal policy is the central goal in Reinforcement Learning.

In Reinforcement Learning, P and R define the world model and represent, respectively, the environment's dynamics and the long-term reward for each policy. If the world model is known (i.e. for every suitable input we know the values of P and R) there is no need to learn to estimate the transition probability and reward function; thus, we can directly calculate the optimal strategy (policy) using model-based approaches such as dynamic programming. These methods, including value iteration and policy iteration, are mathematically elegant and computationally efficient when applicable, but they rely on full knowledge of the environment.

If, instead, the world model is unknown, we may want to approximate the transition and the reward functions by learning estimates of future rewards given by picking action a in state s. Another option could be a model-free approach, where the agent interacts with the environment to gather experience and learn a policy directly from observations, without building an explicit model of P and R. Popular model-free methods include Q-learning, SARSA, and policy gradient algorithms. These approaches are particularly useful when dealing with complex environments where the dynamics are either too difficult to model or entirely unknown.

We then calculate our policy based on these estimates. The process involves balancing exploration—trying new actions to gather more information—and exploitation—using the current knowledge to maximize reward. This exploration-exploitation trade-off is one of the fundamental challenges in RL and is typically handled through strategies such as ε-greedy policies or entropy regularization. As the agent continues to interact with the environment, it updates its estimates and refines its policy, aiming to converge to an optimal or near-optimal behavior over time.

To make this formulation clear, let us see the very simple example of an agent that plays tic-tac-toe.

First of all, we consider S to be the set of all the possible positions that can be reached on the board (if we assume the X player goes first we don't even have to specify which player's turn it is).

We define A as the set of possible moves. For the position s we can furthermore define A_s, a subset of A, containing all moves that the current player can make, like in the following example:

$$s^* = \begin{array}{c|c|c} \text{X} & \text{X} & \\ \hline & & \text{O} \\ \hline & & \text{O} \end{array} \quad \Rightarrow \quad A_{s^*} = \{3, 4, 5, 7, 8\}$$

where a move is represented by the number of the cell where the symbol is put (in left-to-right reading order). In a game like tic-tac-toe, where the rules are known and there isn't a random component, the P function can output only two values:

$$P(s \mid s', a) = \begin{cases} 1 & \text{if } s' \text{ is the only position action } a \text{ yields to} \\ 0 & \text{otherwise} \end{cases}$$

As we just saw, in tic-tac-toe there is a unique position reachable given the current state and the action, hence the reward function depends only on s and a. In position s^* we will get the highest reward for $a = 3$, since it wins the game. This reward is immediately assigned since we reach a terminal state. Every other move lets the match continue and hence does not give immediate reward. If in the following time step the agent will win the game, a positive reward will be assigned to this state-action pair and a negative reward will be back-propagated to the previous one. This makes tic-tac-toe an example of a deterministic environment with a very limited state and action space, where it is often easy to assign clear and unambiguous values to the reward function. The rules are simple, the number of possible states is finite and relatively small, and the consequences of each action can often be computed or predicted exactly.

In this position it was very simple to understand which values R should have returned, but it isn't always the case. Let us think about the first move of a game: which values must the reward function output for the 9 different legal moves? How can we determine it? Tic-tac-toe is one of the simplest game to solve, so what about a chess match, or more complicated games, where even the strongest human players cannot judge with precision the goodness of a position?

In the early stages of a game, the effect of a move may only become evident many steps later, after a long sequence of actions and reactions. This delay in the outcome is what makes the definition of the reward function especially difficult. In complex games like chess or Go, the number of possible configurations is astronomical, and the relationship between an action and its eventual payoff is often obscured by a long horizon of intermediate moves. As a result, the reward signal is sparse and delayed, which complicates the learning process significantly.

It appears clear how one of the main challenges in RL is to find an optimal way to compute or approximate the reward function.

In real-world applications, this challenge becomes even more pressing. Consider autonomous driving, where rewards must be designed to reflect safety, efficiency, and comfort; or healthcare, where actions may have delayed effects on patient outcomes. In these domains, the reward function is not only hard to define, but also difficult to observe directly, requiring domain knowledge or expert data to be inferred. In many cases, RL systems rely on proxy rewards or learned approximations through experience, such as using neural networks to predict expected returns. Therefore, developing robust techniques to estimate and shape reward functions remains a central research focus in reinforcement learning.

1.2 RL algorithms characterization

Various RL approaches have been characterized depending on the following factors:

- Episodic versus incremental: the episodic class comprehends all types of algorithms that work offline and within a finite horizon of multiple training instances. The finite sequence of states, actions and reward signals received within that horizon is called an episode. In the incremental class, learning occurs online and it is not bounded by a horizon. Episodic learning is suitable when the task has a clear beginning and end, such as a game or a simulation with defined boundaries. It allows the agent to reset and learn from bounded trajectories. Incremental learning, instead, is more appropriate in continuous control tasks or real-time systems where interaction with the environment never ends, and updates must be performed at each step without waiting for episode completion.

- Off-policy versus on-policy: an off-policy algorithm approximates the optimal policy independently of the agent's actions. Instead, an on-policy RL algorithm approximates the policy as a process tied to the agent's actions, including the exploration steps.

- Bootstrapping: it estimates how good a state is based on how good we think the next states are. Methods that do not use bootstrapping have to learn each state value separately.

- Backup: with backup we go backwards from a state in the future to the current state we want to evaluate, and consider the in-between state values in our estimates. The backup operation has two main properties: its depth, which varies from one step backward to a full backup, and its breadth, which varies from a randomly selected number of sample states within each time step to a full-breadth backup.

Thanks, but not only, to these features it is possible to classify the most popular RL algorithms:

- Dynamic programming: for this class of methods the complete knowledge of the world model (P and R) is required. These are off-policies algorithms where the optimal policy is calculated via bootstrapping. Dynamic programming was the first RL approach, developed by R. Bellman in the

1950s [5]. Value iteration and policy iteration are two foundational algorithms in this category. They rely on a full sweep of the state space at each iteration, which is computationally feasible only in problems with a limited number of states. Despite these limitations, dynamic programming laid the theoretical groundwork for all subsequent RL algorithms.

- Monte Carlo methods: knowledge of the world model is not required for these kinds of methods. Algorithms of this class are ideal for episodic training and they learn via sample-breadth and full-depth backup. Monte Carlo methods do not use bootstrapping.

- Time Difference (TD) learning: as with Monte Carlo methods, knowledge of the world model is not required and it is thus estimated. TD algorithms are used to solve incremental problems. Algorithms of this type (like Q-learning) are off-policy and learn from experience via bootstrapping and variants of backup.

- Policy gradient methods: algorithms in this class (like TRPO and PPO) are on-policy. They do not necessarily learn a value function to derive a policy, but directly a policy.

These four families of algorithms represent the foundational spectrum of reinforcement learning strategies. While each has its strengths and limitations, modern research often combines elements from different classes to overcome individual problems and improve performance across diverse environments.

2 Multi-armed bandit

The multi-armed bandit (MAB) problem focuses on the optimization of choices in unknown environments; for this reason, it has been deeply studied in literature.

In this problem we have to choose time after time between a finite number K of actions (or arms); in this case we talk about a K-armed bandit. Every time an action is selected, we get a reward based on a certain probability distribution, depending on the action, and the episode concludes. The goal is to maximize the total reward after a fixed amount of episodes. A classical example to better understand the multi-armed bandit problem is the slot-machine; in particular, if we are working with a K-armed bandit we can imagine having K slot-machines in front of us, for which we do not know the distribution of rewards. The only way to gain information is to pull the arm of those slot-machines and in this case the goal is to maximize the win after a fixed amount of plays.

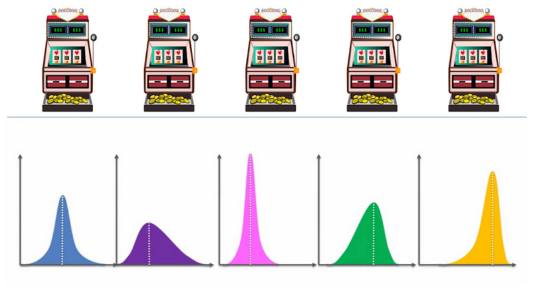

Figure 1: Every slot-machine has a hidden distribution. Every time we select one of them we receive a sampling from that distribution.

