TÉCNICO LISBOA

Reinforcement Learning for Job Shop Scheduling Problems

Ricardo Miguel Alves Magalhães

Thesis to obtain the Master of Science Degree in

Mechanical Engineering

Supervisors: Prof. Susana Margarida da Silva Vieira
Eng. Miguel de Sousa Esteves Martins

Examination Committee

Chairperson: Prof. Duarte Pedro Mata de Oliveira Valério
Supervisor: Prof. Susana Margarida da Silva Vieira
Member of the Committee: Prof. Paulo Jorge Fernandes Carreira

January 2021

To my warm-hearted parents.

Acknowledgments

I would first like to thank my supervisors Miguel and Susana for believing in me and for always being there when I needed support, insights and counselling. I have always felt welcome to talk to them and that my ideas were appreciated and respected. Your guidance throughout these months definitely helped me develop both my project and as a graduate.

Secondly, I would like to thank professor João Sousa for his interest in my work and close support, making sure I have always had everything I needed for my work to flourish. His been attentive not only to my needs, but also to my potential and development, keeping me up to date with all the opportunities that he thinks would suit me.

I would also like to thank the IS4 team, who received me with open arms and made me always feel integrated. It was privilege to get in touch with this talented group of people and learn from their example.

I would like to express my gratitude also to Pedro Rodrigues and Tomás Pereira for their assistance. When I required it, they did not hesitate in dedicating some of their time and energy to the sole purpose of helping me. The same goes to Eng. Camilo Christo and Eng. Luís Raposeiro for their assistance in the lab.

My words of appreciation also to my housemate and friend Miguel Sousa who, even without any background in Artificial Intelligence, still shared with me hours of discussion about my work, helping me challenging my ideas and dissecting my problems. Moreover, he did it always with a strong belief in my talent and capabilities. I could feel it in every interaction we had and I know that he expects great things of me.

A special paragraph also to my roommate, colleague and friend Filipe Santos. For whatever reason, we got to know each other in our first day at Instituto Superior Técnico and our fellowship and companion still endures as we are finishing our MSc. No one challenged me more along the way than he did and I have no doubts that I am a much better scientist thanks to all our discussions and arguments. It was a pleasure to have them, especially when I was the one wrong, because during those times I could witness and absorb a bit of his genius. He gave many contributions and insights that greatly improved the quality of the work I produced for this thesis.

My gratitude to all the other people that crossed paths with me during my BSc and MSc journey that now comes to an end. I am a different person than I was 5 years ago thanks to all the experiences, challenges and knowledge that we shared.

A final word to my family, to whom I owe the vast majority of the qualities I have today. They gave me the time, love and guidance I needed to turning into who I am. That is a bond I will take with me my whole life. Without them I would not be standing here and all the things that I end up doing in the future will also come as a result of their support.

Resumo

Está a decorrer uma revolução na forma como as indústrias funcionam. A grande abundância de dados e o crescimento exponencial do poder computacional desencadearam o potencial da Inteligência Artificial. Esta nova ferramenta poderosa ganhou popularidade com o aparecimento da Indústria 4.0 e promete tornar as fábricas inteligentes, mais produtivas e eficientes.

O método proposto nesta tese aplica Deep Q-Learning para criar um sistema inteligente capaz de resolver o Dual Resource Constrained Flexible Job Shop Scheduling Problem. Uma nova arquitetura de rede neural Encoder-Decoder com Mecanismo de Atenção é apresentada. Esta arquitetura permite que a rede seja alimentada com informações de problemas mais completas e significativas, além de ter comandos de saída mais precisos e poderosos. Para diminuir o custo computacional desta técnica, os comandos de saída são selecionados como nas arquiteturas Pointer Networks.

O agente desenvolvido alcançou resultados promissores, mesmo tendo sido treinado limitadamente. Estes resultados mostram que, com uma implementação otimizada do método e um procedimento de treino mais completo, esta técnica tem o potencial de competir com os atuais algoritmos meta-heurísticos de última geração na solução deste tipo de problema.

Finalmente, o trabalho futuro a desenvolver pode consistir em colocar restrições adicionais ao problema para aplicação na indústria. A possibilidade de haver tempos de processamento diferentes para a mesma operação, dependendo do par de recursos alocado, é discutida. Além disso, um conceito redesenhado do método proposto é apresentado.

Palavras-chave: Job-Shop Scheduling, Aprendizagem por Reforço, Deep Q-Learning, Encoder-Decoder, Mecanismo de Atenção, Pointer Networks

Abstract

There is a revolution going on in the way industries work. The large abundance of data and the exponential growth of computational power have unchained Artificial Intelligence's potential. This new powerful tool gained popularity amidst the advent of Industry 4.0 and comes with the promise of making factories smart, more productive and efficient.

The method proposed in this thesis applies Deep Q-Learning to create an intelligent system capable of solving the Dual Resource Constrained Flexible Job Shop Scheduling Problem. A novel Encoder-Decoder neural network architecture with Attention Mechanism is presented. This architecture allows the network to be fed more thorough and meaningful problem information, as well as having more precise and powerful output commands. In order to diminish the computational cost of this technique, the output commands are selected as in Pointer Networks architectures.

The agent developed achieved promising results, even having just been scarcely trained. These results show that, with an optimized implementation of the method and a more thorough training procedure, this technique holds the potential of challenging current state-of-the-art meta-heuristic algorithms in solving this kind of problems.

Finally, future work might consist in adding additional constraints to the problem for application in the Industry. The possibility of having different processing times for the same operation, depending on the allocated resource pair, is discussed. Also, a redesigned concept of the proposed method is presented.

Keywords: Job-Shop Scheduling, Reinforcement Learning, Deep Q-Learning, Encoder-Decoder, Attention Mechanism, Pointer Networks

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

α	Learning rate.
δ	Error.
ε	Exploration factor.
γ	Discount factor.
ϕ	Activation function.
π	Policy.
σ	(Logistic) Sigmoid function.

Roman symbols

Avg	Average.
C	Cost Function.
C_{max}	Makespan.
E	Eligibility matrix.
J	Job.
K	Worker.
M	Machine.
N	Large number.
N	Memory.
O	Operation.
Q	Q-Value function.
R	Reward.
S	Set of possible environment states.

Std Standard deviation.

V Value function.

W Weight vector.

a Neuron activation.

act Action.

b Bias.

d Decoder hidden state.

e Encoder hidden state.

h Hidden state.

idx Index.

p Processing time.

r Reset gate.

rt Ready time.

s Current environment state.

st Starting time.

$targ$ Target.

U Weight vector.

w Weight.

x Input.

y Output.

z Update gate.

Subscripts

d Decoder step index.

f Neuron index in the current layer.

g Neuron index in the following layer.

h Number of the iteration in the RL cycle.

i Operation.

j Job.

k Worker.

m Machine.

n Final index.

t Time step.

v Vector components index.

x Input.

y Output.

z Final index.

Superscripts

π Policy.

d Decoder step index.

r Reset gate.

z Update gate.

H Total number of iterations in the RL cycle (episode length).

L Last layer.

l Layer.

T Transpose.

Chapter 1

Introduction

1.1 Motivation

1.1.1 Industry 4.0

We are living amidst a revolution in the way industries work and companies do business, as Industry 4.0 is on the rise and taking its first steps. Even though some might still dismiss it as a fancy marketing buzzword, there are huge transformations happening right now in the manufacturing processes of some of the world leading companies, that definitely make these new systems worthy of our attention and consideration [1].

Industry 4.0 is constituted by a set of production management frameworks which are powered by a network of intelligent machines, where data can be shared and processed, using technologies like Artificial Intelligence, Machine Learning and the Internet of Things. It is this interconnection and communication between computers, sensors and machines, that can give our factories the label of smart and ultimately allow them to be more efficient, productive and flexible, as well as less wasteful [1].

1.1.2 Artificial Intelligence

Artificial intelligence (AI) is a wide-ranging branch of computer science. This field's work and research is related with building smart machines capable of performing tasks that typically require human intelligence. AI is an interdisciplinary science with multiple approaches, but recent advancements in machine learning and deep learning are creating a paradigm shift in virtually every sector of the tech industry [2].

The large abundance of data and the exponential growth of computing power have unchained AI's potential and made its costs significantly decrease. The evidence seems to point out that we are on the verge of an immense upshift of the field's development and investment, and so a vast progress and fierce solutions are expected in the next few decades [3]. Areas where algorithmic problem-solving used to be found as extremely complex are now, with the introduction of AI, getting to consider these methods as feasible and economically viable [4]. The opportunities are immense and difficult to thoroughly grasp and so are the uncertainties about the changes it will create to our future. One thing is for sure: AI stands out as one of the most promising tools for companies to leverage their future. And perhaps to lose the trail of this transformation might cause irreversible damage to a

company competitiveness in the market.

1.2 Scheduling in Production Systems

Scheduling is the process of allocating scarce resources to perform a number of competing tasks over time [5]. It is a decision making process of great importance in the fields of manufacturing and production [6]. This important role derives from the need for survival that companies find in an extremely competitive economic scenario, where profit margins are getting smaller and smaller [5]. Optimized production schedules can give an enterprise a more efficient utilization of resources, which has a considerable impact on their production's capacity of meeting deadlines while maximizing profits.

Resources in manufacturing systems may include material storage, transportation and processing equipment, manpower and utilities (such as water, electricity and oil). Whereas, processing operations, transportation and maintenance may be considered as tasks [5]. In a production system the most critical resources typically are manpower and processing equipment, whereas the processing operations are the most critical tasks. Therefore, production scheduling problems are usually simplified according to manufacturing models. In these models, resources are typically boiled down to *machines* and tasks are known as *jobs*. Jobs may be composed by a single operation or by a collection of them to be done in several different machines, following a predefined sequence. One of the most well known manufacturing models is the Job Shop, because of its flexibility, room for customization and adaptability to change. The Job Shop optimization problem is presented in section 1.3.

Flexible multipurpose plants are able to produce a large variety of different products using a set of production routes. This feature makes such plants particularly effective for the manufacturing of products that exhibit a large degree of diversity and which are subject to fast-varying demands. Due to their inherent flexibility, the scheduling of such plants is a problem of high complexity. Compared to other parts of the supply chain management, such as distribution management and inventory control, the production scheduling is often by far the most computationally demanding part [5].

Optimization-based methods for scheduling were subject of a huge variety of research works published by the scientific community over the last 30 years. However, despite all this effort, the practical implementation of these methods in industrial real-life applications is still limited. Most optimization tasks of production scheduling are seen as extremely complex. Therefore, most schedulers end up using a simulation-based software or even making manual decisions, which result in suboptimal solutions [5].

A study as shown that the lack of quality in planning and scheduling can reduce productivity by 5 percent [4]. Quite often, inventories are increased to improve supply reliability and uphold stable production, which represents an increase in production costs. Excellent planning and scheduling is thus a significant competitive advantage.

1.3 Job Shop Scheduling Problem

The Job Shop Scheduling Problem (JSSP) stands out as one of the most important combinatorial optimization problems in manufacturing systems. It models a problem where one wishes to determine the ideal sequence of jobs on every machine, given that each operation can only be done in one specified machine. This means that each

job has its own predetermined route to follow. The JSSP has been widely studied for decades not only because of its significance and applicability in real manufacturing systems but also because it is a NP-hard problem [7]. NP stands for Non-Deterministic Polynomial problem, which means that the computation time is not a polynomial function of the size of the problem, but rather exponential or factorial. This means that as the size of the problem gets bigger (e.g. more jobs to schedule) it gets impossible for a simple algorithm to find optimal solutions in a limited amount of computing time.

Flexible Job Shop In real manufacturing systems the route that each job has to follow may not be predetermined. For example, there may be several machines capable of executing the exact same operation. That's where the Flexible Job Shop Scheduling problem (FJSSP) comes has an extension of the traditional JSSP. It introduces a new decision dimension, as it takes charge not only of the sequence of operations but also of their machine allocation (job routes) [8].

1.4 Dual-Resource Constrained Flexible Job Shop

In the FJSSP, the resources required for the machines to process their assigned tasks, like manpower, maintenance equipment and tooling, are still assumed to be plentiful. However, quite frequently some of these are in fact expensive or limited in abundance [9]. Usually, both machines and workers represent decisive capacity constraints. The Dual-Resource Constrained Flexible Job Shop Scheduling problem (DRC-FJSSP) comes as an attempt to better incorporate the real manufacturing process dynamics and constraints, which will hopefully generate more relevant and valuable solutions to the industry. It consists of 2 routing sub-problems: job scheduling and resource dispatch [10].

As the manufacturing industry reveals an increasing reliance on agility and flexibility, with the expansion of product customization, which lead to smaller lot sizes and shorter cycle times, the different kinds of machine and human resources must be considered and effectively managed to assure quick responses to the market demands. Previous works have been released addressing the relevance of the workforce specificities in the final solution, such as cross-training staffing levels, worker allocation, worker fatigue and recovery, learning and forgetting levels [10].

The DRC-FJSSP has been traditionally solved using analysis and simulation approaches. However, some details of the DRC-FJSSP are difficult to model for analysis and, being an NP-hard problem, simulations have an unbearable computational cost. Therefore, there has been a trend towards the use of meta-heuristic methods, since they have proved to be capable of quickly find near optimal solutions [10].

1.5 Thesis Contribution

The state-of-the-art collection of methodologies regarding the optimization of DRC-FJSSP is composed almost exclusively by heuristic or meta-heuristic algorithms that aim to outperform all the others at minimizing the makespan, which will be presented in section 2.1.4. However, no matter how good they can turn out to be (and there are many really good ones), they are hopelessly captive of a truly intelligent intuition. They can achieve a

high level of smartness, depending on the way these algorithms are designed, but there is always a high level of randomness on the optimization process. And randomness is not intelligence, intuition is [11]. The work described in this thesis was developed from the belief that a level of truly intelligent (not random) intuition on the optimization of DRC-FJSSP can be achieved using AI. And that this differentiating ability holds the potential of surpassing state-of-the-art results.

Additionally, there are other disadvantages that come with the use of meta-heuristics for combinatorial optimization problems. First of all, they are non-deterministic. This means that if you give them the same input twice, they will not most likely generate the same output. Because most meta-heuristics employ random choices, the computing time, as well as the solution quality, are actually uncertain. The stochastic nature of most meta-heuristics makes their rigorous analysis difficult. Given these considerations, researchers have been led to take into account empirical methods to measure the performance of a meta-heuristic and to compare meta-heuristics with each other. The approach is essentially based on standard statistical methods and the goal is to be able to ensure that the results are statistically significant.

Other than that, except for rescheduling tasks, which are outside of the scope of this work, every time there is a change in the problem at hand, like a new job arrives or there is a machine breakdown, the meta-heuristic needs to be ran again. In other words, any current good solutions are dismissed to consider the introduction of this new information. Consequently, even for small changes in the schedule, the algorithm will take as much time to find a new solution as it did when no good solution was known. Other algorithms can use knowledge from previously found solutions and quickly derive a new solution, with respect to the new problem information.

The solution proposed in this thesis was inspired in two different works that applied Q-Learning and Neural Networks (NN) to create an intelligent system capable of solving the DRC-FJSSP. In this thesis, the NN architechture used in these works was completely redesigned, as explained in section 3.2.1, in order to be able to feed the network thorough and meaningful problem information and allow it to perform more complex and powerful changes to the solutions. In the end, the proposed approach stands out as a deterministic method, problem independent and capable of giving fast and prolific results, if fully trained. The proposed method proves to be competitive with the current state-of-the-art scheduling meta-heuristics.

All in all, the work on this thesis was conducted with the following goals in mind:

- Use Deep Q-Learning to create an intelligent system capable of generating quality solutions to the DRC-FJSSP;

- Design a flexible agent, with direct applicability to the Industry and able to deal with complex real-world constraints;

- Verify the efficiency of the proposed method and compare it with other state-of-the-art approaches;

- Introduce new performance metrics to evaluate the quality of the DRC-FJSSP solution;

- Generate solutions in a reasonable amount of computational time, so that this method can be used in the Industry;

- Extend the limits of the application of Deep Learning to the DRC-FJSSP and pave the way for future research.

1.6 Thesis Outline

This thesis consists of six chapters. In the first one, Introduction, the motivation behind the research on this subject is approached, as well as a reflection on why is there a window of opportunity. An introduction to scheduling in production systems and to the DRC-FJSSP model is given. In the second chapter, Background, a small overview of commonly used Scheduling Algorithms is given and some state-of-the-art metaheuristics are highlighted. It is followed by a presentation of some fundamental Machine Learning concepts for this work, such as Neural Networks, Reinforcement Learning and the Encoder-Decoder architecture with Attention Mechanism. Chapter 3, Implementation, starts with a numerical model of the DRC-FJSSP. Then, the proposed solution is explained, and so are the steps that led to its development, regarding the usage of Deep Q-Learning and the implemented RL algorithm. At last, two novel performance metrics for evaluating the solutions of a DRC-FJSSP are defined. In the forth chapter a first quickly trained Agent is tested against some very basic heuristic algorithms. The fifth chapter has detailed results and discussion of the application of the proposed method to a Benchmark dataset. The obtained solutions are compared with the ones generated by a state-of-the-art metaheuristic. Chapter 6, Conclusions, contains a summary of the overall quality of the solutions obtained and a thorough conceptual development of future work to be done based on this thesis. In Appendix A, the ideal number number of move operations per episode during training is deducted. And finally, in Appendix B, the full results of the simulations in Chapter 4 are presented.

Chapter 2

Background

2.1 Scheduling Algorithms

2.1.1 Exact Algorithms

Exact algorithms provide solutions that are always guaranteed to be optimal. Furthermore, for a finite size instance of a combinatorial optimization problem that solution can be found in finite time. Nevertheless, for NP-Hard problems, like the JSSP, there are no exact algorithms capable of solving these problems in polynomial time. This means that the computational time of these methods grows exponentially with the problem dimension and quickly becomes unbearable [12]. Branch and bound algorithms [13] and mixed integer programming [14] are the most popular exact methods used to solve scheduling problems.

2.1.2 Heuristics

A heuristic method is a practical problem solving method whose solution is not guaranteed to be optimal. For problems where finding an optimal solution is impossible or impractical, heuristic methods offer a quick way to find a satisfactory solution. Some of the most simple heuristics used in production scheduling include the Earliest Due Date, Longest Processing Time and the Shifting Bottleneck heuristics [6]. Examples of other heuristics used to solve the DRC-FJSSP are given in [15].

Heuristics are problem-dependent techniques, which means they are usually adapted to the problem at hand, taking full advantage of the problem specificities and expert knowledge. Also, they are often too greedy and get frequently trapped in local optima, unable to find the global optimum solution [16].

2.1.3 Meta-heuristics

Meta-heuristics serve quite the same purpose as heuristics but they are problem independent, even though some fine-tuning of its parameters is often needed in order to achieve good results for the problem at hand. Generally, in order for them to explore the solution space more thoroughly, there is a temporary deterioration of the solution. This way, they can eventually find a better solution by exploring beyond local optima [16].

Today, a wide variety of optimization problems are solved using meta-heuristics. They are a powerful tool

that makes it possible to solve NP-hard problems by taking a black box approach. In other words, by using these techniques a good enough solution can be quickly reached for a problem one may not know much about how to solve.

Popular meta-heuristics used in production scheduling include genetic/evolutionary algorithms, simulated annealing, variable neighborhood search and particle swarm optimization, although many more exist [15]. There are even some hybrid techniques which get together some of best features of these individual meta-heuristics into a novel and more proficient method.

2.1.4 State-of-the-art Meta-heuristics

In the research conducted for this thesis a set of meta-heuristics were found to constitute the backbone of state-of-the-art techniques to solve the DRC-FJSSP. The biggest challenge for a meta-heuristic is always to moderate its greediness, so that it does not get stuck in a local optima, while keeping a top performing speed of convergence. The search for the optimal point of the solution space can be divided in two different phases: 1) the identification of regions of interest (global exploration) and 2) finding the local optima (local exploitation). All in all, the goal of these meta-heuristics is to reach an optimal balance between these two. The following list is organized by number of citations in ascending order.

Hybrid Genetic Algorithm A Hybrid Genetic Algorithm was successfully used to minimize the DRC-FJSSP makespan, with consideration of worker's learning ability [17]. Because of the learning ability, the worker's processing efficiency will increase with the accumulation of processing time but it cannot be improved indefinitely, it will stop at a given highest level. A genetic algorithm was utilized to run the global exploration and it was integrated with a variable neighbourhood search, responsible for the local exploitation. During the first, greedy selection operations are executed together with two different types of permutation-based search operators. During the later, three other types of permutation-based search operators are used.

Hybrid Discrete Particle Swarm Optimization A Hybrid Discrete Particle Swarm Optimization provided good results in minimizing the makespan for the DRC-FJSSP [18]. A Discrete Particle Swarm optimization was used to conduct the global exploration. An improved Simulated Annealing with a Variable Neighbourhood structure was used for the local exploration. Like specified before, Particle Swarm Optimization, Simulated Annealing and Variable Neighbourhood Search are meta-heuristics themselves. Thus, we call this algorithm a hybrid meta-heuristic, since it combines different components from various meta-heuristics [19].

Branch Population Genetic Algorithm A Branch Population Genetic Algorithm was used to try to minimize the makespan and the cost of the DRC-FJSSP [10]. The novelty here is in introducing a branch population to accumulate and transfer evolutionary experience, strengthening the population diversity and accelerating convergence. A permutation-based search operator is used to improve the thoroughness of global exploration and a greedy selection of the best solutions is used for local exploitation.

