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

Renato Luiz de Freitas Cunha

Markov decision processes for optimizing job scheduling with Reinforcement Learning

Belo Horizonte
2022

Renato Luiz de Freitas Cunha

Markov decision processes for optimizing job scheduling with Reinforcement Learning

Final Version

Dissertation 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 Doctor in Computer Science.

Advisor: Luiz Chaimowicz

Belo Horizonte
2022

Cunha, Renato Luiz de Freitas.

C972m Markov decision processes for optimizing job scheduling with reinforcement learning [manuscrito] / Renato Luiz de Freitas Cunha – 2022.
103 f. il.

Orientador: Luiz Chaimowicz.
Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f. 95-104.
.
1. Computação – Teses. 2. Aprendizado por reforço– Teses. 3. Computação de alto desempenho – Teses. 4. Aprendizado do computador – Teses. 5. Markov, Processos de – Teses. I. Chaimowicz, Luiz. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III. Título.

CDU 519.6*82(043)

Ficha Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

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

FOLHA DE APROVAÇÃO

MARKOV DECISION PROCESSES FOR OPTIMIZING JOB SCHEDULING WITH REINFORCEMENT LEARNING

RENATO LUIZ DE FREITAS CUNHA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Prof. Luiz Chaimowicz - Orientador
Departamento de Ciência da Computação - UFMG

Dra. Ana Paula Appel
Client Engineering - IBM

Prof. Dorgival Olavo Guedes Neto
Departamento de Ciência da Computação - UFMG

Profa. Jussara Marques de Almeida Gonçalves
Departamento de Ciência da Computação - UFMG

Prof. Marcos Dias de Assunção
Department of Software and IT Engineering - École de technologie supérieure

Belo Horizonte, 19 de julho de 2022.

Documento assinado eletronicamente por **Luiz Chaimowicz**, **Professor do Magistério Superior**, em 23/08/2022, às 17:47, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Jussara Marques de Almeida Goncalves**, **Professora do Magistério Superior**, em 24/08/2022, às 16:48, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Dorgival Olavo Guedes Neto**, **Professor do Magistério Superior**, em 24/08/2022, às 18:50, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Ana Paula Appel**, **Usuário Externo**, em 14/09/2022, às 15:42, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcos Dias de Assunção**, **Usuário Externo**, em 15/09/2022, às 17:16, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1699724** e o código CRC **F17F9FD4**.

To my family.

Acknowledgments

As I finish authoring this dissertation, two points are clear to me. First, this contribution is a small drop in the ocean of scientific knowledge, and secondly, there would be no dissertation at all if not for the support of many people, of which I will name a few below. For the ones that go unmentioned, my sincere apologies.

I still clearly remember how, while walking on the beach in early 2016, my wife Alice encouraged me to apply for a PhD program. Between then and now, I had her utmost support, love, and encouragement. Especially this last year, while I finished this dissertation, I was able to doubly count on her support, as she had to take care of our pride and joy, our baby boy Caio, while I was writing and performing experiments. Similarly, my mom Nati, my dad Luiz, and my sister Lívia were always there for me, continually providing kind words when I hit the inevitable roadblocks that are so present in research life.

Before there was any hint of a dissertation, I had to take my mandatory coursework, and, since taking graduate-level classes remotely was anything but the norm in 2016, I was able to count on the kindness of my dear friend, Rafael Barra, who welcomed me at his house during my back-and-forth travels between São Paulo and Belo Horizonte.

I have much to thank my thesis advisor and friend, Luiz Chaimowicz, who is not only a great researcher but also the coolest dude in the department. Luiz accepted me as a student even though I was working full time and living in another state. Other professors would have thought this PhD was doomed to fail. Still, Luiz ensured this was a success by always providing positive feedback, asking the right questions, and being open to new research ideas, while also proposing alternatives when those ideas did not pan out.

Many thanks to the thesis committee, who provided invaluable feedback and helped point this research in the right direction. In alphabetical order: Ana Paula Appel, Dorgival Olavo Guedes Neto, Jussara Marques de Almeida Gonçalves, and Marcos Dias de Assunção; my sincere thank you!

Finally, I would like to thank Marco Netto, who was my manager at the IBM Brazil Research Lab and who supported me during most of my time as a PhD student. Likewise, my colleagues in the ICT team, Bruno, Eduardo, and Lucas, I owe you a lot for helping me grow as a researcher. I also would like to thank my colleagues at the VeRLab and J labs and the staff at DCC and PPGCC at UFMG, who keep the department and the program running smoothly. Thank you all!

"*So do all who live to see such times.*
But that is not for them to decide.
All we have to decide is
what to do with the time that is given us."
(Gandalf, the Grey)

Resumo

À medida que nossos sistemas computacionais se tornam maiores e com interações mais complexas, apresenta-se um potencial para o uso de técnicas de aprendizado que se adaptem a variações nas condições de sistemas durante a evolução das cargas de trabalho. O arcabouço de tomada de decisão sequencial fornecido por Aprendizado por Reforço (RL) se adapta bem a problemas de gerenciamento de recursos. Ainda assim, quando consideramos o uso de Aprendizado Profundo para escalonamento de *jobs* de sistemas de Computação de Alto Desempenho (HPC), vemos que trabalhos existentes ou focam em problemas menores, como a decisão de escolha de heurísticas dentro de um conjunto, ou em instâncias simplificadas do problema.

Nesta tese, investigamos modelos de Processos de Tomada de Decisão de Markov (MDP) para resolver o escalonamento de *jobs* HPC, apresentando uma abordagem para experimentação mais rápida e reprodutível. Sobre essa fundação, investigamos como diferentes agentes se comportam nesse arcabouço, ao mesmo tempo que identificamos deficiências tanto na representação do problema quanto como o aprendizado se dá nesse cenário.

Dentre as contribuições deste trabalho, propomos um sistema de software para desenvolvimento e experimentação com agentes de RL, bem como avaliamos algoritmos de estado-da-arte nesse sistema, com desempenho equivalente ao de algoritmos específicos, porém com menos esforço computacional. Nós também mapeamos o problema de escalonamento de jobs HPC para o formalismo de SMDP e apresentamos uma solução *online*, baseada em aprendizado por reforço profundo que usa uma modificação do algoritmo PPO para minimizar *slowdown* de *jobs* com máscara de ações, adicionando suporte a grandes espaços de ações ao sistema.

Em nossos experimentos, nós avaliamos os efeitos de ruído nas estimativas de tempo de execução em nosso modelo, observando como ele se comporta tanto em clusters pequenos (64 processadores) quando em clusters grandes (163840 processadores). Nós também mostramos que nosso modelo é robusto a mudanças em carga de trabalho e nos tamanhos de clusters, demonstrando que a transferência de agentes entre clusters funcionam com mudanças de tamanho de cluster de até 10×, além de suportar mudanças de carga de trabalho sintético para seguir a execução de traços de sistemas reais. A abordagem de modelagem proposta possui melhor desempenho que outras da literatura, tornando-a viável para a criação de modelos de escalonamento robustos, transferíveis e capazes de aprender.

Palavras-chave: aprendizado por reforço, computação de alto desempenho, escalonamento em lote, aprendizado de máquina, processos de decisão de markov

Abstract

As our systems become larger and their interactions more complex, there is a potential for learning techniques that adapt to varying system conditions as workloads evolve. The framework for sequential decision making provided by Reinforcement Learning (RL) fits well with resource management problems, as recent literature indicates. Yet, when we consider the use of Deep Learning for the scheduling of batch jobs in High Performance Computing (HPC) systems, we see that work in the literature either focuses on smaller problems, such as deciding which heuristic to use at a given time, or on simplified instances of the problem.

In this dissertation, we investigate Markov Decision Process (MDP) models to solve the problem of scheduling batch HPC jobs, presenting an approach for faster and reproducible experimentation. With this foundation, we investigate how different agents behave under this framework, while identifying deficiencies both in the representation of the problem and how learning proceeds in such a setting.

Among the contributions of this work, we propose a software system for developing and experimenting with RL agents, and we evaluate different state-of-the-art algorithms from the literature in this environment, achieving performance equivalent to that of purpose-built algorithms, but with lower resource usage. We also map HPC batch job scheduling to the SMDP formalism, and present an online, deep reinforcement learning-based solution that uses a modification of the Proximal Policy Optimization algorithm for minimizing job slowdown with action masking, supporting large action spaces.

In our experiments, we assess the effects of noise in run time estimates in our model, evaluating how it behaves in small (64 processors) and large (163840 processors) clusters. We also show our model is robust to changes in workload and in cluster sizes, showing transfer works with changes of cluster size of up to 10×, and changes from synthetic workload generators to supercomputing workload traces. The proposed model outperforms learning models from the literature and classic heuristics, making it a viable modeling approach for robust, transferable, learning scheduling models.

Keywords: reinforcement learning, high-performance computing, batch job scheduling, machine learning, markov decision processes

List of Figures

1.1 Example of the type of system considered in this dissertation. 16

2.1 A possible schedule when tree jobs arrive in the system. 23
2.2 Expected scheduling behavior of FCFS and conservative backfilling. 24
2.3 Major types of jobs found in workloads 26
2.4 Agent-Environment interaction loop. 29
2.5 Graphical representation of the Mao workload model. 30
2.6 Graphical representation of the Lublin workload model. 31
2.7 Entropy of a Bernoulli random variable as a function of θ. 37
2.8 Relationship between an MDP and a Semi-MDP 39
2.9 Example of the RL agent-environment loop and corresponding code. 41

4.1 Example of a test of the simulator 52
4.2 Snapshot of three frames of the base environment. 54
4.3 Image-like state representation. 59
4.4 Difference in rewards in the various types of MDPs. 62
4.5 Compact state representation in the option-based Semi-MDP. 63

5.1 Schematic view of a three-layered neural network. 68
5.2 Learning curves of PG algorithms with an average cluster load of 70%. 69
5.3 Learning curves of the A2C and PPO agents with an average cluster load of 70%. . 71
5.4 Learning curves for various scenarios with $H = 20$. 74
5.5 Average slowdown for the various scenarios considered. 75
5.6 Time needed to train agents for three million iterations. 75
5.7 Difference in episode length given reward computation method. 76
5.8 Performance comparison between transferred and specialized agents. 77
5.9 Performance with the Mao workload model when using different noise factors. . . 79

List of Tables

1.1 Summary of the notation used in this dissertation. 20

4.1 Job features in a compact state representation. 59
4.2 Job features in the state representation for the option-based Semi-Markov Decision Process (SMDP). 64

5.1 Hyperparameters for experiments comparing with the base PG algorithm. 72
5.2 List of hyper-parameters used when training PPO agents. 73
5.3 Key to the scenarios presented in Figure 5.5. 75
5.4 Environment and workload model parameters used in the Mao-Gaussian set of experiments. 78
5.5 Environment, workload model, and learning parameters used in the experiments that used the Lublin workload model. 81
5.6 Comparison of average bounded slowdown between the SMDP model and a model from the literature. 81
5.7 Comparison of average resource utilization between the SMDP model and a model from the literature. 82
5.8 Workload traces used when evaluating our models. 82
5.9 Average bounded slowdown of models when using the Lublin and Tsafrir workload models. 84
5.10 Average bounded slowdown when scheduling according to trace files. 86
5.11 Cluster utilization for the trace files used in this section. 87

List of Acronyms

A2C	Advantage Actor-Critic	37
AI	Artificial Intelligence	48
ALE	Arcade Learning Environment	15
API	Application Programming Interface	18
CNN	Convolutional Neural Network	44
DL	Deep Learning	44
DeepRM	Deep Resource Management	43
DRAS	Deep Reinforcement agent for Scheduling in HPC	43
EASY	Extensible Argonne Scheduling sYstem	67
FCFS	First-Come First-Serve	24
FIFO	First In First Out	24
HPC	High Performance Computing	9
MDP	Markov Decision Process	9
ML	Machine Learning	15
MLP	Multi-Layer Perceptron	54
PG	Policy Gradient	36
POMDP	Partially-Observable Markov Decision Process	88
PPO	Proximal Policy Optimization	38
PS	Parameter Server	44
QoS	Quality-of-Service	47
RL	Reinforcement Learning	9
SJF	Shortest Job First	25
SLA	Service Level Agreement	16
SMDP	Semi-Markov Decision Process	11
SWF	Standard Workload Format	52
TDD	Test-Driven Development	51
VM	Virtual Machine	46

*

Contents

1 Introduction **15**

1.1 Motivation: HPC job scheduling . . . 16
1.2 Objectives . . . 17
1.3 Contributions . . . 18
1.4 Notation . . . 19
1.5 Document layout . . . 19

2 Background **21**

2.1 Batch job scheduling . . . 21
2.2 Deep Reinforcement Learning and Job Scheduling . . . 27
2.3 Workload models . . . 29
2.4 Uncertainty in job run time estimates . . . 32
2.5 Policies and approximators . . . 33
2.6 Options as a closer-to-reality model . . . 38
2.7 OpenAI Gym . . . 40

3 Related Work **42**

3.1 Reinforcement Learning for Scheduling . . . 42
3.2 On predicting job features . . . 45
3.3 Reinforcement Learning in Resource Management . . . 46
3.4 Resource allocation environments in OpenAI Gym . . . 47
3.5 Summary . . . 48

4 HPC job scheduling with RL **49**

4.1 Problem description . . . 49
4.2 On the need for simulators and frameworks . . . 50
4.3 Job Scheduling Simulation . . . 51
4.4 The simulator as an OpenAI Gym environment . . . 53
4.5 Policy network and learning procedure . . . 56
4.6 Alternative MDP formulations . . . 58
4.7 The options-based Semi-MDP formulation . . . 61
4.8 Summary . . . 66

5 Experiments **67**

5.1 Learning performance in the base MDP 67
5.2 Solving the alternative MDP formulations 72
5.3 Performance in the option-based SMDP model 80

6 Conclusion **90**
6.1 Overview 90
6.2 Contributions 91
6.3 Limitations and directions for future research 93

Bibliography **95**

Chapter 1

Introduction

In recent years, we have seen significant progress in diverse areas due to the use of Machine Learning (ML) techniques. Examples include healthcare [Beaty et al., 2016], network analysis [Appel et al., 2018], resource management [Cunha et al., 2014, 2017, De Assunção et al., 2016, Mao et al., 2016], robotics [Glaubius et al., 2010], user-support tools [Rodrigues et al., 2016], and, most notably, games, with progress mainly driven by Reinforcement Learning (RL). Both board games, such as Backgammon [Tesauro, 1994], Chess [Silver et al., 2018], and Go [Silver et al., 2018], as well as digital games, such as Atari games in the Arcade Learning Environment (ALE) [Bellemare et al., 2013, Hessel et al., 2018], have seen the development of agents with super-human performance thanks to the application of ML techniques.

A common thread running through all the successful applications mentioned above is that learning is enabled by the availability of extensive datasets, simulators capable of allowing algorithms to learn at a rate much faster than real time, or both. Of these, computational resource management is a candidate application area which has attracted interest in the past, but that still has much room for improvement, both from a theoretical and from a practical perspective. Learning algorithms for computational resource management might be a good solution to deal with changing, heterogeneous workloads that try to optimize and adapt to these workloads.

Despite there being a rich literature with strong analysis of classic resource management algorithms, there is still room for algorithm discovery and the development of systems that support learning for systems. Of these, we are interested in scheduling, and how to model jobs for learning in ML systems. Questions that arise naturally, then, are related to whether ML algorithms are capable of enabling learning agents to efficiently schedule resources when subject to *realistic* workloads. Such problems can be seen as sequential decision problems, since jobs arrive and exit sequentially, and can be solved with sequential learning models, such as RL.

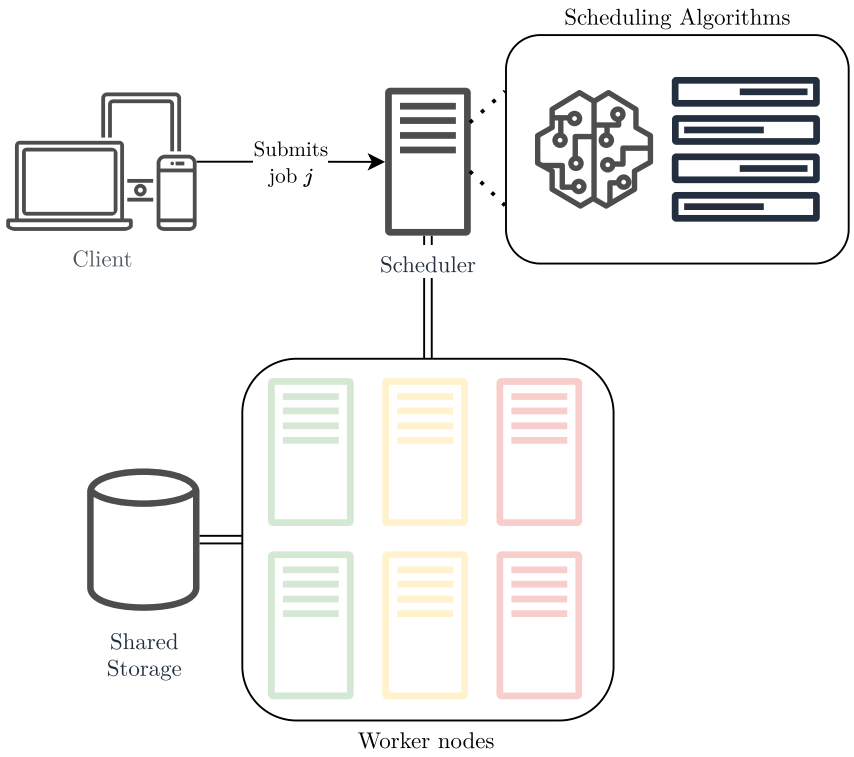


Figure 1.1: Example of type of system considered in this dissertation. A user, via a client computer interacts with the scheduling system, usually via a login node, to submit jobs. The scheduling system implements scheduling algorithms and policies for job scheduling, and decides where to run jobs.

1.1 Motivation: HPC job scheduling

Consider the case of scheduling jobs in a modern enterprise computing system. It may comprise multiple machines over geographically disperse data-centers and, each time a new job arrives, the scheduler must decide where to execute the job considering the resource requirements, Service Level Agreements (SLAs), and current system utilization. The sheer number of variables in such a situation prevents this problem from being solved exactly.

Figure 1.1 shows a high-level example of how a High Performance Computing (HPC) scheduling system may be configured: clients submit jobs to the scheduling system that controls a cluster of machines. The scheduler, then, has to choose, based on resource requirements, resource availability and overall host usage where to run the user job. Since the system has multiple users, it also has to guarantee some fairness, to enable all jobs to run, and so that no single user takes over all the cluster's resources.

In this setting, ML techniques may solve this problem in an adaptive way, enabling the system to evolve and respond to changes in the workloads and underlying resource characteristics.

For this reason, we make use of sequential decision-making formulations and apply

Reinforcement Learning (RL) techniques to explore the impacts of learning algorithms in managing resources in computational clusters. In doing so, we model real systems and use real workload traces to evaluate the impacts of using these algorithms in such systems.

Due to the impossibility of disrupting users of real systems in testing such algorithms live, we make heavy use of simulation to analyze the impact of learning algorithms in such systems. While doing so, we propose a new tool to evaluate RL agents in such systems, and investigate issues that arise when designing automated ways to learn the management of resources.

1.2 Objectives

Our thesis is that, by carefully designing an environment that follows an RL formulation, we are able to train algorithms capable of learning to *efficiently* schedule jobs in realistically-sized HPC systems.

As will become clear later in this text, many applications of ML that attempt to solve scheduling make assumptions about the problem that we consider too limiting. For example, many learning approaches assume job run time estimates are accurate, while others fail to capture the impacts of the sequential nature of the problem of scheduling HPC jobs, yet others assume cluster configurations aren't expected to change, and explicitly discourage transferring agents between clusters.

Our main goal is to model scheduling HPC jobs for realistically sized clusters as a sequential decision-making problem, with a corresponding decision process and the design of agents able to learn in such an environment. We also want to understand the effects of uncertainty in job run time estimates, as that is a crucial factor in the quality of scheduling systems. In our investigation, we will have to remove some simplifying assumptions commonly found in the literature, while also providing a formulation of the problem closer to how such an agent could potentially be used in production systems.

In our research, we adhere to the following guidelines:

(G1) *An agent must learn without relying on the characteristics of a* single *system*: We want to be able to train learning agents that support being transferred between systems. Granted, these transferred agents might not be the best ones, needing retraining, but they should be able to make decisions with changes to cluster configuration.

(G2) *The approach must not depend on powerful systems for learning:* It is common in ML to have models that require hundreds of GPUs and thousands of processors for learning. In practice, current scheduler systems must make their decisions quickly, and we would

like to be able to allow the learning algorithm to tune itself online. For that reason, our approach should work with less powerful hardware, ideally working on what a desktop computer or a modest server can offer.

(G3) *Our approach should work with existing algorithms:* There are many clever learning algorithms available in the literature. Our approach should be able to make use of those algorithms for faster implementation, and convergence.

In handling the challenges of scheduling jobs while adhering to these guidelines, this dissertation presents a solution that follows software engineering best practices, such as reuse, by proposing formulations that follow an Application Programming Interface (API) specification for RL environments, satisfying (G1) and (G3). We exploit the structure of event-based systems and construct a set of features for learning with relatively small neural networks, satisfying (G2). We do so incrementally, and show that, at each of these changes, we reduce the amount of computation needed for learning.

1.3 Contributions

Current approaches that attempt to solve HPC job scheduling with RL seem to use simplified environments [Mao et al., 2016], or to pre-process real workload traces by sorting, filtering, and selecting subsets to enable agents to learn [Zhang et al., 2019]. Moreover, such approaches tend to use custom-built simulation environments, which make it difficult to (I) reproduce previous work and (II) build upon previous work by applying novel techniques from the RL community.

Our approach reduces the need for computational resources for learning, while also removing the need for preprocessing workload traces, while being competitive with state-of-the-art algorithms. We are able to do so by proposing reduced reward functions, reduced state representations which capture the essence of cluster state, and using event-driven systems.

Our contributions are summarized as follows:

- The formulation of new MDPs to model job scheduling as a learning problem;
- An analysis of the impact on learning performance of MDP design decisions;
- A better understanding of the impact of uncertainty in job run time estimates for learning agents;
- Strong scheduling performance without preprocessing on workload traces while delivering scheduling performance on par with custom algorithms from the literature

- Software contributions, including a downloadable OpenAI Gym environment and workload generation wrappers for the Python language;

The following publications are a direct result of this dissertation:

- de Freitas Cunha and Chaimowicz [2020], where we present our ideas for designing our environment, and show algorithms from the literature can learn in our environment faster than our own implementations of different algorithms;
- de Freitas Cunha and Chaimowicz [2021], where we present multiple Markov Decision Process (MDP) formulations and observe the impact these formulations have in learning and computing performance; and
- de Freitas Cunha and Chaimowicz [2022], where we further improved our formulation, allowing our agents to solve learning for clusters with hundreds of thousands of processors while showing reduced need for computational resources.

1.4 Notation

We strive to use consistent notation throughout this document. In this dissertation, we will use capital letters to represent random variables, while we reserve lower case letters to represent scalars, functions, and samples from random variables. Table 1.1 summarizes the notation we use with the meaning for most common symbols. In general, we represent sets with a calligraphic style. So, for example, the set $\mathcal{E}$ of all even prime numbers is defined as $\mathcal{E} = \{2\}$.

1.5 Document layout

This rest of this document is structured as follows: In Chapter 2 we introduce the background needed to completely understand this dissertation. In Chapter 3 we position this dissertation within the broader literature. In Chapter 4 we describe the simulation framework and how it was tested. In Chapter 5 we present experiments, and in Chapter 6 we conclude this dissertation and present next steps.

Table 1.1: Summary of the notation used in this dissertation.

Symbol	Description
j_i	The i-th job in the system
$t_f(j)$	The finish time of job j
$t_s(j)$	The submission time of job j
$t_e(j)$	The execution time of job j
$t_w(j)$	The wait time of job j
$t'_e(j)$	Online approximation of $t_e(j)$
$t'_w(j)$	Online approximation of $t_j(j)$
$t_r(j)$	The requested time of job j
$q(j)$	The queue size in front of job j
$q_w(j)$	The work in the queue in front of job j
$p_f(j)$	The number of free processors at j's submission
$\mathcal{S}$	The set of states in an MDP
$\mathcal{A}$	The set of actions in an MDP
$\mathcal{R}$	The reward function in an MDP
$\mathcal{T}$	The transition function in an MDP
ρ	The set of initial states in an MDP
γ	Discount factor in an MDP
τ_i	A trajectory $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ in episode i
S_t	The state seen by an agent at time-step t
A_t	The action taken by an agent at time-step t
R_t	The reward received by an agent at time-step t
θ	The parameters of an approximation function
π_θ	A policy with parameters θ
$\phi_\theta(\tau)$	The probability of following τ with policy π_θ
$\sigma(\mathbf{z})_i$	The i-th element of the softmax of vector $\mathbf{z}$
$v_\pi(s)$	Value function for state s under policy π
$q_\pi(s,a)$	State-action value function for state s and action a under policy π
$q_\pi(s,\omega)$	State-option value function for state s and option ω under policy π

Chapter 2

Background

Before diving into the main material, in this chapter we introduce the key techniques used in this dissertation. We begin by discussing concepts from batch job scheduling, such as metrics, basic algorithms, and job types (§ 2.1). Then, we present the RL formalism, while mapping a batch job scheduling example to the RL formalism for better understanding (§ 2.2). After this scenario is presented, we present the job generation process from workload models (§ 2.3), and discuss uncertainty models for batch job scheduling problems (§ 2.4). Then, we return to the RL formalism, now focusing on an algorithm for solving it (§ 2.5), and extending the RL problem for temporal abstraction (§ 2.6). Finally, we briefly introduce a high-level library used for implementing solutions to RL problems (§ 2.7).