Thanks to its huge flexibility and the possibility to infer good decisions in uncertain environments, the multi-armed bandit problem has been used in a great variety of applications. It is utilized in the pharmaceutical industry to test new medicines: every kind of medicine designed to solve a certain problem can be considered a different arm and the goal is to select the best performing one while administering all of them, just like we saw during the vaccination phase of COVID-19. Another field of application is adaptive marketing: if an agency has different options for ads or homepages to choose from, it can present all of them and then start to select those leading to a larger income. As already discussed, the Monte Carlo Tree Search is an application itself of the multi-armed bandit problem; in particular, in the case of AlphaZero the MAB is extended with a neural network that gives an a priori estimate to the expected reward of every arm. We will discuss its implications further in this section. In general, every type of choice that can be evaluated only a posteriori is a multi-armed bandit problem.

2.1 Formal definition of MAB

A K-armed bandit can be defined as a set of actions $A = \{a_1, ..., a_K\}$ associated with their random variables $X_1, ..., X_K$ independent and with unknown expected values $\mu_1, ..., \mu_K$. This is a simple formalization of a multi-armed bandit having static distributions, that is, every action keeps its random variable fixed throughout the whole simulation. There is the possibility to study multi-armed bandits with distributions changing over time, but due to the nature of Monte Carlo Tree Search we may focus only on the static case.

An algorithm used to select the arm to pull is called a policy, or allocation strategy. A good policy can balance exploration and exploitation: exploration sacrifices potential reward for a hypothetical future improvement. To quantify this loss and therefore the efficiency of a policy we introduce the concept of regret.

The regret of a policy P after n plays is defined as:

$$R(P, n) = \mu^* n - \sum_{j=1}^{K} \mu_j \mathbb{E}[T_j(n)].$$

Here, μ^* denotes the maximum between $\mu_1, ..., \mu_K$ and $T_j(n)$ represents the number of times that the policy P has selected arm j in the first n actions. It is important to observe that P is an algorithm that chooses every time the next arm according to the previous rewards obtained, hence $T_j(n)$ can vary according to the realizations returned by the past actions: $T_j(n)$ is a random variable. Another fact is that, a priori, we do not know almost anything about the regret formula, except n.

Regret is therefore the expected loss in reward caused by not always selecting the best action and the goal of a policy to maximize the reward is equivalent to one of minimizing the regret. If we denote $\Delta_j = \mu^* - \mu_j$, then the regret can also be expressed as:

$$R(P, n) = \sum_{j=1}^{K} \Delta_j \mathbb{E}[T_j(n)].$$

In 1985, Lai and Robbins [12] found a lower bound for policies when the distributions of rewards are univariate. They proved that:

$$\liminf_{n\to\infty} \frac{\mathbb{E}[T_j(n)]}{\ln n} \geq \frac{1}{D_{KL}(p_j, p^*)}.$$

Here, p_j is the density function of X_j and p^* is the density of the distribution with the highest expected reward. Lastly $D_{KL}(p_j, p^*)$ is the Kullback–Leibler divergence, defined as:

$$D_{KL}(p_j, p^*) = \int_{-\infty}^{\infty} p_j(x) \ln \frac{p_j(x)}{p^*(x)} dx.$$

This lower bound implies that every policy has an asymptotic regret that grows at least logarithmically.

2.2 Classical algorithms

Over the years, many algorithms have been proposed for the multi-armed problem. These algorithms differ in how they balance the need to gather new information (exploration) with the goal of maximizing rewards based on current knowledge (exploitation). In what follows, we will present some of the most popular strategies. These include simple heuristics like ε-greedy, as well as more established approaches such as Upper Confidence Bound (UCB) and Thompson sampling.

2.2.1 ε-greedy

The ε-greedy algorithm is the simplest in both intuition and computation. Fixed $\varepsilon \in [0, 1]$, every time an arm must be selected, we take the action as follows:

Algorithm 1 - ε-greedy

```
function GreedyPolicy($\varepsilon$, arms)
    sample $r$ uniformly random in $[0, 1]$
    if $r < \varepsilon$
        $a \leftarrow$ uniformly random selected arm
    else
        $a \leftarrow$ arm with highest empiric mean reward
    return $a$
```

Hence, the ε-greedy policy explores with probability ε and exploits with probability $1 - \varepsilon$ independently of previous selections or rewards.

Despite its simplicity, this algorithm can perform reasonably well in practice. This is especially true when the parameter is finely tuned according to the problem nature. A larger choice of ε can be motivated by a large variance in the distributions; of course, some knowledge of the problem is necessary to infer the best value.

Often, the ε-greedy algorithm is performed with a decaying parameter: ε is slowly decreased action after action. In this way, we can encourage the exploration in the first phase of the simulation and then favor the exploitation of the best mean reward when our approximations are more robust.

This algorithm ignores the relative magnitudes of mean rewards: it focuses on the highest and treats all the others in the same way. This causes the search to regularly explore arms that could be easily discarded.

Even if the policy immediately finds the best possible arm, the regret grows linearly:

$$\begin{aligned} R(\varepsilon\text{-greedy}, n) &= \sum_{j=1}^{K} \Delta_j \mathbb{E}[T_j(n)] = \\ &= (1-\varepsilon)\cdot 0 + \varepsilon \sum_{j=1}^{K} \Delta_j \cdot \frac{n}{K} = \\ &= n\frac{\varepsilon}{K}\sum_{j=1}^{K}\Delta_j. \end{aligned}$$

In the second row of the equality, the first addend represents the exploitation phase and the second the exploration.

2.2.2 Upper Confidence Bound

To solve this limitation of the ε-greedy policy we must take into account the empirical mean reward of every single arm. A strategy of this kind has been proposed by Auer et al. [4], called Upper Confidence Bound (UCB). The simplest UCB is UCB1, defined for every arm j as:

$$\text{UCB1}_j = \overline{X}_j + c\sqrt{\frac{2\ln n}{n_j}}$$

where $\overline{X}_j$ is the empirical mean reward value of arm j, n is the number of episodes so far and n_j is the number of times arm j has been selected. The UCB1 policy just computes these values for every arm and then selects the one with the highest result.

The first addend of the sum clearly advantages the best performing arm, while the second addend increases every time the arm isn't selected and hence it advantages exploration, making sure that no arm is unvisited for too long or for a good reason. The balance between these two components can be tuned by the positive parameter c.

The name for Upper Confidence Bound is justified by the fact that this algorithm selects an optimal arm based on an optimistic prediction (upper bound) on the true distribution mean.

It has been proved [3] that, for $c = 1$, this algorithm achieves the best possible regret growth: a logarithmic one. This is the policy that we will use in the numerical experiment section, hence it may be convenient to show a proof of the associated theorem.

Fact 2.1 (Chernoff-Hoeffding bound)**.** Let $X_1, ..., X_n$ be random variables with common range $[0,1]$ and such that $\mathbb{E}[X_t|X_1,...,X_{t-1}] = \mu$. Let $S_n = X_1 + ... + X_n$. Then for all $a \geq 0$,

$$\mathbb{P}\{S_n \geq n\mu + a\} \leq e^{-2a^2/n} \qquad \text{and} \qquad \mathbb{P}\{S_n \leq n\mu - a\} \leq e^{-2a^2/n}.$$

Theorem 2.1. *For all $K > 1$, if policy UCB1 is run on K machines having arbitrary reward distributions with support in $[0,1]$, then its expected regret after any number n of plays is at most:*

$$R(\mathit{UCB1}, n) \leq \left(1 + \frac{\pi^2}{3}\right)\left(\sum_{j=1}^{k} \Delta_j\right) + 8 \ln n \sum_{j:\Delta_j > 0} \left(\frac{1}{\Delta_j}\right)$$