Shuffled Multi-Swarm Micro-Migrating Birds Optimizer A Shuffled Multi-Swarm Micro-Migrating Birds Optimizer tries to minimize the makespan of a Multi-Resource Constrained Flexible Job Shop Scheduling Problem

[9], similar to the DRC-FJSSP but with more than two resource constraints. The algorithm forms a number of micro-swarms, each of which performs its own Migrating Birds Optimizer independently. It has been found that a genetic algorithm with a small population of three individuals is sufficient to converge, irrespective of having different lengths of chromosomes involved [20]. A random shuffle process is periodically applied to propagate the knowledge acquired by the micro-swarms. Also, there is a renewing process of the population, based on the aging phenomenon of life, that promotes the diversity of the population. Two different types of permutation-based search operators are used that try to efficiently balance global exploration and local exploitation.

Knowledge Guided Fruit Fly Optimization Algorithm A Knowledge Guided Fruit Fly Optimization Algorithm (KGFOA) proved to be effective in solving the DRC-FJSSP [21]. Two types of permutation-based search operators were used for exploration and optimized by being combined with a knowledge-guided search stage. In summary, this knowledge-guided search utilizes the experience provided by the best solutions found so far to enhance the probabilities of choosing of operations and resource assignments that have proved to work in the past. Finally, a greedy selection of the best solutions takes place. This whole procedure is applied with the goal of minimizing the solution makespan. Since this is the method with the most recognition from this set, it was the algorithm chosen for comparison with the solution proposed in this thesis.

2.2 Machine Learning

Machine Learning (ML) is the field of science that studies how to make computers learn from data. There is, however, a more precise definition which may prove to be useful to improve our understanding when working on engineering projects. According to Tom Mitchell, a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E [22].

There are so many different ML techniques that it can be useful to divide them into different categories, according to a specific criterion. The most common division considers 3 main branches, depending on whether or not they are trained with human supervision: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

In Supervised Learning (SL), the training data you feed to the algorithm includes the desired solutions, called labels. It is the task of learning from labeled data and a human is needed to decide which data to collect and how to label it. A program can then be created to identify bananas in an image (classification) or to estimate car prices given their brand, model, age and miles (regression). The idea is to generate a model that can look at the given data and learn to generalize for unseen examples.

In Unsupervised Learning (UL), the goal is to learn from unlabeled data. The program can learn without a teacher how to divide the data into meaningful clusters or apply dimensionality reduction.

Reinforcement Learning (RL) constitutes a whole different ML technique. In RL, a software agent makes observations of an environment and takes actions within it. In return, it receives rewards and its objective is to learn to act in a way that will maximize its expected long-term rewards [23]. Many impressive applications have been created implementing RL algorithms. RL is part of this thesis backbone, so a more thorough explanation of

this technique will be provided in section 2.2.3.

2.2.1 Neural Networks

Just like birds inspired us to fly and the velcro technology idea came from the burdock plants, nature has inspired many other human inventions. From here it is not a big stretch to understand that the human brain's architecture can provide tremendous insights on how to build an intelligent system. And that is where the idea that inspired artificial neural networks (NN) came from [23]. Before getting in touch with a simple NN architecture, one should first have an elementary understanding of its basic inner structure.

2.2.1.1 Perceptron

The perceptron is the most simple unit in a neural network, just like a neuron in the human brain. Understanding what a perceptron is and how it works helps to undercover what is going on inside a neural network. It consists of 4 parts: the input layer, the weights and biases, the net sum and the activation function. A representation of a perceptron is given in figure 2.1.

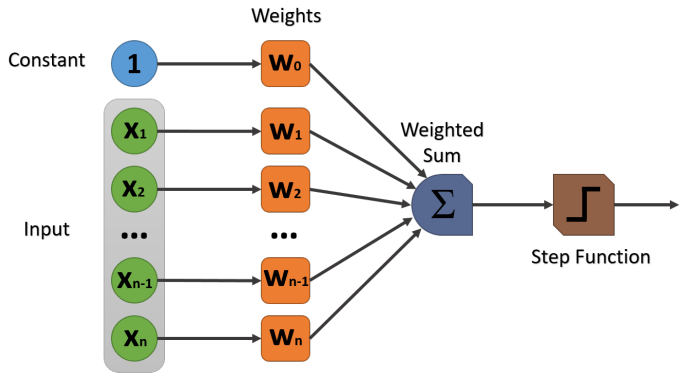

Figure 2.1: A representation of a perceptron.

The weight w_0 in figure 2.1 represents the bias. The weights and the bias are used to calculate the input weighted sum, which is then passed through an activation function, the step function, giving a binary number as the output. Equation 2.1 rules the output of a perceptron. It is worth noticing that the weights and the bias have different roles. The weights show the strength of a particular node, whereas the bias shifts the activation function up and down [24].

$$output = \begin{cases} 0 & if \sum_v w_v x_v \leq threshold \\ 1 & if \sum_j w_v x_v > threshold \end{cases} \tag{2.1}$$

$\sum_v w_v x_v \leq threshold$ can be simplified to $w_v \cdot x_v + b \leq 0$, with $b \equiv -threshold$ being the bias.

Perceptrons can work as Linear Binary Classifiers, dividing the data into two parts. Additionally, they can be used to compute elementary logical functions, such as AND, OR and NAND. And since these functions are so simple, a set of weights and biases can be easily devised without the need for any sort of training and exploration. But suppose there is a need to compute a more complex function, one that can no longer be trivially derived. In that case, training is needed. However, as we will see in section 2.2.1.4, it is impossible to train a perceptron with

a binary output. In order to train the perceptron, it would be needed that small changes in the net's weights and biases would provoke a small change in the output. For a binary output that cannot happen, since only two values can be expelled from the NN. This situation originated a stalemate that lasted for decades in the field, when at last the rediscovery of the backpropagation algorithm reignited the interest in studying NN [25].

2.2.1.2 Activation functions

An activation function maps the result of $w_v \cdot x_v + b$ to a number between 0 and 1. To be fair, the binary output of the original perceptrons was itself an activation function. A step function, more specifically. The problem was that that specific branch function was not continuous, and so it was not differentiable. The activation function differentiability is required so that the backpropagation algorithm can be applied for training, as explained in section 2.2.1.4.

Different activation functions are used that make it possible to train a NN. The most popular one is by any means the (logistic) sigmoid function, which is defined by equation 2.2 and whose shape can be seen in figure 2.2.

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \tag{2.2}$$

where $z = w_v \cdot x_v + b$.

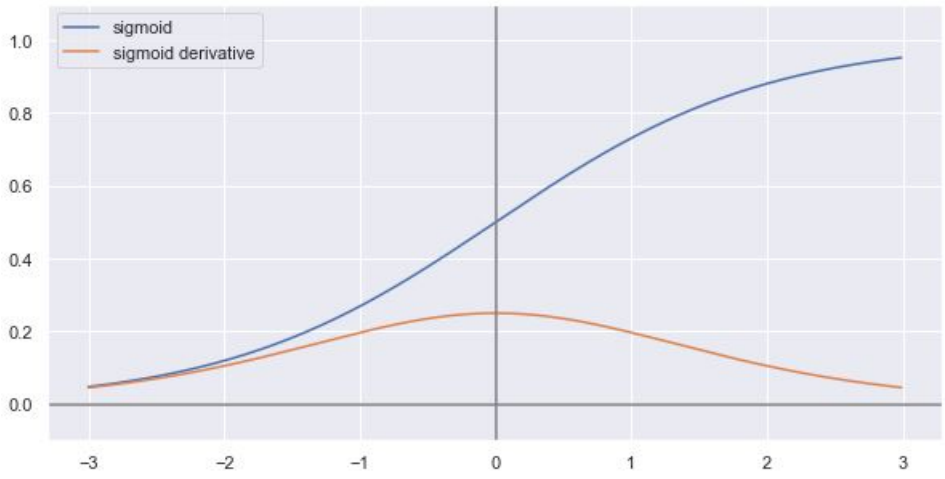

Figure 2.2: Sigmoid function and its derivative.

The sigmoid function possesses all the good mathematical properties of being fully differentiable and continuous, keeping the output values between 0 and 1.

Another popular activation function is the tanh, which is defined by equation 2.3 and whose shape can be seen in figure 2.3. Actually, evidence points out that tanh activation functions are superior to sigmoid functions when it comes to train a NN [26]. When having sigmoid as the activation function of the previous layer, since the input values are all positive or null, it is not possible for some weights of a node to increase while some others decrease. In other words, the weights of a node can either all increase or all decrease in a single step of the gradient descent. If the weight vector needs to change direction, it can only do so by adding and removing different amounts to the weights until the change in direction is completed. This is highly inefficient. For tanh, on the other hand, the

direction of the weight updates are independent of one another, since the input values can now also be negative. This allows the weight vector to change direction more easily.

$$tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.3}$$

where $z = w_v \cdot x_v + b$.

Figure 2.3: tanh function and its derivative.

The tanh function also possesses all the good mathematical properties of being fully differentiable and continuous, but keeps the output values between -1 and 1. Additionally, the tanh derivative is tendentiously much higher than the sigmoid derivative, which will accelerate training when applying the backpropagation.

Like it was previously pointed out, the perceptron (with an activation function) is the basic unit of a NN. Having already been introduced to both of them, one is ready to observe and comprehend a NN basic architecture.

2.2.1.3 Basic Neural Network architecture

An example of a simple NN is presented in figure 2.4:

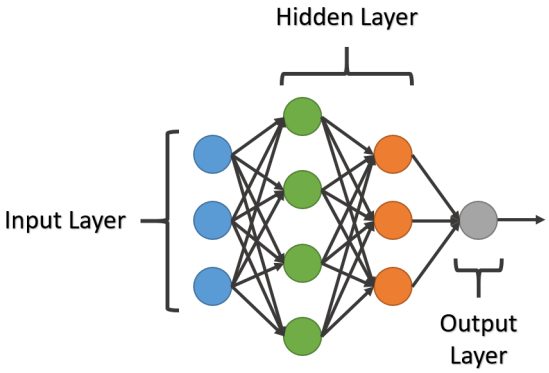

Figure 2.4: NN with an input layer, an output layer and 2 hidden layers.

Each circle in the image represents a neuron: a perceptron with an activation function. The left side column is called the input layer and it is made of the input neurons. This is the layer that receives the input data and first

processes it before propagating it to the following layers. The right side column is called the output layer and its neurons are called the output neurons, whose outputs constitute the output of the NN. The middle layers are called the hidden layers. The inputs of each neuron in an hidden layer are the outputs from the previous layer and the outputs of an hidden layer are the inputs of the following layer's neurons. Having understood all the basic components of a NN and their interconnections, one can finally understand how it is trained.

2.2.1.4 Backpropagation

Backpropagation is the method by which the fine-tuning of the weights and biases of a NN is done in order to reduce the error rate obtained at each learning step. In other words, it is how the model learns. The backpropagation method for updating a NN weights and biases has 2 different stages: the forward propagation and the backward propagation.

During the forward propagation phase, the inputs of the training samples are fed into the NN and this information is then processed and propagated throughout the network until it generates an output. A loss function needs to be defined, so that the error between the output and the training sample target output can be computed. A commonly used loss function is the sum squared error (used throughout the rest of this explanation).

Next, the second stage begins. During the backward propagation, the gradient descent weight update rule is used [27]. In other words, the weights and biases are adjusted in the most effective way such that the activation of the neurons throughout the layers will produce the desired output. In order to do so, the goal is to compute the partial derivatives $\frac{\delta C}{\delta w^l_{fg}}$ and $\frac{\delta C}{\delta b^l_f}$. And for that there is a need to define δ^l_f as the error in the f-th neuron in the l-th layer. The backpropagation method will compute the error δ^l_f and then define $\frac{\delta C}{\delta w^l_{fg}}$ and $\frac{\delta C}{\delta b^l_f}$ as functions of it.

If there is a sigmoid as the activation function, the activation of a neuron is computed by equation 2.2. This means that the greater the values of w^l_{fg}, z^l_f and b^l_f the more activated is the neuron. It is worth highlighting that z^l_f represents the output of the previous network layer (the input of the NN, if the neuron is in the input layer) and that only w^l_{fg} and b^l_f are adjustable. z^l_f can also be adjusted, but only by changing the weights and biases of the previous layers.

Let a^l_f be the output of the activation function of the f-th neuron in the l-th layer. The components of the error vector in the output layer δ^L, where L represents the last layer l, the output layer, can be computed according to equation 2.4 [28].

$$\delta^l_f = \frac{\delta C}{\delta a^L_f}\sigma'(z^L_f) \tag{2.4}$$

where $\frac{\delta C}{\delta a^L_f}$ calculates how much the cost function changes as a function of the f-th output activation and $\sigma'(z^L_f)$ measures how much the activation function σ is changing with z^L_f. For example, $\frac{\delta C}{\delta a^L_f}$ shows that if C does not depend a lot on a particular output neuron, f, then δ^L will be small. Next, there is a need to calculate the error δ^l in terms of the error in the next layer, δ^{l+1}, according to equation 2.5.

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \tag{2.5}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the (l+1)-th layer.

When $(w^{l+1})^T$ is multiplied by the error δ^{l+1} at the (l+1)-th layer, the error is kind of moving backwards in

the network, which creates an estimate of the error at the l-th layer. Then, when the element-wise multiplication of this error with the derivative of the activation function is applied, the error is once more being moved backwards through the activation function in layer l, computing an estimation of the error δ^l in the weighted input to layer l. By combining 2.4 and 2.5 it is now possible to compute the error δ^l for any layer in the network. Now, only the need to relate it with the weights and biases of the network is left. Starting with the biases, it is possible to compute the rate of change of the cost with respect to any bias in the network, according to equation 2.6.

$$\frac{\delta C}{\delta b_f^l} = \delta_f^l \tag{2.6}$$

Finally, equation 2.7 can be derived for the rate of change of the cost with respect to any weight in the network.

$$\frac{\delta C}{\delta w_{fg}^l} = a_g^{l-1} \delta_f^l \tag{2.7}$$

All in all, the gradient descent method finds out how much these parameters earlier in the network influence the output neurons activation. It does that by calculating the composite partial derivatives for each these variables. In the end, the sum of this contributions shows how much each of the weights and biases influence the output neurons activation and how they should be adjusted.

2.2.2 Deep Learning

Deep learning methods usually consist in using a neural network architecture with a great number of hidden layers, quite commonly dozens of them, where normally would be 2 or 3 in a more traditional and simple architecture. This complexity can give models the ability of learning features directly from the data, without the need for manual feature extraction [29].

This additional complexity comes at the cost that deep learning methods need large sets of data to train the model. For example, at least a few thousand images might be needed to get reliable results in a classification task. Moreover, training a deep NN usually takes a lot of time. Having an high-performance GPU is of critical importance in order to obtain results from deep learning methods in a reasonable amount of time. All in all, there are some important constraints to training, such as the availability of the data and computational power to analyse and process it.

Benefiting from the arise of big data and the development of more powerful computation, new algorithmic techniques, mature software packages and strong financial support some outstanding results were achieved [30]. Some of them are being carried in peoples' pockets, like Google Translate and Apple's bot Siri. Other examples are given in section 2.2.3.2.

2.2.3 Reinforcement Learning

Humans learn through trial and error. Every decision and action taken makes a person experience either pain and failure or reward and success. Throughout the learning process people tend to find a way to minimize the first two and maximize the last. This should be enough to gain a proper insight on how RL works. In RL an agent (the machine) gets to learn how to execute a specific task through an interaction cycle with the environment.

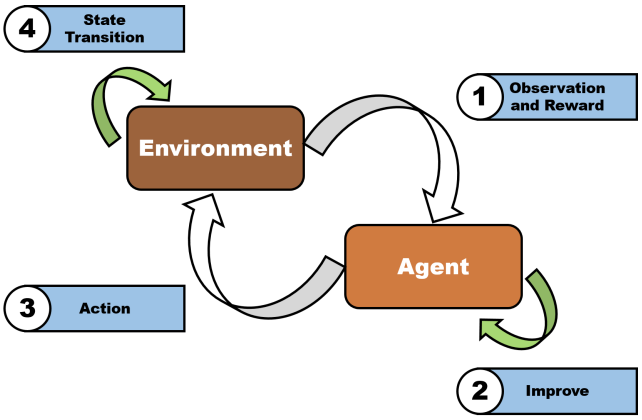

Figure 2.5: A Reinforcement Learning cycle.

As Morales did [3], it is useful to think about how one would train a dog to sit. The dog likes treats and would like to have more of them, so a person can give them as rewards for every time it sits. But the dog does not know that (at least yet), so it tries to interact with you through many different actions. It can run, bark, sit, play dead or even try a combination of them. How can it find out which action or set of actions led to the treat? He received a treat a minute after he ran, so maybe running was the action leading to the reward. Or was it the fact that he barked and sat down after it? In order to figure it out, the dog needs to assign credit to every action that might have possibly led to the reward. But not only that. What if there is an action or set of actions that makes the person give it 10 treats instead of 1? Maybe it is worth exploring different interactions to find that out. This means that, in order to learn how to maximize its rewards, the dog needs to balance exploring new actions and retrying the actions that have worked best so far. Otherwise, it could spend the rest of its life getting only a tenth of the treats it could have.

In this example, the dog was the agent and the person was the environment. Like it was pointed out in figure 2.5, the cycle begins with the agent observing the environment (step 1). Then, the agent does some internal processing of the observation and reward, analyses it and attempts to learn something from it, trying to improve at the task (step 2). After that, the agent takes an action (step 3). This interaction with the environment promotes some changes in its internal state, as a consequence of the previous state and the agent's action (step 4). This may cause the environment to externally react in a way that the agent can see. However, some of those internal changes may only manifest further away in time, so the external reaction that they promote can be delayed. The whole cycle - observation, processing, action, responses from the environment - then repeats.

It is relevant to highlight that RL is concerned with sequential decision-making. This means that the decisions you are making may impact all possible feedback signals you are going to get in the future, not only the immediate one. It is not effective to simply choose the action with the highest immediate reward for the current system state because at each decision the environment is being changed. Thus, one should also take into account the changes to the environment that the action will bring, in order to maximize all future rewards.

Additionally, it is important to grasp the complexity of Evaluative feedback. Unlike Supervised Learning, whose feedback is plain and simple (you either classify an image correctly or incorrectly), in RL it can be quite hard and complex to classify a feedback as good or bad. Going back to the earlier example, the dog may have felt great when it understood that sitting down was the action that led to the reward. It does not look that difficult to

get that 1 treat anymore. But, like it was said before, what if there was a way to get 10 treats at once? Does that accomplishment still look great? Probably not. This is what makes evaluative feedback so complex. One can only make comparisons once data was gathered first. But one shall keep in mind that there could always be some other data that has not been seen yet, which would change the meaning of the current data.

2.2.3.1 Important Reinforcement Learning Concepts

There are some important RL concepts worth highlighting and introducing. They are the policy, the discounted rewards, the value function and the Q-function.

The goal of an agent is to pick the set of actions that will allow it to maximize the total rewards received from the environment [31]. The total reward received by the agent in response to the selected actions is given by equation 2.8.

$$TotalReward = \sum_{h=1}^{H} R_h \tag{2.8}$$

where h represents the number of the iteration of the RL cycle.

However, it is common to use a small discount factor γ to give higher weights to rewards closer in time than to rewards received further in the future. This way, the total discounted reward is defined and can be calculate using equation 2.9.

$$TotalDiscountedReward = \sum_{h=1}^{H} \gamma^{h-1} R_h \tag{2.9}$$

where H is the episode length and $0 \leq \gamma \leq 1$. The reason for using the discount factor is to prevent the total reward from going to infinity. Moreover, it models the agent behavior. It dictates if the agent prefers immediate rewards over rewards that are potentially received far away in the future, or the other way around.

The value function $V(s)$ represents how good is a state for an agent to be in. It is equal to the expected total discounted reward for an agent starting from state s. The value function depends on the policy by which the agent picks actions to perform. So, if the agent uses a given policy π to select actions, the corresponding value function is given by equation 2.10. The policy π is a function that takes the current environment state s as an input and returns an action.

$$V^{\pi}(s) = E[\sum_{h=1}^{H} \gamma^{h-1} R_h] \; \forall s \in S \tag{2.10}$$

Among all possible value-functions, there exist an optimal value function that has higher value than other functions for all states, which is given by equation 2.12.

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \; \forall s \in S \tag{2.11}$$

The optimal policy π^{*} is the policy that corresponds to optimal value function.

$$\pi^{*} = arg \max_{\pi} V^{\pi}(s) \; \forall s \in S \tag{2.12}$$

2.2.3.2 Deep Reinforcement Learning

A method is considered to be Deep Reinforcement Learning every time a deep neural network is used to approximate any of the following components of RL: value function $V(s,a;w)$, policy $\pi(a|s;w)$ and model (state transition function and reward function). Here, the parameter a is the action, s is the state and w are the weights of the deep neural network [30]. Some remarkable results using Deep RL were achieved, especially in games (Alpha GO [32] and Atari games [33]) and robotics [34].

2.2.3.3 Q-Learning

The Q-learning algorithm approximates the policy function $\pi(a|s;w)$, by using a table (the Q-table) which maps all environment states to all agent actions. Each element of the table $Q_{(s,a)}$ is the score value of choosing action a during the environment state s. The agent chooses the action with the highest score value for the current state.

