Facultad
de
Ciencias

Planificador de Tareas en Datacenters Heterogéneos basado en Deep Reinforcement Learning

(Task Scheduler for Heterogeneous Data Centers based on Deep Reinforcement Learning)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Jaime Fomperosa San Sebastián

Director: José Luis Bosque Orero

Julio-2021

Contents

List of Figures

List of Tables

Abstract

The computational capacity needs by both companies and research groups all over the world has increased greatly in the last years. These necessities are due to the increasing amount of data to analyse, training of machine learning models, high performance scientific applications and many other types of jobs that are too costly for the infrastructure of these groups. This means that data centers are getting more and more jobs, and thus need more and more resources to accommodate all of them. The scheduling of all of these tasks in the heterogeneous resources of a data center is too complex computationally to be performed optimally, as it is an NP-Complete problem. For this reason, task scheduling is still performed by heuristic algorithms. However, these algorithms are beginning to fall short for managing the great heterogeneity of both jobs and resources efficiently. This fact, together with the desire of being able to have a method that is adaptable to different objectives such as minimizing energy consumption or job slowdown, opens the door to the idea of using a machine learning approach. The objective of this work is to design and implement an intelligent agent that is able to make use of all the available information from both jobs and resources to take the best scheduling decisions depending on the selected objective.

Keywords: Deep Reinforcement Learning, Task scheduling, Heterogeneous data centers, RLScheduler, Artificial intelligence, Machine Learning.

Resumen

Las necesidades tanto de las empresas como de los grupos de investigación de todo el mundo en cuanto a capacidad de cómputo ha crecido enormemente en los últimos años. Estas necesidades se deben a la creciente cantidad de datos que analizar, entrenamientos de algoritmos de inteligencia artificial, aplicaciones científicas de computación de altas prestaciones y muchos otros tipos de trabajos que son demasiado costosos para la infraestructura de muchos de estos grupos. Esto hace que los data centers cada vez reciban más y más trabajos, y necesiten a su vez más y más recursos para dar cabida a todos ellos. La planificación de esta gran cantidad de trabajos en los recursos tan heterogéneos de un data center es una tarea computacionalmente demasiado compleja como para ser realizada de forma óptima, ya que es un problema NP-Completo. Debido a esto, hasta ahora esta planificación ha sido realizada mediante el uso de algoritmos heurísticos. Sin embargo, estos algoritmos comienzan a quedarse cortos a la hora de manejar de forma eficiente la gran heterogeneidad tanto de los trabajos como de los recursos. Esto, junto a la necesidad de tener un método que sea capaz de adaptarse a distintos objetivos como pueden ser minimizar el consumo de energía o el slowdown de los trabajos, abre las puertas a tratar de usar un enfoque basado en aprendizaje automático. El objetivo de este trabajo es diseñar e implementar un agente inteligente que sepa aprovechar toda la información disponible tanto de los trabajos como de los recursos para hacer una planificación lo más eficiente posible en función del objetivo seleccionado.

Palabras clave: Aprendizaje Reforzado Profundo, Planificación de tareas, Datacenter heterogéneo, RLScheduler, Inteligencia artificial, Aprendizaje automático.

Chapter 1

Introduction

In this chapter, the main concepts to understand the basis of this work as well as its motivation will be explained. Moreover, the main objectives of the project will be listed. Finally, the structure of the document will be summarised.

1.1 Task Scheduling in Data Centers

Task scheduling is the process by which tasks are assigned a set of computational resources to be executed. At an operative system level, these tasks can be for example threads or processes, and the resources are the processors and cores of the system. This is the case for a single computer. However, by connecting several computers together this can be made as complex as desired. This is how data centers came into being.

By grouping several processors together with a shared memory, a compute node is obtained. Then, several nodes can be interconnected to form a cluster and be able to share information between them. Finally, the clusters can also be connected by creating an interconnection network among them [37]. This results in a data center, which will have a greater or lesser complexity depending on the number of clusters and the heterogeneity of the resources that make them up [16].

Nowadays, data centers are formed by hundreds or thousands of very varied computing resources. There can be nodes with regular processors, others with a higher amount of memory for processes that may need it, hardware accelerators like GPUs, etc. These great amount of resources is necessary to be able to allocate the equally great number of tasks arriving every day to be executed in the data center [6].

This area has a great research interest and in general, due to the complexity of the real systems where testing would be too costly, simulators are used to model these systems and test the proposed algorithms.

Moreover, the scheduling necessities of the data center may vary over time. At a certain moment it might be preferred to minimize the energy consumption, while at a different one it might be to maximize resource utilization or minimize the job slowdown. This high variability, together with the complexity of the environment, creates the need for some sort of more advanced scheduler with respect to the heuristic algorithms that are currently used [7].

This more advanced scheduler should be able to consider all the information available in order to make the best possible scheduling decisions, and select which job should be executed first, and in which resource it should be executed. This is precisely where artificial intelligence, and machine learning in particular, comes into play.

1.2 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that allows for models to be learnt with no prior information by the method of trial and error. This paradigm is based on the interactions between an agent and the environment. In the context of this project, the goal is to design an intelligent scheduler which will be the *agent*. The *agent* will interact with a simple simulator of a data center, the *environment*. By taking actions and then receiving feedback for those actions, the *agent* must learn progressively which actions are better than others for a certain state of the *environment*.

This method, RL, together with the use of *Deep Neural Networks* to model the agent, leads to *Deep Reinforcement Learning*. This technique is already being used by several research groups around the world, trying to create an

intelligent scheduler that can replace the current heuristic schedulers. Some of these projects will be summarized in Section 2.3.

One of the most remarkable works is *RLScheduler* [47], which presents fairly good results with the major disadvantage of simulating an homogeneous data center. In spite of the good results, this approach has the main drawback of not being quite realistic nowadays, and the fact that the change from an homogeneous system to an heterogeneous one results in a huge increase in complexity.

1.3 Objectives

According to these observations, the conclusion is that the study of task scheduling algorithms for heterogeneous data centers is a new field of research for which it is necessary to develop new tools. In this respect, the design and implementation of a simulator that allows researchers to test their task schedulers based on RL is essential. This simulator must allow for the representation of a real data center, that even if simplified must be enough to model the heterogeneity of current systems.

RLScheduler is both a simulator and an intelligent scheduler, although for the much simpler homogeneous systems. The aim of this project is to expand the capabilities of RLScheduler, both in terms of simulating and scheduling. Removing the disadvantage of homogeneity will give rise to an intelligent heterogeneous scheduler that should perform more efficiently than state-of-the-art algorithms. This goal can be divided in three main objectives:

- *Adaptation for modelling heterogeneous resources*: RLScheduler works with the assumption that all the resources are equal. To make the simulation more realistic, a new way of modelling the distinct resources must be implemented. The new platform must be able to define the hardware in three levels: the whole data center, the nodes that form it, and the processors of which they are composed.
- *Design and implementation of an intelligent agent*: on the new environment, the original agent from RLScheduler is not capable of making a correct scheduling, as it has no way of selecting the proper resources.

Thus, a new intelligent agent that can select a particular job to be scheduled in a particular resource must be designed.

- *Evaluation to ensure the increase in efficiency*: once implemented, there must be a testing phase to check that this new agent actually represents an improvement with respect to classic heuristic algorithms. The comparison should be done for different objectives to see if the method is generalisable for different purposes.

1.4 Methodology & Work plan

For achieving the described objectives, the following steps must be followed:

- *Study of task scheduling techniques in data centers*: first of all, it is necessary to get familiarised with the state-of-the-art scheduling algorithms. Knowing how this it is currently done allows to recognise the problems of the process, which is the starting point before proposing a solution.
- *Study of the relevant machine learning techniques*: knowing the technique of Deep Reinforcement Learning and how it is implemented is crucial for being able to apply it to a new problem like is task scheduling in heterogeneous data centers.
- *Analysis of RLScheduler*: before doing modifications on the program, an in-depth analysis must be done to identify exactly which parts need to be changed and which changes need to be made.
- *Adding heterogeneity to RLScheduler*: the first change must be to model the different resources with their different attributes, to make a distinction between them.
- *Design and implementation of the agent*: then an agent that is able to schedule jobs and allocate them resources in the new platform must be created.
- *Testing the agents on the new system*: the results of the agent should be tested for different scheduling objectives and compared to those of classic algorithms.

1.5 Document structure

This paper is composed of 6 chapters, including the current Chapter 1 as *Introduction*.

- *Chapter 2*: the basic concepts for understanding the rest of the document are introduced.
- *Chapter 3*: RLScheduler is analyzed to get an in-depth knowledge before planning a redesign to simulate heterogeneous systems.
- *Chapter 4*: the changes made to the simulator are described. Two new intelligent agents are designed for task scheduling in the new environment.
- *Chapter 5*: the experiments conducted for checking whether the agents implemented in Chapter 3 are successful are detailed. To check this success, their results are compared to those of classic algorithms.
- *Chapter 6*: to complete the work, the final conclusions are presented, together with some possible future lines of work in relation to this one.

Chapter 2

Background

In this chapter, the concepts considered the most relevant in order to understand the rest of the paper will be introduced and explained. The main topics are the scheduling algorithms that are being used in real environments nowadays, and some notions about artificial intelligence. Also some works related to this project will be summarized.

2.1 State-of-the-art scheduling algorithms

In the context of a data center, task scheduling is the process carried out by a *Workload Manager* [32]. The Workload Manager is the software responsible for choosing one of the jobs currently waiting in the job queue, and send it to be executed in a set of the resources of the data center. What makes this such a complex process is the fact that the workload manager needs to consider several different factors, while also being able to take a decision in real time.

In the first place, the job queue can receive hundreds of jobs at a given time, each of them with a wide range of different attributes. The standardized way of defining these attributes is the Standard Workload Format (SWF) [13], which indicates several properties of the jobs (e.g., the maximum run time, the amount of memory, or the number of processors requested).

Secondly, the available resources must be also taken into account. In the context of a homogeneous data center this might not be relevant, as sending a given job to any of the resources will produce the same result. However, this is not the case for most modern data centers, which are heterogeneous [27] and composed of many different types of hardware, as well as pieces of the same hardware with different specifications. Thus, creating a difference between scheduling a certain job in one resource or another depending on their characteristics.

Moreover, the objectives might vary over time. These can range from makespan minimization, i.e., the time elapsed from the start of the job to its end; to energy consumption, i.e., minimizing the energy used by the data center to execute the jobs. Each of these objectives would correspond to a different scheduling, and this is not manageable by classic algorithms.

Clearly, it is not feasible to compute all the possible outcomes of scheduling each of the jobs in the job queue to each of the resources, for all the possible states that the environment might take, in order to choose the best combination. This is one example of an NP-Complete problem [43]. To be more precise, it is a particular case of a Generalized assignment problem [36].

Task scheduling being an NP-Complete problem means that it cannot be solved in polynomial time. In fact, no algorithm has been found that can solve this kind of problems faster than in superpolynomial [19] time. However, it is possible to find *near-optimal* solutions using other methods such as *approximation* [45] or *heuristic* [30] algorithms.

Heuristic algorithms, in particular, are the ones currently being used for taking the decisions in the process of task scheduling. They are characterised by sacrificing optimally for speed [4], which is a necessary compromise considering that the optimal solution for these problems would take years to be computed.

The heuristics used are rather simple ones. In most cases nowadays, two algorithms are used: First In, First Out (FIFO) [40, 31] and BackFill [21]. In some situations where slowdown is to be minimized, Shortest Job First (SJF) might also be used [40, 31].

- FIFO algorithm schedules the jobs in the same order as they arrived to the job queue, so it considers only the submission time to do the

scheduling.

- BackFill algorithm is a variant of FIFO. Its advantage lies in being able to recognise the jobs from the job queue that can be processed without interfering with the one at the head of the queue. For a job to be moved forward, it must be able to finish before the time that the actual first one is expected to start. This way, small jobs are not unnecessarily slowed down, and they do not take resources away from the big ones either.

- SJF algorithm, in contrast, considers the running time of the jobs to schedule the shortest one each time.

Even though in theory this approach is good for not having short jobs waiting unnecessarily for longer jobs to finish, it might not be very precise. This is because the running time of a job is just an upper estimation, but not necessarily the real time that the job will take [28]. Not only that, but it can also lead to *starvation* [41]. If short jobs keep arriving during an extended period of time, longer jobs will have to wait indefinitely.

There are more complex heuristic algorithms that consider several attributes of each job in order to compute a score, which is then used to sort and prioritize them. Examples of these are WFP3, UNICEP [42] or F1 [9]. Nonetheless, even these more sophisticated algorithms are not able no produce such good results in all cases. F1, for instance, is capable of minimizing *average bounded slowdown*, but might not be the best option for other objectives.

For all these reasons, a *machine-learning* based solution can prove to be very powerful. The system can be trained to take into account not just a few but all the relevant attributes from both the jobs and the nodes, in order to find the best scheduling decision at each time step. Not only that, but machine learning has already shown its adaptability to different scenarios [3], in contrast with the classic algorithms' invariable approach. This is done by changing the job scheduling policies at any time according to the arriving jobs. Classic algorithms are incapable of anything remotely similar to this.

