



UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia “Galileo Galilei”

Master Degree in Physics of Data

Final Dissertation

Optimizing Workforce Allocation through Reinforcement Learning: A Machine Learning Approach to Human Resource Management

Thesis Supervisor:

Prof. Gian Antonio Susto

Thesis Co-Supervisor:

Dr. Marina Ceccon

Candidate:

Valentina Tonazzo

Academic Year 2024/2025

*Non so come il mondo potrà giudicarmi ma a me
sembra soltanto di essere un bambino che gioca sulla
spiaggia, e di essermi divertito a trovare ogni tanto un
sasso o una conchiglia più bella del solito, mentre
l'oceano della verità giaceva inesplorato davanti a me.*

Sir Isaac Newton

Abstract

Efficient management of human resources is crucial for a company's success, especially in contexts where projects require specific skills and expertise from employees. This thesis aims to address the problem of optimal employee assignment to company projects through the use of Reinforcement Learning (RL) techniques. The main objective is to create an algorithm capable of maximizing the company's productivity while minimizing assignment execution time and improving overall operational efficiency. The assignment problem has been formalized as an RL environment, where each employee is represented by a set of skills and competence levels, while each project is characterized by specific requirements in terms of skills and experience. Using an agent based on Deep Q-Networks (DQN), the algorithm is able to learn the best assignment strategies by considering the rewards associated with employee performance on each project. The reward is defined based on the completion of projects within the expected timeframe and the quality of work performed, thus contributing to improving overall efficiency. During the experimentation phase, the algorithm showed a progressive improvement in its assignment capabilities. The sum of rewards, both in terms of time steps and absolute value, showed a significant increase, indicating that the agent has learned to optimize resource allocation over time. We also evaluated the algorithm through various performance metrics, such as the average number of correct assignments, the overall time reduction needed to complete projects, and the satisfaction of competence requirements for each project. The results demonstrate that the developed Reinforcement Learning algorithm can significantly improve employee assignment compared to traditional methods, which are based on fixed rules or manual assignments. In conclusion, this work provides a solid foundation for future developments in the field of human resource management assisted by artificial intelligence, with potential applications extending to other industrial sectors where optimal resource management is a critical challenge.

Contents

1	Introduction	5
2	Strategic Workforce Planning	7
2.1	Problem Definition	7
2.2	Main Challenges	7
2.3	Traditional Approaches	10
2.3.1	Heuristic Approaches	10
2.3.2	Linear Programming	12
3	Reinforcement Learning	14
3.1	General Overview	14
3.2	Basic Concepts	16
3.2.1	Exploration vs Exploitation	20
3.3	Fundamental Algorithms	21
3.3.1	Q-Learning	21
3.3.2	Deep Q-Learning	22
3.3.3	Policy Gradient Methods (PGM)	24
3.4	Business Applications of RL	27
3.4.1	Relevant Case Studies	27
3.4.2	Advantages of RL in Business Contexts	31
4	Project Description	33
4.1	Problem Specifications	33
4.1.1	Description of Variables: Employees, Projects	33
4.1.2	Constraint Descriptions: budget, priorities, availability	35
4.2	Problem Formalization in RL	36
4.2.1	Environment	36
4.2.2	States and Spaces	37
4.2.3	Actions and Spaces	39
4.2.4	Reward Function: Key Components	40
4.3	Design Choices: PPO Algorithm	42
4.3.1	Introduction	42
4.3.2	Theoretical Foundations	42
5	Implementation	46
5.1	Environment Description	46
5.1.1	Environment Implementation and Its Functions	46
5.2	Creation of PPO Model	48
5.3	Training	50
5.4	Test and Validation	52

6 Scenarios and Results	55
6.1 Baseline	55
6.2 Staff Shortage: Under-sampling	61
6.3 Project Shortage: Over-sampling	65
6.4 Conclusions	70

Chapter 1

Introduction

Reinforcement Learning (RL) represents a paradigm within machine learning wherein an agent iteratively interacts with an environment to maximize cumulative rewards through a series of actions. Distinct from supervised learning, which depends on labeled datasets, RL relies on the principle of trial and error. The agent receives feedback in the form of rewards or penalties and modifies its actions to enhance future performance.

The conceptual foundations of RL trace back to the early 20th century with the work of Edward Thorndike on the law of effect, which postulated that *"responses that produce a satisfying effect in a particular situation become more likely to occur again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation"* [12]. This idea evolved through the mid-20th century with the advent of behaviorism, where researchers like B.F. Skinner developed operant conditioning frameworks. However, it wasn't until the late 20th century that RL gained formal structure through the contributions of Richard Sutton and Andrew Barto, who provided a rigorous theoretical foundation and introduced algorithms such as Temporal Difference (TD) learning and Q-learning. These developments laid the groundwork for modern RL applications.

In an RL framework, the agent operates in discrete time steps. During each step, it observes the current state, selects an action based on a policy, receives a reward, and transitions to a new state. The objective is to derive an optimal policy that maximizes the expected cumulative reward over time. Essential components of RL include the policy (action selection strategy), the reward signal (environmental feedback), the value function (expected long-term reward), and the model of the environment (predictive state transition model).

Strategic Workforce Planning (SWP) involves aligning an organization's human resources with its strategic objectives. The origins of SWP can be traced back to post-World War II era when organizations recognized the need for systematic approaches to manage their labor forces amid rapid industrial growth and technological advancements. Initially, SWP focused on headcount planning and straightforward supply-demand forecasting. During the 1980s and 1990s, SWP evolved with the advent of human resource management theories emphasizing the strategic alignment of workforce capabilities with organizational goals. This period saw the integration of more sophisticated analytical methods and the emergence of software tools to aid in forecasting and planning.

Traditional SWP methodologies often relied on historical data and static models. These approaches involved analyzing past trends to predict future workforce needs, identifying skill gaps, and developing strategies for recruitment, retention, and training. Although effective to some extent, these methods frequently fell short in addressing the dynamic complexities of modern organizational environments, which are characterized by rapid technological changes, fluctuating market demands, and evolving employee expectations.

This thesis introduces several innovative aspects within the broader research on Reinforcement Learning applied to Strategic Workforce Planning. A significant contribution is the development of a novel, custom-designed simulation environment specifically tailored to model the intricate dynamics and constraints typical of an IT consulting firm's human resource management processes. This sim-

ulation environment, unprecedented in existing literature, allows for realistic modeling of operational dynamics, providing a credible and effective testing platform for intelligent agents.

Utilizing the Proximal Policy Optimization (PPO) algorithm, this research demonstrates the agent's ability to learn optimal workforce allocation strategies effectively. Experimental results confirm the agent's proficiency in continuously enhancing its decision-making capabilities, achieving considerable improvements over traditional, static SWP methods. Moreover, the research extends its evaluation to more challenging scenarios, such as over-sampling and under-sampling conditions, revealing the robustness and adaptability of the trained agent in dynamically adjusting to varying operational constraints while maintaining high performance.

These findings underscore not only the validity and effectiveness of the proposed RL-based methodology but also suggest significant potential for real-world applications. The integration of advanced RL techniques and realistic workforce simulations provides a promising foundation for optimizing human resource allocation, thereby enabling more informed, efficient, and adaptive strategic decisions within complex organizational settings.

Chapter 2

Strategic Workforce Planning

2.1 Problem Definition

Strategic Workforce Planning (SWP) is a crucial management process that enables organizations to align their human resources with business strategic goals and evolving market needs. This process is based on identifying the necessary skills, forecasting future personnel demand, and creating strategies to bridge any skills or resource gaps.

According to Bulla and Scott (1987), SWP can be described as the means to ensure that an organization's staffing requirements are met in a timely and optimal manner. It involves forecasting labor needs and determining when and how these will be fulfilled through hiring, training, or other personnel management strategies [6].

An essential component of SWP is the detailed analysis of both the current and future workforce. In the book "Strategic HR," P. Reilly and T. Williams emphasize that SWP is not a static process but a dynamic one, responsive to changes in the political, economic, and social environment. It is fundamental for addressing challenges such as demographic shifts, technological innovation, and fluctuations in the labor market [31].

The article "Strategic Workforce Planning in Healthcare: A Multi-Methodology Approach" [44] highlights how *horizon scanning* is the first essential step in strategic planning. This involves a systematic exploration of emerging factors that could impact the future workforce. This methodology helps identify risks, opportunities, and challenges by analyzing variables such as demographic trends, technological innovations, social and economic changes, and political and environmental transformations. Horizon scanning aids in detecting weak or early signals of change, which can be used to create plausible scenarios for the future of the workforce and to guide strategic planning [43].

Horizon scanning goes beyond simply identifying current trends; it also aims to build a systematic understanding of how the future might unfold, including both predictable and unexpected scenarios. Through this process, SWP gains a forward-looking perspective that helps organizations plan more resiliently and in an informed manner. The importance of this methodology lies in its ability to provide strategic insights useful for defining robust policies that can withstand a wide range of future uncertainties.

In summary, SWP represents a key tool to ensure that organizations are prepared to meet future needs, adapting to both external and internal transformations. Through the application of forecasting analyses, modeling techniques, and stakeholder engagement, SWP not only enhances operational efficiency but also supports the achievement of long-term strategic goals.

2.2 Main Challenges

Strategic Workforce Planning is a fundamental process to ensure that organizations can address evolving operational and strategic needs. However, the effective implementation of this process is complex and requires the management of significant challenges. A clear example of this can be seen in the Irish

public sector, where a reorganization of human resources became necessary to cope with significant staff reductions and to ensure that critical services could continue operating despite workforce shortages. The 12% reduction in the total number of public sector employees from 2008 to 2015 required a strategic approach to workforce planning, with a particular focus on internal reallocation and the restructuring of operational processes [28].

Specifically, three key factors contribute to the complexity of workforce planning: project variability, organizational constraints, and the skills and levels of expertise of the personnel. Let us examine each of these elements in detail to understand how they influence the success of Strategic Workforce Planning.

- **Project Variability**

Project variability represents one of the most critical challenges for workforce planning, especially in environments characterized by frequent shifts in priorities and operational requirements, as well as unpredictable fluctuations in the volume and nature of work. This phenomenon is particularly evident in the consulting sector, where each project has unique characteristics, requiring continuous skill updates and a strong capacity for adaptation.

One of the main issues lies in the difficulty of formalizing and quantifying the skills required for each project. The necessary skills vary based on multiple factors, such as specific client needs, regulatory changes, and emerging market trends. This heterogeneity makes it challenging to accurately plan human resources, as required competencies can differ significantly even between projects within the same industry.

Managing a diverse project portfolio further amplifies these difficulties. For example, a team of consultants may be asked to work on a project focused on optimizing complex Enterprise Resource Planning (ERP) systems, and then move on to initiatives in areas such as digital transformation, cybersecurity, artificial intelligence, big data analytics, or solutions based on emerging technologies like blockchain and the Internet of Things (IoT). This variability requires expertise spanning across multiple disciplines, such as engineering, marketing, law, finance, and management, making a multidisciplinary approach essential.

An additional challenge lies in balancing specialized expertise with the need for rapid adaptability. While experience enables more effective handling of complex projects, adaptability is crucial to respond quickly to highly variable operational, technological, and timing requirements. In this context, consulting firms must implement strategies that promote continuous skills development through targeted training programs and the integration of resources capable of flexibly tackling ever-changing challenges.

The key to addressing project variability lies in optimizing the workforce's adaptability. This goal can be achieved not only through continuous technical and sector-specific skills enhancement but also by adopting tools and methodologies that allow for the definition and quantification of the skills required for each project. In this way, organizations can significantly improve the planning and allocation of resources, ensuring greater efficiency and competitiveness.

- **Organizational Constraints**

Organizational constraints, such as government regulations, budget restrictions, and corporate culture, represent another significant obstacle for workforce planning. In the public sector, these constraints are particularly pronounced. For example, in Ireland, the centralized control of staffing levels and expenditures by the Department of Public Expenditure and Reform has limited the flexibility of individual organizations to respond to changes in strategic priorities.

Budget constraints often force organizations to "do more with less." This requires not only cost reduction but also increased efficiency. However, achieving both objectives can be extremely challenging. International experiences show that workforce reductions, if not strategically planned, can result in the loss of critical skills and a decline in employee morale. For instance, in the United

States, staff cuts during the 1990s negatively impacted the ability of some federal agencies to meet their goals due to the loss of key human capital [15].

Another significant constraint is organizational culture, which can influence the acceptance and support of workforce planning initiatives. Effective implementation requires the active involvement of leadership and clear communication to gain buy-in from stakeholders. Without this, the plan risks being perceived as a bureaucratic exercise rather than a strategic tool for organizational success.

Private sector companies face similar challenges, especially in highly regulated environments such as the financial or pharmaceutical industries. Here, regulatory constraints may impose restrictions on roles that require specific certifications or mandatory continuous training. Additionally, the need to comply with regulatory standards can limit flexibility in resource reallocation. To overcome these obstacles, many organizations are investing in workforce analytics technologies to monitor available skills in real time and align them with regulatory requirements.

- **Skills and Levels of Expertise**

Employee skills and levels of expertise are central to workforce planning, as they determine the organization's ability to meet current and future needs. However, identifying and developing the required skills is a complex challenge. This is especially true in dynamic environments such as the technology sector, where the required competencies are numerous: each role may demand a mix of *technical skills*, *soft skills*, *prior experience*, and *specialized knowledge*. Additionally, there is an issue related to the standardization of skills, which are not always easily measurable or comparable. For example, two employees with the same job title may have very different levels of experience and capabilities, which may not be immediately evident without an in-depth analysis.

A notable example is the Irish Public Service [28], where an aging workforce has highlighted the risk of losing institutional knowledge and critical skills. To address this challenge, some organizations have implemented innovative strategies such as mentoring programs, job rotation, and continuous training to ensure that key competencies are transferred to new generations of employees.

Another crucial aspect, after mapping existing skills, is the need to *forecast* future ones. This process requires reliable data and advanced analytical tools. However, many organizations struggle with a lack of complete and up-to-date data, making strategic planning difficult. Furthermore, creating a skills inventory is only the first step; it is equally important to translate this information into concrete actions, such as succession planning and the development of targeted training programs.

In the technology sector, where innovation moves quickly, companies are constantly required to train their employees on new platforms and tools. For example, the adoption of Machine Learning and Artificial Intelligence technologies demands specialized skills that are often not readily available within the organization. To bridge these gaps, many companies are adopting *Continuous Learning Models*, combining online courses, internal hackathons, and collaborative research projects. Additionally, engaging in partnership networks with universities and academic institutions is emerging as an effective strategy to attract and develop highly qualified talent. Nevertheless, this challenge remains open and significant. Despite the efforts made, many organizations still face difficulties in fully closing these skills gaps. The rapid pace of technological change continues to outstrip the speed at which new competencies can be developed and integrated, making it essential to further strengthen long-term strategies for workforce development.

2.3 Traditional Approaches

Historically, strategic workforce planning has relied on a combination of managerial experience and traditional analytical tools, such as spreadsheets and trend analysis. However, the increasing complexity of organizations and external challenges has rendered these methods insufficient to meet current needs. As a result, there has been a growing adoption of more advanced approaches that leverage both *mathematical models* and *computer algorithms* to support decision-making processes.

Among the most widely used approaches for SWP, two major categories stand out: *Heuristic Methods* and *Linear Programming-based methods*. Both represent powerful tools for addressing workforce planning challenges, but they differ significantly in their core principles, techniques, and application contexts. The choice between one approach or the other depends on various factors, including the nature of the problem, the availability of data, the time available for analysis, and the organization's ability to implement and manage complex models. In many cases, companies combine elements from both approaches to achieve the best possible outcomes. For example, a Linear Programming model may be used to define a baseline solution, which is then refined through Heuristic techniques to adapt to specific operational needs or unforeseen changes.

In the following paragraphs, both Heuristic and Linear Programming-based approaches will be explored in greater depth, highlighting their core principles, benefits, and application contexts. This analysis will help illustrate how these methodologies can be effectively employed to address the complex challenges associated with strategic workforce planning, while also identifying their limitations.

2.3.1 Heuristic Approaches

Heuristic Approaches are problem-solving methods that rely on practical rules, insights, or approximate search strategies to find acceptable solutions within a reasonable timeframe, without guaranteeing a globally optimal result. They are particularly useful for complex problems with large or poorly defined solution spaces, where exact methods become impractical.

Let's look at some examples of Heuristic Approaches applied to the implementation of algorithms for human resource management:

- **Greedy Algorithms:** These focus on making locally optimal choices at each step, with the aim that these will lead to an acceptable global solution. For example, in an SWP context, a Greedy Algorithm might assign the best available employee to a high-priority project without considering future implications.
- **Genetic Algorithms:** Inspired by natural selection, genetic algorithms start with a population of random solutions and use biologically inspired operations to evolve them over time. The concept of genetic algorithms was introduced by American scientist J. H. Holland in his book "*Adaptation in Natural and Artificial Systems*" [14]. The main steps in defining a genetic algorithm are as follows:
 1. *Initialization:* Creation of an initial population of solutions.
 2. *Evaluation:* Each solution is evaluated using an objective function.
 3. *Selection:* The best solutions are selected for reproduction.
 4. *Crossover and Mutation:* The selected solutions are combined and modified to generate new solutions.
 5. *Iteration:* The process continues until a stopping criterion is met, such as a maximum number of iterations or a performance threshold.
- **Simulated Annealing:** Based on physical processes such as metal annealing, this method temporarily accepts worse solutions to avoid local minima and improve solution quality over the long term. The idea behind Simulated Annealing was first proposed in 1983 by Scott Kirkpatrick,

C. Daniel Gelatt, and Mario P. Vecchi in their article "*Optimization by Simulated Annealing*" [20].

1. *Initialization*: Starting from an initial solution.
 2. *Local Search*: A new solution is generated in the neighborhood of the current one.
 3. *Acceptance*: Worse solutions may be accepted with a probability that decreases over time.
 4. *Stopping*: The process ends when the "temperature" falls below a predefined threshold.
- **Tabu Search**: This method uses a list of forbidden solutions to avoid cycles and encourage exploration of new areas in the solution space, making it useful for planning problems with many interdependent constraints. The concept of Tabu Search in the field of Heuristic Algorithms was introduced by Fred W. Glover in 1986[11].
 1. *Iterative Search*: New solutions are generated while ignoring those on the tabu list.
 2. *Update*: The tabu list is updated at each iteration to ensure diversity in the solutions explored.

In conclusion, heuristic approaches have established themselves as one of the most widely used solutions for addressing the complex problems of workforce planning, particularly in Strategic Workforce Planning (SWP). Their applicability in real-world scenarios makes them highly valued tools in the business world. However, as with any technique, they come with both advantages and limitations.

Advantages of Heuristic Approaches

Adaptability: One of the key strengths of Heuristic Approaches is their adaptability to specific contexts and situations. While exact methods require a detailed understanding of the problem and its constraints to be correctly applied, Heuristic Algorithms can be easily modified to address new challenges or changes in working conditions. In the context of SWP, this means that solutions can be quickly adjusted in response to labor market fluctuations, variations in required skills, or changes in business requirements.

Computational Efficiency: Heuristic algorithms are known for their computational efficiency, particularly in scenarios where the problem to be solved is large or complex. In many cases, solving a workforce planning problem with exact methods is computationally prohibitive, especially as data volume and complexity increase. While Heuristic Approaches do not guarantee an optimal solution, they can provide satisfactory solutions in relatively short time frames, significantly reducing the computational burden compared to exact optimization methods.

Uncertainty Management: Another fundamental advantage of Heuristic Approaches is their ability to handle uncertainty. In business contexts, SWP is influenced by multiple unpredictable factors; Heuristic Algorithms, especially those like tabu search or genetic algorithms, can explore various solutions in uncertain and dynamic environments, finding reasonable answers even in the absence of complete data or in the presence of variability.

Limitations of Heuristic Approaches

Lack of Guaranteed Optimality: Although Heuristic Approaches are capable of finding practical solutions, one of their main limitations is that they do not guarantee optimality. In other words, while exact algorithms can ensure finding the best possible solution, Heuristic Algorithms settle for "good" solutions that may not be optimal. In the context of SWP, this could result in workforce planning that

does not fully maximize the organization's potential but is still sufficiently effective to meet business requirements.

Dependence on Parameters: Another critical aspect concerns dependence on initial parameters. The performance of a Heuristic Algorithm is strongly influenced by input configurations, such as the parameters used to guide the search (e.g., the number of iterations or solution selection parameters). The choice of these parameters can significantly impact the quality of the final solution. Consequently, finding optimal parameters may require extensive experimentation, increasing the time needed for the practical application of the algorithm, especially in situations where planning constraints or objectives are complex and variable.

2.3.2 Linear Programming

Linear Programming (LP) is a mathematical optimization technique that seeks the optimal solution to a problem by expressing the objective and constraints as linear functions. LP algorithms operate within a solution space defined by linear constraints and provide the best possible solution (if one exists) for well-defined problems. A fundamental concept underlying many Linear Programming algorithms is the "*Feasible Region*". This represents the set of all possible solutions that satisfy the constraints of an optimization problem. It is often geometrically described as a manifold in the decision variable space, defined by the intersection of a series of half-planes or hyperplanes derived from linear constraints. This geometric and convex region guides the behavior of LP algorithms: while some, like the *Simplex Method*, navigate the vertices, others, such as *Interior Point Methods*, explore the interior. Below, we outline these methods in detail:

- **Simplex Method:** The Simplex Method is one of the most well-known and widely used algorithms in Linear Programming. Introduced by George Dantzig in 1947, it identifies the optimal solution, if one exists, at one of the vertices of the feasible region.[8]
 1. Definition of the *objective function* and constraints in standard form.
 2. Identification of an initial feasible *basic solution* that satisfies the defined constraints.
 3. Construction of a *Simplex tableau*, a structured matrix containing the coefficients of the objective function, constraints, and introduced variables.
 4. This tableau is used to iteratively move toward the *optimal solution*.
- **Interior Point Methods (IPM):** Interior Point Methods are iterative algorithms that explore interior points of the feasible region rather than its vertices. Introduced as an alternative to the Simplex Method, they are particularly suited for large-scale problems.[17]. Unlike the Simplex Method:
 1. Starts from an *interior point* within the feasible region, not necessarily on a vertex.
 2. No tableau is used; instead, systems of *nonlinear equations* are solved iteratively to determine the direction of movement.
 3. Convergence is guaranteed in *polynomial time* and can be extended to nonlinear optimization problems.

Advantages of Linear Programming Approaches

Guaranteed Optimization: When the problem is well-formulated and the system is consistent (the constraints are feasible), Linear Programming guarantees an optimal solution. This feature is a significant advantage when achieving the best possible resource allocation is critical.

Transparency: The formulation of linear programming is clear and structured, enabling precise modeling of constraints (e.g., limitations on working hours, specific skills required for projects, etc.) and objectives (e.g., minimizing costs, maximizing productivity). This clarity facilitates understanding the solution and validating the decision-making process.

Suitable for Simple Models: If the problem is relatively straightforward (e.g., with few variables and linear constraints), linear programming is an effective and powerful approach, even when dealing with large datasets.

Limitations of Linear Programming Approaches

Linearity of Models: Linear Programming assumes that the relationship between decision variables (e.g., number of employees, working hours) and the objective is linear. However, many real-world problems in Strategic Workforce Planning involve non-linearities. For example, the marginal return of skills or labor costs may not follow a linear pattern.

