UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Abner Sousa Nascimento

An Investigation on Deep Reinforcement Learning Algorithms for Resource Management and Workload Scheduling

Belo Horizonte
2022

Abner Sousa Nascimento

An Investigation on Deep Reinforcement Learning Algorithms for Resource Management and Workload Scheduling

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Luiz Chaimowicz

Belo Horizonte
2022

Nascimento, Abner Sousa.

N244i An investigation on deep reinforcement learning algorithms for resource management and workload scheduling/ Abner Sousa Nascimento – 2022.
1 recurso online (75 f. il., color.) : pdf.

Orientador: Luiz Chaimowicz.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 70-75.

1. Computação – Teses. 2. Aprendizado do computador – Teses. 3. Aprendizado profundo – Teses. 4. Computação de alto desempenho – Teses. 5. Aprendizado por reforço – Teses. I. Chaimowicz, Luiz. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*82(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

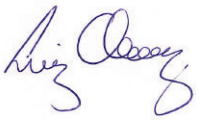
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

An Investigation on Deep Reinforcement Learning Algorithms for Resource Management and Workload Scheduling

ABNER SOUSA NASCIMENTO

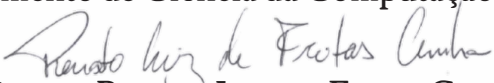
Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:



PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ANDERSON ROCHA TAVARES
Instituto de Informática - Universidade Federal do Rio Grande do Sul

PROF. GEORGE LUIZ MEDEIROS TEODORO
Departamento de Ciência da Computação - UFMG



Doutor RENATO LUIZ DE FREITA CUNHA
Software Engineer - Microsoft Research

Belo Horizonte, 21 de dezembro de 2022.

Acknowledgments

Throughout every life-changing decisions involved in the pursue of this degree, the nurturing and support of my family was a stepping-stone that firmly led me to achieve every goal. I cannot, hence, begin to express my gratitude for my loving mother Elisangela, my encouraging father Edivan and my supportive brother Aminadabe. Their unconditional love gave me stamina to overcome the challenges, solace to embrace my failures and the purest joy while celebrating the victories. I extend my gratitude to my beloved Danilo, who stood by my side to soothe my anxieties and comfort me with his love.

I'm indebted to Federal University of Minas Gerais and the Graduate Program in Computer Science for giving me the opportunity to extend my education in the knowledge domain I chose to explore. I'd also like to extend my deepest gratitude to my advisor, professor Luiz Chaimowicz, whose guidance and understanding were fundamental from the first day to the last; and Renato Cunha, whose research was basal to this work. I sincerely thank professor Ialis Cavalcante, whose years of mentorship in my *alma mater* Federal University of Ceará continue to inspire me to this day and will for the remaining of my life. Special thanks for professor Jermana Lopes and professor Wendley Silva, who supported my decision of taking the next academic step at UFMG.

This achievement would not have been possible without the love and support of my lifelong friends that accompanied me through the struggles, filled my heart with joy and laughter that made even the hardest burdens far easier to be carried. There are no words to express my appreciation for Mauro, Raquel, Thales, Elisabete, Luanderson and Evangelista. Our mutual love and support knows no barriers and cannot be weakened by distance, they will forever be my chosen family.

I'd also like to thank the support from the people I met since this journey began: Gustavo and Pedro, who shared with me their immensely valuable companionship and intelligence; Evelyn, who solicitously attended to me before I started my studies; Sheila, who helped me to keep order through the chaos of the last few weeks; and many others, either virtually or in person. I truly believe in the power of human connections and consider myself very blessed to be surrounded with so much goodness in life.

"There's a real danger of systematizing the discrimination we have in society [through AI technologies]. What I think we need to do — as we're moving into this world full of invisible algorithms everywhere — is that we have to be very explicit, or have a disclaimer, about what our error rates are like."

(Timnit Gebru)

Resumo

A eficiência é um requisito operacional fundamental para a maioria dos sistemas computacionais, visto que os recursos necessários para tais processos geralmente estão sujeitos a restrições de disponibilidade. É desejável que os clusters de computação operem para concluir o maior número possível de tarefas enquanto aproveitam ao máximo os componentes de hardware, por exemplo, CPU e memória. Nesse contexto, a ordenação temporal das tarefas submetidas a um cluster pode interferir na sua capacidade de funcionar em uso máximo. É, portanto, importante que tais tarefas sejam programadas adequadamente para garantir a eficiência. Vários algoritmos e técnicas, tanto orientadas por regras fundamentais quanto baseados em aprendizado de máquina, podem ser aplicados a esse problema, mas a natureza centrada em objetivos do aprendizado por reforço amplificada pelo uso de redes neurais profundas pode ajudar a lidar com as particularidades e complexidades dele de forma robusta. Neste trabalho, investiga-se o uso de técnicas de aprendizado por reforço e redes neurais profundas para alocação de tarefas em clusters computacionais, aplicando busca pelo conjunto ideal de hiperparâmetros e comparando o desempenho e a estabilidade de treinamento das soluções baseadas em aprendizado com algoritmos previamente projetados, com referência a uma métrica alvo. Os resultados apontam que é possível obter desempenho igual ou melhor, desde que sob as condições ambientais corretas e dentro do domínio paramétrico apropriado. Observa-se também que tais agentes podem alcançar melhor generalização se treinados em uma configuração de dificuldade graduada, com cenários cada vez mais desafiadores, em vez de uma abordagem de inicialização aleatória que parte de uma configuração difícil.

Palavras-chave: aprendizado por reforço; aprendizagem profunda; computação de alta performance; gerenciamento de carga de trabalho; gerenciamento de recursos.

Abstract

Efficiency is a key operational requirement for most computer systems, given that the resources necessary to such processes are usually subjected to constraints in availability. It's desirable that computing clusters operate in order to complete as many tasks as possible while making the most of hardware assets, for example, CPU and memory. In this context, the temporal ordering of the jobs submitted to a cluster can interfere in its capacity to function at maximum use. It is, thus, important that such tasks are scheduled properly to ensure efficiency. Several algorithms and techniques, both principled and learning-based, can be applied to this problem, but the goal-oriented nature of reinforcement learning powered by the use of deep neural networks can help deal with the particularities and complexities of it robustly. In this work, we investigate the usage of deep reinforcement learning techniques for job allocation in computing clusters, applying hyperparameter search and comparing the performance and training stability of the learning-based solutions with previously designed algorithms for a target metric. We found that it is possible to obtain equal or better performance under the right environmental conditions within the appropriate parametric domain. Results also indicate that such agents can achieve better generalization if trained in a graduated difficulty set-up, with increasingly challenging scenarios, instead of a random initialization approach that starts from a difficult configuration.

Keywords: reinfocement learning; deep learning; high power computer clusters; workload management; resource management.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Workload management

For most computer systems, completing the task for which they were projected while demanding the least amount of resources is a desirable goal. Apart from abstract scenarios in theoretical experiments, the supplies of any system are finite and subjected to constraints, whether they come from operational limitations or other aspects intrinsic to any application, such as time sensitivity and business logic. Efficiency is, thus, a key parameter of a system's performance whose concern spans through the entire procurement cycle, from the input, through processing, storage and output of information. Improving over efficiency requires, then, careful evaluation tailored to each demand and part of an architecture, in addition to comparisons with the other solutions considered during development [19].

Several are the factors that influence the performance of a computational system, each one coerced by its own set of restrictions that, in turn, depend highly on the application to which said system is dedicated. In order to function, from high-capacity servers, to aircraft control, robots, smartphones, etc., every system demands a certain set of resources that are subjugated to operational constraints, whether from limited hardware capacity and energy availability. Namely, processing units and memory space availability can be obvious limiting factors in computational operations, but overall performance itself can also depend on the work to which hardware appliances like these are subjected [51]. A system can have little processing power, but be capable of performing its own work more efficiently, proportionally, than powerful, but poorly optimized ones can execute theirs, depending not only on the characteristics of the tasks, but also on the nature of the problem itself and the system's capacity of properly applying its own resources [11].

Thus, to ensure maximum performance, allocation plans must be engineered carefully, in observation to all the aforementioned aspects. As the complexity of the environment and the number of variables to consider in optimization grows, so must do task execution plans; with decisions informed by knowledge about the workload, regarding the

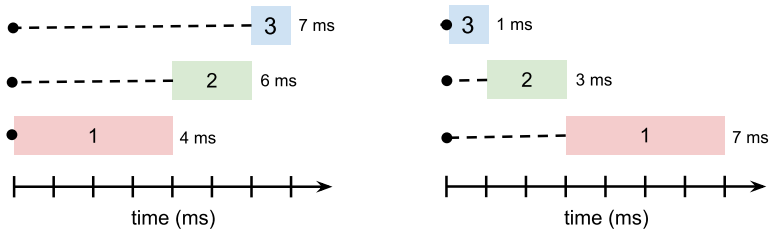
amount and nature of the typically demanded resources.

Common examples of scenarios in which workload management is a particularly important optimization problem include parallel computing, where the distribution of jobs among CPU cores must ensure all of them are loaded with a balanced amount of work over time [26]; and server systems, which often operate in highly dynamic workload scenarios that alternate quickly from low to high request rates [7]. Furthermore, as the demand for cloud computing services increases with the development of data-centered industry solutions, efficient resource allocation in these contexts becomes a prominent subject to many other economic sectors [36]. In such data-intensive scenarios, where the rate and volume of data to process can start to challenge the limits of the currently available technology, special systems that intensively rely on parallelization and distribution of resources, namely High-Performance Computing Clusters, or HPCCs, are necessary [14].

In essence, HPCCs can be roughly compared to a batch computing system, with individual processing nodes that collectively execute a set of tasks, often expected to take a long time with little to no user intervention in the process. In the early days of operating systems, batches of tasks were executed simply in the order they were submitted, lined up in a non-preemptive queue, a process known as first-come, first-served (FCFS) [43], with no need for prior information about the tasks, like the estimated running time. Albeit simple, this approach has several disadvantages, since it can yield high average turnaround – the difference between the time a job was completed to the time it was submitted – with short tasks being held back in the queue by long-running tasks. If, however, run times are known in advance, or can be reasonably estimated, a scheduling strategy that prioritizes short jobs, that is, shortest-jobs-first (SJF), can reduce average turnaround by relegating jobs that block the execution queue for longer to the end of the wait list [51].

Consider the simple scenarios illustrated in Figure 1.1, with a single processing unit and 3 jobs. In FCFS, jobs were executed in the same order they were enqueued, the shortest ones at the end have to wait until the longest ones are completed. Average turnaround time is $(4 + 6 + 7)/3 = 5.66$ milliseconds. SJF, on the other hand, provided with the expected running times of each job, schedules the shortest ones first. Average turnaround time becomes, then $(1 + 3 + 7)/3 = 3.66$ milliseconds. The scheduling plan can affect how long each process will have to wait between submission and completion.

This simple example goes on to demonstrate the importance of a well-designed scheduling scheme, but HPPCs are far more complex environments. They contain multiple nodes working in a parallelized fashion, which adds a new dimension to the operation. Moreover, they run under highly dynamic circumstances, with variance in job sizes, frequency, and rate of submission. Principled strategies like SJF may fail to meet the requirements for a sufficiently optimized allocation plan in these scenarios. Analysis of previously collected data about the events involved in a typical runtime session of a system can support the development of heuristics and models [11]. However, factors such



(a) FCFS (b) SJF

Figure 1.1: Comparison between average turnaround time for a scheduling regime based on a First-Come, First-Served (FCFS) fashion and one based on Shortest-Jobs-First. In each scenario, 3 jobs were submitted at the same time, but queued in different orders. Turnaround time is indicated by the numbers on the right of each job.

as sudden load variations, noisy data and the scale of the environment in which a system operates can degrade model fitness, making the problem hard to tackle in a principled way [27].

1.2 Learning-based approaches

Learning-based approaches can offer alternative solutions to the challenges typically involved in workload control. Artificial intelligence methods like genetic algorithms and fuzzy logic have been successfully employed for that purpose in the past [13, 8]. In particular, reinforcement learning (RL) provides an unsupervised, goal-oriented approach specially suitable to circumvent the difficulties of resource management tasks. Overall, the process consists of training an artificial agent to achieve a specific objective by exploring the possible actions it can take within its environment, observing their outcomes (rewards) and learning from these experiences [47]. As a result, agents essentially become automated machines, capable of decision-making in a flexible and adaptable framework [27]. Moreover, advances in deep neural networks applied to reinforcement learning have achieved remarkable optimization goals, with agents developing efficient behavior policies in many complex scenarios [32, 45, 3, 20].

1.3 RL-based Workload Management

Given the broad range of sophisticated applications to which deep reinforcement learning has been applied, it's also expected that the principles involved in automated agents could be applied with interesting results on the domain of workload management, task scheduling and resource control [27, 28]. In order to do so, the implications of such frameworks must be considered carefully: inherent descriptive aspects to the problem need to be modelled as neural network inputs; environments must be designed considering the functional logic and business rules of the entities involved in the scheduler-centered application; all of this considering implementation details, such as the appropriate set of parameter values for the underlying algorithms that ultimately guide the learning process, or the most adequate architectures and training protocols for the networks used to model the agent's reasoning. Finally, the reproducibility and testability of such methods must be built over solid software foundations, in order to ensure that they can be comparable to other solutions as well as prevent overfitting to the conditions they were trained upon [53]. This work aims to explore the applicability of RL to this problem, investigating the aspects involved in its viability, how hyperparameters can influence on training and evaluation performance, compare the resulting agents to other principled methods and analyze how challenging environment conditions can be mitigated with the appropriate protocol for the development of learning-based solutions.

1.4 General Objectives

This study holds as a general objective to investigate deep reinforcement learning techniques to the task of workload management, exploring difficulties and limitations related to these approaches and how they compare to other existing methods that are not established upon an automatic learning foundation. Specifically:

1. Develop deep reinforcement learning agents applied to the task of workload management, particularly for applications where principled techniques and manually engineered heuristics are usually employed. Compare the performance of learning-based and traditional approaches under similar conditions regarding a well-defined metric, namely average slowdown.

2. Explore the challenges associated with designing learning-based workload sched-

ulers, including the modeling constraints required for effective training and the inherent limitations of the problem. Consider how these challenges might impact the practical application of these techniques. Examine potential approaches to address these obstacles and enhance generalization in diverse operational scenarios, where performance improvements are valuable but may be more difficult to achieve.

3. Implement carefully designed evaluation procedures that emphasize statistical confiability and reproducibility, mitigating issues that derive from the highly stochastic nature of deep reinforcement learning and may hinder the reliability of results interpretation.

1.5 Contributions

The following results are expected to be achieved upon the completion of the aforementioned objectives:

- Develop deep reinforcement learning agents for workload management that are comparable to principled techniques or outperforms them.
- Evaluate the most efficient solutions to commonly encountered problems in deep RL for task scheduling, eventually developing better approaches as well.
- Contribute to future research in this domain through comparison and evaluation of state-of-the-art techniques. Collaborate on the development of community-available software tools, improving over existing open source frameworks or implementing new appliances.