2.2 Machine learning

Apart from the simulation aspect, this project has an important part of machine learning. The main algorithms used related to this topic are *Deep Reinforcement Learning* and *k-means*. These will be summarized in this section, that may be skipped if the reader has previous knowledge about the subject.

2.2.1 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) arises from idea of using Deep Neural Networks (DNN) to tackle Reinforcement Learning (RL) problems.

Deep Neural Networks

Deep Neural Networks differ from regular Artificial Neural Networks [5] in that they have several hidden layers instead of a single one, as shown in Figure 2.1.

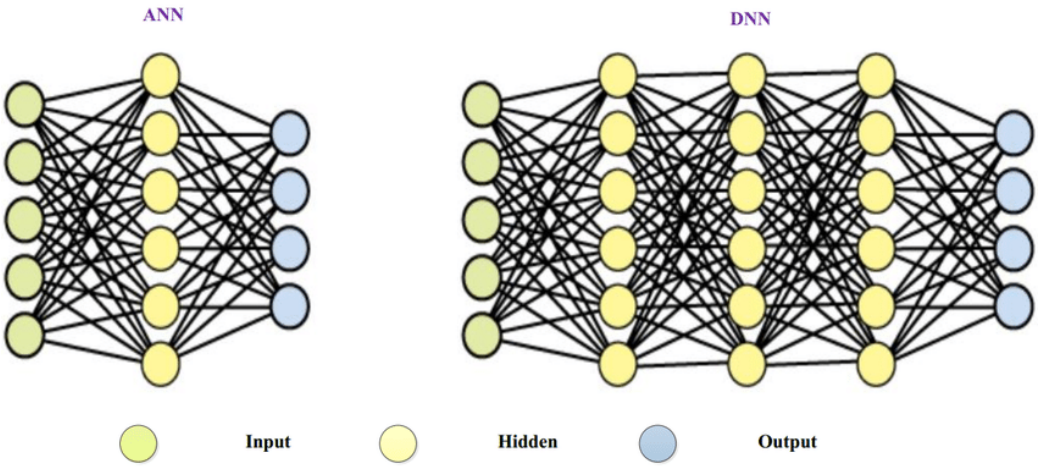

Figure 2.1: Artificial Neural Networks vs Deep Neural Networks [1]

[1]Figure from Sheraz Aslam et al. "Deep Learning Based Techniques to Enhance the Performance of Microgrids: A Review". In: Nov. 2019. DOI: 10.1109/FIT47737.2019.00031

Artificial Neural Networks are *universal function approximators* [10]. In their most basic form, they have a layered structured formed by three layers of fully-connected artificial neurons (dense layers). The input layer receives external data which is passed to each neuron in the hidden layer. A neuron is defined by a weight and a bias $w_{ij}, b_{ij} \in \mathbb{R}$. After getting an input vector of size m, each neuron produces its output as:

$$z = \sum_{i=1}^{m} w_i * x_i + b_i \tag{2.1}$$

Then, an *activation function* [35] is applied to the output of the whole layer, in order to decide if each one of the neurons should be activated or not according to some threshold. The vector of outputs, which will have the same size as the number of neurons of the layer, is then finally passed to the output layer. The output layer repeats the aforementioned process and returns an output of the desired size, which will depend on the problem.

When instead of using a single hidden layer, several of them are put together, that is called a Deep Neural Network. These work in the same way as explained before, but can create more complex models. This makes it possible to solve more complex problems, thus why they have been used in this project.

The model of a neural network is defined by the weights of its neurons and the connections between them. This means that for a neural network to behave as expected, it must be trained beforehand to adapt its initial random weights to ones that produce the desired result. The training process is based on two phases, *forwarding* and *backpropagation*, which are repeated a number of times, called *training epochs*.

- *Forwarding* [17]: The neural network takes an input and process it layer by layer as mentioned before. The output of each layer serves as input for the next one. Finally, the output layer returns a vector of size equal to its number of neurons.
- *Backpropagation* [33]: Once the *forwarding* step has been done, the result is compared to the expected value for the current input. Applying the *loss function*, it is possible to see how far from the correct result the network was. An example of a simple loss function could be the difference between the expected result and the actual result. Once the *loss value* has been computed, the *gradient descent* [18] algorithm is

applied backwards to each layer: from the last hidden layer up to the first one. This process alters the weights of each neuron depending on their relevance for obtaining the output result. With this method, each training epoch manages to reduce the *loss* of the network, thus getting more precise results. For these changes to be progressive and avoid continuous big changes, the *learning rate* α is used: a small value that reduces the loss by a constant resulting in smaller steps.

Reinforcement Learning

Reinforcement Learning (RL), is a machine learning technique, where one or more agents interact with an environment through actions, trying to maximize a *cumulative reward* [38]. This process is repeated several times, until a certain objective is completed. Figure 2.2 illustrates what happens at each step.

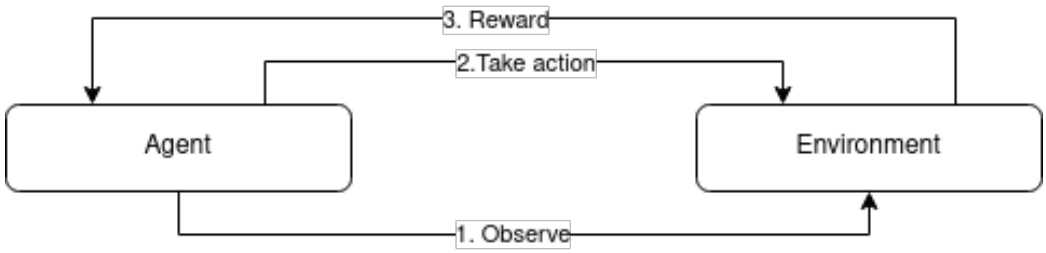

Figure 2.2: Interactions between the agent and the environment in Reinforcement Learning

For each time step t, the agent observes the *current state* (s_t) of the environment. With this information, that will be encoded in some data structure depending on the problem (normally an array), the agent must select an *action* (a_t) to take. In *Deep Reinforcement Learning, (DLR)*, this selection is performed by a *Deep Neural Network*. The DNN gets the observation as input, and outputs an score (or probability) of taking each possible action.

Once the agent has decided, it executes the action over the environment. This *alters* the environment into a *new state* (s_{t+1}). As feedback of how good the action was, the environment returns a *reward* (r_t). This reward will depend on the objective to achieve, and might be difficult to calculate in some cases. This occurs when it is unknown how good the actions taken have been towards achieving the final objective before getting the final result.

For each step the agent takes, it collects one of this rewards. Once the agent is *done*, the rewards are processed by means of the *discounted cumulative sum* (Equation 2.2).

$$R = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.2}$$

where r_t is the reward at step t, and $\gamma \in [0, 1)$ is the discount-rate. A value of γ closer to one gives more relevance to the last actions taken, while a value close to zero gives more relevance to the ones from the first steps.

In order to calculate the *loss* of the network, the difference between this cumulative reward and the *expected reward* for the training is taken. This value indicates how off the agent has been, and is finally used in the backpropagation step using the gradient descent algorithm as explained in the previous section.

Combining DNNs with RL

As previously mentioned, DNNs are used in DRL as the agent for selecting the actions. The advantage of DNNs with respect to other methods is that they are able to learn in a *model-free* environment [39]. In contrast to a model-based one, in model-free environments the transitions between states are unknown, as well as the reward for a certain action. The agent has no previous information of how the environment will react to an action, needing to focus on the exploration. This is precisely where DNNs are more powerful, being capable of creating a model starting from a really limited knowledge.

Actor-Critic

Actor-Critic algorithms [20] represent a dual approach to model the agent in RL problems. An actor-critic agent is composed of two structures: the *actor* selects an action to be applied on the environment; the *critic* estimates a value for the current environment state and *criticizes* the action taken by the actor according to it.

The actor and the critic are modeled by means of two DNNs. At each time step, they both take an observation as input. The output of the actor is the action to take, selected from a probability distribution among the scores of taking each action (computed by the DNN). Meanwhile, the output of the critic's DNN is a single value that represents the expected value for the current state.

The purpose of the critic's value is to strengthen or weaken the tendency of the actor to choose certain actions. As both of them begin with no knowledge of the environment, the actor and the critic become progressively better at their task in parallel. The actor gets better at selecting actions and the critic gets better at rating observations.

This process is done by an *optimizer* [12] whose goal is to minimize the *loss function*. The value of this *loss function* is computed at the end of an epoch, using the rewards and the critic values gotten at each time step, and then used for training both the actor and critic network.

2.2.2 K-means clustering

K-means clustering [23] is an algorithm that allows to group similar data together from data sets with values of any dimension. It is specially useful when working with high-dimensional data, as it becomes increasingly difficult to use graphical representations. The algorithm not only classifies each one of the input vectors in one of the k clusters, but also computes their *centroids*. *Centroids* are the most representative points for each cluster.

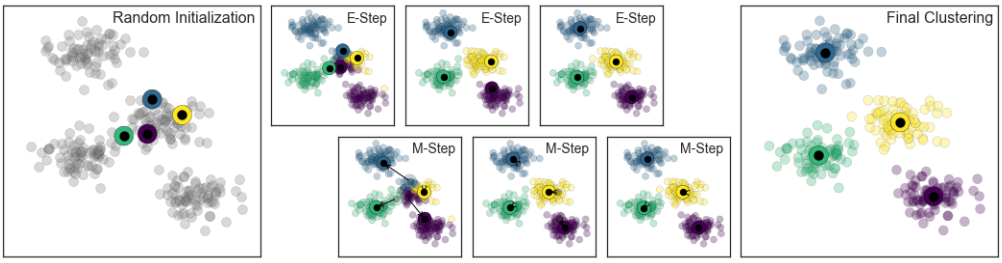

Figure 2.3: Clustering process over several steps using k-means algorithm [2]

[2]Figure from Jake VanderPlas. *Python data science handbook: Essential tools for working with data.* " O'Reilly Media, Inc.", 2016. Chap. 5.11

This algorithm's complexity is NP-Hard, so in practice heuristic algorithms are used. These are able to find local optimum solutions efficiently, although not able to guarantee the optimal solution. The most used one, *naive k-means*, begins with k randomly selected centroids, and iteratively tries to refine them until reaching a (local) minimum. The operator to minimize is the sum of squared Euclidean distances (d) between each point in the cluster ($x^{(i)}$) and its centroid (c_j). Formally,

$$\frac{1}{m}\sum_{j=1}^{k}\sum_{x^{(i)}\in C_j} d(x^{(i)}, c_j)^2 \tag{2.3}$$

2.3 Related work

With the rise in High-Performance computing and cloud computing in the last years the scheduling complexity has augmented. But the scheduling algorithms used for the increasingly large workloads have not kept up with the changes. This combined with the tendency to apply machine learning to new fields, has led to many groups around the world to focus on this problem.

One of the first projects that, motivated by advancements in DRL tried to present a solution was *DeepRM* [25]. In this work, the scheduling problem is modelled as a learning problem. This way, scheduling is no longer performed in a deterministic way as with heuristic algorithms, but by more adaptable means.

DeepRM was in turn an inspiration for *HDeepRM* [2, 3]. HDeepRM, now renamed to *IRMaSim* [1], emerges as a tool to develop and test reinforcement learning algorithms on a simulator of a customisable heterogeneous data center. In its second iteration, an intelligent agent is created that is able to select an optimal policy for a certain workload given an optimization objective.

Another interesting project is *Decima* [24], by researchers from the MIT, focuses in jobs with interdependencies represented as *directed acyclic graphs* (DAGs) [48]. They recognise the problem of using simple heuristic algorithms to solve the complex problem of scheduling batch jobs. The proposed solution uses RL and ANNs to learn optimal scheduling policies given an objective, such as *job completion time*. Besides reinforcement learning, they created

methods for working with their DAG representation of jobs.

Other projects focus on the energy consumption aspect. This is also quite relevant as the powering all the computation nodes has a huge cost. Examples of projects with an special focus on energy efficiency are *ExpREsS* [26] or *Concurrent Application Bias Scheduling for Energy Efficiency of Heterogeneous Multi-Core platforms* [34].

Even though all this works are remarkable, the main focus for the rest of the project will be on *RLScheduler* [47]. RLScheduler is an automated HPC batch job scheduler for homogeneous clusters. Its main problem is that homogeneous system are not realistic nowadays. In the following sections, RLScheduler will be described in depth and a way to adapt it to heterogeneous environments will be proposed and tested.

Chapter 3

Analyzing RLScheduler

In this chapter, RLScheduler's characteristics will be detailed, with an special focus on its greatest shortcomings. This will serve as a preliminary step to understand the improvements that will be proposed in the next chapter.