The table values can be randomly initialized and are updated according to the Bellman's equation:

$$Q^{new}(s_t,a_t) \equiv (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{R_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}\right)}^{\text{learned value}} \tag{2.13}$$

There are two parameters in the Bellman equation worth highlighting. The first of them is the *learning rate*, α, which takes control of how fast the Q-table values are changed [35]. Lower learning rates lead to a slower but more stable learning convergence. The α value can be varied throughout the training. It is common to start training with a higher learning rate, in order to allow fast initial changes, and then progressively lower it.

The second is the *discount factor*, γ, already presented in section 2.2.3.1. It is responsible for stating how much the agent should care about future rewards. It is a real value between 0 and 1, where $\gamma = 0$ means that it only cares about the current reward, whereas $\gamma = 1$ means that it takes into consideration all future rewards equally.

A final aspect needs to be taken into account when applying algorithm 1. As previously mentioned, we can select the action to take using the Q-table, but during training it is useful to introduce an exploration factor, ε, with values between 0 and 1. It represents the probability of choosing a random action, rather than using the Q-table. The ε value decays over training, so that the agent progressively behaves more autonomously. However, the Q-table is always updated, no matter how the action was chosen.

Algorithm 1: Q-Learning Algorithm

```
Result: Trained Q-table
Initialize Q-table;
while epoch ≤ n_epochs do
    Pick an action;
    Evaluate the action;
    Measure the reward;
    Update Q-table;
end
```

Q-Learning works well for a simple environment, but when the number of actions and states gets computationally too high, it is possible to use neural networks to approximate the Q-value function. That is called Deep Q-Learning.

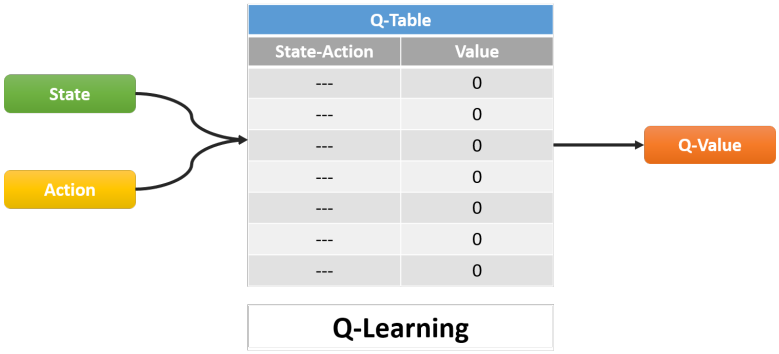

Figure 2.6: Q-value gathering in Q-Learning.

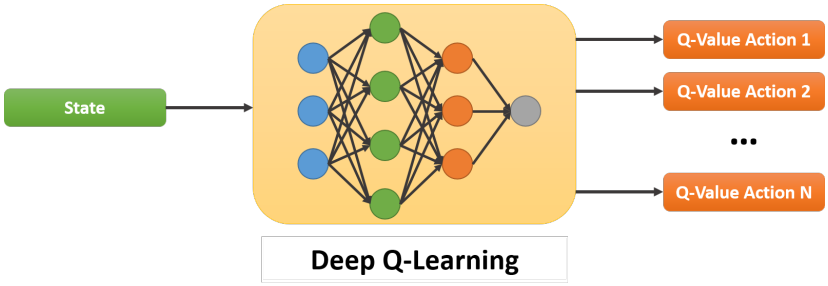

Figure 2.7: Q-value gathering in Deep Q-Learning.

2.2.4 Recurrent Neural Networks

Unlike the classical NN, Recurrent Neural Networks (RNN) receive not only the input x_t at the input layer, but also its output from the previous time step y_{t-1}. They are usually used for applications dealing with sequences of data. To perceive how they work, one can either picture a NN feeding itself backwards (from the output to the input) or can unroll the network through time and take a look at the interconnections between different time steps [23].

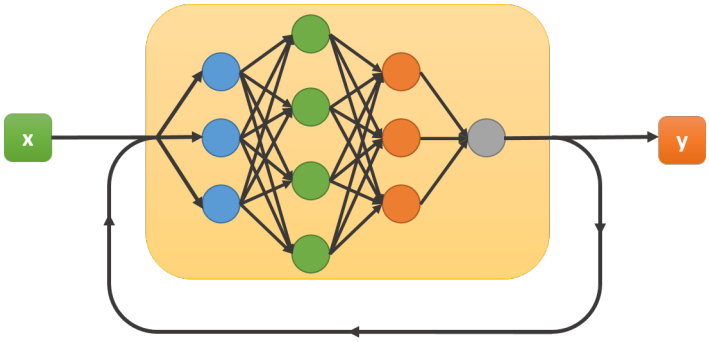

Figure 2.8: A representation of a RNN.

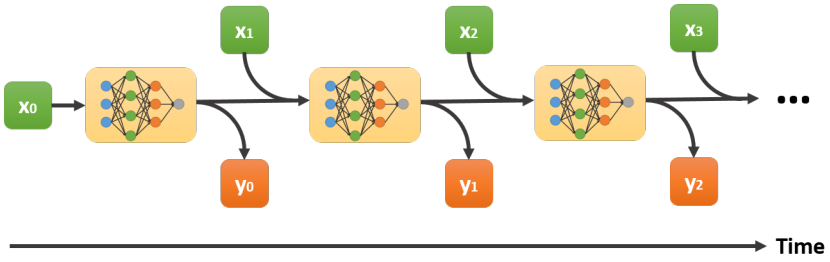

Figure 2.9: A representation of a RNN unrolled through time.

Each recurrent neuron has one set of weights w_x for the inputs x_t and another set w_y for the outputs y_{t-1} of the previous time step. Consider a RNN composed by one single recurrent neuron. The output of the recurrent neuron is given in equation 2.14.

$$y_t = \phi(x_t^T \cdot w_x + y_{t-1}^T \cdot w_y + b), \tag{2.14a}$$

$$y_t = \phi([x_t^T \;\; y_{t-1}^T] \cdot W + b), \quad with\, W = \begin{bmatrix} w_x \\ w_y \end{bmatrix}, \tag{2.14b}$$

2.2.4.1 Memory Cells

A recurrent neuron has a form of memory, since its output at time t is a function of all inputs from previous time steps. A memory cell is a part of a neural network that preserves some state across time steps. Once again, a single recurrent neuron is an example of a very basic cell.

The state of the memory cell at time step t is called the hidden state h_t. h_t is a function of the input x_t and of the hidden state of the previous time step h_{t-1}. Its output y_t is a function of the previous hidden state h_{t-1} and the current input x_t. These interconnections are more clearly visible in figures 2.10 and 2.11.

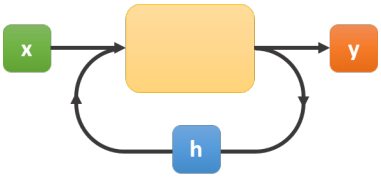

Figure 2.10: A representation of a memory cell.

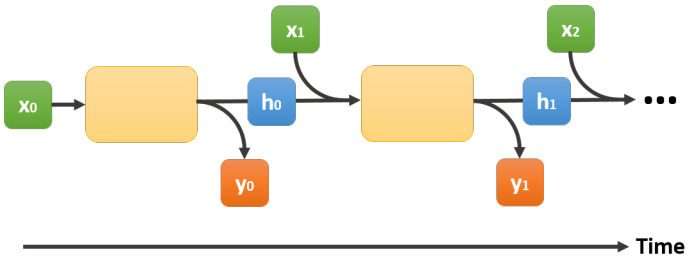

Figure 2.11: A representation of a memory cell unrolled through time.

2.2.4.2 The Vanishing Gradient Problem

The Vanishing Gradient Problem is a problem caused by certain activation functions when using gradient based methods (such as the backpropagation) for training a NN. It makes it really hard to learn and tune the parameters of the earlier layers in the network [36].

Some activation functions (like the sigmoid and tanh) squash the output values into a range $\in [0,1]$. The function is extremely flat close to the range limits, therefore there are large regions of the input space that are mapped into a tiny range in the output. This means that gradients are very small, that even large changes in the input values only produce small changes in the output [36].

This problem definitely worsens when we concatenate multiple layers of this functions, since the first layer would squash its input values into a much smaller output range, which would then be mapped to a much smaller region by the second and so on. In the end, huge changes in the parameters of the first layer would not end up in a relevant change in the output [36].

In practice, this means that the network has a hard time memorising inputs from far away in the sequence.

RNN's can solve a variety of problems, which include speech recognition, language modeling, translation and image captioning. As a small example, you can teach a RNN to predict the last word in a sentence. One can sequentially give it the words in the sentence "My mother's daughter is my..." and expect it to output the word "sister". This works really well for small sentences.

However, there are cases for which more context is needed. Consider the text "I've studied in Spain for 6 months (...) I can now speak a bit of Spanish". Although the most recent inputs suggest there is a name of a language coming, it is impossible to figure out which one is it without taking into consideration the information at the beginning of the text [37]. RNN's might struggle to connect these dots when they become too far apart from each other. In other words, there is an observable degradation in the long-term memory capabilities of these networks.

To avoid this problem, many people try to choose a different activation function that does not squash the input values. One popular example is the Rectified Linear Unit, which maps x to $max(0,x)$ [36].

Moreover, there are some RNN architectures which can tackle this problem and achieve great long-term memory results, with LSTM's and GRU's being the two most popular ones. In this thesis, the RNN architecture chosen was the GRU. Therefore, a more thorough explanation of this network is given in section 2.2.4.3.

2.2.4.3 GRU

The Gated Recurrent Unit aims to solve the vanishing gradient problem. It is an improved version of a standard recurrent neural network. The incorporation of two different features (the update gate and the reset gate) allow the agent to be trained to keep information from long ago. These gates are two vectors that decide which information should be passed to the output [38].

In figure 2.12, there is an overlook of how a GRU works. A more detailed and step by step explanation follows.

The update gate shown in figure 2.13 is responsible for determining how much of the past information should be propagated in the future. This solves the vanishing gradient problem, since it allows the agent to propagate all information from the past if it wants to.

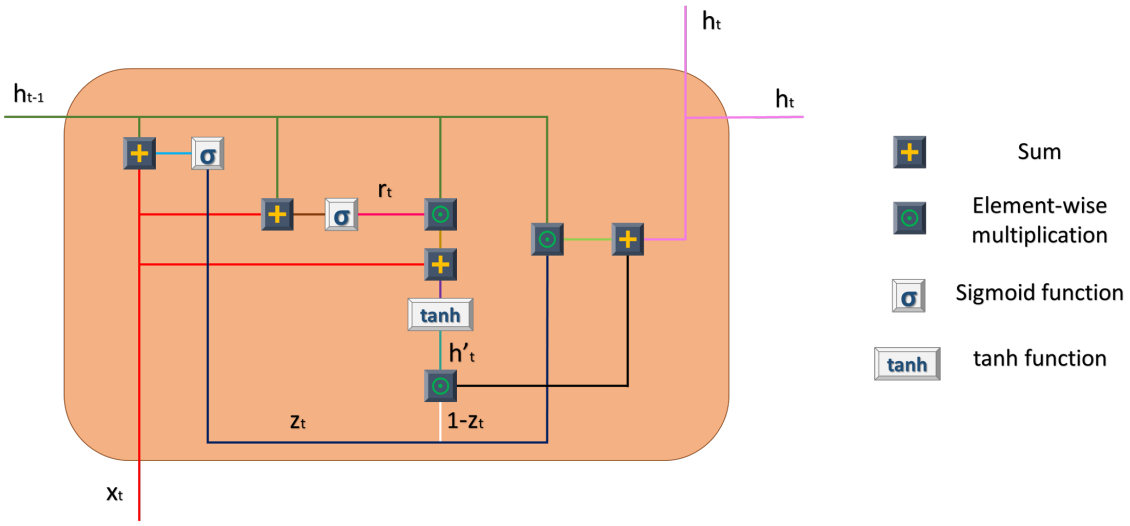

Figure 2.12: A GRU representation.

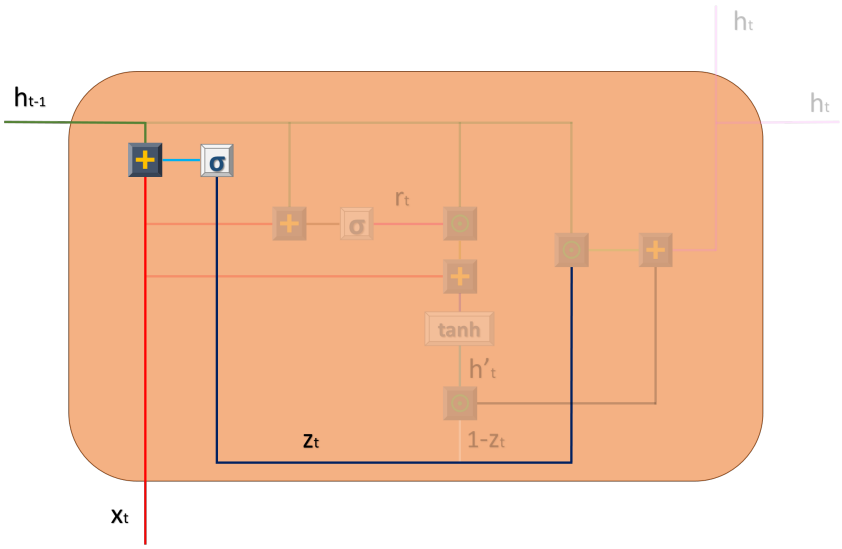

Figure 2.13: The update gate of a GRU.

The update gate z_t for time step t is calculated using equation 2.15.

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{2.15}$$

When x_t enters the GRU, it is multiplied by its own weight W(z). Also, h_{t-1}, which holds the information for the previous t-1 units, is multiplied by its own weight U(z) when it enters the network. Both results are added together and a sigmoid activation function is applied to squash the result between 0 and 1.

The reset gate illustrated in figure 2.14 serves the purpose of allowing the model to decide how much of the past information should be forgotten. The reset gate r_t for time step t is calculated using equation 2.16.

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{2.16}$$

This formula is similar to the one for the update gate, but the weights are different. Then, the reset gate output

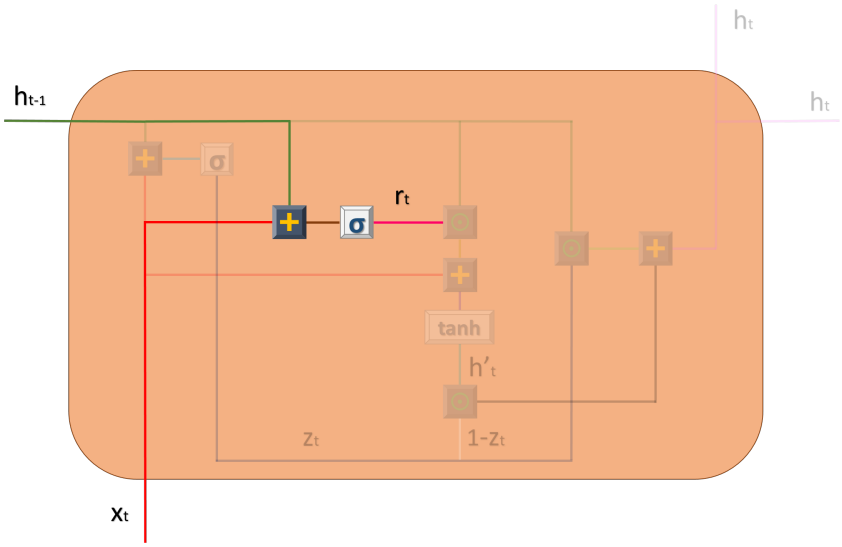

Figure 2.14: The reset gate of a GRU.

is used to select the relevant information from the past, which should be stored in the current memory $h_t^{'}$, according to equation 2.17.

$$h_t^{'} = tanh(Wx_t + r_t \odot Uh_{t-1}) \tag{2.17}$$

This process is shown in figure 2.15. The input x_t is multiplied with a weight W and the hidden state h_{t-1} is multiplied with a weight U. After that, an element-wise product is calculated between the reset gate r_t and Uh_{t-1}. This will remove the information from the previous time steps that is not relevant for the output. An r_t value of 0 means that the information from the past is not relevant, whereas a value of 1 would mean that all information from the past is relevant. Finally, the sum the left and the right side is done before the tanh is applied in order to get h_{t-1}.

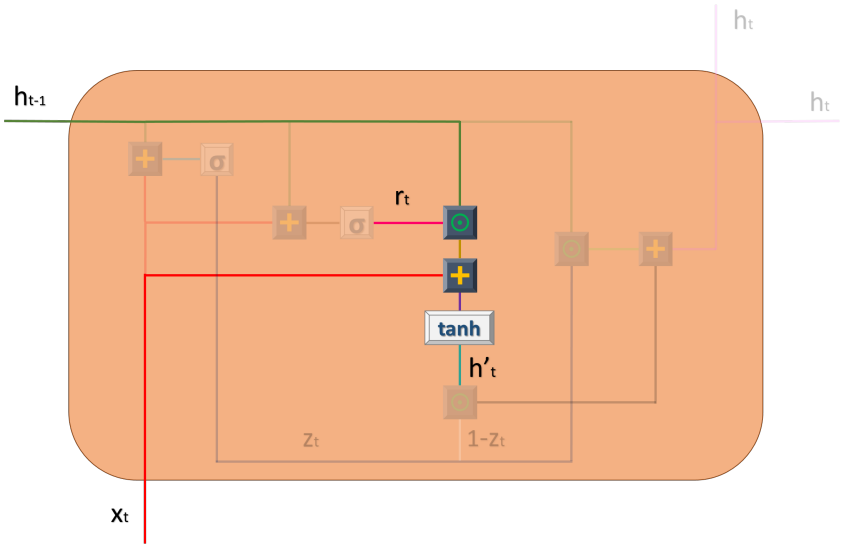

Figure 2.15: The calculation of the current memory in a GRU.

At last, it is time to calculate the output hidden vector h_t, which passes the information from the current GRU unit to the following one. This calculation is done according to equation 2.18, which is illustrated in figure 2.16.

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t^{'} \tag{2.18}$$

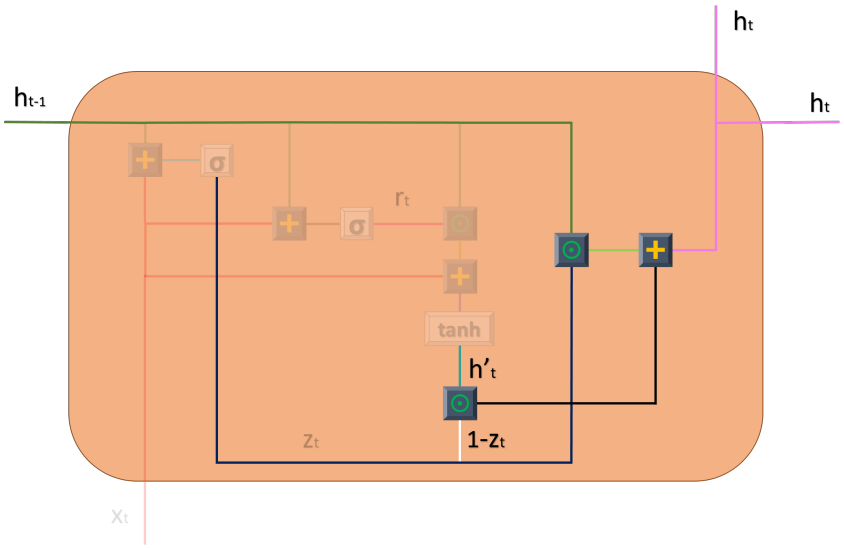

Figure 2.16: The calculation of the output in a GRU.

2.2.5 Encoder-Decoder with attention mechanism

In this thesis, a Deep Q-Learning approach is used to train an intelligent agent to solve the DRC-FJSSP. The NN architecture chosen to approximate the Q-value function was the Encoder-Decoder with attention mechanism. This choice will allow the agent to receive inputs with variable sizes, a required feature for it to be able to solve problems with any number of operations to schedule, without the need to generate a size-fixed simplified representation of the input information. In figure 2.17, a first layout of the chosen architecture is given. A step by step presentation of it will be given in the following sections.

2.2.5.1 Encoder-Decoder

The fundamental idea behind an Encoder-Decoder architecture is to have two different RNN's: one which sequentially receives the input information and summarizes it in a vector representation of fixed dimensionality (encoder) and another which receives this vector and sequentially generates the output (decoder). This means that the only information the decoder receives is the last encoder hidden state. A representation of this architecture is given in figure 2.18.

The problem with it is that, when the input length gets really large, it is extremely difficult to sufficiently summarize all the input data in a single fixed-sized vector. In other words, this might lead the agent to forget input information. That is why an attention layer is introduced.