2.1 Batch job scheduling

Job schedulers are used to manage the job queue and coordinate execution of jobs in supercomputers and HPC clusters. Their primary goal is to enable successful execution of computing jobs on parallel machines, with the main problem being that of matching jobs to resources in an efficient way. They guarantee jobs execute when requested resources are available and, usually, guarantee there won't be oversubscription of resources[1]. Given this main goal, there are many secondary goals schedulers may want to achieve, depending on whether the institution that owns the cluster prefers to satisfy the needs of individuals, or of the whole group of users of the system. Therefore, when centered on the user, schedulers may want to optimize response time, whereas when centered on the system, or on the whole group of users, they may want to optimize for utilization, or throughput [Feitelson and Rudolph, 1996].

When optimization of response time is a subgoal, it is usually modeled as the minimization of the average response time, with response time used as a synonym to turnaround

[1] Some schedulers allow for oversubscription of memory resources in their default configuration. The point being that jobs don't necessarily use peak memory during their complete lifetimes. In this document, though, we will consider memory cannot be oversubscribed, and will also assume jobs use all requested memory.

time: the difference between the time a job was submitted to the time it *completed* execution. A metric commonly used to evaluate this is the *slowdown* of a job, which, for job j is defined as

$$\begin{aligned} \text{slowdown}(j) &= \frac{(t_f(j) - t_s(j))}{t_e(j)} = \frac{t_w(j) + t_e(j)}{t_e(j)} \\ &\approx \frac{1}{t'_e(j)} \left(\sum_{i=1}^{t'_w(j)} 1 + \sum_{i=1}^{t'_e(j)} 1 \right), \end{aligned} \tag{2.1}$$

where $t_s(j)$ is the time job j was submitted, $t_e(j)$ is the time it took to execute job j, and $t_f(j)$ is the finish time of job j. The equality in the middle holds because the wait time, t_w, of a job j is defined as $t_w(j) = t_f(j) - (t_e(j) + t_s(j))$. The form of slowdown presented in equation (2.1) is useful in our context because we can compute an approximation of slowdown online, as the job is still in the system. As soon as the job finishes execution, equation (2.1) converges to the actual slowdown.

In order to maximize return on investment, institutions may want to maximize utilization (the number of resources in use divided by the number of available resources) of their clusters, since the more usage the cluster sees, the less resources are "wasted". Unfortunately, though, as system utilization increases, the load on the system also increases, potentially increasing response time as well. Additionally, solely optimizing for utilization may cause starvation when there are many jobs that use a large number of processors, but few jobs that use few processors. In such a case, the small jobs may wait indefinitely to run.

A metric similar to utilization, in the sense that it depends on the load of the system, is throughput, defined as the number of jobs completed per unit of time. Contrary to utilization, maximizing throughput will have the effect of prioritizing small jobs, and may starve bigger jobs.

Sometimes it is useful to compute the total execution time of the schedule, its makespan. Makespan is defined as the maximum finish time for all jobs in the system:

$$\text{makespan} = \max_{j \in \mathcal{J}} t_f(j), \tag{2.2}$$

where $\mathcal{J}$ is the set of all jobs that completed execution at a given time, and $t_f(j)$ is the finish time of job j. When there are no completed jobs in the system, we define makespan to ∞.

From the above discussion, we see that although simple metrics are useful, they may have side effects of starving certain job classes, or attributing greater effect to relatively small differences. As mentioned, maximizing utilization may not schedule jobs that cause fragmentation, while maximizing throughput may not schedule large jobs, and minimizing response time may give larger emphasis on small differences between jobs, especially in smaller ones. For this reason, designing scheduling algorithms for systems with heterogeneous workloads is a challenge which may be solved with learning algorithms.

An issue with using slowdown is that the metric is sensitive to small jobs: since small jobs will have smaller $t_e(j)$, any delays in scheduling them will increase their slowdown. Humans don't tend to notice small delays in small jobs, and thus, an alternative metric that is not so sensitive to small jobs is the bounded slowdown, defined as

$$\text{bsld}(j) = \max\left(1, \frac{t_f(j) - t_s(j)}{\max(\epsilon, t_e(j))}\right), \tag{2.3}$$

where ϵ is a configurable parameter, set to the smallest time to consider when computing (bounded) slowdown. In practice, and in this dissertation, ϵ is usually set to 10. For a study of how different values influence bounded slowdown, we direct the reader to Feitelson [2001].

2.1.1 Scheduling decisions and their impacts

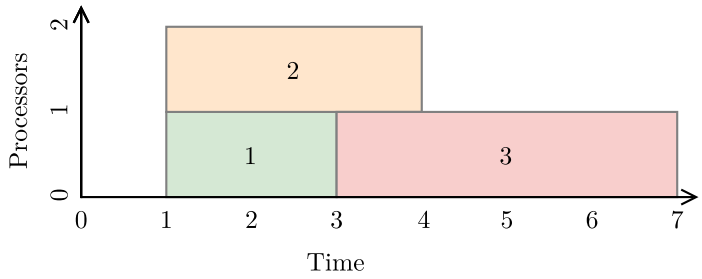


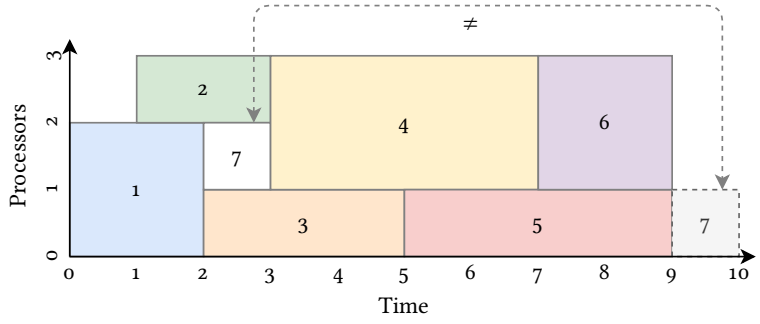
Figure 2.1: A possible schedule when three jobs arrive in a scheduling system at discrete time step 1 and no more jobs are submitted to the system at least until time step 7, the last one shown in the figure.

Consider the case of three batch jobs, j_1 = , j_2 = , and j_3 = , submitted to a scheduling system with two processors, and that the three jobs were submitted "between" time step 0 and 1, such that, when transitioning from the first time step to the second, now there are three jobs waiting. Also consider that, for these jobs, the generated schedule is the one displayed in Figure 2.1. As shown in the figure, the jobs execute for two, three and four time steps respectively, and all of them use a single processor.

In this dissertation, we will focus our discussion on what happens when a scheduling system based on RL tries to minimize the average slowdown. The reader should observe that different schedules can yield substantially different values of average slowdown (with average slowdown defined as the average of the slowdown for all jobs in the system). For example, the schedule shown in Figure 2.1 has an average slowdown equal to $\frac{1}{3}\sum_{i=1}^{3} \text{slowdown}(j_i) = \frac{1}{3}(1 + 1 + \frac{3}{2}) = \frac{7}{6}$, whereas, if we swapped j_3 with j_1, and started j_1 soon after j_2 finished, the slowdown would be $\frac{1}{3}(\frac{3+2}{2} + 1 + 1) = \frac{9}{6} = \frac{3}{2}$, a $\approx 29\%$ increase. Therefore, a scheduler should choose job sequences wisely, otherwise its performance can be degraded.

2.1.2 Base scheduling algorithms

The most basic scheduling algorithm in use is First-Come First-Serve (FCFS), also known as First In First Out (FIFO). In this algorithm, jobs are kept in a queue sorted by order of arrival. Whenever the first job in the queue fits in the system, it is allocated the resources it needs. The problem with this approach is that whenever a large job is at the head of the queue, smaller jobs are delayed, since scheduling them would violate the FCFS nature of the queue. Due to that, backfilling algorithms were implemented extensively. Figure 2.2 shows a distinction between FCFS and a backfilling algorithm. In the figure, basic FCFS would schedule job 7 at time unit 9, while a conservative backfilling algorithm would schedule it at time unit 2. Also seen in Figure 2.2 are the slowdowns computed for all jobs in the system. As shown in the figure, backfilling can have great impact on the slowdown of smaller jobs. The makespan of the scheduling shown in Fig. 2.2 is 9 when using backfilling, and 10 when using FCFS.



Job	Submission time	Duration	Processors	Slowdown	
				FCFS	Backfilling
1	0	2	2	1	1
2	1	2	1	1	1
3	1	3	1	1.3	1.3
4	1	4	2	1.5	1.5
5	1	4	1	2	2
6	1	2	2	4	4
7	2	1	1	**8**	**1**

Figure 2.2: Expected scheduling behavior of FCFS and conservative backfilling algorithms for a set of seven jobs submitted to a system with three processors. Job 7 appears twice to indicate the difference between the two algorithms. In this context, time is discrete. Therefore, jobs can only run for integral time as well. In a pure FCFS algorithm, job 7 would be scheduled at time unit 9, since it was the last submitted job to the system. In a Backfilling algorithm, job 7 would be scheduled at time 2, since its scheduling would not delay any other jobs. The bottom part of the figure shows statistics about the jobs, with slowdown computed after completion of all jobs.

Backfilling algorithms tend to come in at least two flavors: EASY (from the Extensible Argonne Scheduling sYstem) and conservative backfilling [Tsafrir and Feitelson, 2006]. The key idea of backfilling algorithms is that, whenever there is a new event that triggers a change

in the state of the system (such as new jobs arriving or finishing execution, for example), the system scans the queue in the base underlying algorithm order (FCFS in this section), scheduling jobs as the algorithm would, as long as the jobs fit in the system. Upon reaching the first job that cannot be scheduled, the system makes a *reservation* for that job. This reservation is an upper bound on the time the job would start: if all preceding jobs run to completion, this new job will start running at the reserved time. If earlier jobs fail or finish early, this job may start earlier. After this reservation is made, the scheduler keeps scanning the queue. All jobs that fit the system and that would finish prior to the reservation can be scheduled without delaying the reserved job. EASY backfilling makes a reservation for the first "big" job only, while conservative backfilling makes reservations for all skipped jobs.

Another classical algorithm that also has backfilling variants is the Shortest Job First (SJF) algorithm [Srinivasan et al., 2002], in which the job queue is kept sorted by requested job time, with smaller jobs first in the queue.

For the backfilling procedure to work, algorithms need an estimate of how long a job will execute. Therefore, backfilling systems tend to ask users for run time estimates of their jobs. If users over-estimate job run times, execution of later jobs advance, but if users under-estimate job run times, the system tends to kill such jobs, otherwise reservation times would be violated. Therefore, users tend to over estimate their job run times to avoid having their jobs terminated. There is work in the literature that analyses user estimates and their correctness [Lee et al., 2004]. Other systems tend to use a prediction in the absence (or despite) of runtime estimates, which is the case of the EASY++ algorithm [Tsafrir et al., 2007].

2.1.3 Job types in workloads

The most basic type of job that may be submitted to a scheduling system are rigid jobs, which are submitted with a fixed number of required processors set by the user. This is not the only type of job we may encounter, though, and a summary of job types that may be found in workloads is shown in Figure 2.3. In the following paragraphs we describe the other types of jobs for completeness.

Rigid A rigid job is a job for a program that is written to execute on a specific number of processors and, even if more processors are available, they cannot take advantage of those processors. Similarly, if fewer processors are available, the job will fail to run.

Moldable A moldable job is a job that allows for flexibility in the number of required processors. But, once the number of required processors is set, the job runs from start to finish with

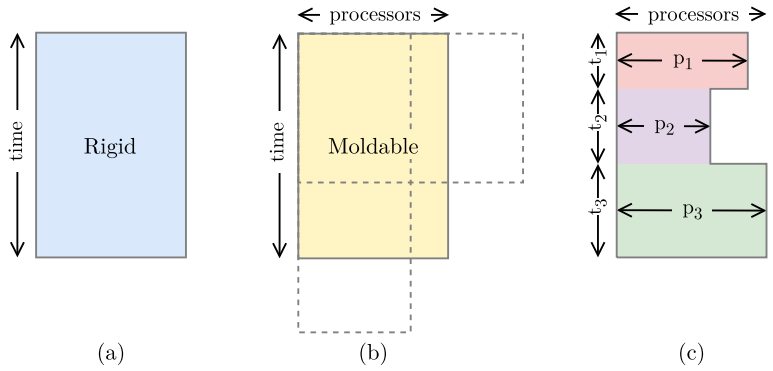


Figure 2.3: Major types of jobs found in workloads: (a) shows rigid jobs, which define a fixed rectangle in processor × space, (b) shows moldable jobs (jobs in which the scheduler may choose the number of processors from a set of possibilities), and (c) shows evolving and malleable jobs (jobs which may change their number of processors as time passes.)

the same number of processors. The number of processors used by moldable jobs is defined at job submission time.

Evolving Evolving jobs are jobs of applications that may change their resource requirements during execution, with such a change being initiated by the application itself. These changes in resource requirements are a consequence of many applications having different phases, such as having waves of data input and output, followed by parallel computation.

Malleable In contrast to evolving jobs, malleable jobs (or jobs that support dynamic partitioning) adapt to changes in the number of processors during execution (with the change initiated by the system). This is the most flexible type of jobs that can be submitted to a system, but with this flexibility comes complexity, since jobs must be robust to the removal of processors that may hold shared data other processors may depend on.

In this dissertation, we will center our discussion on rigid and moldable jobs: in both cases, the number of processors required by a job is determined before a job enters the queue. Therefore, the number of processors required by the jobs we consider here will *not* change at run time. Hence, considering the number of processors required by a job in one axis, and the amount of time required by a job in another axis, the jobs considered here define a rectangle in the time × processors space, as exemplified by Figure 2.3. To the area of the time × processors rectangle we give the name of *work*. For example, a job that requests 3 processors for 2 time steps is expected to take 6 units of work. In this dissertation, the actual amount of work taken by a job is only known after if finishes. Before that happens, our methods only used the *requested* amount of work, which serves as an upper bound for *actual* work: in our model, schedulers never give more time, nor more processors than those requested by a job.

2.2 Deep Reinforcement Learning and Job Scheduling

In a Reinforcement Learning (RL) problem, an agent interacts with an unknown environment in which it attempts to optimize a reward signal by sequentially observing the environment's state and taking actions according to its perception. For each action, the agent receives a reward. Thus, in the end, we want to find the sequence of actions that maximizes the total reward, as we will detail in the next paragraphs.

RL formalizes the problem as a Markov Decision Process (MDP) represented by a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \rho, \gamma \rangle$[2]. At each discrete time step t the agent is in state $S_t \in \mathcal{S}$. From S_t, the agent takes an action $A_t \in \mathcal{A}$, receives reward R_{t+1} from $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and ends up in state $S_{t+1} \in \mathcal{S}$. Therefore, when we assume the first time step is 0, the interaction between agent and environment creates a sequence $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ of states, actions and rewards. To a specific sequence $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ of states, actions, and rewards we give the name of trajectory, and will denote such sequences by τ. The transition from state S_t to S_{t+1} follows the probability distribution defined by $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ or, in an equivalent way, $\mathcal{T}$ gives the probability of reaching any new state s' when taking action a when in state s: $p(s'|s,a) = p(S_{t+1} = s'|S_t = s, A_t = a)$. ρ is a distribution of initial states, and γ is a parameter $0 \leq \gamma \leq 1$, called the discount rate. The discount rate models the present value of future rewards. For example, a reward received k steps in the future is worth only γ^k now. This discount factor is added due to the uncertainty in receiving rewards and is useful for modeling stochastic environments. In such cases, there is no guarantee an anticipated reward will actually be received and the discount rate models this uncertainty.

To map our presentation of RL into our problem of job scheduling, we consider $\rho(\llcorner\!\!\lrcorner) = 1$ (the only possible initial state is the empty cluster), with the first state consisting of the empty cluster, with no jobs in the system, $S_0 = \langle \llcorner\!\!\lrcorner \rangle$ and $A_0 = \emptyset$, since there is no job to schedule. We also consider an episodic setting, with an episode consisting of the submission and scheduling of a set of jobs. For example, if we consider 256 jobs in an episode, the episode starts with the empty cluster, has 256 jobs submitted according to a workload model, and finishes once the 256-th job is scheduled. Every time an episode finishes, the state of the system is reverted to the empty cluster state.

In this dissertation, we allow the agent to choose the next job, and so the agent is learning a scheduling policy. In our example, one can obtain a reward function by using the sequential version of slowdown, shown in the rightmost equality of (2.1), such that the reward at each time step is given by the sum of the current slowdown for all jobs in the system: $\mathcal{R} = -\sum_{j \in \mathcal{J}} {}^1/_{t_c(j)}$. When the reward function is such that it computes the online version of slowdown for *all jobs in the system*, if $A_1 = \emptyset$, $R_2 = {}^1/_2 + {}^1/_3 + {}^1/_4$. Moreover, if

[2]Some authors leave the γ component out of the definition of the MDP. Leaving it in the definition yields a more general formulation, since it allows us to model continuous (non-episodic) learning settings.

jobs j_1, j_2, and j_3 are chosen in sequence, the next state, shown in Fig. 2.1, will be given by sequentially applying the transition function $\mathcal{T}$ as $\mathcal{T}(\,,\,)\mathcal{T}(\,,\,)\mathcal{T}(\,,\,)$. If the episode finished immediately after the state shown in Fig. 2.1, the trajectory τ_1 would be given by $\tau_1 = \langle S_0 = \,, A_0 = \,, R_1 = 0, S_1 = \,, A_1 = \,, R_2 = 0, S_2 = \,, \ldots\rangle$[3].

The reward signal encodes all of the agent's goals and purposes, and the agent's sole objective is to find a policy π_θ parameterized by θ that maximizes the expected return

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T \\ &= \sum_{t=0}^{T-1} \gamma^t R(S_t, A_t \sim \pi_\theta(S_t)), \end{aligned} \tag{2.4}$$

which is the sum of discounted rewards encountered by the agent. Note that for this summation to converge, either $T < \infty$, or $\gamma < 1$. In this dissertation, we will use time-bounded episodes, such that $T < \infty$. $\pi_\theta(S_t)$ is a function that, given a state, returns an action for the agent to take. As a shorthand notation, we also define $G(\tau) = G_0$, meaning that the expected return of a trajectory is the expected return of all rewards in that trajectory, starting from state S_0. In our example, a deterministic policy that implemented the SJF algorithm would choose the shortest job $\pi_\theta(\langle\,,\,,\,,\,\rangle) = \,$, while a stochastic policy would assign a probability to each job, and either choose the one with highest probability or sample from the jobs according to that distribution. In our example, for each job j_i in time step 1, π would give the probabilities of choosing each job given an empty cluster: such that, by total probability, $\pi(\,|\,) + \pi(\,|\,) + \pi(\,|\,) = 1$. In practice, when neural networks are used for approximation, the last layer of the neural network is usually a softmax, or soft-argmax, function, defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{2.5}$$

for $1 \leq i \leq K$, and $\mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$, where $\mathbf{z}$ is the output vector of the neural network, and K is the number of classes to choose from (for example, when choosing jobs, K may represent the size of the vector with jobs to choose from). Since the softmax is normalized by the sum of the exponential of all individual components, the sum of the elements of $\sigma(\mathbf{z})$ equals 1, with each individual element $0 \leq \sigma(\mathbf{z})_i \leq 1$. This allows us to interpret the softmax as a probability mass function, and when K equals the number of actions available to an agent, each element of the softmax $\sigma(\mathbf{z})_i$ can be interpreted as the probability of taking the i-th action available. Apart from neural networks [Silver et al., 2018, Tesauro, 1994], another popular type of function approximation are linear [Liang et al., 2016] or polynomial combinations [see Sutton and Barto, 2018a, Chapter 9] of features.

In Figure 2.4, we show the agent-environment interaction loop. As represented by the thickness of the arrows between agent and environment, the agent communicates a scalar

[3]The value shown for R_2 might contradict the previous discussion, but the MDP is set in a way that, *when jobs are scheduled successfully*, $R_{t+1} = 0$.

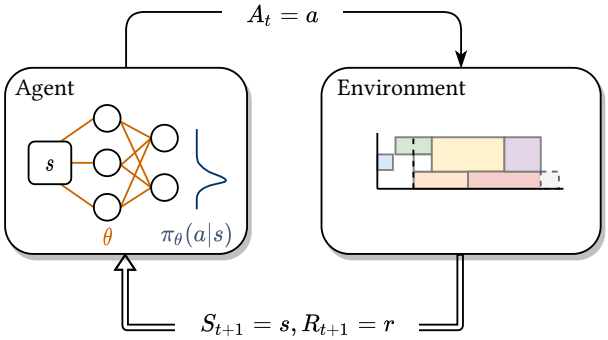


Figure 2.4: Agent-Environment interaction loop.

action A_t (represented by a thin line) and, in turn, receives a state (which may be a scalar, vector, matrix, or, more generally, a tensor), and a scalar reward signal, both represented by a thick line. The figure also shows a component present in modern agents: a function approximator that implements policy π_θ, in which the state is processed and yields a probability distribution for the set of actions it should take at a given time step.

In the following two sections, we describe models of job arrival in the system and then we describe the formalism to solve the RL problem described in this section.

2.3 Workload models

So far, we've discussed the state transition function $\mathcal{T}$, but we still haven't mentioned how jobs *arrive* in the system. This is the role fulfilled by workload models. These types of models not only determine the "shape" of jobs (how many resources they need, and how long they take to run), but also the time at which they arrive in the system. Here we consider two synthetic workload models: the Mao model [Mao et al., 2016] (§ 2.3.1), and the Lublin model [Lublin and Feitelson, 2003] (§ 2.3.2).

2.3.1 The Mao model

In the Mao model, adapted from the model proposed by Mao et al. [2016], job length, job arrival times and job sizes are sampled independently. The Mao model assumes that, for each simulation time step, a job has probability p of arriving in the system. If a job arrives, to determine its length, the workload model decides with probability l whether j is a long job or

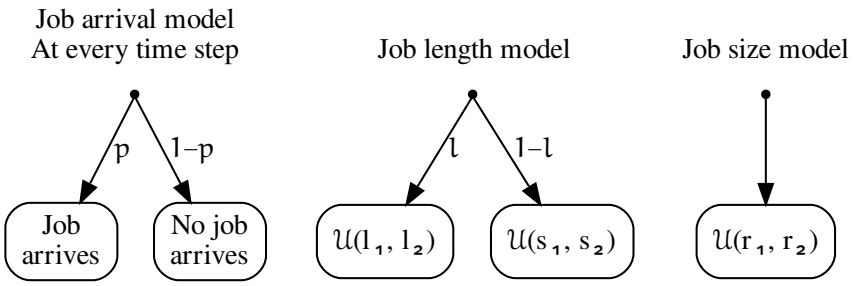


Figure 2.5: Graphical representation of the workload model based on the one proposed by Mao et al. [2016]. Job sizes, job arrivals, and job durations are sampled independently.

a short job. Long jobs are sampled from $\mathcal{U}(l_1, l_2)$ and small jobs are sampled from $\mathcal{U}(s_1, s_2)$, where $\mathcal{U}(a, b)$ is the discrete uniform distribution between a and b, with $l_1 > s_1$ and $l_2 > s_2$. The job's size is sampled from $\mathcal{U}(r_1, r_2)$. Figure 2.5 shows a graphical representation of this model.

Although this model is useful for testing smaller problems and for verifying whether an agent is learning, this is an unrealistic model which relies on time-based simulations due to the probability p of sampling a job depending on the current time step. A consequence of using this model is that it becomes very hard to tune, generating either too many jobs, or too few jobs, once the number of time steps in a simulation increases.

2.3.2 The Lublin model

The Lublin model is a statistical model that attempts to replicate the workloads of real supercomputers. It was fitted to traces of large supercomputing sites, and has three major components: a model of job sizes, a model of job run times, and a model of job arrival times. In the Lublin model, job arrivals are sampled from two Gamma distributions, with one being used for the peak inter arrival times, and another for the daily inter arrival times. Once a job is determined to arrive, modeling of the job itself follows the algorithm depicted in Figure 2.6. The Lublin model is hierarchical, and selects job sizes from a two-stage uniform distribution with four parameters: the minimum and maximum job sizes desired, and the fractions of serial (p_1 in the figure) and power-of-two (p_2 in the figure) jobs[4]. After a job size is selected, job run times are sampled from a hyper-Gamma distribution ("Run time selection" in the figure) which depends on the job size sampled previously. Lublin and Feitelson [2003] published the original workload model implementation as a C++ source file, and we use a Python wrapper for the C code from the `parallelworkloads` library[5].