Rigid and Non-Representative Constraints: The linear constraints of LP are relatively rigid. In many real-world scenarios, constraints can be more complex (e.g., non-linear work schedule limitations, variable availability depending on the time of year, complex interactions between skills). Consequently, Linear Programming may oversimplify reality, leading to impractical solutions.

Limited Adaptability: Strategic Workforce Planning often involves dynamic changes, such as project modifications, new hires, or variations in employee behaviors (turnover, sick leave, vacations). Linear Programming can be inflexible in responding quickly to such changes. Adapting to these dynamics requires continuous updates to the model, which can be costly and time-consuming.

Characteristics	Linear Programming	Heuristic Methods
Optimization	Optimal if linear	Good solutions but not optimal
Flexibility	Limited to linear models	Suitable for complex models
Complexity	High for complex problems	More efficient
Adaptability	Low adaptability	Highly adaptable to changes
Implementation	Easy if linear	Simple, less transparent
Accuracy	Precise solutions	Approximate solutions
Scalability	Difficult with many variables	Excellent for large problems
Constraints	Only linear	Suitable for complex constraints
Interpretability	Easy to interpret	Solutions hard to interpret

Table 2.2: Comparison between traditional methods

Chapter 3

Reinforcement Learning

3.1 General Overview

Reinforcement Learning (RL) stands out as a distinctive paradigm within the domain of machine learning, focusing on goal-oriented learning through interaction. Rooted in the dynamic relationship between agent and environment, RL provides a computational framework for learning to make decisions that maximize cumulative rewards over time. This chapter delves into the historical evolution of reinforcement learning and its distinctions from other machine learning paradigms, offering a comprehensive view of its position within the broader research landscape of machine learning.

Historical Evolution of Reinforcement Learning

The conceptual foundations of Reinforcement Learning trace back to the early 20th century, spanning fields such as psychology, neuroscience, and cybernetics. Psychologists like Edward Thorndike and B.F. Skinner explored the principles of learning based on rewards and punishments, formulating theories such as operant conditioning [24]. These ideas laid the groundwork for understanding how behaviors can be shaped through reinforcement signals.

In the 1950s, the development of dynamic programming by Richard Bellman introduced mathematical tools central to RL. Bellman's "Principle of Optimality" provided a framework for solving decision-making problems where outcomes depend on sequences of actions, a pivotal insight for RL algorithms [2],[3]. Concurrently, advancements in cybernetics and control theory, particularly adaptive control systems, enriched the theoretical basis of RL.

The 1980s marked a resurgence of interest in RL, fueled by progress in computational methods and an increasing awareness of its potential applications in Artificial Intelligence. Researchers such as Andrew Barto, Richard Sutton, and others formalized RL as a distinct field, separate from supervised and unsupervised learning [39]. During this period, foundational algorithms such as Temporal Difference (TD) learning and Q-learning emerged, offering practical methods for learning from delayed rewards. These developments established the groundwork for modern RL and highlighted its unique capabilities in addressing sequential decision-making problems.

The advent of the 21st century ushered in a new era for RL, characterized by its integration with deep learning. Deep Reinforcement Learning, popularized by systems like Deep Q-Networks (DQN) and applications such as AlphaGo, demonstrated the ability of RL agents to achieve superhuman performance in complex domains such as board games, video games, and robotics [26],[36]. This synergy between RL and deep learning spurred a wave of research and applications, solidifying RL's role as a cornerstone of modern artificial intelligence.

Reinforcement Learning vs Other Learning Paradigms

Reinforcement Learning (RL) occupies a unique position in the landscape of Machine Learning paradigms, distinguished by its emphasis on sequential decision-making and interaction with dynamic environments. To better understand its distinctiveness, it is crucial to compare RL with the two traditionally dominant paradigms: *Supervised Learning* and *Unsupervised Learning*.

Supervised Learning

Supervised learning is characterized by its reliance on labeled datasets, where each input is paired with a corresponding output or label. The goal in Supervised Learning is to generalize from the training data by minimizing the error between predictions and true labels [7]. For example, a supervised learning model might classify emails as spam or not spam based on historical examples.

In contrast, RL operates without labeled examples. Instead, it relies on a reward signal to evaluate the quality of an agent's actions. This lack of explicit supervision allows RL to address problems where labeled data is unavailable or impractical to obtain. In RL, the agent must explore the environment and learn from the consequences of its actions, discovering strategies that maximize cumulative rewards over time. This exploratory nature introduces challenges such as delayed feedback and the need to balance exploration and exploitation, making RL fundamentally different from Supervised Learning.

Additionally, RL systems are inherently closed-loop, meaning the agent's actions influence subsequent inputs and outcomes. In Supervised Learning, the training process is static and does not involve such dynamic interactions. This distinction highlights RL's suitability for problems involving sequential decisions, where the goal is to optimize long-term outcomes rather than immediate accuracy.

Unsupervised Learning

Unsupervised Learning focuses on identifying patterns and structures within unlabeled data. Techniques like clustering and dimensionality reduction aim to uncover hidden relationships or simplify data representations[10]. For instance, Unsupervised Learning might group customers based on purchasing behaviors to enable targeted marketing strategies.

While RL also operates without labeled data, its objective is fundamentally different. Rather than uncovering patterns, RL seeks to maximize a reward signal through strategic decision-making. The primary task of an RL agent is to learn an optimal policy—a mapping from states to actions—that yields the highest cumulative reward. This goal-oriented nature sets RL apart from Unsupervised Learning, positioning it as a distinct paradigm within machine learning.

Although RL is distinct from supervised and unsupervised learning, it often interacts with these paradigms in practical applications. For instance:

- Supervised Learning techniques can approximate value functions or policies in RL, leveraging labeled data to improve sample efficiency.
- Unsupervised Learning methods can aid in feature extraction and representation learning, enhancing the agent's ability to perceive and interpret the environment.

These synergies underscore the complementary nature of RL and other machine learning paradigms, fostering interdisciplinary innovation.

The unique characteristics of Reinforcement Learning, including its emphasis on interaction, sequential decisions, and reward maximization, make it a distinct paradigm in the field of Machine Learning. RL bridges the gap between static learning models and adaptive real-time systems, enabling agents to operate effectively in complex and uncertain environments. Its versatility and applicability across diverse domains highlight its significance as an independent area of research and application.

Practical Applications of Reinforcement Learning

Reinforcement learning has demonstrated remarkable success across a variety of domains. Some notable applications include:

1. **Gaming:** RL has achieved superhuman performance in games such as chess, Go, and video games, with systems like AlphaGo and Deep Q-Networks (DQN) [36].
2. **Robotics:** RL enables robots to learn tasks such as navigation, manipulation, and locomotion by interacting with their physical or simulated environments[18].
3. **Finance:** Applications include portfolio optimization, algorithmic trading[13], and risk management.
4. **Healthcare:** RL has been used for personalized treatment planning, drug discovery, and optimizing medical interventions[45].
5. **Autonomous Systems:** From self-driving [19] cars to drones, RL provides a framework for decision-making processes in complex and uncertain environments.

Reinforcement Learning represents a powerful framework for learning through interaction, guided by the goal of maximizing cumulative reward. Its unique focus on sequential decision-making and its distinction from supervised and unsupervised learning highlight its importance as an autonomous paradigm. With roots in various disciplines and applications spanning many fields, RL continues to push the boundaries of what machines can learn and achieve through experience.

3.2 Basic Concepts

The central elements that characterize reinforcement learning can be described as follows:

- **Agent**

The "*agent*" is the entity that learns to make optimal decisions through its experiences. It is the active component of the RL system, which, through an iterative process, acquires knowledge that progressively improves its performance. To draw an analogy with the animal kingdom, the agent can be compared to a **young lion**, who, as a beginner, must learn how to hunt.

This element can be implemented in different ways, depending on the complexity of the problem and the computational resources available. For example, in simpler cases, it may consist of a basic algorithm following a predefined strategy, while in more complex scenarios, it can use deep neural networks to learn highly sophisticated policies. The agent is the core of the entire learning process, as the entire system is designed to enable it to improve its behavior over time.

An important aspect in the design of the agent is its ability to generalize from situations already observed to new situations. This requires Advanced Learning techniques, such as Knowledge Transfer and the use of approximation functions. Furthermore, a well-designed agent must be capable of handling situations where data is noisy or incomplete, ensuring robustness and reliability.

- **State**

The concept of state is central in reinforcement learning. Each state represents a specific configuration of the context in which the agent is operating at a given moment. States therefore provide the agent with the necessary information to make informed decisions. For the young lion who must learn to hunt, a state is described by a combination of variables that represent the agent's situation at the current moment. Some examples of such variables could be:

- *Position of the cub:* where it is in the territory,
- *Position of the prey:* whether it is visible, distant, or hidden,

- *Energy level*: how much energy it has available to run.

Formally, a state s is an element of a set of states \mathcal{S} , which constitutes the state space. Each state $\mathbf{s} \in \mathcal{S}$ contains all the necessary information to determine the possible actions the agent can take. Mathematically, a state can be represented as a vector of features

$$\mathbf{s} = (s_1, s_2, \dots, s_n), \quad (3.1)$$

where each s_i is a variable describing a specific aspect of the environment.

States can be either observable or partially observable. In the first case, the agent has complete access to all the information necessary to describe its surroundings. However, in more complex situations, the agent might only have access to part of the information, which leads to the problem of partially observable states. In these cases, the agent must rely on more sophisticated strategies, such as using internal memory or applying probabilistic inference techniques.

The definition of the state is crucial to the success of a Reinforcement Learning algorithm. A state representation that is too detailed might make learning inefficient, while a representation that is too simplistic might not provide the agent with enough information to make optimal decisions. For example, in complex systems, states can be described by high-dimensional vectors or hierarchical structures. In these cases, dimensionality reduction techniques or representation learning can be used to simplify the learning process without losing critical information.

• Action

Actions represent the commands that the agent can execute. Each state offers a set of available actions, which constitutes the action space.

Returning to the aforementioned analogy, actions correspond to a series of behaviors that the cub can perform in an attempt to catch prey, such as:

- *Approaching the prey*,
- *Hiding*,
- *Jumping*.

An action represents a decision or a move that the agent can make within the environment. Formally, an action $\mathbf{a} \in \mathcal{A}$ is an element of a set of actions \mathcal{A} and can be represented as a feature vector:

$$\mathbf{a} = (a_1, a_2, \dots, a_n), \quad (3.2)$$

where each a_i is a variable describing a specific aspect of the decision made by the agent.

The idea behind this paradigm is that each type of action impacts the next state in a different way.

It follows that actions can be divided into two important categories: discrete or continuous actions, depending on the problem. In discrete cases, $\mathcal{A} = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k\}$, the agent chooses from a finite number k of options, such as moving in one of the four directions in a maze. In continuous problems, however, $\mathcal{A} = \{\mathbf{a} \in \mathbb{R}^n\}$, the agent can select a value within a range, such as the steering angle of an autonomous vehicle.

• Environment

The environment constitutes the context in which the agent operates and provides the necessary feedback for learning. The environment is often described as a function that, given a current state and an action taken by the agent, returns a new state and an associated reward. In other words, the environment responds to the agent's actions and determines the consequences.

In the specific example of a lion cub, the environment represents everything around it and with which it can interact:

- *The territory*: The geographical context in which the lion cub is situated, which might include different areas, such as a Savannah, a jungle, or an open area with plants and trees. The characteristics of the environment will influence the decisions the agent can make (for example, a densely vegetated area might be a good place to set an ambush).
- *The type of prey*: For example, large prey might require more complex actions than small prey.
- *Resources and obstacles*: These can affect the cub’s energy levels by providing better prospects for future actions.

Formally, let \mathcal{S} be the state space, and \mathcal{A} be the action space. The environment is a dynamic system described by::

- **Probabilistic transition function:**

$$P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$$

where $P(s'|s, a)$ represents the probability that the system transitions to the next state $s' \in \mathcal{S}$, given the current state, $s \in \mathcal{S}$ and the action $a \in \mathcal{A}$

- **Reward function:**

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

which associates a reward $r \in \mathbb{R}$ with a transition characterized by the initial state s , the action a , and the subsequent state s' . It represents a measure of success or failure of the action in relation to the overall goal.

A crucial aspect of the environment is its dynamism. State transitions can be determined by fixed rules or probabilistic processes. In the latter case, the dynamics of the environment are modeled through probability distributions $P(s'|s, a)$ that describe the possible transitions. This introduces an element of uncertainty that the agent must deal with during learning.

Rewards, on the other hand, can be immediate or delayed. An immediate reward is received right after an action is performed, while a delayed reward might depend on a sequence of actions. For example, in a turn-based game, the final reward may only be assigned at the end of the game, requiring the agent to evaluate the contribution of each action to the overall outcome.

Another important characteristic of the reward is its scalability. The agent receives a single numerical value for each action, regardless of the complexity of the environment. This simplicity is both a strength and a challenge: the reward provides a direct signal for learning but does not explicitly indicate how to improve. Therefore, the agent must explore the environment and deduce which actions lead to the desired rewards.

Designing the reward function is often one of the biggest challenges in reinforcement learning. A poorly defined reward function can lead to undesirable or suboptimal behaviors, while a well-designed function can significantly accelerate the learning process.

- **Policy**

The policy is the strategy adopted by the agent: it is the element that guides the agent’s behavior, indicating which action to choose in each state to maximize the cumulative reward in the long run. It is represented by a function that maps the states of the environment to the actions that the agent must take in each of them.

A fundamental concept is to note that the policy should not be confused with or identified with the actions themselves. While actions can be seen as the ingredients that the agent always has available but is not required to use at every moment, the policy is the actual recipe for achieving the goal, often consisting of a specific combination of those ingredients. For example, returning to the analogy with the animal kingdom:

- If the cub is *far from the prey* and the prey is unaware of its presence, the policy might suggest slowly *approaching*.
- If *the prey is aware* of the cub's presence, the policy might suggest *hiding* to avoid danger.

This example clearly highlights the relationship between the states of the environment and the actions to take accordingly.

Formally, there are two main types of policies: deterministic and stochastic. A *deterministic policy* assigns a specific action to each state, so the agent will always follow the same action for a given state:

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

On the other hand, a *stochastic policy* assigns a probability distribution over which action to take, allowing the agent to explore different possibilities in each state with a certain probability:

$$\pi(a|s) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$$

The goal of an agent in a reinforcement learning (RL) environment is to learn an optimal policy π^* , i.e. the one that maximizes the total sum of expected rewards over the long run.

• Value Function

The last fundamental element we introduce is the value function. It quantifies *how good* it is to be in a certain state or to perform a certain action. There are two main types of value functions:

1. State Value Function $V(s)$: Measures the expected value of being in a state s while following a policy π . Formally, it is the expectation of the cumulative future rewards starting from state s .

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s \right] \quad (3.3)$$

2. State-Action Value Function $Q(s, a)$: Evaluates the expected value of performing an action a in state s , while subsequently following policy π . It is defined as:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s, A_0 = a \right] \quad (3.4)$$

The value function can be seen as the *"sense of how advantageous"* it is to be in a certain situation or to perform a certain action in order to reach the final goal.

1. $V(s)$: The cub might perceive that being hidden in tall grass near a herd of antelopes is a favorable situation because it increases its chances of success in the future. The value function helps answer the question: *"How close am I to success in this situation?"*
2. $Q(s, a)$: If the cub is hidden in the grass (state s) and evaluates different actions, such as *"run towards the herd"* or *"stay still"*, the value $Q(s, run)$ might indicate that running is a better choice than staying still at that moment. Similarly, the state-action function answers the question: *"If I take this action, will I improve or worsen my chances of success?"*

3.2.1 Exploration vs Exploitation

Two concepts strongly tied to the system definition constraints that characterize the Reinforcement Learning (RL) paradigm are *Exploration* and *Exploitation*. These concepts were introduced through the *multi-armed bandit* problem [32] and later developed and formalized in the field of RL by Sutton and Barto [38]. They represent two fundamental components in the agent’s decision-making process and are a key compromise for the success of learning.

Exploration

Exploration involves actively searching for new actions or strategies that might improve the agent’s long-term performance. During this phase, the agent selects less familiar actions, even at the cost of receiving lower immediate rewards, in order to gather information about the environment and identify potentially better solutions. This process is essential to avoid the model becoming stuck in a *local optimum*, thereby limiting its ability to reach the global optimal solution.

A simple and intuitive example can be found in any learning game, such as tic-tac-toe. In this case, exploration could lead the player to make moves that don’t immediately seem advantageous, such as placing a mark in an apparently useless corner, to understand how the opponent reacts to that move. This behavior could reveal winning strategies that were not obvious at first.

Exploitation

Exploitation, on the other hand, focuses on using the knowledge already acquired to make decisions that immediately maximize the reward. The agent chooses actions that, based on accumulated experience, seem to guarantee the best possible result at the present moment. Referring again to the tic-tac-toe example, exploitation would involve continuously repeating moves that have already proven to lead to victories, such as occupying the center of the board to maximize the chances of forming a line.

The tradeoff between exploration and Exploitation is crucial for the functioning of Reinforcement Learning. On one hand, excessive Exploration may lead to inefficiency, with the agent continuing to test suboptimal configurations even after discovering good solutions. On the other hand, premature exploitation may prevent the discovery of globally optimal or more effective solutions in dynamic conditions.

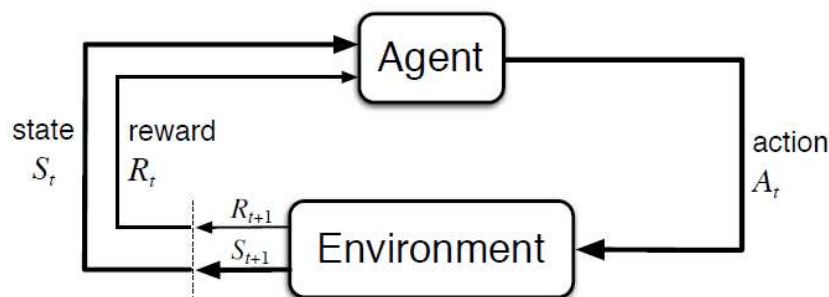


Figure 3.1: The agent-environment interaction in reinforcement learning.

In conclusion, to summarize, we can identify **Reinforcement Learning** as a learning paradigm in which an agent learns to make optimal decisions by interacting with a dynamic environment. The agent observes the current **state** of the environment, $s \in \mathcal{S}$, takes an **action** $a \in \mathcal{A}$, and receives a reward $R \in \mathbb{R}$. After the action, the environment evolves into a new state $s' \in \mathcal{S}$, determined by the system’s dynamics, which are described by a **transition function** $P(s' | s, a)$. Through this iterative cycle, the agent gradually learns to maximize the sum of **rewards** over time. The strategy that guides the agent in choosing actions is called the **policy**, $\pi(a | s)$, and it specifies the probability of selecting an action a given the state s . At the same time, the agent builds a **value function**, $V^\pi(s)$ or $Q^\pi(s, a)$,

which evaluates how advantageous it is to be in a certain state or perform a specific action while following the policy π . The agent’s ultimate goal is to discover an **optimal policy**, denoted as π^* , that maximizes the expected cumulative reward.

3.3 Fundamental Algorithms

Reinforcement learning has become a central area in today’s artificial intelligence landscape, providing both a theoretical and practical framework for solving decision-making problems in dynamic and uncertain environments. Before introducing the *Proximal Policy Optimization* (PPO) algorithm, which represents the approach adopted in this thesis, it is essential to outline the key concepts and algorithms that form its foundation. In particular, we will present *Q-learning* and its extension, *Deep Q-learning*, which are examples of value-based methods for solving reinforcement learning problems. Subsequently, *policy gradient methods* will be introduced, an alternative paradigm that, compared to value-based methods, focuses on directly learning the optimal policy. This progression will allow us to understand the motivations and advantages that led to the development of more advanced and stable algorithms like PPO, which effectively combine the strengths of previous techniques while addressing some of their most relevant limitations.

3.3.1 Q-Learning

The Q-learning algorithm is fundamental in the field of Reinforcement Learning (RL) and was introduced by Watkins and Dayan in 1992 [41]. The core idea behind its operation is to learn a function, commonly called the *Q-function*, that associates a value to each state-action pair, representing how useful it is to perform that specific action in a given state. This value, called $Q(s, a)$, reflects the maximum cumulative reward that the agent can expect to obtain in the long term, starting from that state and choosing future actions optimally. It is worth noting that the value-action function $Q(s, a)$ was not first introduced with Q-learning. It originates from previous work on dynamic programming and concepts developed by Richard Bellman in the 1950s [2].

The key contribution of Q-learning was:

1. Formalization and use of the $Q(s, a)$ function: Watkins used the $Q(s, a)$ function as a basis to learn the optimal action values directly without requiring an explicit model of the environment.
2. Iterative update algorithm: The Q-learning update equation, based on the off-policy approach, allows the calculation of $Q(s, a)$ incrementally through direct experiences, without prior knowledge of the environment’s dynamics.

One of the main features of Q-learning is that it does not require prior knowledge of the model of the environment with which the agent interacts, making it particularly suitable for problems where the environment’s dynamics are unknown or complex.

To begin working with Q-learning, an initial table, called the *Q-table*, is defined, which contains the $Q(s, a)$ values for all possible combinations of states s and actions a . Typically, these values are initialized to zero or some arbitrary number. Subsequently, the agent begins interacting with the environment: it observes a state s , chooses an action a according to a decision policy (for example, an ϵ -greedy policy, which alternates between exploration and exploitation), performs the action, and receives a reward R and a new state s' as feedback from the environment. With this information, it updates the value $Q(s, a)$ in the table using an update rule based on the following fundamental equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [R + \gamma \max_{a'} Q(s', a') - Q(s, a)]. \quad (3.5)$$

In this formula:

- α is the *learning rate*, which controls how quickly the Q values adapt to the new data.

- γ is the *discount factor*, which determines how much weight is given to future rewards compared to immediate ones.
- $\max_{a'} Q(s', a')$ represents the maximum estimated value for the new state s' , considering all possible actions a' .

This process is repeated iteratively. The agent continuously explores the environment, updating the Q-table and progressively improving its understanding of which actions are most advantageous in each state. As the agent explores the environment sufficiently and the policy stabilizes, the Q-table converges to the optimal value $q_*(s, a)$, which is the maximum expected value for a given state-action pair. In Alg.[1] an example of a Q-learning algorithm is presented: SarsaMax.

To make this more concrete, let's consider a simple example: an agent that has to navigate a maze to reach an exit. The states correspond to positions in the maze, the actions are "up", "down", "left", and "right", and rewards are only assigned when the agent reaches the exit. Initially, the agent explores the maze randomly, updating the Q-values based on the rewards received. Over time, using the update rule, the agent learns which actions bring it closer to the exit, and eventually, it will be able to follow the optimal path based on the values in the Q-table.

Despite its simplicity and versatility, Q-learning has some limitations. One of the main ones is scalability: for environments with a very large number of states or actions, the Q-table becomes impractical to manage. Additionally, the algorithm requires an effective strategy to balance exploration and exploitation. Finally, in environments with continuous states or actions, traditional Q-learning is insufficient and must be extended, for example, by using neural networks to approximate the Q-function, as done in Deep Q-Learning (DQN).