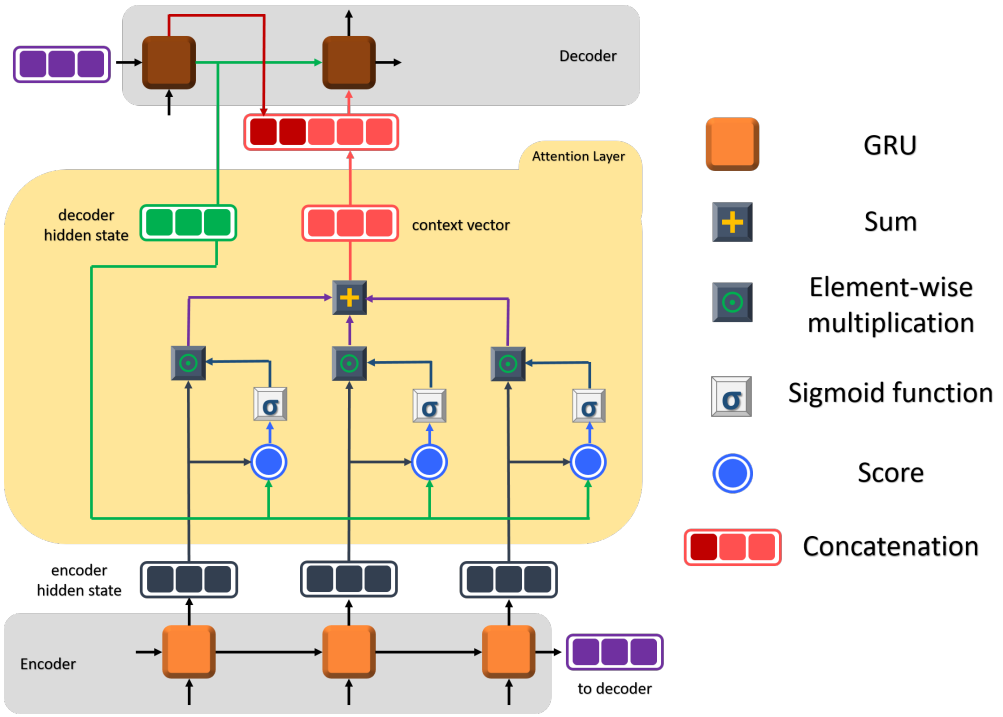

Figure 2.17: An overview of the Encoder-Decoder with Attention mechanism architecture.

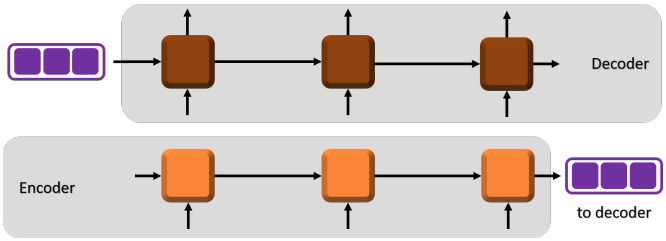

Figure 2.18: A basic Encoder-Decoder architecture.

2.2.5.2 Attention Mechanism

The attention layer is an interface that allows the Decoder to access information about all the Encoder hidden states at all times. Moreover, it highlights useful parts of that input data in order to let the model focus on those key areas and learn the connections between them. This way, the model can effectively remember and process long input sequences [39].

There are two types of attention mechanisms: one that uses all encoder hidden states (global attention) and another that uses only a subset of them (local attention). All throughout this thesis, global attention was the type of mechanism used and referred to as "attention".

During the steps on the attention layer, a set of attention weights is calculated, one weight for each encoder hidden state. These attention weights are the ones who define how relevant each input is for the output of the current time step and so are used to derive the context vector fed as input to the decoder.

One of the fields where the usage of these Encoder-Decoder with attention mechanism architectures are popular is Neural Language Processing, or more specifically the translation task. A representation of the attention weights is given in figure 2.19, where the darker lines represent higher weight values.

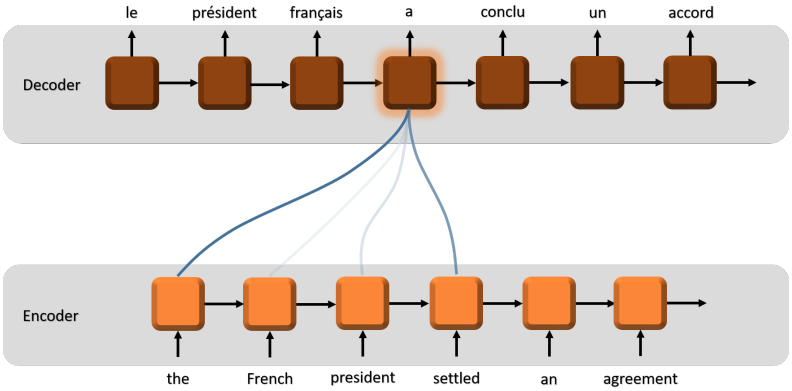

Figure 2.19: An intuitive example about the influence of the attention weights.

The first step which takes place in the attention layer is the calculation of a score between the current decoder hidden state and all the encoder ones, with exception of the one in purple (see figure 2.20), fed as an input to the first decoder step.

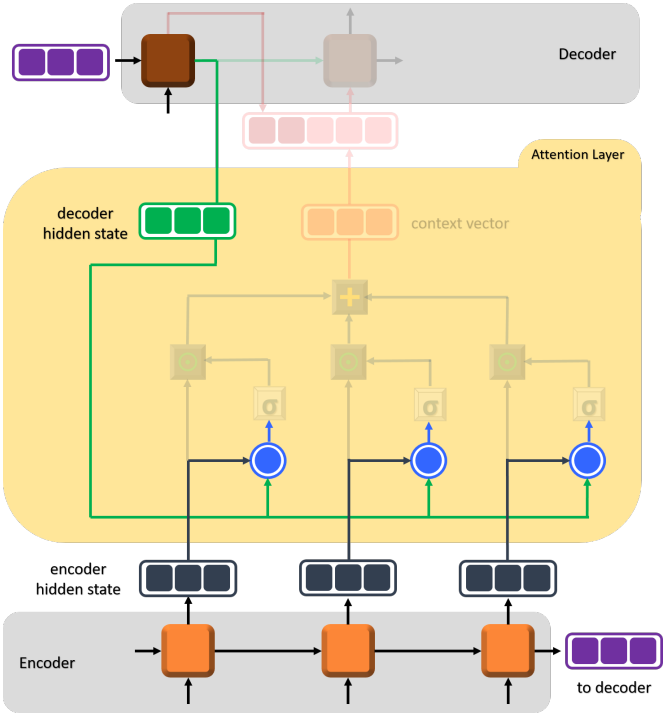

Figure 2.20: Hidden state score calculation in the attention layer.

Different operations between the vectors can be applied to calculate this score. One option is to calculate the dot product between the two vectors. In this thesis, each of the weights u_v^d, the attention weight in the decoder step d relative to the v-th encoder hidden state, was calculated according to equation 2.19.

$$u_v^d = w^T tanh(W_1 e_v + W_2 d_d),\ v \in (1,...,n) \tag{2.19}$$

where w, W_1 and W_2 are learnable weights, e_v is the v-th encoder hidden state and d_d is the d-th decoder hidden state.

Then all the scores go through a softmax function, which is a generalization of the logistic function for multiple dimensions. After applying softmax, each score will be in the interval $(0, 1)$ and the components will add up to 1. After that, each encoder hidden state is multiplied by its own softmaxed score. And finally, all the resulting vectors are summed up, generating the context vector. This set of steps ends up being a linear combination the encoder hidden states, where the constants that are multiplied to the each hidden vector is its own score. The calculations made during these steps are presented in equation 2.20 and their representation is given in figure 2.21.

$$a_v^d = softmax(u_v^d),\ v \in (1, ..., n) \tag{2.20a}$$

$$d_d^{'} = \sum_{v=1}^{n} a_v^d e_v \tag{2.20b}$$

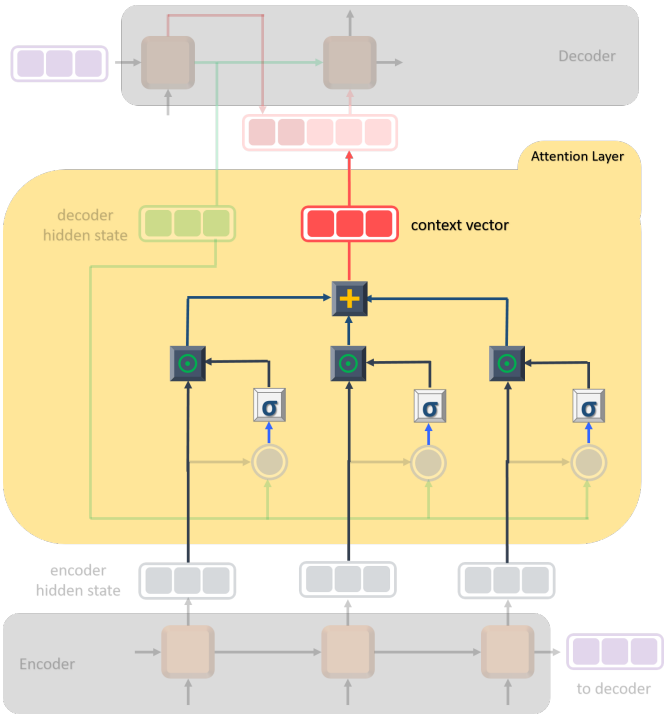

Figure 2.21: Context vector derivation through a linear combination of the encoder hidden state vectors.

A final step is left, during which the context vector is fed into the decoder. Although there are different common ways of doing this, in this thesis the Bahdanau's method was chosen [40]. According to this method, as represented in figure 2.22, the input to the next decoder step is the concatenation between the output from the previous decoder time step (dark red) and context vector from the current time step (pink).

Despite its complexity, this network architecture can still be trained using Backpropagation. The algorithm will change the weights in the RNNs and in the score function in order to ensure that the outputs will be close to the ground truth. These weights will affect the encoder hidden states and decoder hidden states, which in turn affect the attention scores.

At last, there is one small detail still worth highlighting. It is possible to run as many steps of the decoder as needes. In other words, for the translation task, the agent can output as many words (one by one) as needed. However, the output of a GRU is constant in size. This means, that size of the dictionary of words that can be

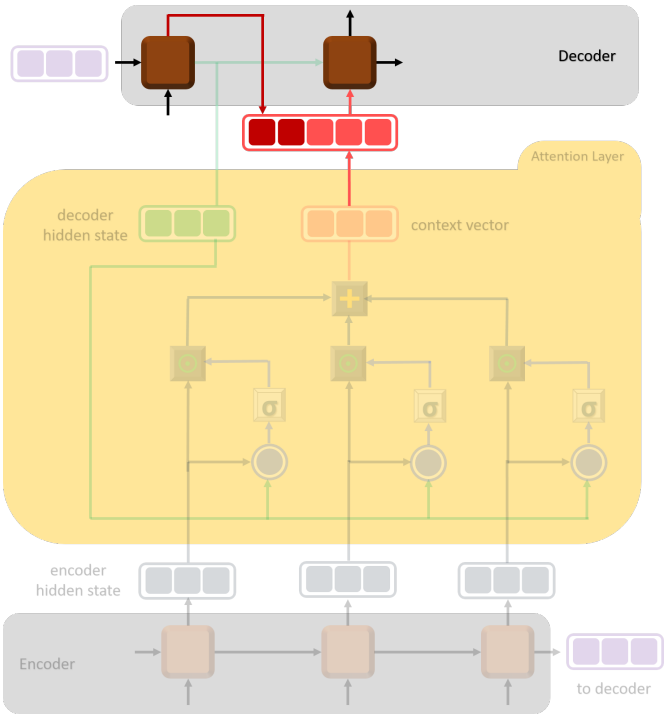

Figure 2.22: The concatenation of the context vector with the output of the previous decoder time step.

used is fixed. It cannot be adapted from problem to problem. This constitutes a limitation that Pointer Networks architectures aim to solve.

2.2.6 Pointer Networks

Pointer Networks introduce a very simple modification to the attention model that make it possible to solve combinatorial optimization problems where the output dictionary size depends on the number of elements in the input sequence [41].

It does that by using the attention weights as pointers to the input elements and taking them as the output of each decoder step. The softmax function generates these weights such that they will add up to 1, which means that they can be interpreted as probabilities, just like in equation 2.21.

In other words, it uses the attention weights as pointers to the input elements.

$$u_v^d = w^T tanh(W_1 e_v + W_2 d_d),\ v \in (1,...,n) \tag{2.21a}$$

$$P(C_d|C_1,...,C_{d-1},\mathscr{P}) = softmax(u^d) \tag{2.21b}$$

Here $\mathscr{P} = \{P_1,...,P_n\}$ is the input sequence and $C^{\mathscr{P}} = \{C_1,...,C_{z(\mathscr{P})}\}$ is the output sequence.

So C_d is the output of the d-th decoder step. It is a vector with the attention weights of that step (one weight for each encoder hidden state).

Chapter 3

Implementation

3.1 Numerical Model

In a DRC-FJSSP there is a set of n jobs $J = \{J_1, J_2, ..., J_n\}$ to be processed at a set of m machines $M = \{M_1, M_2, ..., M_m\}$ operated by a set of w workers $K = \{K_1, K_2, ..., K_k\}$. Each job has a predefined sequence of n_i operations $O = \{O_{i,1}, O_{i,2}, ..., O_{i,n_i}\}$. Let p_{ij} be the processing time of the operation $O_{i,j}$. Operation $O_{i,j}$ can only be processed at one machine out of a set of $M_{i,j}$ eligible machines. Each machine can process only one operation at a time and there is no preemption, which means that since an operation has started it cannot be stopped until it is finished. Each worker K_k can only operate a subset of M, for which K_k is an eligible worker. Thus, there is an eligibility matrix E_{mk}, where the element (m,k) is a 1, if K_k is an eligible worker for M_m, or, otherwise, it is a 0. All jobs, machines and workers are available at time 0. The goal is to minimise the maximum completion time, the makespan C_{max} by assigning a compatible machine and an eligible worker to each operation as well as arranging the processing order of operations on each machine.

Let st_{ij} be the starting time of $O_{i,j}$, and rt_{mk} be the ready time of machine M_m operated by worker K_k, and N be a large enough number. Mathematically, the DRC-FJSSP with makespan minimisation can be formulated as follows:

$$Min\, C_{max} \tag{3.1}$$

Subject to:

$$st_{i(j+1)} \geq st_{ij} + \sum_{m}\sum_{k} p_{ij} x_{ijmk},\ J_i \in J,\ j = 1,2,...,n_i - 1,\ M_m \in M_{i,j},\ E_{mk} > 0 \tag{3.2}$$

$$st_{i^{`}j^{`}} + (1 - \zeta_{ijm-i^{`}j^{`}m})N \geq s_{ij} + \sum_{k} p_{ij} x_{ijmk},\ J_i \in J,\ j = 1,2,...,n_i,\ M_m \in M_{i,j},\ E_{mk} > 0 \tag{3.3}$$

$$rt_{m^{`}k} + (1 - \xi_{mk-m^{`}k})N \geq rt_{mk},\ K_k \in K,\ E_{mk},\ E_{m^{`}k} > 0 \tag{3.4}$$

$$rt_{mk} + (1 - x_{ijmk})N \leq st_{ij},\ J_i \in J,\ j = 1,2,...,n_i,\ M_m \in M_{i,j},\ E_{mk} > 0 \tag{3.5}$$

$$\sum_{m}\sum_{k} x_{ijmk} = 1,\ J_i \in J,\ j = 1,2,...,n_i,\ M_m \in M_{i,j},\ E_{mk} > 0 \tag{3.6}$$

$$x_{ijmk} = \begin{cases} 1, \text{ if } O_{i,j} \text{ is processed on } M_m \text{ operated by } K_k \\ 0, \text{ otherwise} \end{cases} \tag{3.7}$$

$$\zeta_{ijm-i'j'm} = \begin{cases} 1, \text{ if } O_{i,j} \text{ is processed before } O_{i'j'} \text{ on } M_m \\ 0, \text{ otherwise} \end{cases} \tag{3.8}$$

$$\xi_{mk-m'k} = \begin{cases} 1, \text{ if } M_m \text{ is operated before } M_{m'} \text{ by } K_k \\ 0, \text{ otherwise} \end{cases} \tag{3.9}$$

where equation 3.2 ensures that the precedence constraints are not violated; equation 3.3 guarantees that a machine can process only one operation at a time; equation 3.4 ensures that a worker can operate one machine at a time; equation 3.5 ensures that an operation cannot start unless the assigned resources are ready; equation 3.6 guarantees that each operation is assigned to only one compatible machine operated by one eligible worker.

3.2 Proposed Solution

A great inspiration for this thesis was retrieved from the work in [42] and in [33], regarding the use of RL to solve the DRC-FJSSP and the RL algorithm, respectively. In the following subsections, both these aspects are discussed as the proposed solution is presented.

3.2.1 Solve the DRC-FJSSP using RL

In [42], a novel approach, using Deep Q-Learning, was proposed for solving the DRC-FJSSP. In their work, they initialized a schedule according to a set of rules and ran a step by step optimization process. At each step, the agent would receive a representation of the current schedule, the state, as input and output a choice between 2 possible actions: Move and Reassign Pool. Each of them was an heuristic that the agent could choose to apply in order to update the state. Soon, they realized that a full schedule representation, a vector with all the problem variables, could not be used as a state, since its size would vary with the number of jobs and resources of a scheduling problem and the number of inputs of a NN has to be fixed. Therefore, a 30 feature state representation was used. This arbitrarily chosen set of features was supposed to summarize all the state information.

In the early research of this thesis, this was identified as a plausible source of the limited optimization results achieved, since the agent's intelligent intuition had limited power and information to act upon. Allowing the intelligent agent to analyse, move and assign individual operations would give it total control and thus unchain its true potential. In order to do that, one would need to able to feed the NN a variable number of input features and let it choose between a variable number of possible actions. For example, consider a simple problem with n jobs. As

an input, the agent needs to receive n input vectors, each one of them with information regarding each individual job. Whereas, as an output, it needs to be able to choose to move one job, out of the whole set, into a new position in the schedule. Thus, it would need to output a set of n values, being each of those values the score of choosing to move each job. Also, unlike in [42], the initial schedule is randomly generated and only feasible states are allowed throughout the optimization process. This means that the initial state is feasible and every move made by the agent needs to keep it like that, to be allowed.

In order to do this, the encoder-decoder with attention mechanism was used. It is a RNN architecture, typically used for Natural Language Processing, that is able to receive inputs with variable sizes, deliver outputs with variable sizes and have a high quality memory, even for long input sequences. These features make this architecture potentially powerful for the DRC-FJSSP. The overview of this proposed solution is given in figure 3.1.

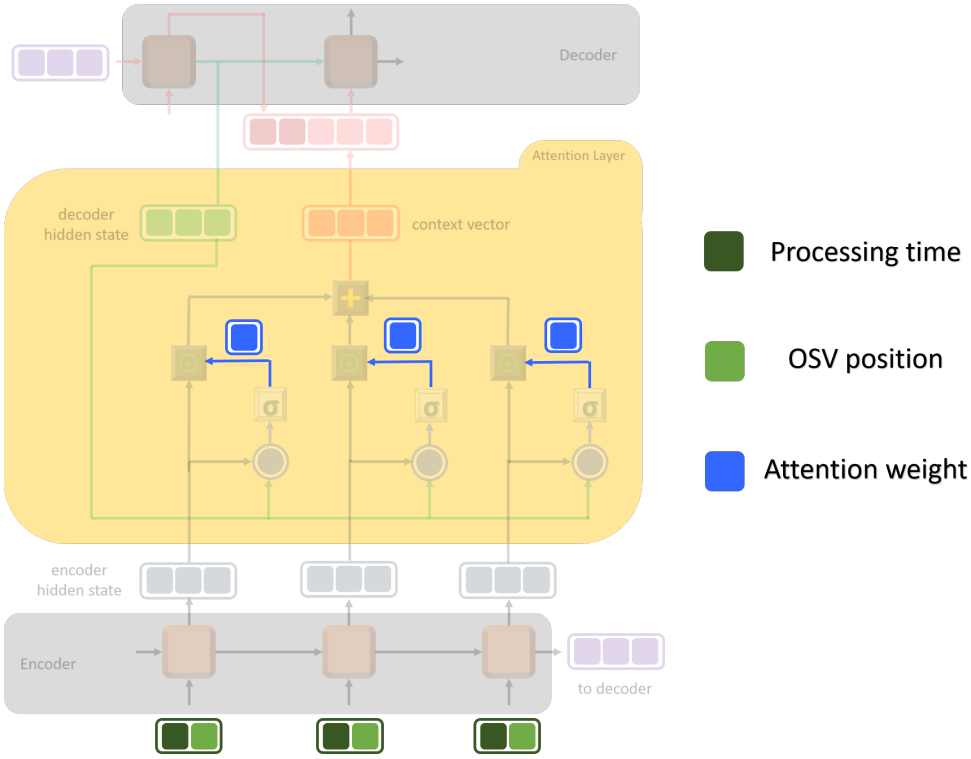

Figure 3.1: Overview of the proposed RNN architecture, with highlight to its inputs and outputs.

This architecture is quite complex on its own, so there was an attempt to simplify the inputs to the maximum. Therefore, there will be a number of encoder steps equal to the number of operations to schedule. At the v-th encoder step, information about the v-th operation is submitted to the agent. That information is composed by a 2 component vector. The first component represents the normalized processing time of that operation. Whereas, the second component represents the normalized position of that operation in the OSV.

The Operation Sequence Vector (OSV) is one of three different vectors responsible for encoding the schedule information. The other vectors are the Machine Allocation Vector (MAV) and the Worker Allocation Vector (WAV). This encoding scheme is similar to the ones used in many state-of-the-art methods and it aims to simplify schedule representation and manipulation.

All of these vectors have a number of components equal to the number of operations to schedule. The OSV gives us the order by which operations are put in the schedule, which means that the v-th component of the vector

Table 3.1: Example of OSV, WAV and MAV schedule encoding vectors.