[4]Jobs with sizes that are a power of two are explicitly modeled because such powers are common in real clusters, even in clusters whose network topology does not require power-of-two sizes.

[5]`https://github.com/renatolfc/parallelworkloads`, DOI: 10.5281/zenodo.7068617.

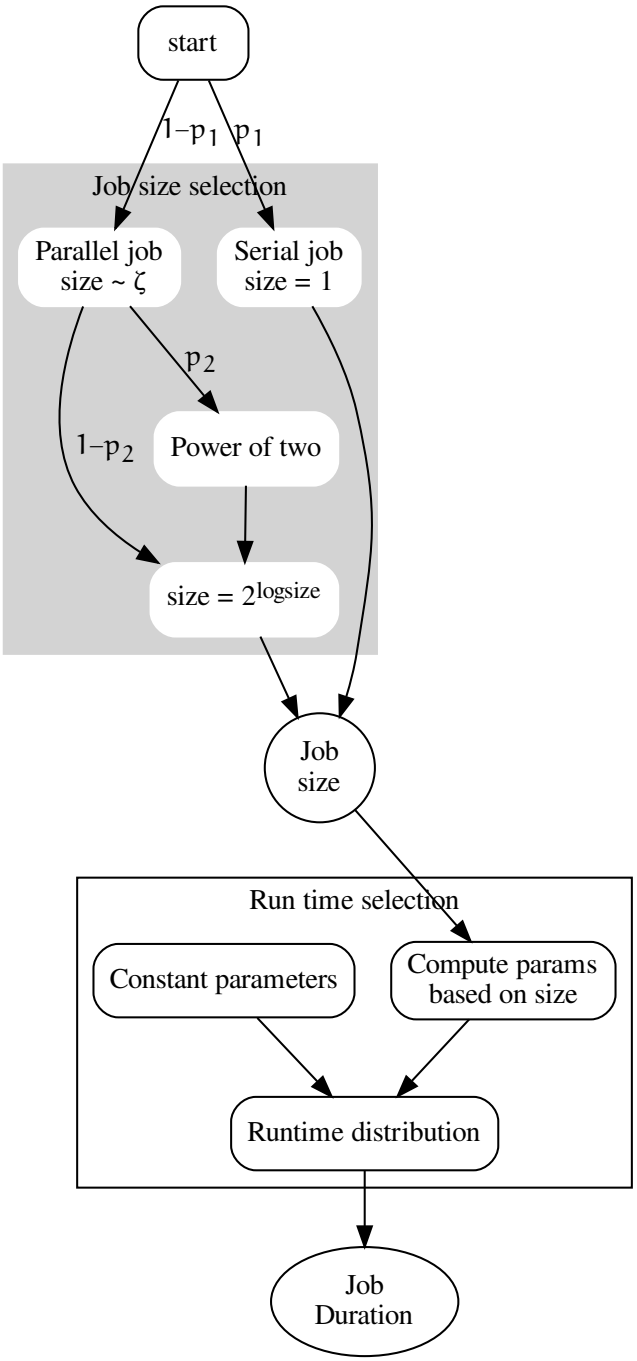


Figure 2.6: Graphical representation of the workload model proposed by Lublin and Feitelson [2003]. Job sizes are sampled from a distribution and then are fed to another distribution to sample job length. Outputs are the job size and job duration. These job characteristics are then combined with the arrival model to determine the job start time.

2.4 Uncertainty in job run time estimates

In our discussion so far, we assumed the run time of jobs are known by the scheduler. Various models in the literature [Domeniconi et al., 2019, Mao et al., 2016, Zhang et al., 2020] assume the availability of run time estimates, yet, real-world average job run time estimates, even when available, are inaccurate [Lee et al., 2004]. For this reason, even though some scheduling algorithms designed with accurate run time estimates in mind may perform well with noisy job run time estimates [Mu'alem and Feitelson, 2001, Zotkin and Keleher, 1999], others will suffer significant performance degradation [Chiang et al., 2002]. Additionally, some systems use ML techniques to infer job resource and time requirements [Cunha et al., 2017, Rodrigues et al., 2016], and those predictions will not be perfect either. In this section, we present two models that can be used to generate noisy run time estimates from jobs sampled by the workload models discussed in the previous section.

2.4.1 The Gaussian model

In the Gaussian model, job run time estimates are sampled from a Gaussian distribution centered at the actual job run time, with a configurable standard deviation scaling parameter. The model supports three variations: one-sided overestimation, one-sided underestimation, and two-sided Gaussian. Usually, it doesn't make sense to generate consistently underestimated times, as jobs that exceed their run times are usually terminated by schedulers. Still, we wanted to observe the impact of such changes on learning algorithms, and left that option in the model. Formally, in this model, the estimate for job j is its actual execution time, $t_e(j)$ plus a difference, sampled from

$$\text{diff}(j) = \begin{cases} -|\mathcal{N}(0, \beta(j))|, & \text{if underestimated} \\ |\mathcal{N}(0, \beta(j))|, & \text{if overestimated} \\ \mathcal{N}(0, \beta(j)), & \text{otherwise,} \end{cases} \tag{2.6}$$

where $\mathcal{N}$ is the Gaussian distribution and $\beta(j) = vt_e(j)$ is a scaling parameter, controlled by parameter v and the actual execution time of the model. For cases in which the run time estimate is smaller than 1 second, we set them to 1.

2.4.2 The Tsafrir model

User run time estimates tend to be modal due to users preferring to repeat time estimates. A more appropriate run time estimate model might be the one proposed by Tsafrir et al. [2005], which was built with the modal nature of job run time estimates in mind, and which uses a histogram of the twenty most popular estimate values, as these tend to account for 90% of job run time estimates in production machines. In this dissertation, we used a Python wrapper from the `parallelworkloads` library, which wraps the original C++ code available from the parallel workloads archive.

2.5 Policies and approximators

Now that we know how jobs arrive in the system and how they can be selected for scheduling, we turn to the formalism to solve the problem in an RL setting. Given that agents have partial control over received rewards by being able to choose an action and ending up in a new state, it is natural that agents will choose actions that maximize G_t (2.4).

In practical problems, which tend to have large state spaces, it is hard to learn a direct mapping from states to actions. Because of this, we turn to parameterized functions that allow us to approximate $\pi(a|s)$ by a function $\pi_\theta(a|s)$ with tunable parameters θ. Approximation not only allows us to represent large spaces with a relatively small number of parameters, but it also enables generalization of models, at the cost of using a sub-optimal objective function. Popular function approximators include linear combinations of features [Liang et al., 2016] and neural networks [Silver et al., 2018, Tesauro, 1994].

If we have a policy π, we are able to determine its value by letting the agent follow it or, more formally, for MDPs:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right],$$

where $\mathbb{E}_\pi[]$ denotes the expected value given that the agent starts by taking action a while in state s and subsequently following policy π, starting at time t. Methods that try to learn and optimize $q_\pi(s,a)$ are called value-based methods, while methods that only use the policy π directly are called policy-based methods.

From $q_\pi(s,a)$, we can also define the value of a state as the weighted average of taking each action a when in state s. Since the probability of taking action a is given by $\pi(a|s)$, we

have

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a), \tag{2.7}$$

the value function of state s.

2.5.1 Policy gradients

In this section we present the main optimization method we use to find policies: policy gradients. As implied by the name, we compute gradients of policy approximations, and use them to find better parameters for those functions.

Formally, we generalize policies to define distributions over trajectories with

$$\phi_\theta(\tau) = \rho(S_0) \prod_t \pi_\theta(A_t|S_t) \underbrace{\mathcal{T}(S_{t+1}|S_t, A_t)}_{\text{Environment}}, \tag{2.8}$$

in which π_θ is being optimized by the agent, and ρ and $\mathcal{T}$ are provided by the environment. What (2.8) says is that we can assign probabilities to any trajectory, since we know the distribution of initial states ρ, and we know that the policy will assign probabilities to actions given states, and that, when such actions are taken, the environment will sample a new state for the agent. Clearly, we need a model of the environment $\mathcal{T}$ to be able to evaluate (2.8). Assuming we have it, we can build a performance function

$$J(\theta) = \mathbb{E}\left[v_\pi(s_0), s_0 \sim \rho(S_0)\right] = \int_\tau G(\tau)\phi_\theta(\tau) d\tau$$

that gives the value of using parameterization θ over *all trajectories*. Unfortunately, even if we did have access to the model $\mathcal{T}$, the integral in $J(\theta)$ would be intractable. Still, we can approximate $J(\theta)$ with Monte Carlo estimation, such that

$$\mathbb{E}\left[J(\theta)\right] \approx \widehat{J(\theta)} = \frac{1}{N} \sum_{i=1}^{N} G(\tau_i)\phi_\theta(\tau_i).$$

Therefore, if π_θ is differentiable, ϕ_θ is differentiable, and we can find the optimal set of parameters θ iteratively with gradient ascent, such that $\theta_{j+1} \leftarrow \theta_j + \alpha \nabla \widehat{J(\theta)}$, with $\alpha > 0$, j representing the iteration number, and θ_0 being a random initialization of parameters θ.

When using a neural network, we can use an automatic differentiation system [Baydin et al., 2018] to perform this approximation by sampling trajectories from an environment in which we use the neural network for decision-making, and perform back propagation from the returns of episodes (Section 4.5). From the Policy Gradient theorem [Sutton et al., 2000],

the gradient of $J(\theta)$ is

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla_\theta \pi_\theta(a|S_t) \right] \tag{2.9}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \nabla_\theta \ln \pi_\theta(A_t|S_t) \right] \tag{2.10}$$

$$= \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \nabla_\theta \ln \pi_\theta(A_t|S_t) \right]. \tag{2.11}$$

Combining this with the previous gradient ascent update, we get the REINFORCE update:

$$\theta_{j+1} = \theta_j + \alpha G_t \nabla_\theta \ln \pi_\theta(A_j|S_j), \tag{2.12}$$

with G_t as defined in equation (2.4). An issue with Monte-Carlo methods in general, and with the update (2.12) in particular, is that they tend to have high variance, and require a large number of samples for an accurate estimate of $\nabla_\theta J(\theta)$[6]. One way to reduce the variance *without* drawing multiple samples is to introduce a state-dependent baseline function[7] $b(S_t)$. This baseline function can be any function, as long as it does not depend on the action A_t [8]. With a baseline, we can change the update 2.12 to

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t)) \nabla_\theta \ln \pi_\theta(A_t|S_t). \tag{2.13}$$

Although we've added a new term to the update (2.12), this new term doesn't affect its expected value. To see why, let us expand (2.9) by including the baseline component: $\nabla_\theta J(\theta) = \mathbb{E}_{a\sim\pi} \left[(q_\pi(S_t, a) - b(S_t)) \nabla_\theta \pi_\theta(a|S_t) \right]$. Then, by focusing on the component that contains the baseline, we have the expansion:

$$\begin{aligned}
\mathbb{E}_{a\sim\pi} \left[b(S_t) \nabla_\theta \ln \pi_\theta(a|S_t) \right] &= b(S_t) \mathbb{E}_{a\sim\pi} \left[\nabla_\theta \ln \pi_\theta(a|S_t) \right] \\
&= b(S_t) \sum_a \pi_\theta(a|S_t) \nabla_\theta \ln \pi_\theta(a|S_t) \\
&= b(S_t) \sum_a \cancel{\pi_\theta(a|S_t)} \frac{\nabla_\theta \pi_\theta(a|S_t)}{\cancel{\pi_\theta(a|S_t)}} \\
&= b(S_t) \sum_a \nabla_\theta \pi_\theta(a|S_t) \\
&= b(S_t) \nabla_\theta \sum_a \pi_\theta(a|S_t) \\
&= b(S_t) \nabla_\theta 1 = 0
\end{aligned}$$

[6] Due to the Central Limit Theorem, the standard deviation converges to zero at a rate of $1/\sqrt{n}$.

[7] This seems to be related to the Common Random Numbers technique from the Monte-Carlo literature [Glasserman and Yao, 1992], which states that, when computing the expected value of the difference of the function of two random variables ($\mathbb{E}[f(X) - f(Y)]$), the variance of the difference is $\mathbb{V}[f(X) - f(Y)] = \mathbb{V}[f(X)] + \mathbb{V}[f(Y)] - 2\text{Cov}[f(X), f(Y)]$. In this specific case, $\mathbb{E}[f(Y)] = 0$ is the baseline function, and we expect $2\text{Cov}[f(X), f(Y)] > \mathbb{V}[f(Y)]$, which reduces variance.

[8] Some authors [Thomas and Brunskill, 2017] argue that, even with action-dependent baselines, under certain conditions, such functions will be unbiased.

Due to their variance-reduction effects, recent Policy Gradient (PG) methods incorporate the concept of the baseline, which tends to be learned together with the parameters θ but, as we shall see in Chapter 4, this is not required, and there are successful algorithms in the literature that use simpler baselines.

When the baseline function is learned and used to approximate $\hat{v}_{\pi,\zeta}(s) \approx v_\pi(s)$ the value of state s with parameters ζ, it is possible to replace the full return G_t of the REINFORCE algorithm with the approximation $\hat{v}_{\pi,\zeta}$. From (2.13), approximating G_t by the one-step return $R_{t+1} + \gamma\hat{v}_{\pi,\zeta}(S_t)$ and assuming $b(S_t) = \hat{v}_{\pi,\zeta}(s)$, we get the update rule

$$\theta_{t+1} = \theta_t + \alpha(\underbrace{R_{t+1} + \gamma\hat{v}_{\pi,\zeta}(S_t) - \hat{v}_{\pi,\zeta}(S_t)}_{\text{Advantage function}})\nabla_\theta\pi_\theta(A_t|S_t). \tag{2.14}$$

When this is done, this update gives rise to Advantage Actor-Critic Algorithms, with the name deriving from usage of the advantage function, highlighted in (2.14).

2.5.2 Policy entropy as regularization

With the updates (2.13, 2.14), the agent may be encouraged to exploit knowledge too early in the learning process, overfitting to early positive returns. As with other ML techniques, one solution to reduce overfitting is by means of regularization. Since policy $\pi_\theta(\cdot)$ encodes a probability density function, one way to implement regularization in this case is to use the entropy of the distribution, which, for random variable X is defined as

$$h(X) = -\sum_{x\in X} p(x)\ln p(x). \tag{2.15}$$

To get an intuition of why introducing entropy helps in this case, let us consider the case of a Bernoulli random variable. In this case, $X \in \{0, 1\}$ and we can write $p(X = 1) = \theta$ and $p(X = 0) = 1 - \theta$ to represent the probabilities of successes and failures, respectively[9]. The entropy of this random variable, called the binary entropy function, is $h(X) = -[\theta\ln\theta + (1-\theta)\ln(1-\theta)]$, which, when plotted, gives us Figure 2.7. Now, observe that we get maximum entropy when $\theta = 0.5$, which corresponds to the uniform distribution (flipping a fair coin). Since we are adding the entropy to our gradient ascent objective, when considering two policies that yield the same performance, the optimization algorithm will prefer the one with higher entropy and, therefore, distributions that tend to be "more diverse", not concentrating density over a small range of values.

When entropy regularization is used, a deeper effect is observed as well. According to Williams and Peng [1991], with entropy regularization, agents tend to perform better in

[9]This arises from the definition of the Bernoulli distribution as $\text{Ber}(X = x|\theta) = \theta^x(1-\theta)^{(1-x)}$.

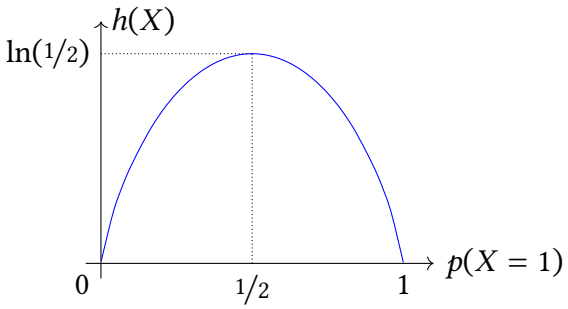


Figure 2.7: Entropy of a Bernoulli random variable as a function of parameter θ. As shown, the maximum is found when $\theta = 1/2$, which corresponds to the probability of flipping a fair coin.

tasks that have hierarchical characteristics. The reason being that, when sampled points yield high rewards, the network will tend to capture regularities in such points, biasing the policy to search for points that share features with the ones found, potentially increasing performance. When adding a regularization term, it is important to add a hyper parameter to govern the influence of the regularization term on the final update function. Combining equations (2.15) and (2.13) and adding hyper parameter β to govern regularization strength, we have the update

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t))\nabla_\theta \ln \pi_\theta(A_t|S_t) + \beta\nabla_\theta h(\pi_\theta(A_t|S_t)). \tag{2.16}$$

Similarly, the update (2.14) can be extended to regularize based on the entropy of the policy. In such a case, we have the Advantage Actor-Critic (A2C) algorithm's [Mnih et al., 2016] update rule:

$$\theta_{t+1} = \theta_t + \alpha\left[R_{t+1} + \gamma\hat{v}_{\pi,\zeta}(S_t) - \hat{v}_{\pi,\zeta}(S_t)\right]\nabla_\theta \ln \pi_\theta(A_t|S_t) + \beta\nabla_\theta h(\pi_\theta(A_t|S_t)). \tag{2.17}$$

In both equations (2.16, 2.17), we have the expansion of the gradient of the entropy as

$$\begin{aligned}
\nabla_\theta h(\pi_\theta(A_t|S_t)) &= \nabla_\theta\left[-\sum_{a\in\mathcal{A}_t}\pi_\theta(a|S_t)\ln\pi_\theta(a|S_t)\right] \\
&= -\sum_{a\in\mathcal{A}_t}\left[\frac{\pi_\theta(a|S_t)}{\pi_\theta(a|S_t)}^{1}\nabla_\theta\pi_\theta(a|S_t) + \ln(\pi_\theta(a|S_t))\nabla_\theta\pi_\theta(a|S_t)\right] \\
&= -\sum_{a\in\mathcal{A}_t}\left[1 + \ln\pi_\theta(a|S_t)\right]\nabla_\theta\pi_\theta(a|S_t).
\end{aligned}$$

As an example, when $\mathcal{A} = \{0, 1\}$, and we make $\pi_\theta(A_t|S_t) = \text{Ber}(A = a|\theta)$ equal to zero to find the maximum, $\nabla_\theta h(\pi_\theta(A_t|S_t)) = -\{[1+\ln(\theta)]\cdot 1 + [1+\ln(1-\theta)]\cdot -1\} = \ln(1-\theta) - \ln(\theta) = 0 \rightarrow \ln(\theta) = \ln(1-\theta) \rightarrow \theta = 1/2$, the same maximum we found graphically in Figure 2.7.

As with any regularization method, the β parameter requires tuning, as setting it too low would disable the regularization, and setting it too high would hinder learning.

2.5.3 Maskable PPO

Optimizing the policy gradient objective (2.12) is the main method we use for learning, but some improvements to (2.12) have been proposed over the years. One such improved method is the Proximal Policy Optimization (PPO) algorithm, whose main ideas are (I) limiting the updates of the policy to within a region near the current parameters, (II) using multiple agents for collecting trajectories, and (III) estimating the advantage function of each state to be used as a baseline for variance reduction 4.5. For this last idea, PPO uses two networks: a *policy* network, for predicting the next action, and a *value* network, for estimating the advantage of a state.

Especially in early parts of training, the learned policy will generate actions that are "invalid". The definition of what is or is not a valid action might vary, but as an example, when using a multi-layer perceptron, the neural network requires a representation of fixed size. In such a case, if jobs were chosen from a window over the waiting queue, and the queue was smaller than the window, there would be a non-zero probability of the agent choosing the position in the fixed representation corresponding to a job that does not exist. An approach for solving this problem is masking out invalid actions (jobs that do not exist), and sampling from the set of valid actions. This becomes especially useful as the number of actions available increase, speeding up learning by not letting the agent waste time sampling useless actions. For this reason, instead of using PPO directly, we use a *maskable* PPO implementation, which applies a *state-dependent differentiable function* during calculation of action probabilities [Huang and Ontañón, 2022].

2.6 Options as a closer-to-reality model

Both in state-driven simulations, and in actual job schedulers, job scheduling decisions don't necessarily take place at, say, *every second* or other fixed time intervals. Rather, scheduling decisions are made when external events happen, such as the arrival of a new job, the finishing of a running job, or a request from a user of the scheduler, such as changing the priority of a job, or requesting its termination, or when a timer expires.

In Section 2.2, we presented a way to model job scheduling as an MDP, but, when we consider the fact that decisions need only occur when events happen, the actions available to the agent are, in fact, *options*: temporal abstractions over actions. This means that, once a decision is made, the next state seen by the scheduler might not necessarily be of the next time

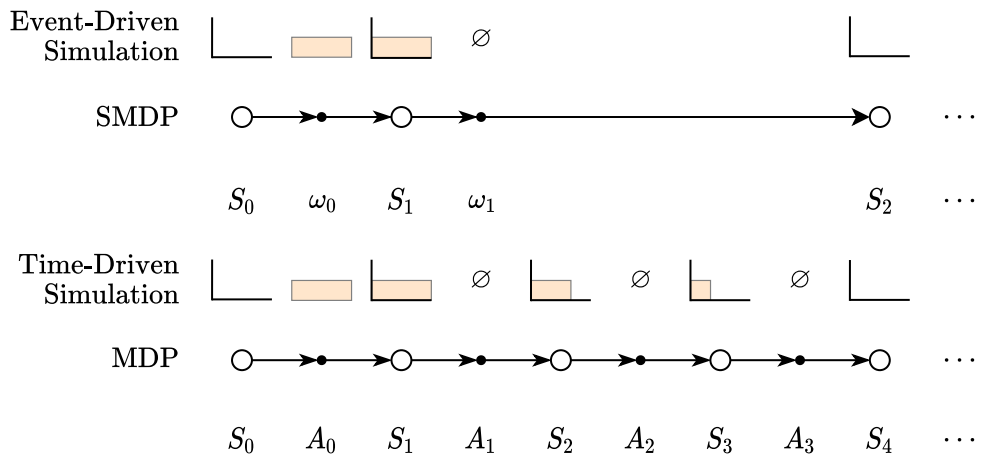


Figure 2.8: Relationship between an MDP and a Semi-MDP, and how an MDP is tied to a time-driven simulation, whereas an event-driven simulation is closer to an SMDP. The illustration shows an empty cluster ⌞ where the agent chooses a job ▭ in the first state. In the MDP, the agent sees all time steps in which the job ▭ is processed, while in the SMDP, the agent only sees the job to select, the event for the job starting, and the event for the job finishing. In both cases, state S_1 shows the state of the cluster immediately after the job ▭ starts running. Also, in both cases, no other job arrives after ▭. Notice how the SMDP allows temporal abstraction over the underlying MDP.

step, allowing the agent to see the next *event*. Moreover, any MDP with a decision process that only selects options over that MDP is an *SMDP* [see Sutton et al., 1999, Theorem 1]. Figure 2.8 gives an intuition of the differences between an MDP and an SMDP. In the example in the figure, one agent sees five states (MDP), while the other only sees three (SMDP). With real-life examples, this temporal abstraction makes for even greater differences.

Formally, an option $\omega = \langle \pi, \gamma_\omega \rangle$, where π is a policy and γ_ω is a state-dependent termination function, is a generalization of actions, and can take more than one time-step to execute. To execute option ω at time t, an agent chooses the action $A_t \sim \pi(\omega|S_t)$, with ω terminating at time $t+1$ with probability $1 - \gamma_\omega(S_{t+1})$, at $t+2$ with probability $1 - \gamma_\omega(S_{t+2})$, and so on, until termination (of the option.)

In our event-driven context, then, the probability $\gamma_\omega(S_{t+k})$ is 1 for all time steps k between choosing a job for execution, and the time at which the first event happens after time step t. In other words, let u be the time step of the *first* event that happens after choosing action A_t. Then, $\gamma_\omega(S_u) = 0$ and the option terminates with probability 1 at time step u and $\gamma_\omega(S_w) = 1$ for all $t \neq u$. As exemplified by Figure 2.8, in our model, when an option takes more than one time step to execute, intra-option actions are set to $\emptyset$. Since there are no decisions to be made, doing nothing is the only decision that makes sense. The advantage of using this model as opposed to an MDP is that the reinforcement signal needs only flow through actions that make a difference in agent performance, skipping "unneeded" actions, as in the bottom part of Figure 2.8.