Below is the pseudocode for the Q-learning algorithm, taken from Sutton and Barto's "*Reinforcement Learning: an introduction*" [38]. The structure of the algorithm highlights the main steps, including action selection according to an ϵ -greedy policy, calculating the action with the maximum value for the next state, and updating the Q function to improve estimates of action values based on observed rewards and expected future values.

Algorithm 1 Example of a Q-Learning algorithm (Sarsamax)

Require: Policy π , positive integer $num_episodes$, small positive fraction α , GLIE $\{\epsilon_i\}$

Ensure: Value function Q ($Q \approx q_\pi$ if $num_episodes$ is large enough)

```

1: Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in S$  and  $a \in A(s)$ , and  $Q(\text{terminal-state}, \cdot) = 0$ )
2: for  $i \leftarrow 1$  to  $num\_episodes$  do
3:    $\epsilon \leftarrow \epsilon_i$ 
4:   Observe  $S_0$ 
5:    $t \leftarrow 0$ 
6:   repeat
7:     Choose action  $A_t$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
8:     Take action  $A_t$  and observe  $R_{t+1}, S_{t+1}$ 
9:      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$ 
10:     $t \leftarrow t + 1$ 
11:  until  $S_t$  is terminal
12: return  $Q$ 

```

3.3.2 Deep Q-Learning

Deep Q-Learning represents one of the milestones in the field of Reinforcement Learning. Introduced by Volodymyr Mnih et al. in the paper *Playing Atari with Deep Reinforcement Learning* (2013) [25], this approach integrates classical Q-Learning with deep neural networks (Deep Neural Networks, DNN), enabling the handling of high-dimensional inputs such as images or complex data. As seen earlier, traditional Q-Learning suffers from significant limitations: for problems with large or continuous state

spaces, it is impractical to store a Q-table for every combination of s and a . Additionally, classical Q-Learning does not generalize between similar states, requiring explicit data for each state-action pair, and becomes inefficient as the dimensionality of the data increases.

Deep Q-Learning overcomes these limitations by introducing a deep neural network as a replacement for the Q-table in approximating the Q-function. In this context, a neural network, which can be convolutional (CNN) or fully connected, receives the state s as input and outputs the corresponding $Q(s, a; \theta)$ values for each possible action, where θ represents the network's parameters. The goal is to minimize the mean squared error between the network's output and the target Q, which is defined by the aforementioned Bellman formula (3.5).

The network is trained using a loss function $L(\theta)$, which measures the discrepancy between the predicted value $Q(s, a; \theta)$ from the main network (with parameters θ) and the target $r + \gamma \max_{a'} Q(s', a'; \theta^-)$, calculated using a secondary network called the *target network*:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim D} [(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2] \quad (3.6)$$

The target network is a frozen copy of the main network that is only updated periodically, not at every optimization step. This measure helps stabilize learning: indeed, if the main network were continuously updated and used both to calculate the target and to update the weights, the target itself would continuously change during optimization, leading to instability. In practice, the weights θ^- of the target network are updated by copying the weights θ from the main network only every N steps.

At the subscript of the expectation value in (3.6), we find the expression $(s, a, r, s') \sim D$, which represents one of the main innovations introduced by Deep Q-Learning: the replay buffer. This is a circular memory that stores past experiences in the form of tuples (s, a, r, s') . During training, minibatches of experiences are sampled to break the temporal correlation between samples and improve stability.

Finally, to balance exploration and exploitation, an ϵ -greedy policy is adopted, where with probability ϵ , a random action is selected; otherwise, the action with the maximum Q value is executed.

Algorithm 2 Deep Q-Learning with Experience Replay

- 1: Initialize replay memory D to capacity N
 - 2: Initialize action-value function Q with random weights θ
 - 3: Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$
 - 4: **for** episode = 1, M **do**
 - 5: Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
 - 6: **for** $t = 1, T$ **do**
 - 7: With probability ϵ select a random action a_t
 - 8: Otherwise select $a_t = \arg \max_a Q(\phi(s_t), a; \theta)$
 - 9: Execute action a_t in emulator and observe reward r_t and image x_{t+1}
 - 10: Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
 - 11: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
 - 12: Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
 - 13: Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
 - 14: Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
 - 15: **if** every C steps **then**
 - 16: Reset $\hat{Q} = Q$
-

Deep Q-Learning is, therefore, capable of handling complex inputs such as images, generalizing between similar states, and does not require explicit storage of the Q-table. In the aforementioned 2013 paper, Mnih et al. [25] applied DQL to various Atari 2600 games, achieving human-level performance

in several games, demonstrating the effectiveness of the method in learning complex strategies directly from raw pixels. However, Deep Q-Learning presents some challenges: the use of neural networks introduces instability due to correlations between samples and the continuous update of the targets, and the algorithm tends to overestimate Q values. Additionally, the performance strongly depends on hyperparameters such as ε , the learning rate, and the capacity of the replay buffer.

Among the most significant extensions of DQL are Double DQN, which reduces the overestimation bias by separating action selection and evaluation, Dueling DQN, which splits the network into two parts for the advantage function and the value function, and Prioritized Experience Replay, which samples experiences with a probability proportional to their relevance. In Alg.(16) we find an example pseudocode highlighting the main characteristics of Deep Q-Learning.

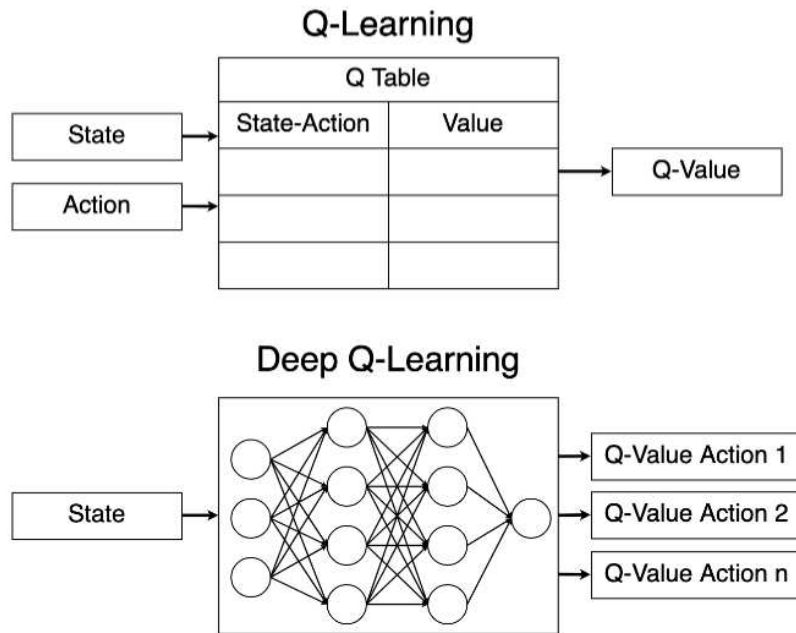


Figure 3.2: Comparison between Q-learning and Deep Q-learning from 29th Mediterranean Conference on Control and Automation [35].

3.3.3 Policy Gradient Methods (PGM)

In the previous sections, we introduced algorithms such as Q-Learning and Deep Q-Learning, whose goal is to learn a function $Q(s, a)$ that represents the quality of performing a certain action a in a state s . These methods aim to estimate the optimal value function $Q^*(s, a)$, from which the optimal policy π^* can be derived by choosing, for each state, the action that maximizes Q . However, it is important to emphasize that the general goal of *Reinforcement Learning* (RL) is not only to find the best Q function, but rather to learn the optimal behavior of the agent, represented by the policy π^* . In other words, Q is a means to achieve a broader goal: learning how to act optimally in an environment.

This distinction leads to a fundamental difference between two main approaches in RL: *Value-Based* methods, which focus on learning value functions such as $Q(s, a)$ or $V(s)$, and *Policy-Pased* methods, which aim directly at optimizing the policy without estimating any value function.

Value-Based Approaches

Value-Based methods, such as Q-Learning and Deep Q-Learning, estimate value functions that allow evaluating the quality of actions or states. These approaches work well in environments where states and actions are discrete and of moderate size, as they enable the explicit construction of a $Q(s, a)$

function and its use to derive the optimal policy:

$$\pi^*(a | s) = \arg \max_a Q^*(s, a). \quad (3.7)$$

However, Value-Based methods have some significant limitations:

- **Lack of an explicit policy:** They do not directly learn a parametric policy, making it difficult to apply them to complex or continuous problems, where searching for an optimal action can be intractable.
- **Dependence on a value function:** In some contexts, it is not guaranteed that a well-defined value function exists or is easily approximable, especially in environments with complex dynamics.
- **Uncertain state information:** In partially observable environments (e.g., with hidden states), constructing an accurate value function can become impractical, as the state information is incomplete or noisy.

Policy-Based Approaches

The *policy-based* approaches adopt a different strategy, directly learning a parametric policy $\pi_\theta(a | s)$, where θ represents the parameters that are optimized to maximize the expected return $J(\theta)$:

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^T G_t \right] = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^T \gamma^t r_t \right] \quad (3.8)$$

In this case, each time step t corresponds to a point in time where the agent observes the state s_t , takes an action a_t , receives a reward r_t , and transitions to a new state s_{t+1} ; T represents the time horizon, i.e., the maximum number of time steps the agent can take in an episode; finally, γ is the discount factor, which determines the importance of future rewards compared to immediate ones.

Thus, we conclude that Value-Based and Policy-Based methods are two complementary paradigms for tackling RL problems. To understand this difference, one can consider the analogy with a traveler:

- A Value-Based method can be interpreted as a detailed map of all possible routes to a destination, using this map to choose the best path.
- A Policy-Based method corresponds to learning an instinctive behavior, which allows directly reaching the destination without having to build a map.

Despite overcoming certain obstacles that characterize Value-Based methods, Policy-Based methods are not without limitations:

- **Convergence to local optima:** Policy-Based algorithms are subject to the risk of converging to suboptimal solutions, especially in environments with non-linear or highly complex rewards.
- **Sensitivity to hyperparameters:** The choice of parameters such as the learning rate can significantly affect the stability of learning.

Fundamental Principles of Policy Gradient Methods

We have seen that the goal of *Policy Gradient Methods* (PGM) is to optimize the parameters θ so that the policy maximizes the expected return Eq.(3.8). The key to optimizing the policy is to calculate the gradient of $J(\theta)$ with respect to the parameters θ and use it in the gradient update rule.

$$\theta_{t+1} = \theta_t + \gamma_t \nabla_\theta J|_{\theta=\theta_t} \quad (3.9)$$

By expanding the gradient of the objective function $J(\theta)$, we obtain the following expression:

$$\nabla_\theta J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s, a) \quad (3.10)$$

where:

- $d^{\pi_\theta}(s)$ is the stationary state distribution under the policy π_θ ,
- $Q^{\pi_\theta}(s, a)$ is the action-value function under the policy π_θ .

However, calculating this exact expression is particularly complex [40]. First, estimating $d^{\pi_\theta}(s)$ requires knowing the stationary state distribution induced by the policy, which depends both on the environment's dynamics and the policy itself. This dependency introduces a complex relationship between the probabilities of visited states, making its analysis and estimation computationally expensive. Similarly, estimating $Q^{\pi_\theta}(s, a)$ requires accumulating information on all possible future returns, which is prohibitive, especially in environments with many states and actions.

To overcome these difficulties, the *Policy Gradient Theorem* allows reformulating the gradient in a simpler, more tractable form [22]:

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{\pi_\theta} [Q_{\pi_\theta}(s, a) \nabla_\theta \log \pi_\theta(a | s)], \quad (3.11)$$

The approximation provided by the Policy Gradient theorem is computationally more accessible because:

1. It does not require explicitly estimating $d^{\pi_\theta}(s)$, as it relies on running the policy directly in the environment.
2. It does not need to calculate $Q^{\pi_\theta}(s, a)$, replacing it with the return G_t , which can be collected as the sum of observed rewards.

Below is an example in pseudocode of the application of the policy-based method Alg.(3):

Algorithm 3 REINFORCE Algorithm

- 1: Initialize the policy parameters θ randomly
- 2: **repeat**
- 3: Execute the policy $\pi_\theta(a | s)$ in the environment for a full episode
- 4: Collect the sequence of states, actions, and rewards: $(s_1, a_1, r_1, \dots, s_T, a_T, r_T)$
- 5: **for** each time step $t = 1, \dots, T$ **do**
- 6: Compute the accumulated return from time t :

$$G_t = \sum_{k=t}^T \gamma^{k-t} r_k$$

- 7: Update the parameters θ according to the rule:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t | s_t) G_t$$

- 8: **until** the convergence criterion is satisfied
-

The REINFORCE algorithm [42], introduced as one of the first Policy-Based methods, is based on using the Policy Gradient Theorem to compute an update proportional to the product of the logarithmic gradient of the policy. The proportionality factor is the accumulated return G_t , which represents an estimate, via Monte Carlo sampling, of $Q^\pi(s, a)$.

The accumulated return G_t serves as a reward signal: if G_t is high, the probability of selecting action a_t is increased; otherwise, it is decreased. This makes the algorithm intuitive and simple to implement.

Theoretical Challenges

Despite the many advantages of Policy Gradient Methods (PGMs), there are also some theoretical challenges associated with them. The main difficulty lies in the high variance in gradient estimates,

which can slow down the learning process. Techniques like introducing a baseline are useful to address this issue.

Another challenge concerns the convergence of the algorithm. Gradient-based optimization methods, such as those used in PGMs, are often sensitive to hyperparameters like the learning rate and may not converge stably if these parameters are not chosen correctly. This problem becomes even more relevant in complex and highly dynamic environments, where the stochastic nature of rewards can introduce additional difficulties in algorithm stability.

Furthermore, due to the reliance on simulations to gather trajectories, training stochastic policies can be computationally expensive, especially in complex environments.

3.4 Business Applications of RL

3.4.1 Relevant Case Studies

The paradigm of Reinforcement Learning (RL) is emerging as a powerful tool to optimize complex processes in industrial contexts. The following sections summarize three seminal papers that explore the application of RL in industry, highlighting its potential in addressing challenges in automation, supply chain management, and dynamic decision-making.

A Modular Test Bed for Reinforcement Learning Incorporation into Industrial Applications

The work of Kozlica et al. [21] introduces a *modular testbed*, a simulation environment, to evaluate the application of Reinforcement Learning (RL) in industrial contexts. The aim is to address the demands of Industry 4.0, such as reducing product development and time-to-market, the ability to customize production to meet specific customer needs (mass customization), and the capability to produce single pieces on demand without the need for large batch production (batch size-one production). The authors propose a case study simulating a production system in a model factory, where the task of the RL agent is to transport and assemble products while adhering to predefined rules. Specifically, the goal is to coordinate the flow of goods, install components, and sort products based on their color.

The main innovation of the work is the integration of RL with the OPC UA architecture, a standard communication protocol for industrial automation systems. This combination allows the RL model's state and action spaces to be mapped with the OPC UA architecture, enabling the agent to interact directly with real or simulated devices via the client-server mechanism. The use of OPC UA also enhances transparency and interconnectivity between machines, promoting a modular and scalable approach for adopting RL in industrial systems.

The work demonstrates that Reinforcement Learning is particularly useful in this context for several reasons. First, unlike other machine learning techniques, RL does not require a pre-existing dataset but learns by exploring the environment and refining its action policy. This is crucial in dynamic production environments where data may be scarce or difficult to obtain. Second, RL agents can adapt to changes in operational conditions and optimize decision-making, reducing errors and inefficiencies.

Another significant contribution of the study is the evaluation of various RL configurations for the product assembly and sorting problem. The authors compare two reinforcement learning algorithms: Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO). Experimental results show that both agents are capable of learning the task, with PPO achieving better performance in terms of mission completion and cumulative reward.

Reinforcement Learning for Multi-Product Multi-Node Inventory Management in Supply Chains

Sultana et al. [37] propose an innovative approach based on Reinforcement Learning (RL) for inventory management in multi-node, multi-product supply chains, addressing the typical challenges of large-scale distribution. The goal is to optimize product replenishment between warehouses and retail points, taking into account various constraints such as maximum warehouse capacity, stochastic demand, and delivery delays.

The proposed architecture is based on a hierarchical multi-agent framework, with a distributed decision-making level between retail points and the central warehouse. Each store is represented by an RL agent that makes replenishment decisions based on local information, such as the forecasted demand and the current stock status. The warehouse, in turn, is controlled by another RL agent that adjusts supplies based on the aggregated store requests and its own storage and procurement capabilities from suppliers.

The authors implement the framework using the Advantage Actor-Critic (A2C) algorithm with quantized action spaces. This algorithm is chosen for its ability to combine value-based and policy-based learning, allowing for more flexible and adaptive inventory management. Additionally, the system uses an exploration strategy based on multinomial distributions to improve the convergence of the RL agents.

To manage interactions between agents, the model introduces a reward-sharing mechanism, where the warehouse receives a fraction of the rewards obtained by the stores, incentivizing it to adapt to demand without penalizing the overall efficiency of the supply chain. Furthermore, replenishment decisions are constrained by truck and retail point capacity limits, ensuring that the agents' actions are compatible with the physical constraints of the logistics system.

Once again, Reinforcement Learning proves crucial in addressing the complexity of the problem, as it allows optimal strategies to be learned without the need to design static rules. Specifically, the framework enables the management of operational constraints and dynamic conditions with a high level of autonomy. Experiments show that the proposed RL method outperforms the performance of traditional heuristic strategies used as baselines for evaluation. Additionally, the system demonstrates a significant learning transfer capability: agents trained on a specific number of products or stores can adapt to new scenarios without the need for retraining, making the approach suitable for real-world implementations in dynamic retail environments.

Deep Reinforcement Learning for Solving Management Problems: Towards A Large Management Mode

Jiang et al. [16] propose a framework based on Deep Reinforcement Learning (DRL) to tackle complex management problems, including inventory management, dynamic pricing, and recommendation systems. The goal of the work is to develop a unified decision-making model that coordinates these interconnected activities, improving overall business operations efficiency compared to traditional methods based on heuristics or analytical models.

The main innovation lies in the introduction of a Large Management Model (LMM), a system inspired by generative artificial intelligence models, capable of learning optimal decision-making strategies through pre-training on large amounts of historical data. The LMM uses a Decision Transformer architecture, which allows for modeling sequences of past decisions and predicting the most effective actions to maximize business performance. This approach differs from traditional RL techniques by employing attention mechanisms and sequential modeling typical of transformer models. These elements improve the stability of the learning process, allowing the model to avoid oscillations or divergences in results, and enhance generalization, enabling the system to quickly adapt to unforeseen scenarios without requiring retraining.

From a technical standpoint, the problem is formalized as a Markov Decision Process (MDP), defined by states that include information on demand, prices, inventory, and recommendations, ac-

tions representing replenishment, pricing, and product exposure management decisions, and a reward function that evaluates business performance in terms of profit and operating costs.

To find the optimal policy that maximizes cumulative reward, the model was trained using advanced RL algorithms such as Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A3C). These algorithms effectively balance exploration and exploitation, adapting to dynamic market variations and ensuring robust solutions even in the presence of uncertainties.

To evaluate the effectiveness of the DRL approach, several baselines were implemented:

- *Base-stock List-price* (BSLP), an optimal policy for additive demand functions under specific distribution assumptions.
- (s, S, p) , which optimizes pricing and replenishment in the presence of fixed costs.
- *Myopic*, which calculates periodic prices based on the initial inventory level, combining them with base-stock policies for replenishment.

Experimental results show that the DRL framework outperforms these strategies, increasing overall profit, reducing stock-outs, and improving recommendation personalization. The LMM also stood out for its ability to generalize to new scenarios, confirming the practical potential of the approach in real-world business contexts.

As these studies demonstrate, the application of RL in industry is still in its early stages, but its ability to optimize dynamic and stochastic processes positions it as a key technology for the future of industrial automation and management. Now, let us delve deeper into the specifics and present three examples of the application of Reinforcement Learning in the context of human resource allocation, which is the focus of this project. Over the years, various strategies have been experimented with, each highlighting a different aspect of the potential of this new learning paradigm.

Reinforcement Learning for Enhancing Human Security Resource Allocation in Protecting Assets with Heterogeneous Losses

Ahire e Abdallah [1] address the problem of allocating cybersecurity resources in cyber-physical systems (CPS) vulnerable to attacks by proposing a Reinforcement Learning (RL) approach to optimize resource distribution in multi-round scenarios. The distinguishing element of their work is the integration of RL with Prospect Theory to model human cognitive distortions in risk assessment and enhance the protection of assets with heterogeneous losses.

In their model, the *human defender* must allocate limited resources to protect multiple assets, each characterized by a loss value and a probability of compromise. However, due to the human tendency to overestimate minimal risks and underestimate high ones, resource distribution may become inefficient. The authors demonstrate that this distortion leads to increased overall system vulnerability and propose an RL agent that, through an iterative learning process, progressively optimizes resource allocation, correcting imbalances caused by subjective risk perception.

The RL algorithm employed utilizes a multi-round architecture based on a *Markov Decision Process* (MDP) model, where each state represents a configuration of the defender's resource distribution and the active threats in the system. The available actions correspond to various resource allocation strategies, while the reward function is designed to minimize expected financial loss by optimizing protection coverage.

The RL agent is trained through a policy iteration-based reinforcement learning algorithm, where the defender iteratively updates their strategy based on the results of previous simulations. To accelerate convergence, the authors adopt a controlled exploration-exploitation approach, adjusting the exploration rate as the model learns the optimal resource allocation configuration.

In their experiments, Ahire and Abdallah evaluate the model’s effectiveness in simulated scenarios with multiple assets of varying value and differing attack probabilities. By comparing the RL approach with heuristic and static decision-making strategies, they demonstrate that the algorithm achieves up to a fourfold reduction in financial losses over the long term. Additionally, they test the effect of key parameters, such as the security budget and the learning rate, showing that a higher budget reduces overall risk and that an appropriate learning rate accelerates the model’s convergence to an optimal resource distribution.

One of the most innovative aspects of their work is the introduction of Prospect Theory to model the distorted risk perception of the human defender. This integration enhances the realism of simulations and refines resource allocation strategies by accounting for the suboptimal decisions that a human operator might adopt in a real-world context.

Can Reinforcement Learning solve the Human Allocation Problem?

Nguyen et al. [27] tackled the problem of human resource allocation, a combinatorial optimization challenge classified as NP-hard, by applying advanced Reinforcement Learning (RL) techniques. Their work introduces and compares several RL approaches, including Contextual Bandit, Double Deep Q-Network (DDQN), and an advanced version of DDQN combined with Monte Carlo Tree Search (MCTS).

One innovative aspect of their study is the adaptation of RL techniques to a task assignment problem with temporal and spatial constraints, characterized by a high number of possible combinations. The authors formulated the problem as a Markov Decision Process (MDP), defining states that include *worker availability*, *assigned tasks*, and *remaining time*. The reward function was designed to maximize allocation efficiency, penalizing inefficient assignments and rewarding those that reduce the total task completion time.

To evaluate the performance of the different models, the authors conducted experiments on increasingly complex configurations, starting from 3 workers and 2 tasks up to 7 workers and 6 tasks. The models were compared using the average task completion time and the time reduction relative to random assignment. Results showed that Contextual Bandit performed well for small-scale problems but faced computational limitations as complexity increased. DDQN improved resource allocation with an average 20% time reduction compared to the random method. The addition of MCTS further enhanced the results, achieving an average time reduction of 30%.