OSV	4	1	7	9	5	10	6	2	8	3
MAV	1	3	1	2	2	3	1	1	3	3
WAV	2	1	2	2	2	1	1	2	1	1

is the i-th operation to be scheduled. The v-th component of the MAV and the WAV tell us what is the machine and worker allocation for the operation in the v-th component of the OSV. An example of this encoding scheme is presented in table 3.1.

During the decoding process, operations are put in the schedule one at a time, following the order specified in the OSV. They are put at the minimum feasible time, which will logically depend on the availability of the allocated worker and machine. For example, according to table 3.1, the first component in the OSV is operation number 4. So, this would be the first operation to be put in the schedule. It would be allocated to machine number 1, operated by worker number 2. Next, the second component of the OSV, operation number 1, would be put in the schedule, allocated to machine number 3, operated by worker number 1. And so on, until operations are put in the schedule. This procedure is represented in figure 3.2.

Finally, as can be seen in figure 3.1, the decoder outputs will not be considered as the agent outputs. In fact, the attention weights will be considered as the output of the network, as in a Pointer Network. This is done for 2 main reasons: computational power and intuition.

First of all, computational power. The number of outputs our agent needs to deliver is equal to the squared value of the number of operations to schedule. It has n operations to choose from and n positions in the OSV to move them to, since the agent is allowed to "move" an operation to the same position it is in. So, all in all, it has ${n_{operations}}^2$ possible moves to choose from. Since this number varies with the problem dimension, the agent would need to output the score of each of this moves one by one in the decoder, which means it would need ${n_{operations}}^2$ decoder steps. At each decoder step, there are n attention weights calculated, one for each encoder hidden state. This means that, if they are used as the network's outputs, there is only the need to run n decoder steps to find the ${n_{operations}}^2$ score values. This means that decoder's GRU is ran a much smaller number of times, especially when the problem size increases a lot. Therefore, a lot of computational effort is saved.

Secondly, intuition. It was shown that the attention weights are score values that basically evaluate the importance of each input the encoder received. In a way, they be can pictured as the "amount of time" the agent would "look" at each of these inputs before deciding what would be the best output to deliver. So the word intuition is used in the sense that it is reasonable to assume that if the agent is concerned a lot ("looks a lot of time") with an operation, then that operation is probably the most critical one to improve the current solution quality. So, the operation with the highest attention weight is probably the most suitable choice to be moved. So, there are n decoder steps, each one with n attention weights. First, the highest attention weight of them all is located. If it is the v-th attention weight of the d-th decoder step, then the agent chooses to move the v-th operation to the d-th position in the OSV. Of course, the justification behind this reasoning is not rigorous, but it is an assumption that can save a lot of computational effort and, so, it is definitely worth testing and evaluating the results.

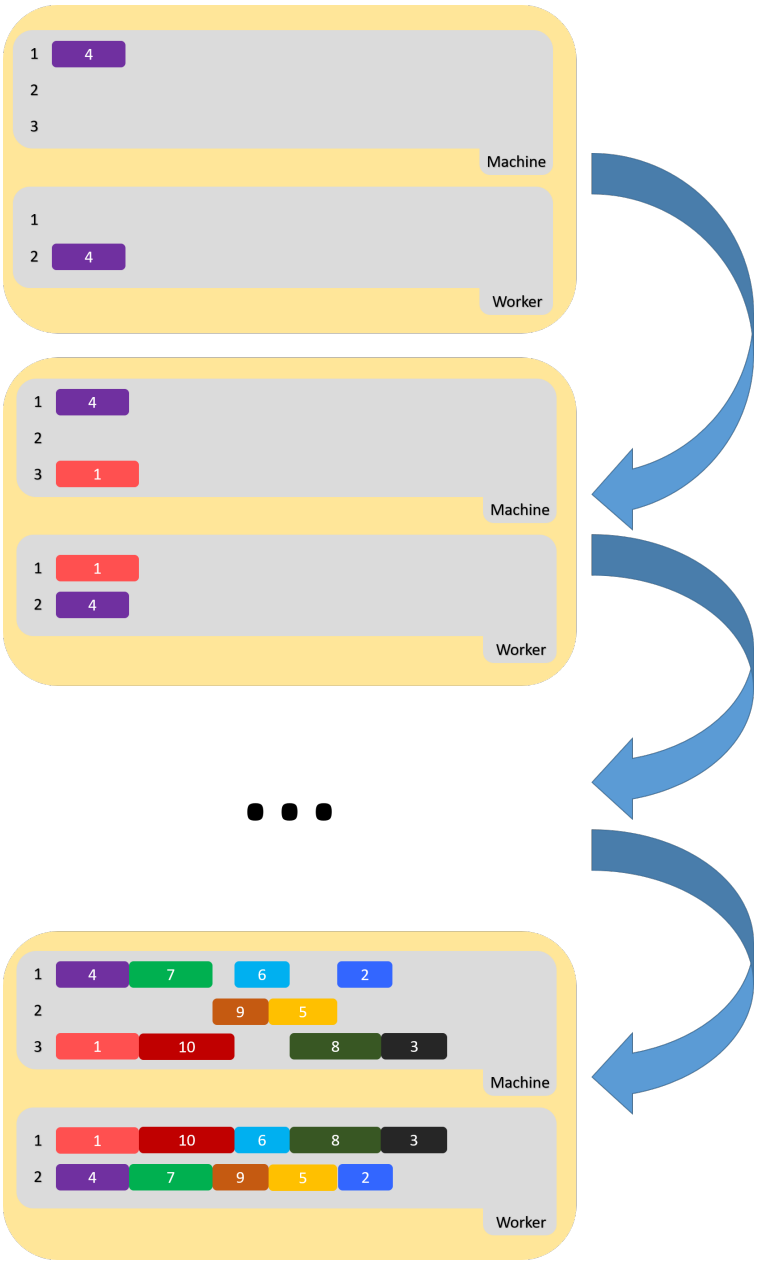

Figure 3.2: Schedule decoding procedure steps.

The simplified inputs given to the agent do not contain any information regarding the workers and machines available for allocation. That is why the agent only manipulates the OSV. The allocation of machines and workers will be made at the schedule decoding phase, according to the Earliest Feasible Time heuristic. This means that each operation will be allocated to the worker-machine pair that can process it the earliest. If there are more than a single pair available at the same earliest time, then the first of them is chosen. Worker-machine pairs (M_m, K_k) are listed according to equation 3.10. So, the first would be the pair with the smallest index. This was made in order to ease programming implementation.

$$Pair\ index = (m-1)n_{workers} + k \tag{3.10}$$

3.2.2 RL algorithm

In [33], a deep Q-network was presented. Using end-to-end reinforcement learning, the agent was able to achieve a level comparable to that of a professional human games tester across a set of 49 Atari games, using the same algorithm, network architecture and hyperparameters. Motivated by those impressive results, the same reinforcement learning algorithm was followed in this thesis, like described in algorithm 2.

Algorithm 2: Deep Q-Learning with experience replay

Initialize replay memory D to capacity DN ;
Initialize action-value function Q with random weights w ;
Initialize target action-value $\hat{Q}$ with weights $w^- = w$;
for *episode = 1, ..., M* **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  for *t = 1, ..., T* **do**
    With probability ε select a random action act_t ;
    otherwise select $act_t = argmax_{act} Q(\phi(s_t), act; w)$;
    Execute action a_t in simulation and observe reward r_t and state x_{t+1} ;
    Set $s_{t+1} = s_t, act_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
    Store transition $(\phi_t, act_t, r_t, \phi_{t+1})$ in D ;
    Sample random minibatch of transitions $(\phi_v, act_v, R_v, \phi_{v+1})$ from D ;
    if *episode terminates at step v+1* **then**
      $targ_v = R_v$;
    else
      $targ_v = R_v + \gamma\, max_{act'} \hat{Q}(\phi_{v+1}, a'; w^-)$;
    end
    Perform a gradient descent step on $(targ_v - Q(\phi_v, act_v; w))^2$ with respect to the network parameters w ;
    Every C steps reset $\hat{Q}$ = Q ;
  end
end

All in all, during training the agent will try different moves and save that information (initial state, move and

final state) in a memory, together with the respective reward. Exploration and exploitation are balanced according to the ε-greedy algorithm. This means that the agent will take a random action with probability ε, while the rest of the times it will choose the action elected by the output of the RNN. This ε value was changed linearly between 100%, in the beginning, and 10%, after 80% of the training cycles. During the rest of the training cycles, the ε is of 10%. This is a typical range of values when applying ε-greedy.

Each training cycle is composed by a number of episodes. Every episode, the agent receives a new problem and does $3 * n_{operations}$ moves, where $n_{operations}$ is the number of operations to schedule for that problem. This number of moves is justified in appendix A, while the number of episodes is chosen empirically.

Then, a mini-batch is generated by randomly select 32 instances from the memory. Different mini-batch sizes were tested, but the best results were found for that value. After that, a target value $targ_v$ is calculated for each of those instances according to the if statement in 2. The γ value is a parameter that is common practice to establish as 0.99, even though it might be optimized to the problem at hand. Finally, a gradient descent is performed trying to minimize the squared error between the target values and the predictions (output moves) dictated by the RNN.

The rewards play a huge part in the success of a RL method and they usually depend and need to be adapted to the problem at hand. In this thesis implementation, some different reward combinations were tried, until a combination was found that delivered good results. In the end, it was found that to deliver strong rewards when good moves were made was better for the algorithm to learn then to deliver penalties for bad movements. So a reward of 0.05 was given for every unit of time a move could improve the solution makespan relative to the best makespan found so far. Otherwise, if a new best solution was not found, a small penalty of 0.01 was given. Furthermore, a growing penalty of 0.001 times the number of moves so far was given in order to incentive the agent to find good solutions fast. At last, the agent was given the possibility to suggest moves that do not change the schedule. It happens when the agent wants to move one operation to a position in the OSV it is already in. In that case, it was considered that the agent is stating that the schedule is already optimized. No penalties are given if that is the chosen move.

After the training process, hopefully the Agent will be ready to autonomously generate quality solutions for the DRC-FJSSP. Those solutions are generated through a step by step optimization process. At each step the agent receives as an input information about the current solution state, just as it did during training, and outputs the selection of a move operation. A rule was established to put an end to the agent's optimization process. A schedule is considered to be optimized when the agent chooses a move that does not change the OSV. When that happens, the best schedule found so far, the one with the lowest makespan, is considered to be the optimized solution. Additionally, the agent only has a limited number of moves to optimize the solution, equal to 50% of the number of operations to schedule for the problem at hand. After those moves, the best schedule found so far, the one with the lowest makespan, is considered to be the optimized solution. The 50% threshold was defined empirically as a trade-off between the minimization of computational time and the maximization optimization performance.

3.3 Performance Metrics

In order to evaluate the quality of the solutions generated by the agent, a performance metric is needed. However, there is a lack of good performance metrics in the literature, with many of the most successful papers evaluat-

ing its proposed models by comparing their solution's makespan with the ones from solutions of another state-of-the-art techniques. In this work, it was considered that it would be useful to have a performance metric that would evaluate the model's performance independently of other methods.

The goal was to develop a performance metric whose value would give a perception about the quality of the solution and which would be independent of the problem's dimension. For example, the optimal makespan is extremely variable from problem to problem. Therefore, the makespan by itself is not sufficiently enlightening, since it does not state how far we are from an optimal solution. In the end, two new performance metrics are proposed: the Job Dimension Percentage and the Job Difference Per-mille, calculated according to equations 3.11 and 3.12, respectively.

Aiming to minimize the makespan of the DRC-FJSSP is equivalent to minimizing the idle time of the resources, which is the time during which the resources are not being used. More specifically, having considered that in every problem the number of available workers is always smaller than the number of available machines, something common in the DRC-FJSSP, the number of workers becomes the bottleneck of the production system. So, both performance metrics developed evaluate the idle time of the critical resource, the workers or other, if that is the case.

The Job Dimension Percentage gives the ratio between average idle time per worker and the average dimension of the operations to schedule. The intuition behind this metric is that an idle time of 10 units of time should have the same meaning for a problem with an optimal makespan of 40 units of time and for another of 80 units of time. Relatively, that idle time is undoubtfully better for the problem with 80 units of time. However, if the dimension of operations in the schedule is similar and the only difference from one case to another is the number of operations to schedule, then that relative improvement is not necessarily reflected in a better scheduling capability. Consider a problem with 16 operations and another with 32 operations, all of them with 5 units of time and to be distributed between 2 workers.

In both problems one worker will be assigned two more operations than the other worker. Assuming that all operations can be performed consecutively by workers, without idle times in between, then in the end one worker will work more 10 units of time than the other. Relatively, the solution to the problem of 32 operations seems better, because there is an idle time of 10 units of time for a makespan of 85, instead of 45 for the 16 operations. However, the scheduling error was the same: assigning 2 more operations to one worker than to the other. Despite of that, the proposed metric would give the same score to both solutions.

$$av\_worker\_idle\_time = \frac{\sum_{k=1}^{K} idle\_time_k}{K} \tag{3.11a}$$

$$av\_proc\_time = \frac{\sum_{i=1}^{n_{operations}} proc\_time_i}{n_{operations}} \tag{3.11b}$$

$$Job\ Dim(\%) = 100 \times \frac{av\_worker\_idle\_time}{av\_proc\_time} \tag{3.11c}$$

The Job Difference Permille gives the ratio between the average idle time per worker and the average difference between the dimensions of the operations to be scheduled. This metric was developed based on the intuition that, in order to eliminate idle times, the person responsible for optimizing the scheduling looks at the difference between

the processing times of operations. Consider again a case with 2 workers. We assume that one of them ends his operations 4 units of time before the other. One of the operations under the responsibility of that worker had 15 units of time, while one of the operations for which the other (more busy) worker was responsible had 17 units of time. In this case, if the worker assignment of these two operations is switched, then both workers will be left with the same workload, since the first (less busy) will be occupied for another 2 units of time and the second will be occupied for less 2 units of time. Added together, they cancel the 4 units of time difference there was initially. In this example, the influence of the difference in processing times of operations in the optimization of a scheduling was noticeable.

$$av\_worker\_idle\_time = \frac{\sum_{k=1}^{K} idle\_time_k}{K} \tag{3.12a}$$

$$av\_proc\_time\_diff = \frac{\sum_{i=1}^{n_{operations}-1} \sum_{v=i+1}^{n_{operations}} (proc\_time_i - proc\_time_v)^2}{\sum_{i=1}^{n_{operations}-1} \sum_{v=i+1}^{n_{operations}} 1} \tag{3.12b}$$

$$Job\ Diff(‰) = 1000 \times \frac{av\_worker\_idle\_time}{av\_proc\_time\_diff} \tag{3.12c}$$

Chapter 4

Applying the model to a DRC-FJSSP: A Proof of Concept Example

In order to start to verify the above mentioned model, it was considered that by tackling a basic version of the DRC-FJSSP it would be possible to quickly assure the model's feasibility and convergence. Therefore, a set of assumptions were used to simplify the DRC-FJSSP. First of all, there were no release or due dates considered. Secondly, it was defined that each machine is eligible for any operation and that each worker can operate any machine. Finally, jobs were only allowed to have a single operation, so that no operation set would need to be scheduled in a precise sequence. This effectively reduced the number of constraints in the problem and eased the model's implementation.

4.1 Solution Quality Comparison

Both performance metrics presented in section 3.3 will evaluate by themselves the quality of the generated solutions for the DRC-FJSSP. However, it would still be useful to compare the results of the agent with the ones generated by other methods. For the basic DRC-FJSSP tackled first, some very basic heuristics were considered to be efficient for comparison. Namely, the Longest Processing Time (LPT) and the Best Fit Decreasing (BFD) heuristics were chosen for their simplicity. The LPT is an heuristic that is commonly applied to Job Shop problems, whereas the BFD is normally used to solve bin packing problems. With the limited number of constraints left in the problem, the DRC-FJSSP becomes sort of a bin packing problem, when giving emphasis to the bottleneck resource. One starts by considering that it is possible to pack all operations in a number of bins equal to the number of workers and with a size equal to the lower bound. The lower bound is calculated according to equation 4.1 and is always less or equal to the optimal makespan.

$$LowerBound = ceil(\frac{\sum_{i=1}^{n_{operations}} proc\_time_i}{n_{workers}}) \tag{4.1}$$

where the ceil function rounds the input to the nearest higher integer.

If the BFD cannot fit all the operations in these bins, it tries to apply the algorithm again, now with bins with

one unit of time larger. The heuristic keeps doing that until a feasible schedule is found. This adaptation to the BFD is needed since in the bin packing problem the variable is the number of bins, not the size of them. Whereas, in the Job Shop problem the variable is the makespan, not the number of workers.

4.2 Agent Training

This first agent was trained with only 25 training cycles, which means only 25 updates were made to the original randomly generated network weights. Of course, this amount of training cycles is by far insufficient to thoroughly train a NN, independently of its architecture, especially when RL is being used. Nevertheless, the goal of this first agent was to find out if the proposed architecture would be able to learn how to optimize the DRC-FJSSP, so at this point the aim was to notice if the agent was learning. The weights that during training gave the lowest Job Dim Percentage for the training instances were used. Since only 25 training cycles were executed, there is no danger of over-fitting. So, there is no problem in using the training set for validation. For the hyperparameters, a combination that achieved faster but steady convergence during training was used. For the same reason, a learning rate of 0.01 was chosen, together with no target network $\hat{Q}$ weights update, a batch size of 32 and the reward system described earlier. The training set was composed of 10 different problems, each with 10 operations to schedule, with processing times between 2 and 18 units of time, randomly generated.

4.3 Simulation Description and Results

The 3 methods (Agent, LPT and BFD) were applied to 25 different problems, divided in 5 different subsets. Each subset had 5 problems with the same number of operations to schedule. The number of operations to schedule in each of the subsets was 10, 30, 100, 300 and 1000, with 3, 10, 25, 50 and 100 machines and 2, 5, 10, 20 and 35 workers, respectively. All the above mentioned methods were implemented in Python and ran on a laptop with a 2.60GHz Intel i7-4720HQ CPU. The average results of the simulations are presented in table 4.1. The full results are listed in appendix B.

The Min Finish time refers to the unit of time at which a worker first finished all its assigned operations, whereas the Max Finish time refers to the unit of time at which the last worker finished all its assigned operations. Logically, by definition, the makespan of a solution is always equal to the Max Finish Time.

4.4 Discussion of Results

4.4.1 Performance

Looking at the results, it is noticeable that the agent achieves really good solutions, even for problems of high dimensionality. The makespans obtained using the agent are on average less than 2 time units away from the Lower Bound of the problems, a distance that corresponds to an average of 1.32% of the problem's makespan on the worst subset. Comparatively with the other methods used, for problems with 10 operations to schedule, the dimensions for which the agent was trained, the agent's performance is superior to the performance of both heuristics, like one can see, for example, from the average Distance to Bound. As the dimensionality of the problem grows, the

Table 4.1: Averaged results and standard deviation from the Proof of Concept simulation.

Prob Size: 10 jobs	Agent		LPT		BFD	
	Avg	Std	Avg	Std	Avg	Std
Job Dimension (%)	**4.18**	3.72	4.48	4.17	14.37	8.92
Job Difference (‰)	15.42	17.42	**10.83**	9.58	40.77	22.80
Min Finish time	48.00	5.14	48.00	5.40	47.00	4.94
Max Finish time	**48.80**	5.11	**48.80**	4.83	49.80	5.42
Lower Bound	48.60	4.96	48.60	4.96	48.60	4.96
Distance to Bound (%)	**0.38**	0.77	0.44	0.89	2.41	1.45
Execution Time (s)	1.06	0.08	**0.01**	0.01	**0.01**	0.01
Prob Size: 30 jobs	Agent		LPT		BFD	
	Avg	Std	Avg	Std	Avg	Std
Job Dimension (%)	10.73	5.02	**8.86**	4.05	9.21	2.58
Job Difference (‰)	29.93	13.52	**24.73**	10.77	26.11	8.32
Min Finish time	59.40	4.22	60.00	4.20	59.40	4.13
Max Finish time	62.20	4.83	**62.00**	4.69	**62.00**	4.20
Lower Bound	61.40	4.59	61.40	4.59	61.40	4.59
Distance to Bound (%)	1.28	0.65	**0.97**	0.80	1.03	0.85
Execution Time (s)	9.35	0.07	**0.01**	0.01	0.02	0.01
Prob Size: 100 jobs	Agent		LPT		BFD	
	Avg	Std	Avg	Std	Avg	Std
Job Dimension (%)	16.96	3.66	**5.22**	2.55	7.19	3.03
Job Difference (‰)	51.07	10.89	**16.23**	8.89	21.87	9.33
Min Finish time	99.60	2.58	102.00	2.53	280.80	2.67
Max Finish time	104.40	2.80	**103.20**	2.93	103.40	2.87
Lower Bound	103.20	2.93	103.20	2.93	103.20	2.93
Distance to Bound (%)	1.17	0.41	**0.00**	0.00	0.20	0.39
Execution Time (s)	111.17	0.09	0.06	0.00	**0.05**	0.00
Prob Size: 300 jobs	Agent		LPT		BFD	
	Avg	Std	Avg	Std	Avg	Std
Job Dimension (%)	22.73	5.16	6.95	3.60	**4.97**	3.03
Job Difference (‰)	62.18	12.07	18.92	9.56	**13.42**	7.60
Min Finish time	149.40	2.42	151.40	2.42	150.20	2.79
Max Finish time	154.40	2.06	152.80	2.04	**152.60**	2.06
Lower Bound	152.40	2.42	152.40	2.42	152.40	2.42
Distance to Bound (%)	1.32	0.43	0.27	0.33	**0.14**	0.27
Execution Time (s)	1219.10	0.51	**0.28**	0.00	**0.28**	0.00
Prob Size: 1000 jobs	Agent		LPT		BFD	
	Avg	Std	Avg	Std	Avg	Std
Job Dimension (%)	24.28	1.99	14.35	2.01	**6.40**	3.00
Job Difference (‰)	69.67	5.70	41.17	5.71	**18.28**	8.45
Min Finish time	285.20	2.23	286.60	2.15	285.00	2.19
Max Finish time	290.20	2.23	289.20	2.23	**288.40**	2.33
Lower Bound	288.20	2.23	288.20	2.23	288.20	2.23
Distance to Bound (%)	0.69	0.01	0.35	0.00	**0.07**	0.14
Execution Time (s)	22272.31	13.68	2.32	0.01	**2.31**	0.02

heuristics achieve a slightly better performance, but since the quality of the agent's solution is still very good, it is fair to say that, having just trained for really small problems, the agent generalizes well its knowledge for much higher problem dimensionalities. These are extremely promising results as this means that one can train the agent much faster, using only low dimensionality problems and then expect it to apply that gained knowledge in proficiently solving high dimensional problems.