The reward when using an option model is given by

$$r(s,\omega) = \mathbb{E}\left[\sum_{i=0}^{\kappa-1} \gamma^i R_{i+1} \mid S_t = s, A \sim \pi, \kappa \sim \gamma_\omega\right]. \tag{2.18}$$

In other words, the reward of an option is the sum of discounted rewards for all rewards received at intermediate time steps while the option was being executed. It is important to note that γ_ω affects option termination, while γ is used for discounting.

Similarly, the value of state s, $v(s)$, needs to take into account the multiple time-steps an option can take, and is defined as

$$v_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s)\left[r(s,\omega) + \sum_{s'} \mathcal{T}(s'|s,\omega) v_\pi(s')\right], \tag{2.19}$$

where $\mathcal{T}(s'|s,\omega) = \sum_{t=1}^{\infty} \gamma^t p(S_i = s', \kappa = t \mid S_t = s, A \sim \pi, \kappa \sim \gamma_\omega)$ is a discounted weighting of state, option pairs from (s,ω), and $\Omega(s)$ is the set of options available at state s. The value of executing option ω at state s is given by

$$q_\pi(s,\omega) = r(s,\omega) + \sum_{s'} \mathcal{T}(s'|s,\omega) v_\pi(s'). \tag{2.20}$$

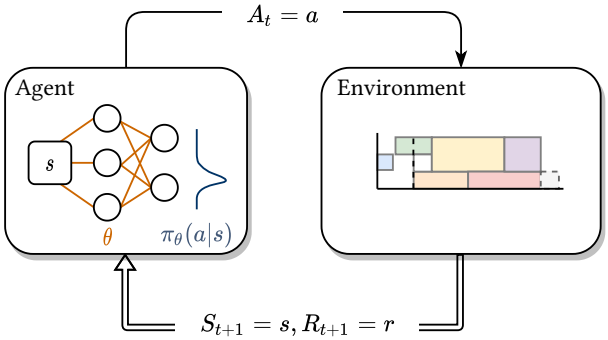
Both value functions give the value of scheduling a job until the next event happens. The advantage function can also be defined with options, and becomes $a_\pi(s,\omega) = q_\pi(s,\omega) - v_\pi(s)$.

In practice, what changes in the implementation from an action-based agent to an option-based agent is how the discount factor is handled. While in the state value and in the state-action value functions the γ term appears explicitly, when using options, discounts are made implicitly, within $r(s,\omega)$.

2.7 OpenAI Gym

Traditionally, prior to the mid-2010s, there was a shortage of open-source libraries that decoupled the development of RL environments from the development of agents that solved such environments. This was unfortunate, as experimentation with RL algorithms was accessible to few people.

With the objective of providing a software package that was convenient and accessible to RL researchers to access benchmark problems and to create new environments, the OpenAI Gym software package (also referred simply as Gym in this text) was proposed [Brockman et al., 2016] in 2016. In the years after its introduction, OpenAI Gym became a standard environment for both the introduction of new episodic RL environments, and for testing general RL algorithms.




```
done = False
state = env.reset()
while not done:
    action = agent.select_action(state)
    state, reward, done, info = env.step(action)
```

Figure 2.9: Example of the RL agent-environment interaction loop and corresponding Python code. With discrete time, scalar actions taken at time t are processed by the environment and, at time $t + 1$, the environment returns an observation vector of and a scalar that corresponds to the reward for taking action a_t. States given as input to the agent are passed into a parameterized function $\pi_\theta(a|s)$ that outputs probabilities of taking various actions when in s. The reward signal is used to adjust the values of parameters θ. In the bottom part, code that shows how agent interacts with the environment is displayed. Interaction proceeds until a terminal state is reached.

OpenAI Gym makes it easy to map an RL implementation to the RL problem, as exemplified in Figure 2.9 with an equivalent block of code that implements that loop. As can be seen in the figure, the agent chooses an action given the current state of the environment, while the environment is responsible for updating the state of the world, computing the reward of the last action taken, determining whether a terminal state was encountered, and providing any additional information to the agent, usually in the form of statistics of the episode. These responsibilities are represented, respectively, by the variables `state`, `reward`, `done`, and `info` in the bottom part of the figure.

Chapter 3

Related Work

In this chapter, we position our work within the broader literature. Our work intersects the areas of deep reinforcement learning, scheduling, and workload models, and its inspiration comes from our previous research on job placement in hybrid clouds [Cunha et al., 2017], in which we used Machine Learning (ML) to predict wait times and run times of High Performance Computing (HPC) jobs and, based on those predictions, decide whether to submit jobs to a local cluster, causing jobs to wait for the cluster to free up resources to run them, or to submit them to a cloud cluster, which would have no wait time[1]. Migrating a job to the cloud would incur additional costs, and would impact the performance of networked applications, due to network interconnects of cloud providers tending to be slower and have higher latency than their supercomputing center counterparts [Jackson et al., 2010, Marathe et al., 2013, Sadooghi et al., 2015, Xavier et al., 2013].

From the initial work on cluster selection, we've decided to take a step back from the multi-cluster setting, and investigate how well learning agents can deal with scheduling job realistically-sized clusters with realistic user run time estimates. In the following sections, we will discuss papers that share at least one characteristic with ours, but while they all share some idea with us, none of them model job scheduling as an RL problem with temporal abstraction and function approximation, nor do they consider the impacts of uncertainty in job run time estimates or study how learned algorithms behave under such settings. Additionally, none of the papers studied also provide an OpenAI Gym environment that is general enough to be used to solve scheduling problems without knowledge of the inner workings of the environment.

3.1 Reinforcement Learning for Scheduling

We begin our exploration by surveying papers that use RL in job scheduling. RL applied to scheduling problems has a somewhat long story, but early solutions tended to be applied on small problems, and with tabular (exact) methods, without state or time abstraction [Aydin and

[1] Due to the illusion of infinite resources in the cloud [Armbrust et al., 2010].

Öztemel, 2000, Zhang and Dietterich, 1995]. For example, Aydin and Öztemel [2000] deal with scheduling jobs by training an agent that selects scheduling rules. Although they formulate the problem as an RL problem, the structure of the reward function used implies a classifier could have good performance, since the reward is a function of the selected rule, the size of the queue, and the mean slack time of the queue[2]. Still, this was an early example that showed agents were able to learn to select dispatching rules. Another work also shows that, when jobs are sampled from a fixed set of tasks, and task information is passed down to the scheduler, tabular RL solves these problems [Tong et al., 2014].

In the recent wave of attempts to solve scheduling with RL, the Deep Resource Management (DeepRM) [Mao et al., 2016] algorithm was an influential one. DeepRM uses the Reinforce [Williams, 1992] policy gradient algorithm with state baselines to schedule jobs based on their time, CPU, and memory requirements. DeepRM's state representation uses matrices to treat jobs as images, and using those images as input to choose which job to schedule next. DeepRM mainly optimizes for job slowdown, but both it and other methods are able to optimize for other metrics [Fan et al., 2021, Huang and Ontañón, 2022]. DeepRM also influenced our work, as our initial experiments reproduced Mao et al. [2016] results both with a Reinforce implementation, and with state-of-the-art RL agents that use the techniques discussed in the previous chapter.

Continuing with the papers influenced by DeepRM, Domeniconi et al. [2019] proposed CuSH, a system that built on DeepRM to schedule for CPUs and GPUs, but using a hierarchical agent. The agent works in two phases: in the first, it chooses the next job to send to the second phase, a policy network that chooses the scheduling policy to use to determine when the job will run. It is important to highlight a major difference between DeepRM and CuSH: whereas DeepRM learns the scheduling *policy* itself, CuSH is essentially a classifier, which chooses between two existing policies. This means that, even without training, CuSH's behavior is expected to be more stable than that of DeepRM, since DeepRM-style schedulers might get stuck in local minima 5.1.2.

Another agent that has been proposed recently, and that learns the scheduling policy itself, is rlscheduler [Zhang et al., 2020]. rlscheduler is a PPO-based agent with a kernel-based design that looks at job features for scoring them. rlscheduler scores jobs in a window of up to 128 jobs over the queue. The major innovation in RLScheduler is in the training setting, in which the authors combine synthetic workload traces with real workload traces to present the agent with ever more difficult settings, similar to learning a curriculum of tasks. A potential deficiency with rlscheduler is that it only uses local job information for making decisions, without considering the cluster's occupancy state.

Similarly to CuSH, other agents that use a classification approach have been proposed. A recent one is the Deep Reinforcement agent for Scheduling in HPC (DRAS) [Fan et al., 2021], which classifies jobs in three categories: *ready*, *reserved*, and *backfilled*. After this classifica-

[2] the amount of time a job can be delayed without causing another job to be delayed.

tion step, the cluster scheduler takes the output of this classification and allocates jobs accordingly (for example, by reserving slots in the future for reserved jobs, scheduling immediately ready jobs, and finding "holes" in the schedule for backfilled jobs). DRAS uses a five-layer Convolutional Neural Network (CNN) that works in two levels, with the first level selecting jobs for immediate and reserved execution, and the second layer for backfilled execution.

The papers discussed above are the closest to the work we describe here, with the most notable differences being in the type of learning (whether a reinforcement learned algorithm selector, or a learner that chooses jobs directly), whether a single or multiple resource requirements were handled, in the sizes of the clusters considered, and whether the learned agents are transferrable between clusters of different sizes. As is usual with algorithms with multiple parameters, some works have investigated the performance of different policy-gradient algorithms, while others have taken the approach of investigating the impact of different network architectures on agents that use a single optimization algorithm [Liang et al., 2019].

Peng et al. [2019] propose DL^2, a Deep Learning (DL)-based scheduler for DL clusters. They do so by first training a classifier on offline data, which is then used to train an RL agent online. Since they focus on DL workloads, their jobs use a Parameter Server (PS) architecture, in which training of a neural network is split between a group of Parameter Server (PS) (which hold and serve copies of the neural network parameters), and workers, which sample data and compute gradient updates for those networks. In this domain, their joint offline-online training model outperforms other learning agents and state-of-the-art heuristic systems.

Apart from HPC jobs, there have been approaches for scheduling workflows with reinforcement learning [Baheri and Guan, 2020, Kintsakis et al., 2019, Mao et al., 2019b, Tong et al., 2020, Wang et al., 2019], using machine learning for deriving static scheduling policies for job scheduling [Carastan-Santos and De Camargo, 2017], and optimization techniques [Fan and Lan, 2019] for doing so. Outside of reinforcement learning, many ML techniques have been used to aid in the scheduling of jobs, such as predicting resource and time requirements [Cunha et al., 2017, Fan et al., 2017, Rodrigues et al., 2016, Smith et al., 1999, 2004, Xu et al., 2022]. Additionally, with the rise in popularity of DL as a whole, learning schedulers for DL have also been proposed lately [Peng et al., 2019].

From our point of view, few, if any, papers have recently reported the effects of decisions in modeling the scheduling HPC jobs for solving with RL while considering the impacts of uncertainty in job runtime estimates. Nor do any of the papers related to job scheduling model the problem as an SMDP, both topics discussed in this work.

3.2 On predicting job features

It is interesting to contrast end-to-end RL approaches, such as the one taken in this dissertation, with approaches that try to predict job characteristics or features to aid in resource management. For example, Kumar and Vadhiyar [2013] developed a technique that defines which jobs can be classified as *quick starters*. These are jobs with short waiting times compared to the other jobs waiting for resources. Their technique considers both job characteristics such as request size and estimated runtime, and the state of the system, including queue and processor occupancy states. With Kumar and Vadhiyar's work, one could perform a classification of *quick starters* to, for example, aid in backfilling decisions. In RL systems, such a classification step may not be needed, as deep RL can learn to leave room for smaller clusters automatically [Mao et al., 2016].

Even in the work we mentioned at the beginning of this chapter, an RL approach could bring benefits. For example, there we estimated the variance of the predictions and used a cut-off function to determine whether predictions should be trusted or not. The parameters of such cut-off function encoded the propensity to risk of the system operator. Still, this function was modelled manually, and could be improved to support fully automatic decisions. One option to stop using the cut-off function and migrating to a fully automated approach would be to use RL, which naturally solves these kinds of sequential problems. In the following paragraphs, we quickly survey techniques that can be used to augment data in classical schedulers to improve performance.

Related to job characteristics other than CPU usage, research that focuses on memory usage predictions tends to be based on benchmarking [Marin and Mellor-Crummey, 2004, Matsunaga and Fortes, 2010, Nudd et al., 2000, Snavely et al., 2002, Wood et al., 2008, Yang et al., 2005]. For example, Matsunaga and Fortes [2010] assessed various machine learning techniques for predicting spatio-temporal utilization of resources by user applications; memory is one of the resources investigated. Their experiments focused on two bioinformatics applications and involved the execution of the applications with different parameter configurations. Another example is from Wood et al. [2008], who designed an approach for estimating the resource requirements of user applications motivated by the need to move such applications to virtualized environments. Their approach relies on a set of microbenchmarks to profile the different types of virtualization overhead on a given platform and a regression-based model to map the native system usage profile into a virtualized one.

Other researchers predict different metrics using, instead of a single model, an ensemble of models, both for memory [Rodrigues et al., 2016] and for run time [Chen et al., 2020].

Similar to memory, some researchers built systems and proposed techniques that mined scheduler logs to make predictions [Hariharan et al., 2020, Smith, 2007, Tsafrir et al., 2007, Yang et al., 2005], while others used benchmarks [Yang et al., 2005], and others used Instance-

Based Learning techniques to predict not only memory and run time, but also wait time and data transfer times [Smith, 2007, Smith et al., 2004]. Yang et al. [2005] proposed a technique to predict the execution time of jobs in multiple platforms. Their method is based on data collected from short executions of a job and the relative performance of each platform.

The techniques above might be helpful for building a comprehensive system that augments job information before making scheduling decisions, but as presented, they are outside the scope of this work. Still, with the above presentation we hope to have given the reader a perspective of how different research groups approach the problem of predicting different job features.

3.3 Reinforcement Learning in Resource Management

In the wider context of resource management in general, some papers provide interesting conclusions and guidance for us. For example, in some contexts, it has been shown that some environments already enforce an appropriate level of exploration, indicating that agents that prefer to exploit learned information might perform better than agents that include explicit exploration [Glaubius et al., 2010]. In our context, we see that is not the case, and including an explicit exploration component in agents improves performance. In other contexts, it has been shown that it is possible to choose resources in a multi-objective setting [Zhang et al., 2017].

Kumar et al. [2019] surveyed the literature of scheduling techniques in cloud computing. Related to learning techniques, their findings indicate it is difficult to predict upcoming workloads in clouds, with more powerful learning techniques needed, with machine learning techniques also having potential for failure prediction. In the body of work they identified related to resource scheduling, few pieces of work are related to scheduling tasks with learning, adaptive algorithms. Although some use RL techniques that are able to successfully make accurate predictions [Wang and Gelenbe, 2015], others focus on resource provisioning, with the potential of reducing costs in hybrid clouds [Kumar and Ravichandran, 2013], Virtual Machine (VM) placement with online monitoring to minimize performance degradation [Sotiriadis et al., 2018], which resources to lend from public clouds to extend private clouds [Bittencourt and Madeira, 2011] in order to reduce operating costs while maintaining desired execution times.

Luong et al. [2019] surveyed the area of applications of deep RL in communications and networking. Of special interest to us is the literature review on resource sharing and scheduling. Of these, they identify DeepRM [Mao et al., 2016], which we mentioned above, as an example of learned scheduler. Also included in the survey is a model-free actor-critic

method for scheduling in distributed data stream processing systems [Li et al., 2018], in which the authors write an agent to assign workloads to workers, and which, when compared to a default algorithm, can reduce average processing time by up to 33.5%.

Other areas which have seen successful application of both tabular and deep RL for resource management are in allocation of resources for Internet-of-Things systems [Gai and Qiu, 2018], Quality-of-Service (QoS)-aware scheduling for cloud service providers [Wei et al., 2018], and in the reconfiguration of Data Stream Processing applications [da Silva Veith et al., 2019].

3.4 Resource allocation environments in OpenAI Gym

To finish this chapter, given the practical approach we take in simulating and evaluating our agents, it might be useful to perform a brief exploration of resource management environments based on OpenAI Gym, or similar systems. In general resource management, there have been environments proposed for fields as diverse as the optimization of networking systems [Gawłowicz and Zubow, 2019], the optimization of demand response for electricity [Vázquez-Canteli et al., 2019], and internet congestion control algorithms [Jay et al., 2019].

Still, few studies in resource management use RL frameworks for performing research. Mao et al. [2019a] propose a set of environments (Park) for studying computational resource management issues, but they do so by proposing a new framework with similar, but not identical, APIs of existing RL frameworks. Due to that, it becomes harder to adapt agents and algorithms designed for these standard frameworks. Also, algorithms written for Park will only be able to be evaluated in Park as well, which might reduce its appeal for researchers outside the group that proposed the Park framework. Similarly to Park, DRAS [Fan et al., 2021] uses its own environment on top of the CQSim simulator.

There are groups that use OpenAI Gym as a framework, such as RLScheduler [Zhang et al., 2020], but often there is not a clear separation between the code for the environment, and the code for the agents that solve the environment, which differs from the approach taken in this work. Here, we set to share an environment that might be solved with *any* algorithm and, in fact, we use standard algorithm implementations to solve our environments with our parameterization.

3.5 Summary

In this chapter, we have seen that although ML techniques have been studied and applied extensively in resource management and scheduling, most are applied to the goal of making predictions of a single aspect of resource managers. When RL is used to learn schedulers, it usually uses a small, fixed set of actions in the action space. Moreover, even applications based on deep learning tend to be applied to a limited set of configurations, breaking when applied to models that resemble more realistic systems.

From our review of the literature, we identify the need for learning algorithms that are able to adapt to large-scale, realistic clusters, while also being able to properly select jobs and allocate them without arbitrary limitations in their action spaces. In a recent survey paper, leading researchers in the area of cloud computing stress the need for novel resource management and scheduling methods for hybrid clouds and federated clouds, while acknowledging that current ML and Artificial Intelligence (AI) methods can be brittle [Buyya et al., 2018].

Part of this dissertation uses an approach that is conceptually similar to that of Mao et al. [2016], but we show that a simpler design based on standardized components allow for faster convergence of learning algorithms. Our proposal also differs from theirs by modelling the problem in a scalable way, allowing us to study problems at the scale of actual, existing clusters. Additionally, we test our approach with real-world workload traces, while evaluating the generalization capabilities of the learned agents.

Chapter 4

HPC job scheduling with RL

Now that we have reviewed the theoretical underpinnings of deep RL and the problem of HPC job scheduling, we present our methods and models. We begin by describing the problem we are set to solve, followed by a description of how such a problem might be simulated to solve it. Then, we present a series of incremental MDP formulations, culminating in one definition that allows us to solve it in an elegant and efficient way.

4.1 Problem description

In the past, HPC clusters were dominated by bag-of-tasks applications and tightly-coupled message-passing applications. In recent years, we have seen this scenario changing, as AI applications gain popularity in HPC centers. Evidence of this comes from the increasing number of clusters in the top-500 list with specialized accelerators (most notably GPUs) for processing AI workloads. Increasingly, HPC clusters are now handling high-throughput, data-intensive, stream-processing applications [Rodrigo et al., 2018], in addition to traditional applications.

Being based on a new simulator, we focus on rigid and moldable jobs, which are easier to implement, that run in parallel machines, leaving improvements to the simulation model for future work. These types of jobs represent a hyperrectangle in the space of the number of types of resources being scheduled and time. For example, if we only consider processor usage, the job is represented as a rectangle of processors × time. If memory and processors are considered, the job is a rectangular cuboid of processors × memory × time and so on [1]. Considering these types of jobs allows us to validate our approach with similar approaches from the literature. Also, such models allow us to study execution environments for scientific computing applications, and neural network training procedures that use distributed neural network training frameworks, such as Horovod [Sergeev and Del Balso, 2018] or the Distributed Deep Learning (DDL) [Cho et al., 2017] library.

[1] In these cases, we consider peak memory consumption to represent the memory dimension.

Given the above definition, the problem we set to solve is that of learning, by reinforcement, an efficient scheduler for rigid jobs that supports machines of realistic sizes, of up to hundreds of thousands of processors, and that also supports *transfer* of learned schedulers between machines, such that a scheduler learned on one machine could be used in another without the need for retraining.

4.2 On the need for simulators and frameworks

Besides algorithmic breakthroughs [Mnih et al., 2015, 2016, Schulman et al., 2017], the availability of environments in which agents can observe the consequences of their decisions, such as simulators, is of crucial importance for deep RL. Not coincidentally, the vast majority of success stories with deep RL comes from gaming. Although there are plenty of environments for gaming agents and, to some extent, for agents focused on specific aspects of robotics problems, there is still a lack of environments that could benefit systems research [Mao et al., 2019a], which tend to be sequential in nature. An implementer could set up agents that learn directly from the systems they were to optimize, but that would be costly in number of samples, and inefficient, particularly as algorithms choose exploratory actions in search for better policies. A cheaper and more efficient way of learning, then, is to simulate these problems.

Clearly, real-world performance of a policy learned through simulation will only be as good as the simulation process and the workload data fed into the simulator. In this dissertation, we make the assumption that communication (data loading, networking, etc.) costs between processors are negligible, an assumption shared by work we compare ours with, making our results at least comparable with those in the literature. As for the workload fed into the simulator, we go all the way from simplistic models (§ 2.3.1) to simulating with real workload traces, with simulating realistic workload models (§ 2.3.2) in-between, even considering uncertainty in run time estimates, both simplistic (§ 2.4.1) and realistic (§ 2.4.2).

To connect an environment, simulated or otherwise, to a (deep reinforcement) learning agent, we need to somehow present the environment to the agent as an MDP. From a high-level perspective, one might argue why use a toolkit for RL research, when the agent-environment interface (exemplified in Figure 2.9 and formally defined in Chapter 2) is simple. This is a valid concern, but although the interface itself might be simple, environments themselves are not, making it easy to introduce bugs that hinder learning. Moreover, deep RL has faced reproducibility issues in the past, in part due to the high variance of learning methods, and in part due to the different metrics used for reporting results by different researchers [Henderson et al., 2018, Machado et al., 2018]. Additionally, by using an existing toolkit, researchers get helper functionality for free. For example, OpenAI Gym has functionality for representing

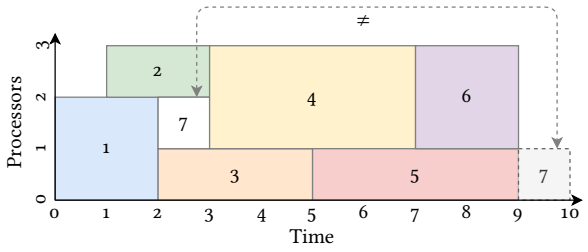
state and action spaces, functionality for modifying inputs (state and rewards) and outputs (actions), recording trajectories both for imitation learning and for human consumption, and libraries of peer-reviewed baseline algorithm implementations that help speed up research.

For the reasons above, instead of creating our own interface, we decided to connect our simulation environments to the agent while following the OpenAI Gym interface. We also made the simulation and environment code publicly available on the Internet, encouraging the reproducibility of the work presented here. As a consequence of this decision, after reading this chapter, the reader should be capable of understanding how we implemented a simulator and connected it to the OpenAI Gym environment, while also being able to evaluate scheduling policies for HPC clusters in that environment.

4.3 Job Scheduling Simulation

We started by implementing a discrete-event simulator for simulating job submission and execution. We could have started from an existing simulator used in previous research [Cunha et al., 2017], but after careful analysis, we came to the conclusion that the amount of work needed to refactor a simulator to interface with an RL framework would be better spent writing a new one from scratch with extensibility in mind, while ensuring it works well with the new environment. The key point to consider is that, to represent state, the RL environment needs access to data structures internal to the simulator, while also requiring fine control of the simulation process, which becomes easier when the simulator is already written with inspection and control from external applications in mind.

