Alma Mater Studiorum · Università di Bologna

**SCUOLA DI INGEGNERIA**
**Corso di Laurea Magistrale in Ingegneria Informatica**

# A Reinforcement Learning Agent for Distributed Task Allocation

**Relatore:**
**Chiar.mo Prof.**
**PAOLO BELLAVISTA**

**Candidato:**
**ALESSANDRO STAFFOLANI**

**Correlatore:**
**Chiar.mo Prof.**
**MIRCO MUSOLESI**

**Correlatore:**
**VICTOR DARVARIU**

**Sessione**
**Anno Accademico 2018-2019**

# Abstract


## Italian version

Al giorno d'oggi il *reinforcement learning* ha dimostrato di essere davvero molto efficace nel *machine learning* in svariati campi, come ad esempio i giochi, il riconoscimento vocale e molti altri. Perciò, abbiamo deciso di applicare il *reinforcement learning* ai *problemi di allocazione*, in quanto sono un campo di ricerca non ancora studiato con questa tecnica e perchè questi problemi racchiudono nella loro formulazione un vasto insieme di sotto-problemi con simili caratteristiche, per cui una soluzione per uno di essi si estende ad ognuno di questi sotto-problemi.

In questo progetto abbiamo realizzato un applicativo chiamato *Service Broker*, il quale, attraverso il *reinforcement learning*, apprende come distribuire l'esecuzione di tasks su dei lavoratori asincroni e distribuiti. L'analogia è quella di un cloud data center, il quale possiede delle risorse interne - possibilmente distribuite nella server farm -, riceve dei tasks dai suoi clienti e li esegue su queste risorse. L'obiettivo dell'applicativo, e quindi del data center, è quello di allocare questi tasks in maniera da minimizzare il costo di esecuzione.

Inoltre, al fine di testare gli agenti del *reinforcement learning* sviluppati è stato creato un environment, un simulatore, che permettesse di concentrarsi nello sviluppo dei componenti necessari agli agenti, invece che doversi anche occupare di eventuali aspetti implementativi necessari in un vero data center, come ad esempio la comunicazione con i vari nodi e i tempi di latenza di quest'ultima.

I risultati ottenuti hanno dunque confermato la teoria studiata, riuscendo a

ottenere prestazioni migliori di alcuni dei metodi classici per il task allocation.

## English version

Nowadays *reinforcement learning* has demonstrated to be very effective in machine learning in several fields, such as games, speech recognition and many others. Therefore, we have decided to apply *reinforcement learning* to *allocation problems*, because they are a research field which has not been studied yet with *reinforcement learning* and because these problems in their formulation contain a wide set of sub-problems with similar characteristics; thus a solution for one of them is extended to all these sub-problems.

In this project we have created an application called *Service Broker*, which, through *reinforcement learning*, learns how to distribute the execution of tasks on asynchronous and distributed workers. The analogy is that of a cloud data center, which owns internal resources - possibly distributed in the server farm -, receives tasks from its clients and executes them on those resources. The objective of the application, thus of the data center, is to allocate those tasks in a way to minimize the cost of execution.

Moreover, with the aim of testing the *reinforcement learning* agents developed, has been created an environment, a simulator, which allows to concentrate on the development of the components necessary to the agents, instead of having to concentrate on additional implementation aspects necessary in a real data center, such as the communication between the nodes and the latency caused by it.

The results obtained have confirmed the theory studied, since we have obtained better performances than those achieved with the classic methods used for *task allocation*.

# Introduction

Nowadays *reinforcement learning* has demonstrated to be very effective in learning how to solve several different kinds of problems, such as games, speech recognition, web service personalization and many others. Therefore, the promising results obtained by such learning techiniques have motivated many other researchers to apply it to new types of problems with the aim of finding new methods which solve them with better results.

Thus, we have decided to apply *reinforcement learning* to a family of problems, *allocation problems*, in particular to distributed task allocation, which, at the best of our knowledge, has never been studied through this learning technique. The reasons for studying such a problem are manifolds. Firtsly, it is a promising research field because nobody is working on it. Secondly, it is a challenging case study because it can be generalized to a plethora of real world scenarios, such as the allocation of deliveries in a carrier delivery service, or the assignment of goods for manufacturing products, just to name a few.

The following work has been structured in four chapters, which are aggregated in two main parts. In the former, we will start from the definition of *allocation problems* with all its variants. Subsequently, we will give an overall introduction on *reinforcement learning* and its typical solution methods. Finally, we will present the state of the art related to both, *allocation problems* and *reinforcement learning*, describing the main solutions and the main results which have been obtained. In the latter we will describe the actual work proposed; we will start formulating a specific *allocation problem*, which will

be addressed during the whole project. Afterwards, we will present how we map the case study in the field of *reiforcement learning*, deeply showing all the aspects which have guaranteed to solve the problem through *reiforcement learning*. Subsequently, we will introduce firstly the environment that we have developed to allow us to simulate the *reinforcement learning* solution proposed, then we will present the implementation details related to both the environment and the *reinforcement learning* agents. Finally, we will show the results obtained by our solution, demonstrating how the *reiforcement learning* proposal behaves better than the typical *allocation problems* techniques.

The contribution which we would like to give to the research with this work consists of two different aspects. Firstly, we have developed an environment which can be used for simulating this kind of problems. Thus, it allows the research community to continue working on *allocation problems* through *reinforcement learning*, improving the results obtained. Secondly, an initial contribution is given by the solution proposed, in which several different *reinforcement learning* methods have been applied to the problem showing the huge potentiality of *reinforcement learning* as the main solution for solving *allocation problems*.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Problem presentation and relevance reasons

In this chapter, we firstly present *allocation problems* with their variants, their main applications and the challenges encountered when we try to solve it in a distributed environment or we have to allocate tasks in a decoupled and asynchronous system. Moreover, we present possible applications in scenarions which are completely different from computing systems, such as the carrier delivery services or the market allocation of investments, just to cite a few.

Secondly, we deal with *reinforcement learning*, illustrating the reasons for its sucess and the reasons for which we consider it as a possible solution for task allocation problems. Moreover, we present the reinforcement learning problem formulation passing through the simplified *multi-armed bandit* problem, and then through the *contextual bandit* problem to finally arrive at the full reinforcement learning problem definition.

Finally, in this chapter, we illustrate the methodology used to study, approach and evaluate the allocation task problem using reinforcement learning.

## 1.1 Allocation problem

The *allocation problem*, which is a fundamental combinatorial optimization problem, involves distributing the available resources between different tasks, or jobs, in order to minimize total costs or maximize the total return. In such a problem we have the following components: a set of resources available in a given amount; a set of tasks to be done, each requiring a specified amount of resources; a cost or return associated to the execution of a task using a resource. The problem is to determine how much of each resource has to be allocated to each task.

If the amount of jobs is lower than the amount of resources, the solution of the allocation problem indicates which resources are to be used, taking into account the cost associated to its use. Correspondingly, if the amount of jobs is greater than the available resources, the solution indicates which jobs are not to be executed, again taking into account the cost related to it.

If each task needs exactly one resource and each resource can be used on only one task, the resulting problem is referred to as an *assignment problem*. Otherwise, if the resources are divisible, and if both tasks and resources are expressed with the same scale, the problem is referred to as *transportation* or *distribution*. If the scale of jobs and resources is different it is a general *allocation problem*.

In the *allocation problem* exists also another classification, which depends on the number of tasks and resources to be allocated. In particular, if the number of tasks to be allocated is the same as the number of resources available the problem is called *balanced assignment*. Otherwise, if the number of tasks and the number of resources are different, independently of which is greater and which is lower, the problem is called *unbalanced assignment*.

The spectrum of applications for allocation problems is very ample and heterogeneous. For example we can consider the case in which we own a carrier delivery service and our resources are the means of transport that we own. Our task is to deliver the goods from our customers to the destinations that they provide to us, while our objective is to deliver as many goods as

possible minimizing both the time and the cost for delivering those goods to the destinations. Another example can be a trade company which has to allocate investments in the market. In this case the resources are the money to invest and the jobs are the different markets in which to invest the money. The objective is to maximize the return of those investments with respect to the rate of interest and the time it takes to obtain them. Otherwise, we can consider a manufacturing factory with several product chains, our resources, and the different types of products that can be realized by those product chains, our tasks. The objective is to maximize the return from the selling of those products with respect to the cost and time of production, where cost and time vary depending on which product chain produces the item.

Allocation problem fits very well also in many scenarios of computing systems. For instance exists a specific case of allocation problems, which is called *load balancing*: it is used to improve the distribution of the workload, our tasks, across multiple computing resources, our resources. In *load balancing* the objective is to maximize throughput, to minimize response time and to avoid the overload of a single resource. Another example in computing systems is *scheduling*, which is the method used to assign tasks to the resources of a computer. Tasks can be a process, a thread or a data flow, while resources can be processors and network links. The objective in *scheduling* algorithms is to keep all computer resources busy, and to allow users to share system resources among them or to achieve a target quality of service.

### 1.1.1 Case study

As we have seen so far, the applications of the allocation problem are numerous and this proves the importance and the relevance of this field of study. Therefore, we have decided to study the *allocation problem*, in particular what we want to try to solve is the following problem: we are a data center and we have a set of internal resources, such as server machines, then we have a flow of tasks that comes from our clients, each of these tasks may have different requirements in terms of time of execution and resources

necessary to be executed. Our objective is to assign each task to one of our machines and minimize a function that takes as arguments the time of execution and the cost for the execution of that task on a specific machine.

The problem described so far represents an unbounded allocation problem, in which the number of tasks is much greater than the number of resources and, in particular, this number is unknown at the beginning of the allocation and to complete the problem all the tasks have to be executed in a distributed and asynchronous environment. This kind of problem can be seen as a simplified version of the *job shop scheduling* problem or *job-shop problem* (JSP), in which each job is executed in parallel on the available resources, or machines, instead of requiring to be executed sequentially on all the machines, or a subset of them, as it is in the complete *job shop scheduling* problem.

JSP problems have been studied a lot in the literature. Firstly, it is widely known that the problem is *NP-hard* [1] [16]. Secondly, plenty have studied algorithms and heuristics for sub optimal solution of the problem, among the others Conway et al. (1967) [7], who wrote the first book on scheduling theory, Błażewicz et al. (1996) [5], who described conventional solution techniques for solving the problem, Glover et al. (1989) [19] and Dell'Amico et al. (1993) [12], who solved the JSP *combinatorial optimization problem* using *tabu search*, and Pezzella et al. (2008) [41], who proposed a solution that uses genetics algorithms.

### 1.1.2 Formal definition

The formal definition of the *assignment problem* is the following: given two sets $\mathcal{J}$ and $\mathcal{R}$, the jobs and the resources sets respectively, both with a cost function $\mathcal{C} : \mathcal{J} \times \mathcal{R} \to \mathbb{R}$, where $\mathbb{R}$ represents the real numbers set, the

[1] In computational complexity theory, NP-hardness *non-deterministic polynomial-time hardness* is the name of a class of problems that are informally "at least as hard as the hardest problems in NP"

objective is to find a bijection function $f : \mathcal{J} \to \mathcal{R}$ such as the cost function

$$f_* = \sum_{j \in \mathcal{J}} \mathcal{C}(j, f(j)) \tag{1.1}$$

is minimized.

### 1.1.3 Classic solutions

A naive solution for the *assignment problem* is to check all the possible allocations, then to calculate the cost of each of these allocations and choose the ones that minimize the equation 1.1. This may be very inefficient since, with n resources and n tasks, there are n! (factorial of n) different assignments.

Since the *assignment problem* is a special case of the *transportation problem*, which in turn is a special case of a *linear program*, it is possible to solve it using the *simplex algorithm* [11]. This algorithm has a complexity in the worst case not polynomial. However, it has been shown [52] that - in typical cases of use of the *simplex algorithm* - it performs polynomially, resulting one of the most used algorithms in linear programming.

Additionally, for the specific *assignment problem*, better algorithms have been found which solve an instance more efficiently than the *simplex algorithm* using the specific structure of the problem. Indeed, the *Hungarian method* [31] has been proved to solve the problem in time strictly polynomial $O(n^4)$, which can be reduced to $O(n^3)$ using the variants proposed by Munkres in 1957 [38].

#### Load Balancing algorithms

As we have described so far, load balancing is used to spread workloads among available resources. Therefore, it is widely used to provide internet services from a pool of servers, which replicate their instances for purposes of reliability and scalability, also known as *server farm*. In this specific case, the aim of the load balancer is to redirect the client request to one of the available servers in the farm, in order to reduce the response time for the

client and to maintain the load - the tasks that have to be executed - as much as possible similar across the resource, the servers.

One of the simplest and most efficient methods for load balancing is *round robin*. All the resources are considered to be the same with the same capabilities, while tasks are assigned, or, better, requests are redirected to one server in a rotating sequential manner. In such a way, all resources receive the same amount assignment. This simple method ensures good allocation of the workloads if the tasks to be executed are all almost the same, with the same requirements in terms of resources. Nevertheless, if the tasks to be executed are different or if the resources have different capabilities and cost, *round robin* results in being no good because, in its simplicity, it does not take into account the current load of the server farms, even the type of tasks that have to be executed.

A different solution to round robin, which takes into account the current server farm load, is *least connection*. This algorithm redirects requests to the server that has the least number of active sessions, in such a way that it can consider the current load and avoid overloading a server. Still, this algorithm does not take into account the type of tasks to be executed, assuming that they are all the same.

Despite that, in real applications - such as balancing the workloads in a cloud data centre - it is infeasible to apply the techniques described so far because all those solutions are centralized and thus they represent a bottleneck for the system. Therefore, many distributed solutions have been proposed, with the aim of balancing the workloads without degrading the overall performances. Among those who studied distributed load balancing solutions, it is worth mentioning the study done by Randles et al. (2010) [43], who investigated and analyzed three possible distributed solutions. Firstly, they studied the *honeybee self forager allocation* proposed by Nakrani et al. (2004) [40], which takes its name from the analogy with a colony of honeybees foraging and harvesting food. Secondly, they examined the solution presented by Rahmeh et al. (2008) [42], who proposed to randomly sample the *grid*

*network*, which is a parallel and distributed computing network system that has the ability to achieve higher throughput [15]. Thirdly, they investigated the solution proposed by Saffre et al. (2009) [45], who proposed an *active clustering* algorithm able to self aggregate and rewire the network grouping, or clustering, together similar instances.

### Scheduling algorithms

As already pointed out, scheduling algorithms are used to distribute resources among parties which request them. In particular, these algorithms are used in routers for packet traffic handling, as well as in operating systems, disk drivers and most embedded systems. The main purpose of the scheduler, name used to call the entity which performs the scheduling, is to minimize resource starvation, which is caused by a process which is continuously waiting for the resources that it needs to execute its operations.

The simplest algorithm that we can develop for scheduling is FIFO *first in, first out*, also known as FCFS *first come, first served*. This algorithm queues tasks to be executed, or processes, in the order in which they arrive at the ready queue. Thus, when a new task is required, the one that has arrived first is taken. Despite its simplicity, this algorithm does not guarantee fairness and can lead to starvation, because when a task gets control of the resources it could keep it for a long time causing starvation.

An alternative solution for a scheduling algorithm is *fixed priority pre-emptive scheduling*, in which the operating system assigns a fixed priority to every task to be executed, then the scheduler orders these tasks by their rank, by their priority. This solution guarantees no starvation for processes with high priority, but can easily lead to starvation for those processes with lower priority.

Even for scheduling, a possible algorithm is *round robin*. In this case the resources, or the cpus, are assigned to a task for a fixed amount of time and then they are cycled through them. Therefore, if a task completes in its time unit, it is terminated, otherwise it is rescheduled after giving a time unit to

all the other tasks. This method is considered to be fair because in absence of priority it is starvation free. Nevertheless, if a task needs to use a lot of resources for a long period of time it would happen that the task will never end.

Although the solutions proposed are widely used because of their simplicity and efficacy, in complex scenarios, such as audio or video streaming applications, more sophisticated algorithms are needed to ensure the various levels of *Quality of Service* QoS. From the literature many algorithms have been proposed. Among those it is worth pointing out the *weighted fair queueing* WFQ proposed by Demers et al. (1989) [13], who presented a scheduling algorithm for controlling congestion in datagram newtorks. WFQ has been developed to emulate the hypothetical bit-by-bit weighted round robin in which the amount of bits of a flow served in a round is proportional to the weight of the flow. Afterwards, Goyal et al. (1996) proposed an updated version of WFQ, which is called *start-time fair queueing* [20]. This solution aims to achieve fairness, high throughput and efficiency regardless of variation in a server capacity.

## 1.2 Reinforcement Learning

The idea that we learn from the interaction with our environment is probably the primary belief on the manner by which nature learns. Indeed, if we think of an infant trying to stand and walk, or looking around, he has no explicit teacher who tells him how to behave. He is alone with his world, which reacts to his actions. This interaction between the world (or the environment) and the infant creates a knowledge used to perform these tasks in the future.

The computational approach to learning through interaction is known to us as *Reinforcement Learning* (RL). In reinforcement learning we map the interaction between the environment and the learner, or agent. During this interaction, the learner is not told which action to take and, when it

takes one, it observes a change on the environment, or state, and a given numerical reward signal associated to that action, (figure 1.1). Thus, the main objective of the agent is to maximize the whole reward during its execution. In most of the challenging and interesting cases, the actions affect not only the immediate reward, but also the next states and consequently the future reward [53].

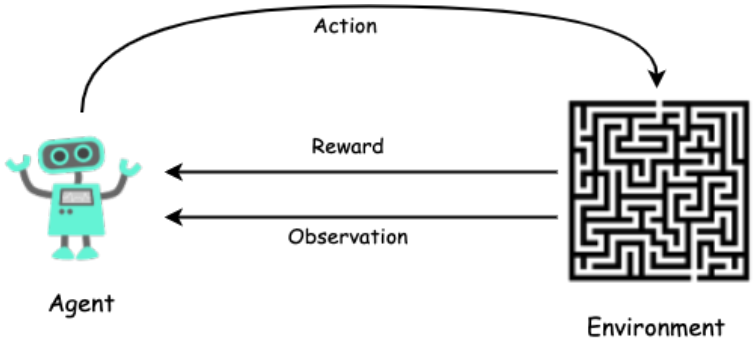



**Figure 1.1:** Reinforcement learning interaction: the agent takes an action and then the environment retruns a reward and an observation of the environment resulted from the effect of the action taken.

One of the most delicate and crucial challenges in reinforcement learning is the trade-off between exploitation and exploration. The former is the case in which the agent tends to follow - and thus to repeat - actions that it has already tried in the past and for which it found to be effective in producing reward. The latter instead is the case in which the agent selects actions that have not been selected before and which can result in better rewards in the following states. The dilemma is that neither exploitation nor exploration can be seeked exclusively without failing at the learning task. The agent must try all the actions, if possible, and progressively favour those that appear to give the best reward. Mathematicians have studied intensively for decades the exploration–exploitation dilemma and yet it remains unresolved. In control engineering the dilemma is kwnown as identification (or estimation) and control (Witten et al. 1976 [61]), while Holland et al. (1975) emphasized the importance of this conflict in genetics algorithms referring to it as the conflict

between the need to exploit and the need for new information [24].

Reinforcement learning belongs to the family of *machine learning*, in particular it is different from *supervised learning* because in supervised learning an external supervisor provides explicit knowledge for the learner through the specification of the proper label for each situation. This kind of learning has been proved to be effective and for this it is widley used, but for the task of learning through interaction it is infeasible, because it is often impractical to retrieve examples of desired behavior that are both representative and correct of all the situations in which the agent has to act.

Moreover, reinforcement learning is also different from *unsupervised learning*, which is typically about finding hidden structures in collections of unlabelled data.

Another important feature of reinforcement learning is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment, in which the agent can sense aspects of this environment, choose actions that affect it and have a specific goal.

For all these reasons, reinforcement learning represents a very promising and challenging field of research that effectively can lead to model the learning process in a way more similar to the one used by animals and humans.

### 1.2.1 History of Reinforcement Learning

Reinforcement learning history comes from two main threads, on the one hand psychology has deeply studied trial and errors in animal learning since the end of the nineteenth century, in which many psychologists argued *trial-and-error* learning as essence of learning [58]. On the other hand, optimization control problem and its solution using value functions and dynamic programming represent the mathematical basis of the modern reinforcement learning. It is worth mentioning the mathematician Richard Bellman, who developed the so called "optimal return function", also known as Bellman equation, which is a necessary condition for optimality associated with the dynamic programming [4]. Moreover, Bellman also introduced the discrete

stochastic version of the optimal control problem known as *Markovian decision processes* (MDPs) [3]. Those two threads came together in the late 1980s to produce the modern field of reinforcement learning.

However, for the reinforcement learning to be well known and applied, we need to wait until the last two decades. The reasons for such a delay in its applications come from two main reasons. Firstly, the increase of the computational power of today's computers, which allows processing tasks that yesterday's computers struggled with. Secondly, the recent success of machine learning and in particular deep learning in fields such as computer vision [33] and natural language processing [6]- just to name a few - has motivated many to study and apply both machine learning and reinforcement learning to a plethora of applications.

### 1.2.2 State of the art

The motivation for studying and applying reinforcement learning derives from the many examples of hugely successful applications which develop reinforcement learning agents capable of solving hard problems, of different nature, with results comparable, or even better, to those achieved by humans.

A common path in all the *Artificial Intelligence* (AI) fields is to start studying and applying those techiniques to problems of relative simplicity, such as games. Reinforcement learning is not an exception; in fact between the end of the twentieth century and the beginning of the twenty-first century Gerald Tesauro (1992, 1994, 1995, 2002) published several works in which he demonstrates how to develop a reinforcement learning agent capable of playing the game of backgammon [54, 56, 57, 55]. Tesauro's programme, *TD-Gammon*, required little backgammon knowledge, yet learned to play extremely well, near the level of the world's strongest grandmasters.

Subsequently, in 2013 and 2015 a Google DeepMind team led by Mnih ([37, 36]) developed a reinforcement learning solution, which, starting from the principles derived by Tesauro's works, was able to reach human-level capabilities, sometimes even more, in the video games of the well known Atari

console. They demonstrated that it was possible to reach such a high level on those games without any knowledge and without any need to define any domain specific features. In fact, they developed an algorithm, called *Deep Q-Network*, by which they were able to master all the 49 video games of the Atari console using the same architecture and the same features.

Following the work done by Mnih et al., in 2016 and 2017, another group of researchers from Google DeepMind led by David Silver made the history of reinforcement learning and consequently of artificial intelligence beating the world champion of the ancient game of Go [49, 50, 48], developing the well famous *AlphaGo* agent. This event was considered so disruptive for two main reasons, firstly the game of Go has a search space significantly larger than other board games such as chess and thus an exhaustive search would result infeasible. Secondly, in any Go programme it is difficult to define an adequate position evaluation function, which would allow to truncate the search at a feasible depth. Therefore, the game of Go, before *AlphaGo*, was considered by everyone a game where artificial interaction would have failed on solving it.

In addition to the results obtained applying reinforcement learning on games, it is worth mentioning, among many, the work done by Li et al. (2010), who developed a personalized web service for recommending news articles from the *Yahoo! Front Page Today* webpage (one of the most visited pages on the Internet at the time of their research). Their goal was to maximize the *click-through rate* (CTR), which is the ratio of the total number of clicks all users make on a webpage to the total number of visits to the page. The novelty of the solution provided by Li et al. comes from the development of an algorithm called *LinUCB* [34], which uses *contextual-bandit* for serving the recommended article to the users.

Therefore, the results obtained by the examples discussed so far have had the effect of motivating many others to study reinforcement learning and thus to increase the number of successful results achieved by this learning framework and solutions.

### 1.2.3 Elements of Reinforcement Learning

We have discussed so far reinforcement learning talking about the interaction between the agent and the environment to learn how to perform a task. Beyond these main elements, we can identify four main subelements of a reinforcement learning system:

1. a *policy*, it defines the learning agent's way of behaving at a given time. Roughly speaking, a policy maps the action taken from two consecutive states of the environment. Generally, policies may be stochastic, specifying a probability for each action.

2. a *reward signal*, it defines the goal of a reinforcement learning problem. The agent, at each time step, receives from the environment a *reward* in response to an action. Thus, the goal of the agent is to maximize the total reward. Therefore, the reward signal defines what actions are good or bad, in the immediate, for the agent.

3. a *value function*, it specifies what is good in the long run. Roughly speaking, a state value is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

4. optionally a *model* of the environment, it is something that mimics the behavior of the environment. Typically, we use a model in *planning* problems and in these cases the reinforcement learning problem is known as *model-based* method. On the contrary, those cases without a model are known as *model-free* methods, which are explicitly trial-and-error learners.

To arrive at a complete definition of the reinforcement learning problem we have to firstly define some intermediate problems that represent a sort of simplified version of the complete reinforcement learning problem. Thus, in the following sections we will define the *multi-armed bandit* problem, then we will define the *contextual bandit* problem, to finally define the *markov*

*decision process*, which gives us the complete definition of the reinforcement learning problem.

## 1.3 Multi-armed Bandit

*Multi-armed bandit* allows us to study the evaluative aspect of reinforcement learning in a simplified setting, one that does not involve learning how to act in more than one situation. In particular, multi-armed bandit problems were introduced in 1952 by Robbins [44] and they are primarily used to model, in automated agents, the trade-off between gaining new knowledge by exploring the agent's environment and exploiting its current, reliable knowledge.

### 1.3.1 A *k*-armed Bandit Problem

Multi-armed bandit is considered to represent the following learning problem. You are faced repeatedly with the decision of selecting among $k$ different options, or actions. After each selection a numerical reward is received, chosen from a stationary probability distribution that depends on the action selected. Your objective is to maximize the expected total reward over a period of time, or *time steps*.