4.4.2 Execution Time

The big disadvantage of the agent, compared with the heuristics, lies on the execution time used to solve the problems. For the heuristics, being such simple rules, it is with no surprise that the execution times are extremely low. However, it is important to notice that the execution time the agent is taking is still quite fast in an industrial context, so these results are quite promising. Note that the heuristics used for comparison for these simple problems will not be able to solve complex problems. In those situations we will need more complex methods to solve the DRC-FJSSP, which will take more time for execution, just like the agent.

4.4.3 Performance Metrics

Regarding the performance metrics, it is noticeable that for the same lower bound there is a positive correlation between their values and the distance to bound, especially for the Job Dimension Percentage. Although this was expected from the way they were designed, it is positive to confirm this feature. On top of that, one can tell from the higher values in the mean and standard deviation values for the Job Difference Per-mille that this performance metric is more sensitive than the Job Dimension Percentage. However, since this last one has a higher correlation with the distance to bound, it feels more reliable and still sensitive enough to give a proper insight about the solution quality.

Both the performance metrics are not as insightful as the Distance to Bound if one wants to perceive how far we are from an optimal solution. However, the Distance to Bound can also be misleading, since the Lower Bound can be lower than the optimal makespan for the problem at hand. The main advantage of the proposed performance metrics is that their values are not dependent on the number of operations to schedule, but only on their processing times. In other words, they are not dependent on the size of the problem at hand. That is why one can tell from the data on the table that optimal or at least nearly optimal solutions will have Job Dim Percentages below 10 and that very good solutions will still score below 30. The same estimation can be done for the Job Difference Permille. All in all, these performance metrics cannot by themselves provide a thorough intuition about the solution quality, but they are definitely helpful, together with other metrics, like the Distance to Bound, into deepening one's understanding of it.

4.4.4 Worker Machine Allocation

Finally, one shall have a look at an example of schedule created by each method. These examples are given in figures 4.1, 4.2 and 4.3, where the worker allocation is represented on top and the machine allocation is represented below.

First of all, looking at the worker allocation, the bottleneck resource, all the solutions look very good quality-wise. Nevertheless, there are two interesting features noticeable from the machine allocation representation. The first of them is the distribution of operations that looks much like a wave propagation. This comes from the way the resource allocation was design. All the used methods, only deal with the sequence by which the operations are scheduled. They do not work with the resource allocation. Therefore, for all of them a Minimum Feasible Time heuristic was used to assign each operation to a machine-worker pair. Because of the way it was programmed, the heuristic will allocate an operation into the machine-worker pair that was available earlier. In case of a draw, the machine selected is the one that was available the earliest. Finally, if a draw still persists, the machine chosen for allocation is the one with the lowest index. These rules are the ones who originated that distribution of operations one can see in the machine allocation figures. Additionally, the LPT solution stands out from the other two. The operations seem much more cleanly organized. This results from the way the algorithms work and manipulate the operation sequence for scheduling. It is also the main reason why the wave propagation like shape of the machine allocation figures seems to transform into noise further away in time, for the agent and BFD solutions. Whereas, for the LPT this shape seems to be held until the end.

One final comment, relative to the machine allocation. It is possible to change the way the resource allocation was programmed, especially regarding the rules to break ties. One interesting configuration would be to continue using the Minimum Feasible Time heuristic but to always choose the machine with the smallest index, when breaking ties. This way, the solution generated would use the minimum required machines to ensure that the makespan is minimum. These different machine allocations are possible because the bottleneck resource, because of its scarcity, are the workers that operate the machines. This allows the machine allocation flexibility, that was just discussed.

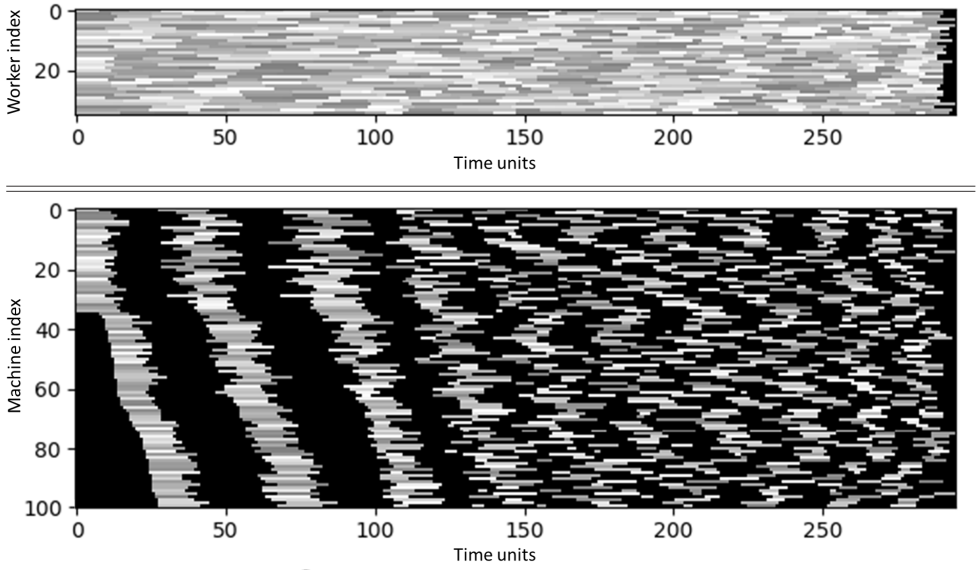

Figure 4.1: An example of a schedule created by the agent.

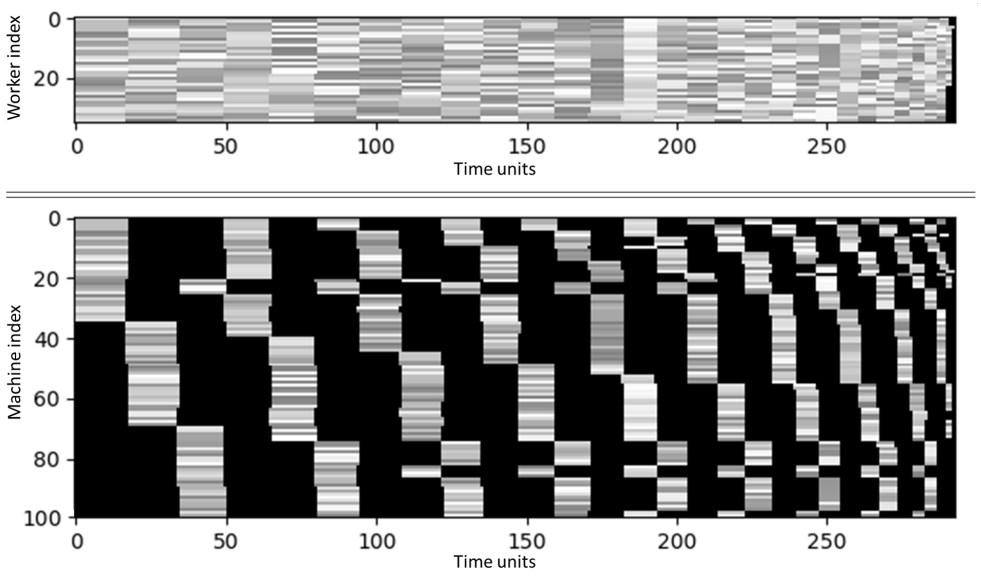

Figure 4.2: An example of a schedule created by LPT.

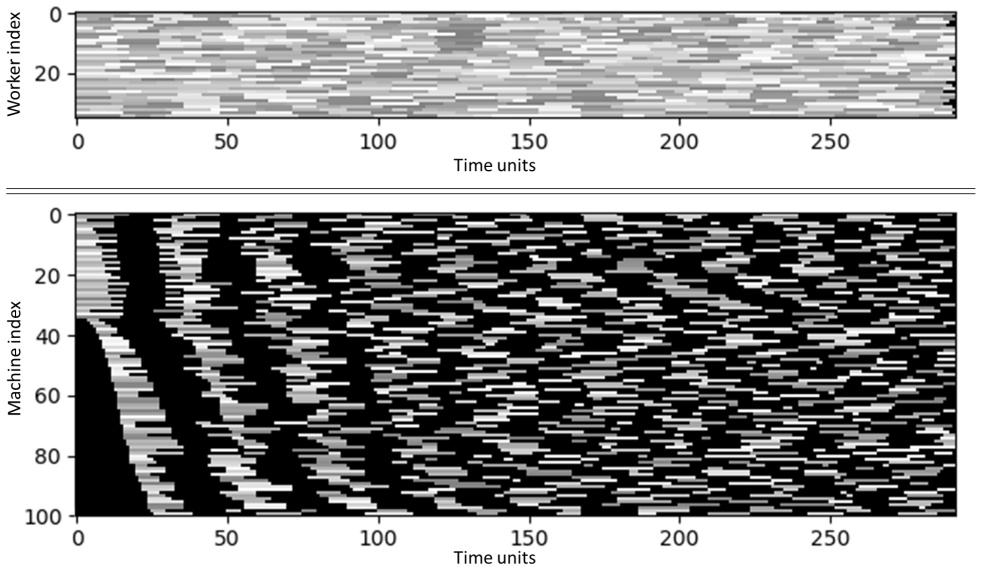

Figure 4.3: An example of a schedule created by BFD.

Chapter 5

Applying the model to a DRC-FJSSP: Benchmark Dataset

The successful simulations in chapter 4 proved that the agent was able to quickly and effectively learn how to solve a very basic version of a DRC-FJSSP. With that confidence in mind, a new agent was trained more extensively for application in a widely used benchmark dataset, the MK1-10 [43]. This benchmark adds 2 additional layers of complexity comparatively to the problems solved before. The first one of them is that jobs can now have multiples operations, that need to be executed in a precise sequence. The second one is that only a subset of the machines is eligible to execute each operation and only a subset of the workers is able to operate each machine. The KGFOA was chosen as a state-of-the-art metaheuristic for comparison.

5.1 Simulation Description and Results

The new agent was trained using the same set of parameters as before, but now it was given a higher number of training cycles for learning. The training set was kept the same but instead of the original 25, now the agent was given 1000 training cycles and the set of weights that during training achieved the lowest Job Dimension Percentage for the training instances were selected. Once again, this is an extremely low number of training cycles for training. So low that there is still no danger of overfitting. That is why the training set can still be used for validation. The ideal number of training cycles could not be computed due to a lack of computational resources.

The output selection was adapted to account with the new constraints imposed by the test dataset. Since now there are jobs with multiple operations that are required to be processed in a specific order, some moves that were previously available for the agent to choose from will now result in unfeasible solutions. It was decided that the best way to approach the situation was to avoid creating unfeasible schedules in the first place. So, the agent outputs ${n_{operations}}^2$ attention weights as before. It then evaluates if the highest attention weight represents a move that will result in feasible solution. If that is not the case, the agent moves on to the second attention weight and so on, until a feasible move is found. It is as if the agent orders all the moves according to preference and chooses the best eligible one. This by itself is not enough to ensure this constraint is respect. During the decoding process of the OSV, as the schedule is being built, the Earliest Feasible Time of an operation depends not only on the worker

and machine availability, but also on the completion time of its preceding operations. The worker and machine eligibility constraint did not result in any changes to the agent's structure. It is during the schedule construction that it is taken into account. Now only the eligible worker-machine pairs are available for allocation.

The changes made to the output selection constrain the freedom of the agent to look for optimal solutions. For example, suppose that job 1 has 2 operations. Operation 1 is in the 9-th position of the OSV, whereas operation 2 is in the 13-th position. Suppose that in an optimal configuration, operations 1 and 2 would need to be in the 2-nd and 7-th positions of the OSV. With the restriction added to the moves the agent can make, it would need to first move operation 1 to the 2-nd position and only after it would be allowed to move operation 2 to the 7-th position. If the agent would try to do it the other way around, during the intermediate step an unfeasible solution would be generated, so the agent would not be allowed to do it. Logically, a perfectly intelligent agent would be aware of this restriction and would understand that that configuration would not be allowed as a final solution. However, asking for a perfectly intelligent agent, especially for the extremely low number of training cycles computed, would be unrealistic. Also, it would need this information about the operation precedence to be fed as an input so that it could process it and make decisions from it. The input structure was not changed to keep the computational cost low. In chapter 6.2.1, a potential adaptation is discussed. All in all, for simplicity, it was considered better to not allow the agent to generate unfeasible configurations throughout the whole optimization process. The agent choice freedom will be more limited, but there will be no need to execute any arbitrary schedule repair operations.

The KGFOA algorithm was implemented as described in [21] for 1000 generations. All the above mentioned models were implemented in Python and ran in a Google Colab environment, with GPU enabled. The results of the simulation are presented in table 5.1.

5.2 Discussion of Results

5.2.1 Performance

The results are interesting. For half of the test instances (MK's 4, 5, 6, 8 and 10) the KGFOA performed considerably better than the Agent. However, for the rest of them, the results from both methods were quite competitive. The Agent was even able to surpass the KGFOA's performance for MK2. It is true that on average the KGFOA had a superior performance, but these are still impressive results given that the Agent had not been exposed to the additional constraints of the MK1-10 dataset during training. In other words, the Agent had never had to solve a problem before where jobs had multiple operations, with a predefined sequence of execution, or where there were worker-machine pairs that were not eligible for allocation. This means that a scarcely trained agent can still occasionally compete with a state-of-the-art technique when solving some problems way more complex than the ones it was trained with.

5.2.2 Execution Time

Additionally, the execution times of the Agent were more than 60 times lower than the KGFOA. One may point out that for a fair analysis, the Agent and the KGFOA results should be compared for a similar execution time. However, for the MK1-10 instances, the KGFOA execution times, even tough much higher than the Agent's, were

Table 5.1: Simulation results of the Agent and the KGFOA for the MK1-10.

	MK1		MK2	
	Agent	KGFOA	Agent	KGFOA
Job Dimension (%)	319.07	**162.79**	**109.88**	231.98
Job Difference (‰)	1577.47	**804.83**	**666.10**	1406.21
Min Finish time (units)	53	55	66	71
Max Finish time (units)	66	**60**	**69**	74
Lower Bound (units)	54	54	65	65
Distance to Bound (%)	22.22	**11.11**	**6.15**	13.85
Execution Time (s)	**1.26**	65.46	**1.34**	75.27
	MK3		MK4	
	Agent	KGFOA	Agent	KGFOA
Job Dimension (%)	1523.67	**734.71**	640.80	**537.36**
Job Difference (‰)	2537.50	**1223.58**	2463.10	**2065.47**
Min Finish time (units)	285	311	107	114
Max Finish time (units)	408	**328**	129	**123**
Lower Bound (units)	254	254	92	92
Distance to Bound (%)	60.63	**29.13**	40.22	**33.70**
Execution Time (s)	**3.02**	262.98	**1.85**	131.87
	MK5		MK6	
	Agent	KGFOA	Agent	KGFOA
Job Dimension (%)	683.17	**328.45**	1637.44	**992.65**
Job Difference (‰)	9156.44	**4402.13**	5347.98	**3242.04**
Min Finish time (units)	341	324	121	155
Max Finish time (units)	359	**332**	207	**169**
Lower Bound (units)	307	307	111	111
Distance to Bound (%)	16.94	**8.14**	86.49	**52.25**
Execution Time (s)	**2.23**	202.06	**2.93**	282.48
	MK7		MK8	
	Agent	KGFOA	Agent	KGFOA
Job Dimension (%)	405.03	**339.85**	1352.33	**949.40**
Job Difference (‰)	616.76	**517.51**	3612.11	**2535.86**
Min Finish time (units)	281	290	646	586
Max Finish time (units)	312	**305**	688	**637**
Lower Bound (units)	269	269	517	517
Distance to Bound (%)	15.99	**13.38**	33.08	**23.21**
Execution Time (s)	**2.75**	142.46	**4.61**	534.35
	MK9		MK10	
	Agent	KGFOA	Agent	KGFOA
Job Dimension (%)	1026.24	**933.42**	1304.43	**967.27**
Job Difference (‰)	3179.76	**2892.14**	3782.19	**2804.60**
Min Finish time (units)	627	636	449	465
Max Finish time (units)	667	**655**	528	**487**
Lower Bound (units)	535	535	370	370
Distance to Bound (%)	24.67	**22.43**	42.70	**31.62**
Execution Time (s)	**4.84**	549.72	**4.74**	486.88

still within a reasonable range. However, for larger problems, like the ones in chapter 4, that probably would not be the case anymore and so the Agent's celerity could come as a decisive advantage.

There is another feature of the Agent worth highlighting, relative to the execution time of the optimization process. As mentioned in section 3.2.2, the agent can put an end to the optimization process if it identifies the current schedule as being fully optimized. This skill is supposed to be acutely develop at least for a fully trained agent. However, for the scarcely trained agent being tested, it shall not come as surprise that most of the times it fails to recognize that a schedule is fully optimized. It would already be remarkable if it could even achieve such a schedule. More often than not, the agent will finish the optimization process when it reaches the maximum number of moves stop criterion. Therefore, it is expected that the execution times sharply decrease for a thoroughly trained agent.

5.2.3 Performance Metrics

A final word about the performance metrics proposed in this thesis. Both the Job Dimension Percentage and the Job Difference Per-mille value ranges rose a lot, just as it happened for the Distance to Bound. That is natural to happen due to the additional constraints that these solutions are subject to, comparatively with the ones from chapter 4. The Job Difference Per-mille shows some instability, as it sometimes escalates a bit to much. Take MK5 as an example, where this metric overwhelming values suggest that incredibly bad solutions were produced by both methods when in fact they were quite acceptable, if compared to the other instances results. The Job Dimension Percentage behaviour seems to be much more coherent with the quality of the solutions being evaluated. Once again, there seems to exist a positive correlation between the Job Dimension Percentage and the Distance to Bound, but a much weaker one than observed in chapter 4. For example, a similar Job Dimension Percentage evaluation of the KGFOA performance for MK6 and MK9 is matched with extremely different Distance to Bound results. All in all, the absolute and relative Distance to Bound metrics seem to give more acute intuition and evaluation of the solution quality than the Job Dimension Percentage and the Job Difference Per-mille. Nevertheless, the Job Dimension Percentage still looks accurate at evaluating the quality of solutions, particularly when comparing different solutions of the same problem. The Job Difference Per-mille instability reduces the confidence on its measurements, but it is still useful most of the times.

5.2.4 Worker Machine Allocation

At last, an example of schedule created by each method is given in figures 5.1 and 5.2, where the worker allocation is represented on top and the machine allocation is represented below.

Looking at the figures, one can confirm that both methods produce good quality results and that the KGFOA performs sightly better. Both this conclusions were already evident in the results at table 5.1. There is however a considerable difference at the machine allocation figures, comparing with the ones at figures 4.1 and 4.2. Back then, it was pointed out that those results were achieved using the Minimum Feasible Time heuristic to manage the resource allocation. It was said that in case of a draw different rules could be applied for breaking the tie. That time the preference was to allocate operations to the machines that had been idle for a longer time. This time, in order to point out the difference, the operations were allocated to the earliest available machine with the

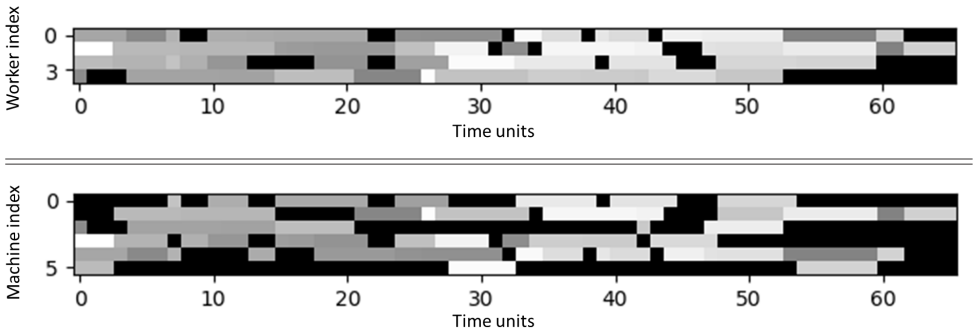

Figure 5.1: The schedule created by the Agent for MK1.