The use of RL was critical in addressing the exponential complexity of the problem. Conventional approaches, such as mathematical optimization methods and classical heuristics, proved inadequate for handling the dynamic nature and growing dimensionality of real-world cases. The integration of DDQN with MCTS improved action exploration and selection capabilities, making the model more effective for resource allocation in dynamic and large-scale scenarios.

Task distribution and human resource management using reinforcement learning

Paduraru et al. [29] addressed the challenge of task distribution and human resource management in large organizations by developing a Reinforcement Learning (RL)-based system to automate task assignment. The main innovation of their study lies in the integration of RL with *Natural Language Processing* (NLP) to enhance the system’s effectiveness and autonomy.

The study aimed to reduce the human workload required for task distribution by automating decision-making through an RL agent. The proposed system relies on a historical dataset of manually assigned tasks and leverages NLP techniques to process request details, constructing a global dictionary that links tasks to the employees who completed them. Each worker is profiled based on their accumulated experience, defined by the set of tasks previously performed. The RL agent is then trained on task stacks of various sizes, optimizing assignments to minimize the estimated completion time.

The implemented RL algorithm employs a variant of the Deep Q-Network (DQN), specifically Double Q-Learning, with a deep neural network structured with three convolutional layers followed by pooling and dropout layers. The network's output provides two key insights: the probability of a particular worker being the most suitable for a task and the estimated time required to complete it. The agent's input combines NLP-extracted features with worker information, enabling continuous optimization based on data.

To evaluate the model's effectiveness, the authors conducted experiments with various parameter sets, comparing different learning and optimization configurations. The reward function accounted for critical factors such as the average time each worker takes to complete tasks, task difficulty, and the employee's current workload. Experiments showed that combining the Adam optimization algorithm with a learning rate of 0.025 and 20,000 steps yielded the best results, achieving an average reduction of approximately 200 days in task completion time compared to conventional methods.

RL proved essential in enabling a fairer workload distribution, reducing the risk of overloading certain employees, and improving the organization's overall efficiency.

3.4.2 Advantages of RL in Business Contexts

Analyzing various significant application examples, the previous section demonstrated how the reinforcement learning (RL) paradigm is rapidly gaining prominence in business and industrial applications. In this section, we summarize the main advantages of adopting RL in the corporate domain, with a specific focus on Workforce Planning. Understanding these opportunities is crucial to forming a comprehensive view of this technology's potential and identifying the objectives necessary to achieve optimal outcomes from its application.

The main advantages are summarized in Table [3.1]:

We can now outline the key objectives for designing a robust reinforcement learning (RL) algorithm capable of ensuring *effective* and *efficient* human resource management:

- **Skill Optimization:** The algorithm should optimally match employee skills to the specific requirements of projects. This involves identifying both technical and soft skills and dynamically learning combinations that maximize productivity and project success.
- **Balancing Business Needs and Employee Well-Being:** A good RL algorithm should account not only for the strategic goals of the organization, such as timely project completion or cost reduction, but also for employee well-being. This includes factors like balanced workloads, job satisfaction, and minimizing burnout risk, fostering a sustainable and productive work environment.
- **Dynamic Priority Management:** Workforce Planning often requires addressing sudden changes, such as new client demands or unavailability of key resources. An effective RL algorithm should adapt quickly to such changes, reorganizing assignments flexibly to ensure operational continuity.
- **Resource Waste Reduction:** The algorithm should minimize employee downtime and optimize resource utilization, reducing waste and enhancing operational efficiency. This aspect is particularly critical for organizations with a large workforce and multiple ongoing projects.
- **Long-Term Forecasting and Planning:** An RL algorithm for Workforce Planning should learn from historical data and use it to support long-term strategic planning. For instance, it could recommend training paths to ensure employees are prepared for future market demands.
- **Automation and Transparency in Decision-Making:** Lastly, a good RL algorithm should automate decision-making processes, reducing the cognitive load on managers while ensuring transparency and traceability. Every decision should be explainable in terms of the agent's learned policy, facilitating system monitoring and auditing.

Adaptability to dynamic scenarios	RL allows agents to adapt in real-time to evolving conditions. This is particularly useful in business contexts where variables such as market demand, raw material costs, or resource availability are constantly changing. For example, in the logistics sector, an RL-based system can optimize vehicle routing by considering dynamic factors like traffic or weather conditions.
Multi-objective optimization	Many business problems require balancing multiple and sometimes conflicting objectives, such as cost reduction and service quality improvement. RL algorithms can effectively manage this complexity due to their ability to learn a policy that considers the full spectrum of rewards and trade-offs.
Advanced decision automation	One of the main strengths of RL is its ability to make complex decisions without direct human intervention, relying on knowledge derived from experience. This not only reduces the decision-making burden on managers but also improves the efficiency and accuracy of decisions. For example, in finance, RL has been used to autonomously and dynamically optimize investment portfolios.
Robustness to uncertainty and noise	Business environments often present incomplete or noisy data. RL algorithms are designed to learn robust policies even in the presence of these challenges, making them ideal tools for risk management and optimization in complex contexts.
Scalability capabilities	RL is highly scalable and can be implemented across various functional areas of an organization, from production to logistics, from sales to marketing. This flexibility makes it a versatile technology for the continuous improvement of business performance.

Table 3.1: Characteristics and advantages of Reinforcement Learning in business contexts.

Reinforcement learning is one of the most promising technologies for addressing the challenges of modern business management. Its general advantages make it a powerful tool for improving organizational efficiency and competitiveness. Specifically, in Workforce Planning, these benefits, combined with the ability to scale and adapt to various contexts, position RL as a key technology for the future of human resource management.

Chapter 4

Project Description

This chapter provides a detailed illustration of the developed research project, with particular emphasis on the problem definition and the design decisions that guided the system’s implementation. The main objective of the project is to optimize the assignment of a company’s employees to available projects, maximizing an overall criterion through the application of advanced Reinforcement Learning (RL) techniques.

Firstly, an analytical description of the variables involved in the problem, including employees and projects, is provided, followed by the identification of the main constraints characterizing the operational context, such as budget, priorities, and resource availability. Subsequently, the problem is formalized within the Reinforcement Learning paradigm, with a detailed description of the environment, the state space, the action space, and the reward function used to guide the learning process.

Finally, the chapter discusses the design choices made, focusing on the selection of the PPO (Proximal Policy Optimization) algorithm as the reference solution. The motivations behind this choice are explored, along with the tools used for the system’s development and evaluation.

4.1 Problem Specifications

4.1.1 Description of Variables: Employees, Projects

One of the most challenging aspects of the Strategic Workforce Planning process is to represent in a simple and effective way what, in reality, is extremely complex: *human capital* and *project units*. These two elements are not only the main subjects of the business context but also the key players in the planning cycle. The following section focuses on their interaction and the pivotal role they play in building an optimized resource allocation system. For this reason, the modeling of these elements was chosen as the starting point, with particular attention given to their representation.

An employee is not merely a set of technical skills but a person with experiences, attitudes, and capabilities that are difficult to fit into a rigid structure. Similarly, a project is not just a list of requirements but the result of a blend of business needs, deadlines, and priorities.

However, when working with a Reinforcement Learning agent, it is necessary to translate all this complexity into a form that the model can understand. This means making choices, simplifying where possible, and sacrificing some of the richness and nuance of the real world. Only in this way was it possible to build a solid foundation to address the optimization problem at the heart of this study.

Employees

As we mentioned, to create a meaningful representation of human capital, it is necessary to consider a range of characteristics that go beyond simply enumerating technical skills. These aspects, which define the nature, abilities, and potential of an individual within an organization, are fundamental to ensuring that the model can accurately capture and reproduce real-world dynamics. Based on the

analysis in *The Oxford handbook of human resource management* by Boxall et al.[4], it is possible to identify some of these key characteristics.

- **Skills and Abilities**

- **Technical and specialized knowledge:** Specific knowledge required to perform specialized activities. Implies the ability to deliver high-quality results in a short time.
- **Soft skills:** Communication, problem-solving, leadership, time management; therefore, the ability to work in a team, often essential in complex work environments.
- **Adaptability:** Ability to learn and apply new skills in different contexts. E.g., the ability to adapt to and use innovative technologies. This can include the ability for professional growth through training programs or hands-on experiences that lead to more advanced skills.
- **Attitude:** A consistent work style that helps predict future performance.
- **Flexibility:** Ability to respond to changes in projects or priorities without compromising results.

- **Experience**

- **Years of experience (seniority):** Greater experience in relevant sectors or roles improves the quality and effectiveness of contributions. Often, in corporate contexts, seniority is classified into discrete categories that may overlook individual nuances of experience.
- **Experience with similar projects:** Direct experience with projects or tasks similar to those to be undertaken.

- **Motivation and Aspirations**

- **Intrinsic and extrinsic motivation:** Level of commitment to work, which may derive from both internal factors (personal satisfaction) and external ones (rewards, career advancement).
- **Career goals and preferences:** Professional ambitions that influence the willingness to work on certain projects.

- **Availability**

- **Available working hours:** The amount of time an employee can dedicate to projects. This varies depending on the individual's capabilities, type of employment contract, and external factors beyond the individual's control.

- **Growth Potential**

- **Ability to innovate:** Tendency to propose new ideas and creative solutions that can improve business processes.

Despite the importance of the characteristics described above for a comprehensive representation of human capital, not all of these dimensions have been considered in the Reinforcement Learning model developed. This is due both to the computational complexity that would arise and to the limited availability of complete and accurate data.

However, the model focuses on the most relevant aspects for optimizing assignments, such as *technical skills*, *seniority*, *availability*, and *adaptability*, ensuring that this information, though limited, is sufficient to reproduce realistic and easily interpretable dynamics.

Projects

A project, in its common definition, is a set of coordinated activities and actions that mobilize resources over a specific period of time, with a defined start and end, in order to meet a clearly identified need.

Similar to the formalization of employees, business projects can also be represented in a structured way. This section will explore the main characteristics and elements to consider when aiming to create an effective representation of business projects, with a specific focus on the context of an IT consulting company.

According to *Methods of IT Project Management* [5] by J. Brewer e K. Dittman, when formally representing a project, it is essential to take several key aspects into account. These elements are crucial to ensure proper planning, monitoring, and project management. Here are the main characteristics that should be considered:

1. **Project Objectives:** Clearly define the objectives the project aims to achieve. The objectives should be specific, measurable, achievable, realistic, and time-bound (SMART criteria). In particular, it is essential to formally define the project deliverables, meaning the tangible results to be produced. These must be measurable and well-defined.
2. **Resources:** Specify the necessary resources, which may include the type of team required, key roles, and the skills needed for each individual. Resource management is crucial for the efficiency and sustainability of the project.
3. **Timeline:** Plan a clear timeline indicating the project phases, deadlines, and main milestones. Defining realistic deadlines is essential to avoid delays.
4. **Budget and Costs:** Establish a clear budget, with an analysis of estimated costs and the financial resources needed. Accurate cost control is crucial to keep the project within financial limits.
5. **Change Management:** Create a plan for managing changes, capable of identifying and controlling any modifications during the project lifecycle.

Despite the importance of all these aspects for a comprehensive representation of business projects, not all of these elements have been fully considered in the Reinforcement Learning model developed. Similar to the employee representation, this choice was driven by the need to simplify the problem and reduce computational complexity, focusing exclusively on the most relevant aspects for optimizing assignments, such as: *Required resources*, *Timeline*, and *Budget and costs*.

It should also be noted that these points were considered because they have a direct correspondence with the elements that characterize the representation of employees. This suggests the existence of a natural connection between the two representations, which can be leveraged to create an effective matching between employee and project representations.

The next section will further explore this connection, analyzing how the two representations can be integrated to build a Reinforcement Learning environment that optimizes corporate resource allocation.

4.1.2 Constraint Descriptions: budget, priorities, availability

In the modeling of a human resource allocation problem, defining constraints is a crucial step to ensure that the system is capable of generating realistic and applicable solutions. Without constraints, the model could converge towards strategies that are optimal only in an abstract sense, but lack practical value: an allocation that ignores economic limits, individual capabilities, or project priorities would be unfeasible in practice and, therefore, ineffective as a decision-making tool.

Mathematically, constraints act by narrowing the space of possible decisions. Let \mathcal{X} be the space of feasible configurations for human resource allocation, and let \mathcal{X}^* be the set of optimal configurations according to a criterion of maximizing a suitable utility value. The goal of a realistic model is not simply to find an element of \mathcal{X}^* , but to ensure that the structural conditions governing the system are satisfied, that is:

$$\mathcal{X}^* \subseteq \mathcal{X}_{\text{realistic}} \subseteq \mathcal{X}. \quad (4.1)$$

Where $\mathcal{X}_{\text{realistic}}$ represents the set of configurations that meet the constraints imposed by the problem.

The constraints selected in this context have been defined with the aim of establishing a structured comparison between the characteristics of employees and those of the projects, as outlined in the previous sections. They can be grouped into three main categories:

- **Skill constraints (compatibility between resources and roles)**

Each position in a project requires a specific set of skills, and each employee has their own skill vector, which may be more or less suited to the various available roles. An optimal allocation is one that maximizes the compatibility between the employee’s profile and the assigned role.

- **Economic constraints (budget and personnel costs)**

Each employee has an associated cost, determined by their level of experience and the role they fill. The budget available for allocating resources to projects is limited and must be distributed efficiently: cost minimization should not compromise the quality of the assignment but rather guide choices towards economically sustainable solutions.

- **Fairness and workload constraints**

The distribution of assignments must avoid overloading some employees while underutilizing others. If an employee is repeatedly assigned without considering workload balancing, organizational inefficiencies will arise.

In the problem modeling, it would have been possible to introduce more *restrictive constraints* to guide the algorithm towards acceptable solutions from the very first iterations. However, these constraints were not adopted in order to respect the intrinsic nature of the Reinforcement Learning paradigm, which prioritizes the exploration of the solution space. For instance, it would have been possible to exclude non-optimal assignments in advance, such as preventing an employee with a skill level below the required minimum from being considered for a certain role. Although this approach drastically reduces the solution space, it limits exploration and could lead to the underestimation of unexpected but effective combinations. Alternatively, one could have imposed that an employee be considered only for roles in which they possess an exact match of the required skills. While this would ensure high-quality assignments from the outset, it would eliminate the opportunity to train employees to improve their profiles or to fill roles in emergency situations.

These constraints, while improving the algorithm’s convergence speed, contradict the essence of Reinforcement Learning, which is based on iterative interaction with the environment to balance exploration and exploitation. Allowing a certain degree of flexibility enables the algorithm to dynamically adapt, explore alternative strategies, and identify more robust and generalizable solutions over time.

Ultimately, the integration of well-defined constraints in the problem modeling not only ensures the feasibility of the proposed solutions, but also allows for an effective balance between practical needs, economic sustainability, and organizational well-being, creating an allocation system that is fair, efficient, and realistic at the same time.

4.2 Problem Formalization in RL

4.2.1 Environment

In this section, the Stable-Baselines3 library, used in this work to implement and train the Reinforcement Learning agent, will be analyzed. Stable-Baselines3 stands out for its ease of use, broad compatibility with custom environments, and efficiency in training models based on advanced algorithms. The reasons behind the choice of this library over other alternatives will also be explained, highlighting its advantages and innovative features.

What is Stable-Baselines3?

Stable-Baselines3 is an open-source library for Reinforcement Learning, developed in Python and based on PyTorch, which provides optimized implementations of RL algorithms. The library is an evolution of Stable-Baselines, originally written in TensorFlow, and was created with the goal of improving modularity, code readability, and integration with custom environments [30].

Among the main features of Stable-Baselines3, we find the **standardized implementation of algorithms**; this library offers the possibility to leverage advanced algorithms such as PPO (Proximal Policy Optimization), DQN (Deep Q-Network), and A2C (Advantage Actor-Critic), which are among the most widely used in both practical and academic applications.

The choice to use Stable-Baselines3 was motivated by several factors, with the most important being its **ease of use** and **flexibility**, which allow for the rapid and intuitive implementation of Reinforcement Learning environments. This feature made it possible to model the specific environment of the problem addressed.

Another key factor that supported this choice is its **compatibility with OpenAI Gym**: the latter is an open-source library developed by OpenAI that provides a standardized set of environments to test and develop Reinforcement Learning algorithms. This was crucial in the development of the project as it supports *custom environments*, making it possible to model specific and abstract problems such as human resource allocation.

Furthermore, despite being a relatively recent library, it already offers **extensive documentation** covering every aspect, from installation to the implementation of new environments and algorithms; the presence of an active community and educational resources has facilitated the learning process and the resolution of technical issues during development, providing support even for advanced scenarios and real-world use cases.

Finally, the algorithms have been thoroughly tested on well-known benchmarks (e.g., Atari, MuJoCo), ensuring **robustness** and **stability** in agent training, even when dealing with complex problems or dynamic environments.

Feature	SB3	OAI Baselines	PFRL	RLlib	Tianshou	Acme	Tensorforce
Backend	PyTorch	TF	PyTorch	PyTorch/TF	PyTorch	Jax/TF	TF
User Guide / Tutorials	✓✓	–	✓	✓	–	✓	–
API Documentation	✓✓	×	✓✓	✓	✓	✓	✓
Benchmark	✓	×	✓	✓	✓	✓	×
Pretrained models	×	×	×	✓	✓	✓	✓
Test Coverage	95%	49%	?	94%	74%	81%	
Type Checking	✓	×	✓	?	✓	✓	×
Issue / PR Template	✓	×	✓	✓	✓	×	
Last Commit (age)	<1 week	>6 months	<1 week	<1 week	<1 month	<1 week	<1 month
Approved PRs (6 mo.)	75	0	13	222	85	5	7

Table 4.1: Comparison of SB3 to a representative subset of active or popular RL libraries. Key: – means that the feature is only partially present; OAI: OpenAI; TF: TensorFlow; PR: Pull Request. From [30]

4.2.2 States and Spaces

The concept of *stato*, as introduced in section 3.2, is fundamental in the Reinforcement Learning paradigm. It represents the specific configuration of the environment in which the agent operates at a given moment, providing the necessary information to make decisions. In this work, the state is initialized as a *collection of variables* that describe the current situation of the system, structured to capture the essential elements of the assignment problem.

Among the variables observable by the agent are, first and foremost, the key entities of this study: the *Employees* (workforce) and the *Projects*, which, as we will see, are represented respectively by a skill matrix and an organized project structure.

The main requirement was then to provide the agent with a representation that keeps track of the current assignments: a sort of *snapshot* showing which employees have been assigned to which projects. This is managed by the **Assignments** list. This list, which mirrors the structure of the set of projects, is initialized with values of -1 and, at each iteration, is populated with the ID of the assigned employee. This structure allows for dynamic monitoring of how resources are allocated over time.

Additionally, the state includes a one-dimensional array called **Levels**, which represents the experience levels of the employees under consideration. As we have discussed, an essential part of defining the workforce is the initialization of employee levels. In this array, at the i -th position, we find the level of the i -th employee. This allows the agent to have a temporal snapshot of one of the workforce's key characteristics. It is important to note that both the level assignments and the representation of employees, once initialized, will remain fixed until the end of the episode (at least in this initial phase). While in a more realistic scenario, an employee might be promoted over time or acquire new skills—thus altering the representation—as a first approach, a *static* representation of the workforce has been chosen to simplify, above all, the interpretability of the results. In the initial phase, it could in fact be challenging to determine whether the agent is learning correctly, especially when evaluation metrics are not yet well-defined.

To balance the workload among employees, there is also an individual counter called **Employee Assignments**, which tracks how many times each employee has been allocated. This helps prevent imbalances in resource allocation, as this variable, structured as a one-dimensional array with a length equal to the number of employees, maintains a count of how many times each individual has been assigned.

Finally, a temporal progress indicator, **iter**, expressed as the number of completed iterations, is used to manage and structure the decision-making process over time.

Definition 1. *We define the state as the single instance of observation of the variables: Workforce, Projects, Assignments, Levels, Employee Levels, and Iter.*

Let $\mathcal{E} = \{e_1, e_2, \dots, e_{N_e}\}$ be the set of employees, with N_e being the total number of employees, and let $\mathcal{P} = \{p_1, p_2, \dots, p_{N_p}\}$ be the set of available projects, with N_p being the total number of projects. We define the state s_i at the i -th iteration as the tuple:

$$s_i = (w, p, l, a_i, \epsilon_i, i) \quad (4.2)$$

where:

- $w \in [0, 1]^{N_e \times N_s}$, with N_s being the number of skills; represents the normalized skill matrix of the employees.
- $p \in [0, 1]^{N_p \times N_\rho \times N_s}$, with M_p being the maximum number of positions required per project; represents the projects and the skills required for each position.
- $l \in \{0, 1, \dots, N_l - 1\}^{N_e}$, with N_l being the number of experience levels; represents the level of each employee.
- $a_i \in \{-1, 0, \dots, N_e - 1\}^{N_p \times N_\rho}$ represents the assignment matrix of employees to projects, where -1 indicates a vacant position.
- $\epsilon_i \in \{0, 1, \dots, I_{max}\}^{N_e}$, with I_{max} being the maximum number of iterations executable by the system; represents the number of times each employee has been assigned to a project.
- $i \in \{0, 1, \dots, I_{max} - 1\}$ represents the number of iterations elapsed in the decision-making process.

The system state is therefore formally defined as a tuple that describes the current configuration of the elements observable by the agent. The observation space, on the other hand, generalizes the concept of state. It is not a single instance, but rather the set of all possible states that the agent can observe during its interaction with the environment.

Definition 2. *The observation space O is defined as the Cartesian product of the component spaces of a state.*

$$\begin{aligned} O &= \mathcal{O}_w \times \mathcal{O}_p \times \mathcal{O}_a \times \mathcal{O}_l \times \mathcal{O}_\epsilon \times \mathcal{O}_i \\ &= [0, 1]^{N_e \times N_s} \times [0, 1]^{N_p \times N_p \times N_s} \times \{-1, 0, \dots, N_e - 1\}^{N_p \times N_p} \times \{0, 1, \dots, n_l - 1\}^{N_e} \times \\ &\quad \times \{0, 1, \dots, I_{max}\}^{N_e} \times \{0, 1, \dots, I_{max} - 1\} \end{aligned} \quad (4.3)$$

One of the key features of the Stable-Baselines3 library is the *spaces* module, which allows for the formal definition of both observation and action spaces within a Reinforcement Learning environment. This module enables the representation of state variables according to different types of spaces, depending on the nature of the problem, thus providing a rigorous and well-defined mathematical structure.

Within this framework, the observation space is modeled using `spaces.Dict`, a dictionary-based data structure that organizes information into distinct subspaces. Each subspace represents a fundamental component of the system, such as the employee skill matrix, the project assignment matrix, experience levels, and temporal timelines.

- **Discrete:** used to represent discrete spaces, where instances (either single state observations or actions) are labeled by integers.
- **Box:** employed for continuous spaces, where each dimension is associated with a range defined by minimum and maximum values.
- **MultiDiscrete:** suitable for spaces involving multiple discrete dimensions, such as when an action consists of simultaneously assigning an employee to a project.
- **MultiBinary:** used for spaces composed of binary variables, such as "yes/no" decisions.