Proof. Let I_t be the random variable that outputs the arm used at time t and let $c_{t,s} = \sqrt{(2\ln t)/s}$. For any arm i, we upper bound $T_i(n)$ on any sequence of plays. More precisely, for each $t \geq 1$ we bound the indicator function of $I_t = i$ (denoted as $\{I_t = i\}$) as follows. Let l be an arbitrary positive integer. Keeping in mind that the first K time steps are spent exploring once every arm we get for $n \geq K$:

$$\begin{aligned}
T_i(n) &= 1 + \sum_{t=K+1}^{n} \{I_t = i\} && (1)\\
&\leq l + \sum_{t=K+1}^{n} \{I_t = i,\ T_i(t-1) \geq l\} && (2)\\
&\leq l + \sum_{t=K+1}^{n} \Big\{\overline{X}^*_{T^*(t-1)} + c_{t-1,T^*(t-1)} \leq \overline{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)}, && (3)\\
&\qquad\qquad T_i(t-1) \geq l\Big\} && (4)\\
&\leq l + \sum_{t=K+1}^{n} \left\{\min_{0<s<t}\left[\overline{X}^*_s + c_{t-1,s}\right] \leq \max_{l \leq s_i < t}\left[\overline{X}_{i,s_i} + c_{t-1,s_i}\right]\right\} && (5)\\
&\leq l + \sum_{t=1}^{\infty}\sum_{s=1}^{t-1}\sum_{s_i=l}^{t-1} \left\{\overline{X}^*_s + c_{t,s} \leq \overline{X}_{i,s_i} + c_{t,s_i}\right\}. && (6)
\end{aligned}$$

Now we observe that $\overline{X}^*_s + c_{t,s} \leq \overline{X}_{i,s_i} + c_{t,s_i}$, implies that at least one of the following must hold:

$$\begin{aligned}
\overline{X}^*_s &\leq \mu^* - c_{t,s} && (7)\\
\overline{X}_{i,s_i} &\geq \mu_i + c_{t,s_i} && (8)\\
\mu^* &< \mu_i + 2c_{t,s_i}. && (9)
\end{aligned}$$

We bound (7) and (8) using Fact 1 (Chernoff-Hoeffding bound):

$$\mathbb{P}\{\overline{X}^*_s \leq \mu^* - c_{t,s}\} \leq e^{-4\ln t} = t^{-4}$$
$$\mathbb{P}\{\overline{X}_{i,s_i} \geq \mu_i + c_{t,s_i}\} \leq e^{-4\ln t} = t^{-4}$$

We also observe that for $s_i \geq (8\ln n)/\Delta_i^2$, condition (9) is false. In fact in this case:

$$\mu^* - \mu_i - 2c_{t,s_i} = \mu^* - \mu_i - 2\sqrt{(2\ln t)/s_i} \geq \mu^* - \mu_i - \Delta_i = 0$$

In the regret formula $\mathbb{E}(T_i(n))$ is multiplied by Δ_i. Therefore, we need to bound the expected value only for the arms with $\Delta_i > 0$. Fixing for these arms $l = \lceil (8 \ln n)/\Delta_i^2 \rceil$ we finally get:

$$\begin{aligned}
\mathbb{E}[T_i(n)] \leq & \left\lceil \frac{8 \ln n}{\Delta_i^2} \right\rceil + \\
& + \sum_{t=1}^{\infty} \sum_{s=1}^{t-1} \sum_{s_i = \lceil (8 \ln n)/\Delta_i^2 \rceil}^{t-1} \Big(\mathbb{P}\{\overline{X}_s^* \leq \mu^* - c_{t,s}\} + \mathbb{P}\{\overline{X}_{i,s_i} \geq \mu_i + c_{t,s_i}\} \Big) \\
\leq & \left\lceil \frac{8 \ln n}{\Delta_i^2} \right\rceil + \sum_{t=1}^{\infty} \sum_{s=1}^{t} \sum_{s_i=1}^{t} 2t^{-4} \\
\leq & \frac{8 \ln n}{\Delta_i^2} + 1 + 2 \sum_{t=1}^{\infty} t^{-2} \\
= & \frac{8 \ln n}{\Delta_i^2} + 1 + \frac{\pi^2}{3}.
\end{aligned}$$

The bound for the regret follows from its definition.

□

Another Upper Confidence Bound policy is UCB2. Even if it can appear very different from UCB1 the core concept is the same: it computes a value UCB2$_j$ for every arm j and then selects the one with the highest return.

The formula to compute this value is slightly different from the previous one and furthermore the arm selected is not pulled just once, but an amount depending on the number of times the arm was previously selected:

$$\text{UCB2}_j^{\alpha} = \overline{X}_j + \sqrt{\frac{(1+\alpha)(1 + \ln(n/\tau(n_j))}{2\tau(n_j)}},$$

where $\alpha \in [0, 1]$ and

$$\tau(n) = \lceil (1+\alpha)^n \rceil.$$

Once again, $\overline{X}_j$ is the empirical mean reward value of arm j, n is the number of plays so far and n_j is the number of times arm j has been selected.

The selected arm is then pulled $\tau(n_j + 1) - \tau(n_j)$ times; in UCB2 it is crucial to distinguish the difference between 'selected' and 'pulled': an arm j is selected (or chosen) when, while computing UCB1 or UCB2, it is found to be the one with highest value. When this happens in UCB1 the arm is then pulled once, hence the number of times it has been selected and the number of times it has been pulled is the same. In UCB2, however, when an arm is selected it is pulled a variable number of times, hence the two numbers can differ.

Just like UCB1, UCB2 also reaches a logarithmic growth. When the distributions have support in $[0, 1]$, the upper bound reads as follows:

$$R(\mathrm{UCB2}_j^\alpha, n) \leq \sum_{j:\Delta_j>0} \left(\frac{(1+\alpha)(1+4\alpha)\ln(2e\Delta_j^2 n)}{2\Delta_j} + \frac{c_\alpha}{\Delta_i} \right),$$

and this is guaranteed when

$$n > \max_{j:\Delta_j>0} \left(\frac{1}{2\Delta_j^2} \right).$$

2.3 Multi-armed bandits with prior

Until now, we talked about the multi-armed bandit problem in environments where we are not informed about the possible distributions of the arms' rewards. However, in many kinds of applications the nature of the random variables to predict is known. This a priori information can be used to facilitate the search for the best arm.

In such cases, the agent can start with a belief about the expected reward of each arm, instead of learning everything from scratch. These beliefs, often referred to as prior knowledge, may come from previous experience, domain expertise, or external predictive models. When this prior information is used properly, it can significantly speed up the learning process and reduce the number of suboptimal choices made during exploration.

This idea is at the core of Bayesian approaches to the bandit problem, where reward distributions are treated as random variables with probability distributions that evolve over time. A well-known method following this principle is Thompson sampling, which uses prior distributions to guide exploration in a natural way.

Similarly, more complex systems, like AlphaZero, integrate prior knowledge into tree search algorithms using predictions from neural networks. In these contexts, the multi-armed bandit problem appears locally at each decision node, and prior-based strategies, such as PUCT, are used to guide the search more efficiently. This will be discussed after in this work, in the Monte Carlo Tree Search chapter.

2.3.1 Thompson sampling

As just said, a popular example of this class of policies is the Thompson sampling (TS)[17]. This algorithm was born to solve the Bernoulli bandit problem: a multi-armed bandit problem where the rewards of every arm are either 0 or 1; therefore, the probability of success (obtaining 1) from pulling arm j is equal to the expected value of the arm, that is μ_j. The Thompson sampling policy associates a Beta distribution to each arm; let us recall that Beta distributions depend on two parameters ($\alpha > 0$ and $\beta > 0$) and have density:

$$f(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-q}.$$

This probability density function has $[0, 1]$ as support and $\frac{\alpha}{\alpha+\beta}$ as expected value. This is very convenient when we have to update the distributional approximations of a Bernoulli distribution: if we obtained the reward we increase α by one, otherwise we increase β by one. Furthermore, these two parameters can be set in advance to exploit the prior information of the model.

The only thing left to understand is how we select the arm to pull. In Thompson sampling there is no need to balance exploration and exploitation since this is automatically managed by the selection strategy: we sample a realization from every distribution and we select the highest one. This gives the majority of the choice in favor of the most promising arms without renouncing to sporadic explorations. It also clarifies why we approximate the unknown random variables with continuous guesses instead of Bernoulli distributions: this allows us to add variability (and hence exploration) in the sampling phase of the algorithm.