This is the original form of the k-armed bandit problem, so called by analogy to a slot machine, or "one-armed bandit", except that it has k arms, or levers, instead of one. Each action choice represents a play of one of the slot machine levers, and the reward is the payoff for hitting the jackpot. When repeating the arm selection you are trying to maximize your winnings by concentrating your actions on the best levers.

In the $k$-armed problem so far described, each of the $k$ actions has an expected or mean reward given if the action is selected; let us call this the *value* of an action $a$. We then denote as $A_t$ the action selected at time step $t$ and its corresponding reward as $R_t$. Thus, the value of an arbitrary action $a$, denoted as $q_*(a)$, is the expected reward given that $a$ is selected:

$$q_*(a) \doteq \mathbb{E}[R_t|A_t = a] \tag{1.2}$$

We then assume that we do not know the action value with certainty, although you may have estimates. We call the estimated value of action $a$ at time step $t$ as $Q_t(a)$ and we would like that $Q_t(a)$ will be close to $q_*(a)$.

Therefore, the easiest way to solve the $k$-armed bandit problem is to always select the action with the highest estimated value. We call these the *greedy* actions. More generally *greedy methods*.

To use the *greedy methods* we have to estimate the action value. One natural way to estimate it is

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \tag{1.3}$$

where $\mathbb{1}_{predicate}$ denotes the random variable that is 1 if *predicate* is true and 0 if it is not. We call rule (1.3) the *sample-average* method for estimating action values. Therefore, the *greedy* method selects the action $A_t$ as

$$A_t \doteq \underset{a}{\operatorname{argmax}}\, Q_t(a) \tag{1.4}$$

The equation (1.3), already presented, from the computational point of view is inefficient because at each step they require additional memory to store the reward and additional computation to compute the sum. We can instead use an incremental implementation to reduce the amount of space and time necessary to compute the estimated value of an action, as follows

$$Q_{n+1} = Q_n + \frac{1}{n}\left[R_n - Q_n\right] \tag{1.5}$$

where $Q_n$ denotes the estimate of its action value after it has been selected $n$ - 1 times and $R_n$ denotes the reward received after $n$ selections of the action $a$. This implementation requires memory only for $Q_n$ and $n$, and only a small computation for each new reward.

There is an alternative method to the *greedy* methods, in which it behaves greedily most of the time, but every once in a while, with a small probability $\varepsilon$,

instead of selecting randomly from among all the actions with equal probability, independently of the action-value estimates. We call this method *ε-greedy* method and we present the pseudocode for this algorithm in the box below 1, where the function *bandit(a)* is assumed to take an action and return a corresponding reward.

**Algorithm 1:** *ε-greedy for k-armed bandit*

Initialize, for $a = 1$ to $k$:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$
**while** *true* **do**
  **if** *probability is* $1 - \varepsilon$ **then**
    $A \leftarrow \text{argmax}_a\, Q(a)$
  **else**
    $A \leftarrow$ a random action
  **end if**
  $R \leftarrow$ *bandit(A)*
  $N(A) \leftarrow N(A) + 1$
  $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\,[R - Q(A)]$
**end while**

## 1.4 Contextual Bandit

*Contextual Bandit* or *Associative Search* represents a step towards the full reinforcement learning problem. Indeed, so far we have considered, in the Multi-armed bandit, only nonassociative tasks, where actions do not need to be associated with different situations.

In more realistic scenarios we may have that the action which we are going to execute is related to the specific situation in which we are. For example, we may consider the case in which we have several different $k$-armed bandit tasks and that on each step we face one of them randomly. Using the

$k$-armed bandit algorithms defined so far, but causing that the true action values change slowly, this method will not work well.

The case already described is an example of *associative search* task or *contextual-bandit*, because as in the $k$-armed bandit it involves trial-and-error learning to search the best actions, but it also involves the association of these actions with the situation in which they perform the best. This association is usally called context because context is a way of referring to the situation in which the action has been selected.

Therefore, contextual-bandit represents a problem in the middle between the $k$-armed bandit and the reinforcement learning, because it uses the context to better associate the action to the current situation, as we do in the reinforcement learning problem using the state of the environment. However, it does not look forward in the action selection, and it tries to maximize only the current action selection. On the contrary, in the full reinforcement learning problem we try to maximize the total reward of the entire execution.

It is worth mentioning that the definition of contextual bandit was firstly formulated by Langford et al. (2007) [32] and consequently the first algorithm was developed by Li et al. (2010) [34].

## 1.5 Finite Markov Decision Process

Finite *Markov Decision Processes* or MDPs are the last steps for the full formulation of the reinforcement learning problem. They involve evaluative feedback, as in bandits, but also associative aspects, in which they have to choose different actions in different situations. In MDPs, actions influence not only immediate rewards, but also subsequent situations, or states, and as a result the future rewards.

Finite MDPs problem can be formalized by a learner, usually called *agent*, who has to learn and take decisions. This learner interacts with everything outside of it, which is called *environment*. The agent and the environment iteract continually, the former selecting actions and the latter responding to

them and presenting new situations to the agent. Moreover, the environment also gives rise to rewards which the agent seeks to maximize over time through its actions selection (fig. 1.2).

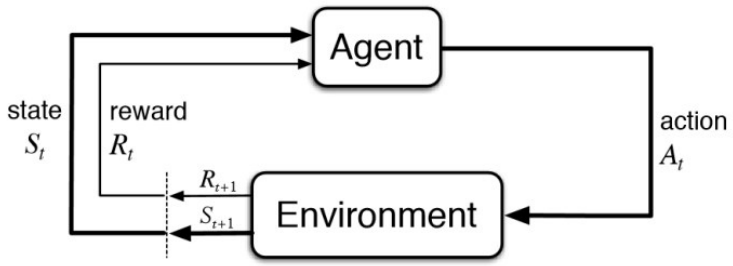



**Figure 1.2:** The agent-environment interaction in a Markov decision process, courtesy of [53].

MDP formalism was used to refer to and describe reinforcement learning firstly by Andreae et al. (1969) [1] and subsequently by Witten et al. (1977) [60], who experimented with a reinforcement learning system which was analyzed using the MDP formalism.

### 1.5.1 Mathematical formulation

More specifically, at each time step $t$, the agent receives some representations of the environment, called *state* $S_t \in \mathcal{S}$, and, on the basis of that, it selects an *action*, $A_t \in \mathcal{A}(s)$. One step later, as a consequence of its actions, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, $S_{t+1}$.

In the *finite* MDP , which we have been describing so far, the sets of states, actions and rewards ($\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) all have a finite number of elements. Therefore, considering the random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occuring at time $t$, given particular values of the preceding state and action:

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1.6}$$

for all $s'$, $s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. Function p, rule 1.6, defines the *dynamics* of the MDP.

As we have discussed so far, the agent's goal is to maximize the cumulative reward it receives in the long run. Therefore, it can be formalized in maximizing the *expected return*, denoted as $G_t$ and defined as some specific function of the reward sequence. In the simplest case the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{1.7}$$

where $T$ is a final time step. This approach is reasonable when there is a natural notion of final step, which can be expressed in terms of subsequences that we call *episodes* or *trials*. Each episode ends in a special state called *terminal state* followed by a reset to a standard starting state. These tasks are called *episodic tasks*. In this type of tasks sometimes we distinguish the set of all nonterminal states, denoted as $\mathcal{S}$, from the set $\mathcal{S}^+$ which contains all the states plus the terminal state.

On the contrary, in many cases the agent-environment interaction does not break naturally into episodes, but goes continually without any limit. This type of cases are called *continuing tasks*, where the rule (1.7) should be reformulated because in such a case the final step would be $T = \infty$ and thus also the expected reward would be infinite. The additional concept that we need is *discounting*. In particular, it chooses $A_t$ to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{1.8}$$

where $\gamma$ is a parameter, $0 \leq \gamma \leq 1$, called *discount rate*.

If $\gamma < 1$, the infinite sum in (1.8) has a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards. As $\gamma$ approaches to 1, the return objective takes future rewards into account more strongly.

Moreover, considering the majority of reinforcement learning algorithms, they involve estimating *value functions*, which are functions of states that

estimate *how good* it is for the agent to be in a given state.

Accordingly, value functions are defined with respect to particular ways of acting, called policies. Formally, a *policy*, is denoted as $\pi$ and it is a mapping from states to probabilities of selecting each possible action. Therefore, we define *state-value function for policy* $\pi$, denoted as $v_\pi(s)$, the expected return when starting in $s$ and following $\pi$ thereafter. For MDPs, we define $v_\pi$ by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \text{ for all} s \in \mathcal{S} \tag{1.9}$$

Similarly, we define *action-value function for policy* $\pi$, denoted as $q_\pi(s, a)$, the expected return starting from $s$, taking action $a$, and thereafter following policy $\pi$:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = a, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{1.10}$$

For any policy $\pi$ and any state $s$, we can consider the following consistency condition

$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right], \text{ for all } s \in \mathcal{S} \tag{1.11}$$

which expresses a relationship between the value of a state and the values of its successor states, which indeed is the *Bellman equation* [4].

Finally, to solve a reinforcement learning task we need to find a policy that achieves a lot of rewards over the long run. A policy $\pi$ is defined to be better or equal to a policy $\pi'$ if its expected return is greater than or equal to that of $\pi'$ for all states. More formally, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v'_\pi$ for all $s \in \mathcal{S}$. The *optimal policy*, denoted as $\pi_*$, is the policy that is always better or equal to all the other policies. We define the *optimal state-value function*, denoted as $v_*$, and defined as

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{1.12}$$

Moreover, optimal policies also share the same *optimal action-value function*, denoted as $q_*$ and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \tag{1.13}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

Because $v_*$ is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values, rule 1.11. Therefore, the *Bellman optimality equation* is defined as

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right] \tag{1.14}$$

while the *Bellman optimality equation* for $q_*$ is

$$q_*(s) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \tag{1.15}$$

Intuitively, the Bellman optimality equation reveals the fact that the value of a state under an optimal policy must be equal to the expected return for the best action from that state.

### 1.5.2 Solutions

Through the *Markov Decision Process* we have defined the complete reinforcement learning problem and now we are going to illustrate *temporal-differences* (TD) learning which provides a solution to MDPs problem. In particular, TD learning is the union of *Monte Carlo* ideas and *Dynamic Programming* ideas. From the former, TD methods learn directly through raw experience without a model of the environment's dynamics, while, from the latter, TD methods update its estimates based in part on other learned estimates, without waiting for a final outcome, in other words they bootstrap.

#### Q-learning: Off-policy TD Control

One of the major breakthroughs in reinforcement learning was the development of an off-policy TD algorithm called *Q-learning*, which was designed by Waltking in 1989 [59].

The definition of this algorithm follows the *generalized policy iteration* (GPI) pattern [53], the main idea of which is that of letting the policy evaluation and the policy improvement processes interact and, in particular, all have identifiable policies and value functions, fig. 1.3, where the policy is always being improved with respect to the value function and the value function is always being driven toward the value function for the policy. Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. Therefore, this implies that the Bellman optimality equations holds (rules: 1.14, 1.15), and thus that the policy and the value function are optimal.

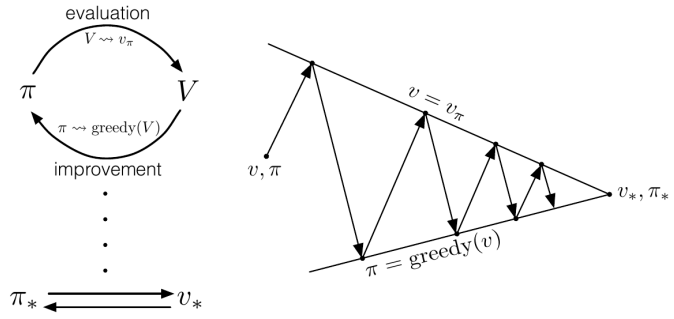



**Figure 1.3:** Generalized policy iteraction: Value and policy functions interact continuously until they are optimal and thus consistent with each other, courtesy of [53].

The first step is to learn an action-value function, through the function $Q$, which directly approximates $q_*$, the optimal action-value function. In particular, the policy determines which state-action pairs are visited and updated, but for a correct convergence it is only required that all pairs continue to be updated. Consequently, the learning is conveyed by the following update of the $Q$ function

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (1.16)$$

which has been shown that $Q$ converges with probability 1 to $q_*$. The complete pseudocode of the Q-learning algorithm is shown below 2, where $y_{t+1}$ represents the target of the update from the equation 1.16.

**Algorithm 2:** *Q-learning: off-policy TD control*

algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrary except that $Q(terminal, \cdot) = 0$

**foreach** *episode* **do**

  Initialize $S$

  **foreach** *step of episode until $S$ is terminal* **do**

    Choose $A$ from $S$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)

    Take action $A$, observe $R$, $S'$

    $y_{t+1} \leftarrow R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$

    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ y_{t+1} - Q(S_t, A_t) \right]$

    $S \leftarrow S'$

  **end foreach**

**end foreach**

### Value-function Approximation

TD methods described so far have been demonstrated to converge under the finite *Markov Decision Processes* assumption. They rely on representing and storing value-functions and/or policies in an exhaustive representation by lookup table. Unfortunately, using such a table requires the state set to be finite and small enough, but in many real-world cases the dimension of the state set is really large, or sometimes it could be even unbounded, to allow the adoption of classic TD methods, such as *Q-learning*. Consequently, *value-function approximation* methods have been developed.

The main idea in *value-function approximation* is that of using a parameterized function with weight vector $w \in \mathbb{R}^d$, instead of a lookup table, for representing the value-functions. Thus, we have $\hat{v}(s, w) \approx v_\pi(s)$ which approx-

imates the value of the state $s$ given a weight vector $w$. The approximator, often called *function approximation*, takes examples from a desidered function, such as a value-function, and attempts to generalize from them to build an approximation of the whole function. This function can be both linear and nonlinear and typically we can use all the function approximators that we use in *supervised learning* for machine learning, artificial neural networks, pattern recognition, and statistical curve fitting.

Notwithstanding, some function approximators have been demonstrated to be more effective than others. Nonlinear approximators through *Artificial Neural Networks* (ANNs) represent one of those with the highest success. In particular, a group of researchers at Google DeepMind developed an impressive demonstration that a deep multi-layer ANN can automate the feature design process and thus approximate the value-function for achieving exceptional results (Mnih et al., 2013, 2015 [37, 36]).

## 1.6 Summary

In this chapter, we have firstly presented what an *allocation problem* is and which are the main variants, illustrating how this problem can be seen from a high-level point of view, such as the case of a carrier delivery service described so far, but also showing how this problem is crucial in computing as it is in the case of *load balancing* and *scheduling*.

Secondly, we have illustrated the first informal formulation of the problem which we are going to study and to solve in the following chapters, which indeed is a simplified variant of the well known *Job Shop Scheduling* problem and for which we are going to give a more detailed and formal description in the third chapter.

Finally, in this chapter we have presented an overview on *Reinforcement Learning*, starting from its definition - both formal and informal - and then introducing some examples of great applications of reinforcement learning, which therefore represent the state of the art in this field. Subsequently,

we have walked through the steps to define the full reinforcement learning problem, from *Multi-armed Bandit*, through *Contextual Bandit* and finally *Markov Decision Processes*. These concepts about RL form the basis for the algorithms which will be used to solve our problem.

The motivations for choosing and studying this field are manifold. Firstly, as we have demonstrated in this chapter, allocation problems are fundamental combinatorial optimization problems which can be applied to many different scenarios; therefore, researching and finding new solutions for those kinds of tasks have extreme relevance for the research society, but also for many companies which have in their business some kind of allocation problems. Secondly, the problem, as it has been formulated, represents a reinforcement learning instance, in which the agent selects actions that are executed by distributed and asynchronous workers, because the main goal remains that of being as fast and efficient as possible and, therefore, the agent observes the rewards resulting in its actions lagging behind the moment in which it took those actions. These particular reinforcement learning instances, sometimes referred as *reinforcement learning with delayed reward*, are a kind of RL problems highly promising because they have not been studied much yet and thus there are many opportunities to contribute and to discover new novel solutions. Thirdly, the field of low level computing has not been associated frequently to reinforcement learning, mainly because systems in those situations require prompt replies, such as in the case of scheduling algorithms; therefore applying classic reinforcement learning represents a bottleneck that can not be allowed. Thus, following the novel work done by Jay et al. (2019), who proposes a congestion control solution which uses reinforcement learning with delayed action [25], and then following the novel approach shown by Sivakumar et al. (2019), who extended the solution proposed by Jay developing a framework for congestion control with delayed actions [51], we want to introduce a new example of application which relies on such reinforcement learning with delayed reward. Moreover, to the best of our knowledge, nobody is applying reinforcement learning on the kind of

allocation problem that we have presented.

In the following chapters we are going to firstly illustrate, in the second chapter, those works which represent the state of the art in both allocation problems and reinforcement learning and that are the basis for the work proposed. Subsequently, in the third chapter, we will present our novel solution for solving distributed and asynchronous task allocation problems using many reinforcement learning techiniques, emphasizing advantages and disadvantages of those proposals. Finally, in the fourth and last chapter, we will present the implementation details of the solutions proposed, showing the noteworthy aspects that have been developed to realize it and we will also exhibit the results that those solutions have achieved on allocating tasks, comparing their outcomes with those of some baselines algorithms.

# Chapter 2

# State of the art

In this second chapter, we illustrate some of those works realized in task allocation and reinforcement learning which represent the state of the art in those fields and the basis, the starting point, for the work that we are trying to solve with this project.

Moreover, we are intent to describe the technique and the algorithms, used in those projects, that have been used to obtain the best results in such fields. Without neglecting any aspects which have not been covered by those solutions, which indeed are relevant to obtain the results that we expect from the study we are proposing with this work.

In particular, our aim here is to firstly introduce those solutions which constitute, in general terms, the state of the art in task allocation problems. Secondly, we describe the works that are considered relevant in the reinforcement learning community. Finally, we present some works which try in some way to use the state of the art in reinforcement learning to solve some kinds of allocation problems.

## 2.1 Assignment problem

*Assignment problems*, as described so far in the first chapter, are a fundamental combinatorial optimization problem which requires to assign tasks

to the available resources in order to minimize total costs or maximize the total return. Moreover, with the objective of better describing the allocation problem which we are willing to solve, we have also presented the *Job Shop Scheduling* (JSP) problem, which is a specific combinatorial problem, where to complete a task it is required to sequentially execute it on several of the available resources with the aim of minimizing the *makespan*, which represents the distance in time that elapses from the start of a work to its end.

Therefore, combinatorial problems have been studied since the late 1950s because of their relevance and importance in many fields, such as logistic management, supply chain optimization and the travelling salesman problem, just to name a few. Regarding assignment problems, one of the principal solutions for solving any instances of the general *balanced* and *unbalanced* assignment problem, among the others, is the *Hungarian method* proposed by Kuhn in 1955 [31]. For job shop scheduling problems in the years are many the solutions that have been proposed by researchers, among those it is worth mentioning the metaheuristic search method known as *Tabu search*, which has been firstly created and formalized by Glover et al. in 1989 [19] and secondly was applied to solve JSP by Dell'Amico et al. in 1993 [12]. Regarding job shop scheduling instances there have been attempts to resolve them using genetic algorithms as in the work proposed by Pezzella et al. in 2008 [41].

### Hungarian method for assignment problem

The *Hungarian method* proposed by Kuhn (1955) [31] takes its name because the author based his intuition on the works done more than 15 years before by two Hungarian mathematicians: D. König [27] and E. Egerváy [14]. The former gives a theoretical laid basis of the algorithm in the case in which the cost, or the return, of a resource associated to a job is indicated by 1 or 0, indicating if the resource is applicable or not to the job. The latter shows and proves how it is possible to reduce the general case to the previous laid case.

The Hungarian method in his original formulation solves the assignment problem with polynomial complexity $O(n^4)$, which is significantly better

to the exhaustive solution where each possible assignment is generated to find the maximum, or the minimum, considering that the number of possible assignments is $n!$. Subsequently to Kuhn, James Munkres in 1957 proposed an improvement of the Hungarian method, which allows to solve the assignment problem with polynomial complexity $O(n^3)$ [38]. Thus, the algorithm is also known as the *Kuhn–Munkres algorithm* or *Munkres assignment algorithm*.

The algorithm proposed, in the case of $n$ resources and $n$ jobs, thus *balanced* instance, can be described by the following procedural steps:

1. create a matrix $M$ $n \times n$, where each row represents one available resource, each column one job that has to be executed and each value of the matrix is the cost of the return corresponding to the assignment $r(i, j)$, with $i$ and $j = 1, ..., n$

2. for each row $i$ of the matrix, find the smallest element and subtract it from every element in its row

3. for each coloumn $j$ of the matrix, find the smallest element and subtract it from every element in its column

4. cover all zeros in the matrix using minimum number of horizontal and vertical lines

5. if the number of lines covering all zeros is equal to $n$, then the algorithms terminate and the optimal assignment corresponds to the zeros found. Otherwise go to the following step

6. determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Finally repeat step 4.

When we have an *unbalanced* instance of assignment problem, whereby we have $n$ resources, $m$ jobs and $n < m$ (or vice versa), we can always reduce the problem to the previous case, adding $d = m - n$ dummies variables, thus columns or rows, to the matrix $M$ with high values if we are minimizing the cost or low values if we are maximizing the return.

### Tabu search for JSP

An instance of *Job Shop Scheduling* (JSP) problem is formalized as follows. A set $M$ of $m$ machines and a set $J$ of $n$ jobs are given. Each job $i$ consists of a chain of $m_i$ operations and each of these operations has to be processed on a machine $\mu_i$ for $d_i$ consecutive time instants. The problem is to assign operations to machines in a way as to respect the chain of operations required by each jobs, to perform in each machine at most one operation for time instant and to minimize the *makespan*, the completion time necessary to execute all $n$ jobs. JSP problems are well known to be *NP-hard* problems [16] and one of the most used method to find a solution is *tabu search* [12].

*Tabu Search* (TS) is a metaheuristic strategy for solving combinatorial optimization problems. It has the ability to make use of many other methods which are used to overcome the limitations of local optimality, therefore it is considered an adaptive procedure. The origin of the algorithm comes from the combinatorial procedures applied to nonlinear covering problems by Fred Glover (1977) [17], who also wrote in 1989 and 1990 two papers [19, 18]: in the former he describes the fundamental principles of tabu search, while in the latter he examines more advanced considerations.

Tabu search has been applied to a diverse collection of problems ranging from computer channel balancing, scheduling, integrated circuit design and job shop scheduling, just to name a few. The name *tabu* comes from the Tongan word to indicate something that cannot be touched because it is sacred.

TS procedure tries to find a solution to a combinatorial optimization problem through a local search in the neighbourhood of an initial feasible solution (e.g. a random solution), then the search moves from one solution to another, choosing the best not forbidden, or tabu, element in the neighbourhood. The aim of forbidding some solutions is to prevent cycling and to guide the search toward unexplored regions. Therefore, during the execution of the search a *tabu list* is stored with the forbidden solutions. A solution $s'$ is to be considered forbidden if the current solution $s$ can be transformed ino

$s'$ by applying one of the moves in the tabu list, while a move is considered admissible if it satisfies an *aspiration criterion* which is associated to each move.

#### Genetic algorithm for JSP

A *genetic algorithm* (GA) is a higher-level procedure designed to find a sufficiently good solution to an optimization problem, it is inspired by the process of natural selection and belongs to the class of *evolutionary algorithms* (EA). GA algorithms were firstly introduced by John Holland [24], who based his theory on the concepts of Darwin's theory of evolution.

Genetic algorithms fit the spectrum of JSP because in GA we start from an initial population, an initial solution as it is in tabu search, then applying genetic operators offsprings are produced, which correspond to exploring the neighbourhood. Then, at each generation of new offsprings, every new individual, referred to as *chromosome*, corresponds to a solution, thus to a schedule for the JSP problem.

The strength of genetic algorithms with respect to other local search methods is due to the fact that in GA it is possible to apply together more strategies on the generation of the offsprings, thus at each algorithm step a bigger portion of solution space is explored resulting in an easier convergence to an acceptable solution for the problem. An example of solution for job shop scheduling problems by means of genetic algorithms is proposed by Pezzella et al. (2008) [41].

### 2.1.1 Load Balancing

Nowadays, in cloud computing load balancer represents a key factor for success and for delivering the best service. Indeed, in the research community load balancing is a very hot topic and there are many noteworthy works. Among those, the comparative study done by Randles et al. (2010) represents a valid source for describing and comparing several solutions which nowadays compose the state of the art in load balancing for cloud computing [43]. Thus,

in Randles's paper is provided an exhaustive analysis of Honeybee dynamic server allocation approaches, as well as Grid networks and Active clusting approaches.

### Honeybee dynamic server allocation

One of the solutions analyzed by Randles in his paper is a distributed load balancing technique inspired by the believed behaviour of a colony of honeybees foraging and harvesting food, which was proposed by Nakrani et al. (2004) [40]. The algorithm proposed in Nakrani's work follows the analogy with the honeybees, where Forager bees are in charge of finding suitable sources of foodand when they find it they advertise this exploration to the Honey bees in the hive through a *waggle dance*, which indicates the suitability of the source found. Honey bees are in charge of exploiting those sources of food.