Table [4.2] summarizes the characteristics of the observation space:

Variable	Module	Shape	lim inf	lim sup	Type
workforce	Box	(N_e, N_s)	0	1	float32
projects	Box	(N_p, M, N_s)	0	1	float32
assignments	Box	(N_p, M)	-1	N_e	int32
levels	Box	(N_e)	0	$N_l - 1$	int32
employees_timeline	Box	(N_e, T)	0	1	int32
projects_timeline	Box	(N_p, T)	0	1	int32
employee_assignments	Box	(N_e)	0	T	int32
iters	Discrete	-	0	M	int

Table 4.2: Variables implementation in observation space

4.2.3 Actions and Spaces

As discussed so far, the objective of the Reinforcement Learning (RL) agent is to optimize the assignment of company employees to available projects, maximizing a reward function defined in terms of efficiency and quality of the assignments.

The goal of the next section is to define the concept of action in the examined context, allowing us to mathematically formalize the decision the agent can make at each time step; finally, we will clearly and precisely characterize the action space as the set of all possible choices available to the agent. In this context,

Definition 3. *we define an action as the decision-making process that assigns an employee to a specific project.*

This choice implies a unique assignment and determines a change in the current state of the system. An action a_t taken by the agent at time t is represented as an ordered pair:

$$a_t = (i, j),$$

where:

- $i \in \{0, 1, \dots, N_e - 1\}$ represents the index of the selected employee;
- $j \in \{0, 1, \dots, N_p - 1\}$ represents the index of the selected project.

Thus, an action represents the realization of a match between an element from the set of employees and an element from the set of projects, taking into account available information about environment and resources at that moment. This choice, as we will see later, can be influenced by operational constraints, such as compatibility between the employee's skills and project's requirements.

Definition 4. *The set of all possible actions constitutes the action space.*

This describes the entire decision domain available to the agent for optimizing the cumulative reward. Formally, the action space is defined as the Cartesian product of the employee indices and the project indices:

$$\mathcal{A} = \{0, 1, \dots, N_e - 1\} \times \{0, 1, \dots, N_p - 1\},$$

where:

- N_e is the total number of available employees;
- N_p is the total number of active projects.

Consequently, the action space contains all possible combinations for assigning an employee to a project and is composed of $|\mathcal{A}| = N_e \cdot N_p$ elements. This discrete structure allows us to systematically model a decision-making process, providing the agent with a well-defined domain to optimize its policy.

4.2.4 Reward Function: Key Components

As mentioned earlier in Section 3.2, within the context of Reinforcement Learning, the reward function acts as the core mechanism for guiding the agent's decisions. Unlike hard-constrained approaches, which outright discard non-compliant solutions, the reward function enables a gradual evaluation of each assignment. It rewards choices that better align with the defined objectives while penalizing less efficient ones. This flexibility allows the model to explore a wider range of allocation strategies and progressively converge towards optimal solutions, free from rigid deterministic rules.

In this resource allocation problem, the reward function is designed to capture key priorities and constraints: matching employee *skills* to project needs, ensuring *cost control*, and maintaining a fair *workload distribution*. These components are quantitatively formulated to guide the learning process towards decisions that balance realism and efficiency.

Based on these considerations, the reward function used in the initial phase of this study is defined as:

$$r_i = \alpha + \beta S_i^M - \gamma c^e - P_i^e \quad (4.4)$$

In this expression we see the constraints described so far incorporated in a simple way without, however, losing generality

1. Reward for Skill Matching

The core component of the reward function is an assessment of how well the assigned employee’s skills match the project role’s requirements. It is computed as a dot product, referred to as *Skill Match*, between the skill vector of the employee S_i^e selected at iteration i and the required skill vector S_i^p for the project position chosen at the same iteration:

$$S_i^M = \sum S_i^e \cdot S_i^p \quad (4.5)$$

The rationale behind this formulation is that an employee is better suited to a position when their skill set closely aligns with the role’s requirements. The dot product gives higher scores to assignments with greater skill overlap, encouraging the model to learn allocation strategies that prioritize compatibility between employees and projects.

The coefficient β applied to S_i^M serves to emphasize the significance of this component within the reward function and acts as a tuning parameter during experimentation.

2. Penalty for Employee Cost

Another key component in the reward formulation is the cost associated with the assigned employee. Each employee is linked to a salary level which, as discussed earlier, is accessible to the agent at each iteration through the *levels* variable in the current state. This cost is subtracted from the reward as follows:

$$\gamma c^e \quad (4.6)$$

with the goal of promoting more economically efficient solutions.

The intuition behind this penalty could be formalized as follow: given the same *Skill Match*, the system should favor assigning lower-cost employees, fostering a more budget-conscious allocation strategy.

The coefficient γ serves to normalize costs and balance them against the other terms in the reward function, preventing potential issues related to scale divergence.

3. Penalty for Overusing a Single Employee

As mentioned in Section 4.2.2, the *Employee Assignments* variable acts as a counter, tracking how many times an employee has been successfully allocated. To prevent the same employee from being repeatedly assigned without considering workload balance, a penalty term P_i^e has been introduced. This term is proportional to both total number of assignments P^e per employee up to the current iteration and the current timestep i :

$$P_i^e = P^e \cdot i \quad (4.7)$$

The purpose of this penalty is to encourage a more balanced distribution of assignments across employees and to discourage excessive reliance on a limited set of individuals. As the episode progresses, the penalty increases with the timestep, making repeated allocations of the same employee progressively more costly.

4. Bias positivo

Finally, a baseline value $\alpha > 0$ has been incorporated to ensure that the algorithm receives non-zero feedback during the early stages of learning. This adjustment helps facilitate model convergence by preventing overly negative rewards, which could otherwise slow down the training process.

Reward Function Characteristics

The reward function adopts a linear formulation with respect to its key terms: skill matching, employee cost, and assignment penalties. This structure was chosen based on several considerations, balancing simplicity, efficiency, and robustness.

From a benefits perspective, linearity stands out for its interpretability, facilitating both the optimization process and convergence of a Reinforcement Learning algorithm. Furthermore, linear reward functions are computationally less demanding than more elaborate alternatives, reducing processing time and resource consumption. An additional advantage lies in their stability, as linear structures are less prone to undesirable dynamics such as oscillations or saturation, which are more common in non-linear models.

Nonetheless, this choice also entails certain limitations. A linear function may lack the expressiveness required to capture complex interactions among the variables involved. Moreover, the assignment of constant weights to each component risks oversimplifying the problem, potentially overlooking the evolving dynamics present in real-world scenarios.

Despite these trade-offs, linearity was deemed suitable for this study. The primary aim is to steer the model towards solutions that enhance skill-to-task compatibility, minimize associated costs, and ensure a balanced workload distribution. The simplicity of this design also facilitates the interpretation and validation of results, making it a pragmatic and effective choice.

4.3 Design Choices: PPO Algorithm

4.3.1 Introduction

In previous chapters, we introduced the Reinforcement Learning (RL) paradigm and discussed Policy Gradient methods, highlighting their strengths and limitations (Chapter 3). In particular, we examined how classical approaches based on direct policy optimization may suffer from high variance and convergence instability, requiring specific techniques to improve training efficiency.

Proximal Policy Optimization (PPO) algorithm, developed by Schulman et al. (2017) [34], was designed to overcome some of the key challenges associated with traditional Policy Gradient methods. Specifically, PPO aims to provide more stable and efficient policy updates compared to *Vanilla Policy Gradient*, while avoiding the computational overhead of more complex algorithms such as *Trust Region Policy Optimization* (TRPO) [33]. PPO introduces a mechanism to limit policy updates through the so-called *clipped surrogate objective*, which prevents drastic changes that could undermine training stability.

In this chapter, we will conduct an in-depth analysis of PPO algorithm, starting from its theoretical foundations and proceeding to its mathematical formulation. We will then discuss the advantages and limitations of this approach compared to alternative policy optimization techniques. Finally, we will present a practical implementation of PPO using the Stable-Baselines3 library, with a focus on the key hyperparameters that influence its performance.

4.3.2 Theoretical Foundations

Policy optimization in Reinforcement Learning relies on maximizing the expected reward accumulated by an agent interacting with an environment. Policy Gradient methods allow learning a parameterized policy $\pi_{\theta}(a | s)$, updated through gradient of the objective function's gradient. However, these methods suffer from several issues, including:

- **High variance:** Weight updates depend on noisy estimates of the policy gradient, making stable convergence difficult.
- **Sensitivity to learning parameters:** optimization can be inefficient without proper learning rate tuning.

- **Instability in convergence:** overly large updates can lead to degradation of the learned policy.

To address these issues, the concept of *Trust Region* has been introduced, used in algorithms such as *Trust Region Policy Optimization* (TRPO). TRPO imposes a constraint on distance between the new policy and the previous one, using KL-divergence to ensure more conservative and stable updates. The **Kullback-Leibler (KL) Divergence** is a measure of the difference between two probability distributions P and Q . It is defined as:

$$D_{KL}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \quad (4.8)$$

or, in continuous form,

$$D_{KL}(P||Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx. \quad (4.9)$$

This statistical measure quantifies how much information is lost when Q is used to approximate P . Within the framework of Reinforcement Learning, it is used to measure the variation between the updated policy π_θ and the previous one $\pi_{\theta_{old}}$. A high KL-divergence value indicates a drastic policy update, which could destabilize the training process.

In Section 3.3.3, we discussed how expression (3.8) for the objective function $J(\theta)$ is not directly differentiable, and it has been introduced the Policy Gradient Theorem, which provides a method to explicitly compute the gradient of the objective function. Building on this result (3.11), Schulman et al. introduced a new expression, the *Surrogate Objective Function* $\mathcal{L}(\theta)$, which directly approximates the gradient of the objective function $\nabla J(\theta)$ and can therefore be exploited for updates. It is precisely this new expression, with simpler interpretation and greater stability, that TRPO imposes an explicit constraint on the KL-divergence, ensuring that the new policy remains within a *trust region* around the previous policy. The constraint imposed by TRPO can be expressed as:

$$\mathcal{L}(\theta) = \max_{\theta} \mathbb{E}_{s \sim \rho_{\theta_{old}}, a \sim \pi_{\theta_{old}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A^{\pi_{\theta_{old}}}(s, a) \right] \quad (4.10)$$

subject to the condition:

$$D_{KL}(\pi_\theta || \pi_{\theta_{old}}) \leq \delta. \quad (4.11)$$

Here:

- $\pi_{\theta_{old}}$ is the previous policy with fixed parameters (θ_{old}).
- π_θ is the new policy to be optimized.
- $A^{\pi_{\theta_{old}}}(s, a) = Q^{\pi_{\theta_{old}}}(s, a) - V^{\pi_{\theta_{old}}}(s)$ measures how good an action is compared to the average actions the policy would have chosen in that state.
- $\rho_{\theta_{old}}$ is the state visitation distribution under the old policy.
- The term $D_{KL}(\pi_\theta || \pi_{\theta_{old}})$ acts as a constraint to limit policy changes between successive updates.
- The parameter δ controls the maximum allowed magnitude of the policy change.

This restriction ensures that policy updates occur in a more gradual manner, thereby preventing abrupt changes that could undermine the stability of training process. However, addressing this optimization problem entails calculating the KL-divergence and solving a quadratic programming problem, which significantly increases computational complexity.

$$D_{KL}(\pi_{\theta_{old}} || \pi_\theta) \approx \frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) \quad (4.12)$$

where H represents the Hessian matrix of the KL-divergence with respect to the parameters θ .

At this stage, the optimization problem can be formulated as:

$$\max_{\theta} \mathcal{L}(\theta) \quad \text{subject to} \quad (\theta - \theta_{\text{old}})^T H(\theta - \theta_{\text{old}}) \leq \delta. \quad (4.13)$$

To enable more controlled updates, Proximal Policy Optimization (PPO) introduces an approximate version of the TRPO constraint, based on a *clipped* objective function that limits the magnitude of policy changes. This reduces the need to compute the KL-divergence explicitly, thereby simplifying the optimization process.

PPO emerges as a simpler and computationally more efficient solution, preserving the benefits of TRPO without its complexity. The objective function is defined as:

$$\mathcal{L}(\theta) = \mathbb{E} [\min (r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (4.14)$$

where

- $r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio between the probability of the action under the new policy and the old policy,
- $e A_t$ represents the estimated advantage,
- ϵ is a small hyperparameter that approximately bounds how far the new policy can deviate from the old one.

The term $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)$ clips the probability ratio $r_t(\theta)$ to ensure that modifications to the policy do not exceed a predefined range. In practice:

- If $r_t(\theta) > 1 + \epsilon$, the gain is limited, thereby preventing excessive updates that could destabilize the training.
- If $r_t(\theta) < 1 - \epsilon$, a disproportionate penalty is avoided.

The min function selects the more conservative value between the original term $r_t(\theta)A_t$ and the clipped term, contributing to the stability of the optimization.

This formulation allows PPO to achieve competitive performance relative to TRPO with a simpler and computationally less demanding implementation. In the following paragraphs, we will analyze its practical implementation in greater detail.

Algorithm 4 PPO-Clip

Require: Initial policy parameters θ_0 , initial value function parameters ϕ_0

- 1: **for** $k = 0, 1, 2, \dots$ **do**
- 2: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
- 3: Compute rewards-to-go \hat{R}_t .
- 4: Compute advantage estimates \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
- 5: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left(r_t(\theta) \hat{A}_t, g(\epsilon, \hat{A}_t) \right),$$

where

$$r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, \quad g(\epsilon, \hat{A}_t) = \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t.$$

- 6: Typically via stochastic gradient ascent with Adam.
- 7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left(V_{\phi}(s_t) - \hat{R}_t \right)^2.$$

- 8: Typically via some gradient descent algorithm.
-

Chapter 5

Implementation

After introducing the problem of *Strategic Workforce Planning*, outlining the fundamentals of the Reinforcement Learning (RL) paradigm, and illustrating the theoretical structure of the problem—including the specific modeling choices and the *Proximal Policy Optimization* (PPO) algorithm—this chapter focuses on the practical implementation of the entire project.

The aim is to provide a detailed description of the operational phases that led to the realization of the RL system, from its initial development to its final validation. The technical and methodological aspects that enabled the translation of theoretical specifications into a concrete and functional application will be thoroughly examined.

Specifically, the chapter is structured as follows:

- **Environment Development:** A description of the construction of the simulator representing the real-world problem, including state-update logics and interactions with the RL agent..
- **Implementation of PPO algorithm:** Technical details on the integration and configuration of the algorithm using the Stable-Baselines3 framework, with particular attention to the customizations introduced to meet the project’s specific requirements.
- **Rollout and Training Phases:** An explanation of the trajectory collection process, the policy updates for the agent, and parameter optimization, highlighting how these phases were managed from a computational perspective.
- **Testing and Validation Phases:** A description of the methodologies used to evaluate the system’s performance, including tests performed on simulated scenarios and comparative analyses of the results.

5.1 Environment Description

5.1.1 Environment Implementation and Its Functions

One of the fundamental elements in implementing a Reinforcement Learning system is the creation of a *environment*, where the agent interacts to learn optimal policies. Through this structure, the agent is able to observe the current state, select an action, and receive a reward based on the quality of the proposed solution.

In this project, the environment was developed by inheriting from the base class `gym.Env` provided by the OpenAI Gym library, one of the most widely-used frameworks for creating simulation environments in RL. This library defines a standard interface that any customized environment must follow to be utilized by learning algorithms. An object inheriting from `gym.Env` must implement some key functions, such as:

- `reset()`, whose role is to reset the environment at the start of each episode. In this project, the `reset()` function was implemented to exclusively initialize the state of the environment and all its components as described previously, in paragraph (4.2.2).

- `step(action)`, a function that applies an action and returns the new state, the reward, and other information. The logical flow and specific implementation details of this function will be further examined in Paragraph (6.1).
- `render()`, to visualize the state of the environment (optional). In this case, it was not utilized; this decision stems from the nature of the addressed problem, which does not present a standard visual representation easily applicable through predefined tools.
- `close()`, to release any resources used (optional); it is typically exploited to close graphical windows opened by the `render()` function, and in this case, it thus proved unnecessary.

Using `gym.Env` as a base guarantees that the environment adheres to a standard format, making it easily compatible with RL libraries such as Stable-Baselines3 or Ray RLlib. The environment developed for the human-resource allocation problem does not merely use functions inherited from OpenAI Gym’s base class; it also includes various customized variables and functions, some of which are provided externally as inputs, while others are defined internally. These additional components are essential to specifically model the context of strategic workforce planning, allowing the agent to interact with the environment realistically and meaningfully. In particular, variables managing employee characteristics, project specifications, and specific assignment requirements were defined. Moreover, internal functions were developed to operate based on these variables, calculating penalties, rewards, and determining resource availability. The main customized variables and functions defining the environment’s behavior—both those externally provided and internally deduced—will now be illustrated.

Externally Defined Input Variables

In the context of our human-resource allocation problem, several variables are defined and passed to the environment upon its creation. These variables provide the fundamental parameters required to manage interactions between employees and projects effectively. The primary input variables are illustrated in Table 5.1:

The input variables are provided externally during the initialization of the environment, rather than being defined internally, to ensure greater *flexibility* and *modularity* in the implementation. This approach allows users to configure the environment according to the specifics of their problem without modifying the underlying code. Specifically, the number of employees, projects, skill requirements, and allocation costs are variables that strongly depend on the application context and business needs. By passing them as external inputs, the environment becomes adaptable to various scenarios without the necessity to alter the source code, thereby facilitating customization and enabling the testing of different datasets and configurations, as will be demonstrated in the next chapter.

Variables Derived from Inputs

In addition to externally defined input variables, the environment uses certain variables computed directly from the initial inputs to simplify and optimize internal logic. These variables do not need to be provided by the user; rather, they are automatically initialized within the environment based on the received parameters.

- `self.num_positions_per_project:`, a list of integers corresponding to the number of positions required for each project, calculated as the length of the array associated with the required positions in the parameter
- `project_positions:` this variable is useful to verify whether all positions within a project have been filled.
- `self.max_positions:` an integer representing the maximum number of positions required among all projects, derived as the maximum value of `self.num_positions_per_project`. This variable is employed to handle constraints and optimize allocation logic.

The use of derived variables avoids redundancy in information definitions and simplifies the internal management of the environment, thus making the resulting environment more efficient and less prone to errors.

Environment Definition Functions

In addition to the definition of input and derived variables, the implemented environment is characterized by a set of functions regulating its overall behavior, constituting the logical framework upon which agent-environment interactions are based. Among these, the *step function* holds a central role, serving as the operational core of the environment. It has received particular attention throughout experimentation, as it governs the evolution of states and implements the agent’s decisions. Alongside this function, the environment is structured with additional supporting functions, including *reward function* computation, verification of episode *termination conditions*, and accessory mechanisms such as *project padding* and management of operational constraints. Collectively, these components have been crucial for the design and calibration of various experimental configurations. Each of these functions will be discussed in detail in subsequent chapters, with specific reference to each conducted experiment.

5.2 Creation of PPO Model

Overview of the Algorithm Structure

The implementation of the Proximal Policy Optimization (PPO) algorithm within the Stable-Baselines3 library exemplifies modular and scalable design in reinforcement learning (RL) algorithms. In this section, the structure of the main PPO class is analyzed, describing its fundamental components, the auxiliary classes involved, and their integration within the library’s overall framework.

Main Class: PPO

The main class, PPO constitutes the core of the implementation. It inherits from the `OnPolicyAlgorithm` class, which provides a generic structure for on-policy RL algorithms. This inheritance enables PPO to leverage common methods and functionalities shared across all on-policy algorithms, while simultaneously ensuring the flexibility required to customize PPO-specific details.

Initialization

The PPO constructor (the `_init_` method) accepts a series of parameters to configure the algorithm. Among these, the most important ones are:

- **policy**: Specifies the policy to use. It can be one of the predefined policies (e.g., `MlpPolicy` for fully connected networks or `CnnPolicy` for convolutional networks) or a custom one.
- **env**: The environment in which the algorithm operates. It can be a single environment or a parallel collection of environments managed by `VecEnv`. In the case of this project, the environment, `EmployeeAssignmentEnv`, is not a standard environment but has been custom-built as introduced in the previous section, specifically tailored to represent the Strategic Workforce Planning problem.
- **Hyperparameters**: Parameters such as `learning_rate`, `n_steps` (steps per episode), `batch_size`, and `clip_range`, which control PPO’s behavior.

During initialization, the main class creates instances of key components, such as the policy neural network, the optimizer, and the rollout buffer. Inheriting from `OnPolicyAlgorithm` ensures the availability of standard methods like `collect_rollouts`, which collects and stores experiences, and `train`, used for training.

Rollout Buffer: RolloutBuffer

A fundamental component in PPO’s implementation is the rollout buffer, represented by the `RolloutBuffer` class. This buffer is responsible for storing the transitions collected during interaction with the environment, enabling efficient calculations for updating the policy and value functions.

Buffer Structure

Before interacting with the environment, a buffer is created to store information related to each timestep of the episode or a batch of episodes. Typically, this information includes:

- Observations: States observed during interaction with the environment, defined within the observation space described in paragraph 4.2.2.
- Actions: Actions performed by the agent, implemented according to the logic described in paragraph 4.2.3.
- Rewards: Rewards received by the agent, varying based on the specific scenarios considered.
- Values: Values estimated by the value function.
- Action log-probabilities (`log_probs`): Used to compute the ratio between the new and previous policies.
- Masks (`done`s): Episode termination indicators, used to distinguish rollouts from different episodes.

This can be implemented either as a dictionary of arrays or a customized class.

Therefore, the rollout buffer is actively involved during each cycle of interaction between the agent and the environment, bridging data collection and model updates. It serves not only as a data repository, storing collected transitions, but, as we will see, plays a crucial role in data preparation for training, followed by optimizing the training process itself, operations that always take place within the buffer.

Policy and Model: ActorCriticPolicy

The `ActorCriticPolicy` class defines the neural network architecture used both for the policy (actor) and the value function (critic). This choice allows for sharing parameters between the two components, thereby reducing computational complexity.

Model Architecture

The policy is implemented as a parametric neural network, with predefined configurations such as `CnnPolicy`, a convolutional neural network ideal for high-dimensional inputs like images, or, as in our case, `MlpPolicy`, a fully connected network suitable for environments with vector-based inputs.

Each network includes:

- Input Layer: Sized according to the environment’s observation space.
- Hidden Layers: Configurable by the user (e.g., number and size of layers).
- Output Layers: One output for the actions (policy distribution) and one for the estimated value.

The policy returns three fundamental outputs. First, it samples *actions* from the policy distribution, which are used to guide the agent in its interactions with the environment. Second, it provides an evaluation of the current state through the estimation of the *value*, which is essential for advantage calculation and for optimizing the critic function. Finally, it computes the *log-probability* of the sampled actions, a crucial parameter for calculating the PPO loss, as it enables comparing the new policy with the previous one during training.

Training Process: train

The training process of PPO is primarily managed by the `train` method, which will be examined in detail in the following paragraph to thoroughly analyze and explore its logical flow.