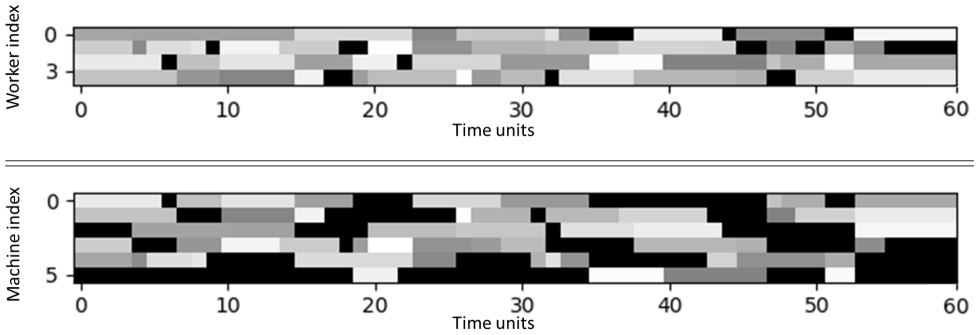

Figure 5.2: The schedule created by KGFOA for MK1.

smallest index number. The biggest benefit of this rule is that one only uses the minimum number of machines required to achieve the solution's makespan. The biggest drawback is that some machines are being much more used than others and this unbalance can originate breakdown problems. It is advisable to balance the workload of the available machines. So, the tiebreak rules used in chapter 4 are more appropriate for application in the industry. However, for simulation purposes, if the factory is being designed or one is considering buying new machines, the tiebreak rules used in this chapter will be very helpful to estimate how many machines are needed to optimally solve the problem at hand, for a specified number of workers. For the problems in chapter 4, the usage of this tiebreak rules would generate trivial machine allocation figures. If any machine-worker pair is eligible for every operation, it is with no surprise that the machine allocation figure is equal to the worker allocation figure. However, in this chapter that is not the case, and the generated solutions prove that all the available machines are required in order to achieve this makespan results.

Chapter 6

Conclusions

6.1 Achievements

The research conducted on this master thesis allowed the development of a novel Deep Q-Learning method for solving the DRC-FJSSP. The idea of applying an Encoder-Decoder NN architecture with an Attention Mechanism, typically used when solving Neural Language Processing problems, really proved to work. Even with just a scarce number of training cycles across a very basic training dataset, the agent designed was able to achieve some competitive results comparatively with a state-of-the-art meta-heuristic when applied to a benchmark dataset. The agent proved to be flexible in dealing with some real-world Industry constraints and was able to generate the solutions in a reasonable amount of time, dozens of times faster than the KGFOA.

The most significant drawback of this implementation has to be the computational effort required to train the NN. The major bottleneck of the work developed in this thesis was indeed the lack of adequate computational resources. This heavy computational requirements are typical in the development of Deep Learning systems. However, as pointed out in the introduction of this thesis, the computational power available in the market has been growing exponentially. If one or two decades ago the amount of computational power required to train a Deep NN would only be available to the market's technological giants, nowadays an average startup budget would probably be enough to invest in those resources. Moreover, the advent of cloud computing diminished even more the price of access to those resources. If that would not be enough, just recently Chinese scientists announced a technological breakthrough, claiming that they had developed a quantum computer capable of performing certain computations nearly 100 trillion times faster than the world's most advanced supercomputer [44]. All in all, the rate at which the computational power available for everyone is growing does not seem to decay, so it should not be long until it becomes easy for everyone to train a Deep NN in their own computer. But even now the required computational power should be available at a reasonable budget.

The two performance metrics proposed for evaluating the solution quality of a DRC-FJSSP proved to be insighful. Nevertheless, they were not found to be suitable for replacing any of the existing metrics like the makespan and the absolute and relative Distance to Bound. They still proved to be nice to have in addition to those metrics, especially the Job Dimension Percentage. On the other hand, the Job Difference Per-mille was found to be a bit unstable, which ends up reducing the confidence on its measurements.

6.2 Future Work

6.2.1 In depth Agent training and additional features

It was impossible to evaluate the full capabilities of the proposed solution due to the lack of computational resources. The complex neural network architecture developed would need to be exposed to at least hundreds of different problems and hundreds of thousands of training cycles, before one could consider it properly trained. Therefore, before trying different approaches it would be interesting to evaluate how well can this model operate after a thorough training procedure and compare the results obtained with the state-of-the-art techniques.

For the application in the Industry to be possible, it is possible that some additional constraints need to be taken into consideration. One common example is for the processing times of the operations to vary with the allocated resource pair. The input given to the agent in the proposed method are insufficient to take that into consideration. The only way the current agent could solve these problems would be to consider a standard processing time for each operation, independent of the allocated resource pair. This processing time would then be switched by the correct one during the vector decoding and schedule building procedure, but it would be impossible for the agent to somehow interpret what was happening, since it would be possible for 2 problems with the same input and output to generate different rewards. It is true that the same happened when jobs with multiple operations and machine and worker eligibility constraints were added. The agent proved to be capable of still delivering partially optimized solutions, but their quality was clearly deprecated. In order to unleash the agent from this limitation, more information about the problem needs to be given as an input.

Additionally, the proposed method only deals with the optimization of the operation sequencing for scheduling. The resource allocation is not dealt with by the agent and was solved using the Earliest Feasible Time heuristic. This can also come as a limitation in the agent's optimization capabilities. It would be possible to address the machine and worker allocation by redesigning the network's output.

In order to fulfil these 2 needs, a different approach to the input and output of the neural network was conceptualized and could be developed in future research. Regarding the input, the goal is to provide not only the processing time and position in the OSV, but also the worker and machine eligibility and allocation information. This new information can be summarized in a $(n_{machines} \times n_{workers})$ by $n_{operations}$ matrix, like shown in figure 6.1.

The input vector dimension as to be constant for every encoder step. The dimensions of all the information matrices and vectors shown in figure 6.1 vary from problem to problem. Therefore, the only way to sequentially feed this input information in constant sized vectors is by feeding these values one by one. The information is fed

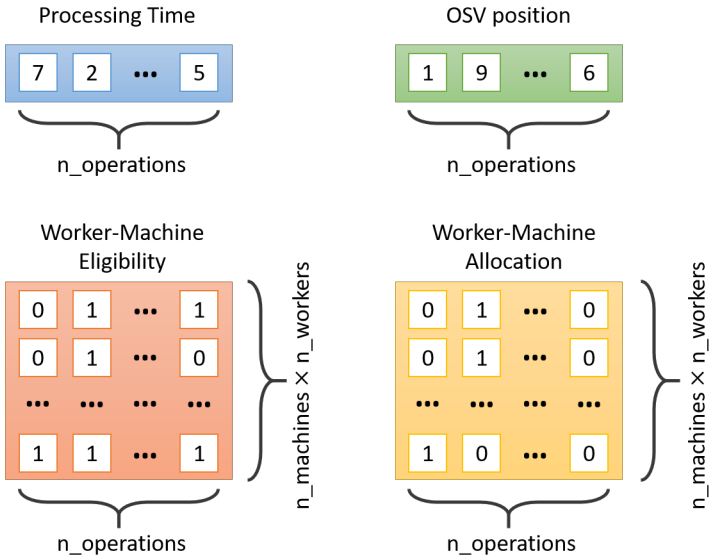

Figure 6.1: Processing Time, OSV position and Worker-Machine eligibility and allocation input information.

according to algorithm 3, as represented in figure 6.2.

Algorithm 3: Network input information feeding process algorithm.

Processing Time vector, $proc\_time$;

OSV position vector, OSV_pos ;

Worker-Machine Eligibility matrix, $Work\_Mach\_Elig$;

Worker-Machine Allocation matrix, $Work\_Mach\_Alloc$;

for $i = 1, ..., n_{machines} \times n_{workers}$ **do**

  for $j = 1, ..., n_{operations}$ **do**

    $input(i,j) = [proc\_time(j), OSV\_pos(j), Work\_Mach\_Elig(i,j), Work\_Mach\_Alloc(i,j)]$

  end

end

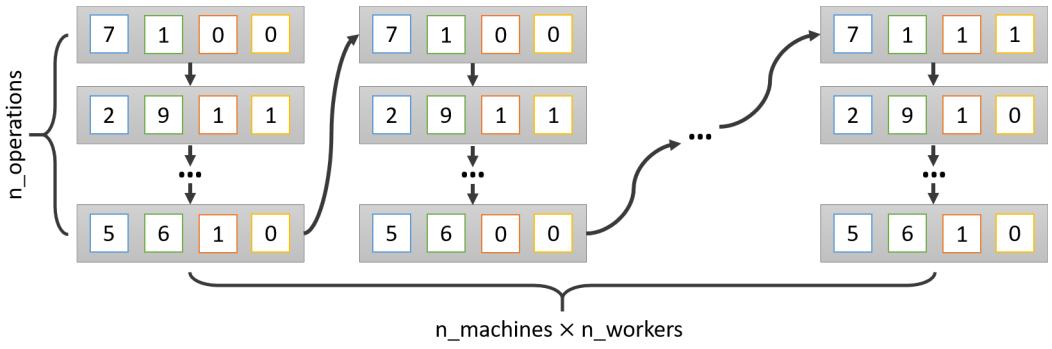

Figure 6.2: Network input information feeding process representation.

What is happening is that the agent first receives the operation information regarding the first resource pair. For the first operation: what is its processing time? What is its position in the OSV? Is this operation eligible to be allocated to this resource pair? Is this operation allocated to this resource pair? Then, it proceeds until this was

done for every operation. At the end of this loop, it moves on to the next resource pair. It then reads once again one by one the operation information for this resource pair. And it continues until the information of all the operations regarding every resource pair was read.

It is worth highlighting that both the processing times and the OSV positions are vectors, whose information is being repeatedly fed to the network. This repetition comes from the fact that the Worker-Machine Eligibility and the Worker-Machine Allocation information are matrices with a number of elements $n_{machines} \times n_{workers}$ times larger than the aforementioned vectors. Also, if the processing times of the operations vary according to the resource allocation, it becomes a $(n_{machines} \times n_{workers})$ by $n_{operations}$ matrix itself and, so, that information is not repeatedly fed.

Another aspect important to refer is that it was considered that when a worker is not eligible to operate a machine, then that resource pair is not eligible to execute any operation. Trivially, if a machine is not eligible to execute an operation, then the any resource pair with that machine is not eligible to execute that operation.

All in all, the agent can now receive the current OSV, MAV and WAV together with the processing times and Worker-Machine Eligibility information. It has now all the inputs it needs to make an informed decision regarding the operation sequencing and resource allocation optimization. However, the output structure still needs to be adapted to incorporate the resource allocation commands.

With the changes made to the input structure, the intuition that led to the use the attention weights as outputs, like in a Pointer Network, is lost, since the inputs are no longer single operations. This way, the decoder outputs will be used for the move operation and resource allocation selection. The number of operations and resource pairs varies from problem to problem. Therefore, the only way that the output vector can have a constant dimension is by being composed of a single element. All in all, there will be $n^2_{operations} \times n_{machines} \times n_{workers}$ decoder steps. Each one of them outputs a score and the highest score of them all is selected. Considering *idx* as the index of the highest score, the output command is given by equation 6.1.

$$ResourcePair\_num = floor(\frac{idx-1}{n_{machines} \times n_{workers}}) + 1, \tag{6.1a}$$

$$Position\_num = floor(\frac{idx - (ResourcePair\_num \times n_{machines} \times n_{workers}) - 1}{n_{operations}}) + 1 \tag{6.1b}$$

$$Operation\_num = idx - (ResourcePair\_num \times n_{machines} \times n_{workers}) - (Position\_num \times n_{operations}) \tag{6.1c}$$

where the floor function rounds the its input to the closest smaller integer. The agent moves operation number $Operation\_num$ to position number $Position\_num$ in the OSV. Additionally, it allocates that operation to resource pair number $ResourcePair\_num$.

6.2.2 Redesigned Implementation

The adaptations made in section 6.2.1 lead to a very computationally demanding technique. The major reason for that is that the number of encoder and decoder steps escalated with the changes made to the input and output structure. So, with the knowledge acquired throughout the work in this thesis, is there anything that could have been done differently and, perhaps, more effectively?

Like discussed in chapter 6.2.1, in order to fully autonomously solve the DRC-FJSSP, the agent needs to receive information regarding the processing times of operations and the worker-machine eligibility. Additionally, since a step by step optimization approach was taken, it also needs to receive information about the current OSV, MAV and WAV states. Looking back on why this approach was taken, the justification lied on the assumption that it would be probably to optimistic to design an intelligent agent capable of optimizing a DRC-FJSSP at one go. However, the only way to find it out is by actually trying it. And looking at the computational advantages that reducing the number of inputs would have, it looks worth a shot.

So a new input structure for the Encoder-Decoder architecture was conceptualized that contemplated only one $(n_{machines} \times n_{workers})$ by $n_{operations}$ matrix to be fed as an input. That matrix carried the processing times of each operation if allocated to each of the available resource pairs. The processing time would be 0 if that combination of operation and resource pair is not eligible. The elements of this matrix would be fed one by one at each encoder step.

The information would then go through the network and the output would be once again defined by the attention weights as in a Pointer Network. However, this time a Local Attention Mechanism would be used. The difference between Local and Global Attention is that for Global Attention all attention weights are calculated, whereas in Local Attention only the relevant ones are calculated. The relevant weights would be the ones pointing towards eligible operation-resource elements in the input. An eligible operation-resource element is one with a processing time greater than 0 and whose preceding operations have all been selected.

The OSV, MAV and WAV are sequentially built along the decoder steps. At each decoder step, the highest local attention weight is select, which defines the operation chosen to be added to the OSV, as well as the machine and worker (resource) allocation, stored in the MAV and the WAV, respectfully. In the end, all 3 vectors all fully constructed and the solution schedule can be decoded from them.

Bibliography

[1] B. Marr. What is industry 4.0? here's a super easy explanation for anyone, 09 2018. URL https://www.forbes.com/sites/bernardmarr/2018/09/02/what-is-industry-4-0-heres-a-super-easy-explanation-for-anyone/#566f10cc9788. Accessed at 28/10/2020.

[2] B. In. What is artificial intelligence? URL https://builtin.com/artificial-intelligence. Accessed at 18/11/2020.

[3] M. Morales. *Grokking Deep Reinforcement Learning*. Manning Publications, 2020. ISBN 9781617295454. URL https://books.google.pt/books?id=IpHJzAEACAAJ.

[4] T. G. W. C. S. F. Roman Hipp, Christian Fiebig. Automation of production planning enhanced by ai, 2019.

[5] G. Georgiadis, A. Elekidis, and M. Georgiadis. Optimization-based scheduling for the process industries: From theory to real-life industrial applications. *Processes*, 7:438, 07 2019. doi: 10.3390/pr7070438.

[6] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, 5th edition, 2016. ISBN 978-3-319-26580-3. doi: 10.1007/978-3-319-26580-3.

[7] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. 1(2):117–129, May 1976. ISSN 0364-765X. doi: 10.1287/moor.1.2.117.

[8] M. Cunha. Scheduling of flexible job shop problem in dynamic environment. Master's thesis, Instituto Superior Técnico, 2017.

[9] L. Gao and Q.-K. Pan. A shuffled multi-swarm micro-migrating birds optimizer for a multi-resource-constrained flexible job shop scheduling problem. *Information Sciences*, 372, 08 2016. doi: 10.1016/j.ins.2016.08.046.

[10] J. Li, Y. Huang, and X. Niu. A branch population genetic algorithm for dual-resource constrained job shop scheduling problem. *Computers Industrial Engineering*, 102, 10 2016. doi: 10.1016/j.cie.2016.10.012.

[11] B. Dohmen. Is intuition a form of intelligence? URL https://www.forbes.com/sites/investor/2018/06/23/is-intuition-a-form-of-intelligence/. Accessed at 19/11/2020.

[12] F. Xhafa and A. Abraham. *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 3540789847.

[13] P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Appl. Math.*, 49(1–3):107–127, Mar. 1994. ISSN 0166-218X. doi: 10.1016/0166-218X(94)90204-6. URL https://doi.org/10.1016/0166-218X(94)90204-6.

[14] W.-Y. Ku and J. Beck. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers Operations Research*, 73, 04 2016. doi: 10.1016/j.cor.2016.04.006.

[15] M. Dhiflaoui, H. E. Nouri, and O. B. Driss. Dual-resource constraints in classical and flexible job shop problems: A state-of-the-art review. *Procedia Computer Science*, 126:1507 – 1515, 2018. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2018.08.123. URL http://www.sciencedirect.com/science/article/pii/S1877050918314017. Knowledge-Based and Intelligent Information Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia.

[16] H. Lehtihet. What are the differences between heuristics and metaheuristics?, Accessed October 8th 2020. URL https://www.researchgate.net/post/What_are_the_differences_between_heuristics_and_metaheuristics.

[17] R. Wu, Y. Li, S. Guo, and W. Xu. Solving the dual-resource constrained flexible job shop scheduling problem with learning effect by a hybrid genetic algorithm. *Advances in Mechanical Engineering*, 10(10), 2018. doi: 10.1177/1687814018804096. URL https://doi.org/10.1177/1687814018804096.

[18] J. Zhang, W. Wang, and X. Xu. A hybrid discrete particle swarm optimization for dual-resource constrained job shop scheduling with resource flexibility. *Journal of Intelligent Manufacturing*, 28, 04 2015. doi: 10.1007/s10845-015-1082-0.

[19] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, Sept. 2003. ISSN 0360-0300. doi: 10.1145/937503.937505. URL https://doi.org/10.1145/937503.937505.

[20] D. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, Menlo Park, CA, 1988.

[21] X.-l. Zheng and L. Wang. A knowledge-guided fruit fly optimization algorithm for dual resource constrained flexible job-shop scheduling problem. *International Journal of Production Research*, 54:1–13, 03 2016. doi: 10.1080/00207543.2016.1170226.

[22] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

[23] A. Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017. ISBN 1491962291.

[24] S. Sharma. What the hell is perceptron? URL https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53. Accessed at 17/12/2020.

[25] S. Schuchmann. History of the first ai winter. URL https://towardsdatascience.com/history-of-the-first-ai-winter-6f8c2186f80b. Accessed at 17/12/2020.

[26] T. Stöttner. Why data should be normalized before training a neural network, 05 2019. URL https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d. Accessed at 29/10/2020.

[27] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[28] M. A. Nielsen. *Neural Networks and Deep Learning*. 2018.

[29] MathWorks. What is deep learning? 3 things you need to know. URL https://www.mathworks.com/discovery/deep-learning.html. Accessed at 27/10/2020.

[30] Y. Li. Deep reinforcement learning: An overview, 2018.

[31] M. Alzantot. Deep reinforcement learning demysitifed (episode 2) — policy iteration, value iteration and q-learning. URL https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e
Accessed at 08/12/2020.

[32] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, Oct. 2017. URL http://dx.doi.org/10.1038/nature24270.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 14764687. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

[34] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2016.

[35] E. Even-dar and Y. Mansour. Learning rates for q-learning. *Journal of Machine Learning Research*, 5, 04 2001. doi: 10.1007/3-540-44581-1_39.

[36] N. Garg. What is the vanishing gradient problem? URL https://www.quora.com/What-is-the-vanishing-gradient-problem. Accessed at 04/11/2020.

[37] Oinkina. Understanding lstm networks. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed at 03/11/2020.

[38] S. Kostadinov. Understanding gru networks. URL https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be. Accessed at 04/11/2020.

[39] R. Karim. Attn: Illustrated attention. URL https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3. Accessed at 07/11/2020.

[40] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2014. URL http://arxiv.org/abs/1409.0473.

[41] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, page 2692–2700, 2015.

[42] W. Zhang and T. Dietterich. Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. 06 2001.

[43] P. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41:157–183, 1993.

[44] S. Chen. Chinese scientists claim breakthrough in quantum computing race. URL https://www.bloomberg.com/news/articles/2020-12-04/chinese-scientists-claim-breakthrough-in-quantum-computing-race. Accessed at 31/12/2020.

Appendix A

Move Operation

The move operation and the necessary number of moves at each episode is detailed in this appendix. The proposed solution in this thesis for solving the DRC-FJSSP is based in elementary move operations on the OSV. An unknown number of these moves is done, until the schedule is finally optimized. As mentioned in section 3.2.2, a rule was established to put an end to the agent's optimization process. A schedule is considered to be optimized when the agent chooses a move that does not change the OSV. When that happens, the best schedule found so far, the one with the lowest makespan, is considered to be the optimized solution. Additionally, the agent only has a limited number of moves to optimize the solution, equal to 50% of the number of operations to schedule for the problem at hand. After those moves, the best schedule found so far, the one with the lowest makespan, is considered to be the optimized solution.

However, during training the main goal is not to find an optimized solution, but rather for the agent to learn. Since the ε-greedy algorithm was used, a random move is done with probability ε. The probability ε decays throughout the training cycles. All in all, during a training episode the agent shall not stop when the solution is optimized. Another rule shall be established to finalize the episode.

First of all, one should have a look at how the move operation works. In figure A.1 there is an example of this operation. The letters A to E represent different operations and the OSV_pos numbers represent their position in the OSV. The green arrow represents the move operation. The changes in the OSV instigated by the move operation are done according to algorithm 4.