1.6 Document Structure

This document is structured as follows: In Chapter 2, the definitions and implementations of the studied algorithms are laid-out in detail, with equations derived from the core principles of Reinforcement Learning, the motivation behind the development of each one of them and the engineering techniques each solution employed. Chapter 3 encompasses past developments that laid the foundation of the concepts explored throughout

this work, spanning from resource management techniques to the modeling of environments for scheduler learning. The overall strategy adopted in order to test and compare all the studied scenarios is described in Chapter 4, with their respective results listed on Chapter 5. Finally, the final considerations obtained from the results and the guidelines for future work are established on Chapter 6.

Chapter 2

Background

2.1 Workload modeling and representations

Workload can be understood as the amount of work to which a system is submitted. This broad definition allows different types of workload to manifest in a variety of domains. For example, in a network server, workload can be represented by requests for a web page; in database applications, queries that trigger I/O operations [11]. For the task scheduling in high power computing units problem, workload is defined in similar terms to operating systems, with jobs submitted by users or other types of system calls. Jobs are represented by a set of parameters, which have been standardized in the literature in a format that describes consistently most of their relevant characteristics, the Standard Workload Format (SWF) [12]. The data fields represented in SWF are described in Table 2.1.

Naturally, not all the fields described by the SWF format are relevant to every application or simulation. For the reinforcement learning environment used to train a learning-based scheduler studied in this work, sched-rl-gym [27][9], the most relevant fields are requested number of processors, requested time and requested memory, that provide the scheduler an overview of the size of the submitted job. In an artificially generated workload log, these fields can be simulated with a certain degree of error, to represent the error that occurs naturally on user estimations. *A posteriori* data fields, *i.e.*, the ones that can only be referred to after a job is completed, such as submitted time, wait time and run time are also useful to compute performance metrics of the scheduler, based, for example, on the average time a job has to wait for others before it's eventually scheduled and other events.

Job field	Description
Job number	A descriptor for identification.
Submit time	The time in seconds at which each job has been submitted.
Wait Time	The difference between the job's submit time and the time it actually began to run.
Run time	The difference between the end time and start time of a job.
Number of allocated processors	Number of processors a job uses.
Average CPU time used	Average over all the CPU time a job has used.
Used memory	Average used memory by a job in kilobytes, for each processor.
Requested number of processors	Number of processors a job requested to run.
Requested time	Either the run time or the average CPU time requested by a job before its execution.
Requested memory	Amount of memory a job requests upon submission, in kilobytes.
Status	Identifies the status of the job after completion: 1 if the job was completed, 0 if it failed, 5 if canceled.
User ID	Identification number of the user that submitted the job.
Group ID	Identification number of the group to which belongs the user that submitted the job.
Executable number	Identification of the executable application associated with the job.
Queue Number	An identification number for systems with multiple submission queues.
Partition number	Descriptor to identify the partition of the system that executed the job. The definition of partition varies with the system, for example, the computer within the cluster.
Preceding job number	Number of a preceding job in the workload from whose conclusion the current job depends.
Think time from preceding job	Maximum number of seconds that should elapse between the termination of the preceding job and the beginning of the current one.

Table 2.1: Job fields described in the Standard Workload Format.

Resources refer to the limited supply of assets that are available for a system to carry through with its workload. Likewise, their types depend on the domain of each

application. For a computing cluster, resources can be processing units; on a file system, disk space would be a more concerning attribute; network-centered applications are usually more sensitive to bandwidth or channel availability. Extrinsic entities can also be considered resources, such as budget and energy, considering that they can also be constraining factors to the operability of any system. In the job scheduling problem encapsulated by sched-rl-gym, resources can be taken as the available number of CPUs and memory space of a cluster. These fields are defined by environment parameters.

The main optimization metric offered by sched-rl-gym and focused in this study is the average slowdown ψ, given by the ratio $\psi_j = C_j/M_j$, in which $C_j = c_j - s_j$ is the difference between the time a job has been submitted (s_j) and when it was finally completed (c_j). M_j represents the time necessary to execute the job. Thus, this metric essentially normalizes the time the job actually takes by the time it was expected to be completed, which helps to mitigate reward bias against large jobs.

A common attribute exists for all workload types that alludes to the amount of operations required per unit of time: their *events rate*. Characterizing load by rate instead of amount enforces the notion of continuous work over time, rather than a finite set of operations. In sched-rl-gym, the job rate is defined as the probability of a job being submitted at each time step of the simulation. Jobs are generated from a Bernoulli distribution, in which the probability p determines the likelihood of a new job being submitted at each step.

Sched-rl-gym samples job sizes from different uniform distributions, one for small jobs, with tighter size bounds, and another for large jobs, with wider size bounds. All of these values are controlled via environment parameters. Both job rate and job size are important variables that determine how crowded and tractable the state representation will be. Since they're associated with the user estimated fields from SWF, it's also possible to introduce gaussian noise to the size of the jobs, in order to simulate human error.

2.2 State Representation

Management of resources through workload scheduling consists in maximizing the exploitation of the available assets by distributing workload over time in an ordered fashion. It is, thus, a concept inherently associated with the efficiency of a system, evaluated by the amount of useful output it produces, given the resources initially provided. Therefore, even though some level of resources waste is typically inevitable for every application, it must be minimized in order to achieve maximum efficiency, specially on the discrete domains of most computing systems.

State in sched-rl-gym is represented by an image-like view of the current jobs scheduled in the slots of the execution queue. A limited number of slots is necessary in order to keep the input size and action space consistent. The backlog provides a summarized information of the number of jobs set to be allocated next – with no additional information other than the presence of a job. The representation also provides an overview of the execution plan for the next few time steps, expressed along the second dimension. In Fig. 2.1 and Fig. 2.2, a scheduler with 4 slots and 4 resources – can be either CPU or memory, since both are modeled the same way – is represented. Slot 1 of Fig. 2.1, for example, contains a job scheduled to run for 1 unit of resource, over 1 moment in time. Its backlog shows that there's only one job waiting to be allocated, even though no additional information about such job is provided just yet. The job waiting on the backlog may eventually be allocated to Slot 4 in a future moment, if its requirements fit that slot's constraints. It's worth noting that the sparse situation illustrated in Fig. 2.1 provides less correlated samples over time, which makes this scenario easier to be treated by neural networks. This contrasts with the situation illustrated in Fig. 2.2, in which large and frequent jobs result in filled inputs that tend to repeat themselves with the passage of time.

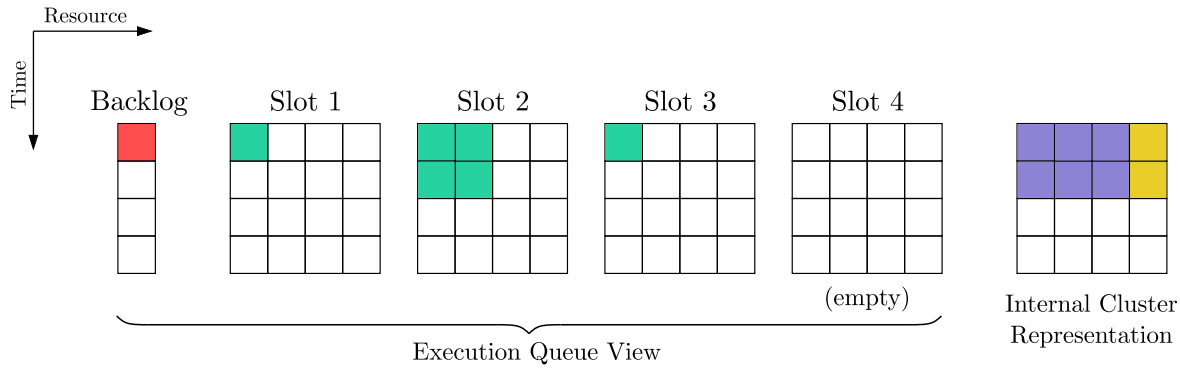


Figure 2.1: A slice view of a state representation from sched-rl-gym. Each job is allocated within a queue of slots, before their time to actually use cluster resources. In a sparse load situation, *i.e.*, smaller, less frequent jobs, the image representation of the state is sufficiently spaced to provide good diversity among samples.

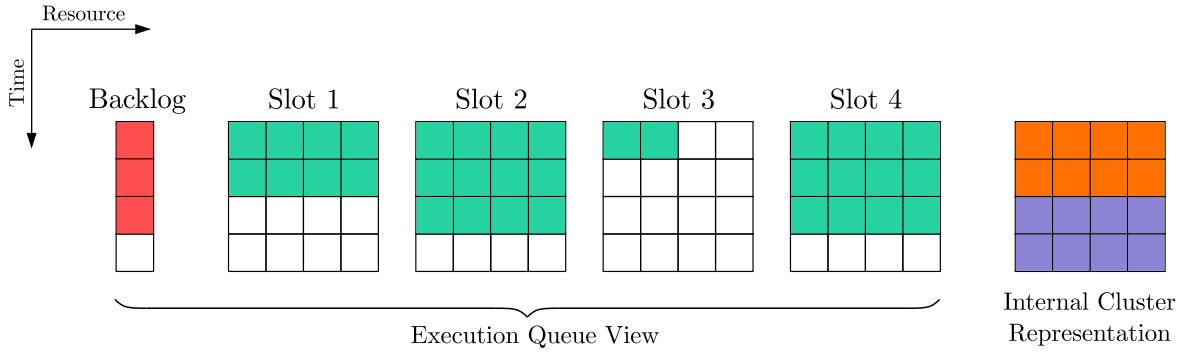


Figure 2.2: A slice view of a state representation from sched-rl-gym for a dense workload situation, *i.e.*, bigger, frequent jobs. In this scenario, the image representation of the state is compact and samples correlate over time.

2.3 Reinforcement learning

Reinforcement learning (RL) is a machine learning paradigm characterized by a central learning problem: given a state and a set of possible actions to take, what steps should an autonomous agent follow in order to maximize a numerical reward? Much like in the learning process that happens organically in nature, an agent has no or very little prior information about the steps it should follow towards its goal. They should be discovered via exploration of the environment and observation of the outcome of the chosen actions. As the agent gains more experience from its exploration endeavors, convergence towards a satisfactory objective should be achieved [47].

2.3.1 Elements of a Reinforcement Learning Framework

Usually, a RL agent interacts with an extrinsic environment over a sequence of discrete time steps. At each instant, a state $s \in S$ represents all the present information an agent can observe from its environment, from the set of all possible observations. The actions set $A(s)$ express all the possible actions a the agent can execute at a given state s. Finally, the transition function $\mathcal{P}^a_{ss'}$ denotes the probability of reaching a state s' by executing an action a at a state s. Therefore, this representation is essentially a mathematical model for stochastic decision-making, *i.e.*, a Markov Decision Process [47].

A sequence of state-action pairs $T = [(s_1, a_1), (s_2, a_2), \cdots, (s_t, a_t)]$ is named a trajectory. Besides states, actions and transition functions, there are a few other elements essential to any reinforcement learning solution, namely, a reward function, a value function, a policy and a model.

The reward function $R(s_t, a_t)$ denotes the immediate expected reward for executing action a_t at a state s_t and a given time instant t. Following the example of sched-rl-gym, given a set J of jobs in the system, the reward signal at each simulation step is given by $\sum_{\forall j \in J} -1/M_j$. Thus, over the number of time steps necessary to run each job, this sum accumulates to the negative of the average slowdown. The goal of an agent is, thus, to maximize the total reward it receives for its entire path, at each state it visits. In those terms, one can also define a broader sense of how good a state is through a value function $V(s_t)$. This function expresses the expected accumulated reward that an agent can receive from a given state, considering not only the current state, but also the future ones that can be reached from it as well. Policy is a function $\pi : \pi(s, a) \rightarrow [0, 1]$ that determines the probability of an agent to pick an action a at a given state s. Intuitively, the policy is essentially the mathematical model that determines and guides the decision-making of an agent. From this perspective, the objective of the agent is to achieve a policy that leads to a bountiful amount of rewards over its run. Finally, the model gives a synthetic representation of the environment. It is usually considered an optional component for most practical applications.

Value is usually defined in terms of a policy. The value function of a state s under a policy π, $V_\pi(s)$, denotes the expected reward the agent can obtain starting from a state s and following π thereafter. A policy is said to be optimal when its value is the maximum possible, *i.e.*, for all other π', we have $V_\pi(s) \geq V_{\pi'}(s), \forall s$. It's also useful to establish how good an action can be at a given state, in the context of a policy. The value of an action, $Q_\pi(s, a)$, defines, then, an expected accumulated reward that selecting an action a in state s can provide in all the subsequent, *future*, time steps, assuming that a policy π will be followed henceforth. The values of policies and actions are estimated during the training of an RL agent, being improved, ideally, at each update, while the agent explores the environment applying its policy. However, oftentimes, the policy that is evaluated and improved may not be the same that is actually being used by the agent in its explorations. Methods in which the behavior policy is different from the one used for estimates are called *off-policy* while the opposite are named *on-policy*.

Considering how, usually, an agent knows next to nothing about its environment and optimizes its estimates based on the policies that it's learning *on-the-go*, a crucial aspect of the learning process of any RL agent is the balance between exploitation and exploration, *i.e.*, how much it should linger to its policy, or take risks and explore new actions. While cautiously following the same policy provides well-known outcomes to the agent, it will not be able to find better solutions without some level of exploration.

In contrast, an agent whose actions tend to be too random may unknowingly find good solutions in its superficial endeavors, but will not be able to actually follow through with them, thus, failing to converge towards a solution. Moreover, from a stochastic reference, the recurrent nature of Reinforcement Learning estimates, *i.e.*, policies being learned from their own sampling, incurs on convergence problems, amplifies noise and increases variance. Several techniques exist to counterbalance those effects, namely Proximal Policy Optimization [41], Advantage Actor-Critic [31] and DQN [33].

2.3.2 Q-Learning

Temporal Difference (TD) is an important class of techniques employed to train Reinforcement Learning agents, named like this due to their incremental approach to exploratory learning. Instead of updating their estimations for value functions at the end of each run, after the agent has collected all the experience it needs to apply an update, TD uses the immediate reward and the *estimations* for the function values thereafter in order to compose its updates. This feature exempts the algorithm from the necessity of an environment model. A simple TD update for a state-value estimate at a time step t is exemplified in Eq. (2.1).

$$V'(s_t) \leftarrow V(s_t) + \alpha \overbrace{[r_t + \gamma V(s_{t+1}) - V(s_t))]}^{\text{TD residual}}, \tag{2.1}$$

in which α is a numeric increment factor for the update, $r_t = R(s_t, a_t)$ and γ is a discount to be applied over future states. A similar formulation can be used to obtain an update rule for the action-value function, *i.e.*, $Q(s_t, a_t)$, named $Q-$learning, as shown in Eq. (2.2).

$$Q'(s_t, a_t) \leftarrow Q(s_t, a_t) - \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{2.2}$$

The maximum recursive term derives from the Bellman Equation and ensures the convergence of the estimator to the optimal action-value function by selecting the action that provides the highest expected action-value (Q). These formulations are defined in terms of a framework of tabular, incremental updates, which are hardly practical for complex problems that demand a good level of generalization. Instead, an approximator function is usually applied. These estimate functions can be represented by neural networks named Q-networks, with parameters updated by executing gradient descent over a loss objective function.