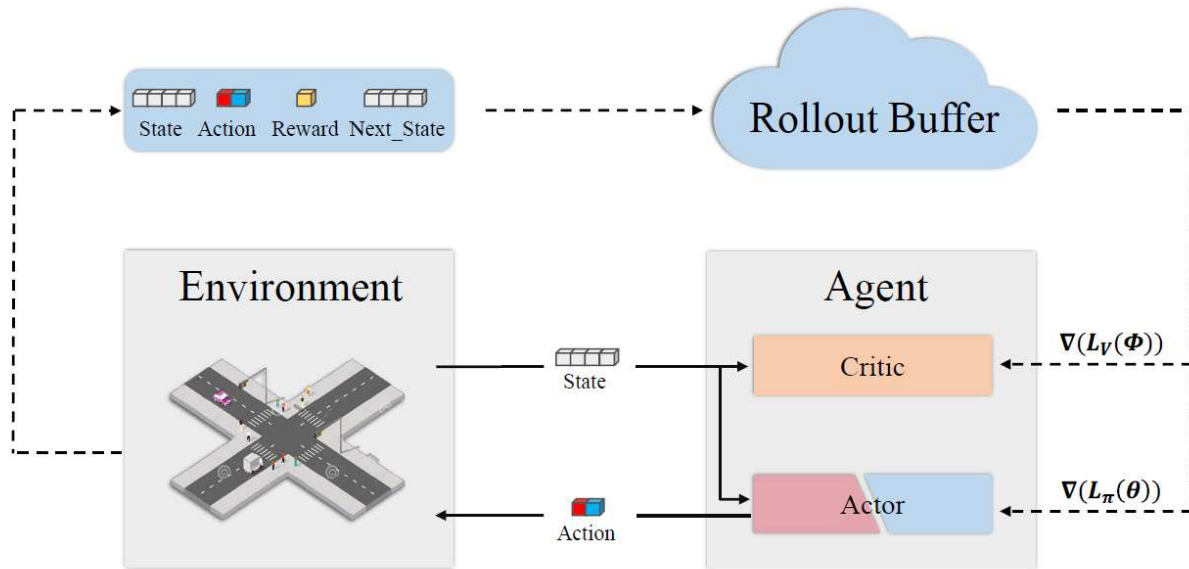


Figure 5.1: Interaction between Agent, Environment and Rollout Buffer [23].

5.3 Training

The training phase of the Proximal Policy Optimization (PPO) algorithm represents the core of the agent's learning process, in which all the components analyzed in previous chapters interact to optimize the policy. Previously, the fundamental elements for understanding this phase have been discussed in detail, including the PPO algorithm logic, the structure of the experience buffer, and the architecture of the neural networks constituting the policy and value networks. In this section, the focus is on the practical integration of these components and the operational flow guiding the entire training process.

1. Initialization

The training process begins with the initialization of the agent, the neural network, and the required buffers:

- Neural network: A network with two main components is defined:
 - A FEATURE EXTRACTION NETWORK that processes each subspace of the state (workforce, projects, assignments, timeline, etc.). Multidimensional matrices such as workforce and projects are processed using convolutional blocks or fully connected modules, followed by nonlinear activation functions (e.g., ReLU).
 - A network calculating the POLICY and another for the VALUE FUNCTION. The policy network, or Actor, generates probability distributions over actions, while the value network, Critic, approximates the state-value function.

Neural network weights are initialized using a uniform distribution, and biases are initially set to zero.

- Experience buffer: As mentioned previously, a circular buffer is initialized as a dictionary to store transitions in the form "state, action, reward, next state, initial action probability." Additionally, the buffer is structured to support mini-batch training.
- Initial parameters: Hyperparameters are configured, including the learning rate, clipping factor, batch size, and the maximum number of iterations per episode.

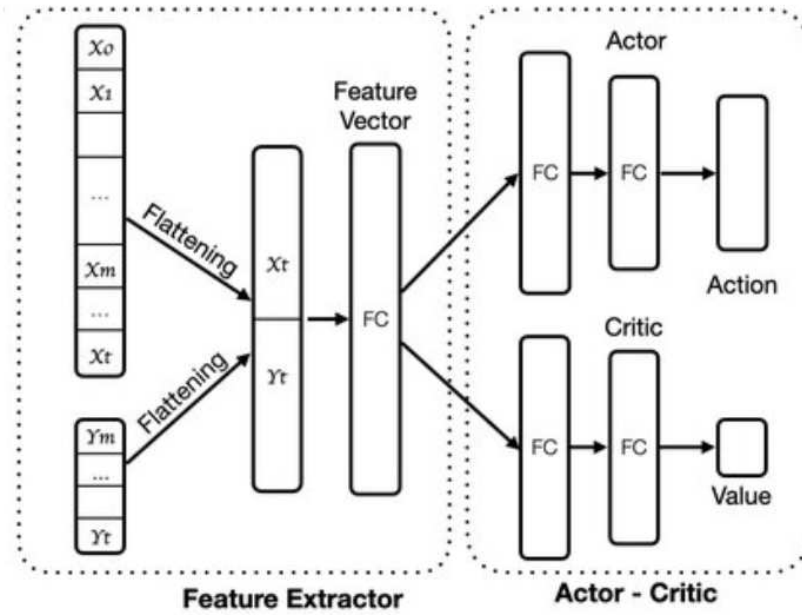


Figure 5.2: Neural Network structure implemented by PPO [9].

2. Data Collection and Interaction with the Environment

The agent interacts with the environment following the current policy π_θ . In this context, the policy corresponds to the parameters associated with the Actor network, which will later be optimized.

At each iteration, the agent observes a state $s_t \in \mathcal{S}$ and samples an action $a_t \in \mathcal{A}$ from the probability distribution defined by the current policy, which is then passed to the environment. The environment executes the selected action through the `env.step(action)` function, returning a new state s_{t+1} , a reward $r_t \in \mathbb{R}$, and an episode termination signal *done*, if applicable. Transitions are stored in the experience buffer. Data collection continues until a predefined batch size (number of timesteps N_t) is completed.

3. Data Preprocessing

After data collection, the buffer data undergo preprocessing:

- Advantage Calculation: Advantages are computed using Generalized Advantage Estimation (GAE), combining temporal differences (TD) with an exponential discount term:

$$A_t^{GAE(\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \quad (5.1)$$

Where:

- γ is the *discount factor* for future rewards.
- λ is the *GAE parameter*, controlling the trade-off between bias and variance. A higher λ utilizes more future information (higher variance, lower bias), while a lower λ utilizes less future information (lower variance, higher bias).
- δ_t is the *TD error* (or Temporal Difference residual), defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (5.2)$$

This represents the difference between expected and estimated values.

- Normalization: Advantages are normalized to stabilize training.
- Mini-batch Reorganization: Transitions are divided into mini-batches to optimize the training process.

4. Network Update

The network update occurs through an iterative process over each mini-batch. For each mini-batch, the combined loss is calculated as follows:

- Policy Loss: Based on the ratio between the current and initial action probabilities, incorporating a clipping term to ensure stable updates.

$$L^{\text{clip}}(\theta) = \mathbb{E}_t [\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (5.3)$$

as previously described in (4.14).

- Value Loss: Measures the squared error between the predicted value and the target value:

$$L^{\text{value}}(\theta) = \mathbb{E}_t \left[\left(V_\theta(s_t) - V_t^{\text{target}} \right)^2 \right] \quad (5.4)$$

Where the target value can be calculated as:

$$V_t^{\text{target}} = r_t + \gamma V(s_{t+1}) \quad (5.5)$$

- Entropy Term: Encourages exploration by adding a penalty based on the entropy of the action distribution:

$$L^{\text{entropy}}(\theta) = \mathbb{E}_t [-\pi_\theta(a_t | s_t) \log \pi_\theta(a_t | s_t)] \quad (5.6)$$

The total loss is a weighted combination:

$$L^{\text{total}}(\theta) = L^{\text{clip}}(\theta) - c_1 L^{\text{value}}(\theta) + c_2 L^{\text{entropy}}(\theta) \quad (5.7)$$

where c_1 and c_2 are hyperparameter weights controlling the contributions of the value loss and entropy term.

Finally, during backpropagation, the total loss is used to update the network parameters through an optimizer (such as Adam).

5. Iteration and Saving

The data collection, update, and evaluation process is iterated for a predefined number of epochs. The updated policy is evaluated by calculating performance metrics; examples of policy evaluations are presented in Chapter 6.

The network parameters are periodically saved to ensure reproducibility and the possibility of restoring the model.

5.4 Test and Validation

In the context of the Proximal Policy Optimization (PPO) algorithm, the testing phase is characterized by the absence of parameter update mechanisms and by the deterministic execution of the learned policy. The following outlines the four main stages of the data flow during the testing phase, emphasizing the procedural and computational aspects typical of practical implementations.

1. Environment Initialization and Initial State Collection

The testing process begins with the creation of a new instance of the simulation environment `env = EmployeeAssignmentEnv()` and the subsequent retrieval of the initial state $s_0 \in \mathcal{S}$. This operation occurs through the `reset` function (`env.reset()`) or, subsequently, via the function `env.step()`. The resulting state is preprocessed, maintaining consistency with the format required by the Actor network.

2. Actor Inference and Deterministic Action Selection

Unlike the training phase, during testing, the state is exclusively provided to the Actor. The Actor generates a distribution over possible actions $\pi_\theta(a_t|s_t)$, but the action is typically selected deterministically. In the case of discrete action spaces, such as the one chosen in this context, the action corresponding to the highest probability is selected (argmax operator); for continuous spaces, the mean value of the parametric distribution would be used. The entire process is executed in a gradient-free context to ensure inference efficiency and speed.

3. Interaction with the Environment

Similarly to the training phase, the action determined by the Actor, $a_t \in \mathcal{A}$, is subsequently sent to the environment using the function `env.step(action)`, which acts as the transition function $T(s_{t+1}|s_t, a_t)$. Indeed, this function returns the next state s_{t+1} , the associated reward r_t , a boolean flag indicating potential episode termination (`done`), and additional optional diagnostic information. Collected data is not stored in any buffer, as it is not required for parameter updates.

4. Iterative Cycle until Episode Convergence

The interaction cycle between the agent and the environment iteratively continues until a terminal state is reached, identified by the flag `done=True`. Upon episode completion, the agent can be re-initialized in a new episode or in a new environment, depending on the adopted evaluation protocol. During this phase, the model stores the value of R_{total} for that episode and resets counters for the subsequent episode.

At the end of the testing phase, global evaluation metrics are computed, including:

- **Average reward per episode:** where N_ϵ is the total number of tested episodes.
- **Policy stability:** Analysis of the variance of rewards per episode.
- **Time-steps per episode:** Number of timesteps needed for episode convergence.
- **Final cumulative reward:** represents the reward obtained at the end of each episode.

During the testing phase, no updates to the policy parameters θ are performed. The model exclusively conducts inference using the learned policy, ensuring that the results reflect the actual performance of the policy without interference from learning processes.

The testing phase provides an objective indication of the effectiveness of the learned policy, highlighting potential limitations or gaps in the agent's behavior. This analysis allows assessing whether the policy is sufficiently generalized and robust for deployment in real-world scenarios.

Variable	Type	Description
<code>self.num_employees</code>	<code>int</code>	Total number of employees available for assignment to various projects. This variable defines the size of the workforce for each simulation episode.
<code>self.num_projects</code>	<code>int</code>	Total number of projects to which employees can be assigned. Each project requires a specific number of resources (employees), each with varying skills and experience levels.
<code>self.num_skills</code>	<code>int</code>	Total number of skills required to fill the positions requested by the projects. Each employee and position within a project is associated with a set of competencies.
<code>self.num_levels</code>	<code>int</code>	Number of employee experience levels. In our case, 4 competence levels are currently defined, determining employee assignment costs and suitability for specific positions.
<code>self.costs</code>	<code>list of int</code>	A list containing the costs associated with the various employee levels. Each employee, depending on their level, has an associated cost that may influence the agent's resource-allocation decisions.
<code>self.project_positions</code>	<code>list of list of int</code>	This list contains arrays specifying the competency levels required for each position within each project. These levels are critical in determining employee suitability for specific positions.
<code>self.learning_rate</code>	<code>float</code>	The learning rate used in the optimization algorithm. It defines how quickly the model adapts during the training process.
<code>self.time_length</code>	<code>int</code>	The length of the episode, i.e., the number of timesteps the agent can perform before the episode ends.
<code>self.max_iters</code>	<code>int</code>	Maximum number of iterations allowed during the agent's training. Each iteration represents an optimization phase.

Table 5.1: Externally Defined Input Variables

Chapter 6

Scenarios and Results

This chapter presents and discusses the experiments conducted to evaluate the effectiveness of the Reinforcement Learning (RL) approach in addressing the Strategic Workforce Planning problem. The primary goal of this experimental analysis is to understand how different environmental configurations and variations in the environment's transition function influence the optimal decisions learned by the agent.

To achieve a meaningful evaluation, three distinct scenarios have been defined, each characterized by different levels of human resource availability relative to project requirements.

The first scenario, used as a *Baseline*, assumes a balanced condition between the number of available employees and the resources required by projects. This simulation represents a standard scenario, providing a simple benchmark against which to compare more advanced alternatives. The ultimate goal is to determine whether the proposed method genuinely improves the situation or if its performance is merely comparable to the baseline strategy.

The second scenario represents a condition of workforce shortage (*Under-sampling*), where the number of available employees is insufficient to meet project demands, necessitating strategic hiring policies. Analyzing this scenario helps identify the optimal timing for hiring new resources and predict the optimal number of hires required to mitigate staffing shortages. The objective here is to provide a straightforward yet clear tool to support decisions such as whether it is necessary to hire externally or whether it is more cost-effective to train internal employees.

Finally, workforce requirements can fluctuate over time due to external factors, such as market changes or seasonal demand. The third scenario considers the opposite case (*Over-sampling*), in which the number of employees exceeds the demand generated by projects, implying the need for workforce reallocation or downsizing strategies. Analyzing this scenario is equally crucial for reducing waste and maintaining business competitiveness, as an excessive number of employees relative to project demand may lead to operational inefficiencies and unjustified costs.

Experiments were conducted by appropriately modifying the environment to adapt it to the specific requirements of each analyzed context. Analysis of the obtained results allows for evaluating RL agent performance in terms of optimizing human resource allocation and comparing it with traditional decision-making policies.

In the remainder of this chapter, the adopted parameters, evaluation metrics, and key observations arising from the comparative analysis of the various experimental scenarios will be illustrated in detail.

6.1 Baseline

In the first phase of the analysis, the hypothesis characterizing the initial Baseline scenario was defined:

Hypothesis 1. *The number of employees equals the number of resources required by the projects.*

$$N_e \simeq \sum^{N_P} N_\rho \quad (6.1)$$

or at least of a comparable order of magnitude.

In other words, it is assumed that every resource needed to complete a project directly corresponds to an assigned employee, without excesses or shortages in resource availability. This purely descriptive, baseline scenario serves as a reference point for subsequent analyses, representing a situation without optimization or misalignment between resource supply and demand.

Hypothesis (1) directly influences the conditions leading to the construction of the *step-function*, the main function implemented in the environment.

In the Reinforcement Learning (RL) context, the step-function is the fundamental concept describing the transition between subsequent states within a dynamic environment. Each time the agent performs an action a_t , the step function determines the transition from the current state s_t to the next state s_{t+1} , representing the core interaction between the agent and the environment.

$$(r_{t+1}, s_{t+1}) = \text{Step}(s_t, a_t) \quad (6.2)$$

with:

$$\text{Step}(s_t, a_t) \text{ with probability } P(s_{t+1} | s_t, a_t)$$

where the transition probability is determined by the function $P(s_{t+1} | s_t, a_t)$, which, in a deterministic environment, equals 1 for a specific next state, whereas, in a stochastic environment, it will follow a probability distribution over multiple states.

In other words, the step-function represents how the environment responds to the agent's actions and how this response changes the environment's state. More precisely, the step function describes the logic through which the environment modifies its state as a consequence of the agent's action, and this state change serves as the foundation upon which the agent builds its future strategy.

Additionally, r_{t+1} represents the reward provided to the agent following the action taken: the agent does not learn directly from the environment, but rather through the feedback received after each action. State updating and reward calculation are mechanisms providing the agent with necessary information to modify its behavior in subsequent iterations to optimize its decision-making strategy.

Baseline Step function

How can the dynamics of employee assignment to projects be effectively modeled to make decisions within a reinforcement learning context? The step function embodies this idea by providing a mechanism for state transitions and reward calculation based on state-action interaction principles. It evaluates an action $a_i = (e_i, p_i)$, updates the environment's state s_i and calculates the corresponding reward r_i based on the validity of the assignment.

The main idea behind the implemented step function is to simulate the assignment of employees to required positions in projects, considering their competency levels, $l(e_i)$, and the competency levels of available project positions, $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$. If the assignment is valid, the environment updates the state of employees and projects and calculates a positive reward using a predefined reward function. Otherwise, the function advances the simulation while penalizing the agent with a negative reward. The function operates as follows:

1. State Variable Extraction

Retrieves:

- the competency level of the selected employee $l(e_i)$,

- the competency levels of available positions in the selected project $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$.

2. Compatibility Check

Compares the employee's competency level with the project positions' competency levels:

- If at least one match is found, assigns the employee to the first suitable project position. Specifically, updates the `_Assignments` matrix with the employee's ID in the corresponding position.

3. Reward Calculation

Uses the `_calculate_reward` function to compute the reward R_t , accounting for the assignment's contribution to the project and overall objectives.

4. Environment Update

- Increments the timestep i .
- Applies skill updates through `_update_skills`, reflecting the impact of assignments on employees' capabilities.

5. Termination Check

Checks if the episode is completed using `_check_done`, based on criteria such as completion of all projects or reaching a maximum number of timesteps.

6. Return Outputs

- **If the assignment is valid:** returns the updated state, reward r_t , termination flag, and an empty dictionary.
- **Otherwise:** penalizes the agent with a reward of -1 , advances the timestep, and returns the same set of values.

The `step` function models the dynamics of environment transitions guided by the agent's decisions, providing feedback in the form of a reward. The skill updates and reward signals steer the agent toward effective decision-making over time.

This framework supports on-policy or off-policy control methods, allowing the agent to progressively refine its policy through interactions with the environment. Policy improvement can leverage exploration strategies such as ϵ -greedy, ensuring convergence to an optimal assignment policy under appropriate conditions.

The algorithm described below (5 simulates the behavior of the implemented step function).

Algorithm 5 Baseline Step Function**Require:** Action $a_i = (e_i, p_i)$ with e_i employee and p_i project

- 1: Extract $l(e_i)$ (competency level of the employee)
- 2: Extract $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$ (competency levels of available positions in the project)
- 3: **for** each position $\rho_{i,k}$ in p_i **do**
- 4: **if** $l(e_i) == l(\rho_{i,k})$ **then**
- 5: Update $Assignments[i][k] \leftarrow e_i$ \triangleright Assign e_i to position ρ_k in p_i
- 6: Compute reward $r_i = \alpha + \beta S_i^M - \gamma c^e - P_i^e$
- 7: Increment timestep: $i \leftarrow i + 1$
- 8: Update skills:

$$s_{e_i,j}(i+1) = \begin{cases} s_{e_i,j}(i) + \eta, & \text{if } \rho_{i,j} = 1 \text{ and } s_{e_i,j}(i) < 1 \\ s_{e_i,j}(i), & \text{otherwise} \end{cases}$$
- 9: Check if episode terminated:
- 10: **if** $Assignments[j][l] \neq -1 \forall j, l$ **then**
 $done \leftarrow True$
- 11: **return** $s_{i+1}, r_i, done, \{\}$ \triangleright If no valid assignment
- 12: Penalize with reward $r_i \leftarrow -1$
- 13: Increment timestep: $i \leftarrow i + 1$
- 14: Update skills of all employees.
- 15: Check if episode terminated
- 16: **return** $s_{i+1}, r_i, done, \{\}$

Training Results

Figure (6.1) shows the results obtained during the agent's training phase, whereas Figure (6.2) illustrates the results from the testing phase.

In this section, the results of the experiment conducted under the Baseline hypothesis are presented.

Training Results

Figure (6.1) shows the results obtained during the agent's training phase, whereas Figure 6.2 illustrates the results from the testing phase.

In the first row of Figure (6.1), the *cumulative rewards* of the first 100 episodes and every subsequent 100 episodes are displayed. The entire training phase involves simulating *100,000 episodes*. The left graph shows that in the initial phases, the agent explores various solutions, obtaining significantly negative rewards (around -8000 points). However, as training progresses, the reward tends to become increasingly positive. This suggests that the agent is learning a more effective strategy to tackle the problem.

This trend is also confirmed by the two graphs below:

- Bottom-left graph: shows the *average number of timesteps* required to complete an episode as a function of the number of episodes. The value was averaged over a temporal window of *50 episodes* to reduce variability.
- Bottom-right graph: represents the *average final reward per episode*, also averaged over *50 episodes*.

Analyzing these two graphs, we can observe that:

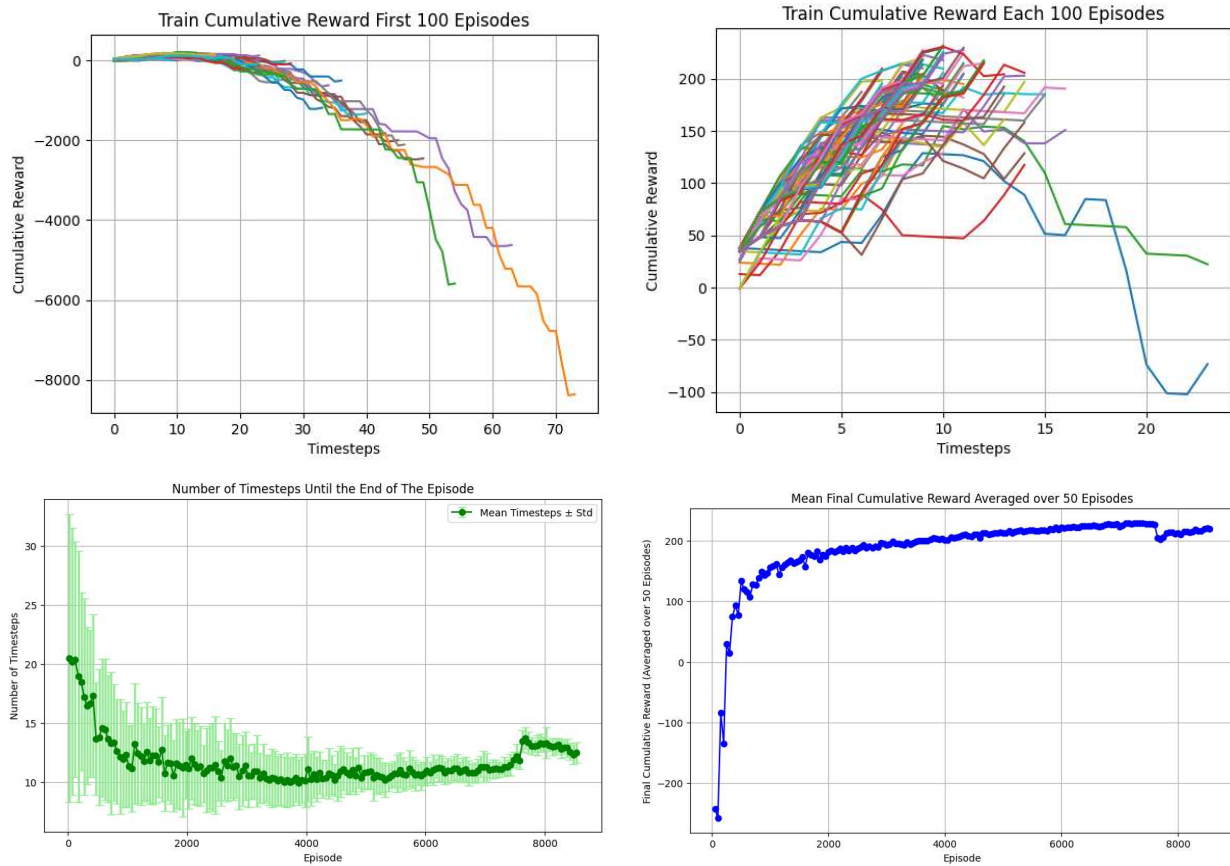


Figure 6.1: RL Agent Training Results

1. The average number of timesteps per episode decreases rapidly within the first 2000 episodes and stabilizes around episode 4000. This indicates that the agent is learning more efficient strategies to complete episodes in less time.
2. The standard deviation of timesteps also progressively decreases, indicating that the agent is exploring solutions increasingly consistent with the optimal policy.
3. After approximately 4000 episodes, there is a slight increase in the average convergence time. This could be due to various factors, such as:
 - A trade-off between *exploration* and *exploitation*, with the agent continuing to test slightly different solutions to further optimize the policy.
 - A possible *learning plateau*, where policy improvement becomes less significant and requires more episodes to fine-tune.