The design and development of the core components of the simulator followed a Test-Driven Development (TDD) approach, in which the design of the components followed from the specification of tests of how the components should and should not behave. This approach enabled us to have confidence about the behavior of the simulation system. One example of test of correctness is shown in Figure 4.1. In the top left part of the figure, we show the expected schedule when seven jobs are submitted to two different scheduling systems. In the bottom left part of the figure, we show the sequence of jobs that generated such schedules. And in the right part of the figure we show how we encoded the sequence of job submissions and their characteristics (number of requested processors and duration for this example) and specified the start, wait, and finish times of each job. In the right part of the figure we show code that implements this test. Ensuring tests such as this passed gave us confidence the implementation was sound. Since tests alone cannot prove correctness, we also inserted assertions to ensure invariants of algorithms were maintained. The TDD approach we followed allowed us to achieve an overall line-based test coverage of 95.8%, suggesting the code has lower chance of



Job	Submission time	Duration	Processors
1	0	2	2
2	1	2	1
3	1	3	1
4	1	4	2
5	1	4	1
6	1	2	2
7	2	1	1

```
s = scheduler.BackfillingScheduler(3)
s.submit(make_job(0, 2, 2))
s.schedule()  # schedules all jobs
s.step()  # forwards time in simulator

s.submit(make_job(1, 2, 1))
s.submit(make_job(1, 3, 1))
s.submit(make_job(1, 4, 2))
s.submit(make_job(1, 4, 1))
s.submit(make_job(1, 2, 2))
s.schedule() # schedules all jobs
s.step()  # forwards time in simulator

s.submit(make_job(2, 1, 1))
s.schedule() # schedules all jobs

assert 0 == j1.start_time
assert 1 == j2.start_time
assert 2 == j3.start_time
assert 3 == j4.start_time
assert 5 == j5.start_time
assert 7 == j6.start_time
assert 2 == j7.start_time
```

Figure 4.1: Example of a test of the simulator. On the top left, we see a graphical representation of two possible schedules, one for a backfilling algorithm, and another for a FIFO scheduling algorithm. On the bottom left, we see, for each job, its submission time, its duration, and number of requested processors. On the right, we see a code excerpt for testing a backfilling scheduler in our simulator. The code begins by creating a scheduler with three processors, submitting a job, scheduling it and advancing time. Then, five jobs are submitted and all are scheduled at the same time. Finally, a seventh job is submitted and scheduled, and the test proceeds to assert that job start times equal those shown in the top-left figure.

containing undetected software bugs.

The jobs that end up being submitted to the system and scheduled are generated by a *Workload Generator* (described in Section 2.3). We added two main methods for simulating job *submission*. Jobs can be sampled from distributions, or they can be generated from workload traces. Traces are supported in our simulator by using the Standard Workload Format (SWF). SWF files are text files that contain, on each line, characteristics of jobs, such as submission time, wait time, requested execution time, processors requested, memory requested, actual run time, and so on[2]. The set of features present in the SWF, or features that can be computed from those present in the SWF, limit the set of features we use in this dissertation. The Parallel Workloads Archive [Feitelson et al., 2012] contains workload files from supercomputers recorded over a period of almost three decades and we use some of those files to simulate the loads of real supercomputers.

[2] Although there are many fields defined in the SWF format, most traces only contain information about the various job times, their ids, groups, queues, and numbers of processors (both requested and allocated).

4.4 The simulator as an OpenAI Gym environment

With a working simulator, we now have to define an environment that would work with the OpenAI Gym framework. Recall from Chapter 2 that we need an MDP definition to solve with RL, so in this section we describe a base MDP and how we linked it to OpenAI Gym.

The MDP comprises the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \rho, \gamma \rangle$. In this context, the combination of a workload generation model with the simulator is used to implement the transition function $\mathcal{T}$. Notice that, at the MDP level, time is discrete[3], and always proceeds from one time-step to the next, so even with an event-driven simulator, all intermediate states between two events need to be presented to the agent, a restriction we will relax shortly in Section 4.7. For this reason, in a software implementation, the RL framework needs to have the ability to control the simulator, and define when time advances for the interactive RL loop to work.

To interface with OpenAI Gym, environments are required to inherit from the `gym.Env` Python class and implement the following five methods: `__init__`, for initialization; `step`, for implementing the MDP itself via the $\mathcal{T}(s, a)$ function, the set of states $\mathcal{S}$, the set of actions $\mathcal{A}$ and the reward signal from $\mathcal{R}$; `reset`, for resetting the environment state; `render`, for rendering the environment (both for human visualization or consumption by other programs); and `close` for closing the environment and freeing any resources.

Once an environment is registered with OpenAI Gym, it can be instantiated with a call to the `gym.make` function with the environment name passed as string. This will instantiate a new environment and call the `__init__` method of our environment. This function takes an optional argument that is passed to the initializer. We use this function to be able to *configure* the environment. One such configuration option is how the workload is going to be generated, which uses the workload generation models described in Section 2.3.

Continuing with the description of the MDP, in this work, we consider an episodic setting which always begins with an empty cluster, meaning that the `reset` function implements $\rho = \{\llcorner\!\!\lrcorner\}$, the empty cluster. Since we don't know beforehand how long an episode will be, we assume $\gamma = 0.99$, meaning that, for each state, there is a 1% chance of the episode ending after that state. To complete our description of the MDP, we now need to define the functions $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{S}$, which are intrinsically linked to how we simulate a system, and present its state to the learning agent.

For job scheduling, the set of states $\mathcal{S}$ may represent the set of all possible combinations of resource usage in the cluster, which may be represented as a set of images (Figure 4.2), or by a set of tabular features. Depending on the type of function approximation method in use, it may only support inputs of a fixed size and, therefore, the state representation may have to be padded or truncated to fit the requirements of the function approximator. For example: al-

[3]SMDPs allow modeling continuous-time transitions [Sutton et al., 1999].

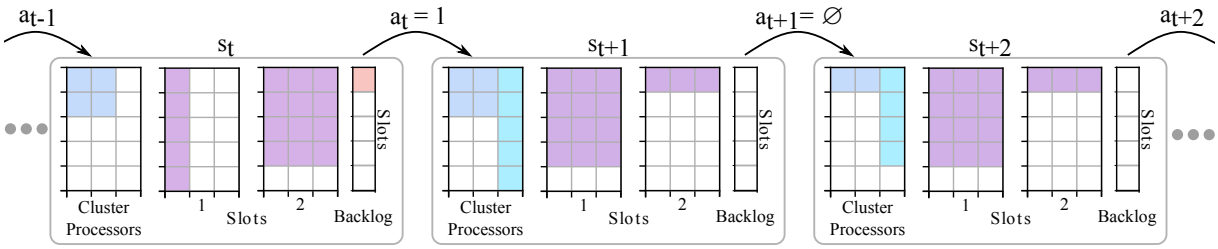


Figure 4.2: Snapshot of three frames of the base MDP implemented in this work when rendered via the OpenAI Gym rendering function, along with annotations to aid in understanding. At S_t, a job is in the cluster, running for two timesteps, while there are two jobs in the queue and one in the backlog. The agent selects the job in slot 1 to run, and transitions to state S_{t+1}. The job that was running will still run from two more timesteps, because *wall clock simulation time doesn't pass when correct scheduling decisions are made*, to allow the algorithm to make as many valid schedules as possible in one time step. From S_{t+1} to S_{t+2}, the agent selects the empty action, transitions to state S_{t+2} and wall clock simulation time proceeds, making the original job in the cluster run for one time step.

though the waiting queue is unbounded, approximators such as Multi-Layer Perceptron (MLP) neural networks require inputs of a fixed size. Hence, one might have to truncate the queue representation should it grow larger than the maximum number of supported inputs in the neural network.

Jobs that enter the system will either be added to the waiting queue, or, when it is full, will be added to a backlog. Jobs in the backlog have their characteristics concealed. All the agent knows is that such position is occupied in the backlog. In our base MDP, we consider state representation $\mathcal{S}$ to be like an image with three major components: (1) a cluster state, (2) a window W of observation into the queue state, and (3) the backlog, jobs in the queue that do not fit the window of observation. The backlog serves only as a representation of "overflowing" jobs, but the components 1 and 2 are repeated for each resource type represented in the cluster and which is used for scheduling. These three components define the width of the image representation, whereas the height is defined by a horizon into the future, meaning that, given the current schedule in the cluster, if no other decisions are made, what will be cluster state in H time-steps in the future. As an example, consider the three states for three different time-steps shown in Figure 4.2. There, we consider a single resource type: processors, of which there are three, and we consider a future horizon of 5 time steps. In the first state, S_t, there are two cluster processors being used for the next two time steps. In the transition from S_t to S_{t+1}, the agent scheduled a job that was in one of the two job slots (which offer the view into the job queue), and that job was added to the current cluster state and, since a job slot was made free, a job that was in the backlog got moved into the view of the queue. If we were scheduling another resource such as memory, instead of two slots, we would have four: two for the processors, and two for the memory (which could have a different width, as the cluster could have any amount of memory units). Similarly, we would also have a view of the cluster processors, and another for the memory currently in use.

Above, we said the agent selected a job for scheduling. In the base MDP, the cardinality of the set of actions $\mathcal{A}$ available at any given state is determined by the window W and, consequently, each potential action in $\mathcal{A}$ corresponds to the index of the job in the job slots, with an additional action, $\emptyset$, which corresponds to not choosing any job, and stepping the simulator. Similarly, we built the MDP in a way that, if an action corresponds to a position that doesn't contain a job (for example, when the window W is larger than the size of the waiting queue), it corresponds to choosing the empty action $\emptyset$. With this description, now we can fully describe the transition function $\mathcal{T}(S_t, A_t)$: when a corresponds to an index in the window into the job queue that has a job that *fits* the cluster, the transition from s to the next state S' updates the state representation, but does *not* increase wall clock simulation time (as the agent may be able to make more than one decision in a single simulation time step), whereas if $a = \emptyset$, or a selects a job that does *not* fit the cluster, then simulation time is increased (and the state representation for S' is updated accordingly).

Defining a useful reward signal $\mathcal{R}$ in an MDP is a challenging aspect of RL, especially when the environment and the task to be performed are complex [Sutton and Barto, 2018b], in which simplistic reward signals may lead agents to optimize for goals different from those of the environment's designer. In the context of job scheduling, one might think that the agent could optimize for the performance metric, such as slowdown, used to evaluate the system. In many cases, this is not possible, though. Recall that the reward signal must be updated after *each* action performed by the agent and since performance metrics usually depend on knowing the finish time of jobs, using performance metrics would make reward signals more *sparse*, complicating credit assignment for actions taken.

In the case of slowdown, we have already devised a suitable, online version of the metric in eq. (2.1), with the key insight being that, the longer it takes for a job to start running, the larger its slowdown will be, and, once the job finishes execution, its slowdown will be completely determined. Therefore, the only component of slowdown that is somewhat under the control of the scheduler is the job's wait time. Now, to convert slowdown to an online metric, notice that the $t_w(j)$ component is exactly the sum of all the time steps in which j was *not* running. Similarly, $t_e(j)$ is exactly the sum of all timesteps in which the job was running. Therefore, an equivalent online definition of slowdown would assign $1/t_e(j)$ to job j for all time steps between submission and end of execution of job j. To make a proper reward signal, recall that we want to *minimize* slowdown but are using a gradient *ascent* update rule. Hence, a correct reward signal that minimizes slowdown for job j in this setting is $-1/t_e(j)$. The same reasoning applies to bounded slowdown and, following the same process, we arrive at optimizing for makespan being equivalent to a -1 reward for each time step a job hasn't finished executing.

We now know how to compute the reward considering a single job j, but we still have to define *which* jobs to use to compute the reward for the agent. For now, following the MDP proposed by [Mao et al., 2016], we will use all jobs in the system to compute the reward for an

action, leaving the discussion of the consequences of doing so to a later section. To encourage the agent to schedule jobs, we add a conditional term giving reward 0 (the maximum possible reward in our MDP) when a job is scheduled successfully by an action. Hence, for now, we define the reward signal as

$$\mathcal{R}(s,a) = \begin{cases} 0, & \text{if a job is scheduled successfully, and} \\ -\sum_{j \in \mathcal{J}} \frac{1}{t_e(j)} & \text{otherwise,} \end{cases} \tag{4.1}$$

where $\mathcal{J}$ is the set of *all* jobs in the system.

4.5 Policy network and learning procedure

With the MDP properly defined, we can now solve it, which we are interested in doing with Monte Carlo methods, and the first method we used is the REINFORCE algorithm. REINFORCE works in two phases: a sampling phase that collects trajectories and a learning phase that updates the weights based on the collected trajectories. In the sampling phase, the agents use fixed weights to interact with the environment, obtaining trajectories, sequences $\{S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T\}$ of states, actions and rewards, where T is the maximum length of an episode, configured at environment creation time (§ 4.4). Algorithm 1 considers multiple workers are used to optimize θ and shows one iteration of the learning procedure. The algorithm takes as input a step size $\alpha > 0$, a number of workers N to be executed in parallel, a number of episodes E to be sampled by each worker, and a maximum length of episodes T.

Since adjustment of weights only occurs in a specific phase, it is possible to perform data collection in parallel. We've implemented two distinct strategies: (i) each worker computed weight updates locally and propagated them to all other workers (in HOGWILD! [Recht et al., 2011] style) and (ii) a worker received trajectories from all other workers, performed learning locally and broadcast new weights to all other workers (in a synchronous Parameter Server style). As shown in the algorithm, baselines are computed as the average reward at each timestep. In variant (i) of the algorithm, baselines are computed locally and are not shared, while in variant (ii), baselines are shared between all episodes, since they are computed by a central worker. Algorithm 1 computes baselines locally, implementing variant (i) above. As we shall see, there seems to be no significant differences in algorithm performance between both methods of computing baselines.

As required by Algorithm 1, π_θ needs to be differentiable. In this work we use a MLP, a fully-connected feed-forward neural network as differentiable function approximator. This neural network works as a classifier, and uses a softmax layer to normalize its outputs, with size

$W+1$, and selects an index in the window over the waiting queue. In the example of Figure 4.2, there are three classes, corresponding to the first waiting job, the second, or selecting no job, as discussed earlier.

Also in Algorithm 1, we compute baselines for variance reduction by using the average return over all trajectories at time step t (where me make $b_t \leftarrow 1/E \sum_{i=0}^{E-1} G_t^i$). This approach has two problems: (1) it defines an implicit synchronization barrier, as learning can only happen after all agents have collected a trajectory, and (2), a bigger problem which requires that all trajectories are made of the same size, either via padding trajectories to match the longest one, or via truncation to the length of the shortest one. Both cases of (2) are undesirable, as they either throw data away, or require choosing a value for padding, which may skew the averages. A better approach would be to *learn* the baseline function, as performed by actor-critic algorithms such as A2C [Mnih et al., 2016] and PPO [Schulman et al., 2017], as discussed in Section 2.5.1, and highlighted in eq. (2.14).

A natural question to ask is whether these alternative implementations, which perform bootstrapping and learn both the policy, and the value function together, are any better than the simple baseline of Algorithm 1, and whether they have any advantages. The main advantage of the baseline of Algorithm 1 is that it is simple to compute. The main disadvantage is that it forces us to keep all trajectories in memory for a given set of weights θ, and to either pad or truncate the trajectories so that they all have the same length. A learned baseline, although harder to compute, especially in the first episodes of training, is more scalable, in the sense that its size is not coupled to the number of workers used in learning.

We compared the performance of both approaches and, for the learned baselines, we used a set of publicly available, reviewed implementations [Dhariwal et al., 2017] of the A2C and PPO algorithms to optimize our MDP.

Algorithm 1: REINFORCE with average baselines

Result: Learned weights θ for policy $\pi_\theta \approx \pi_*$
Input : Differentiable policy $\pi_\theta(a|s)$
Input : Parameters $\alpha > 0, E, T, N$
for *each worker* $w \leftarrow 0$ **to** $N-1$ **do**
  $\Delta\theta_w \leftarrow 0$
  for *each episode* $i \leftarrow 0$ **to** $E-1$ **do** // Sampling phase
    $e_i \leftarrow \{S_0^i, A_0^i, R_1^i, \ldots, S_{T-1}^i, A_{T-1}^i, r_T^i\} \sim \pi_\theta(\cdot|\cdot)$
    foreach $\{t \mid 0 \leq t < T\}$ **do** $G_t^i \leftarrow \sum_{k=t}^{T} \gamma^{k-t-1} R_k^i$
  end
  for *each time step* $t \leftarrow 0$ **to** $T-1$ **do** // Learning phase
    $b_t \leftarrow 1/E \sum_{i=0}^{E-1} G_t^i$
    foreach $\{i \mid 0 \leq i < E\}$ **do** $\Delta\theta_w \leftarrow \Delta\theta_w + \alpha \nabla \log \pi_\theta(A_t^i|S_t^i)\,(G_t - b_t)$
  end
end
foreach $\Delta\theta_w$ **do** $\theta \leftarrow \theta + \Delta\theta_w$

4.6 Alternative MDP formulations

Although the model we've discussed so far is useful for understanding the behavior of learning agents in simple job scheduling settings, it has serious limitations. For example, the MDP we've presented uses jobs that have perfectly accurate runtime estimates, which, as we've discussed in Chapter 3, is not a characteristic of actual clusters. Moreover, when we consider the base MDP requires unit time step increments, we come to the conclusion that this is not a practical model, as it makes simulations much longer than needed, and is noisy for training, as the agent is forced to act even when there are no jobs to schedule. Also, as we deal with longer episodes and larger numbers of workers, we start to face a storage problem.

The MLP neural network described in the previous section have too large memory requirements when evaluated on clusters with realistic sizes. Recall from the description of the state representation that, for each resource type we represent, there is a cell in a matrix for each unit of that resource. On top of the current utilization, each job slot also takes one cell to represent a resource used by the job it represents, with an additional column vector representing each job in the backlog. These requirements are multiplied by the number of time steps we have to represent in the time horizon.

We can formalize the problem as follows: let N be the number of workers, E be the number of trajectories collected by each worker, H be the number of time steps in the time horizon, J be the number of job slots in the state representation, P be the number of processors in the cluster, L be the average maximum length of an episode[4], and B be the backlog size such that $B/H = k \in \mathbb{N}_1$. In this case, the number of cells to store in memory for the learning phase of Algorithm 1 when considering a single resource type is given by $N \cdot E \cdot L \cdot H \cdot (J+1) \cdot P \cdot k$. To put it in practical terms, 6 workers collecting 10 trajectories with an average size of 100 steps, with a time horizon of 20 time steps with 10 job slots and 32 processors in the cluster and 60 entries in the backlog would require approximately 483 MiB of memory when using 4 bytes to represent a matrix cell. A rather high use of resources for such a modest configuration.

Due to the issues above, we propose a series of incremental changes to the base MDP: (i) compact state representations for reduced memory usage, (ii) sparse state transitions, (iii) limiting the set of jobs used for reward computation, and (iv) an option semi-MDP model.

[4]Assume smaller episodes are padded for the learning procedure.

Table 4.1: Job features in a compact state representation.

Feature	Description
Submission time	Time at which the job was submitted
Requested time	Amount of time requested to execute the job
Requested processors	Number of processors requested at the submission time
Queue size	Number of jobs in the wait queue at job submission time
Queued work	Amount of work that was in the queue at job submission time
Free processors	Amount of free processors when the job was submitted
Remaining work	Amount of work remaining to be executed at job submission time
Backlog	The number of jobs waiting outside window W
Can start now	Whether job j fits the cluster to start at current time

4.6.1 Compact state representation

Particularly when working with larger clusters, or with a larger number of jobs in the agent's window of attention, the state representation of the base MDP is wasteful, and it may be the case that trajectories take too much space in memory, reducing the computational performance of learning agents. Due to that, we devised a set of features that can represent states in a compact way. In our new state representation, jobs in the queue are represented by the features shown in Table 4.1, where "work" is computed by multiplying the number of processors a job requires by the time it is expected to run, and cluster features are a pair that indicates the number of processors in use, and the number of free processors. The features related to the cluster state still use a time horizon H but instead of using a matrix, we used a pair of integers representing how many resource units are in use, and how many resource units are free in a given time-step. As an example, consider the state shown in Figure 4.3, which represents scheduling for jobs only considering processor usage. Assuming the job in the cluster was submitted at time step 1, the job in slot 1 was submitted at time 2, and the job in slot 2 was submitted in time 3, that state could be fully described by the concatenation of vectors with cluster state $\langle(2,1),(2,1),(0,3),(0,3),(0,3)\rangle$, jobs in window W $\langle(1,5,1,0,0,1,6,0,1),(2,4,3,1,5,1,4,0,0)\rangle$ and backlog $\langle 1\rangle^5$. The features in the jobs slots are presented in the same order as the ones shown in Table 4.1. To understand

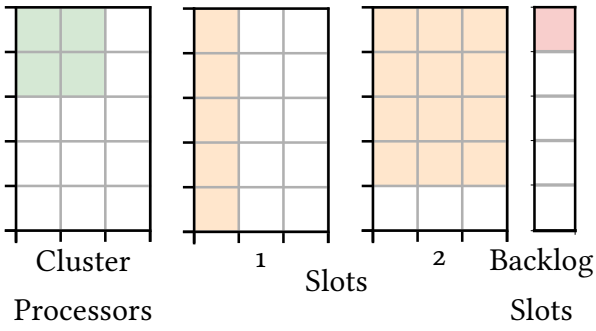


Figure 4.3: Image-like state representation. In the figure, there is one job in execution (with two processors for the next two time steps), three waiting jobs in total, two of them within window $W = 2$, one using one CPU for at least five time steps, and another using three CPUs for four time steps. Details for the third job, in the backlog, are omitted.

[5]Parentheses group elements. In the first vector, there are five parenthesized pairs to indicate the time horizon of 5, and two parenthesized elements to represent jobs shown in window W.

the relationship between the tuples and the figure used in this example, notice that there are three processors in the cluster (the width of the "Cluster Processors" and "Slots" rectangles), and that the time horizon used is of five time steps (the height of the rectangles in the figure). The green square in the "Cluster Processors" rectangle represents the fact that a job is using two processors for two time steps, meaning that there is still a single processor free in these two time steps. Hence, for each these two time steps we get the tuple $(2, 1)$. When that job is finished using the cluster, all cluster processors are free, giving the tuple $(3, 0)$ for each of the remaining time steps of the time horizon of the figure. Combining these two components gives us the first vector $\langle(2, 1), (2, 1), (0, 3), (0, 3), (0, 3)\rangle$. The second vector, which represents jobs in the job slots, will have a tuple for each entry in the job slots, with the tuple entries themselves representing the features listed in Table 4.1. For example, the job in Slot 1 was submitted at time 1, requested 5 time steps for 1 processor. When it was submitted, there were 0 jobs in the queue, with 0 work queued, and 1 free processor. Of the jobs still in the cluster, there were 6 units of work to be done when the job in Slot 1 was submitted and 0 jobs in the backlog. Concatenating all these numbers yields the tuple $(1, 5, 1, 0, 0, 1, 6, 0, 1)$ shown earlier, with the last entry representing the fact that the job in slot 1 *can* be scheduled now.

A side effect of using this new compact state representation is that, when H and W are fixed between different cluster configurations, learned features are directly transferable between clusters even when using function approximation methods that depend on a fixed number of features.

4.6.2 Sparse State Transitions

Another deficiency we've identified in the base MDP is that the agent sees *all* time-steps in the simulation, but this causes the agent to have to take an action even when there is no good action to take. Consider, for example, the case in which all resources are in use (there are no free resources). In cases such as this, any action the agent takes will lead to the same outcome: increasing the simulation clock, receiving negative rewards related to the slowdown of the jobs, and having *no* new jobs scheduled. This will be repeated for all time steps between the start of the last job that exhausted resources until the finish of the first job that frees them, causing non-negative rewards to be more sparse, making the reinforcement signal noisier and, therefore, harder to learn. The opposite is also true: if there are no jobs waiting to be scheduled, no matter what the agent chooses, the outcome will be the same: no jobs will be scheduled.

Due to that, we updated the environment to only call the agent and, therefore, to only add states, actions and rewards to a trajectory, when it was possible for the agent to take an

action that could result in a job being scheduled. In short, we change the transition function $\mathcal{T}(S_{t+1} \mid S_t, A_t)$ so that all state transitions from S_t to S_{t+1} will always have at least one job that may be scheduled by the agent in state S_{t+1}. We did not change the initial state, though, so $\rho = \{\llcorner\!\!\_\!\!\_\}$ still holds. The reader will notice this changes the MDP, as intermediate states, which would have non-zero rewards now have zero rewards. We fix this by introducing the options-based semi-MDP formulation later.

4.6.3 Reducing the noise of the reward signal

Based on the idea of only showing the agent what it can use to learn and act, we noticed that the reward signal could be further improved by, instead of computing the online slowdown of all jobs in the system $\mathcal{J}$, considering only the jobs that are in the waiting queue, and within the job slots window W: the jobs that can be directly influenced by the agent's actions. Therefore, we defined the set $\mathcal{W}$ that contains the subset of jobs from $\mathcal{J}$ that are within the window W, and the reward function became $\mathcal{R} = -\sum_{j \in \mathcal{W}} \frac{1}{t_e(j_i)}$ when the action taken doesn't schedule a job, and zero otherwise.

4.7 The options-based Semi-MDP formulation

Up to this point, we were operating at the MDP level, but it turns out we can use a semi-MDP formulation that works with the job scheduling problem and that is particularly useful to event-based systems (and simulations). The astute reader will notice that the sparse state transition model of Section 4.6.2 also supports event-based systems, so we have to discuss the difference between that model and the SMDP. The main difference between the MDP presented there and the semi-MDP presented here is that, there, the MDP dynamics had to be changed between the base, time-based MDP and the sparse transitions model, whereas, with the semi-MDP, we use the same underlying MDP we had used in the time-based case.