In the following algorithm we use S and F instead of α and β: they stand for Success and Failure.

Algorithm 3.1 - Thompson sampling for Bernoulli bandits

```
for each arm $j = 1, ..., K$ set $S_j = 0$ and $F_j = 0$.
create root node $v_0$ with state $s_0$
while within computational budget do
    for $i$ in $1, ..., K$:
        sample $\theta_i$ from Beta$(S_i + 1, F_i + 1)$
    end for
    play $i^* = \arg\max_i \theta_i$
    observe reward $r$
    $S_{i^*} \leftarrow S_{i^*} + r$
    $F_{i^*} \leftarrow F_{i^*} + 1 - r$
end while
```

This policy can be adapted to the general case [2]. To extend it we only need to have an interval bounding the support of the distributions, let us say $[0, 1]$; this can be easily transposed to a more general interval $[a, b]$.

The only change necessary to allow Thompson sampling to work in the more general scenario is the following: after the sampling is made and a reward $\tilde{r}$ is obtained, a Bernoulli trial with success probability $\tilde{r}$ is performed and the new result $r \in \{0, 1\}$ is treated just like in the previous case. Equivalently, one could consider skipping the Bernoulli trial step and simply add $\tilde{r}$ to α and $1 - \tilde{r}$ to β; in fact, $\tilde{r}$ is precisely the expected increment to α when performing the trial and $1 - \tilde{r}$ is the one for β

The algorithm becomes:

Algorithm 3.2 - Thompson sampling for bounded bandits

```
for each arm $j = 1, ..., K$ set $S_j = 0$ and $F_j = 0$.
create root node $v_0$ with state $s_0$
while within computational budget do
    for $i$ in $1, ..., K$:
        sample $\theta_i$ from Beta$(S_i + 1, F_i + 1)$
    end for
    play $i^* = \arg\max_i \theta_i$
    observe reward $\tilde{r}$
    $r \leftarrow$ Bern$(\tilde{r})$
    $S_{i^*} \leftarrow S_{i^*} + r$
    $R_{i^*} \leftarrow R_{i^*} + 1 - r$
end while
```

It is worth noticing that the probability of a success in the Bernoulli trial when the arm j (with unknown density f) is selected is indeed μ_j, in fact:

$$\mathbb{P}(r = 1) = \int_0^1 \tilde{r} f(\tilde{r}) d\tilde{r} = \mu_j.$$

Finally, regarding the convergence rate of Thompson sampling, it has been proved that it reaches the optimal logarithmic growth. Furthermore, when the prior information fits the problem, it can perform better than the UCB policies[6]. The upper bound for Thompson sampling is:

$$R(TS, n) \leq \mathcal{O}\left(\left(\sum_{j:\Delta_j > 0} \frac{1}{\Delta_j^2}\right)^2 \ln n\right).$$

2.3.2 Arms with preferences: PUCT

Another way to obtain prior information in the arm selection is to add weights to every possible choice. These coefficients can be considered as preferences associated with the arms. The process utilized to compute the weights can vary from problem to problem. They can be derived from the knowledge of the environment or alternatively can be estimated by an external estimator, just like a neural network.

If the weights are taken positive and with a total sum of one, they can be considered as the approximate probability that the relative arm is the optimal choice, resulting in an estimation for the policy function.

In this setting, a new metric can be defined to select the most promising arm. The new criterion, called PUCT, is to select the action a maximizing:

$$\text{PUCT}(a) = Q(a) + c \cdot P(a) \frac{\sqrt{N}}{1 + T(a)}$$

where:

- $Q(a)$ is the empirical reward mean obtained when pulling arm a;
- $P(a)$ is the prior policy approximation: the probability that the optimal arm is a;
- $N(a)$ is the number of times that arm a has been selected;
- N is the total number of episodes.

Just like UCB algorithms, the first addend encourages the exploitation while the second increases the urgency of unvisited nodes.

3 Monte Carlo Tree Search for deterministic problems

It is now time to dive into the particular method we are interested in this work. The first piece of our puzzle is called Monte Carlo Tree Search (MCTS). Thanks to MCTS we can evaluate state values more easily, exploring a tree that expands iteration after iteration. Deterministic problems are those for which, given a state s and a valid action a, the next state obtained from s through a is uniquely determined and non-depending on a probability distribution.

This technique stands out for its ability to balance exploration and exploitation while making decisions in complex environments, particularly those with large or infinite state spaces. MCTS iteratively builds a search tree by simulating many possible future trajectories and uses the outcomes of those simulations to estimate the potential value of actions. As the number of simulations increases, the tree grows and becomes more representative of the underlying decisional space.

A fundamental advantage of MCTS is that it does not require a complete model of the environment's dynamics in advance, making it suitable even when the transition and reward functions are partially or entirely unknown. In practice, MCTS has shown great success in applications such as game-playing AI, where the huge number of possible configurations makes exhaustive search impractical.

In deterministic domains, where transitions are predictable and outcomes are fully determined by the current state-action pair, MCTS can more effectively focus its search on the most promising branches. This reduces the need to average over random outcomes and allows the agent to learn a reliable value estimate for each explored state based on direct experience, without the uncertainty introduced by stochastic effects.

Furthermore, the structure of the tree provides a natural way to store and reuse information about previously visited states, increasing efficiency over time. MCTS, as we will see, forms the backbone of several modern reinforcement learning algorithms by enabling intelligent exploration strategies and informed decision-making in large-scale environments.

3.1 How to build a tree

In 1974 Sergio Endrigo stated that "to make a tree, a seed is required"; in the case of MCTS the seed we need is the current state, called root node. From the root node we start expanding: at the beginning of the very first iteration the tree is composed uniquely by its root node and every trajectory helps us enlarge the net more and more. Every iteration starts from the root node, ends in a terminal state and can be subdivided into four steps:

- Selection: starting from the root node we look at all the possible child positions of the state. We select the most urgent child according to our priorities and we reach the next state, then the process is repeated until we find an unexplored child or a terminal node.

- Expansion: the new child is added to the tree.
- Simulation (or play-out): from this new child we simulate the rest of the iteration, until we end up in a terminal state.
- Backpropagation (or back up): we update the values of nodes visited in this iteration with their outcome reward. For every node in the tree we store the mean value of the rewards obtained from trajectories comprehending that node; the backpropagation updates this value as well as other node features that can be useful, such as the number of times it has been visited or how many children it has.

It is important to observe as there are two different policies in this algorithm: the first is utilized in order to select a child node in the selection phase; this is used until the algorithm selects already explored children; this policy is called Tree Policy. The second is utilized in the simulation phase when we compute an outcome; this policy is called Default Policy.

The first choice that comes to mind for both policies is to pick a random action every time a child is to be selected. This idea can work as a good Default Policy, since in the simulation phase we are dealing with unknown nodes, without an associated value. When thinking about Tree Policy a random choice is not optimal; we may want to exploit already explored nodes with high reward values. To do this, we treat the choice as a multi-armed bandit.

3.2 UCB+MCTS=UCT

As previously discussed, the trade-off between exploration and exploitation is fundamental not only in MCTS, but across the entire Reinforcement Learning framework. Focusing just on exploitation may lead the agent to suboptimal long-term behavior, as it might miss out on discovering better strategies. On the other hand, excessive exploration can delay convergence and waste computational resources on bad actions. The UCB algorithms address this challenge offering theoretical guarantees that ensure an optimal balance over time.

The use of UBC formulas within tree search allows MCTS to prioritize promising paths while still maintaining a controlled level of uncertainty sampling. This approach is fundamental in domains with a large or even infinite state space, where a complete search is computationally unfeasible, as it allows us to prune the branches that were discovered to be unpromising iteration after iteration.

A Monte Carlo Tree Search algorithm using an Upper Confidence Bound as Tree Policy is called an Upper Confidence Tree (UCT).

A constant is added to balance the importance given to exploration and exploitation. Given a node w with children $v_1, ..., v_m$, the policy value is computed for every child as:

$$\overline{X}_j + c\sqrt{\frac{2\ln n}{n_j}}\ , \qquad j \in \{1, .., m\},\ c \in \mathbb{R}^+$$

where $\overline{X}_j$ is the current reward value of child v_j, n is the number of times w was explored and n_j is the number of times child j has been chosen. If $n_j = 0$ the policy value is set to infinity; this is not required since in MCTS we do not make use of the Tree Policy until every child node has been visited at least once.