In the bottom-right graph, we observe that the average final reward starts from very low values but rapidly increases within the first 2000 episodes. After this initial phase, the reward continues to grow with a nearly linear trend. This behavior indicates that the agent has surpassed an initial learning phase, establishing an *effective baseline policy*, and from that point onward, it is refining its decisions with more gradual improvements. Additionally, the reward function, being expressed very simply, may not incentivize further significant improvements, resulting in a more stable progression.

Test Results

In Figure (6.2), the results for the testing phase are displayed: the top graph shows the *cumulative reward for the first 100 episodes*. Unlike the training phase, the agent achieves good results from

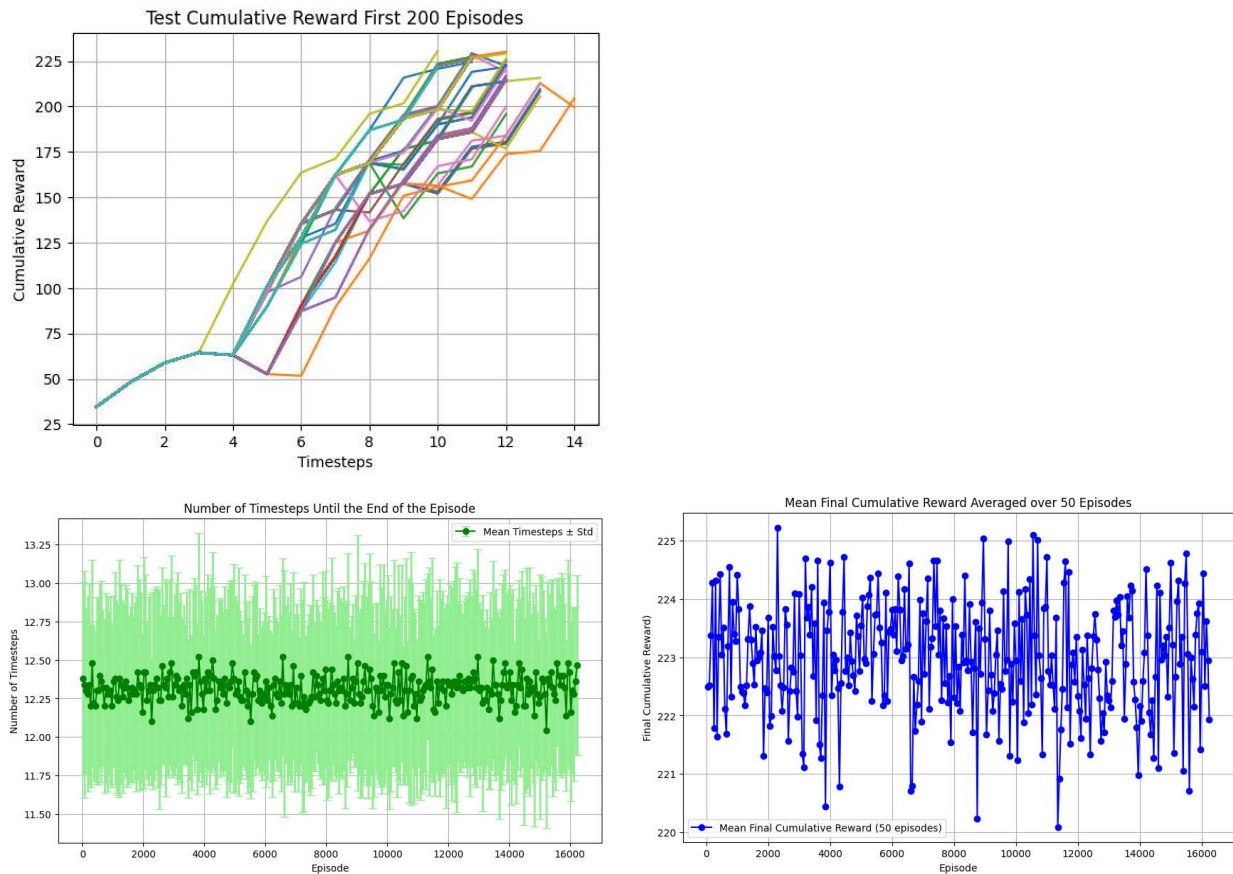


Figure 6.2: RL Agent Test Results

the very beginning. Despite these being the initial steps of the testing phase (which includes 200,000 episodes), the reward already reaches high values (around 200) within relatively few timesteps.

This observation is supported by the two graphs below:

- Bottom-left graph: the *average convergence time* and *standard deviation* maintain a stable trend, with values comparable to the best results obtained during training.
- Bottom-right graph: the *final reward per episode* shows similar stability, suggesting that the agent has learned an effective policy, which it now applies to achieve optimal allocation strategies, successfully generalizing the knowledge acquired during training.

We now compare significant values reported in the training and testing graphs. In particular, we evaluate performance based on:

- $\mu_t \pm \sigma_t$ average number of timesteps [Ts],
- M_t maximum number of timesteps [Ts],
- m_t minimum number of timesteps [Ts],
- $\mu_r \pm \sigma_r$ average final cumulative reward in points [P],
- M_r maximum final cumulative reward achieved [P].

Table 6.1 provides a numerical comparison between training and testing phase results.

Experimental Parameters

Table 6.2 reports the main parameters used during the training of the Reinforcement Learning agent.

Phase	μ_t [Ts]	M_t [Ts]	m_t [Ts]	μ_r [P]	M_r [P]
Train	11.84 ± 2.02	24.20	10.10	183.54 ± 79.55	227.05
Test	13.82 ± 0.26	14.80	13.18	209.25 ± 3.52	215.25

Table 6.1: Comparison of experimental results between Training and Testing phases under the baseline hypothesis

Parameter	Value
Number of employees N_e	6
Number of projects N_p	2
Number of positions N_ρ	5
Number of skills N_s	3
Number of levels N_l	4
Costs per level	$l/10$
Learning rate	0.03
Temporal range T	30
Max Iterations	500

Table 6.2: Parameters used for training the RL agent.

6.2 Staff Shortage: Under-sampling

In the second phase of the analysis, the hypothesis characterizing the under-sampling scenario was defined:

Hypothesis 2. *The number of employees is lower than the number of resources required by the projects.*

$$N_e < \sum^{N_p} N_\rho \quad (6.3)$$

In other words, it is assumed that there is a shortage of personnel relative to project needs, making a direct and complete allocation of the required resources impossible. This scenario, reflecting a more realistic situation in the field of human resource management, introduces greater complexity in assignment decisions. The agent must manage scarcity and seek strategies to minimize the impact of insufficient resources. The objective of this analysis is to evaluate the system's behavior under these conditions and compare it with the baseline scenario.

Unlike the baseline function, which operated in a context where the number of available employees was sufficient to cover the required project positions, the **step** function introduced here must balance *optimal* assignments, where an employee is suitable for the required project position, and *suboptimal* assignments, where an employee of a mismatched skill level must still be assigned. This will influence the reward returned to the agent, incorporating additional penalties.

Similarly to the baseline, the function evaluates the action $a_i = (e_i, p_i)$, updates the environment state, and calculates the reward based on the validity and quality of the assignment. However, in this case, the reward depends not only on the correctness of the assignment but also on the degree of fit between the employee and position, penalizing suboptimal decisions and unnecessary choices. The behavior of the step function is guided by the following logical structure:

1. Extraction of State Variables

The function retrieves:

- The competency level of the selected employee $l(e_i)$.

- The required competency levels for open positions in the project $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$.
- The availability of employees in the system.
- Activity timelines of the employee and the project.

2. Compatibility Check and Assignment

- If a position in the project compatible with the employee's competency level exists, and employees with that level are still available, the employee is assigned to the first available position. The assignment is recorded in the **Assignments** matrix, the reward is calculated similarly to the Baseline scenario, and the employee's timeline is updated accordingly.
- If the employee's competency level is not compatible with available positions, the iteration advances by penalizing the agent, and competencies are updated without making any assignment.

3. Managing Suboptimal Assignments

- If no employee with the required competency level is available, but the position remains vacant, the employee is assigned anyway, with a partial penalty on the reward reflecting the suboptimal nature of this decision.
- If the position is already occupied, the action is considered unnecessary, and the timestep advances without assignment or significant penalty.

4. Environment Update

Analogously to the previous scenario:

- Increment the timestep.
- Update employee competencies, modeling learning from experience.

5. Termination Check

- Check if the episode has concluded using different logic: an episode is considered complete if each position within every project has been assigned *or* no employees remain available.

6. Returning Outputs

- If the assignment is valid: return the updated state, calculated reward, termination flag, and an empty dictionary.
- If the assignment is suboptimal: return a reduced reward and advance the simulation.
- If no assignment is possible: penalize the agent with a negative reward and proceed to the next iteration.

Compared to the baseline, this function introduces greater variability in rewards to reflect assignment quality under resource scarcity. The agent must therefore not only seek valid assignments but also optimize decisions in a more constrained environment, encouraging adaptive and long-term learning strategies.

Algorithm (6) presents the logic implemented by the step function in the case of under-sampling.

Training Results

Analyzing the graphs related to cumulative rewards as a function of timesteps (first row of Figure 6.3) allows for some preliminary considerations. In the first 100 episodes (left figure), no significant differences are observed between the two scenarios. However, when examining the graph representing cumulative rewards every 100 episodes during the training phase, it becomes evident that in the case of undersampling, there is a greater number of episodes characterized by strongly negative final rewards

Algorithm 6 Under-sampling Step Function

Require: Action $a_i = (e_i, p_i)$ with e_i employee and p_i project

- 1: Extract $l(e_i)$ (competency level of the employee)
- 2: Extract $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$ (competency levels of available positions in the project)
- 3: **for** each position $\rho_{i,k}$ in p_i **do**
- 4: **if** $l(\rho_{i,k})$ still has available employees **then** ▷ Check availability
- 5: **if** $l(e_i) == l(\rho_{i,k})$ **then** ▷ Level matching check
- 6: Update $Assignments[i][k] \leftarrow e_i$ ▷ Assign e_i to position $\rho_{i,k}$ in p_i
- 7: Calculate reward $r_i = \alpha + \beta S_i^M - \gamma c^e - P_i^e$
- 8: Increment timestep: $i \leftarrow i + 1$
- 9: Update skills: $s_{e_i,j}(i + 1)$
- 10: Check termination:
- 11: **if** $Assignments[j][l] \neq -1 \forall j, l$ **and** no employees are available **then**
- 12: $done \leftarrow True$
- 13: **return** $s_{i+1}, r_i, done, \{\}$
- 14: **else** ▷ Levels do not match
- 15: Update Skills
- 16: $i \leftarrow i + 1$
- 17: Check termination.
- 18: **Continue**
- 19: **else** ▷ No more employees available for required position
- 20: **if** $\rho_{i,k}$ has not been assigned: **then**
- 21: Update $Assignments[i][k] \leftarrow e_i$
- 22: Penalize with $r_i \leftarrow r_i - \kappa$
- 23: $i \leftarrow i + 1$
- 24: Check termination.
- 25: **return** $s_{i+1}, r_i, done, \{\}$
- 26: **else**
- 27: Update Skills ▷ Assignment operation not needed
- 28: $i \leftarrow i + 1$
- 29: Check termination.
- 30: **Continue**

compared to the baseline. This phenomenon suggests that the agent requires a higher number of timesteps to learn an optimal strategy.

This behavior is further confirmed by the two graphs below, which represent the trend of the number of timesteps and the final cumulative reward (averaged over a 50-episode window) as a function of the number of episodes. Specifically, it is observed that, in the case of undersampling, both graphs show significantly slower and more irregular decreasing and increasing trends, respectively, compared to the baseline. While in the baseline the model reached optimal values after approximately 2000 episodes, in the current case the learning process appears more prolonged and unstable.

Test Results

Let's now analyze the Test phase under the Undersampling hypothesis. From examining the graph of cumulative rewards in the first 200 episodes as a function of timesteps (first row of Figure 6.4), greater variability of trajectories is evident compared to the baseline test phase. This behavior indicates a lower generalization capability of the model in this scenario, suggesting that the agent struggles to effectively transfer the strategies learned during training to new episodes.

However, when analyzing the number of timesteps per episode, it is observed that this number is, on average, lower compared to the baseline, although characterized by higher variability. This

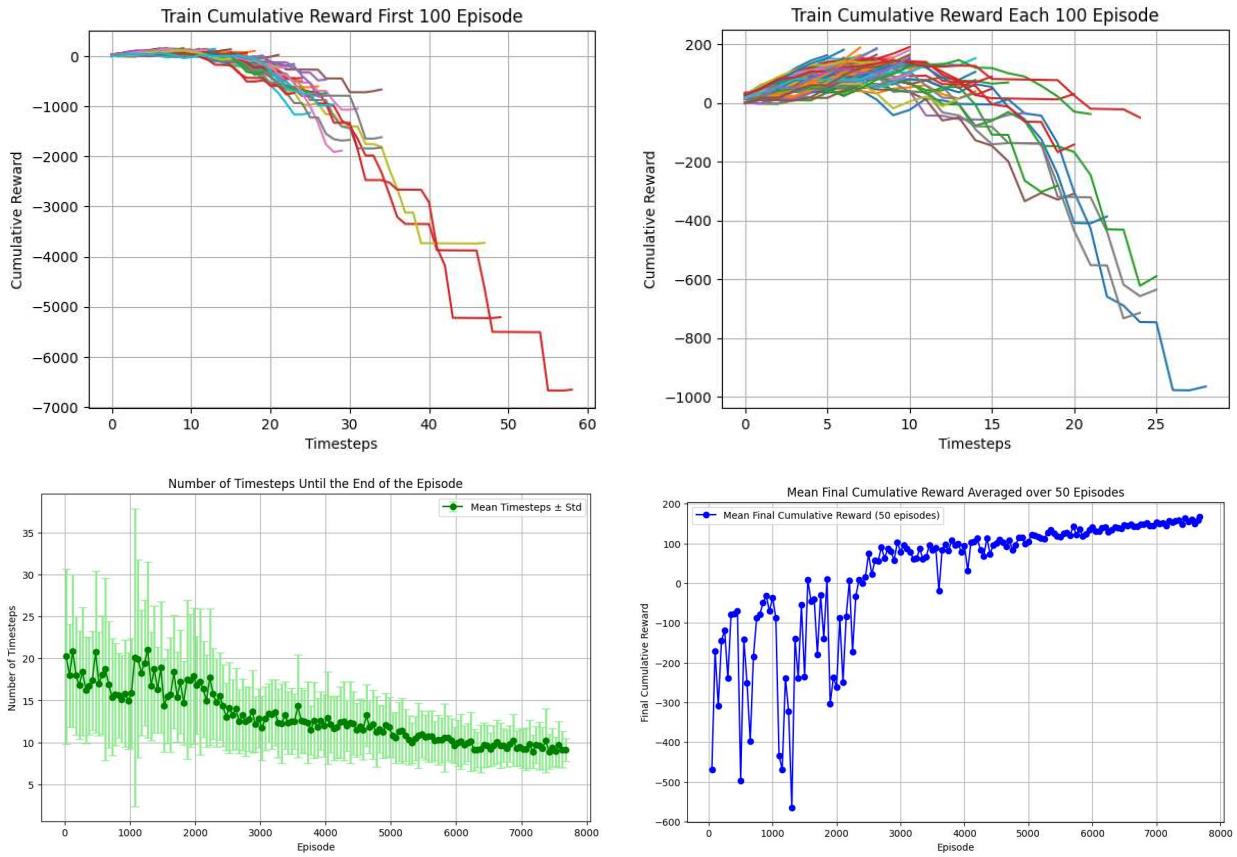


Figure 6.3: RL Agent Training Results under undersampling assumptions

phenomenon is a direct reflection of the additional condition introduced in the episode termination function: whereas in the baseline `_step_function`, the only termination condition was that each position $\rho_{i,k}$ was assigned, ensuring the completion of every project, under the undersampling hypothesis, an additional constraint is introduced—namely, that there are no more available employees. In a context with limited resources, the agent may opt for quicker, less exploratory strategies, concluding episodes in fewer steps without necessarily reaching an optimal solution. Moreover, the model may be less inclined to take actions that extend the episode in an attempt to maximize reward, resulting in shorter but less effective convergence dynamics.

Concerning the final cumulative reward per episode, it is observed that, on average, it is lower than the baseline, with negative peaks reaching very low values (around 130-120) compared to values consistently above 200 in the baseline. This behavior can be attributed to the model’s difficulty in learning robust strategies in a context of limited resources. Specifically, the agent may develop suboptimal policies that, despite reducing the number of timesteps per episode, do not lead to effective allocations, thus resulting in significantly lower cumulative rewards compared to the baseline scenario. Additionally, the increased reward variance indicates that the agent may not have developed a sufficiently stable strategy, leading to inconsistent performance across different episodes.

In conclusion, the analysis of the graphs suggests that when the number of employees is lower than the number of available positions, the reinforcement learning model struggles more to converge towards an optimal solution. This is likely due to the increased complexity of the problem, longer exploration periods, and greater variability in allocation strategies. Nonetheless, despite slower convergence, the model still shows a positive trend, suggesting that with a sufficient number of episodes, it could reach performance competitive with the baseline scenario.

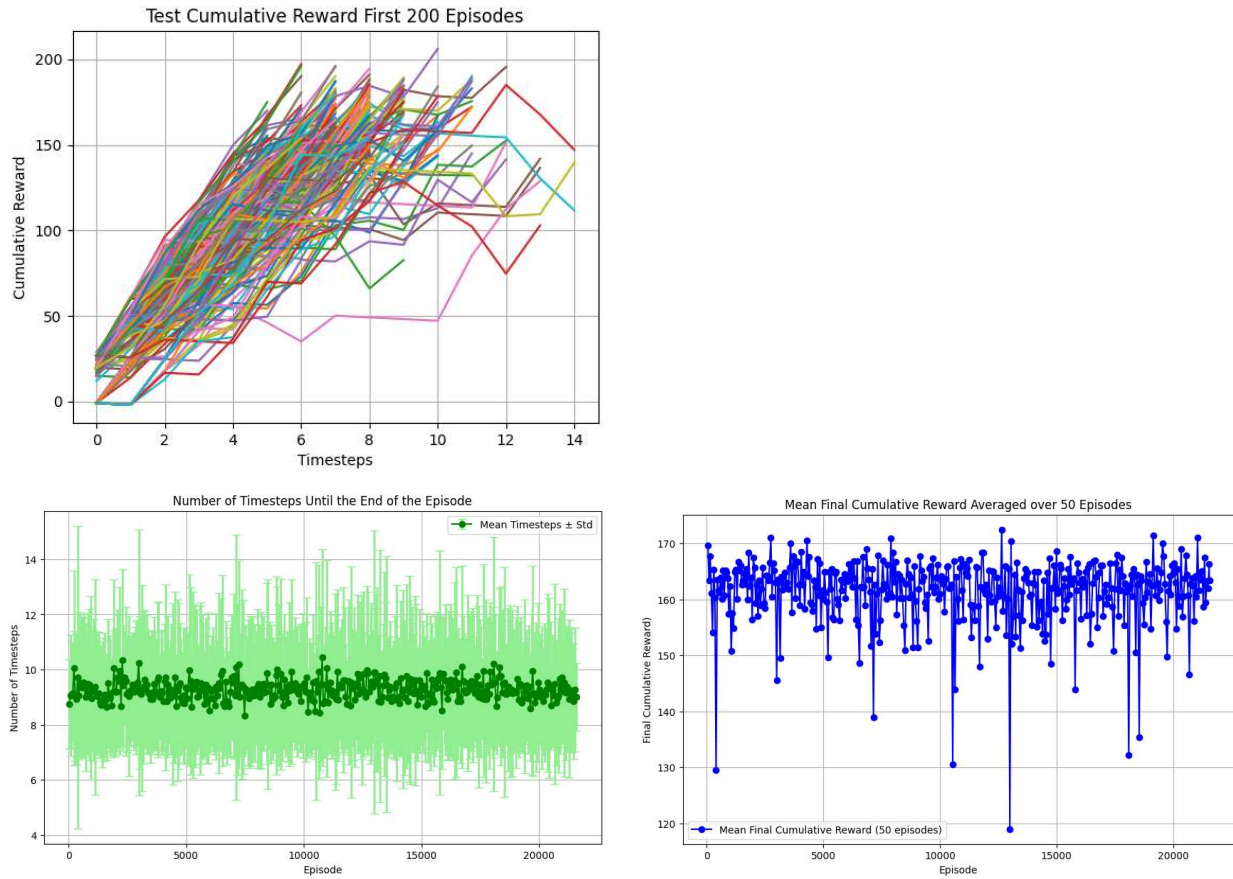


Figure 6.4: RL Agent Test Results under undersampling hypothesis

Phase	μ_t [Ts]	M_t [Ts]	m_t [Ts]	μ_r [P]	M_r [P]
Train	13.24 ± 2.84	20.52	8.94	49.21 ± 116.89	165.37
Test	9.84 ± 1.79	14.08	8.38	166.49 ± 42.85	182.26

Table 6.3: Comparison between experimental results of test and Train under under-sampling assumptions

6.3 Project Shortage: Over-sampling

In the final phase of the analysis, the hypothesis characterizing the over-sampling scenario was defined:

Hypothesis 3. *The number of employees is equal to or greater than the number of resources required by the projects.*

$$N_e > \sum^{N_P} N_\rho \quad (6.4)$$

or significantly higher in terms of magnitude.

In other words, this scenario assumes that the number of available employees exceeds the resources necessary for completing projects, introducing an additional scenario closer to a realistic situation. This experiment, representing an abundance of resources, reflects possible challenges a company may face, such as operational inefficiencies and unnecessarily elevated costs. This configuration will be compared to the Baseline scenario to evaluate the impact of an oversupply of resources relative to the actual demand from projects.

Over-sampling Step Function

Unlike the baseline function, which operated within a context of equilibrium between resource supply and demand, the step function in the case of over-sampling handles a scenario where the number of available employees exceeds the number of required positions within projects. The primary goal, therefore, is to optimize resource allocation by preventing inefficiencies arising from personnel under-utilization and balancing employee assignments with the actual requirements of the projects. This objective is reflected in the reward system, which incentivizes correct and productive assignments while introducing penalties for unnecessary or redundant decisions.