To aid in understanding, we now revisit Figure 2.8, redrawn in Figure 4.4, by showing what the rewards of intermediate states would be should a single job j with duration $t_e(j) = 3$ be scheduled in the base MDP, in the sparse transitions MDP and in the semi-MDP. As shown in the figure, the total reward for both the Semi-MDP and base MDP formulations is -1, whereas in the sparse transitions MDP, it is 0. These different rewards change the sparse transitions MDP into an MDP with a distinct objective than that of the base MDP, while the

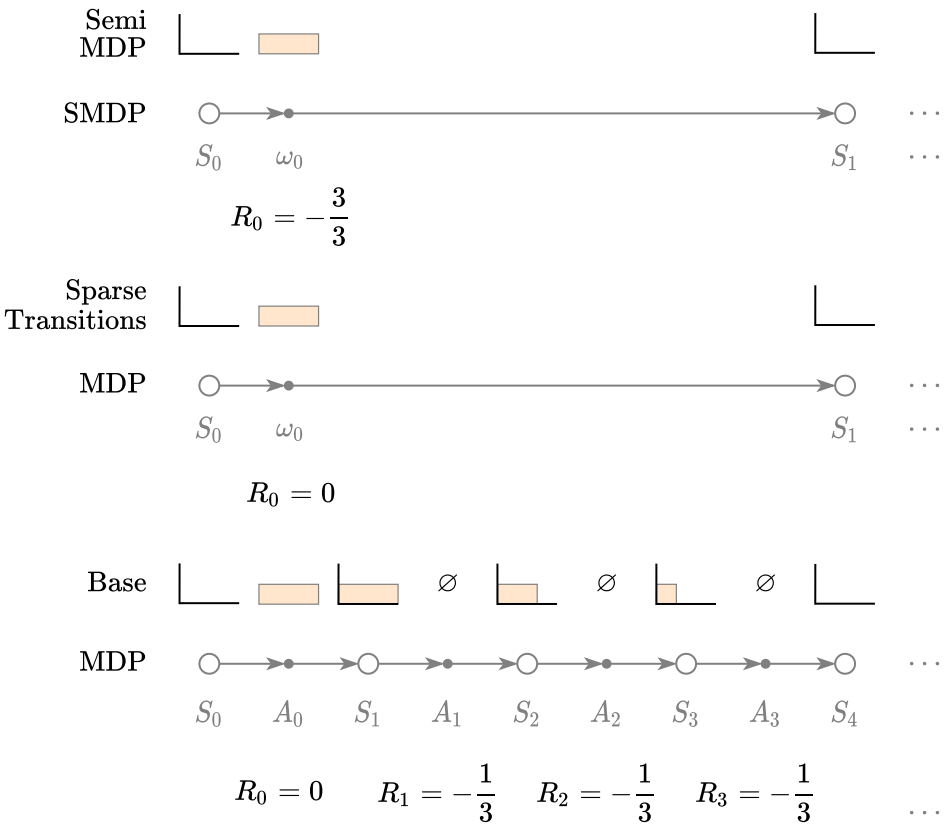


Figure 4.4: Difference in rewards in the base MDP, the sparse transitions MDP, and the semi-MDP.

Semi-MDP formulation still maintains that objective. This difference stems from eq. (4.1), in which $\mathcal{R}(s, a) = 0$ when a job is scheduled, which zeroes all intermediate rewards between two states in the sparse transitions MDP.

Now that we've made clear the distinction between the SMDP and the sparse transitions MDP, we make a complete description of the SMDP implementation in this dissertation, elaborating on the state representation mentioned in Section 4.6.1, summarized in Figure 4.5 (drawn considering a window of attention W of 2 jobs and a horizon of events H of 4 events), and that has three components:

The current allocation state of the cluster with the amount of processors allocated and amount of processors free. This component also includes, up to a horizon size H[6], the time offset of the next H events, with the allocation state of the cluster immediately after those events happen.

Description of jobs in the waiting queue up to a number of jobs in a window of attention of size W. The features used in this component are shown in Table 4.1.

[6]With the change from time-based to event-based systems underlying the MDP, we now call this the event horizon, as opposed to the time horizon, or simply the horizon.

Figure 4.5: Example of state representation construction from the current schedule in the simulator at time step $t = 3$. As shown in the figure, there are no new events after time offset 3 ($t = 6$). Since the event horizon $H = 4$ is larger than the number of events the agent expects to see, the remaining events are filled by copying the last state the agent expects to see, with the time updated. The representation the agent sees is the concatenation of all values in the bottom row of the figure (Agent observation at $t = 3$), namely: $\langle \underbrace{8, 2, 1, 2, 2, 4, 0, 0, 3, 2, 1, 2, 3, 3, 0, 4, 3, 0}_{\text{Cluster state features}}, \underbrace{3, 4, 1, 0, 0, 0, 0, 3, 2, 2, 1, 4, 0, 0}_{\text{Features of jobs in attention window}} \rangle$.

Summary statistics containing: the remaining work for running jobs, the next time at which there will be free processors in the cluster, with allocation state, the size of the backlog (number of jobs in the waiting queue outside of the window of attention), and the current size of the waiting queue.

For the cluster state, the first point to consider is that, since the agent is the scheduler, it has access to the events it expects to happen. For example, if at time step t the agent schedules a job that requests execution for 10 time-steps, assuming the job starts executing immediately, then the agent expects that, at time-step $t+10$, the resources used by that job will be freed. Hence, we are able to compute not only instantaneous features for the cluster, but also features of future states as expected by the agent. Even though the expected event times are not guaranteed to occur at the expected time (a job that requests 10 time steps can run for 5, for example), they help the agent in planning for the future. For each event in the horizon of size H, we represent the amount of processors in use, and the amount of free processors in the cluster. Additionally, we also represent the current size of the queue and the remaining work[7] in the cluster.

[7]The summation of the remaining execution time times the number of processors requested by each job

Table 4.2: Job features in the state representation for the option-based SMDP.

Feature	Symbol	Description
Submission time	$t_s(j)$	Time at which the job was submitted
Requested time	$t_r(j)$	Amount of time requested to execute the job
Requested processors	$r_p(j)$	Number of processors requested at job submission time
Queue size	$q_s(j)$	Number of jobs in the wait queue at job submission time
Queued work	$q_w(j)$	Amount of work that was in the queue at job submission time
Free processors	$f_p(j)$	Amount of free processors when the job was submitted
Can start now	$c(j)$	Whether job j fits the cluster to start at current time

For the job features, we compute them considering the state of the cluster *when the job was submitted*. So, for example, the queue size feature for a particular job is not updated when jobs enter or leave the system. The set of features we use is similar to the compact representation MDP, and are presented in Table 4.2. The "symbol" column in the table enables us to map the feature descriptions to their representations in Figure 4.5.

4.7.1 State transition function

In this section, we discuss how we defined the state transition function $\mathcal{T}(S_{t+1}|S_t, \omega)$. Recall from our discussion of SMDPs (§ 2.6) that we are using an event-driven simulation as our environment. In the discussion below, when we say time proceeds, we mean that the simulation proceeds normally, updating any queues and statistics without necessarily presenting intermediate states to the agent. The relevant state transitions are:

A new job arrives in a cluster with empty queues In this case, when transitioning from state s to S_{t+1}, time advances from the current time t to the submission time of $t_s(j)$ of job j. Since queues were empty before the arrival of j, S_{t+1} is now the representation of an empty cluster with a single job in queue.

The last running job finishes execution If new jobs arrive, the next state S_{t+1} is determined as in the previous case. Otherwise, if no more jobs arrive, this means the workload model has done generating jobs, and this is the end of the episode.

All cluster processors are in use If there are jobs in the queue, time will proceed until the number of free processors in the cluster is such that at least one job j in the queue fits in the cluster, at which point the agent may decide to schedule j, or to schedule no jobs (this will happen, for example, if the agent decides to leave room in the cluster for short jobs, and there

running in the cluster.

are no short jobs in the queue right now.) If there are *no* jobs in the queue, time will proceed until a new job arrives and at least one job in the queue fits in the cluster.

Some job in the queue fits the cluster If the agent decides to schedule a job that doesn't fit the cluster, or if it chooses to not schedule any jobs, time proceeds by one time step, with the state updated accordingly. If this time coincides with the arrival of a new set of jobs, those jobs enter the queue before the generation of the new state S_{t+1}.

Jobs arrive in the system following either one of the workload models present in `sched-rl-gym`, or following a trace of an existing machine.

4.7.2 Reward function

As discussed in Section 4.6.3, we use the negative we use the online slowdown of jobs in the window of attention to compute rewards. Notice that, when the selected job fills the cluster or is the only job in the queue, if such an action was taken at time step t, the next time step seen by the agent will *not* be $t + 1$ (§ 4.7.2). In such case, the reward for the action will be computed after termination of the option, with proper discounts for intermediate time steps, following eq. (2.18).

4.7.3 Learning algorithms and implementation

At first sight, the main difficulty with using an SMDP model as we do here is that there is a shortage of open-source learning algorithms and environments designed to work with SMDPs. In this work, we adapted the `sched-rl-gym` OpenAI Gym environment to support an SMDP formulation by implementing equation (2.18) as reward function. As for the learning agent, we used the Maskable PPO [Huang and Ontañón, 2022, Schulman et al., 2017] implementation of Stable Baselines 3 [Raffin et al., 2021], but to prevent "double" discounting in the policy gradient, we have to set the PPO γ parameter to 1, while leaving equation (2.18)'s γ set to 0.99 to perform discounting. The reason for such a change is that the value function for state s, $v_\pi(s)$, in equation (2.19) does not have an explicit discount factor γ, whereas in the definition of the value function for state S_{t+1} in an MDP there is an explicit discount factor.

PPO is an actor-critic algorithm (§ 2.5.1) that, as the name implies, has two compo-

nents: an actor, which implements policy $A_t \sim \pi_\theta$, and a critic, which estimates the advantage function of taking action A_t at state S_t.

The variables represented in the state of our environment have different ranges and orders of magnitude. To reduce the likelihood of our policy network having too large weights, it might help to normalize the inputs to the neural network. We normalize features by dividing them by their maximum possible value. For time-related features and for features that depend on time (such as work), we log-transform the data prior to normalization, a practice common in workload modeling [Lublin and Feitelson, 2003].

To support Maskable PPO, we've extended the OpenAI Gym environment to return, for each state, a boolean mask indicating which actions are valid for that state.

4.7.4 Workload models

The `sched-rl-gym` environment supports three different types of workloads: jobs generated by the Mao model (§ 2.3.1), jobs generated by the Lublin model (§ 2.3.2), and jobs loaded from workload traces in the Standard Workload Format (SWF). The environment also supports generating uncertain estimates with both the Gaussian (§ 2.4.1) and Tsafrir (§ 2.4.2) models. The workload to use in a given instance of the environment is passed as an argument to OpenAI Gym's environment construction function.

For the Lublin model, `sched-rl-gym` will generate parallel jobs with sizes ranging from 2 processors to up to the size of the cluster following a two-stage uniform distribution, and the run time of jobs depend on the number of nodes in the cluster, with that number feeding a parameter of the hyper-Gamma distribution from which job durations are sampled.

4.8 Summary

In this chapter we described a simulator for HPC job scheduling, then we described how it could be connected to an RL training and evaluation library, and described how to model such an environment as an MDP. Then, we described possible algorithms that can learn to optimize scheduling in this environment. Later, we described some limitations of the current model and presented an alternative representation for more efficient storage and learning, along with extensions to the model that may allow us to investigate other aspects, such as how generalizable are agents that learn in these environments.

Chapter 5

Experiments

Having presented the methods we used, we now evaluate the performance of our agents in the various MDP formulations. We will follow the same outline of the previous chapter. First, by evaluating the simulator's performance according to actual job traces and, then, by evaluating agent performance in each of the proposed MDPs.

5.1 Learning performance in the base MDP

We designed our first set of experiments to evaluate whether we were able to: (i) reproduce results from the literature [Mao et al., 2016], and to (ii) verify how state-of-the-art policy gradients algorithms would fare in the base MDP. In this section, we describe these experiments and discuss their results.

5.1.1 Performance of Policy Gradients methods and heuristics

We begin with an experiment to assess whether there is any difference in performance in how we updated the weights of the multiple workers of a policy gradient algorithm, and to contrast that performance with that of the classical algorithms SJF, FIFO, and Extensible Argonne Scheduling sYstem (EASY) backfilling. To do so, we instantiated an environment with the following configuration: episodes lasted for 50 simulation time steps, the agent's time horizon was set to 20, the cluster was configured to have two different resource types (processors and units of memory), and the agent had a window of attention of 10 job slots over the waiting queue, with a further backlog of 60 entries. For this set of experiments, we used the Mao workload model (Section 2.3.1), with new job probability set to 0.3, with jobs having long durations l 20% of the time. The duration of long jobs was sampled uniformly

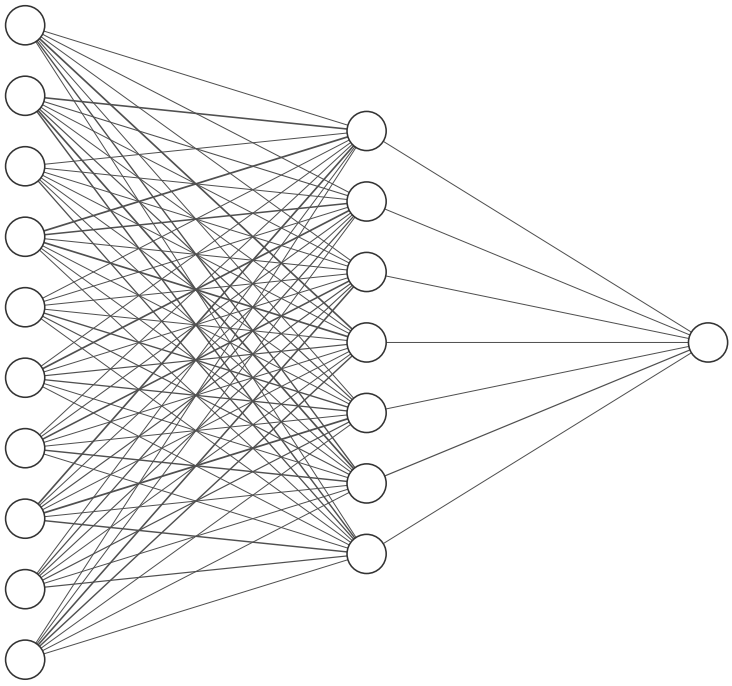


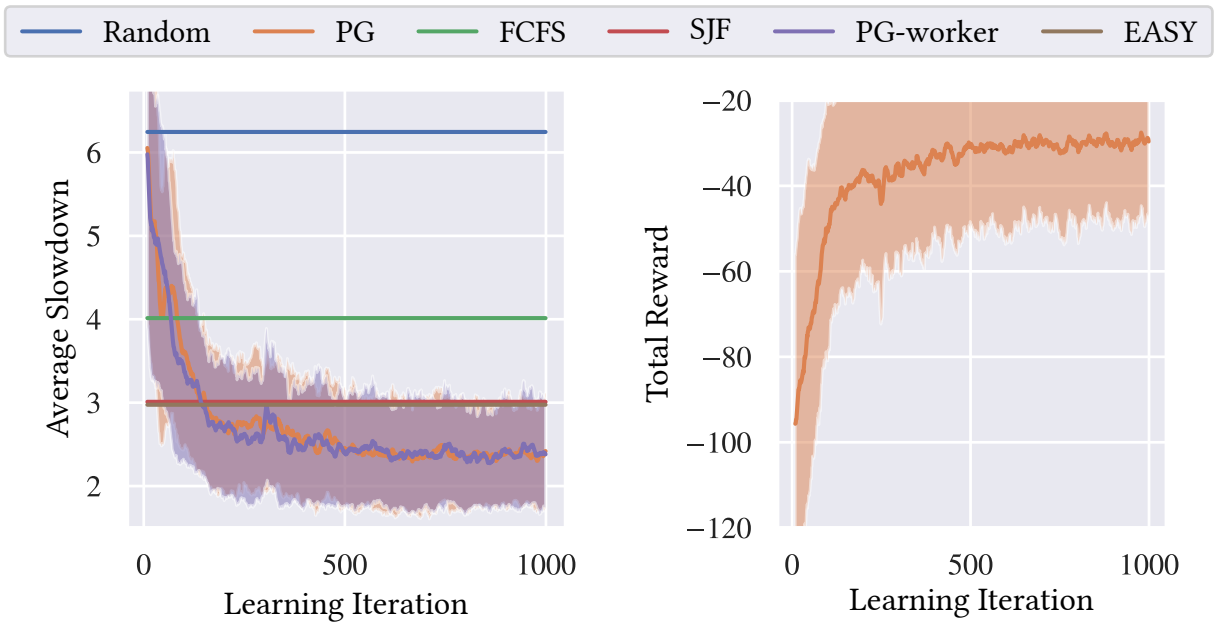
Figure 5.1: Schematic view of a three-layered neural network with 10 input nodes, a hidden layer of 7 neurons, and a single output layer. In this section, our neural network had 4480 input nodes, 20 hidden neurons, and a single output node.

from $l_1 = 10, l_2 = 15$, while the duration for short jobs was set to $s_1 = 1, s_2 = 3$. Jobs are CPU-dominant 50% of the time, and memory-dominant the other 50% of the time, with dominant resource usage was set to $d_1 = 5, d_2 = 10$, and non-dominant resource usage set to $n_1 = 1, n_2 = 2$.

We implemented the policy network as a PyTorch [Paszke et al., 2019] neural network module. Neural networks in PyTorch are defined by inheriting from the `torch.nn.Module` class and implementing the `forward` method. The `forward` method corresponds to the forward pass of the neural network and once it is performed, the automatic differentiation engine of PyTorch keeps track of operations to allow gradients to flow in the backward pass. For this experiment, we used a neural network with a single hidden layer of 20 neurons. A schematic representation of this type of fully-connected neural network (a multi-layer perceptron) is shown in Figure 5.1.

Optimization was performed using the RMSProp [Tieleman and Hinton, 2012] gradient-descent optimizer with learning-rate $\alpha = 0.001$, discount factor $\gamma = 0.99$, with 10 workers, each collecting 200 trajectories per "epoch" [1] prior to learning. Since we are using parallel workers, we implemented both weight update strategies mentioned in Section 2.5.1, naming the centralized weight update strategy "PG", and the distributed weight updates "PG-worker", with both implementations following the overall structure of Algorithm 1 using a fork-join model: the

[1] The term *epoch* comes from the supervised learning literature, and means doing a full pass over the training data to train a neural network. In this context, what we mean is that we perform a training cycle every time our agents collect 200 trajectories. After the weights are updated, this collected data is discarded.



(a) (b)

Figure 5.2: Learning curves of PG algorithms with an average cluster load of 70%. To reduce noise in the plot, signals were smoothed with a moving window of size 10. Curves were averaged over 60 independent trials. Shading represents one standard deviation around the average. (a) shows that both PG implementations (PG & PG-worker) outperform SJF at around 200 iterations, while (b) shows the total reward learning curve. The PG algorithm keeps improving until it plateaus at around -25 average total reward.

main process kept the canonical copy of the parameters in memory and spawned child processes which would sample trajectories. Once sampling was done, updates to the parameters were performed in a shared-memory copy of the parameters. PyTorch facilitates this style of programming by providing a multiprocessing module, responsible for spawning workers, and automatic shared memory of parameters. In the centralized version, we centralized trajectories in the main process before computing gradients. In the decentralized version, each worker computed their own baselines and only propagated gradients to the main process. Each iteration of this version is faster than the centralized one because there is no need synchronize all workers to apply updates.

Figure 5.2 shows the learning curves of the various agents. To reduce noise, the curves were smoothed with a moving average with window size 10. In the figure, PG is the version of the agent with centralized baseline calculation, while PG-worker uses HOGWILD!-style learning. We can see that both PG implementations start outperforming SJF at arount iteration 200. From the behavior shown in Figure 5.2a, we see that learning performance for both update schedules for the Policy Gradient are very similar (in the orange and purple curves), converging at roughly the same rate, and to the same average slowdown value of around 2.5. The straight lines in Figure 5.2a represent the average performance of classical algorithms, and are straight because that graph shows performance as a function of learning iteration and, to

which these classical algorithms are independent.

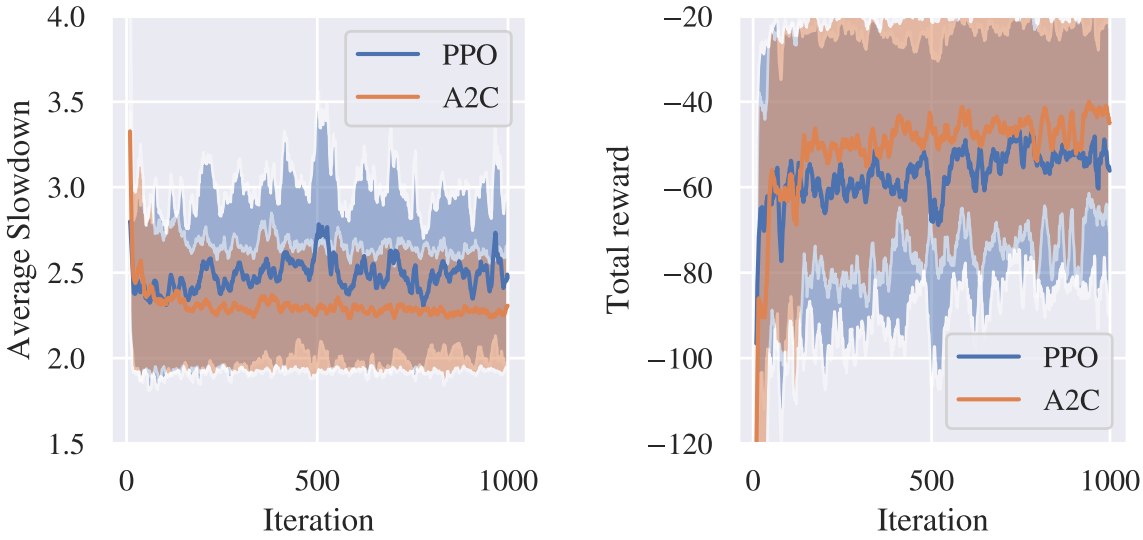
We also evaluated slowdown learning curves, which exhibited similar behavior to those of previous research. This complete integration test gives us confidence that the environment is sound, and enabled us to replicate results from the literature [Mao et al., 2016]. The fact that we were able to reproduce results from the literature should be emphasized. Reproducing existing work and judging their improvements is vital to sustaining progress not only in RL as an area, but in areas that can make use of it. As Henderson et al. [2018] have shown, the dynamics of environments help determine what algorithms are more successful. Therefore, if we are to apply RL to resource management and be able to translate research progress to practical situations, we need stable, sound, and representative environments.

5.1.2 State-of-the-art algorithms in the base MDP

We're now set to evaluate other algorithms in the same environment. Of note is the ease of use of the environment with packages that target the OpenAI Gym toolkit. While our implementation of the PG algorithm plus support code for training and persistence took 212 lines of code disregarding comments and empty lines, our implementation of A2C and PPO, with support code, took 44 lines, a reduction since the core of the algorithm came from the Stable Baselines [Raffin et al., 2021] implementation. Being actor-critic algorithms that *learn* the state-dependent baseline value, both A2C and PPO have, in addition to the policy network, a value network, used for approximating a state's value. Both the value and policy networks used in this experiment had identical topology of the neural network of the basic PG algorithm.

To keep agents comparable, we configured the agents to execute the same number of steps for learning as a PG worker would have used. Recall that, in our experiments, we had 10 agents collecting 200 trajectories for 50 timesteps. Here, we had *a single agent* interacting with the environment for $200 \times 50 = 10,000$ time steps. Therefore, the A2C and PPO agents performed roughly 10× less interactions per learning epoch with the environment than the base PG implementation.

Figure 5.3 shows the learning curves of A2C and PPO. As can be seen from the figure (5.3a), average slowdown performance from PPO and A2C is more stable than PG, learning rapidly at the beginning, crossing the 2.5 average slowdown value, and decreasing slowly from there. Upon inspection of the behavior of the agents in our environment, we noticed the agents learned quickly to output a fixed action corresponding to the first positions of the waiting queue, which was fixed by configuring an entropy bonus hyperparameter. This made slowdown performance better on average than PG, and PG-worker, which exhibited closer to random behavior in early iterations. Even though average performance was better



(a) (b)

Figure 5.3: Learning curves of the A2C and PPO agents with an average cluster load of 70%. (a) shows average slowdown learning curves, whereas (b) shows total rewards learning curves. The figures show the A2C agent converges fast to its final performance.

in early iterations, those iterations had higher variance than later ones, which exhibited more purposeful behavior.

Both A2C and PPO have a regularization term that improves exploration by discouraging premature convergence to suboptimal deterministic policies. This regularization term is a function of the entropy of policy π_θ [Williams and Peng, 1991], and we noticed that when it was zero, the algorithms did get stuck in local minima of deterministic policies. Both the A2C and PPO implementations from stable baselines that we used come with this parameter set to 0.01. Apart from this hyperparameter, we did not tune any other parameters, leaving them at their default values, which are summarized in Table 5.1. Entries with "—" indicate the algorithm does not have that parameter, so it was not set.