2.3.2.1 Deep Q-Networks (DQN)

Neural networks are commonly applied to supervised learning situations, in which the network approximates a function by learning from a given set of labeled data, in order to learn how to map a certain kind of inputs to a certain expected output. This intuition is not applicable to reinforcement learning, since there is no previously known information about the environment that can guide the optimization process the same way. Instead, Q-networks approximation is defined in terms of **estimations** of the action-value function, sampled from a batch of experiences. Since these estimations change at each iteration in the environment, the loss function L_i is unique to each step, as expressed in Eq. (2.3).

$$L_i(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(r_t + \gamma \max_a Q(s_{t+1}, a; \theta_i') - Q(s_t, a_t; \theta_i))^2], \tag{2.3}$$

in which D represents a set of experience samples, from which a tuple of state, action, reward and next state is uniformly sampled; θ_i are the parameters of the Q-network being optimized at the i-th iteration; and θ_i' are the parameters of the network, *i.e.*, the policy, that is actually being used for behavior of the agent in the environment – consider that DQN is an *off-policy* method [33]. Both D and θ_i' represent two of the core ideas behind DQN that allow it to converge to a satisfactorily generalizing policy: respectively replay buffer and target networks. An overview of DQN is expressed in Algorithm 1, considering T time steps and that a simple ϵ-greedy policy (a $1 - \epsilon$ chance of acting randomly) is being used to guide the agent through the environment.

Algorithm 1 Basic implementation of DQN.

for iteration $\leftarrow 1, 2, \cdots$ **do**
  Initiate empty experience replay buffer $D \leftarrow \{\}$
  for timestep $\leftarrow 1, 2, \cdots, T$ **do**
    if $s \sim U(0, 1) < \epsilon$ **then**
      sample action a_t randomly from action space
    else
      $a_t \leftarrow \arg\max_{a_t} \hat{Q}(s_t, a_t)$
    end if
    Act a_t on the environment and collect s_{t+1}, r_t.
    $D \leftarrow D \cup \{(s_t, a_t, r_t, s_{t+1})\}$
  end for
  Sample a set I of random input tuples $I \sim U(D)$, containing state-action pairs and the rewards obtained from them.
  Optimize $L_i(\theta_i)$, using gradient descent over the samples from I.
  Update Q-network parameters towards the direction that minimizes the loss, *i.e.*, fitting the Q-network to the data from input samples.
end for

The replay buffer is essentially a set of experiences that is kept for sampling during the training of a DQN agent. Batches of samples are taken from a uniform distribution in order to mitigate the effects of temporal correlation between states. For instance, in the original problem that sparked the development of DQN, classic Atari games, each state is represented by a screen capture in such a way that samples taken shortly after one another do not exhibit significant visual difference. Such nature of the input could limit the diversity of game configurations in the batch and cause sample inefficiency, if a random uniform sampling strategy was not adopted.

Moreover, replay buffer allows the network to be exposed to a same experience multiple times at different moments during training. By doing so, one can ensure that the agent will have more chances to learn from occurrences that may not have been common within its trajectory. The size of the replay buffer can be a concern, though, in terms of memory efficiency, so a hyperparameter usually determines it.

DQN is an *off-policy* method, which means that the agent's behavior is not determined strictly by the greedy choice over the Q-network. Even though it seems counter-intuitive, this strategy has an effect over the bias/variance trade-off and regularization. For a simple example, allowing the agent to perform a random action once in a while can encourage it to try unexpected paths that wouldn't be explored if the policy was being followed strictly. On the other hand, keeping a purposely outdated Q-network that is only synced to the main one after a few steps, creates a parameter update delay that prevents the greedy policy from collapsing and diverging. This last strategy can be controlled by an interpolation factor that creates a smooth transition between parameter updates over the steps of a Polyak parameter averaging [50].

2.3.3 Advantage Estimation

Oftentimes, during the trajectory of an agent, it can be useful to infer how good an action is at a given state relative to the other possible actions. Consider, for example, that an agent's policy π has led it to a state s with value $V^\pi(s)$ and is confronted with the option to follow either a_π, sampled from its policy, or any other arbitrary a. Suppose, however, that $Q^\pi(s,a) > V^\pi(s)$, *i.e.*, taking a, then following the policy *thereafter* leads to states with higher expected return than simply following the action that the policy determines right at state s. Hence, one can expect choosing a to be a more advantageous option at state s than the expected reward to be obtained from the policy. This intuition is expressed by the advantage function $A^\pi(s,a)$ in Eq. (2.4).

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{2.4}$$

The advantage of a_π, $A^\pi(s, a_\pi)$, in the given example, would be zero, since $Q^\pi(s, a_\pi) = V^\pi(s)$. On the other hand, the advantage of the arbitrary action a, $A^\pi(s, a)$, would be a positive value, given that $Q^\pi(s, a) > V^\pi(s)$. If, however, the action does not increase the expected return, *i.e.*, $Q^\pi(s, a) < V^\pi(s)$, its advantage would be negative. The advantage function has an important baseline role in classes of RL techniques that rely on gradient descent directly over rewards to optimize policy neural networks, reducing variance and improving convergence. In this context, it encodes the intuitive notion that, if an agent's choice cannot be expected to lead to an advantageous path, then it should avoid updating its parameters significantly towards the gradient directions that are taking it there.

Much like other RL functions, the advantage can be either estimated in a recursively stochastic approach, *i.e.*, computing an estimate from other estimates, at the risk of enforcing bias; or by sampling from actual experiences, which, in the context of typical RL problems, can lead to high variance. In other words, the bias/variance trade-off manifests itself. It is useful, then, to define a parameterized approach to advantage estimation that enables a certain level of control over the trade-off. That is the purpose of the Generalized Advantage Estimation technique [40]. Let $\delta_t^{\hat{V}} = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$ be defined as the TD residual component of Eq. (2.1) for an estimated value function $\hat{V}(s)$. Let the sum of k of these terms be denoted as $A_t^{(k)}$ and defined as in Eq. (2.5).

$$
\begin{aligned}
\hat{A}_t^{(1)} &= \delta_t^{\hat{V}} = -\hat{V}(s_t) + r_t + \gamma\hat{V}(s_{t+1}) \\
\hat{A}_t^{(2)} &= \delta_t^{\hat{V}} + \gamma\delta_{t+1}^{\hat{V}} = -\hat{V}(s_t) + r_t + \gamma r_{t+1} + \gamma^2\hat{V}(s_{t+2}) \\
\hat{A}_t^{(3)} &= \delta_t^{\hat{V}} + \gamma\delta_{t+1}^{\hat{V}} + \gamma^2\delta_{t+1}^{\hat{V}} = -\hat{V}(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3\hat{V}(s_{t+3}) \\
\hat{A}_t^{(k)} &= \sum_{i=0}^{i=k} \gamma^i\delta_{t+i} = -\hat{V}(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k\hat{V}(s_{t+k}).
\end{aligned} \tag{2.5}
$$

The terms r_{t+i} encode empirical returns obtained from experience over the agent's exploration in the environment, while $\hat{V}(s_t)$ and $\hat{V}(s_{t+k})$ expresses the estimated value function for the first and last state within the range of k. As k grows, so does the discount factors applied, in such a way that, with $k \to \infty \implies \gamma^k\hat{V}(s_{t+k}) \to 0$, turning $A_t^{(k)} = \sum_{i=0}^{\infty} \gamma^i r_{t+i} - \hat{V}(s_t)$.

From Eq. (2.5), GAE-λ can be defined as an exponential weighted average of all

the $A_t^i, i = 1, \cdots, k$ estimators parameterized by a factor λ, as determined in Eq. (2.6).

$$\begin{aligned}
\text{GAE-}\lambda &= (1-\lambda)(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)}\cdots) \\
&= (1-\lambda)(\delta_t^{\hat{V}} + \lambda(\delta_t^{\hat{V}} + \gamma\delta_{t+1}^{\hat{V}}) + \lambda^2(\delta_t^{\hat{V}} + \gamma\delta_{t+1}^{\hat{V}} + \gamma^2\delta_{t+2}^{\hat{V}})\cdots) \\
&= (1-\lambda)(\delta_t^{\hat{V}}(1 + \lambda\cdots) + \gamma\delta_{t+1}^{\hat{V}}(\lambda + \lambda^2\cdots) + \gamma^2\delta_{t+2}^{\hat{V}}(\lambda^2 + \lambda^3\cdots)\cdots) \\
&= (1-\lambda)\left(\delta_t^{\hat{V}}\frac{1}{(1-\lambda)} + \gamma\delta_{t+1}^{\hat{V}}\frac{\lambda}{(1-\lambda)} + \gamma^2\delta_{t+2}^{\hat{V}}\frac{\lambda^2}{(1-\lambda)}\cdots\right) \\
\text{GAE-}\lambda &= \sum_{i=0}^{\infty}(\gamma\lambda)^i\delta_{t+i}^{\hat{V}}.
\end{aligned} \tag{2.6}$$

Setting $\lambda = 0$ in this formulation leads to GAE-0 $= \delta_t^{\hat{V}} = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$, which is exactly the TD residual of the value estimator, a biased estimator of the advantage. If, however $\lambda = 1$, we have GAE-1 $= \sum_{i=0}^{\infty}\gamma^i r_{t+i} - \hat{V}(s_t)$, which derives from the empirical returns themselves, leading to low bias, but high variance. By controlling the value of λ, it is then possible to balance the bias/variance trade-off for the advantage estimator.

2.3.4 Policy Gradient Methods

Arguably, learning an approximator for the action-value function from a batch of previous experiences is not the only way that deep neural networks can be applied to reinforcement learning. The policy itself is, too, a function and, as such, can also be represented by a neural network, as stated by the universal approximation theorem [17]. Policy gradient methods work by deriving an objective function $J(\pi_\theta)$ from the expected finite-horizon undiscounted return of the policy itself and using it to optimize the parameters of the neural network via gradient ascent [48], as defined by Eq. (2.7).

$$\theta_{k+1} = \theta_k + \alpha\nabla J_\theta(\pi_\theta), \tag{2.7}$$

which represents a simplified update rule of the policy parameters θ, controlled by a step size α. The input of the network is a representation of state and its output is a probability distribution over the actions, from which the chosen one will be sampled. Intuitively, this technique guides the parameters of the neural network towards the direction that maximizes the probability of choosing actions with good returns. A simple version of the gradient term can be derived in terms of $R(T)$, the return function of a trajectory T, as follows from Eq. (2.8).

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \, \mathbb{E}_{T\sim\pi_\theta}[R(T)] \\
&= \nabla_\theta \int_T P(T|\theta) R(T) \\
&= \int_T \nabla_\theta P(T|\theta) R(T) \\
&= \int_T P(T|\theta) \log \nabla_\theta P(T|\theta) R(T) \\
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{T\sim\pi_\theta}[\nabla_\theta \log P(T|\theta) R(T)] \\
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{T\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(T)\right], \qquad (2.8)
\end{aligned}
$$

which makes use of the fact that $\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)$ (chain rule applied for the logarithm derivate) and $\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$ (the gradient log of the probability of a trajectory can be inferred from the sum of gradient logs of the policy choices over all T time steps). Implementations of policy gradient methods usually employ the advantage function $A^{\pi_\theta}(s,a)$ instead of the trajectory return $R(\tau)$ in Eq. (2.8). The intuition behind that derives from the use of baselines for variance control, expressing the maximization of the probability of actions with better advantage over their peers at a every given state. The use of GAE for advantage parameterizes the bias/variance trade-off, as exposed in Section 2.3.3.

In practice, the basal definition of policy gradients can be hard to optimize, demanding regularization and scaling of the update steps in order to achieve convergence. This problem motivated the development of gradient clipping and training regularization techniques, namely Advantage Actor-Critic and Proximal Policy Optimization.

2.3.4.1 Advantage Actor-Critic

As an alternative to replay buffer, parallelized approaches to deep RL leverage multithreaded hardware for training [31]. The main idea behind this concept is to collect experiences from multiple agents at the same time, mitigating the need for a memory of experiences and making overall training faster. This technique can be applied to several contexts, from Q-Learning to policy gradients. Regarding policy gradients, it's commonly employed for actor-critic architectures, in which, in order to reduce the variance and stabilize gradient updates, a baseline function is subtracted from the global returns.

A common choice for baseline is the value function, which estimates how good a given state is. The value acts, then, as a *critic* to evaluate whether the action picked by

the policy, *i.e.*, the *actor* has led to a good state. From the principle of two concomitant components acting together to define the agent's behavior comes the name actor-critic. Starting from Eq. (2.8) and considering the partial trajectories $\tau_{:t}$ (trajectory up to point t) and $\tau_{t:}$ (trajectory after point t), one can rewrite the equations in terms of the action-value $Q(s_t, a_t)$, following the demonstration in Eq. (2.9).

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] = \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)] \\
&= \sum_{t=0}^{T}\mathbb{E}_{\tau_{:t}\sim\pi_\theta}[\mathbb{E}_{\tau_{t:}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)|\tau_{:t}]] \\
&= \sum_{t=0}^{T}\mathbb{E}_{\tau_{:t}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)\,\mathbb{E}_{\tau_{t:}\sim\pi_\theta}[R(\tau)|\tau_{:t}]] \\
&= \sum_{t=0}^{T}\mathbb{E}_{\tau_{:t}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)\,\mathbb{E}_{\tau_{t:}\sim\pi_\theta}[R(\tau)|s_t, a_t]] \\
&= \sum_{t=0}^{T}\mathbb{E}_{\tau_{:t}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)Q(s_t, a_t)]
\end{aligned} \tag{2.9}$$

Please note that, since $\nabla_\theta \log \pi_\theta(a_t|s_t)$ depends only on the action and state taken at point t, their expectation with regard to the partial trajectory $\tau_{t:}$ is a constant. Also note that the term $\mathbb{E}_{\tau_{t:}\sim\pi_\theta}[R(\tau)|\tau_{:t}]$ considers the expected reward of the future trajectory conditioned to the past from time step t behind, hence, it only really depends on the state and actions taken at that step.