Similar to the baseline, the function evaluates the action $a_i = (e_i, p_i)$, updates the environment state, and calculates the reward based on the validity of the assignment. However, in this scenario, optimization involves not only the effective allocation of resources but also managing personnel surplus, introducing penalties for assigning employees when it is not strictly necessary.

The behavior of the step function follows this logical structure:

1. Extraction of State Variables

The function retrieves:

- The competency level of the selected employee $l(e_i)$.
- The required levels for the open positions in the project $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$.
- Activity timelines for the employee and the project.

2. Compatibility Check and Assignment

- If the employee's level matches one of the project's positions, the assignment is made and recorded in the **Assignments** matrix.
- The reward is calculated based on the quality of the assignment and the compatibility between skills and required positions.
- If the project is fully assigned for the first time, a *proportional bonus* is applied based on the temporal coherence of assignments, incentivizing the agent to optimize project completion.

3. Management of Resource Excess

- If a project is completed but there are still unassigned employees, an additional penalty based on the sum of residual employee availability (*unallocated penalty*) is applied.

4. Environment Update

- Advances the timestep and updates employee skills to model learning.
- Checks if a project has been completed and assigns any relevant *bonus*.

5. Termination Check

- The episode is considered complete when all positions in projects have been assigned. However, if there are still available employees who have not been assigned, a penalty equal to the sum of their remaining availability is applied.

6. Returning Outputs

- If the assignment is valid: returns the updated state, calculated reward (including bonuses and penalties), termination flag, and an empty dictionary.
- Returns a negative reward equal to -1 along with any applicable penalties (only in case of episode termination), advancing the timestep regardless.

Compared to the baseline, this function introduces greater sensitivity to inefficiencies caused by personnel surplus, incentivizing the agent to select optimal allocation strategies. The goal is not only to maximize coverage of positions but also to minimize unnecessary resource assignments, avoiding redundancy and optimizing workforce utilization.

Algorithm 7 Over-sampling Step Function

Require: Action $a_i = (e_i, p_i)$ with employee e_i and project p_i

Require: Current state s_i , Assignments $Assignments$, employee skills s_{e_i}

```

1: Extract  $l(e_i)$  (competency level of employee)
2: Extract  $\{l(\rho_{i,1}), \dots, l(\rho_{i,N_\rho})\}$  (required levels for open positions in project  $p_i$ )
3: for each position  $\rho_{i,k}$  in  $p_i$  do
4:   if  $l(e_i) == l(\rho_{i,k})$  then
5:     Update  $Assignments[i][k] \leftarrow e_i$  ▷ Assign  $e_i$  to position  $\rho_k$  in  $p_i$ 
6:     Calculate reward  $r_i = \alpha + \beta S_i^M - \gamma c^e - P_i^e$ 
7:     Update employee availability  $t_{e,i} = 0$ 
8:     if  $p_i$  has been fully assigned: then
9:       if  $p_i$  is completed for the first time: then
10:        Calculate bonus:  $B \leftarrow 1.2 + \sum_{k=1}^{N_\rho} \sum_{j \in Assignments(p_i)} t_{e_j} \cdot t_{\rho_{i,k}}$  ▷ Matching timeline + 20%
11:         $r_i += B$ 
12:         $i \leftarrow i + 1$ 
13:        Update skills
14:        Check termination.
15:        return  $s_{i+1}, r_i, done, \{\}$  ▷ If no valid assignment
16: Penalize with reward  $r_i \leftarrow -1$ 
17: Increment timestep:  $i \leftarrow i + 1$ 
18: Update skills for all employees.
19: Check if episode has terminated
20: return  $s_{i+1}, r_i, done, \{\}$ 

```

Training Results

In Figure (6.5), several analyses of the agent’s behavior under oversampling conditions are presented. In the first row, the graphs show the trend of cumulative rewards per episode as a function of timesteps. The most prominent feature in the first graph on the left, covering the first 100 episodes, is a sharp decline in rewards at the end of each episode. This phenomenon results from a penalty applied to the agent, proportional to the number of employees left unassigned at the end of the episode. This penalty incentivizes the agent to maximize efficiency in resource allocation. The graph on the right, illustrating cumulative rewards every 100 episodes, demonstrates a progressive reduction in the step decline observed previously. This result indicates that the agent is improving its capability to optimize the temporal matching between employee availability and project activities, gradually decreasing the number of unallocated resources.

Moving to the second row of the figure, we find a graph representing the number of timesteps required to complete each episode as a function of the number of episodes (left side). Compared to the baseline scenario, there is an earlier stabilization of this trend, with average values remaining higher, around 25 timesteps per episode. This behavior suggests that, in the oversampling scenario, the agent takes more steps to complete an episode due to the increased complexity of allocation decisions arising from the surplus of employees relative to available positions. Another essential aspect visible from this graph is the lower total number of episodes compared to both the baseline and undersampling cases, consistent with the constraint imposed on the model of not exceeding 100,000 total timesteps. Since

the average number of timesteps per episode is higher in oversampling conditions, the total number of episodes is consequently lower.

A final observation concerns the graph on the right in the second row, showing the final reward per episode as a function of the number of episodes. Here again, compared to the baseline, the progress appears slower and more uncertain, particularly in early stages. However, in the final episodes, reward values reach significantly higher peaks, stabilizing around 300 points compared to 220 in the baseline scenario. This improvement is attributable to the bonus assigned to the agent when a project is fully allocated for the first time. The agent gradually learns to optimally exploit this mechanism, avoiding reassignments of resources to already-completed projects, which would yield no further bonus. This dynamic leads the agent to prioritize more urgent projects, thus optimizing the overall decision-making process.

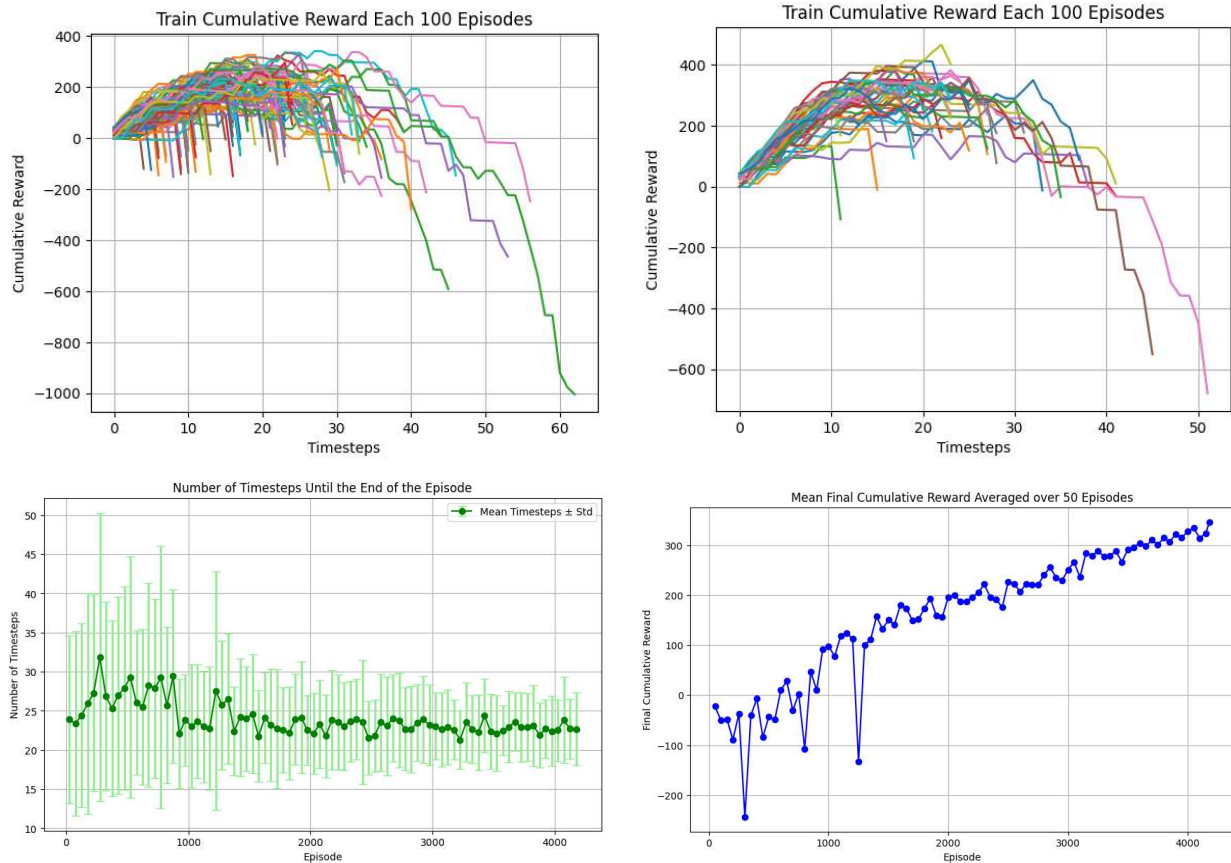


Figure 6.5: RL Agent Training Results

Test Results

The test phase, shown in Figure 6.6, aligns with observations made in previous experiments, reflecting the improved behavior learned by the agent during the training phase. This outcome is justified by the very nature of reinforcement learning, where the agent, having already explored the environment and refined its decision-making strategy, can replicate optimized behaviors acquired earlier during testing. In the case of oversampling, these behaviors further highlight the agent's capability to manage a complex and varied environment.

In particular, all previous observations made regarding oversampling remain valid. The graph in the first row, depicting the cumulative reward of the first 200 episodes, shows greater variability compared to the baseline scenario. This phenomenon reflects the agent's difficulty in learning within an environment characterized by a significantly higher number of employees than required positions.

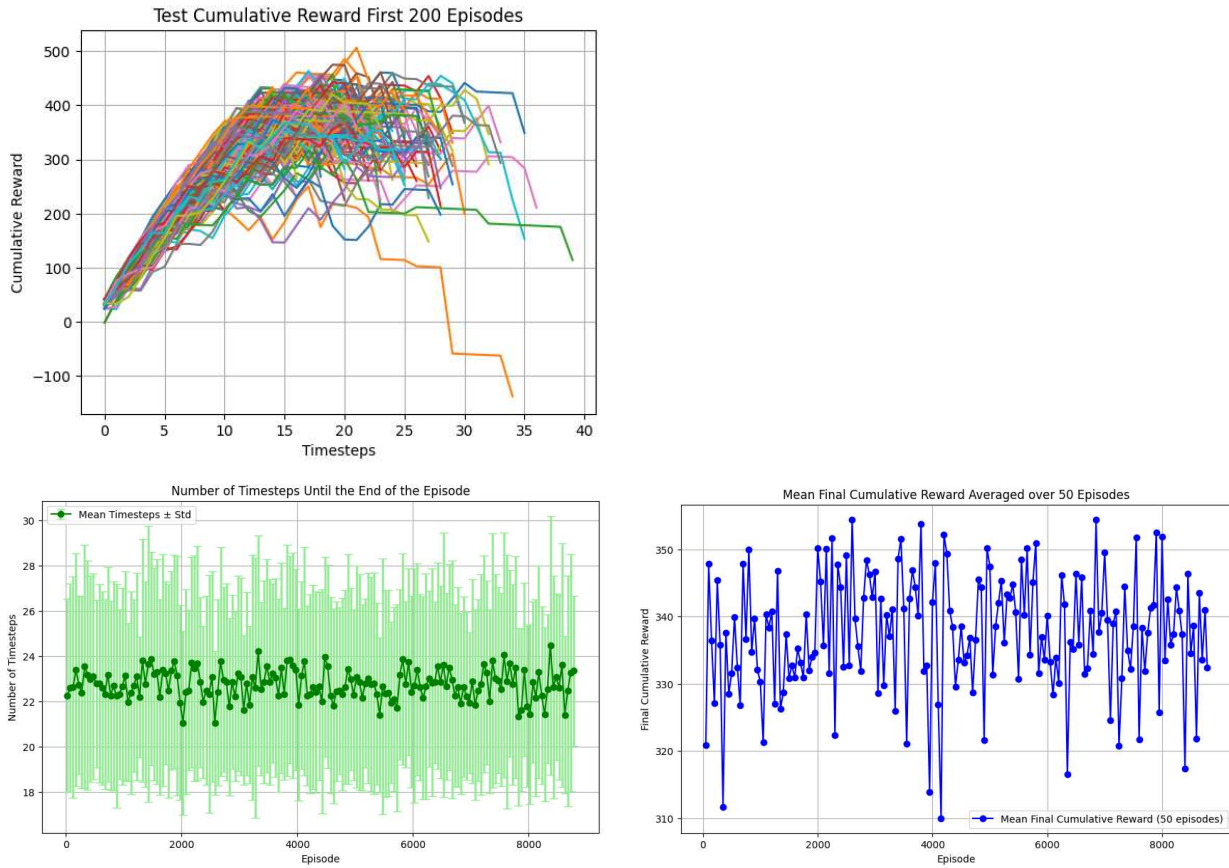


Figure 6.6: RL Agent Test Results

However, the graph also shows strongly positive reward peaks, reaching values up to 500 points. This demonstrates that, despite the complexity, the agent can effectively exploit the opportunities offered by the environment, maximizing the reward in specific episodes.

The graph in the bottom left, representing the number of timesteps needed to complete each episode, indicates a stabilization around an average of 25 steps. However, compared to the baseline, the standard deviation is roughly doubled. This result was expected given the greater complexity of the problem and the necessity for the agent to explore more assignment combinations to optimize its decisions.

Lastly, the graph on the bottom right, showing the final reward per episode, highlights behavior consistent with that observed in the training phase. Specifically, the reward tends to stabilize on average above 300 points, significantly surpassing the baseline values. This result underscores the agent's ability to capitalize on the bonus associated with completing projects for the first time. Such dynamics demonstrate how the agent has learned to prioritize allocating resources toward the most urgent projects, thereby improving overall efficiency.

Phase	μ_t [Ts]	M_t [Ts]	m_t [Ts]	μ_r [P]	M_r [P]
Train	22.92 ± 2.34	28.64	19.12	176.61 ± 133.27	341.03
Test	21.06 ± 1.46	23.30	19.86	338.54 ± 7.65	356.79

Table 6.4: Comparison between experimental results of test and Train under over-sampling assumptions

6.4 Conclusions

This thesis fits into the broader context of Reinforcement Learning research, providing a new contribution. The conducted work led to the development of a novel simulation environment, capable of faithfully representing the complexity of an IT consulting company in managing and allocating human resources. This custom-designed environment stands out in the literature for its ability to realistically model operational dynamics and business constraints, offering a credible testing ground for training intelligent agents.

The experimental results obtained using the PPO agent demonstrate not only the effectiveness of the adopted approach but also the agent's capability to learn optimal resource allocation strategies, progressively improving its decisions. By extending the analysis to more complex scenarios such as oversampling and undersampling, the agent showed remarkable robustness, adapting to new variables and maintaining performance comparable to the baseline scenario. This confirms the solidity and flexibility of the developed environment. Additionally, by varying key parameters—such as reducing personnel or projects—it was observed that the system's behavior could be brought back in line with the initial results, further strengthening the validity of the proposed model.

Looking ahead, this work outlines a pathway toward concrete applications, suggesting that integrating reinforcement learning techniques with realistic simulations can provide decisive tools for optimizing corporate resources.

Beyond the achieved results, numerous directions for further development emerge. Specifically, introducing operational thresholds sensitive to the business context, capable of suggesting hiring new personnel or expanding into new markets based on resource utilization levels, could enrich the environment with predictive capabilities useful for anticipating management risk situations. Additional progress might arise from structured dialogue with the personnel currently managing resources manually, aiming to define comparative metrics that integrate human expertise with proposed solutions. In this context, the presented work establishes a solid foundation upon which more informed and effective decision-making tools can be built.

Whether and how these tools can substantially contribute to evolving resource allocation processes remains an open question. Nevertheless, every step taken in this direction brings us closer to a new vision for corporate management, capable of combining advanced technology with practical operations, thus broadening the horizon of possibilities offered by artificial intelligence applied to real-world scenarios.

Ringraziamenti

Desidero esprimere la mia sincera gratitudine a tutte le persone che mi hanno accompagnato e supportato durante questo tanto travagliato quanto meraviglioso percorso di Laurea.

Innanzitutto un ringraziamento speciale va alla mia famiglia, per l'amore, il sostegno incondizionato e la pazienza che mi hanno sempre dimostrato. L'amore per la Fisica e la Scienza trasmessomi da mio Padre è stato il motore che mi ha spinto durante questi anni, ad affrontare ogni argomento con rinnovata e sincera curiosità, d'altro lato la perseveranza e lo spirito di adattamento di mia madre mi hanno permesso di non arrendermi anche quando si presentavano montagne apparentemente invalicabili. Senza di loro, questo percorso non sarebbe stato in alcun modo possibile.

Desidero ringraziare poi il mio relatore, GianAntonio Susto, per aver creduto in questo progetto e avermi permesso di sviluppare una tesi in un ambito così innovativo nonostante i dubbi e le difficoltà nella definizione stessa del progetto. Un sentito grazie anche alla mia correlatrice, Marina Ceccon, per la sua attenzione ai dettagli e i suoi preziosi consigli, che mi hanno permesso di affrontare con precisione ma anche leggerezza la fase di stesura della tesi; in tal senso vorrei ringraziare anche il mio referente di progetto interno all'azienda Antonio Castaldo D'ursi, che è stato in grado di ricavarmi, non con poche difficoltà, un ambiente lavorativo da un lato stimolante, e dall'altro che mi permettesse di portare avanti gli studi con tranquillità. Per questo un profondo e sincero grazie.

Un pensiero particolare va anche ai miei amici tutti, gli amici della Spartan e le mie allieve delle Patavium, per tutte le serate in cui, consapevolmente o meno, mi hanno fornito conforto nei momenti tristi e hanno festeggiato con me nei momenti felici.

A Daniele, per l'infinta pazienza, il continuo stimolo di crescita, l'amore e la comprensione.

Bibliography

- [1] Ashish Ahire and Mustafa Abdallah. Reinforcement learning for enhancing human security resource allocation in protecting assets with heterogeneous losses. In *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems*, pages 9–15, 2023.
- [2] Richard Bellman. On the theory of dynamic programming. *Proceedings of the national Academy of Sciences*, 38(8):716–719, 1952.
- [3] Richard Bellman. Dynamic programming. *science*, 153(3731):34–37, 1966.
- [4] Peter F Boxall, John Purcell, and Patrick M Wright. *The Oxford handbook of human resource management*. Oxford university press, USA, 2007.
- [5] Jeffrey L Brewer and Kevin C Dittman. *Methods of IT project management*. Purdue University Press, 2018.
- [6] P.M. Bulla, D.N. Scott. Manpower requirements forecasting: A case example. In *Strategic Human Resource Planning Applications*, 1987.
- [7] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In *Machine learning techniques for multimedia: case studies on organization and retrieval*, pages 21–49. Springer, 2008.
- [8] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [9] Jin Fang, Jiacheng Weng, Yi Xiang, and Xinwen Zhang. Imitate then transcend: Multi-agent optimal execution with dual-window denoise ppo. *arXiv preprint arXiv:2206.10736*, 2022.
- [10] Zoubin Ghahramani. Unsupervised learning. In *Summer school on machine learning*, pages 72–112. Springer, 2003.
- [11] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.
- [12] P.O. Gray. *Psychology*. International edition. Worth, 2010.
- [13] Ben Hambly, Renyuan Xu, and Huining Yang. Recent advances in reinforcement learning in finance. *Mathematical Finance*, 33(3):437–503, 2023.
- [14] J.H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence (Complex Adaptive Systems)*. Bradford Books, 1992.
- [15] David R. Howell and Arne L. Kalleberg. Declining job quality in the united states: Explanations and evidence. *RSF: The Russell Sage Foundation Journal of the Social Sciences*, 5(4):1–53, 2019.
- [16] Jinyang Jiang, Xiaotian Liu, Tao Ren, Qinghao Wang, Yi Zheng, Yufu Du, Yijie Peng, and Cheng Zhang. Deep reinforcement learning for solving management problems: Towards a large management mode. *arXiv preprint arXiv:2403.00318*, 2024.

- [17] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, December 1984.
- [18] Ozsel Kilinc and Giovanni Montana. Reinforcement learning for robotic manipulation using simulated locomotion demonstrations. *Machine Learning*, pages 1–22, 2022.
- [19] 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.
- [20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [21] Reuf Kozlica, Georg Schäfer, Simon Hirländer, and Stefan Wegenkittl. A modular test bed for reinforcement learning incorporation into industrial applications. In *International Data Science Conference*, pages 99–101. Springer, 2023.
- [22] Matthias Lehmann. The definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations. *arXiv preprint arXiv:2401.13662*, 2024.
- [23] Haoqing Luo, Yiming Bie, and Sheng Jin. Reinforcement learning for traffic signal control in hybrid action space. *IEEE Transactions on Intelligent Transportation Systems*, 25(6):5225–5241, 2024.
- [24] Saul Mcleod. Operant conditioning: What it is, how it works, and examples. *Diakses pada situs Simply Psychology* <https://www.simplypsychology.org/operant-conditioning.html#:~:text=Positive%20reinforcement%20is%20a%20term,reward%20is%20a%20reinforcing%20stimulus>, 2023.
- [25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.
- [26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
- [27] Phong Nguyen, Matsuba Hiroya, Tejdeep Hunabad, Dmitrii Zhilenkov, Hung Nguyen, and Khang Nguyen. Can reinforcement learning solve a human allocation problem? 2021.
- [28] JOANNA O’Riordan. Workforce planning in the irish public service. *State of the Public Services Series. Research Paper*, 7, 2012.
- [29] Ciprian Paduraru, Miruna Paduraru, and Catalina Camelia Patilea. Task distribution and human resource management using reinforcement learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 96–101. IEEE, 2021.
- [30] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dornmann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.
- [31] T. Reilly, P. Williams. *Strategic HR*. Routledge, 2007.
- [32] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, 1952.
- [33] John Schulman. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

- [34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [35] Alessandro Sebastianelli, Massimo Tipaldi, Silvia Liberata Ullo, and Luigi Glielmo. A deep q-learning based approach applied to the snake game. In *2021 29th Mediterranean Conference on Control and Automation (MED)*, pages 348–353. IEEE, 2021.
- [36] David Silver and Demis Hassabis. Alphago: Mastering the ancient game of go with machine learning. *Research Blog*, 9, 2016.
- [37] Nazneen N Sultana, Hardik Meisheri, Vinita Baniwal, Somjit Nath, Balaraman Ravindran, and Harshad Khadilkar. Reinforcement learning for multi-product multi-node inventory management in supply chains. *arXiv preprint arXiv:2006.04037*, 2020.
- [38] Richard S Sutton. Reinforcement learning: An introduction. *A Bradford Book*, 2018.
- [39] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.
- [40] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.
- [41] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.
- [42] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.
- [43] G. Willis. Robust workforce planning framework:update from practice. In *CfWI technical paper series no. 0010*. Centre for Workforce Intelligence, 2014.
- [44] Graham Willis, Siôn Cave, and Martin Kunc. Strategic workforce planning in healthcare: A multi-methodology approach. *European Journal of Operational Research*, 267(1):250–263, 2018.
- [45] Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.