Figure 5.3b shows that total reward follows the same trend of Figure 5.2b and indicates learning is improving. Deeper inspection shows that the maximum total reward of PG is $\approx$ 20% better than that of A2C, but mean total reward is only $\approx$ 10% better, indicating A2C converges faster than PG. With regards to slowdown (Figs. 5.2a, 5.3a), A2C's minimum slowdown was $\approx$ 9% better than PG's, with A2C's mean slowdown over all training iterations $\approx$ 19% better than PG's.

In short, from these experiments we can see that, although not outperformed easily, algorithms from the literature that have no knowledge of the inner workings of the environment were able to achieve performance comparable to that of PG's. We see the benefits of following the OpenAI Gym interface by being able to solve our environment with standard implementations of algorithms, which facilitates the exploration of new environment ideas.

Table 5.1: Hyperparameters for experiments comparing with the base PG algorithm.

Hyperparameter	Value	
	A2C	PPO
Discount factor (γ)	0.99	0.99
Learning rate	1×10^{-3}	1×10^{-3}
n steps	5	50
Entropy coefficient	10^{-1}	10^{-2}
GAE λ	1.0	0.95
Value coefficient	0.5	0.5
Clipping ϵ	—	0.2
Surrogate epochs	—	10
Surrogate Batch size	—	64
RMSPROP ϵ	1×10^{-5}	—
RMSPROP α	0.99	—

5.1.3 Section Summary

In this section, we've shown that our proposed OpenAI Gym environment and our PG algorithm implementations reproduce results from the literature, giving additional confidence that our algorithms can be used for research of HPC job scheduling algorithms and be comparable with other approaches.

We've also shown that our decision to follow the OpenAI Gym interface enabled us to implement additional agents and assess their performance in our environment with ease. We've also shown that, in doing so, we were able to optimize for the proposed environment using 10× less interactions (and, hence, computation) with the environment. Additionally to these smaller amounts of interactions, we have also shown that the A2C and PPO algorithms achieve the same performance in the environment of that of the PG algorithms quite early in the learning process.

5.2 Solving the alternative MDP formulations

In this Section, we evaluate a single algorithm (PPO) in different MDP formulations with the objective of observing how its performance is affected under each different formulation. For these experiments, we decided to use the PPO implementation of `stable-baselines3`. We also fixed the neural network architecture used for function approximation,

consisting of a two-layer neural network with 64 units in each layer, and with parameter sharing between policy and value networks. The fixed number of units implies the image-like representation will use more parameters, as it contains more data than the compact representation. The hyper-parameters used for training the agent are summarized in Table 5.2. Apart from slight tuning the learning rate and entropy bonus coefficient for better learning in the base MDP, we performed no hyper-parameter optimization, and used values found in the literature when training the image-like agent. For a full description of PPO, we direct the reader to Schulman et al. [Schulman et al., 2017].

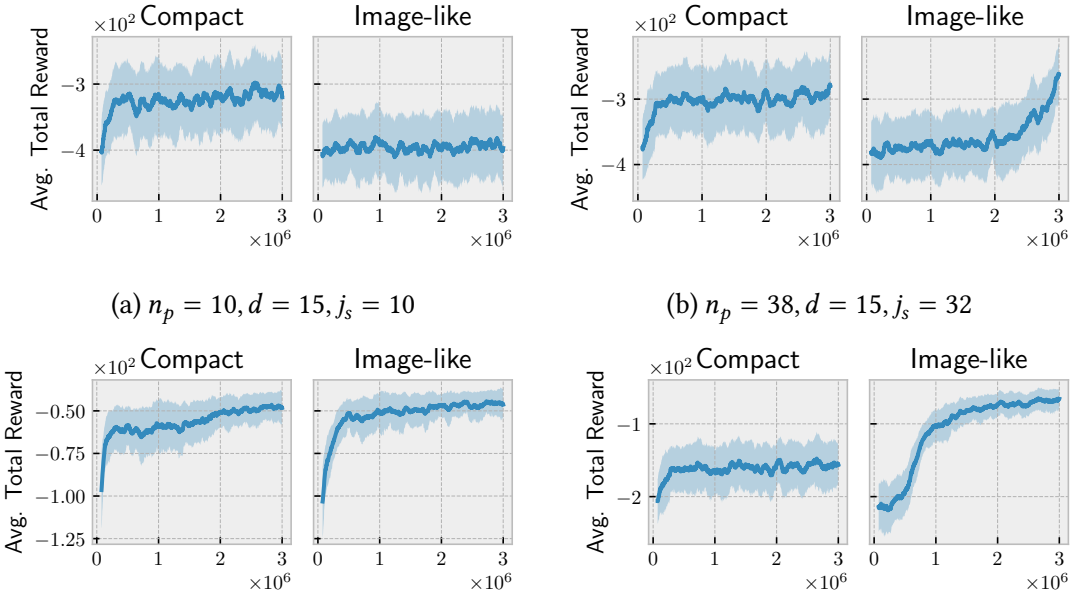
We also maintained the environment specification fixed for all agent evaluations and used $W = 10$ job slots, with simulations of length $T = 100$ time-steps and time horizon $H \in \{20, 60\}$. These two horizon values enable us to contrast cases in which agents can see when jobs will complete, or not. For this case, we also used the Mao workload model, which submitted a new job with 30% chance on each time step. Of these, a job had 80% chance of being a "small" job, and "large" otherwise. The number of processors n_p was chosen in the set {10, 32, 64}, while the maximum job length (duration) d varied from {15, 33, 48} and the size of the largest job (number of processors) j_s came from the set {10, 32, 64}. In the workload generator, the length of small jobs was sampled uniformly from $[1, {}^{d}/_{5}]$, and the length of large jobs was sampled uniformly from $[{}^{2d}/_{3}, d]$. The number of processors used by any job was sampled from $[{}^{n_p}/_{2}, n_p]$.

Table 5.2: List of hyper-parameters used when training PPO agents.

Hyper-parameter	Value
Learning rate	10^{-4}
n steps	50
batch size	64
Entropy coefficient	10^{-2}
GAE λ	0.95
Clipping ϵ	0.2
Surrogate epochs	10
γ	0.99
Value coefficient	0.5

All agents were trained for three million time-steps as perceived by the agent. This means that all agents will see the same number of states, and will take the same number of actions, but the number of time steps in the underlying simulation will vary, due to the event-based case becoming a semi-MDP. We evaluated agents with a thousand independent trials, reporting average values.

In Figure 5.4 we show a sampling of learning curves comparing the learning performance of agents that were trained using the image-like representation and the compact representation with rewards computed from all jobs. The compact representation converges faster than the image-like representation, probably due to its smaller number of parameters. We also notice that although convergence is faster, the compact representation is not necessarily better (Figure 5.4c, 5.4d). There doesn't seem to be a general rule, but we noticed that when jobs are shorter (the d parameter is smaller), the compact representation dominates (Figure 5.4a, 5.4b). When d increases and most jobs use few processors ($j_s \ll n_p$), the compact representation tends to have comparable performance with the image-like representation (Figure 5.4c), whereas when jobs use many processors *and* have a longer duration, agents using the image-like representation learn the environment better (Figure 5.4d). For this set of exper-



(a) $n_p = 10, d = 15, j_s = 10$

(b) $n_p = 38, d = 15, j_s = 32$

(c) $n_p = 38, d = 48, j_s = 10$

(d) $n_p = 64, d = 33, j_s = 32$

Figure 5.4: Learning curves for various scenarios with $H = 20$ contrasting learning using a compact representation with learning with an image-like representation. Curves are an average of six agents, with shaded areas representing one standard deviation, and show a moving average of total rewards received by the agents during training.

iments, the size of the time horizon (H) doesn't impact the learning performance, as curves obtained with $H = 60$ (not shown) are indistinguishable from visual inspection to the ones obtained with $H = 20$. When evaluating agents, we performed t-tests to check whether there was a difference in agent performance when using these different H values. In other words, the null hypothesis was that performance was equal, and the alternative hypothesis was that agent performance varied. In this setting, the null hypothesis was rejected only 36.6% of the time when considering p-values $\leq 1\%$.

When evaluating agents after one million iterations, scheduling performance was similar between agents when the maximum number of processors used by jobs was smaller (which implies less parallelism). Given job submission rates in all environments was the same, clusters were less busy in these situations: as long as jobs are scheduled, there shouldn't be significant differences in average slowdown, due to smaller queues.

In Figure 5.5, with key to scenarios shown in Table 5.3, we show average slowdown of the agents for the scenarios in which there was some variability in performance between agents. From the figure, we see that, apart from scenarios 2 and 6, agent performance in the "Compact + Sparse + Reduced" MDP is not worse than that of the image-like MDP. Of these two, only the difference for scenario 6 is statistically significant, with p-value $\leq 5\%$ when performing a t-test with alternative hypothesis of different distributions. For the cases where "Compact + Sparse + Reduced" agents are better, the results are statistically significant (p-value $\leq 5\%$) in scenarios 3, 4, 5, 7, and 9. Scenarios 2 and 6 are interesting, since they were

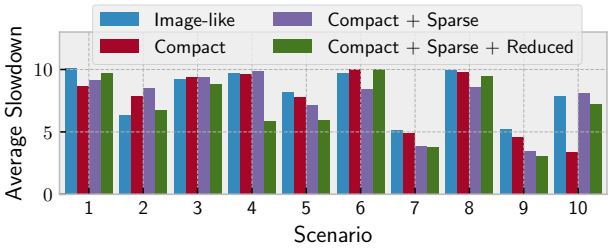


Figure 5.5: Average slowdown for the various scenarios considered. Each bar represents a different instantiation of the (semi-)MDPs. Average slowdowns were computed by averaging the slowdown of a thousand independent trials for each agent in each scenario. All agents were evaluated with same workload and random seed. In the legend, *image-like* corresponds to the base MDP, *compact* to the compact representation, *sparse* to the sparse state transitions, and *reduced* to the reduced set of jobs to compute rewards.

Table 5.3: Key to the scenarios presented in Figure 5.5. *Procs.* refers to the number of processors in the cluster, *Max Length* refers to the maximum job length, and *Max Size* refers to the maximum number of processors used by jobs.

Scenario	Procs.	Max Length	Max Size
1	10	15	10
2	10	48	10
3	38	15	32
4	38	33	32
5	38	48	32
6	64	15	64
7	64	33	32
8	64	33	64
9	64	48	32
10	64	48	64

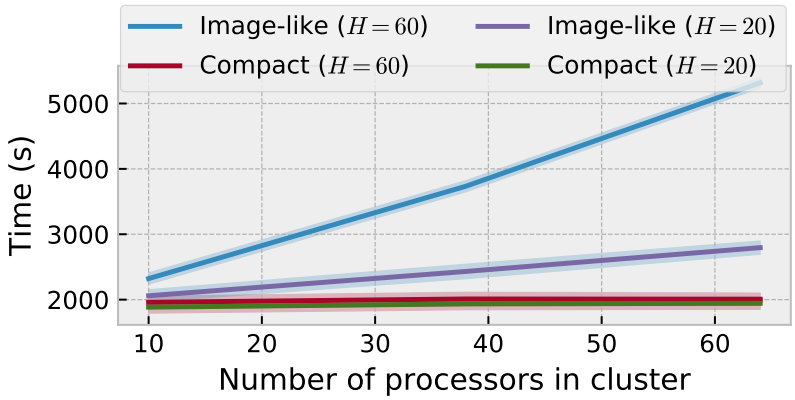


Figure 5.6: Time needed to train agents for three million iterations. The shaded area represents one standard deviation. Increasing the time horizon hardly affects the training time of compact agents, while causing the training times of agents that use an image-like representation to scale linearly with time horizon.

configured to have shorter jobs of at most 15 time-steps, with scenario 1 having 10 processors, and scenario 6 having 64, both with jobs with the potential of using all cluster resources.

In Figure 5.6 we contrast the training times for the various agents. As can be seen, training times for agents based on the base MDP are highly variable, due to the fact that different MDP configurations result in different sizes of state representations, which impacts training performance. As an example, the image-like agent requires 301068, 1089548, and 1821708 parameters for the scenarios with 10, 38, and 64 processors, while all compact agents require a fixed number of parameters: 24332. Times were measured in a Linux 5.10.42 desktop with an NVIDIA GTX 1070 GPU and an i7–8700K processor using the *performance* CPU frequency-scaling governor.

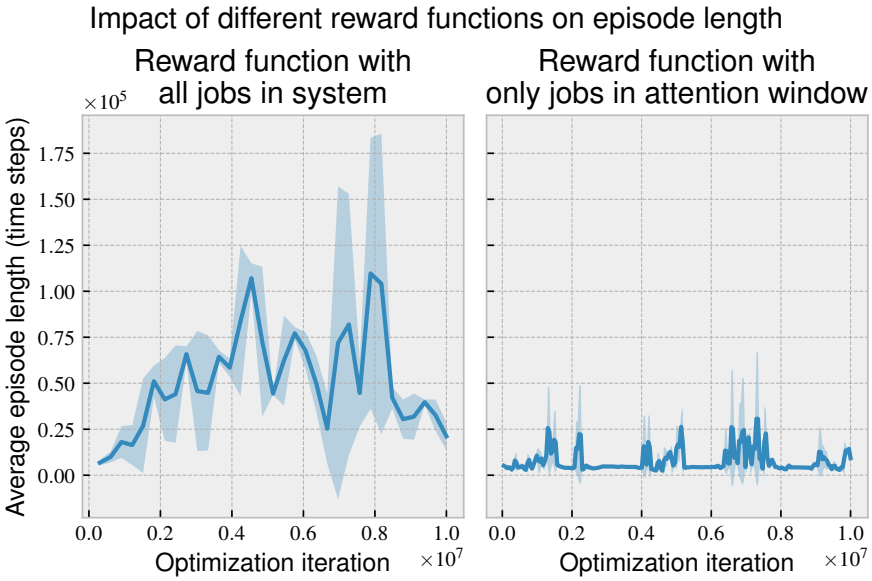


Figure 5.7: Difference in episode length given reward computation method, with all parameters being equal (lower is better). On the left, we see the average episode length when using all jobs in the system to compute reward, whereas, on the right, we only consider jobs which the agent can affect. The curves were computed using the average of six parallel agents, with the shaded area representing one standard deviation. Both agents were trained in clusters with 128 processors and with the Lublin [Lublin and Feitelson, 2003] workload generator generating 512 jobs per episode.

The experiments described previously did not highlight the impact of using a reduced set of jobs for computing the agent's reward. After analysis of the results, we concluded that the event-based simulations using the Mao model were too short to show any meaningful difference between reward computation methods. In Figure 5.7, we show the difference in mean episode length when using all jobs in the system to compute rewards, versus using only the jobs upon which the agent can act, when using longer simulations with the Lublin workload model, which generated workloads corresponding to weeks of work, as opposed to seconds. As can be seen in the figure, computing rewards using all jobs makes learning unstable, whereas using a reduced set of jobs constrains learning to a region in which it trades off performance in the task, with entropy in its action distribution. Moreover, the worst training "performance"[2] achieved by our agent is better than the best performance of the method found in the literature [Mao et al., 2016]. To generate the figure, we used the same hyperparameters for the learning algorithm as used in the other experiments in this subsection. For the Lublin workload generator, we considered clusters with 128 processors, sampling 512 jobs per episode.

The compact MDPs proposed in this work all have the characteristic of having a state

[2]Episode length is usually not a performance measure but, in this environment, it acts as a proxy for the quality of the generated schedules: worse agents will tend to see more states, increasing episode length. This is due to the nature of the environment, as the agent will see a new state whenever there is a new event, or *as long as there are still jobs in the queue that fits the cluster.*

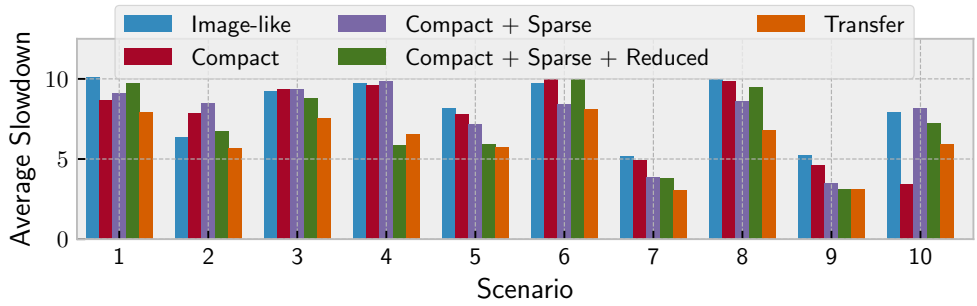


Figure 5.8: Bar chart contrasting the performance of a transferred agent to agents trained specifically in their environments.

representation with a fixed size, which allows for *transfer* of learned weights between clusters. Here, we consider as transfer the ability to change cluster configuration without the need for retraining an agent from scratch, which is simply not possible when using the image-like representation. In Figure 5.8, for example, we show the performance of an agent trained in the bounded reward, event-based, compact MDP with 64 processors and with jobs of length 33 (the best agent in Figure 5.5, corresponding to scenario 9) evaluated in a compact environment *without* event-based updates.

The results displayed in Figure 5.8 show that, with a single agent, we were able to evaluate performance in all different scenarios without the need for retraining. We see that, for the most part, slowdown is kept low, and not only that: this agent outperformed other agents in 80% of scenarios (differences are statistically significant, with p-value $\leq 1\%$, except for scenario 9, since this is the same agent, and scenario 5, where the test has low power to reject the null hypothesis). This highlights the advantage of using a representation that allows for easy transfer between agents, enabling good performance in a variety of cluster settings. With this experiment, we see the full set of modifications proposed to the base MDP yield an agent more capable of dealing with different scenarios, and we kept these modifications for other experiments.

5.2.1 Mao workload model with Gaussian uncertainty

Before performing more complex simulations, it may be worth evaluating how our agents fare with Gaussian noise in the job runtime estimates of the Mao workload model. In this set of experiments, we also considered two different environments, "long" and "short", but their hyperparameters are different from the ones used previously, so their configuration is displayed on Table 5.4. In this model, it's hard to find a set of parameters that don't deliver an overly full, or an empty, cluster. Moreover, the reader will notice that times in the long environment are 10× larger than those of the short environment, with probability p of sampling

Table 5.4: Environment and workload model parameters used in the Mao-Gaussian set of experiments.

	Value	
Parameter	Short	Long
Time limit	100	1000
Job rate (p)	0.3	0.03
Long job lower bound (l_1)	32	320
Long job upper bound (l_2)	48	480
Short job lower bound (s_1)	1	1
Short job upper bound (s_2)	9	90
Smallest job size (r_1)	16	
Largest job size (r_2)	32	
Backlog size (B)	60	
Window of attention (W)	10	
Time horizon (H)	20	
Long job chance (l)	0.2	

jobs 10× smaller.

For both environments, we varied the ν factor of the Gaussian model (§ 2.3.2) from 0 (perfect estimates) to 2 (two standard deviations for noise), with both the overestimated and underestimated Gaussian noise models.

Although unrealistic in practice, for this experiment, we decided to evaluate the impact on the schedule if under-estimated jobs weren't terminated by the scheduler: run time estimates were merely used as "hints" to guide scheduling decisions. So, for each environment, we compared the performance of a baseline model, trained with accurate estimates, with models that used the overestimated and underestimated run time estimates.

The results are presented in Figure 5.9. In the figure, we see that, as expected, as the noise factor ν increases, scheduling performance decreases (average slowdown increases). Also as expected, the models trained with noisy estimates (red and purple bars) tend to outperform the baseline model (blue bars). The models trained with underestimated noisy estimates outperform others in the short environment, whereas, in the long environment, we don't see any strong trends.

In the overestimated environments, there will be more "holes" in the schedule, due to the excess time between actual and estimated run times. This should make these scenarios easier than the underestimated scenarios without job termination when time was exceeded, due to the more "packed" nature of the schedule, giving less room for error for optimization of the learning algorithm. We believe this is the main reason for better performance in the short environment. In the long environment, due to the lower probability of arrivals, the cluster tends to be more free, reducing the impact of bad decisions and, hence, the smaller difference between algorithm performance there.

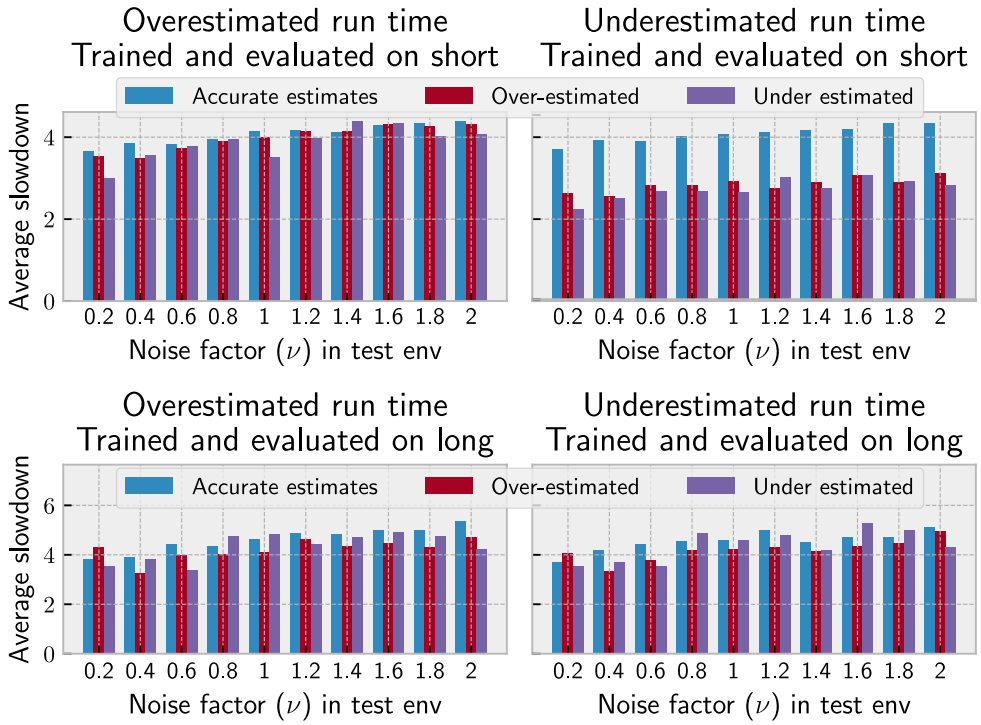


Figure 5.9: Performance with the Mao workload model when using different noise factors with the Gaussian uncertainty model. At the top, we show performance when evaluating agents in the "short" environment, while at the bottom we show the performance of the agents when evaluated in the "long" environment. The different bar colors show the situation under which the models were trained, whereas each graph shows the environment on which the agent was evaluated. "Overestimated" means the Gaussian overestimated model was used, while "underestimated" means the Gaussian underestimated model was used.

5.2.2 Section Summary

In this section, we analyzed the effects learning performance given different MDP design decisions. In particular, we experimented with resource management agents for job scheduling in computing clusters, discussing cases in which a compact representation outperforms an image-like one, and vice-versa. We saw that the compact, feature-based environments support transferring the weights of learned agents between different cluster settings, while also keeping agent memory consumption constant, and processing requirements stable. We also saw that agents that use these compact representations perform no worse than image-like ones, and, thus, might be preferable to those agents, especially when constant memory usage is a requirement. Moreover, our results indicate that transferred agents may outperform specialized agents in 80% of the tested scenarios without the need for retraining.

A remarkable result on performance during learning was obtained when we considered the length of the learning rollouts given the function used to select jobs to compute agent reward. By limiting reward computation to only jobs that are under the agent's control, we

saw rollouts became shorter, indicating learning performance was better, given the dynamics of the event-based MDP on which the evaluation was done.

We also started investigating the effects of Gaussian noise in job runtime estimates. Our results indicate that, if schedulers didn't terminate jobs that underestimated their run times, agents trained with noisy estimates would perform much better than models that relied on accurate estimates. Apart from that result, a Gaussian uncertainty model with the Mao workload model does not allow us to extract more insights from those experiments.

5.3 Performance in the option-based SMDP model

For implementing the experiments described in this section, we used the `smdp` branch of the `sched-rl-gym` package, with the `CompactRM-v0` OpenAI Gym environment. We performed three sets of experiments in increasing order of complexity, described in the next sections. Except where explicitly mentioned, we measured final model performance using the average slowdown (2.1) of the simulation.

5.3.1 Comparison with models from the literature

Here, we want to assess how an agent that learns in the option-based SMDP model proposed in this work compares with another from the literature. To answer that question, we trained a model using the Lublin workload generator for a 256 processor cluster. We then evaluated that model using the same methodology proposed by Zhang et al. [2020]: 10 independent evaluations of random samples of job traces with 1024 jobs in each dataset. To ensure fairness in the comparison, we downloaded and replicated the sampling code of RLscheduler, abbreviated to RLS in the following discussion, by Zhang et al. [2020], such that we used the same offsets in the jobs traces as they used in their evaluation.