3.1 Overview

RLScheduler [47] is an automated HPC batch job scheduler that uses reinforcement learning. *Heuristic priority functions* have been highly studied and discussed by experts trying to find an optimal one for job scheduling. However, all of them have the same drawback: they are hardly adaptable to changes. Considering how variable the conditions in a data center are, where jobs' characteristics, optimization objectives or system settings can vary over time; having an adaptive solution is crucial. RLScheduler claims to be this solution.

RLScheduler is built in such a way that it does not have or need any previous expert knowledge. Using reinforcement learning techniques, it is able to infer the optimal scheduling by itself. As inputs, the system only needs to take the job traces to schedule, and an optimization objective among the defined ones, e.g. *slowdown*, *bounded slowdown* or *average waiting time*.

For being able to learn the scheduling, RLScheduler uses two reinforcement

learning algorithms, actor-critic and proximal policy optimization.

3.2 RLScheduler functionality

Once RLScheduler was selected as basis for the project, it needed to be analyzed in order to understand how it worked and then modify it. Apart from the paper [47] where it showed promising results, the code was available as open-access software [11] both in *zenodo* and *GitHub*. This made it quite simple to clone the project and work on it.

RLScheduler is a command line application, so it is run by passing all the parameters for the training through a command. In the first place, all the operations and structures are defined and then the training itself starts. The training is done by selecting a *workload trace*, and simulating its execution a high enough number of times. Seeing how well or bad the results are, the agent must encourage some actions and discourage others.

The training is divided in *epochs*. After each epoch, the results obtained are processed and used to update the weights of the DNN. The number of epochs must be enough for the network to converge, being 100 a reasonable value.

Each one of the epochs consists of several *trajectories*. In each trajectory, a random subset of the whole trace is selected. This subset of jobs is then simulated using the intelligent agent for scheduling. During this process, the agent gets an observation and chooses an action according to it, performing a *step*. If the selected job can be scheduled right away, it is. Otherwise, it is necessary to wait until enough resources are freed or a shorter job arrives. After the job is scheduled, the agent receives a reward and the next observation, with which it repeats these steps.

Once all the jobs have finished and the job queue is empty, the trajectory ends. Then the final results for the trajectory are computed. This calculations depend on the *objective* that is being optimized, and provide a general vision of how good the scheduling have been.

The reason for using several trajectories per epoch instead of a single one is to reduce the impact of outliers. As sets of jobs are selected randomly, some traces might be really easy to schedule (e.g. many short, time-spaced jobs).

Meanwhile, if most of the jobs are concentrated in a short period of time, slowdowns will happen and it might be unavoidable to get a bad result.

To counter these situations, several trajectories are performed, and at the end the mean values of all of them are the ones used for the training. The complete training process just detailed is shown in Figure 3.1.

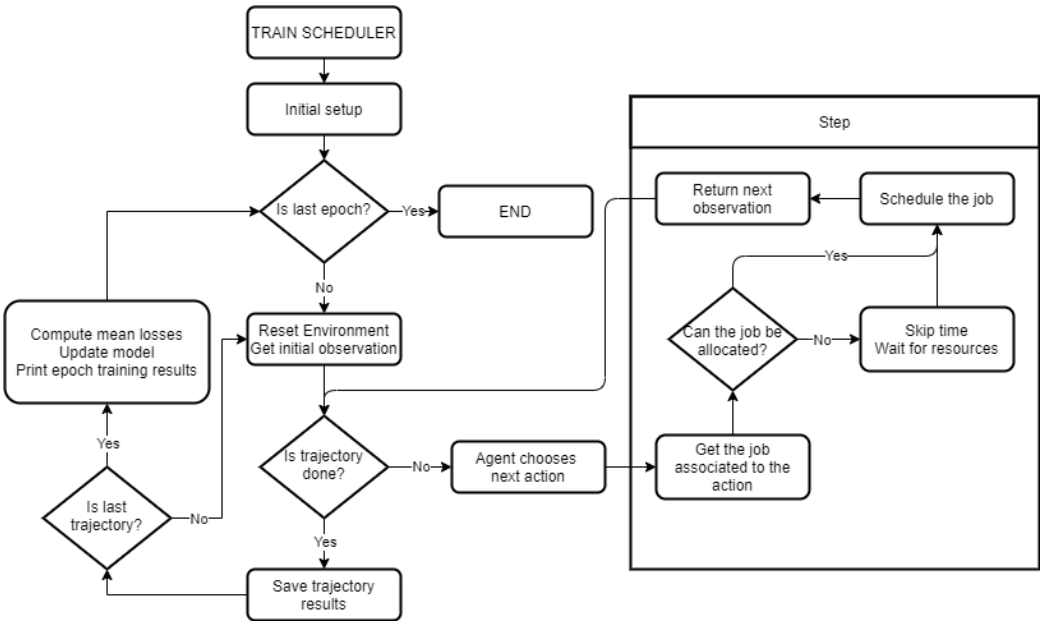

Figure 3.1: The process of training a scheduler across several epochs with several trajectories each.

RLScheduler's machine learning approach is more adaptable to different scenarios as it can potentially take all the information from the environment into account. Instead of relaying only in one or a few of the job attributes, the scheduler is able to learn which are the important ones among all of them. In fact, it learns how important each one is to deliver the best results.

RLScheduler implements a *Gym environment*. The Gym library [8] provides an interface for implementing environments on which to develop and compare reinforcement learning algorithms. The environment is divided in two main parts: an *observation space* and an *action space*. The observation space defines the shape of each observation, which is the information passed to the agent (DNN) to select one of the actions from the action space.

3.2.1 Observations in RLScheduler

At each time step, the intelligent agent in RLScheduler receives an observation, which is a representation of the current state of the environment. In its current release version (v0.1), the way of representing the environment is by means of a vector that contains the concatenated attributes of all the jobs in the job queue. This is all the information available to the agent for making a decision.

The size of an observation in RLScheduler is equal to $|JobQueue| * |Job|$. Here, $|JobQueue|$ is a constant set to 128, a value that is adopted in very used workload managers like Slurm [46]. If the actual job queue is smaller than this value, the rest of the vector is filled with zero values to represent *empty jobs*. In case of being bigger, an slice with the first arrived jobs is taken. $|Job|$ is the *size* of a job, or the number of attributes that represent it, which can be seen in Table 3.1.

Field Name	**Notation**	**Description**
Wait Time	w_j	Time (in seconds) spent by the job in the job queue
Run Time	r_j	Estimated run time (in seconds) for the job
Requested Processors	n_j	Number of processors requested for the job
Requested Memory	m_j	Memory (in MB) requested for the job
User Id	u_j	Id of the user that sent the job
Group Id	g_j	Group of the user that sent the job
Executable Id	e_j	Id of the job in the platform
Can Schedule Now	c_j	Boolean indicating whether there are enough resources to schedule the job at the moment

Table 3.1: Description of job attributes used in RLScheduler v0.1

Once created, the observation of 128 jobs is passed to the *agent*. In order to calculate a *score* for each one of the jobs, the agent feeds the attributes of job to a fully connected DNN. The DNN then outputs an score s_{j_i} for each one of the jobs, representing how good of an action would executing the job j_i be in the current timestamp.

As mentioned earlier, some of the jobs in the observation might be *empty jobs*. These appear because the observation must have a fixed size, but they should never be chosen to be executed. Thus, after getting the scores, a mask is applied to avoid these jobs from being selected.

With this mask applied, a softmax function is used to transform the scores

into a vector of probabilities p_{j_i}. Each p_{j_i} value indicates the probability that the corresponding job will be selected for scheduling. The actual selection is done using a *categorical distribution* [29] while *training* the agent to encourage the exploration, and using a *max* function during *testing* to get the best action.

This whole process from getting an observation from the job queue to the getting the vector of probabilities is represented in Figure 3.2.

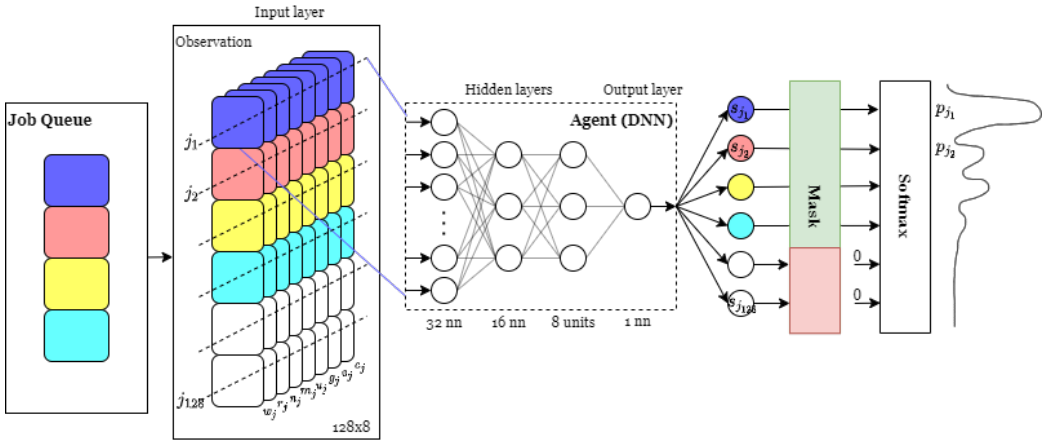

Figure 3.2: The process of selecting an action in RLScheduler

3.2.2 Actions in RLScheduler

The second part of the environment is the *action space* which, as seen above, is made up of all the possible actions that can be selected at a given time step. This is the set of jobs currently in the job queue. However, so far it has not been mentioned where the selected job is going to be executed.

RLScheduler models a homogeneous system. This means that there is a *number* of *resources* defined, but the resources are not differentiated in any way. When a job j is selected, n_j processors are marked as *used*. This means that no other job can be assigned to those same processors until after r_j seconds. But knowing to which processors in particular the job is assigned, does not really matter.

This process is similar to what happens in actual workload managers like *Slurm*. Resources are managed as a queue, and when a job is to be executed,

it is sent to the first node from the queue. At most, it might have to be the first resource that meets a series of requirements (e.g. available memory). Anyways, in no case there exists a process to decide which node would be better to execute the job in.

Even though this is how it is actually done nowadays, the intuition says that it must be possible to do it better. Real systems are heterogeneous, and executing a job in one compute node or another should make a difference. This is what is going to be addressed in the next chapter. With all the information gathered so far, a redesign of RLScheduler will be proposed. This redesign will allow to model an heterogeneous system, and train an agent that is not just able to pick the next best job to schedule, but also the best compute node where to execute it.

3.3 How RLScheduler can be improved

RLScheduler has a major drawback. The scheduler assumes an homogeneous system. Its model has no support whatsoever for heterogeneous environments. It is possible to specify the number of processors of the system, but that is all the information regarding the resources. When a job is scheduled, the number of free processors of the system is reduced by the number of requested processors of the job. There are no differences between scheduling a certain job into one processor or another as they are not actually defined per se.

RLScheduler does show interesting results in the domain of homogeneous scheduling. However, this case is not quite realistic nowadays, specially when focusing on data centers and more so for cloud providers. The most modern data centers can have thousand of compute nodes, and many of them with different characteristics.

A complete solution to the problem should be able to take into account not only job attributes but also resource attributes, and plan efficiently according to both. This is precisely what this project aims to achieve.

Chapter 4

Heterogenizing RLScheduler: Design & Implementation

In this chapter, the redesign for converting RLScheduler in an heterogeneous intelligent scheduler will be detailed. Firstly, the modifications performed for simulating an heterogeneous environment will be explained. These will be followed by the proposal of two architectures for modelling the scheduler agent that is able to select the best possible *job-node* pair at each time step.

4.1 Defining the system resources

As seen in the previous section, RLScheduler in its original release did not model system resources in any way. When scheduled, jobs where only allocated a number of resources that could not be used by any other job until the current one finished. Resources were not defined further than as a number or differentiated in any way. Figure 4.1 represents the class diagram to visualize how little information was being used.

When aiming to model an heterogeneous system, a difference must be made between the resources. In the same way that jobs have attributes that characterize them, resources can have them too. These can range greatly from computational power to energy consumption and many more. The ones used must provide with adequate information for the goal to optimize. If the

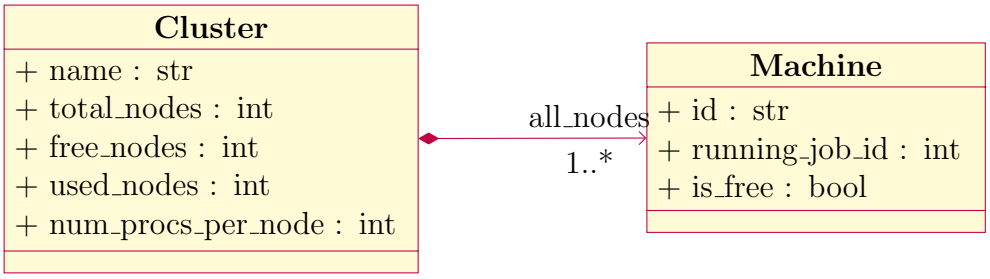

Figure 4.1: Model of a Cluster in RLScheduler v0.1

objective were to minimize energy consumption, for instance, it would be necessary to know how much each of the nodes uses. For this project, the focus is on a small set of these attributes, as RLScheduler is not capable of simulating aspects like resource contention, memory bandwidth or energy consumption.