In a load balancing algorithm based on such honeybees behaviour, each server takes a particular bee role with probabilities $p_x$ or $p_y$, where $p_x$ represents the foraging bee that has to explore new resources, while $p_y$ represents those bees that have to exploit the existing resources. In addition, a distributed share space, called *advert board*, is kept and it is used to post server requests that are successfully fulfilled - this board is the equivalent of the *waggle dance* and contains the profit associated with completed requests. Thus, a group of servers are arranged into virtual servers and they are serving a virtual service queue of requests, then a server will randomly pick a virtual server's queue, thus the server would explore with probability $p_x$, otherwise the server checks the advert board and serves the requests.

### Dynamic random sampling for grid networks

In computing, one technique to achieve high throughput is to take advantage of many computing resources which are allocated in a network in a way to collaborate in the environment, composing the so called *Grid Network* [15]. Those systems to be scalable and reliable need to distribute efficiently the

resources accessible on the network.

Therefore, a distributed load balancer for Grid networks which use biased random sampling has been proposed by Rahmeh et al. (2008) [42]. In their proposal, an initial network is constructed with virtual nodes to represent the server's ones. Each of those nodes is mapped with a number of inward edges, which determines the number of available resources in the node. Every time a job is assigned to a node the number of inward edges is decreased, while at each job execution termination this number is increased. Hence, this number of internal edges is used to increase or decrease the probability to assign a job to a node.

### Active clustering

The last distributed load balancer method analyzed by Randles in his paper is the *Active Clustering* algorithm, in which the topology of a network is modified to aggregate similar services together for creating virtual clusters, exploiting the well known principle by which load balancer performs better when nodes are aware of similar instances and can delegate to them [9] some specific jobs.

An algorithm who exploits such a principle was proposed and studied by Saffre et al. (2009) [45]. In particular, the method he proposes follows three simple steps:

1. an *initiator* is randomly selected among the nodes and randomly selects a *matchmaker* node from its current neighbours, with the only condition of being of a different type

2. then, the *matchmaker* creates a link between one of its neighbours that has the same type of the *initiator* node

3. finally, the *matchmaker* removes the link between itself and the *initiator*.

Those steps are then iteratively executed by the nodes of a network with the aim of creating links, clusters among similar nodes.

### 2.1.2 Scheduling

In computing scenarios are plenty the cases in which a scheduling algorithm has to be used to distribute resources among parties which request them. As we have described in chapter one, solutions for implementing a scheduler are several and they have different behaviours, but they also share the main objective of reducing starvation as much as possible trying to be fair among the involved entities who need the shared resources.

Scheduling algorithm is a very hot topic in the research community, as evidence of that a large number of works have been proposed in the literature in such a field. Among those it is worth pointing out the *weighted fair queueing* WFQ proposed by Demers et al. (1989) [13] and improved by Goyal et al. (1996) with the proposal of *start-time fair queueing* [20].

#### Weighted fair queueing

Scheduling algorithms when applied in process and network scheduling are also referred to us as queueing algorithms, because typically there are sets of entities waiting, in a queue, to use a shared resource and when the main objective in those schedulers is fairness we call this family of algorithms as *fair queuing*. The principle of fair queueing is to use one queue for packet flow and to serve them in rotation, in a way that each flow can obtain an equal fraction of the resources [39]. The advantage of fair queueing compared to other common solutions, such as FIFO and priority queueing, consists on not allowing large packets to take more of its fair share of the total capacity.

Moreover, it is sometimes necessary to divide the shared resources not in equal subparts, therefore we use the *weighted fair queueing* designed by Demers et al. In this variant of fair queueing for each packet $i$ a weight $w_i$ is defined, with $i = 1, ..., N$ and $N$ the number of flows. Subsequently, each flow $i$ will receive an average data rate given by the following rule 2.1

$$\frac{w_i}{(w_1 + w_2 + ... + w_N)} R \tag{2.1}$$

where R is the link rate.

This scheduling algorithm is also known as *Packet-by-Packet Generalized Processor Sharing* (PGPS) because it was designed to reproduce the behaviour of the hypothetical bit-by-bit weighted round robin server, where each bit of a packet is sent separatly from the others in a round robin way. Of course a bit-by-bit scheduler is in practice infeasible because packets have to be sent undivided.

### Start-time fair queueing

Nowadays, in scenarios where integrated network services require to support a variety of different applications, such as video streaming, audio streaming, file transferring and many others - each of which with different requirements in terms of Quality of Service (QoS), with respect to bandwidth and packet delay for audio and video, while for file transferring intense throughput is required - a proper scheduler able to adapt itself to the specific service is crucial.

Therefore, Goyal et al. in 1996 propose a variant of weighted fair queueing called *start-time fair queueing* (SFQ) [20], which is able to adapt depending on the service, keeping the scheduling of the resources fair among variation in the server capacity.

In Goyal's algorithm, two tags are associated to each packet, a start and a finish tag. Initially, a server virtual time is defined equal to 0, while during a busy period $t$, this virtual time becomes equal to the start tag of the packet in service at time $t$. At the end of the busy period, the virtual time is set to the highest finish tag belonging to any packets that have been serviced by time $t$. Thus, during the execution of the algorithm, packets are served in increasing order of the start tags.

This advanced scheduling algorithm is capable of providing fairness even in case of variation of the server capacity. Moreover, it is also efficient because the virtual time depends only on the starting tag of the packet in service. Hence, the computational complexity of SFQ is the same of WFQ, which is

$O(\log N)$ per packet, where $N$ is the number of flows at the server.

## 2.2 Reinforcement Learning

As discussed in the first chapter, *Reinforcement Learning* (RL) is a framework for learning from experience which demonstrates to be extremely effective in the learning task. RL discipline is relatively young: late 1980s for the first formulation and the end of the twentieth century for the first remarkable results. But notwithstanding its short history are plenty the works which are relevant in the research community and thus they are worth mentioning.

In the following sections, we are going to present some of those works which represent the state of the art of reinforcement learning. Firstly, we will illustrate the impressive results which have been obtained by RL in games. Secondly, we will describe how the results achieved in games settings can be used for real-world applications. Finally, we will present some examples of reinforcement learning applied to system level applications. Among those outstanding works there are some which also have represented a baseline for the project under analysis.

### 2.2.1 Playing games at human level

In reinforcement learning and in general for *Artificial Intelligence* (AI), games represent a typical first testbed in which to evaluate the efficacy of an AI technique. The reasons for using games as initial testing system are manyfold, typically games are easy to reproduce, easy to emulate and easy to run for many episodes, but they are also hard to be solved, in particular board games such as chess, backgammon and many others. Thus, a game seems a suitable testbed. Moreover, games can be easily generalized to real-world scenarios, hence a result obtained on a game can be reapplied to many other scenarios.

Among those who have applied RL to games it is worth mentioning the series of papers published by Gerald Tesauro at the end the twentieth

century (1992, 1994, 1995, 2002), where he proposed a reinforcement learning agent capable of mastering the game of backgammon. Subsequently, two different research groups at Google DeepMind presented some outstanding reinforcement learning solution applied to games. The former, led by Mnih in 2013 and 2015, developed a novel algorithm which allowed to play at human level a series of video games called Atari. The latter, leb by David Silver, succeeded in developing an agent who beat the world champion of the game of Go in 2016 and 2017.

### TD-Gammon

The reinforcement learning application proposed by Gerald Tesauro in his series of papers (1992 [54], 1994 [56], 1995 [57], 2002 [55]) and called *TD-Gammon*, is to date considered one of the most impressive solutions, in which it was able to learn playing at a level near to the greatest human world players, even requiring little knowledge of the backgammon game.

The learning procedure was a straighforward composition of the *Temporal Difference* TD($\lambda$) algorithm and nonlinear function approximation using an *Artificial Neural Network* (ANN) trained using backpropagation of TD errors.

Backgammon game is well known for its complexity, which makes ineffective to use classic heuristic search methods that have been proved to be so powerful for games such as chess. Indeed, the branching factor [1] of backgammon is about 400.

Despite the complexity of the game, it is always possible to have a complete description of game's state, as it is highly stochastic. Moreover, there is a clear evidence of episodes because the game evolves following a sequence of moves and then one of the players wins while the other loses. Therefore, such a game represents a good match to the capabilities of TD learning algorithms.

As we have discussed so far, because of its complexity, the number of possible states is too large to fit in memory and to allow a tabular solution,

[1] In tree data structures and game theory, the branching factor corresponds to the number of children at each node.

such as *Q-learning*. Hence, the author proposes using a nonlinear form of TD($\lambda$) where the estimated value, $\hat{v}(s, w)$, of any state $s$ is designed to estimate the probability of winning starting from that state $s$. To realize this estimation the reward was defined as zero for all the time steps except for those in which the game is won.

The implementation of the value function required the use of a standard multilayer Artificial Neural Network, where the input layer was represented by 198 units used by Tesauro for representing the state board of the game, while the output layer was a single unit representing the estimate of the value of that position. Thus, the agent selects a move accordingly to the position with the highest estimation.

The training of such application was performed making two agents, as described so far, playing against each others. They started from random weights for the ANN which led to random moves, but after having played more than 300 thousands of games the application succeeded in beating the previous best backgammon computer programme. The application was called TD-Gammon 0.0 because almost zero knowledge of the game was applied. In the following versions of TD-Gammon Tesauro improved the agent adding some specific knowledge of the game, and the results allowed the proposed solution to play at the level of the best human players of backgammon.

### Human level in Atari video games

In reinforcement learning, one of the main challenges is to decide how to store and represent the action value function and/or the policy. As we have discussed for TD-Gammon, in real world application the state can not be mapped on all its space, therefore typically we rely on function approximation. Whether linear or nonlinear, function approximation should map the characteristics of the environment in such a way to convey the proper information to the learning system for skilled performance. Hence, applications for being successful still require human knowledge to carefully handcraft domain specific features which dramatically improve the performance.

A team of researchers at Google DeepMind led by Mnih, firstly in 2013 and then in 2015, developed an application which uses reinforcement learning, but without any human specific knowledge, and succeeded in mastering all the 49 video games of the Atari 2600 console. One of the main reasons why this work is considered impressive is because for the first time also the design of the features was automated by a deep multilayer ANN.

The process of features selections was realized taking advantage of the impressive results obtained using *deep learning*, which at the time of this work has led to breakthroughs in the field of computer vision [30, 47] and speech recognition [10, 21]. In particular, they used a special class of ANN called *Convolutional Neural Network* (CNN), which applies the theory proposed by Zhang Wei et al. (1988) called *shift-invariant* [62], typically for image recognition [33].

Deep CNNs have been used by Mnih et al. for extracting the features necessary for the learning algorithm directly from the frame of the video game. Each raw pixel, after a stage of preprocessing, has been fed into a deep CNN, whose output represented the value function which estimates future rewards.

Moreover, this work proposed also a novel agent for reinforcement learning called *Deep Q-Networks* (DQN), which learns using the following pipeline, described by fig 2.1. The images captured, once preprocessed are fed into the three convolutional layers, which extract the main features of the image, then those features are propagated to two fully connected layers, which estimate the value function, finally the output corresponds to each valid action and its estimated reward. This type of network has been called by the authors *Q-Network*.

In the first of the two papers published by Mnih et al. [37], they used their DQN agent on seven different games of the Atari console, without changing any aspect of the architecture described so far and they outperformed all the previous approaches on six of those games and surpassed the score of a human expert on three of them. Instead, in their second approach [36], they used a modified version of the DQN algorithm which uses two Q-Networks, one

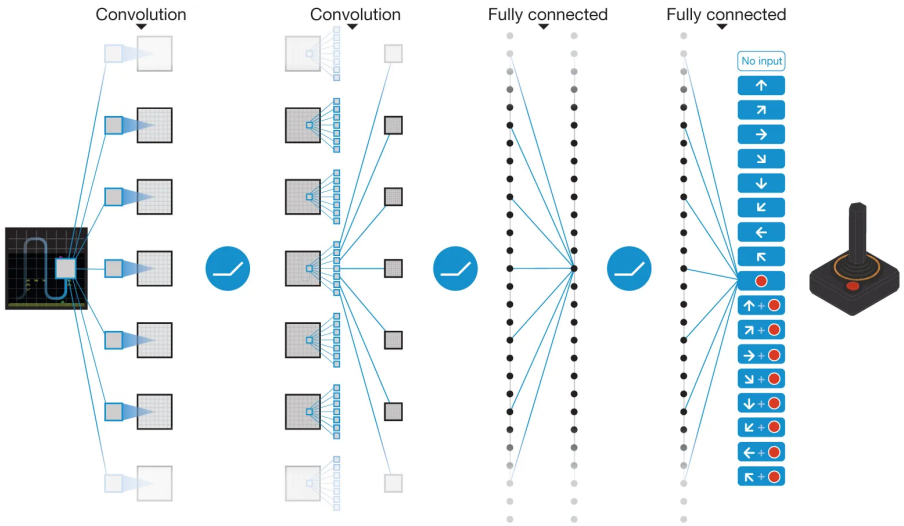



**Figure 2.1:** DQN architecture: the input is the image from the Atari 2600 console, which is processed through several convolutional layers and fully connected layers, finally there is one output for each valid action. Courtesy of [36].

for estimating the value function and one for computing the target used in the backpropagation step of the first network. Using this improved version of DQN they have succeeded in mastering all the 49 games of the Atari console and in the majority of them outperforming human experts, using the same architecture and the same hyperparameters.

### Mastering the game of Go

One of the greatest performance achieved by reinforcement learning was the one developed by another group of research from Google DeepMind led by David Silver, who in 2016 and in 2017 succeeded in beating the world champion of the ancient game of Go, through the so called *AlphaGo* application. Such success derives from two main reasons, as we have already discussed in chapter one. Firstly the game of Go has a search space significantly larger than other board games such as chess and thus an exhaustive search would result infeasible. Secondly, in any Go programme it is difficult to define an adequate position evaluation function, which would allow to truncate the

search at a feasible depth.

The solution proposed by Silver et al. was particularly inspired by both the works described so far. From Tesauro's TD-Gammon approach, *AlphaGo* included reinforcement learning over simulated games of self-play. From the work performed by Mnih et al. with the Atari console, Sivler et al. built its solution on top of the progress of DQN. Another main feature of *AlphaGo* was the combination of the *Monte Carlo Tree Search* (MCTS) with reinforcement learning and *Deep Q-Networks*. MCTS is a search method which uses Monte Carlo rollout to estimate the value of each state in a search tree. It was proposed by Coulom [8] and Szepesvári [29] in 2006. Its main characteristic is to be the more accurate the more the search tree grows, but because of the huge search space of the game of Go, it needs some techniques to reduce the dimension of the tree.

AlphaGo has been trained through a complex pipeline consisting of several stages of machine learning, figure 2.2 left side. Firstly, using a dataset of positions, a fast rollout policy $p_\pi$ and a supervised learning (SL) policy $p_\sigma$ are trained to predict human expert moves. Secondly, a reinforcement learning (RL) policy $p_\rho$ is initialized using the SL policy and then improved through policy gradient learning to maximize the number of winning games against the previous version itself. Finally, the expected outcome, that is winning or losing, is predicted through a value network $v_\theta$.

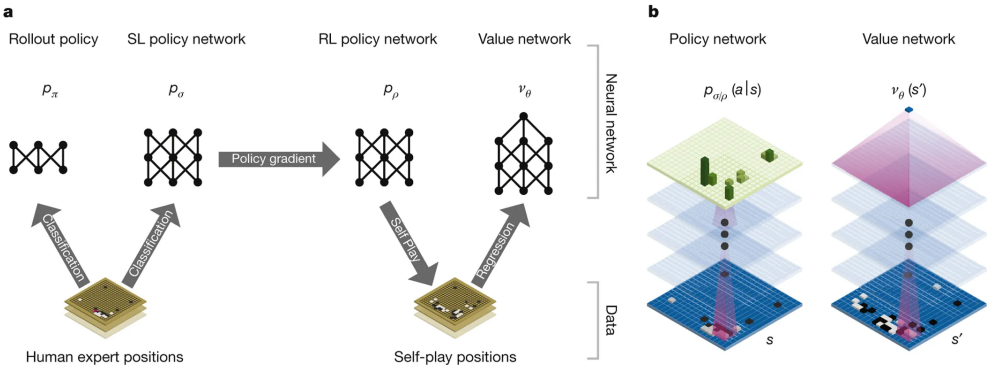



**Figure 2.2:** AlphaGo pipeline on the left and neural network architecture used in AlphaGo on the right. Courtesy of [49].

Figure 2.2 on the right side shows a schematic representation of the neural network architecture used in *AlphaGo*, where the policy network takes as input the representation of the board position $s$; it applies many convolutional layers on it with parameters $\sigma$, for the SL policy network or $\rho$, for the RL policy network, finally the output is a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over all the legal moves $a$. The other network, value network, uses as well many convolutional layers with parameters $\theta$, but the output is the prediction of the expected output in position $s'$, represented by the scalar $v_\theta(s')$.

The architecture described so far uses instensively human knowledge to build impressive playing skills for the game of Go and was presented by Silver et al. in 2016 [49]. This solution succeeded in beating all the previous best Go programmes, and on March 2016 it won four games of the five played against the 18-time world champion Lee Sedol. Subsequently, based on the progress of *AlphaGo*, Silver et al. in 2017 presented a new version of the programme, which used no human knowledge, hence called *AlphaGo Zero* [50], which learns to play exclusively from self-play reinforcement learning and which used MCTS to select moves throughout during learning and not for live play after learning as it was for *AlphaGo*. *AlphaGo Zero* demonstrated to be a better solution than its ancestor; in fact it beats *AlphaGo* for 100 games in a match of 100 games.

Finally, based on the straordinary results obtained by *AlphaGo Zero* without human knowledge, Silver et al. in 2017 presented another version, called *AlphaZero*, which has the same approach as *AlphaGo Zero* and by using self-play reinforcement learning succeeded in mastering diverse board games, such as Go, chess and shogi [48].

### 2.2.2 Personalized web service

We have discussed so far the many impressive results that have been obtained through the application of deep learning and reinforcement learning. Now, instead, we present a different approach, which has been used in a real

world application. This solution is impressive not only for its outcome, but also because, compared to the complex architecture used for Atari and Go, it is relatively simple.

A Yahoo research group, composed by Li, Chu, Langford and Schapire in 2010, proposed a personalized web service for the recommended news articles to the users that every day visit the well known *Yahoo! Front Page Today* webpage and they formalized the subject as a *Contextual Bandit* problem.

The principal approach of the solution proposed was a learning algorithm which sequentially delivers articles to users based on contextual information about the users and the articles, while simultaneously adapts its strategy for article-selection in order to maximize the user feedback, which is computed through the *click-through rate* (CTR), which is the ratio of the total number of clicks all users make on a webpage to the total number of visits to the page.

Typical approaches for recommendation systems rely on the history of the users, which provides information about what they did in the past and so it gives the possibility to define a common behaviour among them. In addition, the items to recommend are usually similar to those previously taken by the user, as it is in an e-commerce for example. Therefore, in such a case it is common to use a technique called *Collaborative filtering*, which recognizes similarities among the users, based on their history, and provides a good recommendation solution [46].

However, in an application such the *Yahoo! Front Page Today* webpage, users are often new to the application, without any history; this is known as *cold-start*. Moreover, the articles to serve are different every day. Therefore, common recommendation solutions are not suitable and this is why Li et al. proposed their solution which also included a contextual bandit algorithm called *LinUCB* [34].

LinUCB algorithm is based on the previous work done by Langford et al. (2007) [32] and can be formalized by the following steps:

1. the algorithm observes a set $A_t$ of actions, the articles to recommend,

and the current user $u_t$. Additionally, a feature vector $x_{t,a}$ for $a \in A_t$ is given, which summarizes information of both the user $u_t$ and the action $a$ and it is referred to as the *context*;

2. based on previous trials observed payoffs, the algorithm chooses an action $a \in A_t$ and receives a payoff $r_{t,a_t}$, the reward, which depends on the expectation on both user $u_t$ and action $a_t$;

3. the algorithm improves its arm-selection strategy with the new observation, composed of the context $x_{t,a}$, the action taken $a_t$ and the payoff $r_{t,a_t}$.

These steps are then repeated for each time step $t$, in order to maximize the total expected payoffs.

In the paper of Li et al., the payoff is computed to be 1 when a presented article is clicked, otherwise 0. This payoff represents the *click-through rate* (CTR) and the objective of the algorithms is to choose the article with the highest CTR, which indeed is equivalent to maximizing the expected number of clicks from users.

### 2.2.3 System level application of reinforcement learning

Another field in which reinforcement learning has been recently proposed is system level application, such as *congestion control*. This specific field involves the necessity to dynamically adapt the transmission rates of different traffic sources to utilize the network resources efficiently and to provide a good user experience. Hence, congestion control has a huge impact on user experience for video streaming, voice-over-IP as well as augmented reality, iternet of things, edge computing and many others.

In a congestion control scenario, each node in the network is composed of a traffic sender and a traffic receiver. The former sends packets to another node receiver, while the latter sends special packets called *acknowledgements*

(ACKs) to senders for notifing to have received a packet intended for its node. Then the goal of protocol is to dynamically regulate the rate of data sent to each node to maximize the total throughput and minimize queuing delay and packet loss.

Typically, network strategies for congestion control rely on hand-crafted heuristic that are reactive rather than predictive, but recently, favoured by the impressive results of reinforcement learning, someone has started to apply those techniques to the problem of congestion control.

One work on congestion control with RL was published by Jay et al. in 2019 [25], who proposed a congestion control with reinforcement learning which takes advantage of deep network policies to capture patterns in data traffic and network conditions. Then using the expectation of the policy it adapts the network rates in order to avoid congestions. In their RL formulation, actions are translated to actual changes in the sending rates, while states are represented by a set of features that are measured in a time window of length $d$ and each state is a statistics vector which contains information about the network in the last $d$ time steps, such as the the ratio between the number of packet receivers and those lost and many others. Finally, the reward is a linear function which gives different weights to three different measures, throughput, latency and packet loss, and depending on the value of those weights the congestion control algorithm adapts to satisfy different requirements specific to the application in which it is used.

Another congestion control protocol which uses reinforcement learning was proposed by a group of research of Facebook, led by Sivakumar in 2019 [51]. In their work they proposed an improved version of the congestion control proposed by Jay et al., in which they used an asynchronous reinforcement learning training. The key point is that, in scenarios such as congestion control, promptness is crucial, therefore the system can not be waiting for an agent which is performing its observation and then take an action; instead the system should continue to operate and when the agent has some valid observations it can change the behaviour. Thus, in their paper Sivakumar et

al. introduced a framework called MVFST-RL, which creates an environment for reinforcement learning with delayed rewards for congestion control and they evaluated this system against typical congestion protocol and the results obtained demonstrated that RL is a promising direction for improving such real-world systems.

## 2.3 Summary

In this chapter we have illustrated some of the most impressive applications of reinforcement learning, showing how those solutions have been applied and which are the main design principles they have used.

More precisely, we have described the evolution of the strategies used by those solutions. We have emphasized which are the crucial aspects of a successful reinforcement learning application, such as features selection for representing the state, reward functions for mapping different cases of use and delayed reward for asynchronous reinforcement learning frameworks. Moreover, we have outlined the main training techniques used in these solutions, such as self-playing for *TD-Gammon* and *AlphaGO*. Finally, we have described the architecture that they have used, such as *deep Artificial Neural Networks* (deep ANNs), *Convolutional Neural Networs* (CNNs) and *Deep Q-Networks* (DQN).

In the following chapters, we are going to use and apply many of the ideas examined in the previous chapters, but we will also try to fill the gap between classic reinforcement learning and reinforcement learning with delayed reward in distributed environments. Thus, we are going to present an environment for this particular case of RL and simultaneously we will propose some implementations of agents, which are capable of solving task allocation problems with distributed and asynchronous workers, on top of this environment. We will also display the outcome obtained by comparing those solutions against some common baseline algorithms.

# Chapter 3

# Service Broker

In this chapter, we present a novel system *Service Broker*, which is capable of solving a modified version of unbalanced assignment problem, where the number of tasks exceed the number of resources, the resources are distributed in the system, thus they execute those tasks asynchronously and all the tasks have to been assigned for completing the allocation problem. This task allocator takes advantages of Reinforcement Learning to find the best policy for spreading those tasks across the distributed environment.

Thus, in this work we present a novel approach that tries to solve the allocation problem using a reinforcement learning agent to discover a policy capable of assigning tasks to the available resources minimizing a reward function which considers time and cost for the execution.

The following sections are logically divided into two different main topics related to the problem under analysis. In the former we will deeply illustrate its mathematical formulation, its variants and its formulation in terms of reinforcement learning problem. In the latter we will present the main aspects of the solution that we have developed for such a problem, considering the type of reinforcement learning algorithms used and the architecture created for *Service Broker*.

Finally, we will leave the presentation of some implementation details and of the results to the final chapter.

## 3.1 Problem Description

In the first chapters, we have introduced the main characteristics and the main results of *allocation problems* and *reinforcement learning*, moreover we have given the first overview of the problem under analysis.

We may think of this problem as if we were a cloud data center with a set of proprietary machines, each of which has different capabilities in terms of speed, cost, memory and so on. We may also have the possibility of using external machines with an increased cost, for example if the current load exceeds our possible load, instead of waiting for some resources to be freed, we can delegate the execution of the exceeding load to an external data center, of course with a higher cost. Then we may have several different types of tasks, or jobs, which have to be executed on those machines; each task may have a different time of execution depending on the machine where it is executed and it may have different requirements in terms of time of execution. Finally, we may have a logically centralized entity, our *Service Broker*, that receives these tasks and has to allocate them to our resources, called workers. Those workers may be distributed and localized in different areas of the data center and, therefore, they are considered fully asynchronous, decoupled and decentralized. The aim of the *Service Broker* is to minimize a function which takes as parameters the time and the cost of execution of a task in a worker.