By subtracting the estimated value $V(s)$ as a baseline from $Q(s, a)$ in Eq. (2.9), we obtain, then, the equation of the advantage estimation $Q(s_t, a_t) - V(s_t) = A(s_t, a_t)$. In order to introduce a new element of randomness to encourage exploration, one can also add the entropy of the policy gradients to the formulation $\nabla_\theta H(\pi_\theta)$, controlled by a hyperparameter β. The final update of the policy gradient described so far follows the statement in Eq. (2.10).

$$\nabla_\theta J(\pi_\theta) = \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)(A(s_t, a_t) + \beta\nabla_\theta H(\pi_\theta))] \tag{2.10}$$

Since the advantage can be obtained from a value estimator and collected rewards only, following the procedure described in Section 2.3.3, two networks are necessary to implement this method of policy gradient: one for the actor that approximates the policy π_θ and another for the value $V(s)$. Architecturally, both networks can share layers, which also improves on training efficiency and stability towards convergence. The value network's parameters θ_v can be optimized by minimizing a squared-differences loss $L_v(\theta_v, \pi_\theta) = \mathbb{E}_{s_t, R_t\sim\pi_\theta}[(V_{\theta_v}(s_t) - R_t)^2]$.

The Advantage Actor-Critic architecture explored in this work is A2C, the *synchronous* variation of Asynchronous Advantage Actor-Critic (A3C), which leverages the

particularities of the algorithm's design to train on multiple threads. A2C has been shown to provide similar performance and better efficiency for training on machines of limited capacity for parallelization [55]. The simplified overall procedure for training an A2C agent is described in Algorithm 2. When updating network parameters from the accumulated gradients, deep learning standard techniques can be employed, such as stochastic gradient descent or Adam [4, 21].

Algorithm 2 Basic implementation of A2C.

for iteration $\leftarrow 1, 2, \cdots$ **do**
  for agent $\leftarrow 1, 2, \cdots, N$ **do**
    Run $\pi_{\theta_{\text{old}}}$ for T time steps, collect (r_t, a_t, s_t, s_{t+1})
    Compute advantages $A^{\pi_{\text{old}}}$ from the value approximator $V_{\theta_v}(s_t)$
    Accumulate gradients for θ using $\nabla_\theta J(\pi_\theta)$
    Accumulate gradients for θ_v using $\nabla_{\theta_v} L_v(\theta_v, \pi_\theta)$
  end for
  Update actor parameters $\theta_{\text{old}} \leftarrow \theta$ using accumulated gradients
  Update critic parameters $\theta_{v_{\text{old}}} \leftarrow \theta_v$ using accumulated gradients
end for

2.3.4.2 Proximal Policy Optimization

Proximal Policy Optimization is an *on-policy* gradient technique that improves on past work towards more stable updates on policy optimization [39][41]. Essentially, PPO uses a clipped, or constrained, objective function whose gradients ultimately result in policy updates that happen within a trust region. The span of this region is controlled by a hyperparameter itself, but the update rules are designed in order to ensure that the parameters of the policy network will be moved as much as possible towards the optimization direction without collapsing. The constraints and clipping applied to the objective function in PPO are first-order operations, considerably simpler to differentiate than the previous divergence methods that PPO derived from.

The objective function that PPO seeks to maximize in relation to its current policy parameters θ is defined as shown in Eq. (2.11). The division of $\pi_\theta(a|s)$ over $\pi_{\theta_k}(a|s)$ expresses the relative likelihood of the actions between policies, *i.e.*, if an action is more likely to be selected in π_θ, then the ratio will be > 1, an augmentative multiplication factor; otherwise, the ratio will be ≤ 1, a reducing factor (considering the absolute value

of the terms).

$$L(s,a,\theta,\theta_k) = \begin{cases} \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s,a), & \text{if } A^{\pi_{\theta_k}}(s,a) > 0 \\ \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s,a), & \text{if } A^{\pi_{\theta_k}}(s,a) \leq 0. \end{cases} . \tag{2.11}$$

If the advantage is positive, in order to maximize L, the likelihood of the actions expressed by $\pi_\theta(a|s)$ must increase. However, a min term imposes a limit on how much far the new policy is allowed to go: once it becomes too high, the expression will be limited again to simply $L(s,a,\theta,\theta_k) = (1+\epsilon)A^{\pi_{\theta_k}}(s,a)$, regardless of the ratio. Similarly, if the advantage is negative, then to maximize L, the likelihood given by π_θ must decrease, but not too much, since, once again, the expression will be capped at $L(s,a,\theta,\theta_k) = (1-\epsilon)A^{\pi_{\theta_k}}(s,a)$. Hence, the caps act as a regularization factor controlled by the hyperparameter ϵ, not allowing the new policy to strive too far from whence it came from at each update step.

A value approximator is necessary in order to compute the generalized advantage estimate as described in Section 2.3.3. Some implementations apply capping to the value function too, but further studies have shown no evidence of improvement and even a chance of decrease in performance [10, 2]. A basic implementation of PPO is described in Algorithm 3, considering N agents and T time steps. It's similar to the A2C procedure described in Algorithm 2, except for the clipped loss function. The policy parameters θ that maximize L, the objective function, can be optimized by standard gradient-based techniques, such as Stochastic Gradient Descent or Adam [4, 21].

Algorithm 3 Basic implementation of PPO.

for iteration $\leftarrow 1, 2, \cdots$ **do**
  for agent $\leftarrow 1, 2, \cdots, N$ **do**
    Run π_{old} for T time steps
    Compute advantages $A^{\pi_{\text{old}}}$ for each time step using an approximator for $\hat{V}_{\pi_{\text{old}}}$.
    Accumulate advantages in minibatches.
  end for
  Use gradient descent to optimize $L(s,a,\theta,\theta_k)$ with relation to θ.
  Update policy parameters $\theta_{\text{old}} \leftarrow \theta$
end for

Chapter 3

Related work

Recent advances in techniques to train deep neural networks have been responsible for notable breakthroughs in machine learning [22, 38, 15]. Their nonlinear nature enables the extraction of abstract representations useful for solving complex classification and regression problems [24]. They have been successfully adapted to reinforcement learning contexts as well, with clever solutions to overcome challenges such as high correlation of input data entries, delayed reward, as well as changes on the distribution of input data [32]. Games make a particularly useful field for deep RL solutions, as a result of the usually finite and discrete action space, controlled environments as well as the clearly defined rewards and goals [42, 44, 45]. Applications of deep RL to solve problems from several other fields can also be found on the literature, spanning from robotics, industrial logistics for manufacturing and energy harvesting [3, 16, 18, 37], to high level optimization problems [25, 57].

A reinforcement learning framework can, thus, encompass a broad variety of techniques. Currently, implementations for the most widely applied algorithms can be found in the form of open-source tools, either for general deep learning, with Tensorflow [1] and Pytorch [34] as notably recognized examples of libraries for model engineering; or particularly focused on RL, such as OpenAI Gym [5]. The latter supports great adaptability to more specific use cases, including extensions for the task of resource management that assist with environment design concerns, in addition to software testing [9]. These tools provide solid foundations for the development of machine learning and deep RL solutions, given the challenges inherent to them, such as the sensitiveness to variance and difficult reproducibility mentioned in Chapter 1. By making the iteration over different model architectures and optimization techniques easier, researchers and engineers can progress faster and more reliably, conforming to both academic and industrial requirements.

Popular classes of RL approaches include value-based methods like Q-learning, in which an agent learns to evaluate its own actions, *i.e.*, estimate its Q function, in order to determine the best ones to compose its policy [54]; and policy gradient methods [49], where agents learn to approximate optimal policies by increasing the probability of taking higher-rewarding actions and decreasing the chance for actions of lower return. With gradient descent at their cores, different deep learning approaches have been developed

with regard to how these types of training can be conducted and how their policies should change throughout training. Trust Region Policy Optimization [39] and Proximal Policy Optimization [41], for example, try to update the policies as much as possible at each step, while observing similarity constraints between the old and new stochastic distributions they give over the actions. Other approaches combine both techniques, using an estimated Q function to learn a deterministic policy [46]. These algorithms propose general solutions to reinforcement learning problems, hence, their main concepts can be adapted to more focused applications as well.

3.1 Resource management approaches

Regarding resource management, Mao et al. [27] devised DeepRM, a simple cluster scheduler based on deep reinforcement learning, aimed to be a stepping stone for the development of RL-based resource managers. In DeepRM, state is modeled as a set of 2D grids, one for each resource and job. One axis represents time, while the other a certain amount of resource to be allocated. Such representation allows jobs to be spatially distributed on the cluster grid, much like a game of Tetris. Action space includes all the jobs slots to be scheduled and rewards are defined according to the goal at which the schedule is aimed to, for example, average slowdown. In the original work, state representation is passed through a simple policy neural network that outputs a probability distribution over all possible actions. During training, the policy network is optimized to produce higher probabilities for the actions that would return better cumulative rewards, should the policy be applied. Over time, Deep RM was able to produce policies that outperformed principled heuristics, like *Shortest Jobs First*. Solutions to improve development and testability of RL environments dedicated to schedule management have been explored in sched-rl-gym [9] with the ad of OpenAI's library Gym [6], an open project broadly adopted by the reinforcement learning research and development community.

Further development has been done on more complex scenarios of cluster scheduling. In [29], Decima, a workload-specific cluster scheduler is proposed for large scale systems with intricate dependency schemes between job states, represented by directed acyclic graphs (DAGs). This type of input may vary in size, which makes it difficult to handle with traditional feed-forward deep networks. To overcome that, Decima employs a graph neural network as a representational encoding step prior to the policy network. In order to enhance the scheduler's ability to generalize over real-world scenarios with stochastic job arrival patterns, variance mitigation techniques for input-driven environments [30] were applied. Decima's results report showed promising performance gains

against scheduling heuristics engineered manually.

Approaches that differ from policy gradients can also be found on the literature, using deep neural networks to represent different entities of a reinforcement learning framework. Instead of learning the policy as a function of the input, one can build, for instance, an approximator neural network for the action-value function $Q(s, a)$, which maps each action to a real-valued accumulated reward, an approach known as Deep Q-learning [23]. From there, a policy can be constructed by greedily choosing the actions that maximize the approximated Q function. This method has been adopted by [56] in their job allocation solution for data centers. The described technique takes temperature, utilization and power consumption metrics from servers into consideration for the reward function, demonstrating how notoriously flexible the goal-oriented character of reinforcement learning can be. Using computational models of memory networks to represent state, they propose an offline training approach, in which RL agents can be trained much faster under controlled conditions. Results show successful optimization towards the desired temperature and energy consumption goals, when compared to traditional scheduling policies like round-robin and short-horizon online optimization.

Flexible representations of job structures can be useful to design layered architectures for workload managers as an alternative to deep learning models. Hugo, described in [52], is a cluster scheduler that aims to maximize the usage of resource groups in distributed data processing systems by taking advantage of complimentary attributes in the workload domains. For example, CPU and I/O intense jobs can be scheduled to run in parallel, in order to guarantee optimal usage of all the available resource types. Additionally, Hugo works in a clustered fashion, grouping jobs into similar categories, according to previously collected profiling data. For each pair of groups, the system computes a goodness score that is used as a criterion to select the next subset of jobs to be scheduled. The selection process and online calculation of the scores is done through reinforcement learning techniques.

Chapter 4

Methodology

Developing applications based on Deep Reinforcement Learning can be a notoriously convoluted task, due to the complex interplay between parameters and hyperparameters, model architecture and many other implementation variables that make training unstable. To mitigate that issue, many open-source solutions have emerged, with broad collaboration from communities all over the world that report issues and fix failures. This project makes extensive use of these code libraries in order to ensure results are reproducible. The developed pipeline includes a parallelized and parameterized training procedure that trains multiple agents with different parameter set-ups at the same time. Agent policies are trained defined by the auxiliary libraries, using built-in tool sets for deep learning.

Initially, a diagnostics procedure was conducted by brute-force testing numerous possible combinations of the policies training parameters for the same number of training iterations of a regular training. From then, we discarded agents that were not able to converge during training, with too erratic and highly unpredictable performance. Once complete, the diagnostics allowed us to fine-tune the parameters with potential to provide interesting results. These parameters were fed to the training pipeline that generated a new process thread for each configuration, being trained isolatedly and independently of one another.

During training, we collected progress data at every few iterations. Less frequently, but also as training progresses, we've performed evaluation procedures within isolated environments, with their neural network left in a frozen state, to prevent gradient descent from considering evaluation as a learning experience. The data included the curve of the reward signal during training, average slowdown progression, as well as action choice and job size distributions. Some algorithms were retrained in different environment conditions.

4.1 Experiment Design and Diagnostics

All the environments were implemented with aid of OpenAI's Gym [6], a framework that aims to standardize the design and testing of reinforcement learning environments with built-in control APIs and reference environments. Within OpenAI's Gym library of RL environment, sched-rl-gym was chosen for its focus on workload management with built-in job generation for training samples and task scheduling capabilities. For the algorithms implementation we've adopted Stable Baselines 3 [35], a set of implementations of well-known RL techniques from the literature with strong community collaboration towards stability and testability. Under the hood, Stable Baselines 3 employs PyTorch [34] as the main deep learning library for its learning-based policies. PyTorch contains a robust tool set for neural network design and training, with solid implementations of state-of-the-art engineering techniques. Although Stable Baselines 3 contains a broad set of RL algorithms ready for usage, we've attained to the 3 techniques focused on this work: Proximal Policy Optimization (PPO), Deep Q Network (DQN) and Advantage Actor Critic (A2C).

Consider $\Pi_a, a \in \{\text{PPO}, \text{DQN}, \text{A2C}\}$ the set of all policy parameters, E, the set of parameters for the environment and N all the different applicable network designs. Initially, all policies were trained with the default values for all their parameters. To conduct a parameter search study with every possible combination of $(\Pi_a \times E \times N)$ can become quickly untractable, due to the amount of possible parameters to be passed to each policy algorithm, plus all the diverse configuration options for the scheduler environment and different combinations of network architecture designs. Therefore, the diagnostic procedure was executed with a subset of each parameter domain, selected accordingly to the documented expected behavior for variations in their respective values. For the same reason, parameters regarding cluster hardware, such as amount of available unit processors and memory constraints were all kept constant throughout the experiments, in order to provide consistent baseline set-ups for comparisons between *algorithm features only*. Such procedure allowed us to test for conditions that not only favorably improved the performance during training and evaluation, but also the ones that caused stability issues or even prevented agents from progressing satisfactorily.

There can be two main categories of environment-related parameters: structural ones, such as backlog size and time horizon; and workload-related ones, such as the rate at which jobs are submitted to the clusters and job size. Similarly to cluster configurations, structural parameters were kept in the default values specified by the sched-rl-gym environment. The intensity at which jobs are submitted is more reflective of real-world bursts of activity and is, thus, our main concern. Therefore, our study was focused on workload-related aspects, namely:

- **Submission frequency**: the probability of generating a new job at every time tick of the workload generator – which does not necessarily synchronizes with environment timesteps as perceived by the agent.
- **Chance of small jobs**: given that a new job is set to be generated, the probability of its parameters being sampled from the small job size distribution.