Since these aspects would have no impact on simulation, there is no need to adapt the program for them. However, there are others that can be included without the need of a more complex simulator. To introduce heterogeneity with these constraints, the machines are differentiated in two ways: size and speed.

In the homogeneous version, a cluster was formed by *machines*, which was just enough to define it. From now on, the concept of *machine* will be substituted by the more precise concept of *node*. Nodes can have different *sizes* regarding the number of processors they are composed of. Moreover, a difference will be made depending on whether these processors are free or not. With respect to *speed*, nodes can be faster than others according to their *clock rate* or *frequency*. This value is the same for all the processors of a node and marks how fast a program is executed in said processors.

These changes mean that some of the nodes in the cluster can execute more jobs at the same time than others, and some nodes can execute jobs faster than others. Also, that *big* jobs will necessarily need to be allocated nodes with a great number of processors. Both of these aspects will need to be taken into account in order for the agent to make the best possible scheduling.

The new model of a Cluster can be seen in Figure 4.2. Apart from the new structure mentioned so far, some other attributes have been included as they were convenient for the simulation process. The *minimum and maximum*

frequency among the nodes of the cluster, for instance, was required for value normalization during the observation process. The *relative frequency* of a node is used to calculate easily how long a processor from that node will take to execute a job, compared to how long one with the minimum frequency would take. At a processor level, the *running job finish time* is used to know which processors become free at each time step.

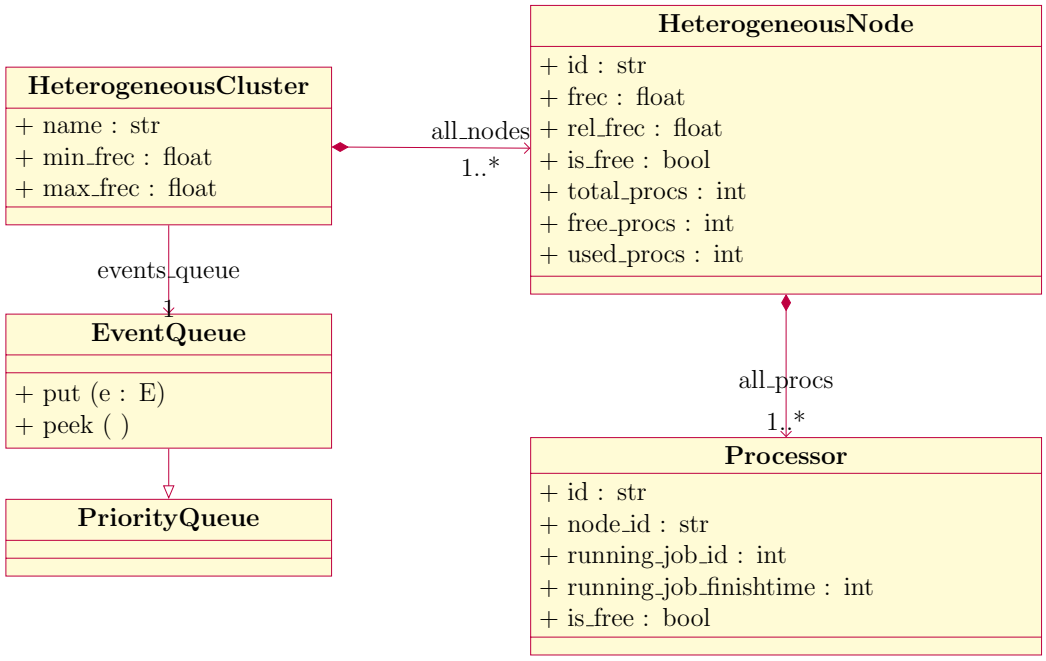

Figure 4.2: Model of a Cluster in the heterogeneous version of RLScheduler

Besides these, the Cluster now has an *Event Queue*, which is a slight modification of Python's *Priority Queue* [15]. An Event Queue is composed of timestamps of all the events that are expected to happen in the platform. These can be of two types: job arrivals, when a job enters the job queue and waits to be scheduled, or job execution finishes. They are used when during simulation, none of the jobs in the job queue can be allocated due to a lack of resources. In these cases, the current time is advanced up to the time indicated by the first timestamp in the queue. Then, the platform can free resources if any job has just finished or process the arrival of a new job. This is with the hope of now having enough free resources or a job small enough to be able to schedule it.

The reason for using a queue is that the timestamps need to be ordered so that the lowest one is always taken and no time is wasted by waiting more than necessary to decide the next action. As the content of the queue are

timestamps, clearly two of them with the same value represent the exact same moment in time. Thus, the change with respect to the original Priority Queue implementation has been simply to avoid repetitions by checking than a value is not already in the queue at insertion time. This way, the problem of trying to advance the time to an already passed moment is avoided.

Once the new *Heterogeneous Cluster* has been defined as just seen, the *Environment* must be adapted as well. Scheduling a job to be executed is no longer enough, now is has also become necessary to decide in which node it will be executed. The reason behind selecting a node and not a processor, which is where a job is really going to be executed, is that some jobs might request several processors to be executed in parallel. However, the agent takes a single decision, so the problem is simplified by considering each node as a whole and assuring that the node has enough free processors, instead of selecting n_j independent processors.

The main limitation of this decision is that there cannot be any job in the workload that requests more processors than the bigger node has. Otherwise, that job will never be executed. This can be solved by defining at least one node with enough processors, or by limiting the number of processors that a job can request up to a certain maximum. Otherwise, apart from a redesign of the agent, a more complex simulator would be needed. For reproducing the communication between nodes it would be necessary to model the interconnection network between them, which is out of the scope of this project. It is possible to simulate independent tasks or parallel tasks with shared memory, but not parallel tasks with distributed memory.

4.2 The new Environment

In Section 3.2.1 and Section 3.2.2, the *observation space* and *action space* that formed RLScheduler's Environment were described. After the Cluster, these, and the whole Environment with them, have been modified to model a heterogeneous architecture. In this section, the changes made to each of them will be detailed.

4.2.1 Observations in Heterogeneous RLScheduler

As seen, the *observation* is the information of the current state of the environment available to the agent for deciding the next action to select. The information of the jobs no longer describes the environment fully, for the agent to have access to a more complete picture, observations must include the state of the cluster as well.

The observation of the cluster is defined as the information of the nodes it is composed of. This results in a second observation of size $NumNodes \times NodeFeatures$. *NumNodes* is a constant defined in the *HPCSimPickJobsHeterog.py* file and indicates the number of nodes that compose the cluster. *NodeFeatures* is another constant defined in the same file, specifying the number of attributes that are being considered for each node when creating an observation. The attributes that are currently being considered for nodes are their *total number of nodes*, the *number of free nodes* at the moment of the observation, and the *clock rate* to be able to differentiate slower and faster nodes.

Moreover, observations have also been updated in the sense that some of the attributes from jobs have been removed. The reason is that they were mostly unused in the workload definition files, so all of them had the same values and were not providing any information. This has been the case for the *Requested Memory*, *User Id*, *Group Id* and *Executable Id*. With this simplification, the input size of the network, and therefore its complexity, is reduced.

Finally, an attribute that has been kept but with a slight change of meaning is *Can Schedule Now*. Originally, it indicated whether there were enough *machines* available in the cluster for that job at the moment. However, jobs are no longer going to be executed in the cluster in general, but in a certain node. Thus, this attribute is not associated with a job anymore, but with a *job-node* pair. Its value is given by $c_{j,n} = n_j \geq fp_n$, and will be used to prevent the agent from selecting invalid actions, i.e., choosing a job to be scheduled in a node that does not have enough resources for it.

The updated set of attributes for the observations with the described changes is shown in Table 4.1. Even though the observations now have one less attribute than before, the information they contain is more relevant and includes the aspect of heterogeneity.

Field Name	Notation	Description
# Requested Processors	n_j	Number of processors requested for the job
Requested Time	r_j	Amount of time requested for the job
Wait Time	w_j	Amount of time spent by the job in the job queue
Total Processors	tp_n	Number of processors in the node
Free Processors	fp_n	Free processors in the node
Clock Rate	f_n	CPU frequency of the node processors
Can Be Scheduled	$c_{j,n}$	Whether j can be scheduled in n

Table 4.1: Description of job, node and job-node attributes

4.2.2 Actions in Heterogeneous RLScheduler

Regarding the actions, the *action space* must change accordingly with the new selections that the agent can take. Formerly, the decision consisted in which job to execute, which makes an action space of dimension the size of the job queue. Action a_i corresponded to executing the job at index i in the job queue.

In the new environment, each action must define which job has been selected to be scheduled and also the node allocated to it. This translates into a new *bidimensional* (although just conceptually) action space, where one dimension consists of all the jobs in the job queue and the second dimension consists of all the nodes in the cluster.

Here, the action $a_{i,j}$ corresponds to executing the i^{th} job from the job queue in the j^{th} node from the cluster. This increases the size of the action space quite significantly, from $|JobQueue|$ to $|JobQueue| \times NumNodes$. However, this bidimensionality is necessary for the purpose of this work. Not only that, but the increase in the number of parameters is inevitable when modelling heterogeneous architectures [6].

4.3 New agents architectures

With new observations and actions already defined for the agent, the only missing part is now the agent itself. Two different architectures are proposed to try to achieve better results than with the current heuristic algorithms. Each one of the designs approaches the problem of *bidimensionality* in a

different way, while maintaining the general working structure of the original RLScheduler, which was an *actor-critic* agent. Besides, each one of the agents has some differences on how they take the observations input, and output the selected action, so these differences will also be highlighted.

4.3.1 Double Network Agent architecture

The first proposed architecture is the *Double Network Agent*. This name comes from the fact that the *actor* part of the agent is composed of two DNNs, as opposed to the original implementation. The complete process of selecting an action in a given time step is represented in Figure 4.3.

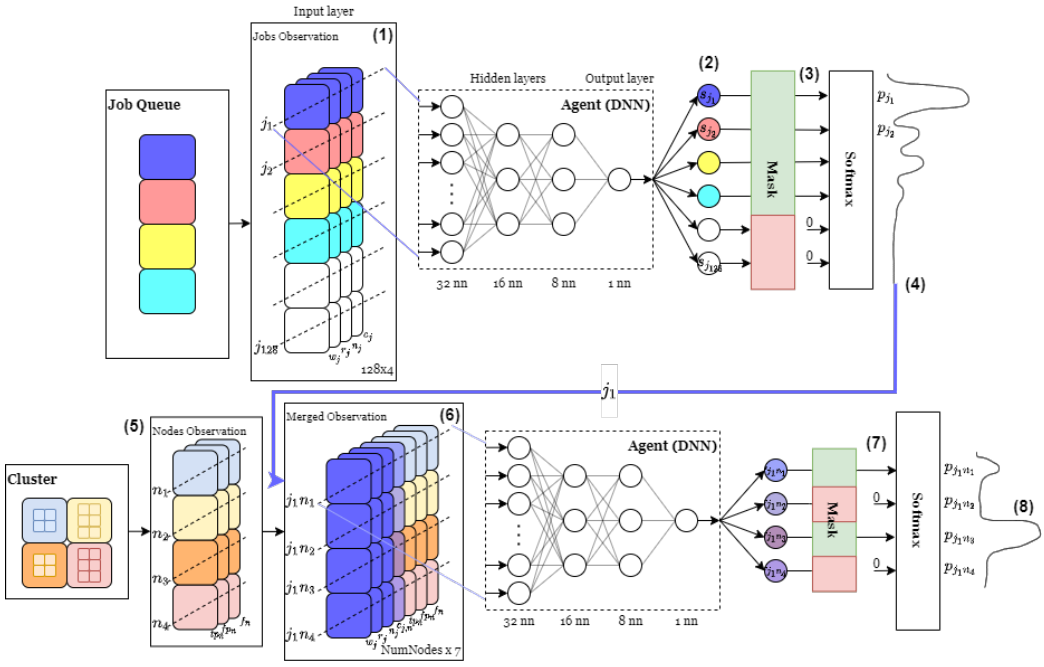

Figure 4.3: The process of selecting an action using the Double Agent in a Cluster with 4 nodes.

In the first place, the observation of the job queue is created **(1)**. The jobs that are currently waiting to be executed are the ones considered. For each one of them, four attributes are selected: the waiting time, the requested time, the requested number of processors and a boolean indicating whether the job can be scheduled at the moment. This creates a matrix of size 128×4. In case of there being fewer than 128 jobs in the queue, the corresponding

rows of the matrix are filled with zeros, as explained in Section 3.2.1. This matrix will be called the *jobs observation*.