This problem represents an *unbalanced assignment problem*, in which the number of tasks is much greater than the number of resources, and, in particular, this number is unknown at the beginning of the allocation. Moreover, as opposed to the classic unbalanced problem, here to complete the problem all the tasks have to be executed in the system. As we have seen in the first chapter, this kind of problem can also be seen as a simplified version of the *job-shop problem* (JSP), in which each job is executed in parallel on the available resources, or machines, instead of requiring to be executed sequentially on all the machines, or a subset of them, as it is in the complete *job shop scheduling* problem.

*Service Broker* is a solution to this problem, which uses *reinforcement*

*learning* (RL) to find the best allocation strategy, or policy, for the jobs in the workers. Therefore, it can be seen as a RL agent which, every time a task has to be executed, uses its knowledge to assign this task to one of the available workers, or to an external one. Thus, its actions are represented by the type of workers present in its resource pool. Later, as effect of the action that it selects, the agent would observe a numerical signal, the reward, which has been computed using a function which considers both the total time of execution and the cost for unit of execution. Based on this reward *Service Broker* will adapt its action-selection criterion to maximize this reward, hence to minimize the time of execution.

The problem that we have described so far does not suit only the specific case of the data center that we mentioned when explaining *Service Broker*. In fact, like all the assignment problems it can be seen under different points of view and usability. Therefore, all the examples that we illustrated in the first chapter match the problem definition that we have specified so far, such as the case of the carrier delivery service or the manufacturing factory.

### 3.1.1 Problem Formulation

The problem described so far, in its base case, can be formulated as follows: we have an initial set of resources $\mathcal{P}$, called *worker pool*, of size $d$. Each worker, $w$, of the pool can simultaneously execute only one task at a time and belongs to a *worker class* $c^w \in \mathcal{C}^w = \{c_1^w, ..., c_n^w\}$. Moreover, each worker class $c^w$ is described by a scalar value $\kappa_{exec} \in \mathbb{R}$, which represents the cost per execution time step.

Furthermore, at each time step $t$ the system receives $b$ tasks, each of which has to be assigned to a worker $w$. Each task $u$ belongs to a *task class* $c^u \in \mathcal{C}^u = \{c_1^u, ..., c_m^u\}$. Each task class $c^u$ and worker class $c^w$ characterise a Gaussian distribution $P_{c^u,c^w} = \mathcal{N}(\mu, \sigma^2)$, which gives the execution time of tasks of class $c^u$ on workers of class $c^w$. In addition, each task class $c^u$ is also characterised by a scalar value $\kappa_{wait} \in \mathbb{R}$, which represents the cost for waiting time step.

The *cost function* $\phi : \mathbb{R} \to \mathbb{R}$, associated to each execution of a task $u_j$ on a worker $w_i$, is given by:

$$\phi(\tau_{wait}, \tau_{exec}) = \kappa_{wait}\tau_{wait} + \kappa_{exec}\tau_{exec} \tag{3.1}$$

where $\tau_{wait} \in \mathbb{R}$ is the total time which task $u_t$ has waited before being assigned, while $\tau_{exec} \sim P_{c^u,c^w}$ is the total time of execution of the task.

The goal is to find the best assignment, $f : c^u \to c^w$, such that the cost function $\phi$ is minimized for each task $u_j$:

$$f_* = \underset{f}{\operatorname{argmin}} \sum_{\tau_{wait}} \sum_{\tau_{exec}} \phi_i(\tau_{wait}, \tau_{exec}) \tag{3.2}$$

The formulation just provided describes the base case of the allocation problem under analysis and it is referred to as *fixed pool* case, because the initial pool $\mathcal{P}$ remains fixed for the whole execution and it can not use an external provider for getting additional resources to complete the tasks quickly at the expense of the cost. Instead, in this case the system sacrifices promptness in order to reduce the cost.

### Extended case

The problem formulated so far can be extended to also consider the possibility of using external resources when all the internal ones are busy, in a way to sacrifice the cost, because of course it would be more expensive, but gaining in promptness.

Therefore, the formulation of the problem, in this case referred to as *expandable pool*, needs to be adapted to also match this possibility. In particular, the cost per execution time step $\kappa_{exec}$ of each worker class $c^w \in \mathcal{C}^w$ is now described by two scalar values $\nu, \mu \in \mathbb{R}$. The former represents the cost per execution time step using a worker in $\mathcal{P}$, while the latter is the cost per execution time step using an external worker (not in $\mathcal{P}$). We also constrain $\mu > \nu$ for all $c_j^w$. Moreover, in this case if a worker is external it is temporally instantiated, but then when it finishes its execution it is released. Indeed,

considering the data center analogy, after having finished using an external provider, we stop paying for its additional cost in accordance with the *pay-per use* pricing model typical of cloud providers.

Finally, in this case cost function $\phi$ (rule 3.1) becomes

$$\phi(\tau_{wait}, \tau_{exec}) = \kappa_{wait}\tau_{wait} + \kappa_{exec}\tau_{exec} \quad \text{where} \quad \kappa_{exec} = \begin{cases} \nu & \text{if } w_k \in \mathcal{P} \\ \mu & \text{otherwise} \end{cases} \tag{3.3}$$

Here in the second term of the sum, the cost per execution time step $\kappa_{exec}$ has value $\mu$ or $\nu$ whether the worker class $c_i^w$ relative to the worker $w_i$ in which the task has been assigned is internal or external to $\mathcal{P}$.

In the subsequent sections, we will give a detailed analysis on how we map this problem in the reinforcement learning setting and on how we approach it, distinguishing between the two variants described so far.

## 3.2 MDP formulation

The aim of this work is to solve the problem described so far using reinforcement learning techniques. Therefore, the first step is to give a formulation which matches the *Markov Decision Process* (MDP) formulation.

Recalling the first chapter, a Markovian system is described by a sequence of discrete time steps $t$ and at each of them the agent observes a representation of the environment, the state $s_t \in \mathcal{S}$, and, on the basis of that, it selects an action $a_t \in \mathcal{A}(s)$. One step later, as a consequence of its actions, the agent receives a numerical *reward*, $r_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, $s_{t+1}$. Finally, in a finite MDP the random variables $s_t$ and $r_t$ have a discrete probability distribution which depends only on the preceeding state and action. Therefore, the *dynamics* of the MDP (rule 1.6) gives an indication of the possible next state and reward, which depends only on the previous state and action.

Typically, MDPs' dynamics is described by a stochastic or deterministic

probability distribution, in the former case accordingly to the previous state and action we can only estimate, based on its probability, which would be the following state, while in the latter case, given the couple previous state and action, we can predict the next state.

In the case under analysis the dynamics is stochastic, because, at each time step $t$, some of the features which describe the next state depend on the current task $u_t$, which in turn depends on the task generation distribution.

Furthermore, for the study of the problem we have defined as $b$, the number of tasks received at each time step $t$ equals to 1, while for the *expandable pool* case we have constrained the external cost $\mu$ to be the double of the internal cost $\nu$. Thus $\mu = 2\nu$ for each worker class $w \in \mathcal{C}^w$.

In the following section we are going to describe in details how we map the components of the problem described so far with the MDP formulation.

### Action space

In the MDP formulation, one of the main components is represented by the *action space* $\mathcal{A}$. It contains the set of action $a$, which at each time step $t$ the agent has to select to perform some changes in the environment. In an *assignment problem*, a change in the environment is given by the assignment of a job to a resource. Thus we can consider as action space $\mathcal{A}$ all the resources of the system and then the agent, when at time $t$ it would select an action $a_t \in \mathcal{A}$ for assigning a task $u_t$, would assign the task to the resource selected.

In our *Service Broker* application, the *action space* is given by the following set:

$$\mathcal{A} = \{\mathcal{C}^w \cup wait\} \tag{3.4}$$

where $\mathcal{C}^w$ is the *worker classes* set and *wait* is the wait action, relatively to the base case with *fixed pool*, while in the case with *expandable pool* the action space $\mathcal{A}$ is

$$\mathcal{A} = \{\mathcal{C}^w \cup wait \cup \mathcal{C}^{w_{external}}\} \tag{3.5}$$

where $\mathcal{C}^{w_{external}} = \{c_1^w, ..., c_n^w\}$ is the set of the *external worker classes*, in which each worker class $c_i^w$ is external to the pool $\mathcal{P}$.

The *wait* action is included in the action space, because we want to allow the *Service Broker*'s agent to have complete control of the system, and, thus, to be able to decide when it is better to wait instead of immediately assigning a task. For example, at time step $t$ the agent may want to assign a task to a particular worker class $c_t^w$, because it is considered the best assignment. But, in the pool there may not be any available workers for that class, because they are all busy. Therefore, instead of choosing another worker class $c_t^{w'}$ which may result in a worse performance, the agent should be able to decide to wait for an available worker of class $c_t^w$. Action *wait* influences the cost function $\phi(\tau_{wait}, \tau_{exec})$ of a task $u_t$ (rule 3.1). Indeed, $\tau_{wait}$ is given by the amount of time steps occurred between the instant $t$ in which a task arrives in the system and the instant $t + \tau_{wait}$ in which the task is effectively assigned to a worker and, thus, it starts its execution.

The *external worker classes* set, for the expandable case, belongs to the action space $\mathcal{A}$, because we want to allow the agent to be fast at the expense of the cost. For example, at time step $t$ there may be the case in which the best assignment for a task $u_t$ would not be available, because all the workers of that class are busy. Therefore, the agent should be able to choose if it is better to wait, to assign to another class or to use the class that it has chosen, going however externally to the pool $\mathcal{P}$.

Typically, *action space* $\mathcal{A}$, in scenarios like board games, such as chess or Go, is composed of all the possible movements allowed in the game. Certainly, based on the current state, so accordingly to the current situation of the environment, the action space should update itself to include only those movements that are allowed for that particular placement on the board.

The necessity of shrinking or expanding the action space depending on the current state $s$ is not exclusive of games, but usually the reinforcement learning real world application requires this skill. *Service Broker* is no exception, because as we have described so far, depending on the current state

$s$ some workers of $\mathcal{P}$ may be busy and, therefore, the action space should dinamically change the set of available actions accordingly to the worker currently available. Hence, the actual *action space* $\mathcal{A}$, for the fixed pool case, at time step $t$, for the current state $s$ is given by:

$$\mathcal{A}(s_t) = \{\mathcal{C}^w \cup wait | \, \forall c_t^w \neq \emptyset\} \tag{3.6}$$

where $c_t^w \neq \emptyset$ represents a worker class $c^w$ for which exists at time $t$ at least one worker $w$ not busy, in this case it may happen that all the workers are busy and, therefore, the agent would have to choose the *wait* action. Considering, instead, the expandable pool case, $\mathcal{A}(s_t)$ is given by:

$$\mathcal{A}(s_t) = \{\mathcal{C}^w \cup wait \cup \mathcal{C}^{w_{external}} | \, \forall c_t^w \neq \emptyset\} \tag{3.7}$$

The initial dimension of the *action space* is $|\mathcal{A}| = n + 1$, recalling that $n$ is the number of worker classes, while in the more complex case with the possibility of using external workers the dimension is $|\mathcal{A}| = 2n + 1$.

### Reward function

Another crucial component, which allows to describe the allocation problem through the *Markov Decision Process*, is the reward function $\mathcal{R} \in \mathbb{R}$, which should return a numerical signal used by the agent to convey the learning. In fact, at time step $t$ a reinforcement learning agent would use the value $\mathcal{R}_t$ of the reward function, received in response to action $a_{t-1}$ and state $s_{t-1}$, as an indicator for having taken a good action, if the reward is high, or a bad action, if the reward is low. Indeed, the main objective of the agent is to maximize the *expected return* $G_t$, (rule 1.8), which is the expected total reward it can obtain following the current policy $\pi$.

Assignment problems have as main objective to minimize the total cost or to maximize the total return, which indeed represents a perfect candidate for defining a reward for MDPs and in general of reinforcement learning. We can consider as reward function the return function of an assignment problem

and then we can try to obtain its maximum total value during the whole execution.

Therefore, for the reward function to be used in *Service Broker*, recalling the cost function associated to the execution, at time $t$, of a task $u_t \in c_t^u$ in the worker $w_t \in c_t^w$, rule 3.1, we consider as reward:

$$\begin{aligned} \mathcal{R}_t &= \frac{1}{\phi(\tau_{wait}, \tau_{exec})} \\ &= \frac{1}{\kappa_{wait}\tau_{wait} + \kappa_{exec}\tau_{exec}} \end{aligned} \tag{3.8}$$

where $\kappa_{wait}$ is the cost associated to the task class $c^u$ relative to the waiting time $\tau_{wait}$ of a task $u$, while $k_{exec}$ is the cost per execution time step associated to the worker class $c^w$ in which $u$ has been assigned, which in the *fixed pool* case has value $\nu$, while in the *expandable pool* case can have value $\nu$ or $\mu$ respectively if the worker is internal or if the worker is external to the pool $\mathcal{P}$.

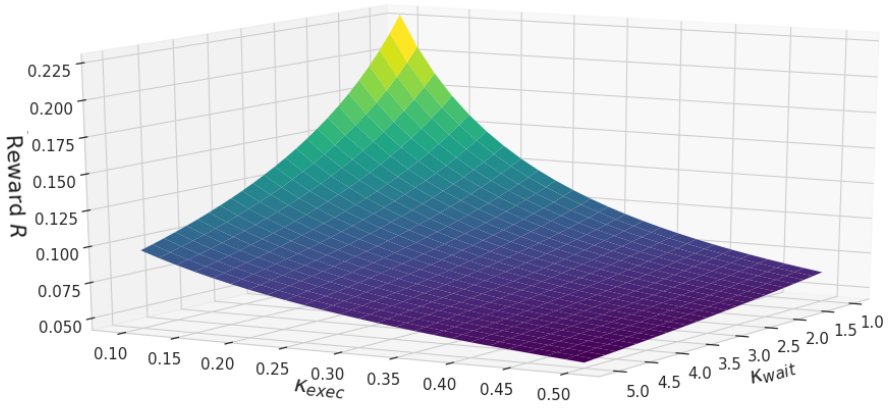
The reward just described differs from the rule 3.1 for the fact that we are computing the inverse of the cost $\phi$; this is due to the fact that in the MDP formulation the agent always tries to maximize the total reward. Therefore, we have inverted the cost function, thus to have that the higher the cost the smaller the reward, while with a lower cost the reward will result higher. With this procedure the agent would try to terminate the tasks as soon as possible, with the smallest cost possible, and to make the task wait as little as possible.

In figures 3.1 and 3.2 we can observe the evolution of the reward function used by *Service Broker*. In particular, in 3.1a and 3.1b the values of the reward function is shown to vary with the values of cost, $\kappa_{wait}$ and $\kappa_{exec}$, or to vary with the values of time, $\tau_{wait}$ and $\tau_{exec}$. As illustrated in the figures, a higher reward is obtained only if both cost and times are low, otherwise the reward drops to lower values.

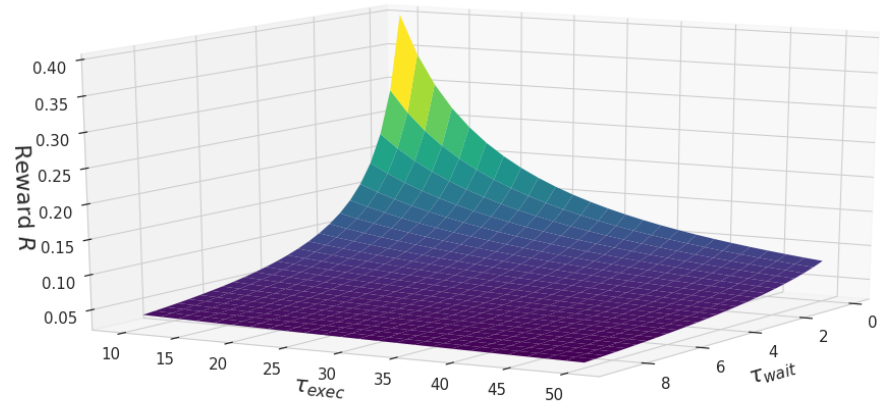
In figure 3.2a, it is shown how the reward function varies with different values of $\tau_{wait}$ and $\kappa_{wait}$, in this case it is evident how the cost per time step in which a task waits to be assigned has a huge impact on the reward obtained.

Therefore, the parameter $\kappa_{wait}$ of each task class $c^u$ can be used to give a priority to a particular task type, because as shown in the figure even if the value of $\tau_{wait}$ is high but the cost $\kappa_{wait}$ is low the final reward obtained is relatively greater.

Finally, in figure 3.2b it is shown how the reward function varies with different values of $\tau_{exec}$ and $\kappa_{exec}$, also in this case the lowest reward is obtained




**(a)** Reward function for different values of $\kappa_{wait}$ and $\kappa_{exec}$.

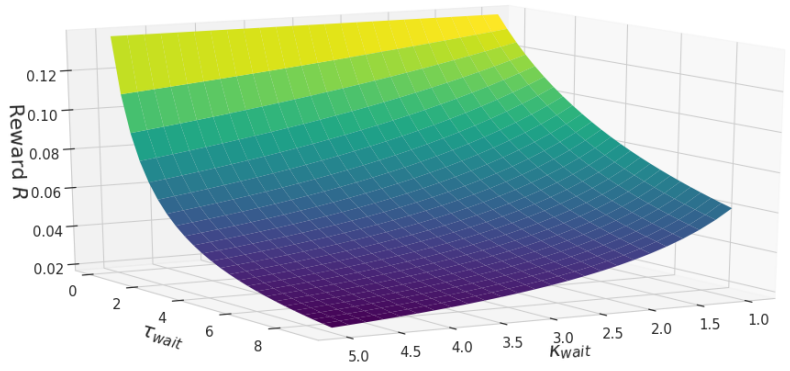



**(b)** Reward function for different values of $\tau_{wait}$ and $\tau_{exec}$.

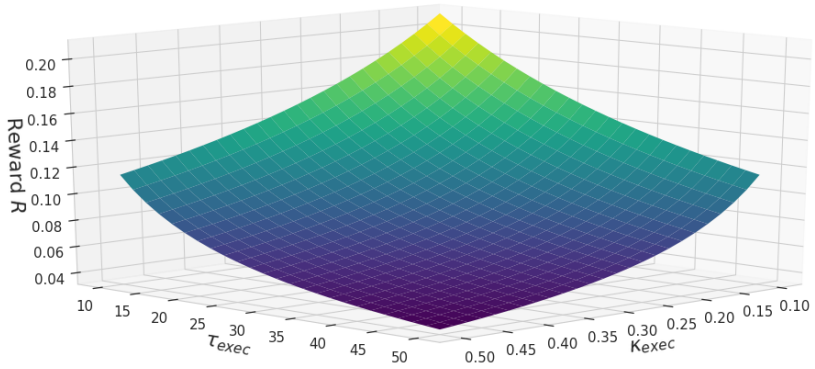
**Figure 3.1:** Reward function with different values of cost or time of execution.

only when both $\tau_{exec}$ and $\kappa_{exec}$ are high, otherwise the reward increases and reaches the maximum when the two parameters have their minimum values.

It is worth pointing out that the range values for $\tau$ and $\kappa$, used to generate figures 3.1 and 3.2, have been chosen accordingly to the minimum and maximum average value obtained during the evaluation execution of the system.




**(a)** Reward function for different values of $\tau_{wait}$ and $\kappa_{wait}$.




**(b)** Reward function for different values of $\tau_{exec}$ and $\kappa_{exec}$.

**Figure 3.2:** Reward function with different values of waiting or execution relative to time and cost.

### MDP with delayed feedback

Another aspect to take into account for *Service Broker* application is the fact that workers are asynchronous and distributed in the data center. Therefore, the typical behaviour of an MDP problem can not be respected, because usually the MDP problem is described by a sequence, or *trajectory*, that begins like this: $\langle s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ... \rangle$, where, at each step $t$ accordingly to the current state $s_t$ and action $a_t$, a reward $r_{t+1}$ and a new state $s_{t+1}$ are received.

However, in the case under analysis, this is not feasible due to the workers' asynchrony, which causes to receive the reward associated with an assignment only after $\tau$ time steps, because in the meanwhile the agent can not stop its execution and it has to continue assigning tasks. Thus, the agent would not observe the reward immediately at the next time step $t + 1$, but, instead, later on when the task would finish its execution, at time $t + \tau$.

Therefore, an extended definition which holds this case, typically referred to as *delayed feedback* or *delayed reward*, is necessary. Joulani et al. (2013) proposed a general framework, supported by a mathematical proof, for MDP with *delayed feedback* [26]. In their formulation of the MDP with *delayed feedback*, Joulani et al. posed the use of a feedback function $h : \mathcal{X} \times \mathcal{A} \times \mathcal{R} \to \mathcal{H}$, where $\mathcal{X}$ is the side information set given by the environment, in our problem the type of the task to assign. Subsequently, at time $t$ the agent would take an action $a_t \in \mathcal{A}$, but it would not observe its outcome at time $t + 1$, rather the reward associated to the action would be scheduled to be revealed after $\tau_t \geq 1$ time instants. Therefore, at each time step $t$, the agent would receive a set $\mathcal{H}_t$, which contains all the outcomes scheduled to be revealed at time $t$ and it would update its action selection strategy accordingly to all those outcomes.

In the case of *Service Broker*, the set of the outcomes $\mathcal{H}_t$ is given by all the tasks which terminate their execution at time $t$ and, because the time of execution of each task class $c_i^u$ depends on the worker class $c_j^w$ in which it is executed, it could be that more than one outcome is observed at the same

time instant.

### 3.2.1 Single state MDP

A *Markov Decision Process* (MDP) can be simplified if we consider a single state for the whole duration of the problem. Thus, the agents in this setting would choose actions according to a general observation of the environment which never changes, the *trajectory* in this case will be described as a sequence of action $a_t$ and observed reward $r_t$. This type of MDP is referred to as *single state MDP*.

MDPs with single state are an alternative way of describing *Multi-armed bandit* and *Contextual Bandit* problems. Recalling the first chapter, we introduced the *Multi-armed bandit* problem as a case in which the learning agent has not to learn how to act in more than one situation.

This setting has a lower learning ability due to the fact that the environment situation is not considered while an action is selected, but, on the other hand, it is a simplified case which allows to easily apply reinforcement learning to complex problems.

Therefore, in *Service Broker* multiple agents have been created and tested on the environment, among those some apply the *Multi-armed bandit* or the *Contextual bandit* formulation.

The former setting does not involve any variation to what we have described so far, because in this formulation the learning agent considers only its history on how it has assigned tasks, without taking into account their class or any observation from the environment and without looking forward to the expected total outcome. Therefore, this type of agents, at each type step $t$, would take an action $a_t \in \mathcal{A}_t$ and would observe, when available at time $t + \tau$, the reward $r_t \in \mathcal{R}$. Thus, it would receive a task $u_t$, it would assign it to a worker $c_t^w$ and when the task completes its execution it would observe its cost $\phi_t$.

The latter setting requires to consider, every time the agent wants to select an action $a_t$, the *context* $x_{t,a}$ for each $a \in \mathcal{A}_t$. The context is a features

vector which summarizes the information of the task class $c_t^u$ and the action $a_t$, the worker class $c_t^w$. Thus, *Contextual bandit* uses a description of the current situation, the context, to create an association among actions and context. This association will be used later to better map similar situations and increases its learning ability.

### Context features

The key point of a *Contextual bandit* agent is the set of features which compose the *context* $x_{t,a}$. To use such agent in *Service Broker* we need to define the features which have been used to summarize the context of our assignment problem.

In particular, at each time step $t$, for each available action $a \in \mathcal{A}_t$, two features compose the context:

- **task class type**: the class $c^u$ of the task $u_t$ which has to be assigned to a worker
- **action**: the worker class $c_i^w$, where the $i$-th index refers to the action $a$ for which the particular context $x_{t,a}$ is built.

Therefore, in the *fixed pool* formulation of the problem, at each time step $t$, at most $n + 1$ contexts are built, one for each action of the *action space* $\mathcal{A}_t$, which contains only the available actions for that time instant. For the *expandable pool* case, at most $2n + 1$ contexts are generated.

In both formulations of the assignment problem, the size of the context is given by the number of task classes and the number of actions of the system, because, in both features, for each element which composes the respective set, a *dummy variable*[1] is used to represent the current task class $c_t^u$ and the current worker class $c_i^w$. Therefore, each context has size $|x_{t,a}| = m + |\mathcal{A}_t|$, while the time complexity for computing one single context is $O(m + |\mathcal{A}|)$,

[1] A *dummy variable* is one which takes as value only 0 or 1 to indicate the absence or presence of some categorical effect.

where $|\mathcal{A}|$ is the whole action space, because in the worst case all the actions are available and because for setting the dummy variables it is necessary to iterate over all the $m$ task classes and the $|\mathcal{A}|$ actions. Thus, at each time step $t$ a context is computed for each action $a \in \mathcal{A}_t$, therefore the time complexity necessary for creating all the contexts used to choose an action $a_t$ at time $t$ is $O(|\mathcal{A}_t|(m + |\mathcal{A}|))$.

### 3.2.2 Multi state MDP

In the previous section we have described a simplified MDP formulation in which there is a single observation of the environment for the whole duration of the learning. Instead, in the full *Markov Decision Process* we have a trajectory which evolves also through new states that describe the changes resulting from the actions taken. Then these new states are fed to the learner for mapping similar behaviours to the outcomes that it receives in response. Finally, it tries to maximize the total expected return, which is given by the total reward which the agent would get following the current policy, the current action selection criterion, from the current time step onwards.