Regarding network architecture, numerous designs can be adopted, from simple multi-layer networks to sophisticated models that employ advanced techniques for regularization, memory, and convolution. For this study, we adopted the simpler approach: a single multi-layer perceptron (MLP), with two hidden layers and a softmax output, with a flattening being applied to the input state representation matrices. All the models were implemented using Stable Baselines 3's own APIs, which underlyingly apply PyTorch. The elementary design also allowed us to test specifically for hidden layer size in the diagnostics stage.

Before starting every diagnostic experiment, a tree with layers was constructed for each parameter of the tuple $(\Pi_a \times E \times N)$. The internal nodes of the tree represent, thus, a parameter value, and its leafs contained the complete sets of specifications for each training session. Once complete, every experiment log was manually analyzed for training stability and average slowdown optimization. Runs with strong divergence that showed no signs of minimum learning progression were not considered in the further evaluation steps. In this step, no automated parameter search solutions were used.

The policy parameters are diversified, and the motivation behind them is expressed in the following sections. All are expressed within the context of the implementation and notation of Stable Baselines 3, but a relationship with the technical variables described in Section 2.3 is established wherever it exists.

4.1.1 DQN hyperparameters

Deep Q-Network is a technique that employs neural networks within the framework of Q-learning. In this context, policy parameters are not optimized from gradient descent over the reward function, like on policy gradient methods. Instead, a state-value function is approximated based on sets of experiences collected during training. DQN also implements several stabilization techniques to prevent its models from diverging, prominently experience replay buffer and target networks. The former consists of a history of the agent's experiences from which the training input is sampled randomly. The latter consists of an auxiliary network that is used to compute the target values used within loss

calculation and is purposefully kept behind the main Q-function being learned, in terms of parameter updates, being synced slowly, at every few steps. The speed at which the target network is updated is controlled by the parameter τ established as an interpolation coefficient that gradually weights the network parameters from adjacent time steps. The chosen hyperparameters for variation in DQN were then, derived from these two features:

- τ: update coefficient that controls the interpolation of network parameters, smoothing the synchronization of the target network with the main Q-network.
- **Replay buffer size**: the number of past experiences kept available for input sampling during training.

4.1.2 A2C hyperparameters

Advantage Actor Critic proposes a framework for training that avoids the use of a replay buffer by setting up multiple agents to explore their own instances of the environment and collect experiences assuming their trajectories will be diverse enough. It then uses a policy gradient approach with two networks, the actor, which outputs an action, and the critic, which computes the state's value considering the output from the actor. The gradients that will optimize the actor network are computed from the values estimated by the critic. No gradient clipping or trust-region delimitation is applied, but an entropy coefficient is implemented in order to encourage the policies to keep exploring and avoiding pitfalls in local minima of the optimization space. Therefore, the chosen hyperparameters for A2C were the ones related to the exploration/exploitation balance inherent to its elements, the actor, and the critic.

- **Entropy coefficient for policy**: controls exploration encouragement in the loss function of the actor, which determines the policy.
- **Entropy coefficient for value function**: similarly to the entropy coefficient for policy, but applied to the critic, which determines the value of the actions that the actor outputs.
- **GAE-λ**: likewise PPO, GAE-λ in A2C controls the bias/variance trade-off for the advantage computation that is applied to the actor's objective function.

4.1.3 PPO hyperparameters

Proximal Policy Optimization is a policy gradient method and as such, works by optimizing an objective function, *i.e.*, the loss, using the gradients of the policy with respect to the expected return. The latter factor is represented by the advantage that an action offers in relation to the others. However, such gradients must be kept within a trust region, in order to prevent destructive policy updates during optimization. PPO works by clipping the loss function within the range defined by a hyperparameter ϵ. Stable baselines also implement an additional clip parameter for the value function, used for advantage calculation. Consequently, the chosen parameters for variation in the studies of PPO were:

- **GAE-λ**: controls the trade-off of bias and variance of the Generalized Advantage Estimator used in PPO's formulation.
- **Policy clip range**: the ϵ clipping parameter used to delimit the trust region of PPO's policy gradient updates.
- **Value clip range**: clipping to be applied over the gradient updates of the value function.

4.2 Training, Evaluation, and Retraining

Once the diagnosis stage was completed – with every possible combination of the evaluated environment, algorithm, and model size – diverging runs were discarded, and the remaining results were clustered according to the density of their workload-related specifications. Each run configuration was then subjected to a second training procedure in which an evaluation step was applied regularly throughout the progression, computing both training monitoring metrics and average-slowdown, *i.e.*, performance metrics. Finally, the best policies trained in environments with easier navigation patterns were subjected to one more training session, at this time, in a denser, harder to grasp scenario. This procedure was applied in order to investigate if a hierarchized learning approach, from an environmental challenge standpoint, could be beneficial for agents with previous familiarity with the problem at hand.

Monitoring metrics included the p-value of the actions distribution picked by the policies, under the null hypothesis that, at every evaluation step, the distribution was the

same from the previous one. Other factors such as the relative size of the selected jobs were also collected, in order to study the behavior of the learning-based policies regarding features used by principled algorithms. Performance metrics, in turn, were evaluated for the target optimization factor, average slowdown, as well as compared to a reference random policy, an SJF approach that always picked the smallest jobs and a FCFS policy that allocated jobs as they arrived.

Chapter 5

Results

In order to study the effects that hyperparameters and different algorithms exert over the progression of training, convergence behavior and optimization of the target average slowdown metric, we performed search experiments combining different setups for each algorithm. Since environmental conditions may also be influential to the performance of an agent, we've also introduced variations on workload characteristics. Finally, for comparison, we've executed rounds of experiments using the default values usually pointed in the literature, or determined by the developers of the most prominent libraries in the field.

Regarding environment conditions, our focus was directed towards workload-related parameters. Hence, the variables associated to the *cluster itself*, such as the number of processors, were kept consistent across all comparative experiments. On the other hand, *simulation parameters* were subjected to variation. Namely, small job chance, *i.e.* the probability of a short job being submitted; and new jobs rate, the frequency with which new jobs are produced, were shown to be the most significant factors over an agent's ability to learn. This allowed us to study how each algorithm adapts to challenging patterns of cluster operation when agents are trained from scratch in comparison to agents previously developed under easier workload circumstances.

At the core of a deep reinforcement learning agent lies the neural network that learns how to map observations to actions. Consequently, the design and architecture of such network can also be influential on how convergence progresses at each episode, ultimately leading to the optimization of the desired metrics. The number of layers and their sizes also impose performance concerns, critical to determine the amount of actual resources a deep reinforcement learning scheduler requires in order to operate. Thus, our experiments concentrated on determining if simpler, smaller, networks could provide stability and ability to converge similar to larger ones.

This chapter is structured as follows: in Section 5.1, a description of the overall procedure adopted for training and evaluation of the obtained agents is explained, as well as the collected data variables. On the next Section 5.2, we list the parameters of the environment in which the algorithms were trained. Section 5.3 contains the results of the baseline experiments, executed with the default parameters for each algorithm, as

specified by the developers and community that adopts them. In Section 5.4 we study the effects of variations across the most prominent hyperparameters of each algorithm. For Section 5.5, we explore the ability of the agents to adapt from a sparse to a dense environment, comparing it to the performance of agents trained on dense environments without prior preparation.

5.1 Training and evaluation procedures

At each simulation environment step from sched-rl-gym, which operates according to the description in Chapter 2, the average reward obtained by the agent throughout the episode was computed and logged into a Tensorboard instance. The algorithms' performances were further evaluated at breaks of 2000 training steps (across episodes). During the breaks, policies were subjected to 50 extra, separated episodes, from which average slowdown was calculated. An episode is an execution session of the simulator that produces jobs until a time limit of 50 simulation steps is reached. Gradient descent was disabled for all models during this procedure, in order to keep parameters frozen. In addition, 50 episodes of a SJF, FCFS and a random agent were computed at each eval step, with the average result plotted for comparison against the results obtained from RL policies. During diagnostics, two-layered networks with 16, 32 and 64 neurons in the hidden layer were tested. Input and output depends on the nature of the network being applied: value-based networks, like DQN or advantage estimators, will receive a state representation as input and compute the corresponding value for each action; while policy-based networks like A2C and PPO will output a probability distribution over the action space, from which an action will be sampled. In the tests, no significant difference was observed from 32 to 64 neurons, but 16 was shown to be insufficient for all possible scenarios, so all the tests were conducted with 32×32 networks.

Additional data was collected at each evaluation cycle to investigate extra nuances and dimensions that may not be explicitly detectable from rewards and policies. Firstly, concerning the progression of the policy behavior as training advances, we kept track of the p-value of the actions choice distribution in comparison to the previous step, under the null hypothesis that the distribution did not not change. By doing so, we were able to check if the policy was actually being updated over time, rather than falling for a local minima and sticking to the same action-picking behavior. Secondly, we've computed the relative size of the job the agent picks compared to the remaining jobs in the queue, in the interest of observing the learned policy behavior regarding apparent job length. Distributions were plotted with the aid of Tensorboard histograms.

5.2 Environment specifications

In order to analyze the workload influence over the agent's performance isolatedly, architectural components of the environment were kept constant throughout all experiments. Number of available job slots was kept in 10, hence, the agent's action space was limited to 11 possible choices: one of the 10 available slots for allocation, or not picking a job at all. Backlog size limit was set to 60, which represents the maximum number of jobs allowed to stay in the waiting queue. Timesteps were capped to a maximum of 100, with a horizon of 20. The simulated high performance clusters were fixed at 38 processing units and memory was not considered for resources simulation, since variations in these factors did not entail substantial and observable effects for the purpose of the study. For the simulations in this study, no user error was introduced in the requested time, memory, and CPU for the generated jobs.

5.3 Default reference experiments

Stable baselines implements each algorithm with a set of predefined hyperparameters, based on general case studies, open source community contributions and literature observations. The values are chosen from a generalist point of view, designed to work as a baseline, before being adjusted to the particular needs of every application. Default values for each of the parameters are expressed on Table 5.1.

	PPO	DQN	A2C
GAE-λ	0.95	not applicable	1.0
Clip range	0.2	not applicable	not applicable
Polyak coefficient	not applicable	1.0	not applicable
Buffer size	not applicable	10^6	not applicable
Entropy coefficient (loss)	10^{-2}	not applicable	0.0
Value coefficient	0.5	not applicable	0.5

Table 5.1: Default parameter values, as implemented by Stable Baselines 3.

5.3.1 Training

Training was conducted throughout 3×10^5 episodes with numerous combinations of workload-related variables. However, for the sake of simplicity and to evaluate the effects that the representation of jobs as states have over the process, these variations were split into two main workload domains: sparse and dense environments. On sparse environments, most jobs tend to be smaller and infrequent, which prevents the queue and execution pipeline from becoming overly crowded and makes the configuration of each moment in time more distinguishable from the others. Dense environments, on the other hand, tend to produce large jobs more frequently, which can create artifacts in the pipeline representation as well as introduce temporal correlation of input samples as the simulation progresses.

5.3.1.1 Sparse job representation distribution

All algorithms tend to produce stable learning reward curves in sparse environments, as shown in Fig. 5.1 and Fig. 5.2. A2C and PPO converge faster, at around $50k$ timesteps, to a plateau of -80 in average reward signal (as defined in Section 2.3.1). Although DQN converges slower and exhibits erratic reward behavior in the first iterations, as soon as its replay buffer benefits start to take effect with a diverse set of experiences, its learning curve quickly increases, reaching a -50 plateau in reward signal, higher than PPO and A2C. Lower submission frequencies make training slightly more stable.

(a) PPO. (b) A2C. (c) DQN.

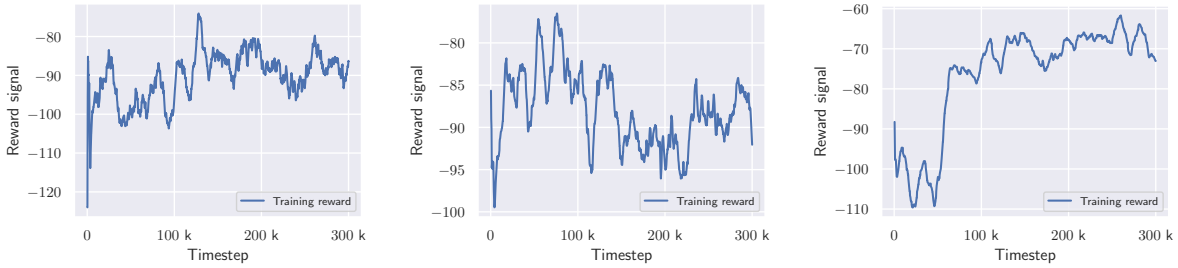
Figure 5.1: Training progress on environment workloads with high chance of small jobs (80%), low submission frequency (0.25).

(a) PPO. (b) A2C. (c) DQN.

Figure 5.2: Training progress on environment workloads with high chance of small jobs (80%), high submission frequency (0.5).

5.3.1.2 Denser job representation distribution

On dense environments each state becomes crowded with jobs, which can make it difficult for neural networks to pick up the patterns for learning and optimizing their loss functions. As shown in Fig. 5.3 and Fig. 5.4, training PPO and A2C in these conditions is unstable, specially when submission frequency is higher. Once again, in this case, DQN starts off unstably, but benefits greatly from the replay buffer after a few time steps. Being exposed multiple times to a same state gives the underlying networks more cycles with each input setup, improving training stability.



(a) PPO (b) A2C (c) DQN

Figure 5.3: Training progress on environment workloads with low chance of small jobs (20%), low submission frequency (0.25).

(a) PPO (b) A2C (c) DQN

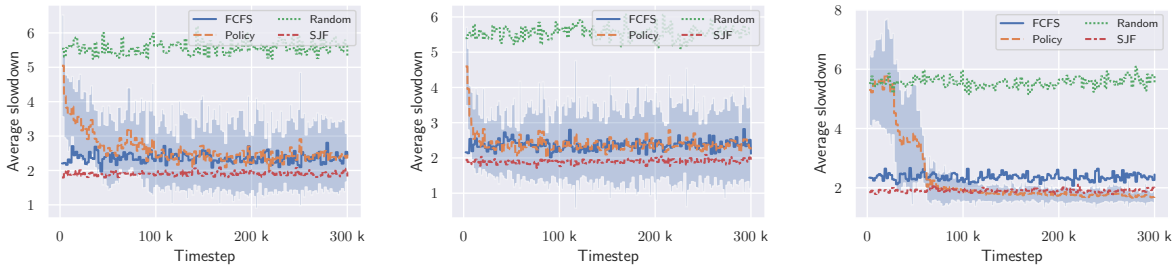
Figure 5.4: Training progress on environment workloads with low chance of small jobs (20%), high submission frequency (0.5).

5.3.2 Evaluation

With default parameters, each algorithm was compared to principled scheduler routines SJF, FCFS and a random baseline. By doing so, we can observe each policy's ability to optimization behavior under the studied environment circumstances. Plots in this section show an average of all the 50 evaluation episodes over the number of timestamps used in the training step. The shaded area indicates standard deviation.