The next step is to let the agent select the job to be executed, among those in the observation. This decision is performed by the first DNN of the two conforming the agent. The input layer of this network has dimension 4, which is just the number of attributes of each one of the jobs in the observation. Then it has three fully-connected hidden layers of 32, 16 and 8 neurons each, with ReLU as their activation function. Finally, the output layer is of size 1. This is because the idea of this network is just to provide a single score value for each one of the jobs.

In practice, this process is done for all the jobs at the same time by inputting the whole jobs observation matrix. Thus, the input for each neuron is not a scalar value but a vector of 128 values, representing the characteristic associated to that neuron for all the jobs. As the input is not a single job but all the 128 jobs, the actual output is not a single score but a vector of 128 scores, one for each of the inputted jobs **(2)**.

These scores are not yet the actual output as they need to be processed to avoid picking an invalid job. This is done by applying a mask that hides those *zero-valued jobs* that were added to match the fixed size of the observation. The ones with $c_j = 0$, i.e., those than cannot be executed at the moment, are also hidden by the mask **(3)**. This way, any job that the agent may choose is assured to be able to be scheduled without waiting for resources to get free.

Once only the scores for valid jobs are left, the whole vector is applied a softmax function. This transforms the scores by normalizing them into values in $[0, 1]$ and that add up to 1, or a *probability distribution*. From this probability distribution is from where a job is finally selected **(4)**. A higher probability should indicate that, according to its attributes, that job should be scheduled earlier than the others.

After the job has been selected, the agent must know decide which is the best node in which to schedule it. For this, the cluster needs to be observed first, creating the *nodes observation* **(5)**. This observation is analogous to the one of the jobs. It has as many rows as nodes in the cluster, and three columns that are the relevant attributes of each node. These attributes are the number of processors, the number of processors that are currently free and the clock rate.

With these attributes and the ones from the selected job, the agent must know decide which pair would result in a higher efficiency. To be able to evaluate both the job and the node together, their attributes are combined. As the job has already been decided, its four attributes are combined to the three attributes of each one of the nodes **(6)**. This results in a *merged observation* with as many rows as nodes, as the selection now must be one of the nodes. In this case the columns are $4+3$. The four previously mentioned attributes of the job, and the three of the nodes.

This combined observation is used as input for the second network of the agent, which has the exact same structure as the previous one with the sole difference of the input and output dimensions, that are now $NumNodes \times 7$ and $NumNodes$, respectively.

The process used to obtain the probability distribution of the nodes and then the selected node is also the same as for the previous step. The only remarkable difference is the mask applied to the scores vector **(7)**, which now hides the nodes that have fewer free processors than the requested by the job. Just as before, this eliminates the possibility of the agent choosing a node that is currently busy or does not have enough processors to execute that job.

With these two networks, a *job-node pair* is selected **(8)**. This information is then passed to the simulator to carry out the corresponding planning action. This performs a *step* action, which results in a new state of the environment. What this implies was explained in Section 3.2. Basically, a reward will be obtained together with a new observation representing the new state of the environment. This is used by the agent again to select the next *job-node pair*, and so on until the last job is processed.

Backpropagation of the error

As explained in Section 2.2.1, the error in a certain step is calculated as the difference between the result obtained by the network and the expected result. The main problem of this Double Network approach is having two error terms, one for each network. A first approach can be to just use each error to train its respective network. However, another way of seeing the issue is that both networks are working towards a common objective, so they should not be trained independently.

Moreover, if an action produces a bad result, is complicated to know what failed. It might be the first network who took a bad decision, making it impossible for the second one to produce a good result. It might also be the opposite, a bad decision of the second network masking a good selection of the job. Or it might have just been a failure in general and both performed badly.

This makes it quite difficult to know whether to incentivize or penalize each network of the agent after a certain decision. Or even if it makes more sense to train each network independently or to combine the errors adding them, for example, so they both learn at the same time. To be able to know for sure which one is the best option, both of these approaches will be tested during the evaluation phase.

4.3.2 Square Network Agent architecture

The second architecture is the *Squared Network Agent*. The name of this one comes from the fact that jobs and nodes are no longer evaluated in two separate steps. Rather, all the possible combinations between them are obtained, so each job is paired with each node. Thus, the term *Squared* stems from this *bidimensionality*. The data flow from this model can be visualized in Figure 4.4.

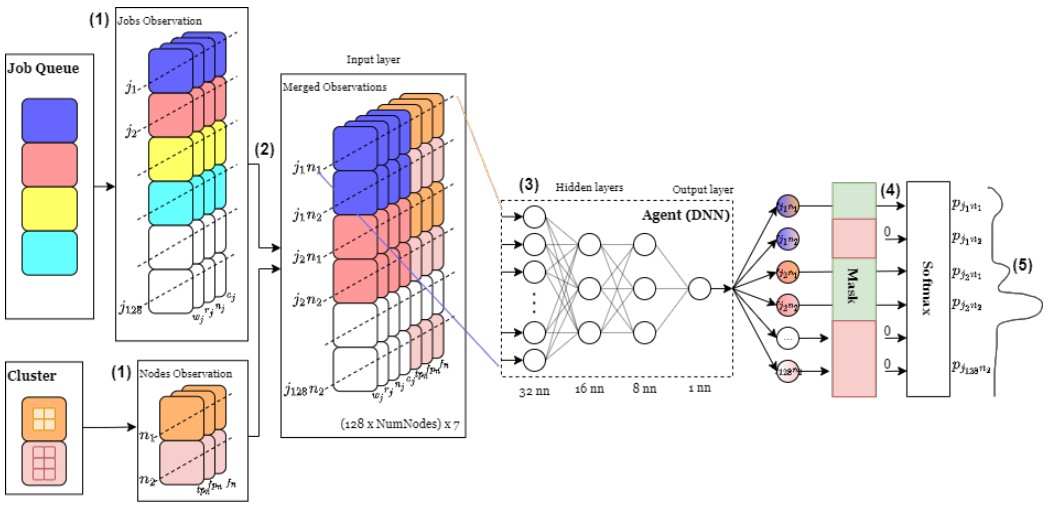

Figure 4.4: The process of selecting an action using the Squared Agent in a Cluster with 2 nodes.

Contrary to the previous case, when the agent *observes* the environment, both the *jobs observation* and the *nodes observation* are computed in the same step **(1)**. The main difference is that instead of making a decision on each one separately, there is a single selection that includes the job and the node.

As in this case the desired output for the network is directly a *job-node pair*, the input must contain information about both of them. This is why the two observations are combined in a single one that pairs every job with each node **(2)**. This way, the best combination of a job with a node, according to their attributes, can be selected.

This combined observation is a matrix of size $(128 * NumNodes) \times (JobFeatures + NodeFeatures)$. Here, $128 * NumNodes$ represents all the possible pairings of a job with a node, and $JobFeatures + NodeFeatures$ is the total number of attributes that define each of these pairs. The attributes used are those described in Table 4.1.

After the combined observation is ready, the selection process is analogous to the one seen earlier. The whole matrix is fed as input to the DNN which has the same structure as before **(3)**, only varying the input and output size.

The output of this network is a column vector with an score for each *job-node pair*. Then, the scores are applied a mask to remove the values corresponding to blank jobs or those that request more processors than the node's free ones **(4)**. Just as before, this is done to avoid invalid choices and selections that would result in a wait.

Finally, the values are normalized using the softmax function and the action is chosen from the generated probability distribution **(5)**. From the obtained action, $a_i \in [0..128 * NumNodes - 1]$, it is necessary to extract the corresponding job and node. The index of the job in the job queue is $i//128$, where $//$ is the floor division. For the node, the operation is $i \bmod NumNodes$.

The rest of the process after this point is the same as described in the previous section and in the first description of the program, also for the back-propagation step. Even though the dimension of the decision has increased significantly, the general structure has remained. There is not a second network like in the previous case or anything that would make the training more complex. Thus, this problem that would arise with the Double Agent is not a concern.

Reducing complexity with k-means

An issue that does appear, though, is the increasing complexity caused by having a single observation with all the possible combinations of a job with a node. For the Double Agent, in a cluster with 16 nodes, the total number of elements of both observations would be $128 * 4 + 16 * 7 = 624$. Meanwhile, with the Squared Agent the size of the only observation is $128 * 16 * 7 = 14336$. Doubling the number of nodes in the cluster results in 736 and 28672 elements respectively.

This difference has a clear impact in the performance of the Squared Agent, which is significantly slower than the Double. Moreover, it greatly harms scalability, as increasing the number of nodes would cause the agent to become too inefficient. This is a really big problem in this context, where clusters tend to have not twenty or thirty nodes, but hundreds or thousands of them.

The solution to this problem lies in reducing the dimension of the input observation somehow. The size of the job queue is a constant already used in other schedulers and kept for consistency with RLScheduler, and the number of attributes is already a rather small value so there is not much room for a reduction. The only variable left is the number of nodes.

Originally, it made sense to provide the network with information about every single node so it could select the ideal one. However, it is arguable whether every single value is needed, or if on the contrary it is possible to group similar ones together.

The idea is that if at a certain moment there are several nodes with the same characteristics in the cluster, it does not really matter in which one of those the job is executed. The proposal is to group the n nodes of the cluster in k *clusters* of similar nodes using the *k-means* algorithm. This is in line with the architecture of real data centers, where nodes are usually divided in groups also named *clusters*. Within each group, all nodes have the same characteristics, whether they are regular processors, GPUs, etc.

Figure 4.5 illustrates a simple case of the use of k-means. From a cluster of 4 nodes, the similarity between them is calculated. In this case, for simplicity, those with the same number of processors are grouped together **(1)**. Then, the combined observation is constructed by combining each one of the jobs

with each one of the clusters, instead of the nodes **(2)**.

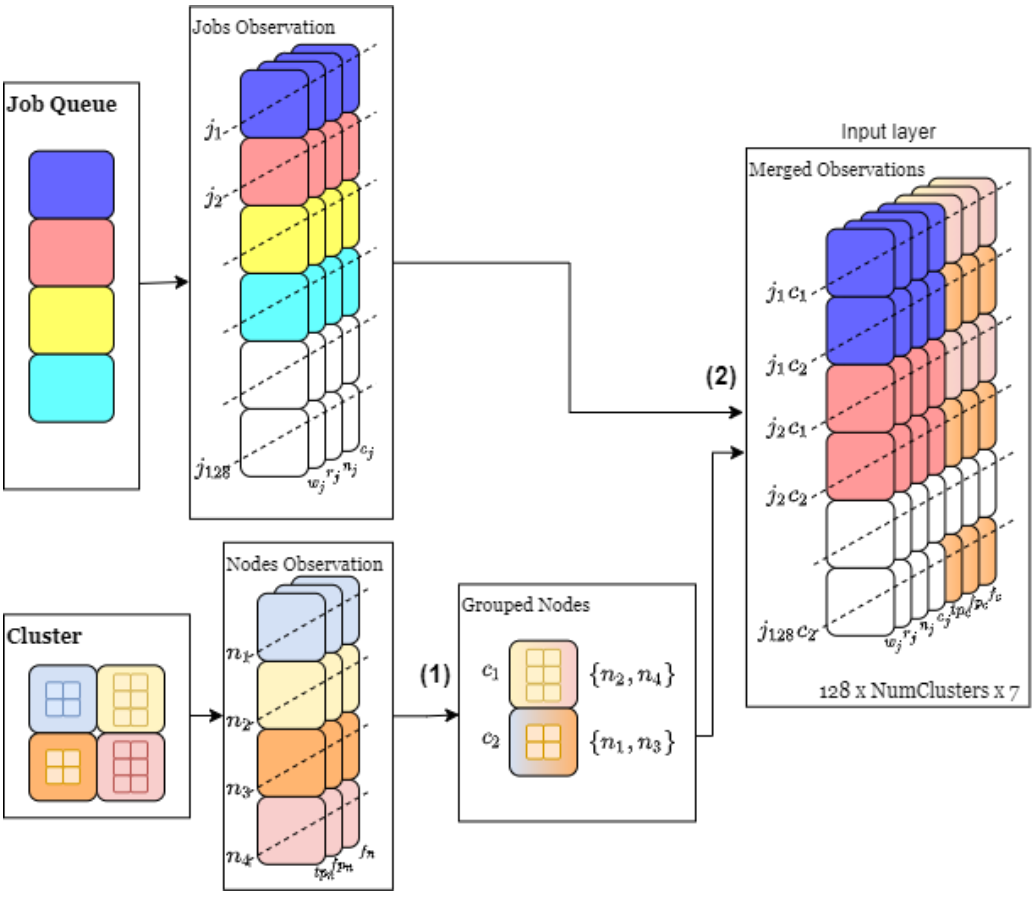

Figure 4.5: The process of reducing the size of a 4-nodes observation by grouping them in 2 clusters.

The values of each cluster for the observation are calculated as the mean value of all the nodes that are in that cluster, for each attribute. Thus, the shape is the same and the values summarize those of the nodes inside of it.