Thus, the state is crucial for any reinforcement learning application because it guides the agent through the learning, hence its representation has to be complete and exhaustive, in a way to include all the relevant aspects of the environment which can be used for properly choosing the actions. On the other hand, because it is continuously queried by the agent during the action selection, it has to be as small as possible to not affect the efficiency.

In order to get the best performances in *Service Broker*, the state representation has been deeply investigated. The result is a contained set of features, represented through a vector, composed of:

- **task class type**: it is the class $c^u$ of the task $u_t$ which has to be assigned to a worker, where each class $c^u \in \mathcal{C}^u$ is represented by a dummy variable, which is set to 1 only if the $u_t \in c^u_i$. This feature has been added to the state because the agent needs to map good and bad actions to similar situations, thus to tasks belonging to the same class.

- **task waiting time**: it is the total time $\tau_{wait}$ since the agent chooses action *wait* for task $u_t$, at time instant $t + \tau_{wait}$. This waiting time is used to compute the reward. The total waiting time $\tau_{wait}$ enormously influences the reward obtained by one execution (recall rule 3.8), therefore it is crucial for the agent to have this information.

- **pool availability**: this feature represents the current image of the internal resources of the system, the *worker pool* $\mathcal{P}$, in particular it encodes the number of workers available of each class $c^w \in \mathcal{C}^w$ at time step $t$. Using this feature the agent maps an action with the environment internal pool status.

- **pool load**: this feature considers the workloads of all the internal workers in a time window of $x$ time steps, thus it represents the number of available workers in the time window, normalized by the total. Using this feature the agent can estimate when it would have some workers available for a certain class $c^w$.

- **task frequency**: this feature provides the frequency of the load of the system, because in the same time window $x$, used by the feature *pool load*, it considers the number of tasks arrived in the system, grouped by their task class $c^u$ and normalized by the total number of tasks received in the time window. Through the *task frequency* the learning agent can forecast the next task it would receive with a certain probability which depends on the distribution of tasks arrivals.

The state $s_t \in \mathcal{S}$, as represented by the features here described, is computed at each time step $t$ by the environment and it is used by the agent for choosing the action $a_t$. The size of this vectorial state is given by the number of task classes $m$ multiplied by 2 for the features *task class type* and *task frequency*, by the size of the *action space* $\mathcal{A}$ multiplied by 2 for the *pool availability* and the *pool load* features and plus one for the *task waiting time* feature. Therefore, $|s_t| = 2m + 1 + 2|\mathcal{A}|$, recalling that $|\mathcal{A}|$ is equal to $n + 1$ in the

base version of the assingment problem, while it is equal to $2n + 1$ in the extended case. The time complexity for computing the state at time step $t$ is given by $O(2n + 2m) \sim O(n + m)$, because for computing all the features it is necessary to iterate twice over all the task classes or the workers classes. It is worth mentioning that in practice the values of the *pool load* and *task frequency* features are updated at each time step $t$ instead of computing them every time, therefore the complexity becomes exactly $O(n + m)$.

It is worth pointing out that the solutions proposed, both *single state* and *multi state* MDPs, do not imply any type of cost in terms of communication or in terms of coordination, because all the costs necessary to perform an assignment are local to the agents and they are related mainly to the cost for computing the context features or the state features where used by the learning technique.

## 3.3 Algorithms considered

With this section we start the description of the solution developed, specifically we describe the type of reinforcement learning agents that we have decided to use in *Service Broker* application, explaining the motivation for their selection and analyzing their main aspects in terms of solution proposed, algorithms used and learning abilities.

### 3.3.1 For the single state MDP

Regarding the MDP with a single state for the whole execution, the agents developed are two. The former implements the simplest learner agent available in reinforcement learning, thus the *Multi-armed bandit* agent, and for this type of agent two different policy strategies have been used (a policy strategy characterizes the way in which actions are selected). The latter implements the other type of learner described in the previous section, the *Contextual bandit* agent; for this type of agent a single algorithm has been developed and tested on *Service Broker* environment.

### $\varepsilon$-greedy

The first type of policy implemented for the *Multi-armed bandit* agent is the *$\varepsilon$-greedy* policy. This straightforward solution has already been presented in the first chapter (algorithm 1), when the bandit problem has been described, and represents a base solution typically implemented for analyzing the tradeoff between *exploration* and *exploitation*, which is one of the most challenging dilemmas of reinforcement learning.

In *$\varepsilon$-greedy* the selection of the action, at each time step $t$, is performed acting as a random agent, which takes a random action among those available, with probability $\varepsilon$, while with probability $1 - \varepsilon$ the selection is done using the agent past experience, which means that it takes the action from which it has received the highest reward since the beginning. This type of behaviour is typically referred to as *greedy*.

Therefore, simply changing the value of $\varepsilon$, it is possible to change the behaviour of the agent. If the probability $\varepsilon$ has value near to 0, the agent would perform most of the times *greedily* and thus it would favour *exploitation* to *exploration*. Otherwise, when the probability $\varepsilon$ approaches values near to 1, the agent would perform most of the times randomly, favouring *exploration*. Thus, choosing a proper value for $\varepsilon$ is crucial to allow the agent to look around for new best actions, but also to continue selecting accordingly to the best already found.

A *Multi-armed bandit* which implements the *$\varepsilon$-greedy* policy is trivial, because it only needs to store the value estimate $Q$ associated to each action $a \in \mathcal{A}$ and at each time step $t$ if the probability is $1 - \varepsilon$ the action $a_t$ selected is the one for which the value estimate $Q_t(a)$ is max, (rule 1.4). After $\tau$ time steps the agent would observe the reward associated to the action $a_t$ and it would update the value estimate for that action accordingly to the rule 1.5.

### UCB

The second policy implemented for the *Multi-armed bandit* agent is a more sophisticated technique, where the criterion for exploring new best actions is

not random, rather it is selected accordingly to their potentiality for actually being optimal. This kind of policy is typically referred to as *Upper Confidence Bound* (UCB) and it was firstly proposed by Auer, Cesa-Bianchi and Fisher (2002).

Therefore, in UCB policy, at each time step $t$ the action $a_t$ is selected accordingly to

$$a_t \doteq \underset{a}{\operatorname{argmax}} \left[ Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{3.9}$$

where the second added is the upper bound associated to the action $a$ and it is composed by the square root of the natural logarithm of $t$ divided by $N_t(a)$, which is the number of time action $a$ has been selected prior to time $t$; then the square root is multiplied by a costant $c > 0$ which controls the degree of exploration.

The denominator of the upper bound (rule 3.9) can be zero if action $a$ has never been selected, therefore in the practice to use this action selection criterion all the actions $a \in \mathcal{A}$ must have been selected at least once.

The idea behind UCB is that the square root term measures the uncertainty or variance in the estimate of $a$ value, while $c$ determines the confidence level. The more we select $a$, the more the uncertainty is presumably reduced. On the other hand, the more the agent selects actions different from $a$, the more $t$ would increase, but $N_t(a)$ would remain fixed and thus the uncertainty estimate of $a$ would increase. This version of UCB was called UCB1 by Auer et al in their work [2] in 2002.

The only difference between UCB and *ε-greedy* is on the action selection, which has already been described, while the update of the value estimate remains the same.

### LinUCB

Considering the *Contextual bandit* agent, in *Service Broker* was implemented the *LinUCB* algorithm presented by Li et al. in 2010, in their work on

personalized news article recommendation for the *Yahoo! Front Page Today* webpage.

The main idea and the main workflow of *LinUCB* for *Contextual bandit* have already been described in the previous chapters. Here we want to highlight some details on how the algorithm chooses actions and improves its selection criterion.

This algorithm is called *LinUCB* because, as it happens for UCB, it tries to compute an upper bound to control the exploration degree in a non random way, contrary to how it would be for *ε-greedy* technique. As we have already described so far, UCB methods choose the action $a$ accordingly to the general criterion $a_t = \text{argmax}_a(\hat{\mu}_{t,a} + c_{t,a})$, where $\hat{\mu}_{t,a}$ is the value estimate of action $a$ at time $t$ and $c_{t,a}$ is the confidence interval (the upper bound).

Therefore, *LinUCB* is a mapping of the UCB concepts on the *Contextual bandit*, indeed at each time step $t$, action $a$ is selected accordingly to

$$a_t \doteq \underset{a \in \mathcal{A}_t}{\text{argmax}} \left[ \hat{\theta}_a^\intercal x_{t,a} + \alpha \sqrt{x_{t,a}^\intercal A_a^{-1} x_{t,a}} \right] \tag{3.10}$$

where $x_{t,a}$ represents the context of the action $a$ at time $t$ with size $d$, $\alpha = 1 + \sqrt{\frac{\ln(\frac{2}{\delta})}{2}}$ is a constant which for any $\delta > 0$ controls the degree of exploration and $\hat{\theta}_a^\intercal = A_a^{-1} b_a$ is a matrix obtained applying ridge regression to the training data $A$ and $b$, with $A$ being an identity matrix of dimension $d$ and $b$ being a zero vector of size $d$.

Subsequently, in response to action $a_t$ the agent receives a reward $r_t$, in *Service Broker* it is received after $\tau$ time steps, and the update of the training variable $A$ and $b$ is performed as follows

$$A_{a_t} \leftarrow A_{a_t} + x_{t,a_t} x_{t,a_t}^\intercal \quad \text{and} \quad b_{a_t} \leftarrow b_{a_t} + r_t x_{t,a_t} \tag{3.11}$$

This action selection criterion and reward update of the training variables compose the *LinUCB* algorithm proposed by Li et al. in [34]; they have also proved that the complexity is linear in the number of arms and at most cubic in the number of features $d$, therefore $O(d^3)$.

### 3.3.2 For the multi state MDP

Regarding the full MDP problem, where at each time instant a new state of the environment is provided, in *Service Broker* application 2 agents have been implemented, both of which use *value-function approximation* through deep *Neural Networks* (NNs) for estimating the value of a state $s$.

The reason for not using a tabular method, such as *Q-learning* presented in the first chapter, is mainly because for a tabular method to learn it is necessary to have a finite set of possible states and because the same state needs to be analyzed by the agent several times. In *Service Broker* application this is not possible, because some of the features used for representing the state of the environment, such the *pool load* and the *task frequency*, are strictly related to the flow of task received and so the agent may see such state for a few number of times which make the learning ineffective.

Therefore, thanks to the advantages demonstrated by Mnih et al at Google Deep Mind in 2013 on using their particular version of *Q-learning* with function approximation, called *Deep Q-network* (DQN) agent, we have decided to apply this type of agent to our assignment problem.

#### DQN

The first DQN agent implemented uses the same structure proposed by Mnih et al. in their papers [37, 36]; the only difference is on the type NN used, because in their case the state was represented by the image of the Atari game console, the features of which are typically approximated using *Convolutional Neural Networks* (CNNs), while, in the case of *Service Broker*, the state is represented as a vector of features, therefore it is approximated through a *fully connected* neural network.

To allow the training of large neural networks without diverting, Mnih et al. proposed two main changes to the basic *Q-learning*.

Firstly, they used the concept of *experience replay*, which was studied for the first time by Lin et al. (1993) [35]. Through the *experience replay* the agent stores experience at each time step in a replay memory that is accessed

to perform the weight updates of the NN. Each entry of this experience buffer is composed by the tuple $(s_t, a_t, r_{t+1}, s_{t+1})$ and every update of the action-value function weights is perfomed using a sample of size $D$, which is randomly drawn from the buffer. The use of *experience replay* improves the training, because if we use consecutive samples the learning is inefficient due to the strong correlation between the samples, thus randomizing the samples breaks these correlations and therefore reduces the variance of the updates.

Secondly, they have modified the basic *Q-learning*, adding the use of a second neural network, called *target network* $\hat{Q}$, aimed at further improving the stability of the method. This second NN is used by the agent for generating the targets used to compute the optimization step of the action-value function network $Q$. Moreover, every $C$ updates of $Q$ the set of weights used by $Q$ is copied into the target network $\hat{Q}$. This modification makes the algorithm more stable compared to the standard *Q-learning*.

In the DQN algorithm proposed by Mnih et al. the agent uses also a $\varepsilon$-*greedy* policy for balancing the trade-off between exploitation and exploration, therefore the agent with probability $\varepsilon$ would perform randomly, while with probability $1 - \varepsilon$ would use its action-value function $Q$ to choose the action. The full algorithm for training *deep Q-networks* is presented in the box below (algorithm 3).

It is worth pointing out that in this version used by *Service Broker* action selection and reward observation are not performed at the following time step, rather after $\tau$ time steps. Consequently, action selection and reward observation must be non blocking for the other allowing the normal execution of the system.

### Double DQN

The second DQN agent implemented is a variant of the previous one which was proposed by Hasselt et al. in 2015 as an improvement of the DQN algorithm proposed by Mnih et al. This variant is called *Double-DQN* because it combines the DQN method with an already known version of *Q-learning*

**Algorithm 3:** *deep Q-learning* in *Service Broker* environment

Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\bar{\theta} = \theta$
**while** *true* **do**
  **if** *probability is* $1 - \varepsilon$ **then**
    $a_t \leftarrow \text{argmax}_a \, Q(s_t, a; \theta)$
  **else**
    $A \leftarrow$ a random action
  **end if**
  Execute action $a_t$ and then observe next state $s_{t+1}$
  After $\tau$ time steps receive reward $r_{t+\tau}$ associated to $a_t$ and $s_t$
  Store transition $(s_t, a_t, r_{t+\tau}, s_{t+1})$ in $D$
  Sample random minibatch of transitions $(s_j, a_j, r_{j+\tau}, s_{j+1})$ from $D$
  $y_j \leftarrow r_{j+\tau} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \bar{\theta})$
  Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to the network parameters $\theta$ Every $C$ steps reset $\hat{Q} = Q$
**end while**

called *Double Q-learning*, which in the tabular case was proposed by Hasselt in 2010 [22] and which is typically used in a very noisy environment because it is able to overcome the noise.

Moreover, the reason for such a noise in *Q-learning* is due to the fact that the optimization step includes a maximization over estimated action values, which tends to prefer overestimated values rather than underestimated ones. The overestimation, if not uniformly distributed on the states, can lead to a negative effect on the learning. Therefore, based on the tabular version of *Double Q-learning*, Hasselt et al. proposed *Double-DQN* and demonstrated that it was able to overcome noisy environments and surprisingly, even in cases without any noise, this method perfomed better than the normal DQN.

Thus, considering that the environment of *Service Broker* has to be considered noisy due to the variability of the task types, their distribution

and their time of execution, trying a solution like *Double-DQN* aiming to overcome this noise seems a natural consequence.

*Double-DQN* differs from DQN only in the computation of the target $y_j$ which is used to calculate the error on the action-value function network $Q$ and, thus, it is used to perform an optimization step over $Q$ weights. The target of the update in *Double-DQN* is given by

$$y_t^{DoubleDQN} = r_{t+\tau} + \gamma Q(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(s_{t+1}, a; \theta_t); \bar{\theta}_t) \tag{3.12}$$

Therefore, the target $y^{DoubleDQN}$ is computed using the value of the target network $\hat{Q}$, which is fed using the action for which the best estimated action-value is obtained from the action-value function network $Q$.

All the other aspects of the DQN method are kept the same by the *Double-DQN*, indeed it uses the experience replay for keeping the variance of the updates as small as possible and it updates the weights of the target network every $C$ time steps copying them from the action-value function network.

## 3.4 The Environment

In this section we focus on a different aspect of *Service Broker* application, the *environment*. So far we have described the mathematical formulation of the assignment problem under analysis and the aspects related to it which maps it into the reinforcement learning setting. However, an RL application, to be able to learn, needs an environment with which to interact during the learning process.

An environment such as the one required for a scenario like a data center is a complex system, composed of many components: some logically centralized and others fully distributed. Moreover, this kind of environment needs a mechanism by which it communicates and collaborates with the reinforcement learning agent.

Therefore, with the objective of creating a system for simulating the data center environment, we have created from scratch a new system capable of

generating tasks, interacting with the agent and assigning those tasks to workers, all maintaining the system completely decoupled and as fast as possible.

### 3.4.1 The architecture

The environment created is a complex system composed of several entities which interact and exhange information to provide the agents with all the knowledge they need about the system.

The architecture of the environment, described in figure 3.3, is composed of four main entities: a *Task Generator*, which at each time step $t$ generates $b$ tasks, a *Task Broker*, which at each time step $t$ takes one of the tasks $u_t$ from the queue and interacting with the agent it assigns the task to a worker $w_i$ in the internal pool $\mathcal{P}$, a *Task* abstraction, which simulates a real task, and a set of $n$ *Workers*, where each worker $w_i \in \mathcal{P}$ waits for receiving tasks to execute.

In the following sections we will describe more in details this entities from a functional point of view, while in the fourth chapter we will also describe some implementation details.

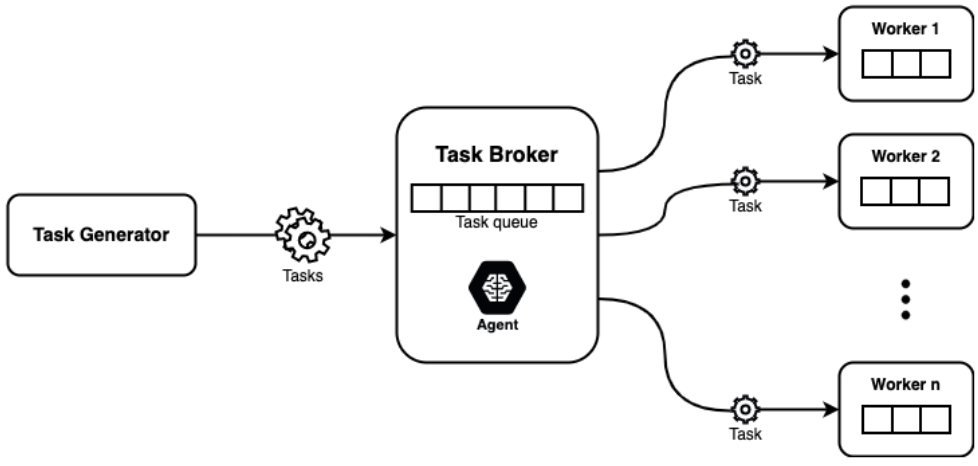



**Figure 3.3:** Environment architecture. Task Generator feeds the task queue of the TaskBroker, which uses the agent for assigning tasks to its $n$ workers.

### 3.4.2 Task

The *Task* entity represents a single unit of execution for the *Service Broker* application. Thus, in the environment created, a task is the abstraction of a real task to execute, where each task belongs to a class $c^u \in \mathcal{C}^u$ and each class $c^u$ is characterised by the cost for waiting time step $\kappa_{wait}$ and by $n$ gaussian distributions $\mathcal{N}(\mu, \sigma^2)$, one for each worker class $c^w \in \mathcal{C}^w$.

Therefore, when at time instant $t$ a task is started - inside one worker $w \in c^w$ - it would execute for $\tau_{exec}$ time steps, where $\tau_{exec}$ is drawn from the gaussian distribution $P_{c^u,c^w}$ which characterises this assignment. Through this behaviour the execution of a real task is simulated, allowing the system to better learn thecharacteristics of each class task for the best assignment on the proper worker.

### 3.4.3 Task Generator

Taking into account the architecture described in figure 3.3, the first entity on the left is the *Task Generator*. This component has the aim of generating tasks for the system, in *Service Broker* application we have considered the *Task Generator* as a single entity, but in a real scenario, such as a data center, this logically centralized component would certainly be a set of sources which provides the tasks that the environment would have to assign.

More precisely, the *Task Generator*, at each time step $t$, would produce $b$ tasks, recalling that for our simulation $b = 1$, and then those $b$ tasks are sent to the task queue of the *Task Broker* for being subsequently assigned to a worker. The class $c^u$ of the tasks generated by the *Task Generator* is given by a certain probability $\rho$ which can be either uniformly or non-uniformly distributed.

### 3.4.4 Task Broker

The main entity in the environment used in the simulation is the *Task Broker*. This logically centralized entity is in charge of two crucial jobs:

firstly it interacts directly with the agent and secondly, based on the agent's decisions, it assigns tasks to the workers.

This entity is internally composed of other two sub entities, *task queue* and *agent*, which allow the *Task Broker* to operate. The former is a standard FIFO queue, which stores tasks from the *Task Generator* and from which tasks are taken for then being assigned. The latter is the reinforcement learning agent, which is logically inside the *Task Broker*, because the agent needs to interact with the TaskBroker every time step and with the purpose of reducing the latency necessary for their interaction.

Therefore, the internal flow of the *Task Broker* at each time instant $t$ is given by the following steps:

1. it dequeues (it takes from the queue) the next task $u_t$ from the *task queue*

2. it updates its representation of the environment, thus it computes the current values for all the features of the state $s_t$ that we have described so far, and it adapts the action space $\mathcal{A}(s_t)$ to contain only the available actions, to contain the worker classes for which at least one worker $w_t \in c^w$ is not busy and the action *wait*

3. if at least one task, started at time $t - \tau_{exec}$, finishes its execution, the environment, in particular the *Task Broker*, will also compute the reward $r_t$ related to the assignment already completed, through the rule 3.8

4. it would feed into the agent the current state $s_t$ and if available all the rewards $R_t$ related to the tasks which terminate their execution in the current time step

5. finally, based on the action chosen by the agent, *Task Broker* would assign one of the available worker $w_t \in \mathcal{P}$ the task $u_t$, where $w_t \in c^w$ and $c^w = a_t$. If we consider the *expandable pool* version of the problem,

the *Task Broker* would instantiate a new worker $w_t$ of class $c^{w_{external}}$ if the agent chooses an action to assign to an external worker.

### 3.4.5 Worker

The last main entity of the environment is the *Worker*, which is an abstraction of a real decoupled and asynchronous server worker in a data center, thus it represents one of the $n$ resources, be it internal or external, of the worker pool $\mathcal{P}$.

The aim of a *Worker* is to check at each time step $t$ its internal queue, implemented as a FIFO queue as the one of the *Task Broker*, when a task $u_t$ has been assigned by the agent through the broker to its queue, it would start the execution of that task and it would set ts internal status to *busy* until $u_t$ terminates, at time step $t + \tau_{exec}$.

When a *Worker* sets its internal status to *busy*, the environment would not be able to assign any tasks to it. Additionally, the internal queue of any worker is set to be of size one, hence any worker can execute only one task at a time.

Finally, when task $u_t$ finishes its execution, at time instant $t + \tau_{exec}$, the *Worker* communicates to the *Task Broker* the end of the execution of the task, sending the total time of execution $\tau$ and its cost per time of execution $\kappa_{exec}$, which in the *expandable pool* case may have value $\nu$ or $\mu$ if the worker is internal or external to the pool $\mathcal{P}$ respectively.

### 3.4.6 Clue component

The environment just described makes also use of another entity which is used by all the others to keep a logical time between all the components necessary for synchronizing them and for following the classic trajectory of a reinforcement learning application. Therefore, a *Global Timestep* has been introduced.

This global clock is increased by the *Task Broker* after a full cycle of

its internal flow, which starts from taking a task from its *task queue* and finishes assigning the task to a worker. The reason for introducing the *Global Timestep* is because the system needs a sort of agreement between all the components for respecting the flow.

Thus, each entity in the environment can communicate with the *Global Timestep* for asking the current time step $t$ and accordingly to its value they act as described so far.

## 3.5 Summary

In this chapter we have deeply described all the relevant aspects of *Service Broker* application. Thus, in the first part of the chapter we have started describing the problem, we have then formulated it, both as *assignment problem* and more in detail as *reinforcement learning* problem. In this latter case, we have illustrated precisely how we map any component of the system on the *Markov Decision Process* setting, showing the action space, the reward function and the state used for *Service Broker*.

In the second part of the chapter, we have concentrated on the description of the type of agents used, showing some of their strengths and eventually some of their weaknesses. We have also introduced the main entities and the architecture used by the environment, which has been created from scratch with the purpose of allowing a simulation of the allocation problem studied.

In the fourth and final chapter, we are going to present some of the implementation details which are worth mentioning, relatively on how we have created the system, on how we have ensured the correct behaviour of the system and on how we have put together all its components.

Finally, in this last chapter we will show the results obtained applying reinforcement learning on the assignment problem addressed so far, describing also the method used to obtain those results.

# Chapter 4

# Implementation and results

In this last chapter, we present the implementation of the system described so far, *Service Broker*. We will illusstrate, in the first part of the chapter, some implementation details regardless the creation of the environment that we have developed from scratch for simulating the task allocation problem addressed. Subsequently, we will present some implementation details of the agents used and also how they interact with environment provided.

In the second part of the chapter, we will show the results obtained, describing firstly the method and the setting used for training the agents on top of the environment and secondly, showing the actual results derivated by the training performed, comparing also those results with some baselines algorithms to prove the effective value of the solution proposed.

The description of the training and of the evaluation we will be performed on both the application cases, the *fixed pool* and the *expandable pool* ones.

## 4.1 Implementation

In this first section of the last chapter we are going to describe the Implementation details that we consider relevant for the creation of an application such as *Service Broker*.

The application has been written using *python* programming language,

because of its wide support from the community for *statistics*, *machine learning* and *neaural networks*, with packages such as *numpy*, *pandas*, *pytorch* and many others.

### 4.1.1 The environment

The environment of the application written represents one of the core point developed for realizing and testing *Service Broker*, because for being able to concentrate only on the *reinforcement learning* perspective we needed an environment from scratch, which allowed to analyze the application of reinforcement learning to such a task allocation problem, without the necessity of considering any other aspect of the system.