Another remark to point out is the abuse of notation of MCTS: in most of the problems MCTS aims to solve, the graph it generates is not a tree. In fact in games like chess, Go or even the simple tic-tac-toe, the same exact position can be reached with different sequences of moves. This creates closed loops in the graph, which therefore cannot be a tree. The consequences of this flaw are not concerning, but we must deal with them; for example during the simulation phase we could find ourselves in an already explored node. In this case, we can continue using the Default Policy, ignoring previous visits, or switch back to Tree Policy. Once the simulation is concluded and the algorithm enters the backup phase, it goes back parent after parent back-propagating the reward. Since there can be multiple parents for a given child we need to store the sequence of visited nodes to remember the one we came from (theoretically there would not be any problem back-propagating to every parent, but practically the branching could increase the computational cost exponentially). This latter problem can be solved very easily thanks to a recursive implementation of the algorithm, even without storing any state sequence explicitly.

Finally pseudo-code can be stated. Hopefully this will help with the comprehension of the MCTS algorithm.

Algorithm 2 - UCT

```
function TreeSearch($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v \leftarrow$ TreePolicy($v_0$)
        $\Delta \leftarrow$DefaultPolicy($s(v)$)
        Backup($v, \Delta$)
    return $a$(BestChild($v_0$))
```

As discussed before, the TreeSearch function uses the Tree Policy until an unvisited node v_l is met; the function s associates node v_l to the state it represents and from there the Default Policy executes a play-out and returns the obtained reward Δ. This reward is back-propagated to the previously visited nodes. When the computational budget is reached, the search stops and the action with the best reward is returned.

```
function TreePolicy($v$)
    while while $v$ is non-terminal do
        if $v$ not fully expanded then
            return Expand($v$)
        else
            $v \leftarrow$ BestChild($v$)
    return $v$
```

The TreePolicy function takes a node as input and looks for unexpanded children. If node v does not have them, it calls the BestChild function to select the most promising follow-up.

```
function Expand($v$)
    choose $a$ between untried actions from $A(s(v))$
    add a new child $v'$ to $v$
        with $s(v') = f(s(v), a)$
    return $v'$
```

The Expand function creates a new child of the node v. Here, $A(s(v))$ is the set of legal actions in the state represented by the tree node. The function f takes as inputs a state and an action and outputs the new state reached when acting in the state; since this algorithm is designed for deterministic settings, the function guarantees a unique result.

```
function BestChild($v$)
    return $\underset{v' \in \text{ children of } v}{\arg\max} \left\{ \frac{Q(v')}{N(v')} + C_p \sqrt{\frac{2 \ln N(v)}{N(v')}} \right\}$
```

The BestChild function evaluates the UCB value of every child and returns the one with the highest value. Here, $Q(v')$ is the cumulative reward of v' and N(v') is the number of times that it has been visited; their ratio is the empirical mean reward.

```
function DefaultPolicy($s$)
    while $s$ is non-terminal do
        choose $a \in A(s)$ uniformly at random
        $s \leftarrow f(s, a)$
    return reward for state $s$
```

The DefaultPolicy function takes as input a state s and performs random actions until a terminal state is reached, then returns the reward associated to that state.

```
function Backup($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta$
        $v \leftarrow$ parent of $v$
```

Finally, the Backup function ascends the tree back-propagating the reward obtained thanks to the roll-out. It also increases by one the total amount of times the nodes have been visited.

It is important to observe that, if the algorithm is used to study a two-player turn-based game, the sign of Δ must be changed at every step of the Backup function. In this way, a move that allowed a player to receive a higher reward will be penalized.

3.3 Neural Networks for Reward Prediction

Neural Networks (NN) are approximating functions. Their architecture is composed of layers: one for the inputs, one for the outputs and between them a variable number of so-called hidden layers. When we go from the input layer to the following one, we apply a linear transformation to the entries, we add a bias, then apply a non-linear function, called "activation function". This procedure is repeated for every other layer until the output layer is reached. The coefficients of the linear transformation and biases are what define the Neural Network hence there is a need to find those guarantee the best approximation.

Computing fitting coefficients is called training: we feed the input to the model with random initial coefficients and use its outcome to back-propagate the error, correcting the weights. This process is repeated until a threshold, for example iteration number or residual, is met.

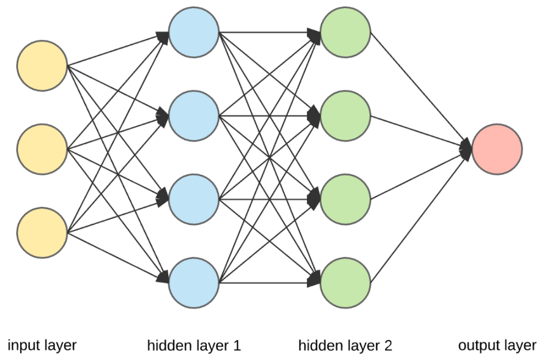

Figure 2: Mandatory image when introducing neural networks.

In MCTS, every time we find an unexplored node, the Simulation Phase starts and it assigns a reward to the position via the Default Policy. The Default policy is often simply a random choice, as in this phase the algorithm just selects random moves until a terminal position is met. This method works in a large number of cases. Despite this, it can be a poor strategy to get a reward estimation.

To improve this approximation we can use Neural Networks. The natural question is the following: how do we train the model?

In 2016, Silver et al.[14] answered this question while working on a model, AlphaGo, to play Go, a two-player deterministic game known for the very large number of possible moves in every position and the difficulty of evaluating them.

In games like Go it is unthinkable to rely on a random sequence of moves to obtain a reward; a match is too long and the importance of a single move can be lost due to the rampant ramification that is generated while waiting for a terminal position.

To solve this problem they made use of a neural network capable, when fed with a state, to approximate the probability of every child node to be the optimal one. The same network could also give an estimation of the value of the position, according to an appropriate metric. Having a policy approximation is crucial: it allows to have an a priori evaluation of the children without having to visit every one of them; in games like Go, where the number of possible moves is in the order of the hundreds, it is crucial to have the possibility to ignore some (or moreover a large portion) of them. This information can be used with the previously discussed PUCT algorithm as Selection Phase criterion of the model.

AlphaZero was trained entirely through self-play using Reinforcement Learning. Starting with no prior knowledge except for the game rules, it played millions of games against itself. At each step, it used a neural network to evaluate positions and suggest moves, which were then refined through Monte Carlo Tree Search. The data from these games was used to continually update the network, allowing AlphaZero to rapidly improve without relying on human matches or game theory.

This model reached levels of strength that at the time were unexpected, being the first Artificial Intelligence Go program able to compete and defeat the best human players consistently on a regular 19×19 board.

4 Stochastic problem

Until this point, every method or result was obtained for deterministic environments; this is a huge limitation since probability can be found in a very large number of problems. For the latter class of problems, there is a need to extend the algorithms we saw for the deterministic scenario.

In the example of AlphaZero, when we train the neural network the stochastic nature of the problem does not affect the procedure. It is in the self-play part where we need to adapt those methods to the probabilistic environment, specifically in the Monte Carlo Tree Search.

In the deterministic case, choosing an action a in state s is equivalent to choosing the state s' that a yields; in the stochastic case, when we choose an action, we are not sure of its outcome state; in fact, it will depend on a random distribution.

In this scenario, we define two types of nodes within the tree: Chance Nodes and Decision Nodes [9]. Chance Nodes are those nodes where the agent has no power to influence the outcome of the next position, which depends only on a random realization of a distribution. On the other hand, Decision Nodes are those nodes where the agent has to choose an action and directly affect the environment.

To make this definition clear, here are two examples:

- Risk: Risk is a multiplayer war simulation board game. At every turn, the current player places some tanks on the board and then decides whether or not to attack a neighboring enemy territory. If the decision is to attack, both players roll dice and remove tanks according to the result; after this, the current player can then judge whether continue his attack, and therefore repeat the dice roll, attack some other territories, or terminate the offensive action (how this precisely works is not fundamental in order to understand the example).