Apart from this change, the rest of the process is just the same that appears in Figure 4.4. For the mask, as there are no individual nodes to check if a job can be executed in them, the boolean use represents whether the job can be executed in *any* of the nodes of a given cluster. This ensures that once a *job-cluster pair* is selected, the job will be able to be immediately scheduled in at least one of the nodes of that cluster.

The final step after the selection has been made is to choose a specific node

for scheduling the job. This is done by simply checking which are the nodes belonging to that cluster, and allocating the first one that has enough free processors. This selection does not need any further considerations, as the assumption is that nodes in the same cluster are similar enough to make a difference.

By grouping the nodes in a fixed number of clusters, the size of the observation becomes a constant. Thus, it is possible to increase the number of nodes in the platform without raising the complexity of the agent's DNN. The number of clusters would depend on the architecture of the data center that is being simulated, and could be easily changed in case of an expansion by retraining the agent. However, as this situation is not quite common, it is considered a constant.

Chapter 5

Experiments & Evaluation

In this chapter, the experiments conducted for testing the efficiency of the architectures proposed in Chapter 4 are detailed. The results are compared with those of classic heuristic algorithms in order to check if any improvement has been achieved, and will be calculated according to three scheduling objectives: *slowdown*, *average bounded slowdown* and *average waiting time*.

5.1 Methodology and setup

In the previous chapter, two architectures for an intelligent scheduler agent were designed. Apart from the basic design, for each one of them some modifications have been thought in order to solve the theoretical problems of each implementation. In order to find out which one is the actual best design, not only the two basic agents will be tested but also their different variants.

For the results to be comparable, all the methods will be tested under the same conditions. These conditions depend mainly on the workload to schedule, the cluster in which to schedule it, and the objective to optimize. Thus, each architecture will be executed for each one of the objectives using the same workload and the same cluster definition. All the variables to consider during experimentation are the following:

- *Workloads*: they are generated from models defined in the Parallel Workloads Archive [14], which is a standard in the field of high-performance computing. In particular, the main one used for this project will be *Lublin, 1999/2003* [22]. The job trace used is composed of 10000 jobs, ordered by their submit time. The jobs are quite varied, as they can request from a few seconds of run time up to several days. Regarding the number of requested processors, the values go from 1 to 256. However, this had to be constrained to a maximum of 64 due to limitations explained at the end of Section 4.1.

- *Cluster*: the one defined for the experiments needs to meet certain characteristics. The most important one, for obvious reasons, is that is must be heterogeneous. Thus, it must have several nodes with different number of processors and frequency. Secondly, the number of nodes should not be too low nor too high. A low number would not show the real capabilities of the program, while one too high would make the scheduling trivial. In the experiments, 20 nodes with different attributes will be used.

- *Scheduling objectives*: the selected ones are *average bounded slowdown*, *average waiting time* and *slowdown*. The goal of the agent is to minimize these values in each execution. The definitions of each one of them are the following:

  - *Slowdown (SLD)*: ratio between the total time of a job in the system, from its submission to the end of the execution, and its requested time. This penalizes more the high waiting times for short jobs.
  - *Average bounded slowdown (BSLD)*: variant of the previous one that levels the execution times of the shortest jobs to avoid too large differences in similar jobs.
  - *Average waiting time (AVGW)*: average time interval between the submission of a job and its execution.

- *Heuristic algorithms*: to be comparable to the agents they must be able to select a job and a node as well. Thus, one algorithm will be used for choosing each. The policies used are summarised in Table 5.1. The notation used in the figures of these chapters will be *ab*, where *a* is the letter corresponding to the job policy and *b* is the one corresponding to the node policy. For instance, using algorithm *sb* means taking the smallest job at each time step and scheduling it in the biggest node.

- *Hyperparameters*: these are the values used to control the training process of the agents. The ones used in RLScheduler will be kept as they had already been tested and found as the best ones for this program. The most relevant ones are the learning rate α, with values of 0.0003 and 0.001 for the actor and the critic, respectively; and gamma γ equal to 0.99. The meaning of these parameters was described in Section 2.2.1.

- *Seed*: to ensure the equivalence of all the experiments the random seed will be fixed. This way, the same distribution of jobs will be chosen for scheduling during training every time. For testing, a fixed seed will also be used but a different one, so that the agents are always tested on unseen sets of jobs. The values used will be 2604 and 500, respectively.

Job Selection Policies		
Random	r	Random job from the job queue is selected
FCFS	f	Job with lowest submit time is selected
SJF	s	Job with lowest requested run time is selected
Smallest	l	Job with lowest requested number of processors is selected
Node Selection Policies		
Random	r	Random node is selected
Biggest node	b	Node with highest number of processors is selected
Fastest node	f	Node with highest frequency is selected

Table 5.1: Heuristic job and node selection policies used for comparing.

Experiments details

All the experiments will follow the same pattern. The agents will be trained during 100 epochs, and 20 trajectories per epoch (see 3.2). All of them will be trained on the same workload traces, ensured by the fixed random seed, and for the same objective, *average bounded slowdown*. Each trajectory will consists on scheduling 256 consecutive jobs extracted randomly from the complete workload. The cluster designed has 20 nodes, each one of them being a combination of 4, 8, 16, 32 or 64 processors, running at 2, 2.5, 3 or 3.5 GHz.

Then for the testing, the trained agents will have to schedule traces of 1024 jobs from the same workload, but generated with a different seed. They will

be tested for the same objective that they were trained for, and compared with the results of heuristic algorithms mentioned before. Each experiment is repeated 20 times to avoid obtaining wrong conclusions due to outliers.

The experiments will be conducted in two different systems. For the training of the agents, the Triton cluster from the ATC Group at Universidad of Cantabria will be used. Triton has six compute nodes, each of them formed by two processors Intel Xeon Silver 4114 CPU @ 2.20GHz and 64GB of RAM memory. Each processor has 10 physical dual cores with a shared L3 chache of 13.75KB. Then, the testing will be carried out in a commercial laptop. In particular, an MSI GF62 8RE-063XES with 6 physical dual cores i7-8750H with hyperthreading and a variable clock rate between 2.20 and 4.10GHz. This computer has 16GB of RAM memory and runs Ubuntu 20.04.2 LTS natively as operative system.

5.2 Double Agent experiments

The first set of experiments will focus on the Double Agent. In particular, the first one must ensure whether the different variants are learning correctly. As discussed in Section 4.3.1, it was not clear which was the best way to combine the losses of both networks for them to learn in parallel. Two approaches are tested: using the sum of both values and their mean. The other option was to use the losses separately. This is also tested in this first experiment, whose results can be observed in Figure 5.1. The goal of this experiment is to discard any agent that is not working properly due to its configuration.

This figure shows the results per epoch of the three agents: the double agent joining the losses of the networks by adding (ADD), the one joining them by doing the mean (MEAN), and the one using the losses separately (SEP). The values are the average results of the 20 trajectories performed during each epoch. As the three agents use the same seed, in the first epoch they all get the same result.

Once they start *learning* and thus taking different actions, the results become different. Around the third epoch, for example, the MEAN Agent does some bad choices and gets an BSLD of 78.4, meanwhile the other two have similar results to the first epoch. Nonetheless, as the epochs pass both the MEAN and the SEP Agent learn progressively and the improvement at the end is

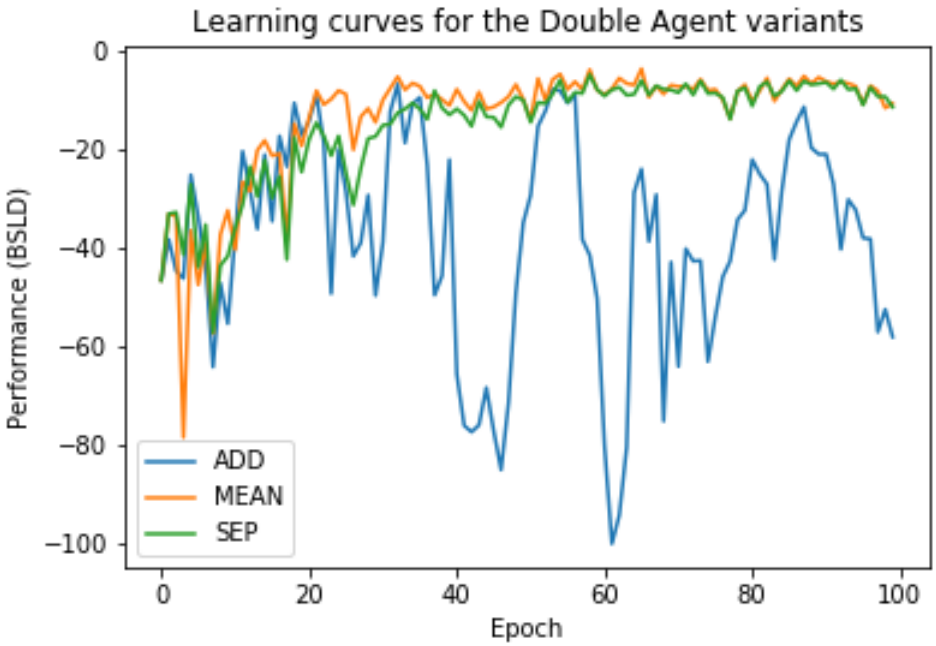

Figure 5.1: Evolution of the epoch results during training for each variant of the Double Agent optimizing Bounded Slowdown.

clear, when they are getting consistently values between 5 and 10. On the contrary, the ADD Agent seems to just be taking random actions and thus having highly variable results.

As the ADD Agent does not seem to be learning anything, it is discarded and the rest of the tests and the comparison with the heuristic algorithms will only be performed for the other two.

One relevant detail from the training is that these Agents take around 45 minutes. However, this amount of time is for 100 epochs, and as seen in Figure 5.1 the results do not improve much after the first 60, so half an hour would probably be enough. Moreover, the training phase is only done once and the Agent can be used until the scheduling needs change, so this amount of time is quite reasonable.

5.2.1 MEAN Double Agent

Once checked that the Agent is able to learn, it is time to perform the testing to see how optimal the results are. The Agent must schedule 20 different job traces, that will also be scheduled by the heuristic algorithms described in the the first section of this chapter. The boxplot shown in Figure 5.2 serves to visualize the results of each scheduler across the 20 executions.

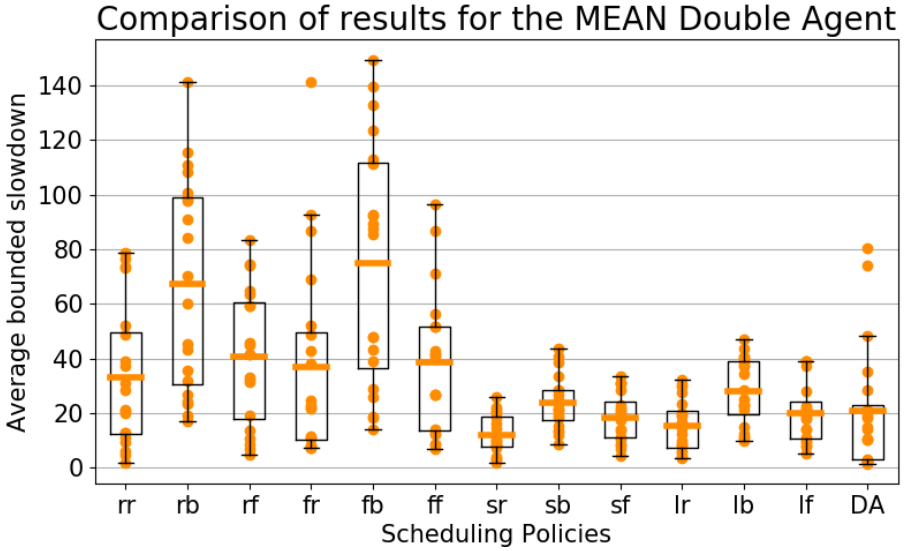

Figure 5.2: Comparison between the results of heuristic scheduling algorithms and the MEAN Double Agent when minimizing average bounded slowdown.

The results from the Double Agent (DA) are the ones at the far right. Even though it is not among the worst selection policies, and the values are more similar to the best ones, it still not as good as the best one. Something that can be concluded from this data is that job selections have a much greater impact on the performance than resource selections. Choosing a random job, or the first submitted job, results in a high bounded slowdown irrespective of the node selection.

As the literature states, *SJF* minimizes the slowdown and the MEAN Double Agent is not capable of producing a better result.

5.2.2 SEP Double Agent

The next step is to check whether the SEP Double Agent can compete with the *SJF*, or if its results are similar to the previous one. Even though Figure 5.1 showed that the learning results were quite similar, they both learnt in different ways. Thus, it is to be expected that each one of them has been trained to take different actions which might produce a different result for the experiment.

However, the results in Figure 5.3 show that this does not seem to be the case. When minimizing *average bounded slowdown*, *SJF* maintains the place as the best policy. The conclusion from the previous experiment apply to this one as well. And even if both agents are not taking the same actions, they are equally sub-optimal when compared to state-of-the-art algorithms. This is the case for the three objectives tested, as the results were similar for all of them.