Moreover, any RL application needs to be trained - needs to perform the learning procedure - a huge number of time for converging to the optimal solution. Thus, using a real data center scenario could lead to problems of speed during the training process, because in a real distributed environment the communication between the entities, the servers, has to go through the network. Hence, due to the huge difference in terms of performance between network communication and *inter-process communication* (IPC) through shared memory in a local machine, we have decided to create our environment in a way that it simulates all the behaviour of a data center, being non blocking and asynchronous, but that it could be executed on a single machine and take all the advantages in terms of speed and promptness of IPC through shared memory.

On the other hand, IPC through shared memory arises issues in terms of concurrency, because in this communication model, we have several entities, the processes, which have to read or write on the same piece of memory, thus some concurrency primitives are necessary to avoid more than one process at a time to interact with the shared resource.

Therefore, it is clear that the natural way of implementing a simulator for distributed systems is through processes. In fact, the main entities of the environment described in the previous chapter are implemented as processes

which are isolated in terms of memory, thus they simulate to be distributed in the data center, are fully asynchronous due to their isolation and are able to communicate with other processes using shared memory, which is faster then IPC through the network.

The three main components of the environment's architecture are implemented as processes. Indeed, the *Task Generator* has been implemented as a process because it has to simulate an external flow which, when it has some tasks, sends them to the environment without the need to block other components or without the necessity of waiting for other components. *Task Broker* entity is as well implemented as a process, because to operate it needs its own internal state to be isolated and, moreover, because logically it represents an individual entity of the system which is not tied to any others, which has its own internal flow of execution that allows it to communicate and interact with the reinforcement learning agents. Finally, the last entity of the system to be implemented as a process is the *Worker*, which has to simulate a distributed node of a data center, therefore its natural behaviour is to be completely independent and to concentrate only on the execution of the eventual tasks received.

Finally, the environment has to execute the *external flow* - which is receiving, assigning and executing tasks in an asynchronous way - but, on the other hand, it needs to have an *internal flow* which is somehow synchronized among the entities, to respect the trajectory and the requirements of a reinforcement learning application, which needs a global time $t$ and accordingly to this time it needs to receive the current state, to take the current action and, when available, to observe the reward. Hence, a component such as the *Global Timestep* is required and it is used in combination with other process synchronization techniques to coordinate the *internal flow*. Another reason for having this internal synchronized flow is that we want the system and the simulation to be reproducible, thus we can not rely on the timings of context switches of the processes, which can lead to unpredictable results.

### Inter-process communication

In the environment created for *Service Broker* application, entities need to communicate for sharing tasks and information about tasks execution. Hence, we need some primitives for *inter-process communication* (IPC) and, as we have already discussed, in our case the best solution is to use shared memory. Therefore, we have implemented two objects which take advantages of *python* primitives for shared memory and on top of that we have developed a *Shared Queue* and a *Shared Table*, which are used to communicate information with other processes.

The former, *Shared Queue*, implements the classic behaviour of a FIFO queue, with the difference that this queue is stored in slots of the memory shared by the processes of the environment. Therefore, this queue needs to use some lock mechanisms to avoid more than one process, at a time, to write or read from it, thus to avoid that the so called *critical section* is executed by more than one process at a time. The *Shared Queue* implemented has been used for sharing tasks between *Task Generator* and *Task Broker*, sharing tasks between *Task Broker* and *Workers* and by the *Workers* to let the *Task Broker* know when a task completes its execution.

The latter, *Shared Table*, implements a shared memory version of the *python* implementation of a *hashtable*, which is called *dictionary*. This table, as the *Shared Queue*, uses locks for controlling the access to the *critical section*. Moreover, the reason for implementing such a table is to allow the environment to share information of any type among the processes. In fact, an instance of this table is created by the *Task Broker* and then is used by the *Task Generator* to create a record for each task generated, which will then be populated by the *Task Broker* and the *Workers* putting information about the time of execution, the cost of execution, the reward and many others.

### Synchronization techniques

As we have discussed so far, some synchronization techniques are required to make the *Service Broker*'s environment work properly with the

reinforcement learning. In *python* it is possible to use classic system processes primitives, such as locks, barriers and semaphores, but there are also some high level constructs built on top of those primitives which are more expressive and powerful. Thus, we have decided to use:

- **Event Objects**: it is a straightforward mechanism for communication between processes, where one signals an event and the others wait for it. Thus, it allows to make a process wait for the occurrence of a certain event and then to take an action only after the event's occurrence.
- **Condition Objects**: it is a mechanism shared by several processes which allows a process to acquire a *critical section* through the use of locks, causing the other processes to wait releasing the acquired resource. The difference between this object and a classic lock is that in *Condition Objects* it is possible to notify all the processes of the releasing of the *critical section*.

These two synchronization objects have been widely used in the environment. The former is used by the *Workers* to notify when they are busy or not and it is also used by the *Task Broker* to communicate to both *Task Generator* and *Workers* when they have to terminate their execution. The latter is used by the *Task Broker* to notify the other entities of the end of one time step $t$, which corresponds also to the increment of the *Global Timestep*.

In addition to *Event* and *Condition* objects for synchronizing the environment we needed also to create an additional synchronization object, which allows to register several processes and to keep an additional process waiting until all the other processes registered notify him. This component is not by default implemented in *python* and it is typically called *Count Down Latch*. We have implemented it using an internal counter which is set equal to the number of processes registered; then, using a lock, each process registered can decrease the counter by one. In the meanwhile, the additional process, which is waiting, would continue to wait until the counter reaches zero and only when it does the lock is released and the process can continue its execution.

*Count Down Latch* has been used to keep the *Task Broker* waiting, while it finishes the execution of a time step $t$, and to allow the *Task Generator* and all the *Workers* to perform their cycle of execution for the following time step $t+1$, notifing and decreasing the counter by one, only when they complete their turn.

### Task

Tasks used in *Service Broker*'s environment are a simulation of a real task which requires $\tau_{exec}$ time step to be executed. Thus, tasks are implemented as normal *python* objects, even if in a real data center a task would be a process which executes isolated inside one of the *Workers*, but for the purpose of the simulation this behaviour is obtained thanks to the fact that tasks are executed inside a *Worker*, which is implemented as a process, and thanks to the fact that each *Worker* can execute only one task at a time.

Therefore, in the task implementation what is crucial is the time of execution $\tau_{exec}$, which is drawn from a normal distribution $P_{c^u,c^w} = \mathcal{N}(\mu, \sigma^2)$, which depends on the class of the task $c^u$ and the class of the worker $c^w$. To obtain this distribution each task class configuration has been generated defining a mean $\mu$ and a standard deviation $\sigma^2$. In particular, $\sigma^2$ has been fixed to have value equal to 2, while $\mu$ is randomly drawn from a uniform distribution in a range between 10 and 40. The actual time of execution obtained by this configuration is shown in figure 4.1 where is shown, accordingly to different

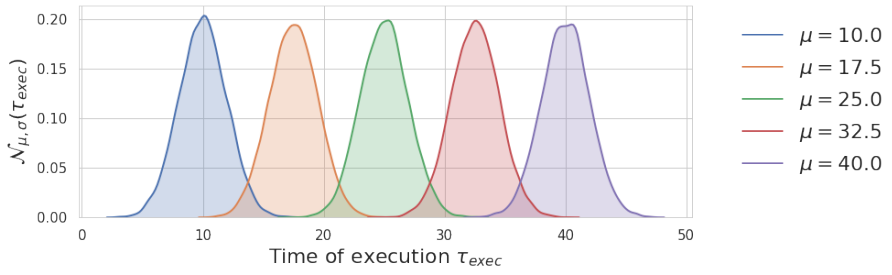



**Figure 4.1:** Probability density function of the Gaussian distribution $P_{c^u,c^w} = \mathcal{N}(\mu, 2)$ with $\mu \in [10, 40]$.

values of $\mu$, the time of execution $\tau_{exec}$ in which a task $u \in c^u$ would have to complete its execution on a worker $w \in c^w$. In addition to the time of execution $\tau_{exec}$, the Task class has also in internal parameter which gives the cost per time step which the task wait before being assigned, $\kappa_{wait}$, this value is important because it can be used to give higher priority to a task class $c^u$, giving an higher $\kappa_{wait}$.

### Internal flow

Now that we have described all the components used to communicate, to synchronize and to the get the actual time of execution of a task $u$, we can finally illustrate the *internal flow* of the environment.

The iternal execution of the environment begins with the initialization of the three main entities and their communication and synchronization components. Subsequently, the *Global Timestep* is initialized and set to time step $t = 0$.

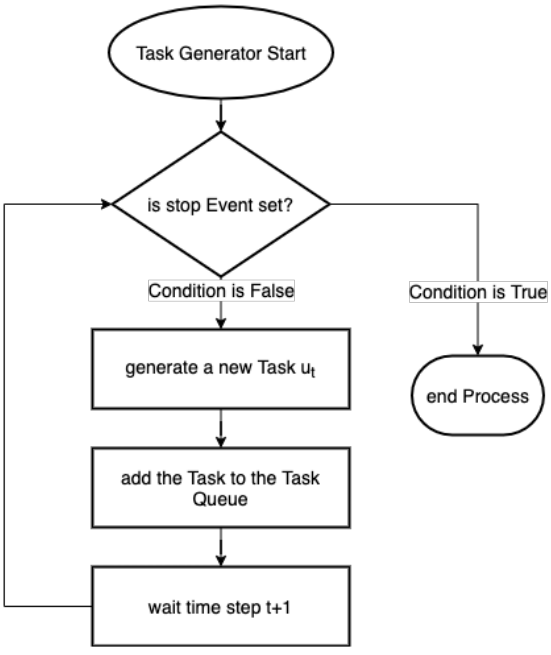



**Figure 4.2:** Internal execution flow of the *Task Generator*.

Following the initialization phase, starts the real execution, in which *Task Generator* and all the *Workers* perform their first execution cycle.

The former, described in figure 4.2, during each time step $t$ would firstly check if *Task Broker* has set its internal *stop Event* and in such a case it would stop its execution. Otherwise, the *Task Generator* would generate a new task $u_t$ which belongs to a task class $c^u$ accordingly to the generation probability $\rho$, which during the simulation performed was drawn from a uniform distribution. Finally, *Task Generator* would put task $u_t$ inside the *Task Broker*'s *task queue*

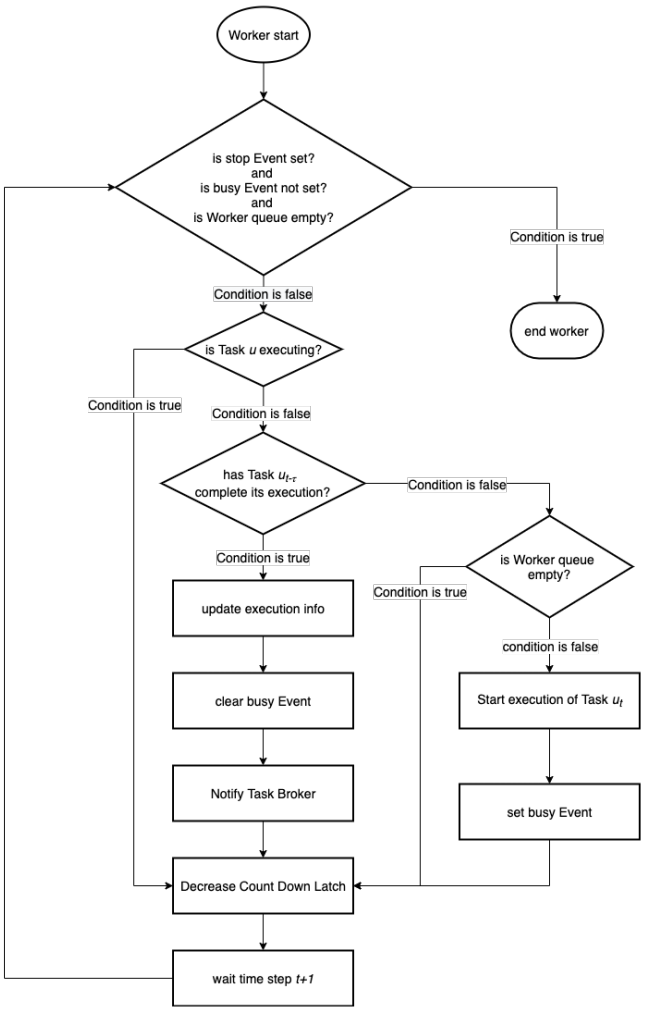



**Figure 4.3:** Internal execution flow of a *Worker*.

and it would start waiting for time step $t + 1$.

The latter, described in figure 4.3, at each time step $t$ checks if it has to stop its execution, which is the case only when the *Task Broker* has set its internal *stop Event*, the *Worker* is not busy, which means that no execution is running and when the *Worker*'s queue is empty. Otherwise, the *Worker* would perform three different checks - with a predefined level of priority - and accordingly to their results it would behave differently. The checks performed are:

1. *is task u executing?* in such a case the *Worker* would not perform any other operation.

2. *has task $u_{t-\tau}$ completed its execution?* if a task started at time step $t - \tau$ has completed its execution, the *Worker* will update the *Shared Table*, shared among itself and the *Task Broker*, inserting the total time of execution $\tau_{exec}$ and its cost for time step execution $\kappa_{exec}$. Then, it will clear the *busy Event* and it will complete its execution cycle.

3. *is Worker queue empty?* in case the queue is empty the *Worker* would not perform any other operation. Otherwise, it would get the first task $u_t$ from the queue, it would start $u_t$ execution and it would set its own *busy Event*.

The *Worker* before completing the execution cycle will always decrease the *Count Down Latch* and it will start waiting for next time step $t + 1$.

Subsequently to *Task Generator* and *Workers* execution cycle, at time step $t$ also *Task Broker* performs its own execution cycle, which is described in figure 4.4.

At each time step $t$, the *Task Broker* checks if it has to stop the whole execution of the environment. This control is carried out on multiple factors which have to be satisfied to stop the environment. The controls are: the check on the current time step if it has reached the value requested for the simulation, the control on the *Task queue* if it is empty, otherwise we have

to firstly execute all the tasks, and, finally, the control on the workers *busy Event*, because to stop the environment all the workers need to finish their executions. Moreover, there is an additional *Event* which can be set externally to force the environment to stop, but also in this case only if all the *Workers* are not busy.

If the stop condition is not met the *Task Broker* would continue to perform

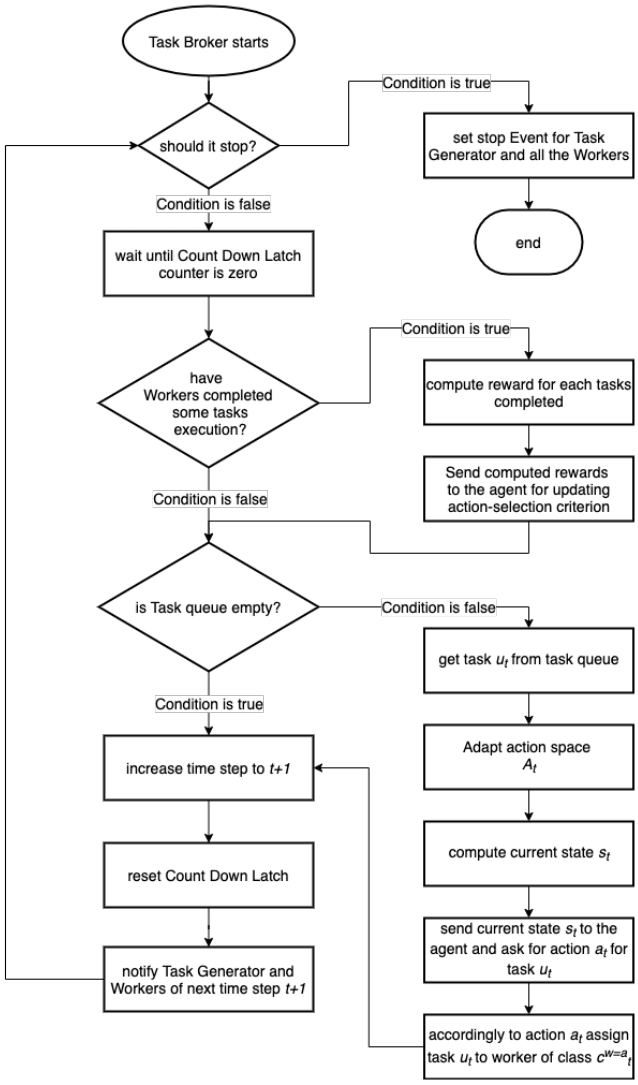



**Figure 4.4:** Internal execution flow of the *Task Broker*.

its execution cycle, which begins waiting for the counter of the *Count Down Latch* to reach zero. Subsequently, it would check the *Shared Queue* used by the *Workers* to communicate the end of the execution of tasks. Thus, if some tasks have completed their execution at time $t$, *Task Broker* would compute the reward for each of them and it would send the agents these rewards to allow the agent to update its action-selection criterion.

After the control on the workers' *Shared Queue*, *Task Broker* checks the *Task Queue*, if it is not empty it would get task $u_t$ from it. It would firstly adapt the action $\mathcal{A}_t$ to contain only actions for which at least one worker is available in the pool $\mathcal{P}$, secondly it would compute the current state $s_t$, accordingly to all the state's features updated to the current time step $t$. Finally, it would ask the agent the action $a_t$ relatively to task $u_t$, accordingly to $\mathcal{A}(s_t)$. The agent would use its knowledge to choose its action and the *Task Broker* would assign $u_t$ to a worker of class $c^w$ equal to action selected $a_t$.

Finally, after having controlled both *Task Queue* and the workers' *Shared Queue*, *Task Broker* would increase the current time step to $t + 1$, it would reset the *Count Down Latch* counter and it would notify all the workers and the *Task Generator* of the beginning of the new time step $t + 1$ and therefore the beginning of a new *internal flow* cycle of execution.

It is worth pointing out that the *internal flow* just described is relative to the *fixed pool* case of the problem, while in the *expandable pool* case there are few differences in the *Task Broker* flow. In particular, when the agent chooses an action $a_t$, if the action belongs to $\mathcal{C}^{w_{external}}$, the *Task Broker* would initialize a new external worker $w \in c^w$ with class equal to the class chosen by the agent. Moreover, in the *expandable pool* case when a worker completes the execution of a task, the *Task Broker* would check if it is an external worker and, if it is, the broker would set its *stop Event* causing the removal of this external worker.

### 4.1.2 Agents

In *Service Broker* we have implemented several different types of reinforcement learning agents that we have illustrated and described in the previous chapter. For their implementation we have applied classic concepts of *Object Oriented Programming* (OOP) paradigm, because all the agents share the same main structure which allows to cooperate with the environment in a complete transparent way, without the need to change any part of the *internal flow* just described.

Thus, we have created an *Abstract Agent* class, which has some utility methods already implemented, such as methods for saving, loading and initializing the model - which represents the knowledge of the agent - or methods for adapting the action space accordingly to the available workers in the internal pool $\mathcal{P}$. On the other hand, this *Abstract Agent* has the signature of other methods but not their implementation, which is left to the real agent that will extend this base version.

To be more precise, among the methods not implemented in the *Abstract Agent* there are:

- *choose action*: it is used by the environment to ask an action at time step $t$. This method takes as arguments three parameters that are task $u_t$, which has to be assigned, the current state $s_t$ and the current time step $t$. As a result, it returns the action $a_t$, which has to be executed by the environment.

- *observe delayed reward*: it is used by the environment to provide the reward $r_{t-\tau}$ to the agent, relatively to a task $u_{t-\tau}$, which completed its execution at time step $t$. This method takes as arguments the action $a_{t-\tau}$, which is the action used to assign the task already completed, the reward $r_{t-\tau}$ and the state $s_{t-\tau}$ relative to the observation of the environment at the moment of the action selection $a_{t-\tau}$.

The *Abstract Agent* class is then extended and an *Agent* class is created for each of the agents used in *Service Broker* application. Moreover, considering

that for *Multi-armed Bandit* and for *DQN* we have used two different variants of those agents we have some additional behaviour for these two types of agents.

### Multi-armed bandit

The *ε-greedy* and *UCB Multi-armed bandit* agents share most of their behaviour, because both do not consider the current situation, the current state $s_t$, when they choose an action and because both update their action-values estimates using the same update technique. Thus, the only difference in these kinds of agents is in the action selection, in *choose action* method.

Therefore, we have decided to implement another object, called *Policy*, which is used by *Multi-armed bandit* agents to define their action selection criterion. This policy object will be used in the method for choosing the action and we have implemented two different policies. The former chooses an action accordingly to *ε-greedy* method. The latter chooses an action based on *UCB* criterion.

### DQN

The *Deep Q-Network* (DQN) agent and its improvement variant, *Double-DQN*, as described in the third chapter share all the behaviour, with the only exception of the computation of the target $y_t$ (recalling algorithm 3 and rule 3.12).

Therefore, the implementation of the DQN agents has been done, firstly creating a *DQN Agent* class which extends the *Abstract Agent*. In this agent class we implement all the methods, with the behaviour of the classic DQN algorithm (algorithm 3). Subsequently, for the *Double-DQN* agent class, we have extended the *DQN Agent* class overriding only the method relative to the computation of the target update $y_t$, inheriting the rest of the behaviour from its parent class.

Moreover, both DQN implementations required the creation of some additional components necessary for the execution of this complex reinforcement

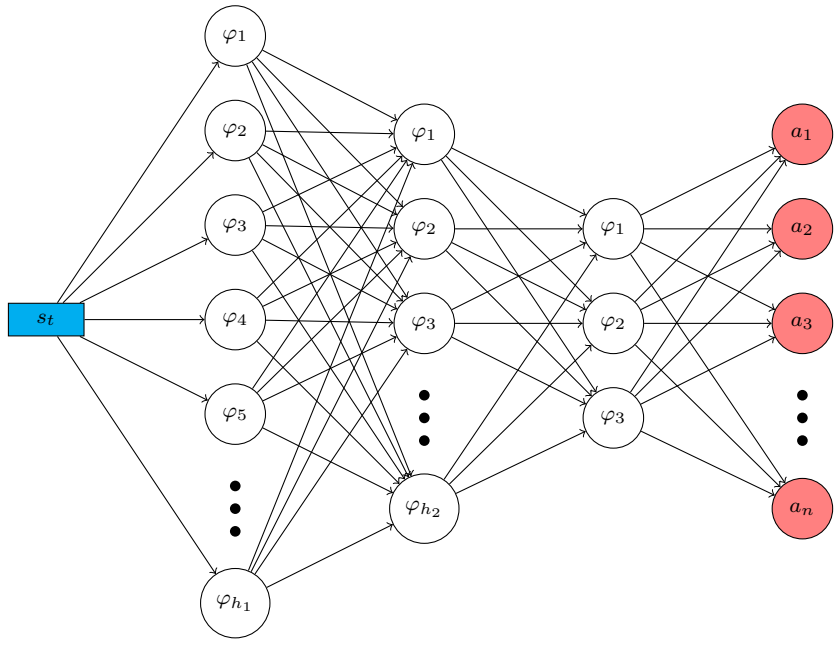



**Figure 4.5:** Illustration of the three hidden layers neural network used by DQN agents.

learning solution, which are the Neural Network model used by the DQN agent to approximate the action-value function $Q$ and the *experience replay* used to reduce the variants of the updates breaking the samples correlations.

The *Neural Network* (NN) used in *Service Broker* is a simple and small *fully connected* NN, which is shown in figure 4.5. This NN has been implemented using *pytorch*[1] python library, which provides utilities for building and running *Neural Networks*.

In the NN implemented the input layers are composed by a neuron for each feature of the state $s_t$, which in figure 4.5 is represented as a single unit for simplicity. Then the network is composed by several hidden layers, where each layer takes as input a number of neurons equal to the output of the previous one and its output is a number of nodes smaller than the number of

[1]https://pytorch.org/

nodes received in input. Finally, the output layer is composed by a neuron for each action $a \in \mathcal{A}$.

Therefore, DQN agents, at each time step $t$, would feed current state $s_t$ into the action-value function, the NN described so far; the result that it would obtain is a probability for each action $a$, given by the value in the output layer, then accordingly to current action space $\mathcal{A}(s_t) \in \mathcal{A}$, restricted to the only actions available, the agent would peek as action $a_t$ the action with the highest probability in $\mathcal{A}(s_t)$.

The reason for such a tiny network is due to the size of the state $s$ which is limited to $2m + 1 + 2|\mathcal{A}|$, where $m$ is the number of task classes. Therefore, to prevent the neural network from overfitting, thus to prevent the agent from adapting the weights of the network closely or exactly to the particular structure of the data used during the training, which indeed can cause worse results on fresh data, we have decided to not have a huge Neural Network.

In *Service Broker* application the *experience replay* has been implemented creating an ad-hoc object which offer some methods for interacting with it.

Thus, the *Experience Replay* class has been implemented using as data structure for keeping experience entries a list of fixed size $N$, which is the capacity of the memory buffer set when the DQN algorithm is initialized. Each entry of this list is then forced to be a tuple of four elements: current state $s_t$, action $a_t$, reward $r_{t+\tau}$ and next state $s_{t+1}$.

Moreover, the class provides two methods for interacting with the memory buffer:

- *push*: it is used to add a new entry in the replay buffer. If the actual size of the buffer is lower than the capacity $N$, then the entry is inserted in the buffer directly. Otherwise, one of the previous entries has to be removed to make place for the new one. Typically, this replacement operation is performed following a FIFO criterion or a random one; in our *Experience Replay* class the replacement is random, thus an entry chosen randomly is removed when the size of the internal list reaches the capacity $N$.

- *sample*: it is used to get from the experience replay buffer $n$ entries, the selection of the entries is performed randomly on those present in the buffer.