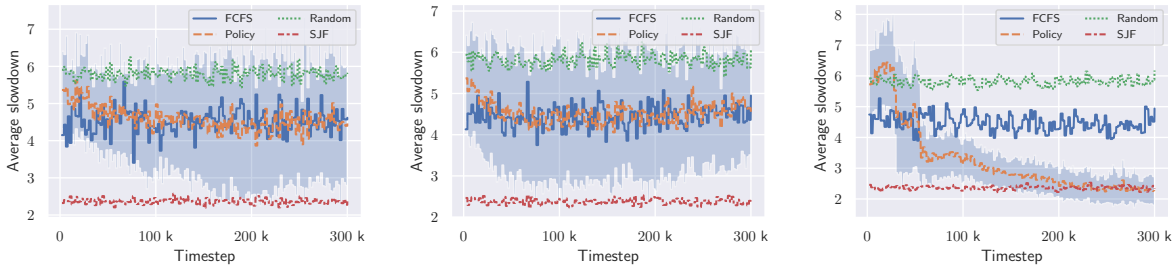
5.3.2.1 Sparse job representation distribution

On sparse environments with low submission frequency, all the learning-based policies managed to reach similar average reward levels of FCFS and, for DQN, even surpass SJF, as plotted in Fig. 5.5. Even though the training curve is stable for both of the tested submission frequency configurations, PPO and A2C exhibited difficulty to achieve good performance, as shown in Fig. 5.6. DQN's performance on the latter took a few extra time steps to reach the same level as the one observed in the low frequency environment.



(a) PPO (b) A2C (c) DQN

Figure 5.5: Average slowdown progression for environment workloads with high chance of small jobs (80%), low submission frequency (0.25).



(a) PPO (b) A2C (c) DQN

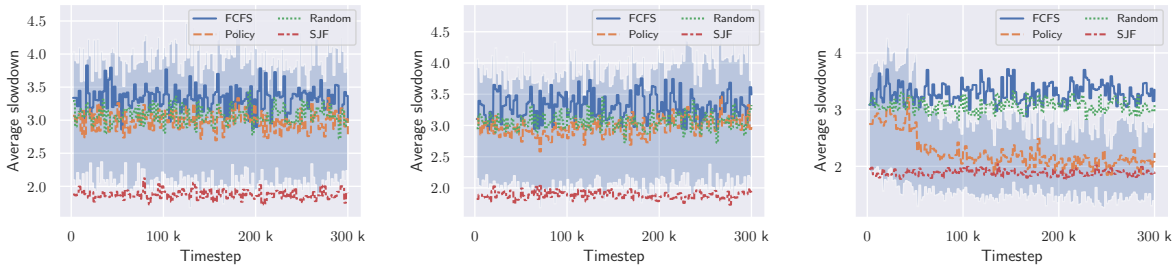
Figure 5.6: Average slowdown progression for environment workloads with high chance of small jobs (80%), high submission frequency (0.5).

5.3.2.2 Dense job representation distribution

As expected from the observed training behavior described in Section 5.3.1.2, with poor convergence results for PPO and A2C under default parameter configurations, dense environments also resulted in subpar performance, as illustrated in the plots from Fig. 5.7 and Fig. 5.8. Both techniques produced policies with performance better than the random baseline and comparable to First-Come-First-Serve, but with a considerable observed variance and not as good as a simple SJF scheduler. DQN, on the other hand, still managed to decrease average slowdown to lower levels than FCFS and close to the rates obtained by SJF. Finally, in this case, submission frequency also influenced on the policies' performance, higher values tend to burden convergence even further, decreasing average performance and increasing variance even for DQN.

(a) PPO (b) A2C (c) DQN

Figure 5.7: Training progress on environment workloads with low chance of small jobs (20%), low submission frequency (0.25).



(a) PPO (b) A2C (c) DQN

Figure 5.8: Training progress on environment workloads with low chance of small jobs (20%), high submission frequency (0.5).

5.4 Hyperparameter variations

Hyperparameter variations were introduced in order to better understand the influence of each of them over the training convergence and performance metrics. However, not all the hyperparameters studied in the empirical diagnosis procedure (Table 5.1) lead to observable influence throughout the experiments. In fact, variations in most of them resulted in complete divergence, *i.e.*, the agent became incapable of properly learning its environment. These parameters were, then, left in their default values. Hyperparameters that demonstrated significant influence, either as an improvement or deterioration in training stability, are listed on Table 5.2.

	PPO	DQN	A2C
GAE-λ	0, 0.5, 1.0	not applicable	0, 0.5, 1.0
Buffer size	not applicable	10, 10^4, 10^6	not applicable
Entropy coefficient (loss)	not applied	not applicable	0, 1
Value coefficient	not applied	not applicable	0, 1

Table 5.2: Values for parameter variations included in the study.

5.4.1 Hyperparameters effects during training

Training deep reinforcement learning agents can be a remarkably unstable process, due to the cumulative effect of the variance introduced by each successive choice. Hence, it can be challenging to find a combination of parameters that successfully allows the agent to grasp the nonlinear patterns of its environments that will better direct it to the optimization of the desired metric. Notably, environment aspects themselves can also influence in this process. Experiments were, thus, conducted by testing each of the predefined set of values against both sparse and dense representation environments.

5.4.1.1 PPO

PPO's main characteristic is allowing the agent's policy to be updated to a maximum extent, but preventing the update from diverging too far by clipping its value. On diagnosis stage, clipping range variations produced no significant effects at all or drastic divergent behavior. Clipping too much may prevent the policy from progressing across the high-dimensional optimization space. Similarly, not enough clipping may keep the policy's network in an unstable state, constantly missing the minimal values, being, thus, unable to properly converge towards the optimized target metric.

A good estimation of the advantage function is a core factor in PPO's definition. Record from Section 2.3.3 that the trade-off between bias and variance in advantage estimation is determined by the λ parameter of the formulation. Setting GAE-$\lambda = 0$ yields to lower variance, but higher bias. When GAE-$\lambda = 1$, however, bias is reduced, at the expense of increasing the variance of the estimation.

On sparser environments, as in Fig. 5.9, PPO did not struggle to achieve convergence, regardless of GAE-λ. A balanced value of GAE-$\lambda = 0.5$ allowed the agent to converge quickly and stably. Lower bias, in turn, produced a slower convergence, due to the exploratory nature of high variance. Finally, higher bias lead to quick convergence, albeit less stable than the one obtained in the balanced GAE-$\lambda = 0.5$ configuration.

(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

Figure 5.9: Influence of GAE-λ for training on sparse environments (80% chance of small jobs, new job rate 0.25).

A dense environment, with long and frequent jobs, posed a challenge to PPO, as expressed by Fig. 5.10. The best convergence stability was obtained from a higher bias set-up (GAE-$\lambda = 0$), which indicates that conservative updates are a better strategy for a space in which patterns are harder to discern over time. Encouraging high exploration with GAE-$\lambda = 1.0$ led to a divergent behavior, indicating that the erratic nature of advantage estimation prevents the policy from finding an optimized path in such spaces.

(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

Figure 5.10: Influence of advantage estimation for training on dense environments (20% chance of small jobs, new job rate 0.75).

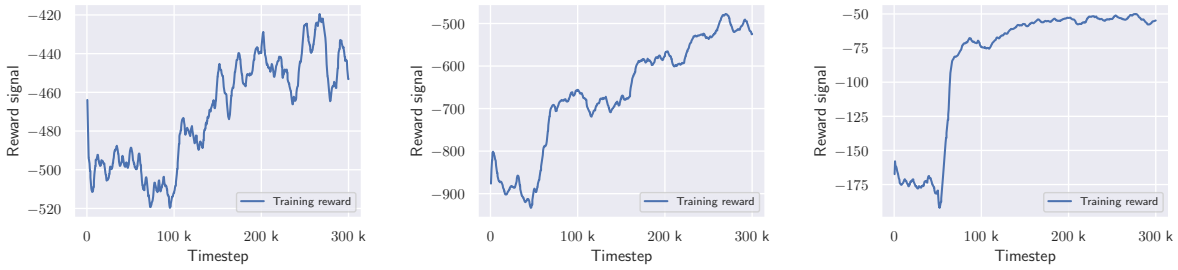
5.4.1.2 DQN

On Stable Baselines' implementation of DQN, the τ coefficient controls how much of the network's parameters visited previously will be considered in the smoothing of the network's updates. For the problem of workload scheduling, network regularization did not exert significant influence during diagnosis stage. Replay buffer size, on the other hand, influences directly at the core functionality of DQN, since it controls how much and for how long throughout training old experiences will be presented to the network.

DQN provided the most consistently stable training curves among the tested techniques, which indicates that the methods employed by DQN to stabilize training – namely: replay buffer, experience replay and target networks – do seem to fit the nature and representation of the scheduling problem at hand. On sparse environments, DQN agents were able to converge quickly and independently of the size of replay buffer, as illustrated in Fig. 5.11. In dense environments, in which samples tend to become hard to distinguish from one another, replay buffer size influence became more prominent. Smaller buffer sizes means that the policy will be exposed less frequently to the features of each sample, leading to increased variance in reward progression over time.

(a) Replay buffer size = 10. (b) Replay buffer size = 10^4. (c) Replay buffer size = 10^6.

Figure 5.11: Influence of replay buffer size for training on sparse environments (80% chance of small jobs, new job rate 0.25).

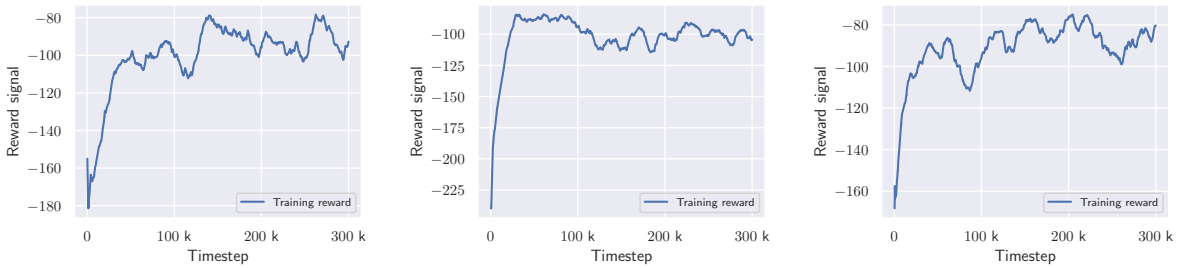


(a) Replay buffer size = 10. (b) Replay buffer size = 10^4. (c) Replay buffer size = 10^6.

Figure 5.12: Influence of replay buffer size for training on dense environments (20% chance of small jobs, new job rate 0.75).

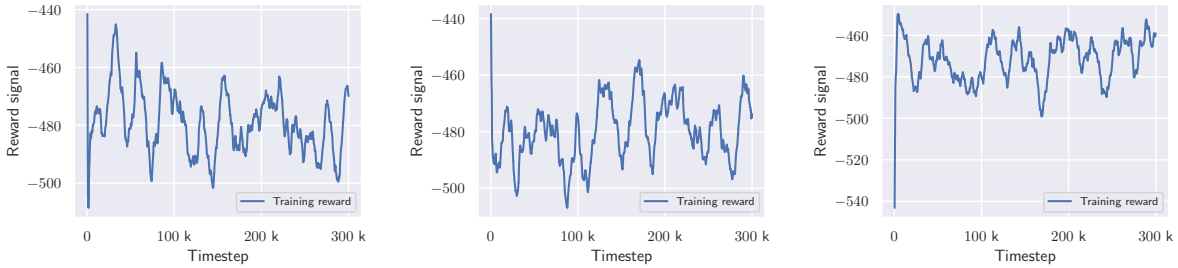
5.4.1.3 A2C

Much like PPO, A2C relies heavily on advantage estimation as a regularizator baseline in its loss function, in order to produce updates that encourage improvements in the relative reward of each action in the trajectory. As illustrated by the plots in Fig. 5.13, A2C managed to sustain stable progress during training on sparse environments, with faster stability being achieved from a low bias and high variance scenario (GAE-λ = 1.0), which, in turn, helps to mitigate loss variance itself. Nevertheless, in dense environments, expressed in Fig. 5.14, higher variance on advantage estimation produces a sensible deterioration in training stability, likely caused by the interplay of environment representation and exploration strategy of the optimization space.



(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

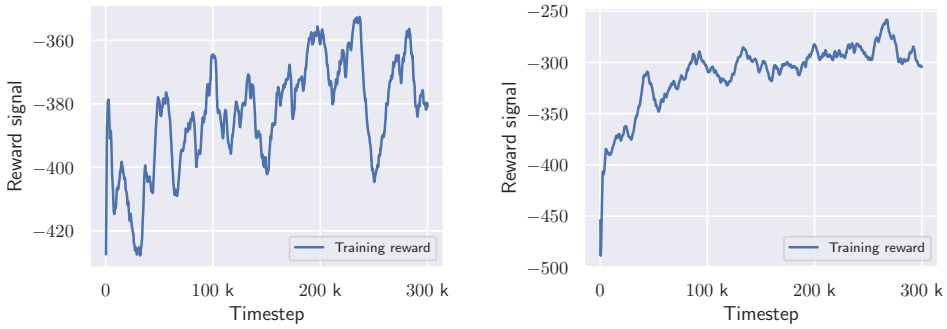
Figure 5.13: Influence of GAE-λ on sparse environments (80% chance of small jobs, new job rate 0.25).



(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

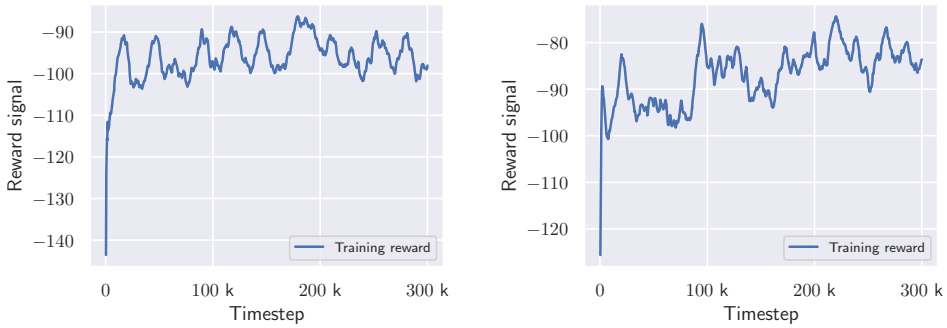
Figure 5.14: Influence of GAE-λ on dense environments (20% chance of small jobs, new job rate 0.75).

Entropy is used as a regularization factor in A2C, since lower action entropy indicates that the policy may be relying too much on a dominant action, which can hinder exploration. Stable baselines' implementation includes entropy regularization for both the actor and the critic models. As shown in Fig. 5.15, however, encouraging repetitive choices in sparse environments produces a more stable training for the scheduling problem, since random job-picking patterns don't tend to result in a good reward progression. The same behavior was not observed in dense environments (Fig. 5.16), in which encouraging exploration did not entail significant difference from a conservative, exploitation-centered balance. It's worth mentioning, though, that the lower job rate (0.25) did improve convergence behavior, even in an exploitative scenario, as opposed to what's observed in Fig. 5.14b.