  At the beginning of its turn, the agent is on a Decision Node: we must choose where to place the tanks and whether it wants to attack. If the action selected by the agent is peaceful its turn ends, otherwise some dice must be rolled and therefore the trajectory on the tree meets a Chance Node. The action that acts on the environment depends only on the dice realization. Once tanks are removed, the agent returns to a Decision Node and the process repeats.

- 2048: 2048 is a single-player video game, played on a 4×4 board. A cell may contain a number, in particular an integer power of two; if a cell does not contain any number it is empty. At every turn, the player must swipe the board in a chosen cardinal direction; every number is shifted in that direction until it hits the end of the grid or a different number. The main mechanic of the game is the following: if two equal numbers hit themselves, they merge and form a single number, that is their sum or equivalently their double, and the total is added to the player's score. After the board is swiped, a new number (a two or a four) is placed randomly on an empty cell; if this is impossible because every cell contains a number the game ends. The goal of 2048 is to score the most amount of points possible before the game ends.

When the agent must choose a direction it is on a Decision Node; in 2048 a Decision Node always has four possible children. After an action is selected (and if the game is not over), we are on a Chance Node: a new number is placed on a random empty cell. Then the process repeats until a terminal state is reached.

Let us visualize the first steps of a trajectory in the Monte Carlo Tree Search taking as root node Position A:

Position A

		2	
16		16	2
	8		

Position B

2			
32	2		
8			

Position C

2			
32	2		
		4	
8			

Decision Node A: in this position the agent selects an action (west direction in this example) following the Tree Policy. The board then transforms according to the rules.

Chance Node B: once the new position is reached, the stochastic component of the game arises; if we were playing a match, the RNG would choose a random cell and insert there a new number. In this case we have 12 empty cells; since either a two or a four can be inserted we have exactly 24 possible children. While building a tree, we can decide in which way this action is performed: we could simply take a realization from a uniform distribution among all children, or study an ad hoc strategy to improve the efficiency of the search, i.e. some kind of "Chance Node Policy".

Decision Node C: a new position is reached and the agent has to choose an action again.

The following figure represents the decision process that we just described.

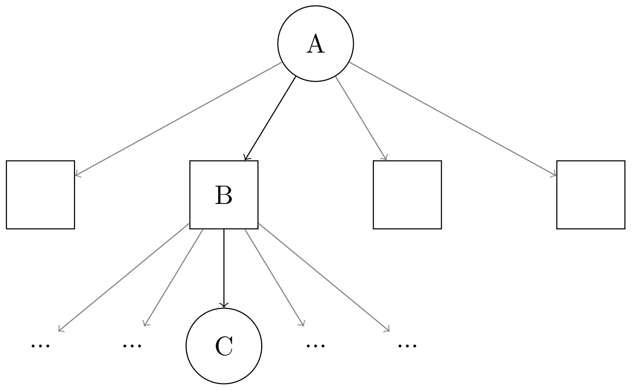

Figure 3: Here Decision Nodes are represented by circles and Chance Nodes by squares.

4.1 Chance Nodes exploration methods

The difference between Decision Nodes and Chance Nodes has consequences inside the Monte Carlo Tree Search: first of all, Chance Nodes can never be root nodes. In Reinforcement Learning algorithms, Monte Carlo Tree Search is called when the agent must perform an action, therefore in a Decision Node; in states corresponding to a Chance Node there is no choice to make. Moreover, this separation between node types enforces a natural alternation in the tree structure, helping to clarify the distinction between agent-driven and environment-driven choices.

Another difference is the following: when we explore a Chance Node we cannot use policies such as UCB1, or in any case policies that aim to find the most promising child node. If we decide to proceed this way, we are just trying to locate the best possible realization of the distribution, and the search will focus mostly on the exploration of this node. This behavior leads to a biased search tree that overestimates the expected value of the Chance Node, since it implicitly assumes the most favorable outcomes occur more frequently than they actually do. This could result in a distorted value propagation and could compromise the overall performance of the Monte Carlo Tree Search algorithm, particularly in domains where variance in outcomes plays a critical role. In the Risk example, this would be equivalent to prioritizing the exploration of the positions yielded by perfect dice rolls, thus ignoring the evaluation of positions coming from unlucky or average rolls: an overly optimistic approach.

The first idea that could come to mind, just like in the Tree Policy discussion, is to randomly select a node, maybe according to the distribution intrinsic in the state. This is a realistic way to sample stochastic realizations, and can be a good policy if the possible outcomes are not a large number. In contrast, if this number grows too much, sticking to the real distribution could lead to a sparse and shallow exploration, hence another idea can be to extract a not-too-big number of realizations and then to explore those nodes with more depth.

In practice, this translates to approximating the full support of the distribution with a manageable subset of representative samples, on which the search can then concentrate. This strategy aims to achieve a balance between representativeness and computational feasibility. Instead of spreading the search effort thinly across a vast number of low-probability branches, the algorithm narrows its focus only to a small set of sampled outcomes, each of which can be explored with greater depth.

This allows the MCTS to gain more informative estimates from each rollout, leading to better value propagation throughout the tree. However, the trade-off is clear: by reducing the number of outcomes considered, we risk missing rare but critical transitions, and so the sampling strategy must be chosen carefully to preserve the statistical integrity of the underlying process.

Plenty of methods to explore Chance Nodes have been proposed over the years. In the next segment of this dissertation we look closer to the most popular choices.

4.1.1 Random sampling

Random sampling is the most environment-coherent method: every time a Chance Node is visited a realization from the real stochastic distribution is generated and explored. This is the main strength of the method, since its trajectories follow the exact dynamics of the domain. The behavior of the search is naturally aligned with how the environment works, ensuring a faithful representation of the underlying stochastic process. This makes random sampling an appealing strategy, especially when one is interested in preserving the statistical properties of the domain.

The performance of this technique depends heavily on the ramification of the nodes: when the number of potential children increases, the power of random sampling decreases. In such cases, the probability of encountering low-probability but relevant outcomes becomes negligible, which might introduce a bias in the value estimation. Having a large number of possible realizations can lead to a very shallow exploration of visited nodes. This drawback can seriously affect the quality of the search, especially in early stages, where the estimates rely on few simulations and the variance is still high.

In the 2048 example with a 4×4 grid, random sampling can work without problems. The number of available cells and possible values that can appear is limited, so the stochastic space remains relatively compact. In this setting, sampling from the true distribution produces realistic sequences of moves and outcomes, allowing the tree to capture meaningful game dynamics without being overwhelmed by excessive branching. In the experimental part of this thesis we will study how the performance changes with the enlargement of the grid. The idea is to observe how the increasing number of chance events and possible outcomes affects the effectiveness of random sampling and whether its limitations become more evident as the state space grows. This will help us understand the scalability of the method and its practical applicability in more complex variants of the game.

4.1.2 Fixed number sampling

Fixed number sampling is a self-explanatory name: in this method, when Chance Nodes are visited during the tree search, the maximum number of child nodes taken into consideration is fixed in advance. Instead of expanding all possible outcomes, which can be extremely costly in terms of computation, only a limited number of child nodes are sampled. The number of children sampled depends not only trivially on the problem domain but also more subtly on the specific Chance Node that is being visited. In fact, different Chance Nodes might represent different kinds of stochastic events and therefore benefit from different sampling strategies. This approach can significantly increase performance in those cases where there is a huge number of possible random realizations and the resulting differences among the following states are small or even negligible. By focusing only on a representative subset, the method avoids unnecessary expansion and computational overhead, especially when exhaustive exploration would yield diminishing returns.

Another strong point of Fixed Number Sampling is its ability to discretize a continuous environment into a limited, manageable subset of events. While Random Sampling can also take realizations from a continuous context, it usually does so by drawing a new random outcome every time a node is expanded. This means that each new sample likely corresponds to a child node that has never been seen before, making the search very shallow and highly variable. In contrast, Fixed Number Sampling selects a closed number of child nodes. This behavior promotes a more focused and deeper search in the most relevant regions of the search space.