During the execution of the *internal flow*, the environment, more precisely the *Task Broker*, has to fill the *experience replay* when a DQN agent is in use. However, in a scenario such the one of *Service Broker* in which we observe the reward after $\tau$ time steps, the operation of feeding the experience buffer is not trivial. Therefore, in such a case *Task Broker* uses an internal table to store partial entry of the memory replay and only when it has the whole entry it can perform the *push* operation on the buffer.

Thus, at each time step $t$ when an action $a_t$ is selected by the agent, the *Task Broker* would store in a partial entry the current state $s_t$ and the action $a_t$; in the next time step, $t + 1$, after having computed the next state $s_{t+1}$ it is added to the partial entry. Finally, at time step $t + \tau$, when also the reward $r_{t+\tau}$ is received, the entry is completed and inserted in the *experience replay* buffer.

Finally, the implementation of DQN agents required also to choose an *optimizer* and an $\varepsilon$ *decay* method. The former is used to control the learning rate, thus to control how much to change the model in response to the estimated error each time the model weights are updated. *Learning rate* is crucial because if it is too small the learning would be slow with probability of remaining stucked in a local minimum, while if it is too big it can lead to a model which approximates too much and thus is unstable. Therefore, in our DQN class we have decided to use the *pytorch* implementation of the adaptive optimizer *Adam*, which adapts the learning rate during the training and was proposed by Kingma et al (2014) [28].

The latter is necessary because it is important at the beginning of the training to allow the DQN agent to explore very often, thus to have an $\varepsilon$ higher, while the more we have trained the more it is necessary to exploit the knowledge built, thus to have a smaller $\varepsilon$. Therefore, in our implementation of DQN we have integrated an adaptive $\varepsilon$ parameter which linearly decays

from an $\varepsilon_{start}$ to an $\varepsilon_{end}$ and this decay is performed in $x$ time steps. Hence, at the beginning the agent would act randomly with probability $\varepsilon_{start}$ and after $x$ time steps it would be random with probability $\varepsilon_{end}$, meanwhile it would decrease at each time step linearly.

## 4.2 Results

In this section we are going to illustrate the results obtained running *Service Broker* simulator with the diverse type of reinforcement learning agents studied and developed, with the aim of providing a proof to support the theory explained so far regarding the assignment problem addressed.

We will start describing the method used, thus how we have trained the models, for whose agent's parameters we have tested several different values, how we have evaluated the agents and with which environment settings we have run it.

Subsequently, we will present two baselines algorithms that have been used to compare the agents developed, with the aim of demonstrating that the proposed solution outperforms those baselines.

Finally, we will display the actual results, for both cases of the problem taken under analysis, *fixed pool* case and *expandable pool* case.

### 4.2.1 The method

To evaluate the agents proposed we have performed a two step pipeline, in which we have first trained the agents and secondly we have evaluated their assignment skill. This kind of evaluation procedure is typical of *reinforcement learning* and more generally of *deep learning*, because when *neaural networks* are used they require to be trained, due to the random initialization of the weights.

For *Multi-armed bandit* and *Contextual bandit* agents a training phase is not theoretically necessary, because they are fully online and typically training and evaluation coincide with a single long run. However, to be completely

fair among all the agents and to put them in exactly the same situation we have applied the same two step pipeline with the same amount of time steps also to *Multi-armed bandit* and *Contextual bandit* agents.

During the *training step*, the first phase of our evaluation pipeline, we have run for $N$ time steps each agent with different values for their learning parameters, performing the so called *hyperparameter tuning*, or *optimization*, which is a typical problem in *machine learning* used to find the optimal set of *hyperparameters* for a learning algorithm. Furthermore, to give a greater statistical validity to the training phase, we have run each agent with each hyperparameters combination for several times, using different initial random seeds. Using this approach with different initial random seeds we are able to better cover the domain of the problem and, more importantly, the best combination found would result to be more resilient to eventual noise caused by the environment.

To be more precise, in the environment developed three random seeds are necessary. The first is used by the *Task Generator* and it is responsible for the task class sampling from the uniform distribution which characterizes the class of the tasks generated and has to be kept fixed among all the training executions, because we want the agents to receive always the same training set. The second is used by the *Tasks* and it is in charge of the normal distribution $P_{c^u,c^w} = \mathcal{N}(\mu, \sigma^2)$, which gives the time of execution for a task class $c^u$ in a worker class $c^w$ accordingly to mean $\mu$ and standard deviation $\sigma^2$ of $c^u$. This seed can be kept fixed across all the training executions because we want the agents to face always the same situation. The last random seed is used by the *Agents* and it regulates the initial starting point of the agent; for example in DQN agents it determines the initial random weights of the network. Therefore, this last random seed is the one which has been changed during the training because we want to find the best combination of *hyperparameters* across different agent initializations.

In *Service Broker*'s environment, because of the asynchrony in the execution of the tasks, when we refer to the number of time steps executed, which

is $N$ for the training step, we actually refer to the number of tasks that the system will generate and thus simulate. In fact, the agent may decide to wait several times and the time of execution of a task may cause a task to finish its execution after time step $N$. Thus, stopping the execution at time step $N$ will result in not respecting one of the main points of the assignment problem proposed, which is to allocate all the tasks.

The second phase of the evaluation pipeline, called *evaluation step*, consists in performing an execution of $M$ time steps, hence a generation of $M$ tasks, over all the agents, but only using the combinations of *hyperparameters* which have obtained the best results from the previous *training step*. It is worth pointing out that also in this phase we run each agent's best combination across all the agents' seeds of the training, because we want to give greater statistical validity also to the process of finding the best performance across the agents proposed and the baselines. Furthermore, because we are in a *reinforcement learning* setting the *evaluation step* is executed for $M$ time steps, but the system restarts from time step $N$, from the training, because in a reinforcement learning application typically we consider the first $N$ time steps as bootstrapping, or training, and the remaining $M$ as actual execution.

Finally, for the evaluation of the system we have set $N = 10000$ and $M = 5000$, while we have configured the environment to generate tasks among 10 different task classes $c^u \in \mathcal{C}^u$ and we have set the number of worker classes $c^w \in \mathcal{C}^w$ to 5. Moreover, the environment in both cases, *fixed pool* and *expandable pool*, starts with a pool $\mathcal{P}$ in which for each class $c^w$ five workers are initialized; this is due to the fact that the mean time of execution of a task is 25 time steps, recalling 4.1, hence this allows the system to rarely be in a situation in which it doesn't have any workers $w$ available in $\mathcal{P}$.

### Hyperparameter trained

For the *training step* the set of hyperparameters that we have compared was:

- $\varepsilon$**-greedy**: we have trained $\varepsilon$*-greedy* agent over different values of $\varepsilon$,

thus varying the level of randomness of the agent. The parameters were trained in the set $\varepsilon \in \{0.1, 0.25, 0.4, 0.5, 0.55\}$.

- **UCB**: *UCB* agent has been training over different values of the costant $c$ which regulates the degree of exploration (recall3.9). The value of the parameter has been trained in the set $c \in \{0.001, 0.01, 0.1, 2, 3, 4\}$.
- **LinUCB**: for *LinUCB* agent we have trained changing values of the costant $\delta$ which is used to control the degree of exploration (recall 3.10). To be precise, $\delta$ is used to compute $\alpha = 1 + \sqrt{\frac{\ln(\frac{2}{\delta})}{2}}$, thus $\delta$ has to be bound to be lower than 2, otherwise $\alpha$ goes in the complex domain, because the natural logarithm would take a value lower than 1, which causes a negative value inside the square root. Therefore the parameter was trained in the set $\delta \in \{0.01, 0.1, 1, 1.5, 2\}$.
- **DQN/Double-DQN**: both *DQN* agents require the same learning *hyperparameters*, hence we consider them together. They have been trained changing the values of initial *learning rate*, of $\varepsilon_{start}$ and of the number of hidden layers. The former represents the starting learning rate that would be used by *Adam* optimizer and has been trained in the set $\{0.1, 0.01, 0.001\}$. The second is the starting $\varepsilon$ used by the agent in the linear decay $\varepsilon$ implemented; $\varepsilon_{end}$ and the time steps necessary to reach it have been kept fixed to 0.1 and 5000 time steps respectively, while $\varepsilon_{start}$ has been trained in the set $\{0.6, 0.9\}$. The latter represents the number of hidden layers to generate in the NN used in 4.5 and has been trained in the set $\{3, 4\}$.

### DQN validation

The DQN agents to perform their training require also to use the concept of validation set, typical of *supervised learning* with NN, where the agent is trained over a training set, but during the training the correctness of the training is obtained through a different set of the same distribution called validation set. This technique is used to avoid overfitting in the NN.

For the case of reinforcement learning through DQN we do a similar procedure: during the training every $x$ time steps a validation run is executed in parallel using the current model, the same seed for the agent, but a different seed for the *Task Generator* and the *Tasks*, due to have a different population of tasks generated. At the end of the validation run, if the total reward obtained is the biggest obtained until that moment, the model used for the validation is kept and considered the actual best. Therefore, at the end of the training the best model among all the validations is the model that would be used for the evaluation step.

In *Service Broker*'s environment, implementing the validation procedure - as we have described - required the introduction of an additional component called *Validator*, which has to be ready to start those validations in parallel, while the main training execution is running.

Therefore, we needed to decouple the validator from the main environment execution and we needed to introduce another inter-process communication procedure. In this case, because of the low number of interactions between the main execution and the *Validator*, we have introduced an IPC at a higher level, which uses an event broker to exhange messages for starting and communicating the results of the validations. We implemented the event broker using a *MQTT* event broker, which is typically used for lightweight communication in *Internet of Things* (IoT) scenarios.

The validation procedure is described in figure 4.6 and it is composed of the following steps:

1. at the beginning of the entire execution, main environment and *Validator* subscribe to receive messages on the *MQTT End validation* and *Start validation* topic respectively.

2. at time step $x$ the main environment execution decides to start a validation run; therefore it publishes a message on the topic *Start validation*, requiring to start a validation, attaching in the payload of the message the configuration to be used for the validation and saving the configuration internally.

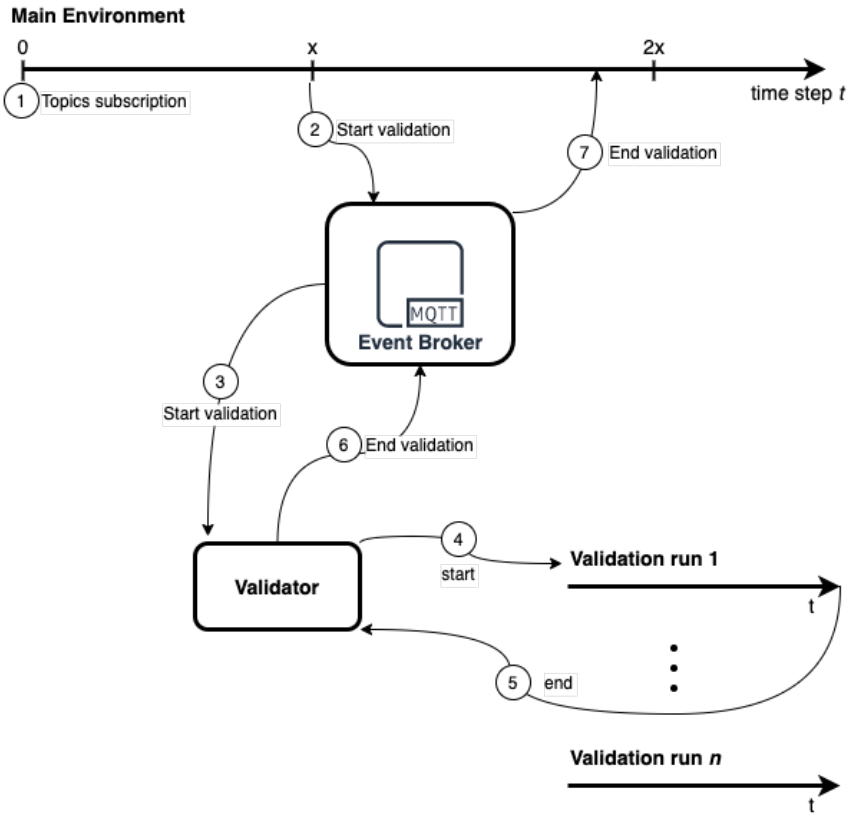



**Figure 4.6:** Validation run mechanism through the MQTT Event broker.

3. the *MQTT* broker receives the message and forwards it to the *Validator*.

4. the *Validator* receives the starting message and starts on a sub-process an execution of the environment using the configuration specified in the payload of the message. The validation runs in a sub-process because the validator should be able to start other validations in parallel.

5. the validation run terminates its execution and communicates the *Validator* the total reward obtained with the configuration requested.

6. the *Validator* receives the total reward of the validation completed and publishes a message on the topic *End validation* attaching the total reward.

7. finally, the *MQTT* broker receives the message and forwards it to the main environment execution, which would compare the total reward received with the previous best and, if this value is greater, then a new best model is saved with the configurations used to start the validation at time step $x$

In the simulation performed each validation run has been executed for 400 time steps and a new validation run has been started every 500 time steps.

### 4.2.2 Baselines

The evaluation of the solution proposed has been performed not only over the agents described so far, but also over two baselines algorithms used to compare the results obtained by the reinforcement learning agents against some techniques typically used to address the problem studied.

In practice, those baselines used have been implemented like the other agents, extending the *Abstract Agent* class and implementing the *choose action* method, while the *observe delayed reward* has been left unimplemented because both baselines do not need to observe the reward to improve their action selection criterion.

#### Random baseline

The former baseline introduced for the evaluation is the easiest possible, available in every type of problems: it is a *Random Agent*, which does not have any knowledge or any criterion for selecting an action. Thus, in *Service Broker* the *Random Agent* at each time step $t$ would randomly choose an action $a_t$ among those available at time $t$, hence $a_t \in \mathcal{A}(s_t)$.

#### LRU baseline

The second baseline introduced in the evaluation procedure is the *Least Recently Used* (LRU) baseline, which is a straightforward technique, but even that is typically used in real task allocation application, and it is similar

to the one described by *Service Broker* for two main reasons. Firstly it is extremely simple to implement, thus it allows fast developing, while on the other hand it results to be particularly effective in those kinds of problems.

LRU solution is based on the principle by which if a resource is the least recently used, with high probability it would be free to execute new tasks and, since this method is applied, it is guaranteed that all the resources are going to be used; thus it is also a fair solution.

To implement LRU baseline we have used a list of size equal to the size of the action space $|\mathcal{A}|$ to store the number of time steps in which each action has not been taken. In this list each index corresponds to the index of one of the actions $a \in \mathcal{A}$ and at the beginning each index is associated to a value of zero. Subsequently, when at time step $t$ an action $a_t$ is selected all the elements of the list are updated adding one to their correspondent values except for the value of the action selected which is set to 0. The selection of the action is then performed choosing the maximum in the list. If more than one action has the same maximum value a random action is selected among those with the highest value in the list, which means that they are the least recently used.

### 4.2.3 Fixed pool case results

Following the method described so far, we have run the *training step* and the *evaluation step* for 10000 and 5000 time steps respectively, using 5 different random seeds for the agent's initialization. Moreover, the evaluation has been performed over both the cases addressed in this work, that is *fixed pool* and *expandable pool*.

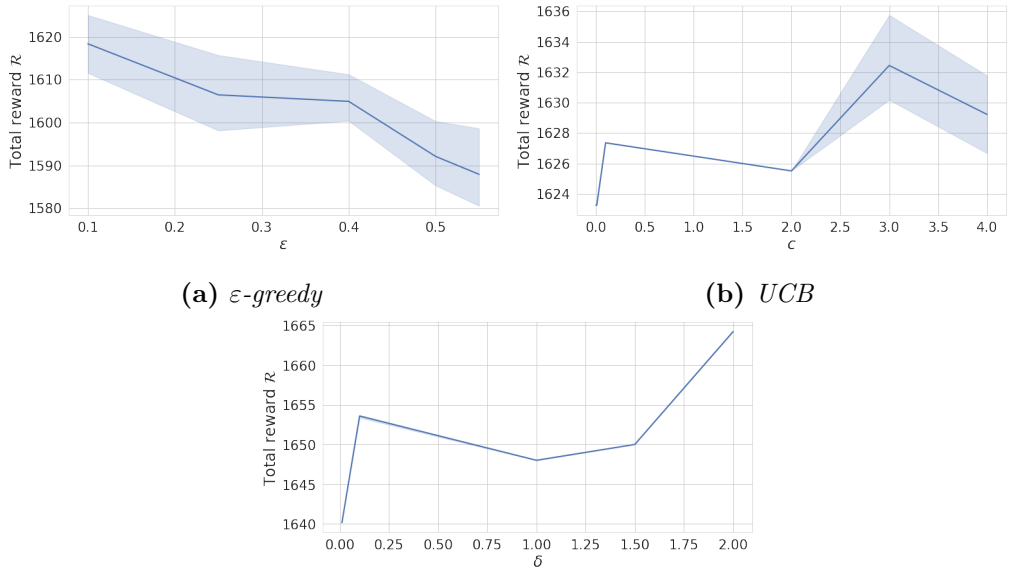
We will start presenting the results for the simpler case, dividing the hyperparameter training from the actual comparison among the agents and, subsequently, we will present the results of the more complex case.

### Training results

In figure 4.7 is shown the exploration of the hyperparameters space, the training, for the *Multi-armed bandit* and *Contextual bandit* agents in *Service Broker fixed pool* case, while in table 4.1 is shown the hyperparameters exploration for the *Double-DQN* agent in the same settings of the other agents.

Moreover, the training has not been performed for the DQN agent, because it has been proven by Hasselt et al. (2015) [23] that *Double-DQN* always outperforms DQN due to its ability to better deal with noise; therefore we have concentrated only on the *Double-DQN*.

For $\varepsilon$*-greedy* agent it is displayed in figure 4.7a, on the y-axis, how the total reward changes in function of different values of $\varepsilon$, x-axis. The figure shows average total reward for each hyperparameter value trained and its confidence interval over the different runs executed with different agent's




**(a)** $\varepsilon$*-greedy* **(b)** *UCB*

**(c)** *Lin-UCB*

**Figure 4.7:** Hyperparameters exploration for *Multi-armed bandit* and *Contextual bandit* agents for *Service Broker fixed pool* case, where the total reward, y-axis, is in function of the training hyperparameter, x-axis.

random seeds. It is noticeable that the best value for the total reward is obtained with a value of $\varepsilon = 0.1$; it is also evident that the confidence interval of the hyperparameters trained tends to overlap.

For *UCB* agent the same graph with total reward in function of the hyperparameter trained, in this case $c$, is presented in figure 4.7b. In this context the best value has been obtained with value of $c = 3$, for this agent the range of confidence intervals for values lower than 2 is infinitesimal, meaning that for those values of $c$ the total reward obtained is stabler to noise, instead for values bigger than the confidence interval it is bigger.

The reason why in both *$\varepsilon$-greedy* and *UCB* we obtain noticeable confidence intervals, even if we have executed them with different hyperparameters, is due to the nature of *Multi-armed bandit*, which is not able to associate the action taken to the current situation and therefore it is not able to generalize.

| $lr$ | $\varepsilon_{start}$ | $layers$ | $\mu$ | $count$ | $\sigma$ | $CI_{high}$ | $CI_{low}$ |
|---|---|---|---|---|---|---|---|
| 0.001 | 0.9 | 4.0 | 71.059 | 5.0 | 5.572 | 75.944 | 66.175 |
| 0.001 | 0.6 | 3.0 | 70.857 | 5.0 | 3.981 | 74.347 | 67.368 |
| 0.001 | 0.9 | 3.0 | 70.664 | 5.0 | 4.029 | 74.196 | 67.133 |
| 0.001 | 0.6 | 4.0 | 70.265 | 5.0 | 5.278 | 74.891 | 65.638 |
| 0.01 | 0.9 | 4.0 | 68.846 | 5.0 | 3.815 | 72.191 | 65.502 |
| 0.01 | 0.9 | 3.0 | 68.499 | 5.0 | 1.73 | 70.015 | 66.983 |
| 0.01 | 0.6 | 3.0 | 68.284 | 5.0 | 1.688 | 69.763 | 66.804 |
| 0.01 | 0.6 | 4.0 | 68.246 | 5.0 | 2.122 | 70.106 | 66.386 |
| 0.1 | 0.6 | 4.0 | 67.745 | 5.0 | 1.892 | 69.404 | 66.086 |
| 0.1 | 0.9 | 4.0 | 66.212 | 5.0 | 2.296 | 68.224 | 64.2 |
| 0.1 | 0.9 | 3.0 | 65.521 | 5.0 | 2.871 | 68.037 | 63.005 |
| 0.1 | 0.6 | 3.0 | 65.379 | 5.0 | 1.164 | 66.399 | 64.359 |

**Table 4.1:** Hyperparameters exploration for *Double-DQN* agent for *Service Broker fixed pool* case.

Finally, in figure 4.7c it is shown how the total reward changes in function

of the hyperparameter $\delta$ for the *LinUCB* agent. For this type of agent it is evident that the lower $\delta$ is, the lower the total reward is. This is due to the impact that $\delta$ has, which is used to compute the parameter $\alpha$ used as upper bound for controlling the exploration, thus the smaller $\delta$ is, the bigger $\alpha$ would be (recall rule 3.10), causing exploring too much and thus performing worse.

Regarding *Double-DQN*, in table 4.1 are represented all the combinations of hyperparameters trained over the different seeds. For each combination is reported the average of the max validation reward over the different seeds $\mu$, where the max validation reward is the maximum total reward obtained by the validation on each different execution. In addition to that average value are reported the following: the number of different seeds used, *count*, the standard deviation $\sigma$ and the confidence interval, both high and low *CI*. The results in the table are sorted for the average of max validation reward and it is evident how the learning rate is the major factor which causes different values of $\mu$. In particular, from this training it has resulted that a smaller learning leads to a better learning. Thus, the best combination of hyperparameters is given by: $lr = 0.001$, $\varepsilon_{start} = 0.9$ and $layers = 4$.

### Evaluation results

Following the results obtained in the previous training step, we have taken the best model of each agent and we have performed the evaluation for other 5000 time steps, restarting from the end of training, thus 10000.

To be more precise, for each agent's random seed we have taken the model resulted from the training related to the best hyperparameters combination, thanks to the possibility of the *Agent class* to save and load its learned knowledge. Moreover, we have set the random seed for the *Task Generator* and the *Task* like the same of the training, because we have evaluated continuing the training, thus restarting from the end time step of the training. For the agent's seed instead we have set the seed to be the same as the one used to train that model, thus also the evaluation has been performed for

each agent for 5 times. The evaluation has been performed not only on the 4 agents trained but also over the two baselines presented so far, thus *Random* and *LRU* agents. The results are presented in figure 4.8 and table 4.2.

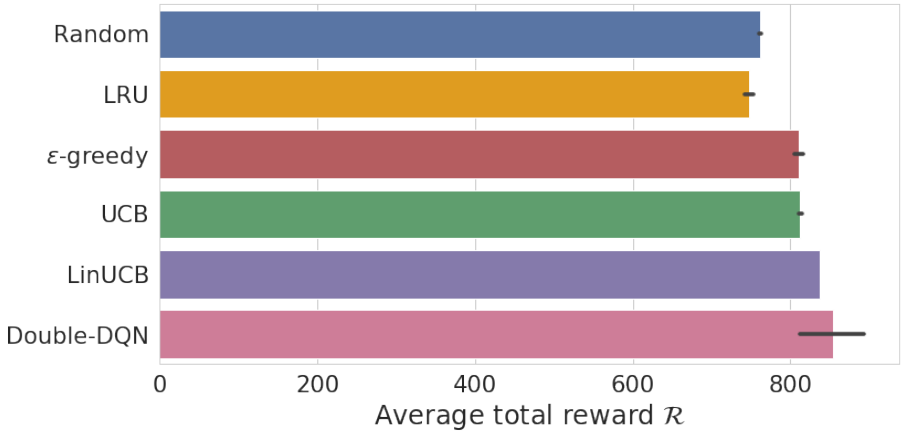



**Figure 4.8:** Fixed pool valuation results, total reward obtained with the best combination of hyperparameters for each agent and comparison with *Random* and *LRU* baselines.

| *agent* | $\mu_{reward}$ | *count* | $\sigma$ | $CI_{high}$ | $CI_{low}$ |
|---|---|---|---|---|---|
| *Random* | 761.562 | 5.0 | 1.351 | 762.747 | 760.377 |
| *LRU* | 747.386 | 5.0 | 7.387 | 753.861 | 740.912 |
| *ε-greedy* | 810.486 | 5.0 | 7.366 | 816.943 | 804.029 |
| *UCB* | 812.681 | 5.0 | 2.514 | 814.885 | 810.478 |
| *LinUCB* | 836.827 | 5.0 | 0.0 | 836.827 | 836.827 |
| ***Double-DQN*** | **853.429** | **5.0** | **53.595** | **900.407** | **806.451** |

**Table 4.2:** *Fixed pool* evaluation results.

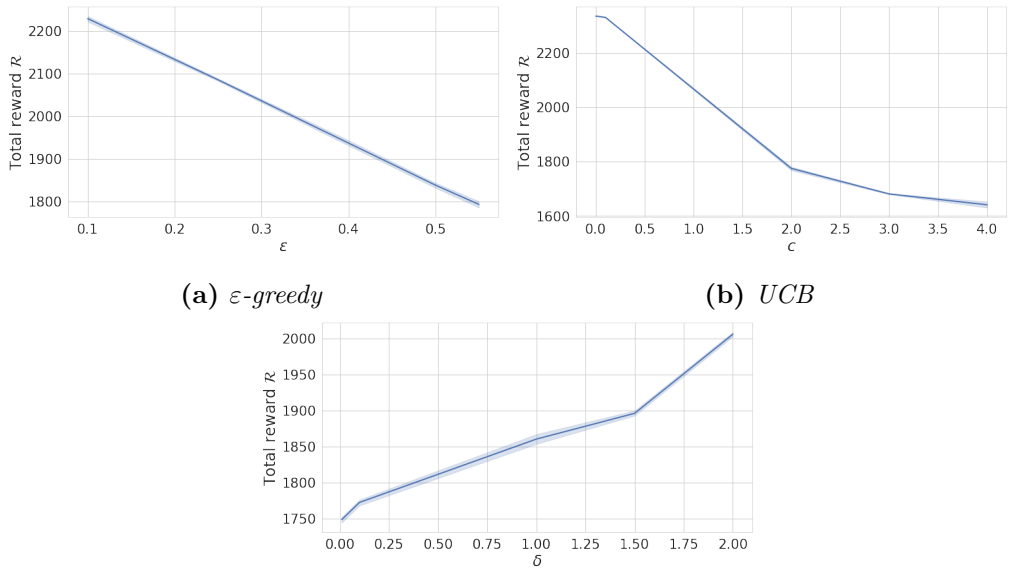
In both hisogram and table, the results are grouped by the agents over the different seeds and the confidence interval is shown. Additionally, in the table are reported also the values of how many different seeds have been used, *count*, the standard deviation $\sigma$ and the confidence interval, both high and

low *CI*.