(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

Figure 5.15: Influence of policy entropy coefficient on sparse environments (80% chance of small jobs, new job rate 0.5).



(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

Figure 5.16: Influence of policy entropy coefficient on dense environments (20% chance of small jobs, new job rate 0.25).

The value estimator is the Critic component of A2C, through which the Actor is capable of making its decisions. In stable baselines 3, its calculation also includes an entropy regularizator that operate similar to that of the policy network. This parameter interacts with the entropy coefficient of the policy producing significant effects: weighting entropy heavily on both prevents training from converging on most cases. On Fig. 5.17, a similar behavior to the policy entropy parameter can be observed in sparse environments, with a stronger exploration component leading to diverging training progress. A similar pattern can be observed for dense environments in Fig. 5.18, with stabler development of the reward signal without entropy penalties.

(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

Figure 5.17: Influence of value entropy coefficient on sparse environments (80% chance of small jobs, new job rate 0.25).

(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

Figure 5.18: Influence of value entropy coefficient on dense environments (20% chance of small jobs, new job rate 0.5).

5.4.2 Evaluation results

Policies produced by the three algorithms were tested against principled schedulers, namely FCFS and SJF, as well as a random action picker. FCFS picks jobs in the order they show up while SJF pick the job in the queue with the smallest expected execution time. Once again, each agent was tested under scenarios with sparse and dense representations. On sparse environments, with distinguishable variance in representation matrices over time, most algorithms managed to produce policies with close or lower average slowdown performance than the best-scoring principled approach, SJF. However, dense environments, with more correlation in the input data, did provide a challenge for the techniques that do not employ some level of resampling recurrence. All the plots

in this section feature the variation for the computed average slowdown value among all runs, expressed by the shaded area.

5.4.2.1 PPO

PPO's dependency over advantage estimation was once again confirmed in evaluation stage. In Fig. 5.19, PPO's policy did manage to come close to principled approaches for the tested values, even occasionally surpassing it, considering the variance of its steps – expressed in the plot as a shading color around the curve. On sparse environments, encouraging higher variance in advantage estimation made convergence slower, when compared to other values for GAE-λ. PPO did not manage to optimize the policy on dense environments in none of the tested parameters setups and regardless of GAE-λ, as expressed in Fig. 5.20. This indicates that the lack of a replay strategy that exposes the network to the same input multiple times, allowing it to better grasp the underlying patterns of such a distribution, is an important factor that must be considered in both representation and network design.

(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

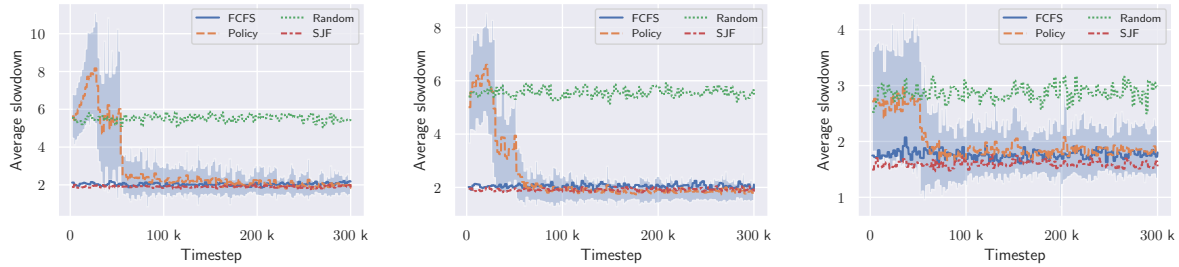
Figure 5.19: Influence of GAE-λ over average slowdown on sparse environments (80% chance of small jobs, new job rate 0.25).

(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

Figure 5.20: Influence of advantage estimation over average slowdown on dense environments (20% chance of small jobs, new job rate 0.5).

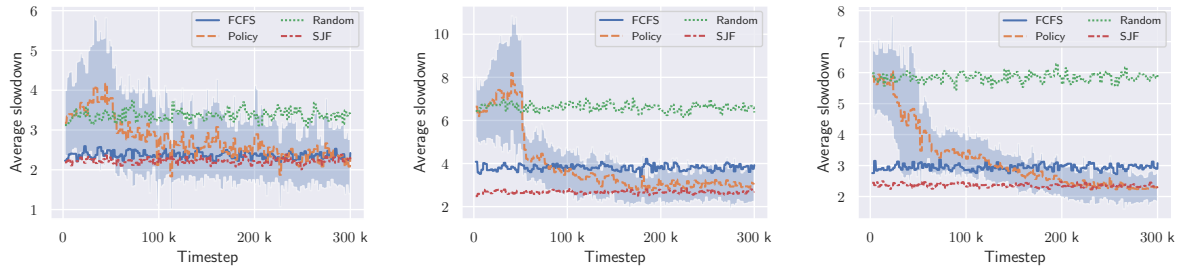
5.4.2.2 DQN

Of all the 3 tested techniques, DQN has a unique characteristic that makes it distinguishable among the others: it is the only one to feature a replay buffer that stores previously visited state-action-reward tuples and presents them multiple times to the network. For scheduling, beyond improving training stability, this feature gives DQN particularly significant advantage over the others, with its agent reaching closer or better performance than principled algorithms. In the sparse environment evaluations, shown in Fig. 5.21, the size of the display buffer did not exert a significant influence on the average slowdown obtained. DQN demonstrated to be well-suited for denser representation environments such as the one expressed in Fig. 5.22, in which even when the display buffer size was orders of magnitude smaller than its testing counterparts, the algorithm managed to learn and produce a capable agent.



(a) Replay buffer size = 10. (b) Replay buffer size = 10^4. (c) Replay buffer size = 10^6.

Figure 5.21: Influence of replay buffer size over average slowdown on sparse environments (80% chance of small jobs, new job rate 0.25).

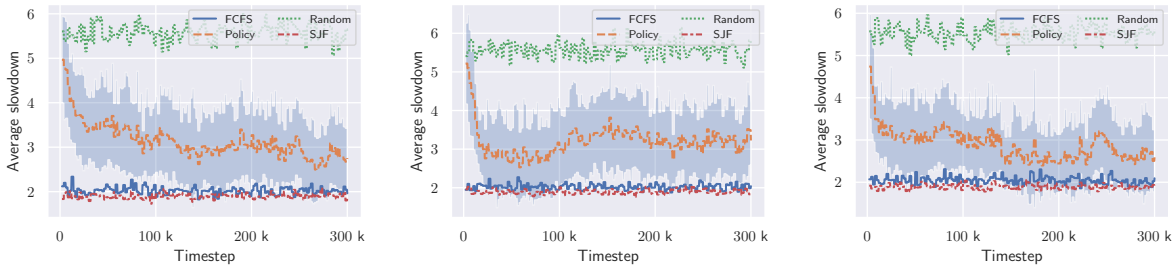


(a) Replay buffer size = 10. (b) Replay buffer size = 10^4. (c) Replay buffer size = 10^6.

Figure 5.22: Influence of replay buffer size over average slowdown on dense environments (20% chance of small jobs, new job rate 0.5).

5.4.2.3 A2C

Although advantage estimation did influence on the training stability of A2C, the same was not observed on evaluation of average slowdown. For both sparse environments (Fig. 5.23) and dense (Fig. 5.24) ones, the balance between exploitation and exploration built into the advantage estimator did not exhibit significant influence over the policy's performance progression over time.



(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

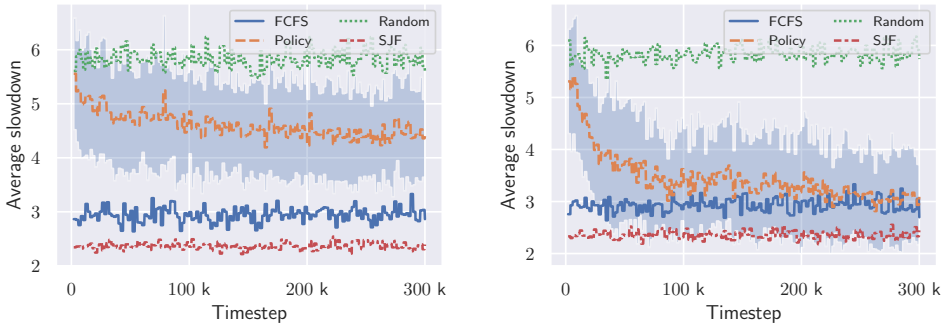
Figure 5.23: Influence of GAE-λ for average slowdown on sparse environments (80% chance of small jobs, new job rate 0.25).

(a) GAE-$\lambda = 0.5$. (b) GAE-$\lambda = 1.0$. (c) GAE-$\lambda = 0.0$.

Figure 5.24: Influence of GAE-λ for average slowdown on dense environments (20% chance of small jobs, new job rate 0.75).

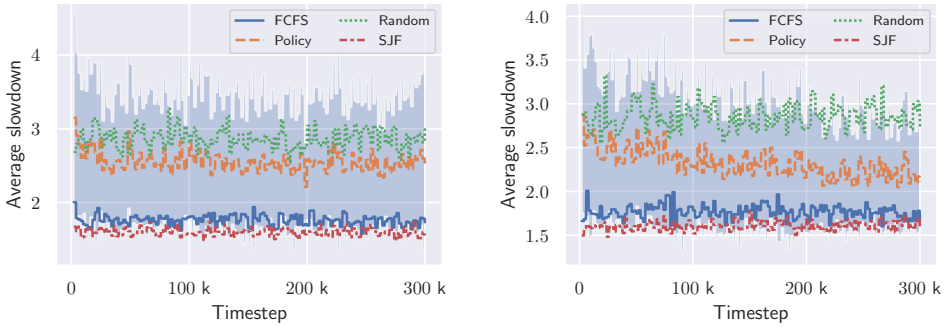
The policy entropy coefficient, in turn, controls the exploration/exploitation dichotomy by weighing on the actor's loss itself. Its influence on average slowdown performance is observably similar to training stability: with a stimulus towards heavy exploration, agents tend to reach marginal performance gains, barely reaching the same level of other simpler algorithms. Encouraging exploitation, on the other hand, allows the agent to be exposed to similar sets of configurations multiple times – an effect roughly comparably, with due proportions, to a replay buffer. Hence, besides providing a stable training curve, lower entropy coefficients on A2C tend to produce better policies that are at least comparable to FCFS. This behavior is observed more clearly on sparse environments,

as exemplified in Fig. 5.25. Overall, A2C struggles to reach satisfactory performance when trained on dense representation contexts, with a modest advantage for low entropy coefficients, as expressed in Fig. 5.16.



(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

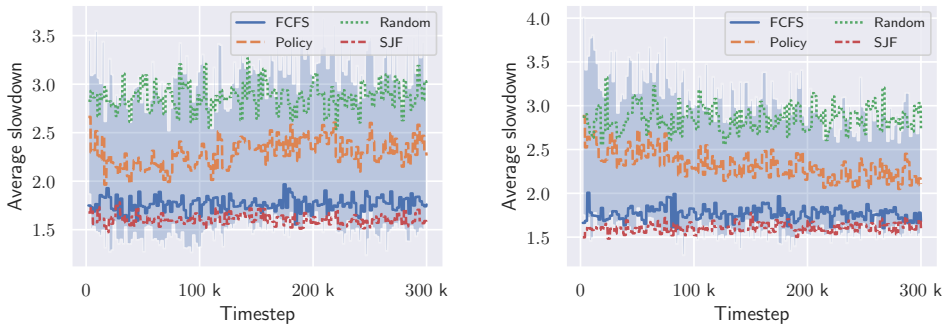
Figure 5.25: Influence of policy entropy coefficient on sparse environments (80% chance of small jobs, new job rate 0.5).



(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

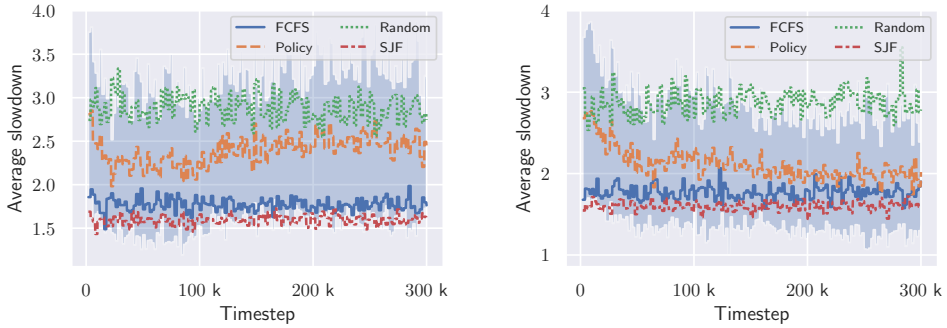
Figure 5.26: Influence of policy entropy coefficient on dense environments (20% chance of small jobs, new job rate 0.25).

Extremes on value entropy coefficient usually tend to deteriorate the quality of the trained agents for both sparse contexts (Fig. 5.27) and dense ones (Fig. 5.28). However, it is still noticeable that avoiding extra stimulus to exploration do tend to provide better results than encouraging the policy towards making more atypical choices too frequently. The nature of the studied job scheduling simulation itself is fairly repetitive, bursts of intense jobs don't tend to happen too frequently, therefore providing a certain level of benefit for repetitive choices over time.



(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

Figure 5.27: Influence of value entropy coefficient for average slowdown on sparse environments (80% chance of small jobs, new job rate 0.25).



(a) Entropy coefficient = 1. (b) Entropy coefficient = 0.

Figure 5.28: Influence of value entropy coefficient for average slowdown on dense environments (20% chance of small jobs, new job rate 0.5).

5.4.2.4 Action choice distribution

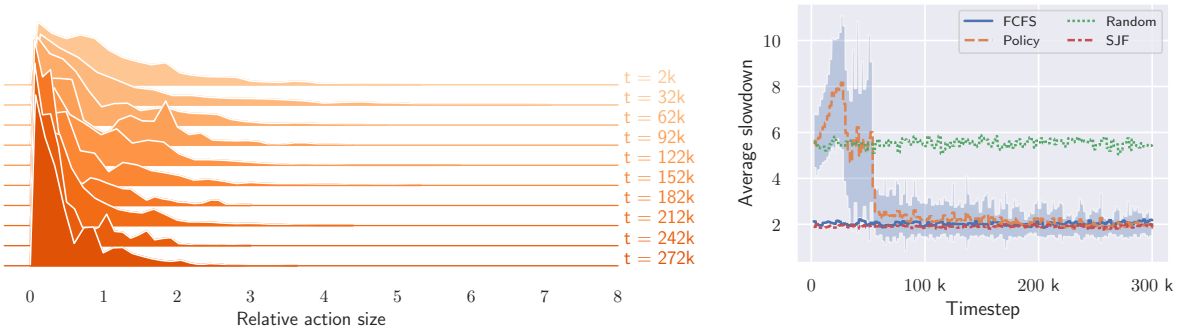
During evaluation, all policies were compared against Shortest-Jobs-First, a principled scheduler algorithm that always picks the smallest job in the queue to be schedule at each time point. The performance of SJF was often matched or eventually surpassed by many of the learning-based policies, specially the ones in which DQN was applied. In order to better understand if matching the performance of SJF also implies that the learned policy automatically acquired a behavior akin to SJF's smallest job principle by itself, we've computed histograms that express the distribution of relative job sizes of the picked jobs compared with other jobs in the waiting queue for every choice made during evaluation step. By doing so, we could evaluate how each policy progresses over time regarding the size metric.