However, this technique is not without limitations. While using Fixed Number Sampling, there is a non-negligible risk of missing some important but rare realizations simply due to chance. If the fixed set of children does not include these rare but important events, the resulting policy may be biased or incomplete. Environments characterized by such rare but significant stochastic outcomes are therefore not very suitable for this method, unless additional mechanisms are used to ensure adequate coverage. Another flaw of fixed number sampling lies in the determination of the optimal threshold number of children to be sampled. This threshold is highly problem-dependent, and more precisely, it depends on the stochastic distribution of the outcomes associated with each specific Chance Node. As a result, it can be difficult to choose a fixed number that works well across the entire domain. If the number is too small, important outcomes might be left out; if it is too large, the method loses its computational advantage.

4.1.3 Progressive widening

Progressive widening can be seen as an improved and more flexible version of fixed number sampling. Like the previous method, it aims to keep the total number of children explored at each node low, which helps to reduce the computational cost and manage the complexity of the tree. However, instead of selecting an a priori fixed number of possible different child nodes, progressive widening regulates the creation of new children dynamically. This is done according to a specific rule, defined by the following formula [7]:

$$C(n) \leq kN(n)^{\alpha}.$$

In this equation, $C(n)$ represents the number of distinct children that have been visited at node n, while $N(n)$ indicates the number of times that node n has been visited during the search. The parameters k and α are tunable constants, where $k > 0$ and $0 \leq \alpha < 1$. These parameters control how aggressively the method expands the node as more information becomes available. Notably, fixed number sampling with a threshold M can be interpreted as a special case of progressive widening, in which $k = M$ and $\alpha = 0$. In that case, the number of children is fixed and does not grow with visits.

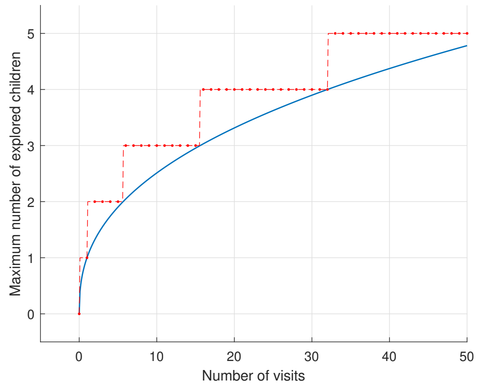

Figure 4: Here is an example of the progressive widening sampling function with $k = 1$ and $\alpha = 0.4$; every time the function surpass an integer value, a new child can be explored.

When the Chance Node can still have new children, we select one randomly among all the possible choices; otherwise, we take one from the pool of already visited.

As in fixed number sampling, the exploration here is initially focused on a small batch of children. This allows the search to stay computationally efficient in the early stages, when not much is known about a node. On the other hand, if a Chance Node n is explored many times, the value of $N(n)$ will gradually increase, and so will the allowed number of children $C(n)$. This behavior allows the tree to grow progressively and adaptively: nodes that seem important, because they are visited often, get more children over time. In this way, the search becomes deeper and more refined where it is more useful, without wasting resources on unpromising areas of the tree.

As previously observed with the fixed number sampling method, the performance of progressive widening is also influenced by the choice of its parameters. In this case, both k and α need to be chosen carefully, depending on the nature of the stochastic distribution associated with the specific node. If the distribution is highly variable or contains rare but important events, setting these parameters incorrectly could result in poor exploration. Despite this challenge, progressive widening tends to adapt better than fixed number sampling when dealing with promising nodes, because it allocates more attention to frequently visited areas of the tree.

5 Numerical Experiments

In this section, we will explain and discuss the results induced by previously introduced methods. In particular, we will test the efficiency of pure Monte Carlo Tree Search in stochastic environments, such as the 2048 game.

Let us briefly recall the rules of the game:

- 2048 is a single-player game played on a 4×4 square grid.
- The grid starts empty and at every turn a random number is placed. The number can be either a 2 (90% chance) or a 4 (10% chance), and it is placed in one of the empty cells of the board. If the grid is full and there is no valid position for a new number the match finishes.
- At every turn the player must choose a cardinal direction; the board is then swiped in that direction.
- If two equal numbers swipe one next to the other in the chosen direction, they merge and form a single number with the value of their sum. This value is added to the match score.
- The goal of the game is to score the highest amount of points possible before the grid is completely full and the match is over.

To study how the efficiency of the method changes with the enlargement of the possible children the experiments are conducted on a regular 4×4 board, as well as a smaller 3×3 board and a larger 5×5 board; for every board size the time limit for a move is at most one second. For every one of these cases the Decision Nodes are selected with the UCB1 policy.

Every histogram is associated with the results obtained with one of the three methods introduced in Section 4.1. These are random sampling, fixed number sampling and progressive widening. The first method is compared with the results obtained while playing random moves. The last two methods depend on some parameters; in their histograms there is also a comparison between different choices. If not specified otherwise, the k parameter for the progressive widening method is set to one. In every histogram, we also show the confidence intervals computed by the Clopper-Pearson method with nominal coverage of 95%. This means that every time the experiments we proposed are repeated, in at least 95% of cases the results obtained will fall in the confidence interval.

Lastly, since 2048 tiles can only contain powers of two, we indicate the magnitude of a cell by its base two logarithm to help the comprehension of the match advancement.

5.1 3×3 grid

For the 3×3 grid, every different policy has more than 200 games played. The speed of the random move method allowed to simulate 10 000 matches. Here are the results.

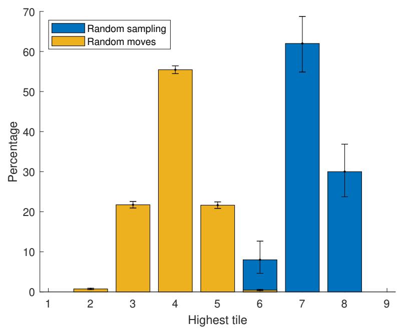

Figure 5: The best game with random moves is almost always worse than the worst game with the random sampling method: encouraging.

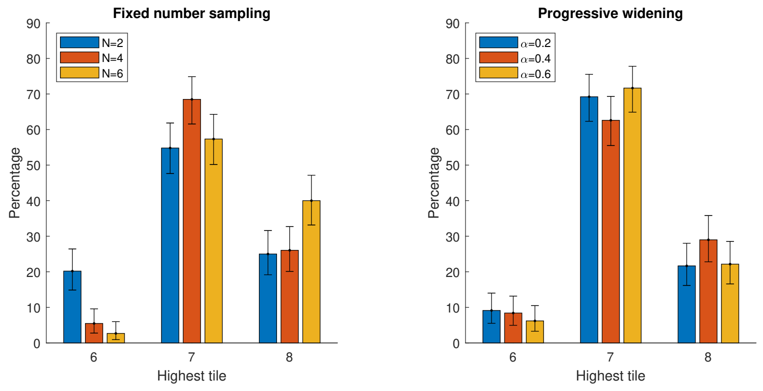

Figure 6: Fixed number sampling and Progressive widening.

In this small grid, there are no particular differences between the various Monte Carlo Tree Search methods arising from the histograms. This is due to the limited number of children that a node can have: on a 3×3 grid there is a total of 9 cells and during our simulation there was a mean of 2.57 empty cells per Chance Node. We can have both 2 and 4 in every cell and therefore around 5 child per node; despite that, the probability of a 2 is 90% while for 4 is 10%, hence when we sample according to the real distribution we are more likely to get child nodes corresponding to the insertion of a 2. This means that in the fixed number sampling, in the majority of cases, $N = 4$ and $N = 6$ led to the same exact behavior, while $N = 2$ was just a little different.

5.2 4×4 grid

For the 4×4 grid, every different policy has 100 games played. Just like in the previous case, the speed of the random move method allowed to simulate 10 000 matches. Here are the results.

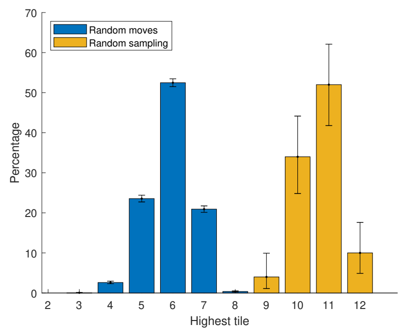

Figure 7: In this case, there is no intersections between the two histograms.