Overall, from the evaluation it has emerged that all the agents have performed in a relatively close range of mean total reward, which goes from 747.386 to 853.429 and the best agent was *Double-DQN*, even if its confidence interval is higher than the other greatest agents, while the worst is *LRU*. For the *Multi-armed bandit* and *Contextual bandit* agents the confidence intervals are very small, suggesting that their results are strongly resilient to random noise, while this does not happen for *Double-DQN*.

### 4.2.4 Expandable pool case results

Now that we have described the results of the *fixed pool* case we can illustrate the results obtained in the *expandable pool* case, trying to give a general understanding of the overall behaviour. As we did for the previous case, we will start displaying the training results and then the evaluation




**(a)** *ε-greedy* **(b)** *UCB*

**(c)** *Lin-UCB*

**Figure 4.9:** Hyperparameters exploration for *Multi-armed bandit* and *Contextual bandit* agents for *Service Broker expandable pool* case, where the total reward, y-axis, is in function of the training hyperparameter, x-axis.

results. The training results are displayed in figure 4.9 and in table 4.3, while the evaluation results are presented in figure 4.10 and in table 4.4.

### Training results

As for the *fixed pool* case, also in the *expandable pool* case the training has been performed using the same method.

Therefore, in figure 4.9 is presented the hyperparameters exploration of the *Multi-armed bandit* agents and the *Contextual bandit* agent. For *ε-greedy* from the exploration of over different values of $\varepsilon$ it is evident that the smaller the better and thus, an $\varepsilon = 0.1$ represents the best hyperparameter (figure 4.9a).

For *UCB* it is evident that a value of the hyperparameter $c$ bigger than 2 causes the agent to learn worse, while better results are obtained for smaller

| $lr$ | $\varepsilon_{start}$ | $layers$ | $\mu$ | $count$ | $\sigma$ | $CI_{high}$ | $CI_{low}$ |
|---|---|---|---|---|---|---|---|
| 0.001 | 0.6 | 3.0 | 50.564 | 5.0 | 28.171 | 75.256 | 25.871 |
| 0.001 | 0.9 | 3.0 | 47.146 | 5.0 | 22.583 | 66.94 | 27.351 |
| 0.01 | 0.9 | 3.0 | 41.999 | 5.0 | 33.306 | 71.193 | 12.805 |
| 0.001 | 0.9 | 4.0 | 41.662 | 5.0 | 19.685 | 58.916 | 24.407 |
| 0.1 | 0.9 | 4.0 | 41.369 | 5.0 | 31.779 | 69.225 | 13.514 |
| 0.01 | 0.9 | 4.0 | 37.593 | 5.0 | 20.719 | 55.754 | 19.432 |
| 0.001 | 0.6 | 4.0 | 31.321 | 5.0 | 6.636 | 37.138 | 25.504 |
| 0.01 | 0.6 | 3.0 | 30.977 | 5.0 | 27.572 | 55.145 | 6.809 |
| 0.01 | 0.6 | 4.0 | 26.368 | 5.0 | 9.752 | 34.916 | 17.82 |
| 0.1 | 0.6 | 4.0 | 20.807 | 5.0 | 4.74 | 24.962 | 16.652 |
| 0.1 | 0.9 | 3.0 | 15.742 | 5.0 | 4.645 | 19.814 | 11.671 |
| 0.1 | 0.6 | 3.0 | 13.663 | 5.0 | 0.013 | 13.674 | 13.651 |

**Table 4.3:** Hyperparameters exploration for *Double-DQN* agent for *Service Broker expandable pool* case.

$c$, with the best performance obtained for $c = 0.01$ (figure 4.9b).

From *LinUCB* hyperparameters exploration emerged that, as well as for the *fixed pool* case, the best performances are obtained when $\delta$ approaches the value of 2 (figure 4.9c).

The results of the training of *Double-DQN* are presented in table 4.3, which reports the same metrics of the *fixed pool* case. By analyzing the table it emerges that, again, the parameter which effects the training the most is the learning rate and, in particular, a lower value ensures the best performances. The best average maximum validation $\mu$ has been obtained with a combination of hyperparameters composed of $lr = 0.001$, $\varepsilon_{start} = 0.6$ and a NN of 3 hidden layers, but the results of this combination have a very high standard deviation and confidence interval; this could be due to noise and to a small number of different seeds trained.

### Evaluation results

The evaluation of the *expandable pool* case, as for the other case, has been performed over the same random seeds used for the training and restarting from the end of the training, that is restarting from time step 10000 and

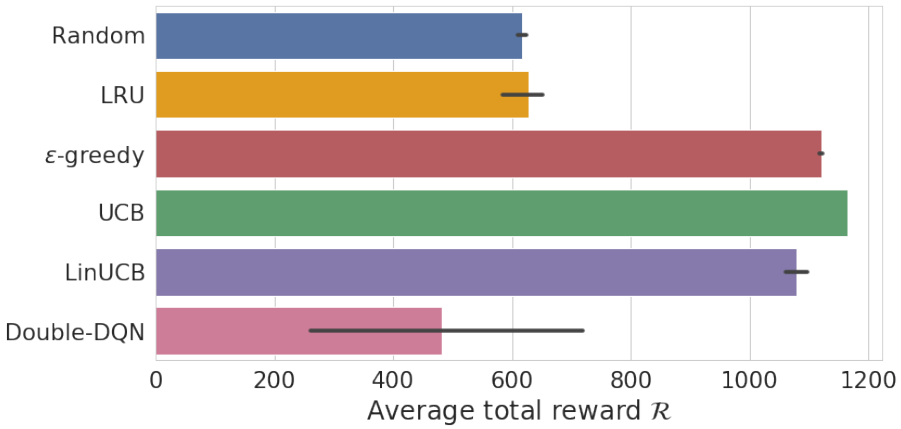



**Figure 4.10:** Expandable pool valuation results, total reward obtained with the best combination of hyperparameters for each agent and comparison with *Random* and *LRU* baselines.

running for 5000 time steps. For each agent we have used the hyperparameters combinations discovered after the fisrt step. The results are shown in figure 4.10 and in table 4.4.

The evaluation in this case resulted with a different best agent, which is *UCB*, immediately followed by the *ε-greedy* and *LinUCB* agents. Overall, all the agents drastically outperform the two baselines proposed, except for *Double-DQN*, which suffers for the lack of a longer training, because as it is for the *fixed pool* case, its confidence interval is very high, causing, in this more complex case, the impossibility of learning a proper assignment knowledge.

From both the table 4.4 and the figure 4.10 it is also evident that for *Multi-armed bandit* and *Contextual bandit* agents the learnt assignment criterion is resilient to random noise, due to the very small confidence intervals.

| *agent* | $\mu_{reward}$ | *count* | $\sigma$ | $CI_{high}$ | $CI_{low}$ |
|---|---|---|---|---|---|
| Random | 616.921 | 5.0 | 10.517 | 626.139 | 607.702 |
| LRU | 627.039 | 5.0 | 48.75 | 669.771 | 584.308 |
| $\varepsilon$-greedy | 1120.633 | 5.0 | 3.528 | 1123.725 | 1117.541 |
| **UCB** | **1164.853** | **5.0** | **0.051** | **1164.898** | **1164.808** |
| LinUCB | 1079.211 | 5.0 | 23.462 | 1099.777 | 1058.646 |
| Double-DQN | 481.601 | 5.0 | 323.797 | 765.422 | 197.78 |

**Table 4.4:** *Expandable pool* evaluation results.

## 4.3 Summary

The overall results obtained, for the *Service Broker* allocation problem, by the agents proposed are to be considered very promising. In fact, in both *fixed pool* and *expandable pool* case the agents have always outperformed the baselines proposed. The only exception is *Double-DQN* in the *expandalbe case*, which results can be seen as inconsistent in the first place, because

*Double-DQN* has been the best on the base case, while it has been even worse than the baselines in the advanced case. In reality, if we consider the confidence intervals on both cases we can notice that this agent always has the biggest range, meaning that the results are less resilient to noise. Therefore, in the simpler case, the *fixed pool* case, even if there is noise it is able to generalize a valid allocation technique, while in the more complex case it is not able to do so. Hence, these results suggest for *Double-DQN* a longer training to allow it to capture all the aspects of the assignment problem to succeed in both cases.

Moreover, from the results on the *expandable pool* case it emerges that *LinUCB*'s context features are not sufficient to outperform the *Multi-armed bandit* agents, as it happens instead in the *fixed pool* case. Therefore, it suggests adding other features in a way to better match all the aspects of the allocation problem in both its variants.

# Conclusion

In this project we have demonstrated how an *allocation problem*, such as distributed task allocation, can be addressed through *reinforcement learning*, obtaining better results than those gained using typical task allocation techniques, such as *Least Recently Used*. Moreover, with the necessity of fast simulation we have also created from scratch an environment capable of simulating the behaviour of a real distributed task allocator, which can be used by the research community to continue studying such a learning field.

Overall, from our study on *Service Broker* task allocation it has emerged that all the agents developed are able to learn how to assign tasks and the policy learnt outperforms the results of a *Random* agent and also of a *Least Recently Used* agent. This result gives a proof of the initial intuition in which we hypothesized that it is possible to find a correlation between task classes and worker classes with the aim of a better assignment.

Notwithstanding the promising results obtained with our *Service Broker* application, the research on *reinforcement learning* applied to *distributed task allocation* is far from being completed. In fact, several future improvements can be done on top of our solution. Firstly, to increase the statistical validity of our results would be opportune to perform both training and evaluation of the agents for a bigger number of agent's random seeds to reduce the confidence intervals obtained and have results more resilient to random noise. Secondly, for an agent such as *DQN* or *Double-DQN* a training of 10000 time steps is considered too short, resulting in poor performances if compared to the other agents and to the potentiality of this technique. Therefore, we

believe that with a proper training the *neural network* of the *DQN* agent would be able to properly approximate the allocation problem resulting in better results. Thirdly, the solution proposed has been tested on top of our environment, which simulates all the aspects of a real task allocator system, but it remains a simulator. Hence, the next step of this project would be to develop and to evaluate the agents on a real task allocator. We believe that the results would be the same, indeed allowing those kinds of system to improve their assignment performances.

To conclude, the study performed on this kind of *distributed task allocation* is very promising, not only because it represents a valid solution for the specific problem addressed, but also because it can be generalized to different types of allocation problems, even those not strictly related to computing systems, such as the examples provided in the first chapter as the carrier delivery service or the manufacturing factory. Therefore, the solution proposed can be used as baseline method also for these types of high level task allocation problems providing a guideline for approaching them and to obtain the best results.

# Bibliography


[1] John Hugh Andreae. *Learning machines: a unified view*. Standard Telecommunications Laboratories, 1966.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2 (2002), pp. 235–256.

[3] Richard Bellman. "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684.

[4] Richard Bellman. "Dynamic Programming". In: (1957).

[5] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. "The job shop scheduling problem: Conventional and new solution techniques". In: *European Journal of Operational Research* 93.1 (1996), pp. 1–33.

[6] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2014).

[7] Richard Conway, William L. Maxwell, and Louis W. Miller. *Theory of scheduling*. English. Originally published: Reading, Mass. : Addison-Wesley, [1967]. Mineola, N.Y. : Dover, 2003.

[8] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[9] George Cybenko. “Dynamic load balancing for distributed memory multiprocessors”. In: *Journal of Parallel and Distributed Computing* 7.2 (1989), pp. 279–301.

[10] George E Dahl et al. “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition”. In: *IEEE Transactions on audio, speech, and language processing* 20.1 (2011), pp. 30–42.

[11] George Bernard Dantzig. *Linear programming and extensions*. Vol. 48. Princeton university press, 1998.

[12] Mauro Dell’Amico and Marco Trubian. “Applying tabu search to the job-shop scheduling problem”. In: *Annals of Operations Research* 41 (1993), pp. 231–252.

[13] A. Demers, S. Keshav, and S. Shenker. “Analysis and Simulation of a Fair Queueing Algorithm”. In: *Symposium Proceedings on Communications Architectures & Protocols*. SIGCOMM ́89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 1–12.

[14] Jeno Egerváry. “Matrixok kombinatorius tulajdonságairól”. In: *Matematikai és Fizikai Lapok* 38.1931 (1931), pp. 16–28.

[15] Ian Foster and Carl Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[16] Michael R Garey, David S Johnson, and Ravi Sethi. “The complexity of flowshop and jobshop scheduling”. In: *Mathematics of operations research* 1.2 (1976), pp. 117–129.

[17] Fred Glover. “Heyristics for integer programming using surrogate constraints”. In: *Decision Sciences* 8.1 (1977), pp. 156–166.

[18] Fred Glover. “Tabu Search—Part II”. In: *ORSA Journal on Computing* 2.1 (1990), pp. 4–32.

[19] Fred W. Glover. “Tabu Search - Part I”. In: *INFORMS Journal on Computing* 1 (1989), pp. 190–206.

[20] Pawan Goyal, Harrick M. Vin, and Haichen Chen. "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks". In: *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '96. Palo Alto, California, USA: Association for Computing Machinery, 1996, pp. 157–168.

[21] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.

[22] Hado V Hasselt. "Double Q-learning". In: *Advances in neural information processing systems*. 2010, pp. 2613–2621.

[23] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015.

[24] JH Holland. "adaptation in natural and artificial systems, university of michigan press, ann arbor,"". In: *Cité page* 100 (1975).

[25] Nathan Jay et al. "A Deep Reinforcement Learning Perspective on Internet Congestion Control". In: *ICML*. 2019.

[26] Pooria Joulani, Andras Gyorgy, and Csaba Szepesvári. "Online learning under delayed feedback". In: *International Conference on Machine Learning*. 2013, pp. 1453–1461.

[27] D 'e nes K "o nig. " "U about graphs and their application to determinant theory and set theory". In: *mathematical annals* 77.4 (1936), pp. 453–465.

[28] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

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

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[31] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97.

[32] John Langford and Tong Zhang. "The epoch-greedy algorithm for contextual multi-armed bandits". In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. Citeseer. 2007, pp. 817–824.

[33] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324.

[34] Lihong Li et al. "A Contextual-Bandit Approach to Personalized News Article Recommendation". In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 661–670.

[35] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[36] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[37] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013.

[38] James Munkres. "Algorithms for the assignment and transportation problems". In: *Journal of the society for industrial and applied mathematics* 5.1 (1957), pp. 32–38.

[39] J. Nagle. "On Packet Switches with Infinite Storage". In: *IEEE Transactions on Communications* 35.4 (Apr. 1987), pp. 435–438. ISSN: 1558-0857.

[40] Sunil Nakrani and Craig Tovey. "On Honey Bees and Dynamic Server Allocation in Internet Hosting Centers". In: *Adaptive Behavior* 12.3-4 (2004), pp. 223–240.

[41] F. Pezzella, G. Morganti, and G. Ciaschetti. "A genetic algorithm for the Flexible Job-shop Scheduling Problem". In: *Computers & Operations Research* 35.10 (2008). Part Special Issue: Search-based Software Engineering, pp. 3202–3212.

[42] O. Rahmeh, P. Johnson, and A. Taleb-Bendiab. "A Dynamic Biased Random Sampling Scheme for Scalable and Reliable Grid Networks". In: *INFOCOMP Journal of Computer Science* 7.4 (2008), pp. 1–10.

[43] M. Randles, D. Lamb, and A. Taleb-Bendiab. "A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing". In: *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*. Apr. 2010, pp. 551–556.

[44] Herbert Robbins. "Some aspects of the sequential design of experiments". In: *Bull. Amer. Math. Soc.* 58.5 (Sept. 1952), pp. 527–535.

[45] Fabrice Saffre et al. "Aggregation dynamics in overlay networks and their implications for self-organized distributed applications". In: *The Computer Journal* 52.4 (2009), pp. 397–412.

[46] J Ben Schafer, Joseph Konstan, and John Riedl. "Recommender systems in e-commerce". In: *Proceedings of the 1st ACM conference on Electronic commerce*. 1999, pp. 158–166.

[47] Pierre Sermanet et al. "Pedestrian detection with unsupervised multi-stage feature learning". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2013, pp. 3626–3633.

[48] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017.

[49] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[50] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (2017), pp. 354–359.

[51] Viswanath Sivakumar et al. *MVFST-RL: An Asynchronous RL Framework for Congestion Control with Delayed Actions*. 2019.

[52] Daniel A. Spielman and Shang-Hua Teng. “Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time”. In: *J. ACM* 51.3 (May 2004), pp. 385–463.

[53] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[54] Gerald Tesauro. “Practical issues in temporal difference learning”. In: *Advances in neural information processing systems*. 1992, pp. 259–266.

[55] Gerald Tesauro. “Programming backgammon using self-teaching neural nets”. In: *Artificial Intelligence* 134.1-2 (2002), pp. 181–199.

[56] Gerald Tesauro. “TD-Gammon, a self-teaching backgammon program, achieves master-level play”. In: *Neural computation* 6.2 (1994), pp. 215–219.

[57] Gerald Tesauro. “Temporal difference learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[58] EL Thorndike. “Animal Intelligence, Darien, Ct. Hafner”. In: *Original work published* (1911).

[59] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. In: (1989).

[60] Ian H Witten. “An adaptive optimal controller for discrete-time Markov environments”. In: *Information and control* 34.4 (1977), pp. 286–295.

[61] Ian H Witten. “The apparent conflict between estimation and control—A survey of the two-armed bandit problem”. In: *Journal of the Franklin Institute* 301.1-2 (1976), pp. 161–189.

[62] Wei Zhang et al. “Shift-invariant pattern recognition neural network and its optical architecture”. In: *Proceedings of annual conference of the Japan Society of Applied Physics*. 1988.

# Acknowledgments

## English version

At the end of this work I would like to remember and thank all those who have helped me, who have given me any suggestion or have supported me during the preparation of this thesis, but more in general during the whole period of study which now is going to be completed.

I would like to thank Professor Bellavista from the University of Bologna, main supervisor of this thesis dissertation, who made this work possible, firstly, making me know about this field and this project and, secondly, allowing me to take part in it. Without you I would have never reached this important goal for my personal and professional growth!

I would like to thank Professor Musolesi from the University College London, second supervisor of this thesis dissertation, who hosted me in his department at the University College London to prepare this thesis and who gave me the help I needed to overcome the difficulties I encountered during the thesis preparation. Professor Musolesi also let Victor Darvariu assist me, one of his PhD student, who was above all a friend during the period in which I lived in London and furthermore he was always ready to help me and to offer his knowledge to support me during the preparation of my thesis. Therefore, I really would like to thank you a lot, Professor Musolesi and Victor Darvariu, for the help you gave to me!

I would like to thank the staff of Imola Informatica S.p.a., in particular Filippo Bosi, CEO of the company, who during the whole period of study of my master degree hired me in their development and research lab, allowing

me to learn and experiment many challenging aspects which improved my knowledge. He also introduced me to Andrea Sabbioni, who worked with me in the lab and with whom I have shared a lot a funny moments while we were working. Andrea, moreover, helped me a lot to improve my knowledge with his advice and his suggestions. Thus, thank you, Filippo Bosi and Andrea Sabbioni!

Now I would like to thank all my friends: those I have known for a long time and those I have met recently. All of you helped me during this period of study, with you I have shared the stress for the my studies, the anxiety before an exam and then the celebration after it; thank you! Moreover, I really would like to thank all my friends from my hometown, Federico, Giammarco, Giacomo and all the others, who have always been ready to make me relax and laugh. I really would like to thank also Angelo and Luca from the University of Bologna, because with them I have shared all the good and bad moments during this period of study. Finally, I really would like to thank all the people that I met in London during my thesis, who helped me through a period far from my loved ones.

A huge thank goes to my family: my brother Alberto and my parents. You support me every day, motivating me to be always the best version of myself and helping me to follow my dreams. I will never forget all the times you have been close to me and you have helped me. Really thank you!

Finally, another huge thank goes to my girlfriend Diletta. She is not only my partner, but she is my best friend, she is my confidant and she is my big love. She is always present and when I need any type of help or every time I need some advice she is always ready to help. She always supports me in every decision I take and she always helps me take all the important ones. She supports me every day and she did it even in the 6 months I was in London for my thesis. I really appreciate her help. Together we always spend unforgettable moments and we are always able to transform a pout into a smile. So thank you my dear love, I love you!

## Italian version

Giunto ora al termine di questo lavoro, vorrei ringraziare tutti coloro che mi hanno aiutato, che mi hanno dato ogni tipo di consiglio o che mi hanno supportato durante la preparazione della mia tesi, ma più in generale durante tutto il periodo di studio che ora volge al termine.

Vorrei ringraziare il Prof. Bellavista dell'Università di Bologna, relatore di questa tesi, che ha reso possibile questo lavoro: prima di tutto mi ha fatto conoscere questa tematica e questo progetto e successivamente mi ha reso parte di esso. Senza di Lei non sarei qui a discutere questo lavoro!

Vorrei ringraziare il Prof. Musolesi dell'University College London, correlatore di questa tesi, che mi ha ospitato alla University College London, nel suo dipartimento, per preparare la tesi e che mi ha dato l'aiuto di cui necessitavo per superare le difficoltà incontrate durante la preparazione della tesi. Il Prof. Musolesi mi ha anche affiancato un suo PhD, Victor Darvariu, che prima di tutto è stato un amico durante il periodo in cui ho vissuto a Londra ed è stato sempre disponibile nell'aiutarmi e nel mettere a disposizione la sua conoscenza al fine di supportarmi durante la preparazione della tesi. Perciò, voglio davvero ringrazie molto entrambi, Prof. Musolesi e Victor Darvariu, per l'aiuto che mi avete dato!

Vorrei ringraziare tutta Imola Informatica S.p.a., in particolare Filippo Bosi, amministratore delegato dell'azienda, il quale durante l'intero periodo di studi della mia laurea magistrale mi ha assunto nel suo laboratorio di ricerca e sviluppo, consentendomi di imparare e sperimentare molti aspetti sfidanti che hanno ampliato la mia conoscenza. Mi ha anche presentato ad Andrea Sabbioni, il quale ha lavorato con me nel laboratorio e con il quale ho condiviso molti momenti divertenti mentre stavamo lavorando. Andrea, inoltre, mi ha aiutato molto ad accrescere la mia conoscenza, con i suoi consigli e suggerimenti. Perciò, grazie, Filippo Bosi e Andrea Sabbioni!

Ora vorrei ringraziare tutti i miei amici, da quelli che conosco da sempre a quelli che ho conosciuto recentemente. Ognuno di voi mi ha aiutato durante questo periodo di studi, con voi ho condiviso lo stress per lo studio, l'ansia

prima di un esame e poi i festeggiamenti al termine di esso: grazie! Inoltre, vorrei davvero ringraziare tutti i miei amici provenienti dalla mia città natale, Federico, Giammarco, Giacomo e tutti gli altri, che sono sempre stati pronti a farmi rilassare e ridere. Vorrei davvero ringraziare anche Angelo e Luca dell'Università di Bologna, perchè con loro ho condiviso i bei momenti e quelli brutti durante questo periodo di studi. Infine, vorrei davvero ringraziare tutte quelle persone che ho conosciuto a Londra durante la preparazione della mia tesi, le quali mi hanno aiutato a superare un periodo lontano dai miei cari.

Un enorme grazie va alla mia famiglia: mio fratello Alberto e i miei genitori. Voi mi supportate ogni giorno, motivandomi ad essere la migliore versione di me stesso e aiutandomi a seguire i miei sogni. Non dimenticherò mai tutti i momenti che mi siete stati vicini e che mi avete aiutato. Grazie davvero!

Infine, un altro enorme grazie va alla mia fidanzata Diletta. Lei non è soltanto la mia compagna, ma anche la mia migliore amica, la mia confidente e il mio grande amore. Lei è sempre presente e quando ho bisogno di ogni tipo di aiuto o ho bisogno di un consiglio lei è sempre pronta ad aiutarmi. Lei mi supporta sempre in ogni decisione che prendo e mi aiuta sempre nel prendere quelle importanti. Lei mi sostiene ogni giorno e l'ha fatto anche nei 6 mesi in cui sono stato a Londra per la mia tesi. Apprezzo davvero tanto il suo aiuto. Insieme passiamo sempre momenti indimenticabili e siamo sempre in grado di trasformare un broncio in un sorriso. Quindi grazie, mio caro amore. Ti amo!