Consider $Q = \{j_1, j_2, \cdots, j_n\}$ the set of n jobs that represents the waiting queue, and $\text{size}(j_i), 1 \leq i \leq n$ a function that maps each job to their respective estimated time. Then, the relative job size r_k of a job $j_k, 1 \leq k \leq n$ is expressed by Eq. (5.1).

$$r_k = \frac{n \times \text{size}(j_k)}{\sum_{i=1, i \neq k}^{n} \text{size}(j_i)} \tag{5.1}$$

Essentially, if a relative job size is smaller than, or closer to the average size of the remaining jobs in the queue, r_k will be a value between 0 and 1. If, however, the picked job is bigger than average, the relative job size will be a value greater than 1.

Results were plotted for each time step and stacked for comparison. In Fig. 5.29, we observe a DQN instance that converged to an average slowdown value marginally better than the one obtained by SJF and FCFS. At the beginning of training, around $t = 2k$ time steps, the relative picked job size distribution had a peak between 0 and 1, but spanned across larger values, implying that jobs larger than average were being picked with certain frequency. Naturally, at the beginning of training, the learning-based agent still knows very little about its environment and most of its decisions are merely exploratory endeavors. As training progressed, around $t = 272k$ timesteps, the peak in the interval $[0, 1]$ got taller, indicating that as the agent learned more about how to optimize its reward metric, its behavior regarding relative job size became progressively similar to the policy of shortest-jobs-first. It is worth noting that the agent learned such behavior unsupervisedly, under the constraints of the reinforcement learning framework.



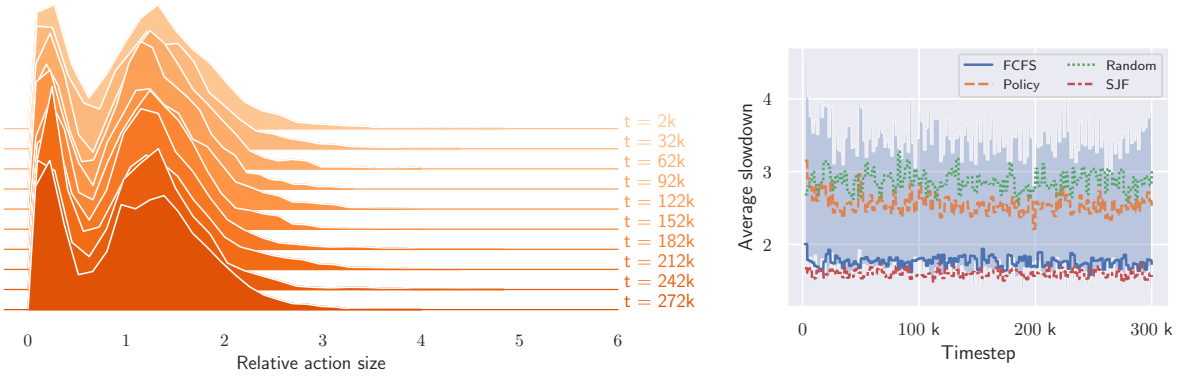
(a) Relative picked job size distribution.

(b) Average slowdown progression.

Figure 5.29: Probability density distribution of job sizes, relative to the average size of jobs in the waiting queue, for a DQN instance that surpassed SJF performance. Sizes are expressed horizontally, while the peak heights show frequency. Progress over time is expressed by the stacked t timesteps.

We've also investigated the relative picked job size distributions for policies that did not achieve comparable performance to that of SJF and FCFS. In Fig. 5.30, an A2C agent trained on a dense environment with high entropy stimulus in its loss is illustrated.

Relative picked job size distribution remained approximately the same from the beginning at $t = 2k$ time steps, all the way to the end at $t = 272k$ time steps, which indicates that the agent was not able to learn much from the environment under the exploration-focused hyperparmater configuration determined for its training. Moreover, even though a peak exists on the range of smaller jobs, the frequency of larger ones is still considerably high, which, combined to the performance plot, show that the algorithm did not acquire SJF-like job picking patterns.



(a) Relative job size distribution.

(b) Average slowdown progression over time.

Figure 5.30: Probability density distribution of job sizes, relative to the average size of jobs in the waiting queue, for an A2C instance that did not achieve comparable performance to SJF. Histograms are plotted similarly to Fig. 5.29

5.5 Retraining from sparse to dense environments

Bearing a certain resemblance to natural human education, that usually happens over a course of several steps hierarchized by the complexity of each knowledge domain, reinforcement learning algorithms can sometimes benefit from a training procedure segmented in stages of progressing difficulty. Challenging environments can sometimes hinder the progression of a policy by preventing it from grasping the basic features necessary to advance its performance to the next level required to navigate in such conditions. Hence, training agents in simpler versions of an environment before exposing them to more demanding ones gives the policy a head start that makes training stabler and convergence towards optimization faster.

We tested all three algorithms by running a regular $300k$-steps training session on a sparse environment (80% chance of small jobs, 0.25 new job rate), then reloading them

into a dense, more challenging, environment (20% chance of small jobs, 0.5 new job rate). The set of parameters for each of them is expressed in Table 5.3. Values were chosen according to the performance they provided for each algorithm in dense environments, showing promising signs of convergence towards average slowdown optimization. Each algorithm was compared to a fresh instance, trained under the same conditions as their retrained counterparts, but only within the challenging environment space.

	PPO	DQN	A2C
GAE-λ	1.0	not applicable	0.5
Buffer size	not applicable	10^3	not applicable
Entropy coefficient (loss)	not applied	not applicable	0
Value coefficient	not applied	not applicable	0

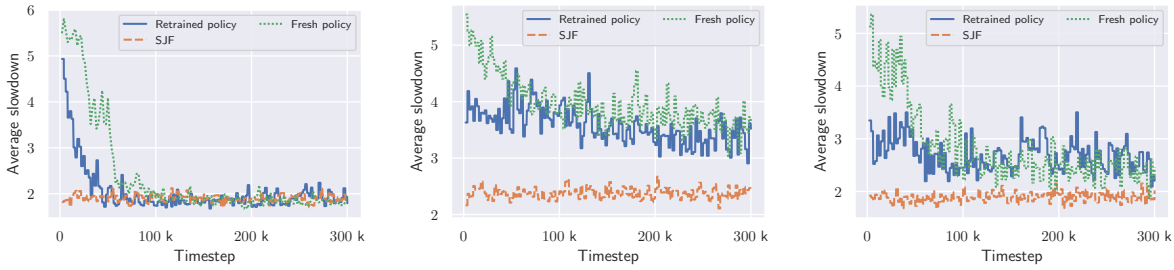
Table 5.3: Values for parameters chosen for the retrain study.

In Fig. 5.31, the training curve for the retrained agents shows slight improvement in stability and a tendency to produce higher reward signals in comparison to policies trained from a clean state at the initial cycles of training. For DQN, which of all three was the one that achieve favorable performance in dense environments more consistently, the difference is marginal. The difference in training behavior is more pronounced for A2C and PPO, specially in the first $100k$ time steps, just before the fresh policy catches up and the two start to progress together with similar levels of reward signal.

(a) DQN. (b) A2C. (c) PPO.

Figure 5.31: Training reward curve for retrained agents, compared to agents trained from a clean state.

All the algorithms were also evaluated for average slowdown performance following a similar procedure than the described the previous sections, on the parameter variation studies. At every 2000 training time steps, neural network parameters were frozen and evaluated for 50 episodes. The advantage of the retrained policies is noticeable in this scenario, as exposed in Fig. 5.32. For all three algorithms, retrained models started off from lower slowdown values, or converged faster to a level close or lower than the ones obtained by the freshly trained policies.



(a) DQN. (b) A2C. (c) PPO.

Figure 5.32: Average slowdown progression for retrained agents, compared to agents trained from a clean state.

On a broader analysis, results show that, despite the faster convergence and slightly better results than freshly-trained agents, adopting a two-step training routine may not provide significant improvements. However, models that were trained under different configurations of environments have shown robustness and resilience to adapt to new environment conditions, in which workload characteristics have changed dramatically in comparison to their original training setup. Thus, through this procedure, it is possible to obtain agents that are flexible enough to operate confidently even under unstable circumstances.

Chapter 6

Conclusions

The problem of scheduling the jobs from a computer cluster in order to ensure that the oftentimes expensive and energy-consuming resources associated with these machines are used with maximum efficiency can be a hard to tackle task that requires more than *ad-hoc*, principled solutions. Moreover, sudden bursts of jobs and other extrinsic factors can suddenly change the nature, size, and frequency of the workload, creating gaps that generalist and simple scheduling policies cannot fulfill, creating demand for logically complex algorithms. Given this context, learning-based alternatives can provide interesting developments in the field of workload management. Machine learning is a field dedicated to the study of computational solutions devoted to problems of high level of non-linearity that exist on domains of high dimensionality. Hence, it provides a resourceful tool set for the problem.

Contrary to supervised learning frameworks, reinforcement learning's core principle consists in not providing its internal mechanics much prior information about the problem at hand. This characteristic makes RL-based applications suitable for dynamic domains with well-defined action spaces, rewards and environments, such as games, in which an automated agent is free to explore and learn, guided only by its experiences over time. The flexibility of this definition can be expanded to multiple other classes of problems, including resource management and scheduling. However, problems caused by the effects of high variance over episode samples, the *bias-variance* trade-off or even poorly chosen hyperparameter choices can make even the best-designed RL solutions difficult to converge towards the desirable results. It is, then, important that these solutions are designed, trained and tested under reproducible contexts, preferably with the aid of libraries that are battle-tested by the community and industry. It can also be helpful that the available hyperparameters are studied under a consistent methodology that provides informed decisions within well-oriented testing protocols.

Under these principles, this work aimed to tackle the problem of learning-based resource scheduling with the aid of reinforcement learning frameworks. The extensive hyperparameter search diagnostics shown that the chosen set of configurations can be a determining factor between convergence and highlighted the importance of extensive ablative studies in order to ensure that the training process will converge stably and as

fast as possible. The chosen training technique is also an important aspect, given that the implementation protocols, internal data flow, input/output representation and the very engineering of these algorithms exert influence in the process. From the three deep RL approaches studied in this work, DQN, PPO and A2C, DQN was shown to be less prone to divergence and provide better results than its peers.

Among the most influential factors over training stability and production of good results were extrinsic characteristics from the agent, *i.e.*, environment particularities. Naturally, aspects such as job rate and job size could exert influence over each algorithms ability to properly navigate through their learning domains. Agents primed on simple environments, for example, performed better than agents trained from scratch, given the demand to navigate within an environment with harder to handle situations. It's also worth noting that these characteristics are inherently associated with job representation and different alternatives could be exploited in future works.

Finally, even though most training configurations did not manage to surpass simple principled scheduling algorithms such as SJF and FCFS for all the tested scenarios, the behavior of the trained agents provided some insights on the internal logic of the associated neural networks. DQN agents with better results for convergence and average slowdown tended to pick the smaller jobs available in the queue, essentially meaning that the policy the agent learned without any prior information about the environment by sampling from its experiences was similar to SJF's core idea: let the smallest jobs be allocated first. On the other hand, agents trained from a poorly chosen combination of RL technique and parameters did not exhibit such behavior. Such result may spark further investigations in future work about the influence of job size for allocation and how to use this information to build better policies.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study, 2020.

[3] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[4] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In Kevin Jeffay, Ion Stoica, and Klaus Wehrle, editors, *Quality of Service — IWQoS 2003*, pages 381–398, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] Zhijia Chen, Yuanchang Zhu, Yanqiang Di, and Shaochong Feng. A dynamic resource scheduling method based on fuzzy control theory in cloud environment. *Journal of Control Science and Engineering*, 2015.

[9] R. L. F. Cunha and L. Chaimowicz. Towards a common environment for learning scheduling algorithms. *2020 Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems- MASCOTS*, 2020.

[10] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International conference on learning representations*, 2019.

[11] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation.* Cambridge University Press, USA, 1st edition, 2015.

[12] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[13] Bogdan Filipič. *A Genetic Algorithm Applied to Resource Management in Production Systems*, pages 101–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[14] Borko Furht and Armando Escalante. *Handbook of Cloud Computing.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[15] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[16] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

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

[18] Christian D Hubbs, Can Li, Nikolaos V Sahinidis, Ignacio E Grossmann, and John M Wassick. A deep reinforcement learning approach for chemical production scheduling. *Computers & Chemical Engineering*, 141:106982, 2020.

[19] R. Jain and R.A.Y.A. JAIN. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley professional computing. Wiley, 1991.

[20] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018.

[21] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[23] M. Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing, 2018.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[25] Ke Li and Jitendra Malik. Learning to optimize, 2016.

[26] Y. Liu, X. Zhang, H. Li, and D. Qian. Allocating tasks in multi-core processor based parallel system. In *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, pages 748–753, 2007.

[27] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[28] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. Association for Computing Machinery, 2019.

[30] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264*, 2018.

[31] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[35] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[36] Gaith Rjoub, Jamal Bentahar, Omar Abdel Wahab, and Ahmed Saleh Bataineh. Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e5919, 2020.

[37] Esmat Samadi, Ali Badri, and Reza Ebrahimpour. Decentralized multi-agent based energy management of microgrid using reinforcement learning. *International Journal of Electrical Power & Energy Systems*, 122:106211, 2020.

[38] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[39] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[40] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[42] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games, 2019.

[43] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[44] Victor Silva and Luiz Chaimowicz. Moba: a new arena for game ai, 2017.

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

[46] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org, 2014.

[47] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[48] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[49] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[51] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Boston, MA, 4 edition, 2014.

[52] Lauritz Thamsen, Ilya Verbitskiy, Sasho Nedelkoski, Vinh Thuy Tran, Vinícius Meyer, Miguel G Xavier, Odej Kao, and César AF De Rose. Hugo: a cluster scheduler that efficiently learns to select complementary data-parallel jobs. In *European Conference on Parallel Processing*, pages 519–530. Springer, 2019.

[53] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *IEEE Network*, 32(2):92–99, 2018.

[54] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[55] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation, 2017.

[56] Deliang Yi, Xin Zhou, Yonggang Wen, and Rui Tan. Efficient compute-intensive job allocation in data centers via deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1474–1485, 2020.

[57] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2016.