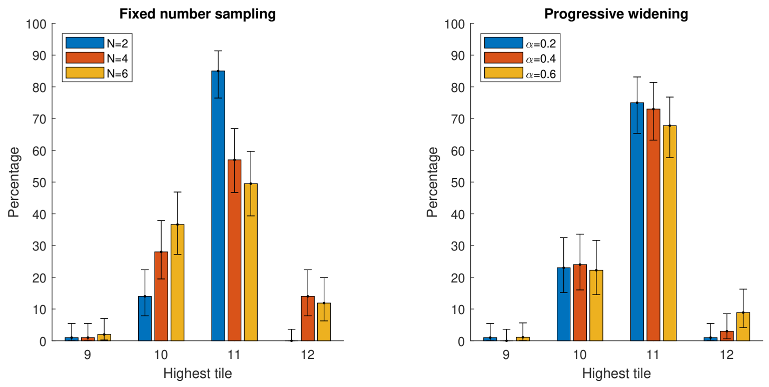

Figure 8: Fixed number sampling and Progressive widening.

This time, some differences arise: in particular, it appears how the fixed number sampling, unlike in the previous case, gives us results much more dependent on the parameter variation; for the 4×4 grid, the mean number of empty cells grew to 4.46: this justifies both the similarity of the $N = 4$ and $N = 6$ results and the $N = 2$ difference. It seems that having a smaller number of children allowed the agent to reach the goal of 2048 tiles more often.

The same difference is not visible for the progressive widening algorithm. The cause of this behavior can be found within the number of children per Chance Node: in progressive widening, the mean number of children grows slower (with respect to its parameter) than in fixed number sampling. On the other hand, the fixed number sampling method averaged 1.39 children per Chance Node for the $N = 2$ case and 1.91 for $N = 6$, while progressive widening registered 1.43 children per Chance Node for $\alpha = 0.2$ and 1.68 for $\alpha = 0.2$.

In any way, all policies reach at least the 2048 tile (corresponding to 11 on the graphs) in more than half of the matches. In a 2014 study by P. Rodgers and J. Levine [13], the authors reached similar results when investing one second for search; unfortunately, they did not mention any information about the hardware they used, this denies us a rigorous comparison.

5.3 5×5 grid

Also for the 5×5 grid, every different policy has 100 games played. Just like in the previous case, the speed of the random move method allowed to simulate 10 000 matches. Here are the results.

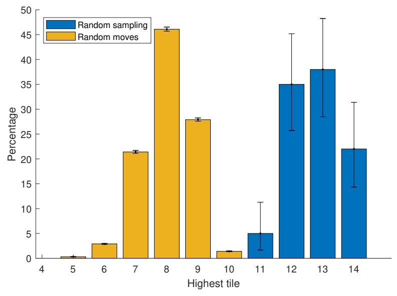

Figure 9: Also in this case, there is no intersections between the two histograms.

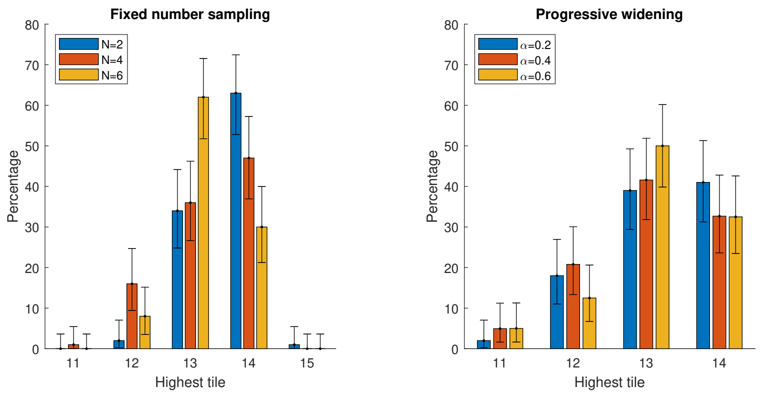

Figure 10: Fixed number sampling and Progressive widening.

In this latter case, the enlargement of the grid allowed the fixed number sampling and progressive widening methods to outperform random sampling; confirming the superiority of these two methods when the number of children is large. The latter two methods reached the 16384 tile (corresponding to 14 in the histograms) with a frequency between one in three and one in two, with a spike in performance for the fixed number sampling with $N = 2$, reaching this advancement two times every three; random sampling only achieved this result less then one time every four. This time, the mean number of children per Chance Node is 5.80; this causes the visible difference in performance between the fixed number sampling. Even this time, the average number of children per Chance Nodes increased more slowly for the progressive widening.

Conclusions

This thesis has examined the connection between Reinforcement Learning, multi-armed bandit algorithms, and Monte Carlo Tree Search (MCTS), with a focus on the role of prior information and stochastic elements in decision-making.

We started by introducing the fundamentals of Reinforcement Learning and the exploration–exploitation trade-off, formalized through the multi-armed bandit (MAB) problem. After reviewing classical MAB algorithms, we explored variants incorporating prior knowledge, such as Thompson sampling and PUCT. We then discussed how bandit strategies are integrated into MCTS, both in deterministic and stochastic settings. For the latter class, we addressed the added complexity of Chance Nodes, comparing different exploration strategies like random sampling, fixed number sampling, and progressive widening.

There are plenty of future developments. These may include extending algorithms to continuous domains and applying them to real-world decision problems with complex dynamics. It could be interesting to test Chance Nodes exploration techniques for games with two or more players or fuse them with policy estimators like in AlphaZero. Another theme is to understand how the methods discussed in this work improve when more time is given to the Monte Carlo Tree Search and most importantly how to recognize random distributions that favor a specific algorithm.

References

[1] M Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*, 55(7):1–38, 2022.

[2] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on learning theory*, pages 39–1. JMLR Workshop and Conference Proceedings, 2012.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[4] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[5] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[6] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. *Advances in neural information processing systems*, 24, 2011.

[7] Adrien Couetoux and Hassen Doghmen. Adding double progressive widening to upper confidence trees to cope with uncertainty in planning problems. In *The 9th European Workshop on Reinforcement Learning (EWRL-9)*, 2011.

[8] Yue Deng, Feng Bao, Youyong Kong, Zhiquan Ren, and Qionghai Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems*, 28(3):653–664, 2016.

[9] Thomas Gordon Hauk. Search in trees with chance nodes. 2004.

[10] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE transactions on intelligent transportation systems*, 23(6):4909–4926, 2021.

[11] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[12] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[13] Philip Rodgers and John Levine. An investigation into 2048 ai strategies. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–2. IEEE, 2014.

[14] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[15] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[16] Gerald Tesauro. Temporal difference learning of backgammon strategy. In *Machine Learning Proceedings 1992*, pages 451–457. Elsevier, 1992.

[17] William R. Thompson. *On the likelihood that one unknown probability exceeds another in view of the evidence of two samples*, volume 1. Biometrika, 1933.

Ringraziamenti

Il primo ringraziamento di questa tesi va al mio relatore, il Professore Giovanni Paolini. Grazie per avermi seguito durante questo lungo percorso, per avermi guidato con grande chiarezza e comprensione anche nei momenti più critici di questa stesura, e per avermi insegnato l'utilizzo corretto della virgola.

Grazie a mia mamma per l'affetto che mi dai ogni giorno, anche quando sono girato male o non ho voglia di parlare. Ti sono e sarò per sempre grato per tutto quello che hai fatto per me e mia sorella. Ti voglio bene.

Grazie alla mia sopracitata sorella Linda. Le tue rare apparizioni portano sempre disordine, talvolta nuovi animali, ma soprattutto leggerezza e felicità. Ti voglio bene anche a te!

Grazie a Giorgia, o meglio Gio. Sono grato di aver vissuto anche questo capitolo della mia vita insieme a te e non vedo l'ora di iniziare il prossimo sempre al tuo fianco. Sei una persona speciale e la tua sola presenza mi tranquillizza, grazie per tutto il tempo passato insieme e per quello che passeremo ancora.

Infine, ci tengo a ringraziare anche i miei amiconi, i miei parenti, e più in generale tutte le persone a cui voglio bene e che mi sono state vicine in questi tre anni. Grazie!