	A	B	C	D	E
OSV_pos (t)	3	1	0	4	2
OSV_pos (t+1)	1	+1 2	0 0	0 4	+1 3
OSV_pos (t+2)	-1 0	-1 1	3	0 4	-1 2

Figure A.1: Move operation and its impacts on the OSV.

Algorithm 4: The OSV manipulation algorithm for the move operation.

Read the current operation sequence, OSV_t ;

Read i, the operation selected to be moved ;

Read j, the new position of the operation i in OSV_{t+1} ;

if $OSV_t(i) = j$ **then**

  $OSV_{t+1} = OSV_t$;

else if $OSV_t(i) > j$ **then**

  for *operation = 1, ..., n* **do**

    if *operation* = i **then**

      $OSV_{t+1}(operation) = j$;

    else if $j \leq OSV_t(operation) < OSV_t(i)$ **then**

      $OSV_{t+1}(operation) = OSV_t(operation) + 1$;

    else

      $OSV_{t+1}(operation) = OSV_t(operation)$;

    end

  end

else if $OSV_t(i) < j$ **then**

  for *operation = 1, ..., n* **do**

    if *operation* = i **then**

      $OSV_{t+1}(operation) = j$;

    else if $OSV_t(i) < OSV_t(operation) \leq j$ **then**

      $OSV_{t+1}(operation) = OSV_t(operation) - 1$;

    else

      $OSV_{t+1}(operation) = OSV_t(operation)$;

    end

  end

end

Even though algorithm 4 is quite trivial, its dynamics, especially after a few moves, can become extremely complex. This happens because moved operations do not become locked in a position for all future moves. In figure A.1 for example, in the first move operation A goes to position number 1 in OSV. However, because in the second move operation C goes from position 0 to position 3, operation A is brought to position 0. When one operation is moved, there is a whole set of other operations that are relocated as well. And that is where the complexity comes from.

The goal when calculating the necessary number of moves per episode is to allow that during the episode any possible configuration can be reached, despite of what the original one was. In order to be able to make this calculation, one shall first get comfortable with how the OSV changes when multiple move operations are applied. Consider a simple example where there are 3 operations to schedule: A, B and C. Consider A-B-C as the initial configuration. In figures A.2 to A.5 all the possible configurations with 2 moves are presented. The arrow letters

represent which operations have been moved. Additionally, an overview of this process is presented in figure A.6.

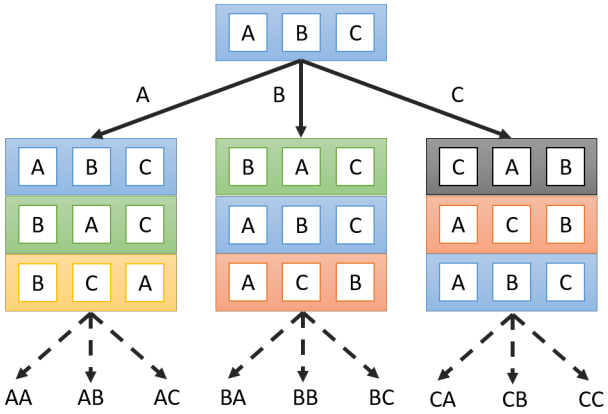

Figure A.2: Possible operation sequence configurations after 1 move.

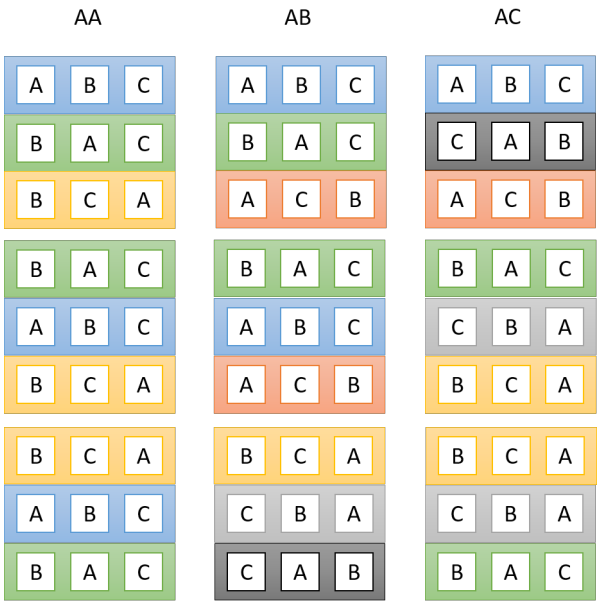

Figure A.3: Possible operation sequence configurations after 2 moves, knowing that operation A was moved before.

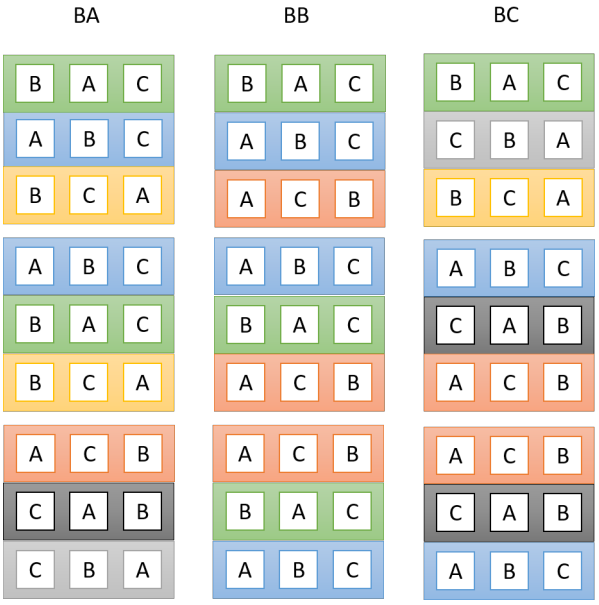

Figure A.4: Possible operation sequence configurations after 2 moves, knowing that operation B was moved before.

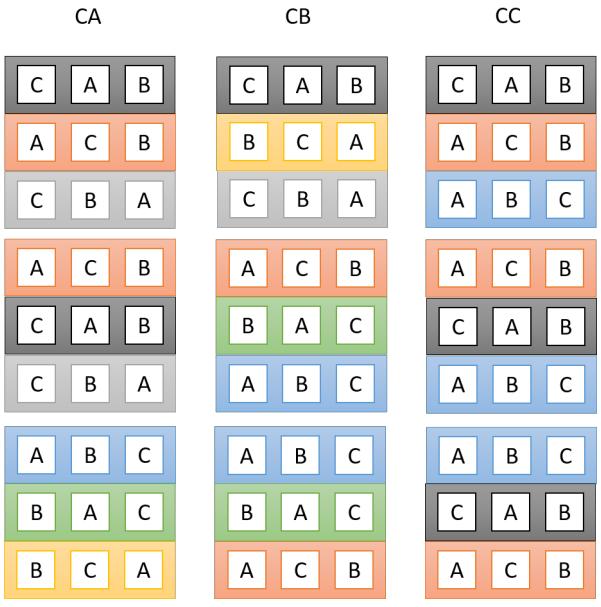

Figure A.5: Possible operation sequence configurations after 2 moves, knowing that operation C was moved before.

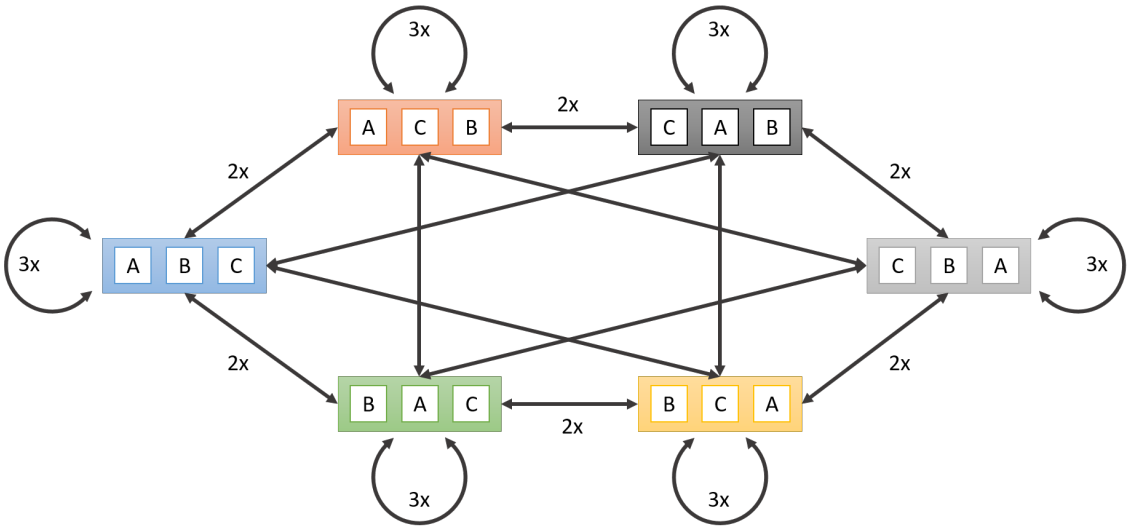

Figure A.6: An overview of the move operation for a problem with 3 operations.

From this, one can conclude that, for 3 operations, at least 2 moves are needed in order to achieve any possible configuration. In fact, as shown in table A.1, the minimum number of moves necessary to arrive at any possible configuration, #M, as a function of the number of operations to schedule, #O, is always $n-1$. Also, the probability of ending up on each configuration starting from A-B-C can be calculated and is given in figure A.7.

Table A.1: The minimum number of moves to arrive at any possible configuration.

#O	**#M**
1	0
2	1
3	2
4	3
...	...
n	n-1

State	#	Probability
A B C	19	24%
A C B	16	20%
B A C	16	20%
B C A	11	14%
C A B	11	14%
C B A	8	10%

Figure A.7: Probability of ending up on each state after 2 moves, having started as A-B-C.

As one can see, these probabilities are clearly unbalanced, which means that any random moves selected by the agent would be affected by a strong bias. Intuitively one can understand, that when the number of moves grows the initial state becomes less influential in the randomly selected output, which means that this probabilities are expected to get balanced as the number of moves grows. The number of moves that balances these probabilities

is expected to be positively correlated with the number of operations to schedule. For simplicity, it was supposed that there is approximately a linear proportionality between this values, as exposed in equation A.1. The k value represents the proportionality constant and needs to be estimated.

$$n_{moves} = k \times (n_{operations} - 1) \tag{A.1}$$

The probabilities in figure A.7 were calculated mathematically. Due to the law of large numbers, one could have estimated this probability using a simulation where this random choice is executed thousands of times. This will be especially useful for larger problems, for which the number of possibilities grows exponentially. These simulations were ran for problems with 3 and 4 operations to schedule and for a natural number k between 1 and 5. For each of these, $1000 \times (n^2)^{n-1} \times (n-1) \times k$ experiences were made. n^2 is the number of possible moves and $(n^2)^{n-1}$ is the number of possible routes for n-1 moves. This means that each route is followed on average $1000 \times (n-1) \times k$ times, ignoring that the number of routes varies with the number of moves. The results of these simulations are presented in table A.2.

Table A.2: Maximum probability percentage deviation between states, for two move operations.

		$n_{operations}$	
		3	**4**
k	**1**	13.45	7.44
	2	2.48	1.57
	3	0.87	0.35
	4	0.10	0.10
	5	0.10	0.00

The result obtained for $k = 1$ and $n_{operations} = 3$ is according to what was mathematically calculated before. The idea is to choose the lowest possible number of moves that sufficiently approximate the probabilities of arriving at each possible configuration, since the higher the number of moves the more expensive the computational effort for training becomes. From the results in A.2, it was considered that, with a maximum probability percentage deviation lower than 1%, $3 \times (n_{operations} - 1)$ moves would be sufficient for removing the bias in the random move choice selection. However, for simplicity, the number of moves per episode implemented was $3 \times n_{operations}$.

Appendix B

Full Simulation Results

The full results from the chapter 4 simulation are presented in the following tables.

Table B.1: Simulation results for the problems with 10 jobs to schedule.

Sample 1	Agent	LPT	BFD
Job Dimension (%)	0.00	11.11	0.00
Job Difference (‰)	0.00	24.73	0.00
Min Finish time (units)	45	44	45
Max Finish time (units)	45	46	45
Lower Bound (units)	45	45	45
Distance to Bound (%)	0.00	2.22	0.00
Execution Time (s)	1.22	0.00	0.00
Sample 2	Agent	LPT	BFD
Job Dimension (%)	6.02	6.02	18.07
Job Difference (‰)	15.40	15.40	46.20
Min Finish time (units)	41	41	40
Max Finish time (units)	42	42	43
Lower Bound (units)	42	42	42
Distance to Bound (%)	0.00	0.00	2.38
Execution Time (s)	1.02	0.00	0.02
Sample 3	Agent	LPT	BFD
Job Dimension (%)	0.00	0.00	17.86
Job Difference (‰)	0.00	0.00	39.89
Min Finish time (units)	56	56	54
Max Finish time (units)	56	56	58
Lower Bound (units)	56	56	56
Distance to Bound (%)	0.00	0.00	3.57
Execution Time (s)	1.02	0.02	0.00
Sample 4	Agent	LPT	BFD
Job Dimension (%)	9.62	0.00	9.62
Job Difference (‰)	47.67	0.00	47.67
Min Finish time (units)	51	52	51
Max Finish time (units)	53	52	53
Lower Bound (units)	52	52	52
Distance to Bound (%)	1.92	0.00	1.92
Execution Time (s)	1.02	0.00	0.02
Sample 5	Agent	LPT	BFD
Job Dimension (%)	5.26	5.26	26.32
Job Difference (‰)	14.02	14.02	70.09
Min Finish time (units)	47	47	45
Max Finish time (units)	48	48	50
Lower Bound (units)	48	48	48
Distance to Bound (%)	0.00	0.00	4.17
Execution Time (s)	1.00	0.02	0.00

Table B.2: Simulation results for the problems with 30 jobs to schedule.

Sample 1	Agent	LPT	BFD
Job Dimension (%)	10.71	10.71	10.71
Job Difference (‰)	28.62	28.62	28.62
Min Finish time (units)	54	55	54
Max Finish time (units)	57	57	57
Lower Bound (units)	56	56	56
Distance to Bound (%)	1.79	1.79	1.79
Execution Time (s)	9.48	0.00	0.02
Sample 2	Agent	LPT	BFD
Job Dimension (%)	9.52	9.52	9.52
Job Difference (‰)	34.73	34.73	34.73
Min Finish time (units)	62	62	61
Max Finish time (units)	64	64	64
Lower Bound (units)	63	63	63
Distance to Bound (%)	1.59	1.59	1.59
Execution Time (s)	9.36	0.02	0.02
Sample 3	Agent	LPT	BFD
Job Dimension (%)	2.15	2.15	12.90
Job Difference (‰)	5.62	5.62	33.70
Min Finish time (units)	55	55	55
Max Finish time (units)	56	56	57
Lower Bound (units)	56	56	56
Distance to Bound (%)	0.00	0.00	1.79
Execution Time (s)	9.31	0.02	0.00
Sample 4	Agent	LPT	BFD
Job Dimension (%)	14.46	14.46	5.42
Job Difference (‰)	33.92	33.92	12.72
Min Finish time (units)	65	65	64
Max Finish time (units)	68	68	67
Lower Bound (units)	67	67	67
Distance to Bound (%)	1.49	1.49	0.00
Execution Time (s)	9.31	0.00	0.02
Sample 5	Agent	LPT	BFD
Job Dimension (%)	16.82	7.48	7.48
Job Difference (‰)	46.75	20.78	20.78
Min Finish time (units)	61	63	63
Max Finish time (units)	66	65	65
Lower Bound (units)	65	65	65
Distance to Bound (%)	1.54	0.00	0.00
Execution Time (s)	9.27	0.02	0.02

Table B.3: Simulation results for the problems with 100 jobs to schedule.

Sample 1	Agent	LPT	BFD
Job Dimension (%)	22.84	2.98	2.98
Job Difference (‰)	60.87	7.94	7.94
Min Finish time (units)	96	100	99
Max Finish time (units)	103	101	101
Lower Bound (units)	101	101	101
Distance to Bound (%)	1.98	0.00	0.00
Execution Time (s)	111.26	0.05	0.05
Sample 2	Agent	LPT	BFD
Job Dimension (%)	17.79	7.91	7.91
Job Difference (‰)	45.35	20.16	20.16
Min Finish time (units)	99	101	100
Max Finish time (units)	103	102	102
Lower Bound (units)	102	102	102
Distance to Bound (%)	0.98	0.00	0.00
Execution Time (s)	111.29	0.06	0.05
Sample 3	Agent	LPT	BFD
Job Dimension (%)	17.58	8.33	8.33
Job Difference (‰)	64.01	30.32	30.32
Min Finish time (units)	104	107	106
Max Finish time (units)	110	109	109
Lower Bound (units)	109	109	109
Distance to Bound (%)	0.92	0.00	0.00
Execution Time (s)	111.12	0.06	0.05
Sample 4	Agent	LPT	BFD
Job Dimension (%)	14.78	4.93	4.93
Job Difference (‰)	51.25	17.08	17.08
Min Finish time (units)	99	101	1000
Max Finish time (units)	103	102	102
Lower Bound (units)	102	102	102
Distance to Bound (%)	0.98	0.00	0.00
Execution Time (s)	111.11	0.05	0.06
Sample 5	Agent	LPT	BFD
Job Dimension (%)	11.79	1.96	11.79
Job Difference (‰)	33.85	5.64	33.85
Min Finish time (units)	100	101	99
Max Finish time (units)	103	102	103
Lower Bound (units)	102	102	102
Distance to Bound (%)	0.98	0.00	0.98
Execution Time (s)	111.06	0.06	0.05

Table B.4: Simulation results for the problems with 300 jobs to schedule.

Sample 1	Agent	LPT	BFD
Job Dimension (%)	30.92	10.65	10.65
Job Difference (‰)	79.89	27.50	27.50
Min Finish time (units)	145	147	145
Max Finish time (units)	151	149	149
Lower Bound (units)	148	148	148
Distance to Bound (%)	2.03	0.68	0.68
Execution Time (s)	1219.76	0.28	0.28
Sample 2	Agent	LPT	BFD
Job Dimension (%)	23.93	4.40	4.40
Job Difference (‰)	61.58	11.31	11.31
Min Finish time (units)	151	153	152
Max Finish time (units)	156	154	154
Lower Bound (units)	154	154	154
Distance to Bound (%)	1.30	0.00	0.00
Execution Time (s)	1218.45	0.28	0.28
Sample 3	Agent	LPT	BFD
Job Dimension (%)	21.74	11.86	1.98
Job Difference (‰)	60.44	32.97	5.49
Min Finish time (units)	149	151	150
Max Finish time (units)	154	153	152
Lower Bound (units)	152	152	152
Distance to Bound (%)	1.32	0.66	0.00
Execution Time (s)	1219.63	0.28	0.28
Sample 4	Agent	LPT	BFD
Job Dimension (%)	14.75	4.92	4.92
Job Difference (‰)	42.39	14.13	14.13
Min Finish time (units)	150	152	151
Max Finish time (units)	154	153	153
Lower Bound (units)	153	153	153
Distance to Bound (%)	0.65	0.00	0.00
Execution Time (s)	1218.84	0.28	0.27
Sample 5	Agent	LPT	BFD
Job Dimension (%)	22.30	2.91	2.91
Job Difference (‰)	66.58	8.68	8.68
Min Finish time (units)	152	154	153
Max Finish time (units)	157	155	155
Lower Bound (units)	155	155	155
Distance to Bound (%)	1.29	0.00	0.00
Execution Time (s)	1218.81	0.28	0.28

Table B.5: Simulation results for the problems with 1000 jobs to schedule.

Sample 1	Agent	LPT	BFD
Job Dimension (%)	26.91	17.10	7.29
Job Difference (‰)	77.84	49.46	21.08
Min Finish time (units)	289	290	288
Max Finish time (units)	294	293	292
Lower Bound (units)	292	292	292
Distance to Bound (%)	0.68	0.34	0.00
Execution Time (s)	22252.59	2.33	2.30
Sample 2	Agent	LPT	BFD
Job Dimension (%)	22.86	12.86	2.86
Job Difference (‰)	68.22	38.37	8.53
Min Finish time (units)	283	285	283
Max Finish time (units)	288	287	286
Lower Bound (units)	286	286	286
Distance to Bound (%)	0.70	0.35	0.00
Execution Time (s)	22266.13	2.33	2.31
Sample 3	Agent	LPT	BFD
Job Dimension (%)	21.48	11.59	11.59
Job Difference (‰)	60.93	32.87	32.87
Min Finish time (units)	286	288	286
Max Finish time (units)	291	290	290
Lower Bound (units)	289	289	289
Distance to Bound (%)	0.69	0.35	0.35
Execution Time (s)	22267.28	2.31	2.28
Sample 4	Agent	LPT	BFD
Job Dimension (%)	26.03	16.02	6.01
Job Difference (‰)	73.44	45.19	16.95
Min Finish time (units)	283	284	282
Max Finish time (units)	288	287	286
Lower Bound (units)	286	286	286
Distance to Bound (%)	0.70	0.35	0.00
Execution Time (s)	22289.44	2.34	2.33
Sample 5	Agent	LPT	BFD
Job Dimension (%)	24.13	14.19	4.26
Job Difference (‰)	67.93	39.96	11.99
Min Finish time (units)	285	286	286
Max Finish time (units)	290	289	288
Lower Bound (units)	288	288	288
Distance to Bound (%)	0.69	0.35	0.00
Execution Time (s)	22286.11	2.31	2.33