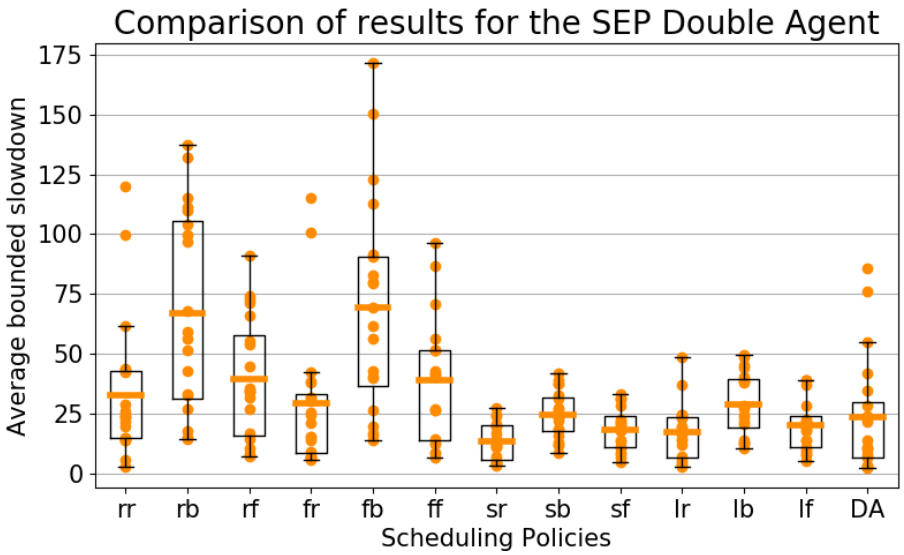

Figure 5.3: Comparison between the results of heuristic scheduling algorithms and the SEP Double Agent when minimizing average bounded slowdown.

5.3 Squared Agent experiments

As the Double Agent has not been able to produce good enough results, now the focus is on testing the Squared Agent (SA). The structure of this section will be the same as the previous one. As this Agent does not have any variants, the results for the three objectives will be shown. Then, they will be compared with the results when applying k-means to reduce dimensionality. This is to know whether the loss of information leads to a decrease of performance.

First, for checking that the learning process is correct for all the objectives, the training results are checked. They can be observed in Figure 5.4, with the values normalised so they can all be plotted at the same time, as each objective had values at different scales.

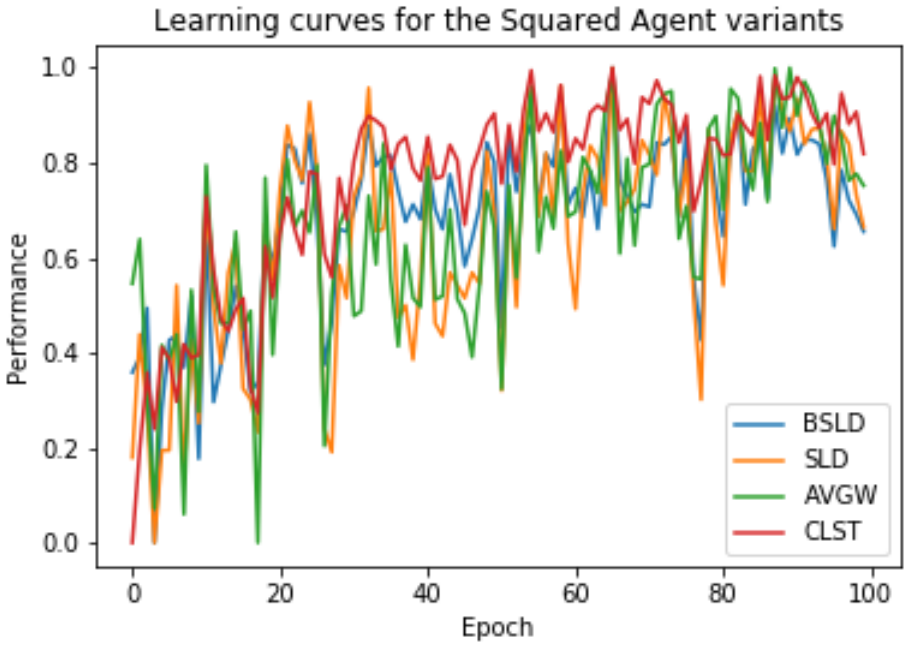

Figure 5.4: Evolution of the epoch normalized results during the training of the Squared Agent for each objective and while applying clustering.

The important aspect here is that the tendency is upwards for all of them, so they are progressively improving their results after the training epochs. In the figure, *BSLD*, *SLD* and *AVGW* represent the results of the Agent

for each one of the objectives, and *CLST* is when clustering is applied. In particular, it was tested for BSLD.

The training in this case was much more time consuming. In average, the Agents needed almost 12 hours to perform the 100 training epochs. Although as mentioned earlier, this process just needs to be done once, and it could be reduced by nearly half as the networks converge around epoch 60. Furthermore, the training of the agents could be parallelized in one or more GPUs, reducing the training time up to two orders of magnitude per GPU.

5.3.1 Minimizing SLD

The first objective to train this agent on has been *slowdown*. The rest of the configuration of the experiments is the same as seen so far. The results of this experiments appear in Figure 5.5. In general they are quite similar to the previous experiments, with a greater impact of the job policies and *SJF* being the best classic algorithm.

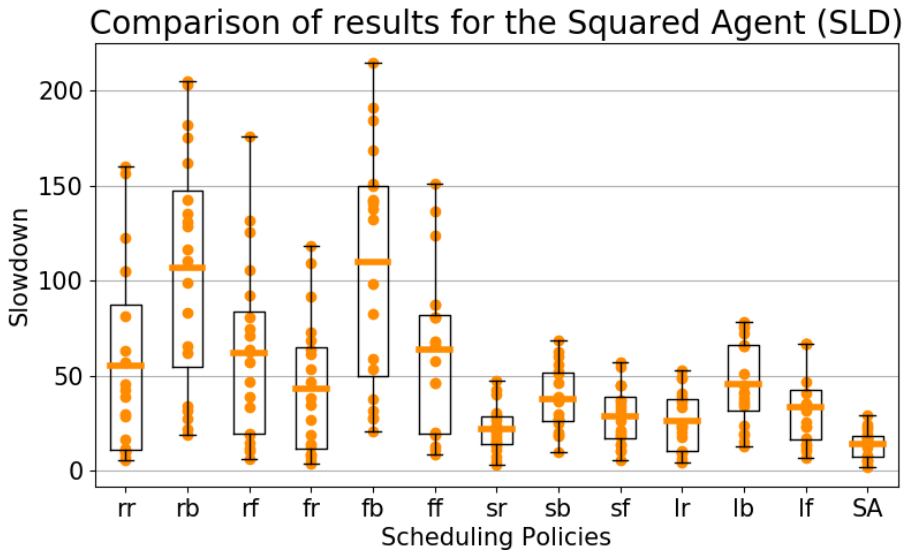

Figure 5.5: Comparison between the results of heuristic scheduling algorithms and the Squared Agent when minimizing slowdown.

The main difference is that in this case, the results obtained by the SA are better than those of any of the other algorithms tested. Compared to the

best heuristic algorithm, *shortest job - random node*, the SA is able to get an average slowdown closer to 0. Even though in the *best cases* the results of both are similar, the SA schedules the worst cases better, which reduces the average.

This is a really promising result as it proves that an intelligent scheduler can perform better in a heterogeneous cluster than the state-of-the-art algorithms, at least for minimizing the slowdown for now.

5.3.2 Minimizing BSLD

Average bounded slowdown is just a variation of the regular *slowdown* with more accurate results, so it is to be expected that the agent's performance remains the best in this case. Figure 5.6 shows exactly this. The values are different than in the previous experiment, but the results are almost the same.

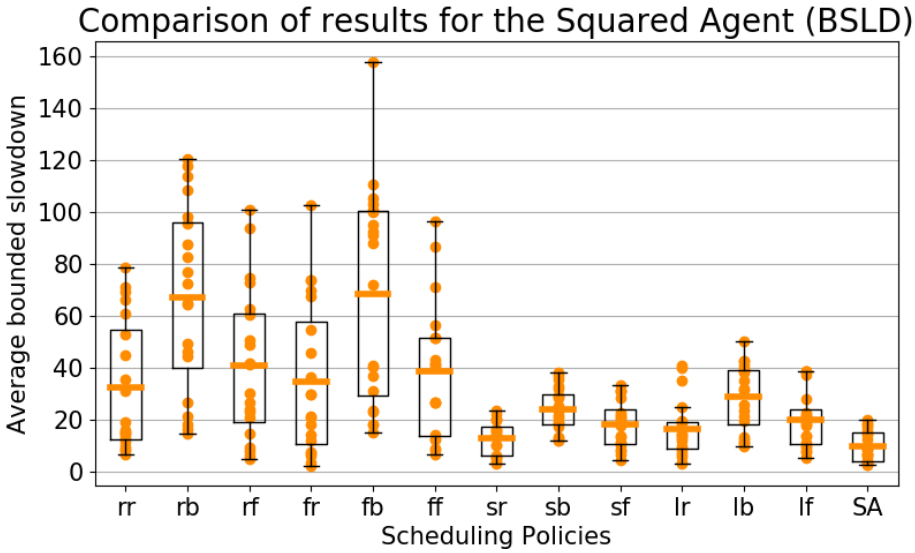

Figure 5.6: Comparison between the results of heuristic scheduling algorithms and the Squared Agent when minimizing average bounded slowdown.

As BSLD reduces the differences between smaller jobs with respect to regular slowdown, the values in this case are closer together and the difference between policies is smaller. Even so, it can be seen that the average result for

the Squared Agent is lower than the average for the best classic algorithm. Also, that the result in the worst case is just 20 seconds for the SA, the lowest across all algorithms.

This experiment proves that the way of measuring the slowdown does not affect the performance of this agent.

5.3.3 Minimizing AVGW

Finally, instead of measuring the slowdown, the goal of the agent is to minimize the *average waiting time* for the 1024 jobs from the test trace, to check that the agent still works for a different metric. Figure 5.7 shows the total amount of time (in seconds) that the 1024 jobs spent waiting in the queue for each scheduling policy.

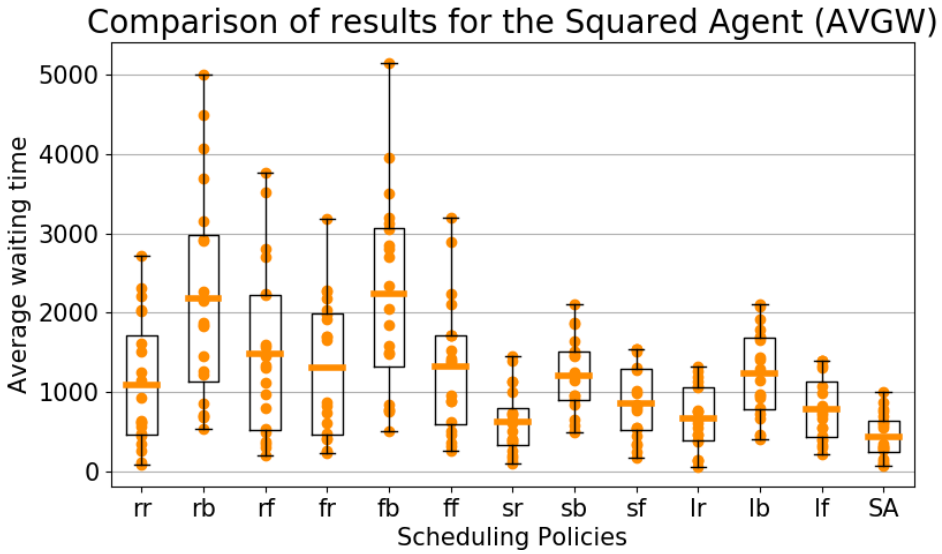

Figure 5.7: Comparison between the results of heuristic scheduling algorithms and the Squared Agent when minimizing average waiting time.

Even though the values are now higher due to how they are computed, the results are quite similar in general to those of the previous experiments. The best classic algorithm is now the one that selects the *smallest job first*, unlike in the previous cases. However, the Squared Agent is still the one taking the best actions to minimize the target objective. It has both the lowest median

value, below 500 seconds of waiting time for 1024 jobs, and the lowest worst cases for trajectories that are more difficult to schedule.

This proves that an intelligent agent is able to learn how to take scheduling actions and obtain better results than classic scheduling algorithms. And this can be done not for a single goal but to different ones, only constrained by the capabilities of the simulator in which it is working.

But not only could the agent run in a simulator, it could act as an actual scheduler in a real system. The main issue in this case is that a previous trace of the jobs executed in the system would be needed for training the agent beforehand.

5.3.4 Reducing complexity

As stated earlier, the main disadvantage of the Squared Agent architecture is that an increase in the number of nodes causes the complexity of the model to rise significantly. As this architecture has proved successful and a solution had been anticipated, it is now worth to test it. This experiment must ensure that using clustering for reducing the model's complexity does not cause a relevant loss of performance.

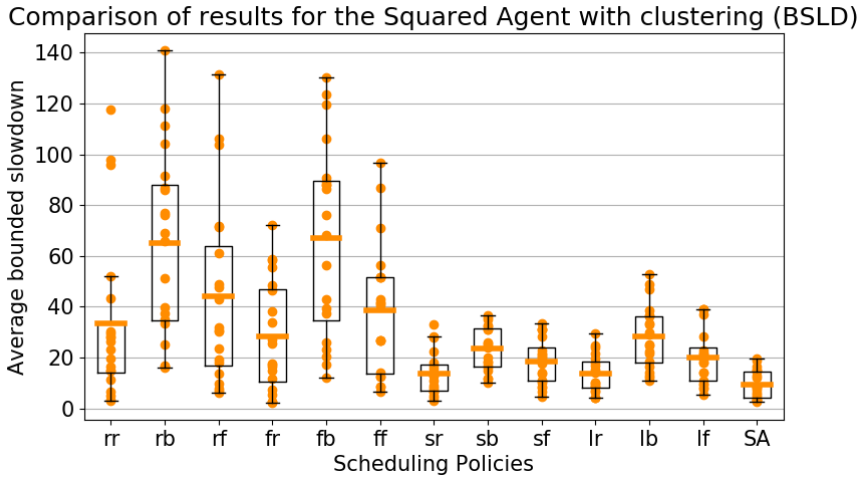

Figure 5.8: Comparison between the results of heuristic scheduling algorithms and the Squared Agent with clustering when minimizing average bounded slowdown.

This experiment has been carried out for the objective of *average bounded slowdown* and under the same conditions of all the others seen so far. In fact, the only different with the one seen in Section 5.3.2 is that clustering is now applied to group the 20 nodes in 10 clusters. This reduces the complexity of the agent's model by half.

This reduction does not have a very significant impact in terms of time, since the lesser time that the agent needs for training is compensated with the time that it takes to execute k-means each time. However, it does reduce the memory usage caused by the observations by half, which was an important limitation of the Squared model.

Figure 5.8 shows how this reduction has not had an impact on the performance of the agent, as the results are almost identical to those from Figure 5.6. Thus, this method can be applied in cases where the combined observation has become too large, and many of the nodes have the same or very similar characteristics.

Chapter 6

Conclusions

In this final chapter, the objectives achieved are highlighted, specially the implementation of an intelligent scheduler capable of overcoming the performance of state-of-the-art algorithms. Lastly, some ideas for future work that could be carried out using this project as the starting point are detailed.

6.1 Objectives Achieved

With the work presented in this paper, the objective of this investigation has been achieved. This was to take RLScheduler, and transform it into an heterogeneous simulator that allows for the testing of scheduling algorithms based on Deep Reinforcement Learning. This can be an important tool for researchers that might want to develop this research area further. In fact, it has already proven its utility with the development of the agents, which was a second objective of this work, and has opened up a new line of investigation for the group of Architecture and Technology of Computers at University of Cantabria.

In Section 1.3, three main objectives in particular were described for this project. In this section, the result for each one of those objectives will be summarised:

- *Modelling of heterogeneous resources*: with the changes made in Section

4.1, RLScheduler can now simulate an heterogeneous data center. The nodes that conform it can have a different number of processors each, and a different frequency. These values will have an impact on the simulation, and the performance of the jobs will vary depending on the node where they are executed. Even though the number of attributes of the nodes is still rather small compared to the real ones, it is a much more realistic representation of a data center than the original one. Furthermore, it would be fairly simple to add new attributes for the agent to consider, so this new model is also much more expandable. This addition of heterogeneity is in itself a great leap in quality in terms of simulation of data centers.

- *Implementation of the intelligent agent*: as the best architecture for the agent was not clear from the beginning, two different ones were designed and implemented in Section 4.3, the Double Agent and the Squared Agent. Both of them meet the necessary requirement that is to be able to select the best possible job-node pair among those available, instead of selecting just a single job.

- *Evaluation of the agents*: to ensure the correct learning of the agents, several experiments were conducted in Section 5. Then, a second set of experiments has been used to compare the results of the agents with those of classic scheduling algorithms. These experiments have proved that the Double Agent cannot compete with those heuristic algorithms, but the Squared Agent can. In fact, it has managed to get the best performance among all the algorithms for the three objectives tested. This means that the Squared Agent is able to do a better than the state-of-the-art scheduling.

These results are really promising not only because of the results themselves, but because of the possibilities that it opens up. With a more advanced simulator, the Squared Agent could very easily be extended to optimize other objectives like energy consumption, which would most probably result in an even greater advantage with respect to classic algorithms.

This is just the first step in a new line of research for the group of Architecture and Technology of Computers at University of Cantabria. There is still further work to do on the topic, but the utility of using DRL for implementing an intelligent scheduler has been proven, and the current implementation can be used as the base for this future work.

6.2 Future Work

In this last section, the main ideas to continue this work are highlighted. These should be the first steps for creating a truly realistic intelligent scheduler using the Squared Agent as the basis.

- *Migration of the agent*: RLScheduler has been a good starting point because it already implemented an intelligent scheduler with good results. Making the changes described in this document and then adapting the agent to scheduled in the new environment was easier than if it had been done directly in a more complex simulator. However, once that the heterogeneous scheduler is working properly, it would be recommendable to migrate it to a more complex and realistic simulator like *IRMaSim*. This would allow to get more precise results and also to be able to consider more information about the state of the data center in the observations.

- *Expansion of the observations*: the new simulator mentioned in the previous point should consider characteristics like memory, memory bandwidth or energy consumption. For the agent to be able to consider these for the scheduling, they should be included at the time of creating the *nodes observation*. The added information would lead to a more precise and efficient scheduling.

- *Definition of new objectives*: after the migration and the expansion, possibilities for new measurements would be available. An example would be the total energy consumption after an execution of the jobs trace. Making the agent minimize this new metric, or others, should be fairly simple and would make it a quite complete and realistic intelligent heterogeneous scheduler.

- *Acceleration of the training with GPUs*: with some changes in the code, the agent could be executed in one or more GPUs. This would allow to parallelize the training of the networks, obtaining a great reduction in the time it takes. This would be a major improvement, as training time is currently quite costly.

Bibliography

[1] Esteban Stafford Adrián Herrera Mario Ibáñez and Jose Luis Bosque. "A Simulator for Intelligent Workload Managers in Heterogeneous Clusters". In: *IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021.

[2] Adrián Herrera Arcilla et al. "HDeepRM: Deep Reinforcement Learning for Workload Management in Heterogeneous Clusters". In: (2019).

[3] Mario Ibáñez Bolado et al. "Gestión de recursos en Datacenters basado en Deep Reinforcement Learning". In: (2020).

[4] Vincent Kenny et al. *Heuristic algorithms*. 2014. URL: `https://tinyurl.com/4cy4mnrc` (visited on 05/08/2021).

[5] Sheraz Aslam et al. "Deep Learning Based Techniques to Enhance the Performance of Microgrids: A Review". In: Nov. 2019. DOI: `10.1109/FIT47737.2019.00031`.

[6] José Luis Bosque and L. P. Perez. "Theoretical scalability analysis for heterogeneous clusters". In: *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), April 19-22, 2004, Chicago, Illinois, USA*. IEEE Computer Society, 2004, pp. 285–292. DOI: `10.1109/CCGrid.2004.1336579`.

[7] José Luis Bosque et al. "A load index and load balancing algorithm for heterogeneous clusters". In: *J. Supercomput.* 65.3 (2013), pp. 1104–1113. DOI: `10.1007/s11227-013-0881-3`.

[8] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[9] Danilo Carastan-Santos and Raphael Y De Camargo. "Obtaining dynamic scheduling policies with simulation and machine learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–13.

[10] Balázs Csanád Csáji et al. “Approximation with artificial neural networks”. In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24.48 (2001), p. 7.

[11] Zhang Di et al. *The v0.1 release of RL-based Batch Job Scheduling*. Version v0.1. June 2020. DOI: 10.5281/zenodo.4009286. URL: https://doi.org/10.5281/zenodo.4009286.

[12] Shanket Doshi. *Various Optimization Algorithms For Training Neural Network*. 2019. URL: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6 (visited on 05/13/2021).

[13] Dror G Feitelson. “Parallel workload archive”. In: *https://tinyurl.com/pcyrmf4a* (2007).

[14] Dror G Feitelson, Dan Tsafrir, and David Krakov. “Experience with using the parallel workloads archive”. In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2967–2982.

[15] The Python Software Foundation. *queue - A synchronized queue class*. 2021. URL: https://docs.python.org/3.7/library/queue.html (visited on 06/19/2021).

[16] Diego García-Saiz, Marta E. Zorrilla, and José Luis Bosque. “A clustering-based knowledge discovery process for data centre infrastructure management”. In: *J. Supercomput.* 73.1 (2017), pp. 215–226. DOI: 10.1007/s11227-016-1693-z.

[17] Mohamad H Hassoun et al. *Fundamentals of artificial neural networks*. MIT press, 1995.

[18] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. “Learning to learn using gradient descent”. In: *International Conference on Artificial Neural Networks*. Springer. 2001, pp. 87–94.

[19] Russell Impagliazzo and Gábor Tardos. “Decision versus search problems in super-polynomial time”. In: *30th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society. 1989, pp. 222–227.

[20] Vijay R Konda and John N Tsitsiklis. “Actor-critic algorithms”. In: *Advances in neural information processing systems*. Citeseer. 2000, pp. 1008–1014.

[21] Sergei Leonenkov and Sergey Zhumatiy. "Introducing New Backfill-based Scheduler for SLURM Resource Manager". In: *Procedia Computer Science* 66 (2015). 4th International Young Scientist Conference on Computational Science, pp. 661–669. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.11.075. URL: https://www.sciencedirect.com/science/article/pii/S1877050915034249.

[22] Uri Lublin and Dror G Feitelson. "The workload on parallel supercomputers: modeling the characteristics of rigid jobs". In: *Journal of Parallel and Distributed Computing* 63.11 (2003), pp. 1105–1122.

[23] James MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[24] Hongzi Mao et al. "Learning scheduling algorithms for data processing clusters". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 270–288.

[25] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM workshop on hot topics in networks*. 2016, pp. 50–56.

[26] Stathis Maroulis, Nikos Zacheilas, and Vana Kalogeraki. "A Holistic Energy-Efficient Real-Time Scheduler for Mixed Stream and Batch Processing Workloads". In: *IEEE Transactions on Parallel and Distributed Systems* 30.12 (2019), pp. 2624–2635.

[27] Jason Mars, Lingjia Tang, and Robert Hundt. "Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity". In: *IEEE Computer Architecture Letters* 10.2 (2011), pp. 29–32.

[28] Ahuva W. Mu'alem and Dror G. Feitelson. "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". In: *IEEE transactions on parallel and distributed systems* 12.6 (2001), pp. 529–543.

[29] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012, p. 35.

[30] Judea Pearl. "Heuristics: intelligent search strategies for computer problem solving". In: (1984).

[31] Michael Pinedo. *Scheduling*. Vol. 29. Springer, 2012.

[32] Mireille Raby and Christopher D Wickens. "Strategic workload management and decision biases in aviation". In: *The International Journal of Aviation Psychology* 4.3 (1994), pp. 211–240.

[33] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[34] Elham Shamsa et al. "Concurrent Application Bias Scheduling for Energy Efciency of Heterogeneous Multi-Core platforms". In: *IEEE Transactions on Computers* (2021).

[35] Sagar Sharma and Simone Sharma. "Activation functions in neural networks". In: *Towards Data Science* 6.12 (2017), pp. 310–316.

[36] David B Shmoys and Éva Tardos. "An approximation algorithm for the generalized assignment problem". In: *Mathematical programming* 62.1 (1993), pp. 461–474.

[37] Esteban Stafford and José Luis Bosque. "Improving utilization of heterogeneous clusters". In: *J. Supercomput.* 76.11 (2020), pp. 8787–8800. DOI: 10.1007/s11227-020-03175-4.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018, pp. 59–60.

[39] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018, p. 12.

[40] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[41] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015, pp. 184–185.

[42] Wei Tang et al. "Fault-aware, utility-based job scheduling on blue, gene/p systems". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE. 2009, pp. 1–10.

[43] Jeffrey D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System sciences* 10.3 (1975), pp. 384–393.

[44] Jake VanderPlas. *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.", 2016. Chap. 5.11.

[45] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[46] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[47] Di Zhang et al. *RLScheduler: An Automated HPC Batch Job Scheduler Using Reinforcement Learning*. 2020. arXiv: 1910.08925 [cs.DC].

[48] Henan Zhao and Rizos Sakellariou. “Scheduling multiple DAGs onto heterogeneous systems”. In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 14–pp.