All the models in this and following sections were trained in an episodic manner, with an episode ending every time 256 jobs were successfully scheduled. Between episodes, the system state was reset to the empty cluster ($\rho = \{\llcorner\!\!\lrcorner\}$). All sets of 256 jobs were generated by the Lublin workload model, with different random seeds, such that each episode is different from the other. The agents were trained with Maskable PPO for a million events, which translates to training for little more than one hour for each setting on a Core i7-8700K Desktop running Arch Linux with kernel version 5.16.2, with the *performance* CPU frequency governor and

Table 5.5: Environment, workload model, and learning parameters used in the experiments that used the Lublin workload model.

Surrogate epochs	10
n steps	50
batch size	64
Clipping ϵ	0.2
Value coefficient	0.5
GAE λ	0.95
Discount factor γ	0.99
Entropy coefficient	10^{-4}
Learning rate	Linear decay $\left[3 \times 10^{-4}, 10^{-5}\right]$
Total iterations	10^6
Event horizon size	60
Window of attention size	128

Table 5.6: Comparison of the average bounded slowdown achieved by our SMDP model versus the ones reported by RLScheduler [Zhang et al., 2020]. For the SMDP model, we report the average bounded slowdown and standard deviation after the ± symbol. Lower bounded slowdown is better. In the table, "RLS" stands for the base RLScheduler model, whereas "RLSB" stands for the RLScheduler model with backfilling. In this experiment, all SMDP models were trained with the Lublin workload model. Values in the Trace column describe the trace file used for evaluation.

Trace	RLS	RLSB	SMDP (Ours)
Lublin-1	254.67	58.64	67.55 ± 20.21
Lublin-2	724.51	118.79	225.58 ± 124.46
HPC2N	117.01	86.14	59.77 ± 87.45
SDSC-SP2	466.44	397.82	121.62 ± 137.81

an NVIDIA GeForce GTX1070 GPU. For training, we used Maskable PPO with learning and environment parameters from Table 5.5. In these experiments, we trained a two-layer neural network with 256 units in the first layer and 128 in the second one, with no parameter sharing between value and policy networks, and the rectified linear unit activation function.

Table 5.6 shows a comparison of average bounded slowdown achieved by our model and compared with RLScheduler. For our model, we report both the average value, and standard deviation. For the other values, we only report the mean, as we only reproduce what was reported by Zhang et al. [2020]. We also evaluated the average resource utilization for the same datasets, and report the results in Table 5.7. The reader should be aware that while we compare performance in the four datasets of tables 5.6–5.7, our model had never seen any of that data before evaluation.

From the data, we see that our SMDP model outperforms the base RLScheduler model (RLS) in most cases, while not necessarily doing so against the backfilling variant of RLScheduler (RLSB). We attribute this from the fact that despite our model not using backfilling explicitly,

Table 5.7: Comparison of the average resource utilization achieved by our SMDP model versus the ones reported by RLscheduler [Zhang et al., 2020]. For the SMDP model, we report the average utilization and standard deviation after the ± symbol. Higher utilization is better. In the table, "RLS" stands for the base RLscheduler model, whereas "RLSB" stands for the RLscheduler model with backfilling. In this experiment, all SMDP models were trained with the Lublin workload model and trained to optimize for slowdown. Values in the Trace column describe the trace file used for evaluation.

Trace	RLS	RLSB	SMDP (Ours)
Lublin-1	0.714	0.850	0.795 ± 0.025
Lublin-2	0.562	0.593	0.647 ± 0.078
HPC2N	0.640	0.642	0.593 ± 0.163
SDSC-SP2	0.671	0.707	0.770 ± 0.061

Table 5.8: Workload traces used in the evaluation with real traces. The trace column represents the name of the log in the Parallel Workloads Archive, with the version column representing the version used. The processors column corresponds to the number of processors in the machine, while the model column tells the number of processors that were used in training our models.

Trace	Version	Processors	Model
ANL-Intrepid [Tang et al., 2011]	1	163840	16384
CTC-SP2 [Hotovy, 1996]	3.1-cln	338	256
HPC2N	2.2-cln	240	256
SDSC-SP2	4.2-cln	128	128
SDSC-BLUE	4.2-cln	1152	2048

the use of an *event horizon* with events the agent expects to happen allows it to perform better planning ahead. Backfilling benefits RLscheduler because it uses a kernel to compute the priority of individual jobs in the queue, making local decisions, whereas backfilling gives a more global view of the schedule to the algorithm.

In the cases where our model has worse performance than the RLSB model (Lublin-1 & Lublin-2), we performed a t-test to check whether of our average bounded slowdown observations when considering the RLSB values as a population mean. Hence, the null hypothesis is that our distributions are the same, whereas the alternative hypothesis is that they are not. For the Lublin-1 case, we fail to reject the null hypothesis, but in the Lublin-2 case, we do reject it (p-value < 0.05). For the other two datasets (HPC2N and SDSC-SP2), we reject the null hypothesis (p-value < 0.05).

A remarkable difference between the SMDP model and RLS is on how performance of models trained on a given trace degraded when applied to another trace. For example, when RLS was trained on datasets other than Lublin-1 [see Zhang et al., 2020, Table VII], its performance degraded by ≈ 89% (SDSC-SP2), 11% (HPC2N), and 31% (Lublin-2). We attribute this difference to the local focus of the kernel layer RLS uses, which contrasts with the more

global approach of the SMDP model.

In the original HPC2N and SDSC-SP2 traces, average bounded slowdown for the same jobs we used in our evaluation are 156.12 ± 173.42 and 202.65 ± 215.75 respectively, and due to the high variance of the average bounded slowdowns of the original schedules, when using Welch's t-test, we fail to reject the null hypothesis that the average bounded slowdows of the schedules of the learned agent *versus* those of the original traces have different means. Still, the schedule generated by a learning agent has much lower variance than the original schedule, making it a better choice for a more stable behavior. Concerning the utilization results in Table 5.7, they are interesting because our SMDP model was trained to minimize bounded slowdown, not utilization, and, despite that, it achieves good performance when compared against models that were explicitly trained to maximize utilization, outperforming them in 3 out of 4 cases.

Another difference in the data is that both in the Lublin-1 and Lublin-2 datasets, run time estimates are missing. What both RLscheduler and our model do is assume accurate run time estimates, while in the HPC2N and SDSC-SP2 datasets, actual user run time estimates are present. This should tend to make models that rely more heavily on accurate estimates to have better performance in the Lublin-1 and Lublin-2 datasets, and worse performance in the datasets with actual user run time estimates. Given that our model uses global cluster state features as well as job features, this would explain why RLscheduler performs better in the Lublin-1 and Lublin-2 datasets: RLscheduler probably gives more weight to the user run time estimates, having its performance degraded as the quality of estimates degrade.

5.3.2 Effects of noisy estimates with synthetic workload models

The models from the previous section were trained with accurate run time estimates, and evaluated with both accurate estimates (Lublin-1 and Lublin-2), and actual user run time estimates (HPC2N and SDSC-SP2). Now that we've established in the previous section that the performance of our SMDP model is at least comparable with, and arguably better than, a state-of-the-art model, we evaluate how the SMDP model fares under different synthetic run time estimate models. We trained models in clusters of 128, 256, 512, 1024, 2048, and 16384 processors with accurate run time estimates, with the Gaussian model (with ν varying from 0.5 to 2.0), and with the Tsafrir run time estimate model. Table 5.9 summarizes the results of the evaluation of the aforementioned models under the same settings. From the values shown, we don't see a trend in any direction, and the average bounded slowdown obtained with models trained with inaccurate user run time estimates are not different from the models with accurate run time estimates in a significant manner. This is different from what we

expected, as we expected that models trained with uncertain run time estimates would perform better than models trained on accurate estimates. Perhaps, due to the way we build the state representation, the model does not rely that much on the run time estimate feature itself. In hindsight, although adding Gaussian noise may help in situations with small amounts of data, due to our use of a simulator and the Gaussian noise having zero mean, it might have been the case that the effects of the noise averaged out, not affecting much the performance of the system.

Table 5.9: Average bounded slowdown of models evaluated with the Lublin workload model and with run time estimates generated by the Tsafrir model. After performing a t-test between the models trained with accurate estimates and the models trained with inaccurate estimates, we didn't find any statistically-significant (p-value < 0.05) differences between models. All models were evaluated with the same random seeds and we report the average and standard deviation of 10 independent evaluations.

	Processors					
Uncertainty model	128	256	512	1024	2048	16384
Accurate estimates	17.9 ± 19.0	22.8 ± 9.48	57.1 ± 43.7	20.7 ± 30.6	33.5 ± 47.0	32.7 ± 35.9
$\nu = 0.5$	18.4 ± 19.7	23.2 ± 9.35	57.9 ± 44.5	36.0 ± 75.6	24.8 ± 33.1	32.8 ± 34.4
$\nu = 1.0$	16.0 ± 18.4	25.3 ± 13.5	60.9 ± 46.8	20.5 ± 31.0	24.5 ± 24.1	32.4 ± 33.8
$\nu = 1.5$	14.5 ± 18.1	22.4 ± 7.82	57.5 ± 43.1	19.1 ± 29.5	34.4 ± 47.4	31.8 ± 35.2
$\nu = 2.0$	16.4 ± 18.5	21.6 ± 6.73	59.6 ± 46.6	35.1 ± 73.7	25.2 ± 24.5	33.4 ± 34.5
Tsafrir	14.6 ± 17.8	20.8 ± 6.83	59.0 ± 45.3	35.8 ± 75.6	27.1 ± 29.5	33.4 ± 33.6

Evaluation with workload traces

Given the performance of the models when trained with uncertain estimates, we now evaluate what happens when we evaluate our models under real production workloads. For doing so, we selected a set of different job traces of clusters with varied numbers of processors. All datasets we used were downloaded from the Parallel Workloads Archive, and are summarized in Table 5.8. In the table we also show the number of processors we used to train our models when mapping to each workload trace. From the table, for example, we see that the same model was used in the CTC-SP2 workload and in the HPC2N workload, since we used the 256-processor model with both workload traces.

We computed both the average slowdown obtained by each model, shown in Table 5.10, and the cluster utilization, shown in Table 5.11. In the tables, we included not only the metrics for our models, but we also computed the average bounded slowdown and utilization achieved by the original scheduler of the logs, and also the metrics of a scheduler that uses the SJF heuristic, and the packing heuristic proposed by Grandl et al. [2014]. Including the real scheduler performance serves as a qualitative comparison, since, differently from our evaluation scenario, the real clusters were not empty when the evaluation jobs arrived in the system. This is especially salient in the CTC-SP2 cluster, which differ by one order of magnitude be-

tween the simulation and actual settings. Due to that, we also included the performance of SJF (which minimizes slowdown with accurate estimates), and the packing heuristic, which tends to increase resource utilization. We see that for most simulation scenarios, the variance of the average bounded slowdown metric is on the same order of magnitude of the metric itself, making it hard to differentiate between models in a statistically-significant way.

In Table 5.10, we underlined models trained with noise that outperformed the corresponding model trained with accurate estimates. In general, although with not too large a difference, at least one model trained with inaccurate estimates outperformed the same model when trained with accurate estimates. What is interesting in the metrics shown is that the learned model is more stable than the heuristics compared. For example, in the SDSC-BLUE case, the average bounded slowdown is $\approx 3\times$ larger than that of the learned models and even when the learning model trained with accurate estimates is outperfomed by a heuristic, the maximum difference (SDSC-SP2, SJF) is on the order of $\approx 3\%$. It is also somewhat surprising that the packing heuristic has worse performance than the actual schedule in the SDSC-SP2 trace.

Observing the results from Table 5.11, we see that utilization with the learning models is, for most workload traces, larger than those of the heuristics (although not statistically-significant), giving us confidence that even when optimizing for one metric (average bounded slowdown), our model is still competitive when we consider other metrics (utilization). In the table, we see that, for some systems, actual utilization is much lower than the computed utilization using the algorithms implemented in this work. We believe this happens due to differences in number of queues (in our simulations, we used a single incoming queue, whereas the actual systems may have multiple queues with different restrictions), and jobs submitted to different partitions of the clusters.

Table 5.10: Average bounded slowdown for the trace files used in this section. Underlined entries represent models that outperform the base model trained on accurate estimates.

Processors	128	256		2048	16384
	SDSC-SP2	CTC-SP2	HPC2N	SDSC-BLUE	ANL-Intrepid
Accurate estimates	247.81 ± 165.76	7.22 ± 4.15	43.53 ± 65.02	22.35 ± 16.90	1.26 ± 0.31
$\nu = 0.5$	236.77 ± 168.06	6.49 ± 3.85	46.58 ± 68.42	20.60 ± 16.92	1.30 ± 0.38
$\nu = 1.0$	230.26 ± 158.46	6.89 ± 4.80	43.41 ± 60.45	21.81 ± 17.79	1.27 ± 0.27
$\nu = 1.5$	244.10 ± 169.84	6.67 ± 3.84	44.74 ± 67.47	21.40 ± 17.39	1.23 ± 0.26
$\nu = 2.0$	225.03 ± 164.49	7.34 ± 4.81	46.32 ± 59.16	21.42 ± 16.68	1.31 ± 0.34
Tsafrir	243.30 ± 143.33	6.76 ± 4.15	39.30 ± 58.26	21.11 ± 15.75	1.34 ± 0.45
Shortest job first	240.10 ± 107.08	60.27 ± 103.52	55.63 ± 62.61	67.99 ± 99.30	1.78 ± 0.77
Packing heuristic	305.04 ± 221.95	6.98 ± 2.93	42.58 ± 59.07	23.12 ± 18.96	3.84 ± 6.17
Actual schedule	271.53 ± 201.28	86.05 ± 104.36	161.90 ± 184.35	242.618 ± 171.508	16.67 ± 16.07

Table 5.11: Cluster utilization for the trace files used in this section.

Processors	128	256		2048	16384
	SDSC-SP2	CTC-SP2	HPC2N	SDSC-BLUE	ANL-Intrepid
Accurate Estimates	0.79 ± 0.036	0.75 ± 0.081	0.61 ± 0.140	0.69 ± 0.076	0.71 ± 0.15
$\nu = 0.5$	0.79 ± 0.034	0.75 ± 0.080	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
$\nu = 1.0$	0.79 ± 0.038	0.75 ± 0.080	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
$\nu = 1.5$	0.78 ± 0.038	0.75 ± 0.080	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
$\nu = 2.0$	0.79 ± 0.039	0.75 ± 0.082	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
Tsafrir	0.79 ± 0.040	0.75 ± 0.080	0.61 ± 0.140	0.69 ± 0.072	0.71 ± 0.16
Shortest Job First	0.74 ± 0.055	0.75 ± 0.031	0.54 ± 0.099	0.64 ± 0.092	0.73 ± 0.14
Packing heuristic	0.77 ± 0.054	0.77 ± 0.033	0.55 ± 0.100	0.66 ± 0.091	0.69 ± 0.13
Actual schedule	0.76 ± 0.082	0.58 ± 0.093	0.41 ± 0.147	0.41 ± 0.107	0.38 ± 0.07

5.3.3 Section Summary

In this section, we proposed a way to model job scheduling as an SMDP with a recently-introduced RL algorithm (Maskable PPO). We showed, through our experiments, that our proposed model works with off-the-shelf implementations, by way of being compatible with the OpenAI Gym environment, and that it is competitive with other learning algorithms and heuristics, while training in a short time. More importantly, we were able to train good models with a third of training iterations of our previous work 5.2 while observing the effects of uncertainty in user run time estimates, a factor that is often overlooked with reinforcement learning agents for resource management, and we have shown these models are competitive when evaluated with real workload traces.

We showed that while training with noisy run time estimates improved the model in simpler settings, doing so failed to provide statistically-significant improvements both with synthetic, but realistic, workload models and with workload traces from real systems. Still, we showed that models trained with accurate run time estimates perform well even with noisy estimates from real workload traces, which we attribute to the model being able to "reason" over possible future cluster states at each decision point. Additionally, we showed our approach of training with realistic workload models outperforms models from the literature even on unseen data. Moreover, our models are competitive when transferred from one workload trace to another, even outperforming models from the literature that were trained in the target environment. Overall, our experiments suggest that training with synthetic workload models yields models that can generalize well to unseen situations.

One limitation of this study is that, for the synthetic workloads we generate, we don't model whether users, or the applications they execute, have consistent run time estimate biases. We don't think identifying the characteristics of user applications, or learning a user model belong, necessarily, in the RL agent, and we think the scheduling system as a whole could implement these models, and provide this information to the RL agent, perhaps as an embedding layer to the agent's policy and value networks. Due to that, we don't model specific user behavior. We don't think that is a problem though, as run time estimates in HPC clusters tend to gravitate around a set of twenty most popular estimates [Tsafrir et al., 2005].

Given the above, we did not model user types in this study as we were focused on the RL model, nor did we think it was appropriate to introduce such a feature, although other models in the literature use it. Moreover, without an appropriate user model, simply adding a user identifier to the job state representation would probably work if jobs were submitted by a fixed set of users. Still, such an agent would fail to generalize once a new user type was introduced, reducing its usefulness in practice.

As presented, the way we model observations make the problem we try to solve a Partially-Observable Markov Decision Process (POMDP), not an MDP. We believe that, sim-

ilarly to the Atari case, in which the approach of stacking frames prior to providing them to the agent effectively turns the problem from a POMDP to an MDP [Hausknecht and Stone, 2015], adding the event horizon to our agent, along with a window of attention of 128 jobs, also brings the problem closer to an MDP. To overcome the limitation of using the backlog in the state representation, it might be possible to integrate an attention layer [Vaswani et al., 2017] to act upon the complete queue state of the system, allowing the agent to select any job for running, and not only jobs in the window of attention. Such a modification is left for future work.

Chapter 6

Conclusion

In this chapter, we revisit all aspects of this dissertation, discussing its contributions, limitations, and proposing directions for future research.

6.1 Overview

In this dissertation we've dealt with how to model High Performance Computing (HPC) job scheduling with Reinforcement Learning (RL). We have presented an approach to build a simulator and corresponding RL environment in which agents can learn scheduling policies directly, by choosing indices in the wait queue as actions. We also analyzed empirically the impacts of design decisions in parts of MDPs, observing the impact on learning. Finally, we proposed an option-based SMDP formulation with relatively low computational resource requirements, removing the need of preprocessing data, while being comparable to other performant approaches in the literature and analyzing the impact of uncertainty in job run time estimates in learning performance. All these contributions were proposed and evaluated in light of the guidelines we proposed: (G1) having agents that did not rely on *specific* environments, but were able to learn general scheduling functions; (G2) being able to learn in modest hardware, such as a desktop computer; and (G3) support algorithms from the literature, without the need for special-case code.

6.2 Contributions

6.2.1 A study of the impacts of MDP design decisions on learning performance

We went through most components of the MDP tuple and proposed new formulations to them. The new formulations are: image-like *versus* compact representations, time-based versus event-based transitions, and adjustments to reward function computation. For each of these proposed changes, we performed a set of experiments to determine the impact of such changes on learning performance.

After all our proposed changes had been implemented, we arrived at a model that supported transferring between cluster configuration and which had good performance across a wide range of configurations. Contrary to one might have imagined, this set of changes *reduced* computational requirements for learning while also *improving* agent performance.

6.2.2 A treatment of HPC job scheduling as an option-based SMDP

Our analysis of the changes to the base MDP had shown the MDPs we were dealing with were powerful enough to support a wide range of scenarios, but some of them were not strict MDPs, as time would pass in increments different from one in some cases. To solve that, we proposed another environment, based on the Semi-MDP formalism that implemented the same objective function of the base MDP, and that had a stronger theoretical grounding than the first-proposed event-based MDP.

With our SMDP formalism, we were able to show our learned models are competitive with other models in the literature, while removing the need for preprocessing steps for successfully processing real cluster workload traces, enabling our model to be trained in approximately one hour in a desktop computer.

6.2.3 An investigation of the impact of uncertainty in run time estimates

We started an investigation of the impacts of uncertainty in job run time estimates, and performed a comprehensive set of experiments that show how an agent trained with the SMDP formulation performed under the effects of varying degrees of Gaussian noise, and also when using the Tsafrir model for run time estimates. We also contrasted the performance of the learned agents with that of two classical heuristics, opening up an avenue for developing agents that perform better under uncertainty.

6.2.4 A design approach that decouples MDP and agent development for HPC job scheduling

We have shown how following a standard API for defining RL environments enabled us to solve the HPC job scheduling problem with different algorithms with ease, relying on open implementations of algorithms that, otherwise, would have taken much more time to implement and execute.

We have assessed whether the approach was sound by being able to reproduce results from the literature, and then we have demonstrated faster convergence with stable baseline algorithm implementations [Raffin et al., 2021].

6.2.5 Software contributions

Our main software contribution is an RL environment[1] that supports time-based and event-based simulations, with support for different workload generators (such as the ones proposed by Mao et al. [2016] and Lublin and Feitelson [2003]), with different job run time uncertainty models (such as the Gaussian model and the model proposed by Tsafrir et al. [2005]), and support for replaying traces in the Standard Workload Format (SWF), including the ones from the Parallel Workloads Archive.

Another software contribution is the library that powers `sched-rl-gym`'s workload models[2], which wraps and exposes the Tsafrir and Lublin models to the Python language.

[1] `https://github.com/renatolfc/sched-rl-gym`, DOI: 10.5281/zenodo.7068617.

[2] `https://github.com/renatolfc/parallelworkloads`, DOI: 10.5281/zenodo.7068617.

6.3 Limitations and directions for future research

6.3.1 Fixed size of the window of attention

A limitation of the current representation is that it assumes the number of job slots in the state representation contains all or most jobs that will ever be waiting in the queue. Although we were able to evaluate our models with success using windows of attention with sizes up to 128 entries, this limits agents into looking into a fixed set size of possible actions.

More importantly, this fixed size limitation made us use an extension to the PPO algorithm that supported the masking of actions, as we have to present a set of jobs of a fixed size to the agent, even if there are fewer jobs in the system than positions in the window of attention.

This is a somewhat artificial restriction, and a true general model should not rely on a representation of a fixed size. Approaches such as a kernel-based neural network combined with more global information, or the usage of an attention mechanism [Vaswani et al., 2017] might help lift this restriction of a fixed-size set of actions. These approaches might also remove the need for using a masked version of PPO, making the learning algorithm simpler and more comprehensible.

6.3.2 Incomplete information state representation

Another limitation of the state representation is the presence of the backlog. Especially in MDPs with image-like representations, the limited window of attention combined with the backlog make the environment violate the Markov property: in heavy-loaded systems, if the number of waiting jobs is greater than the space used to represent the backlog, the agent loses information about the queue, making it possible to have identical state representations even in different states, making learning harder.

Ideally, state representations should maintain the Markov property at all times, enabling queues of any size to be represented and understood by agents. The solution to this is to define a representation that always passes all information to the agent, but once this is done, a fully-connected neural network might not be the best type of neural network to use, as these architectures tend to require fixed-size representations.

6.3.3 Optimization of a single metric

Although we have shown that agent performance is satisfactory in metrics other than slowdown (such as utilization), we still have the limitation that our MDPs only support the optimization of a *single* metric. In more complex environments, the reward function should support join optimization of more than one metric.

6.3.4 Limited use of job information

Even in the SMDP model with cluster traces, we don't make use of the full set of information from jobs. For example, we don't have a user model, nor we use the user identifier information, nor information about the task being performed (SWF has a field for the executable, or the application, number). This makes it hard to make fine-grained decisions, and for determining and using the resource usage profile of an application to instruct scheduling decisions.

It would be interesting to be able to build a user model and application model that could help RL agents make better decisions. The difficulty with such a model is that it would probably make transfer harder. To mitigate this, we could design a hierarchical model that made different decisions based on the availability of a model for users or applications. The moment we do that, though, we will have to keep learning the model continuously with the RL system, lest it becomes outdated and unusable.

6.3.5 A single queue

Throughout this work we made the assumption there is only a single admission queue in the whole cluster, and while this is true for all learning agents we found in the literature, this is not true of real systems, which have multiple queues with different priorities. Additionally, it is frequent for priorities of jobs being affected by the date and time of the day the scheduling decision is being made.

We made no attempt to model such situations in this work, but these issues need to be addressed if we are to adopt scheduling algorithms learned through RL.

Bibliography

Ana P. Appel, Renato L. de F. Cunha, Charu Aggarwal, and Marcela Megumi Terakado. Temporally evolving community detection and prediction in content-centric networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Dublin, Ireland, 2018. Springer.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

M. Emin Aydin and Ercan Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2):169–178, 2000. ISSN 09218890. doi: 10.1016/S0921-8890(00)00087-7.

Betis Baheri and Qiang Guan. MARS: Multi-Scalable Actor-Critic Reinforcement Learning Scheduler. *arXiv e-prints*, art. arXiv:2005.01584, May 2020.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1–43, 2018.

K A Beaty, J M Chow, R L F Cunha, K K Das, M F Hulber, A Kundu, V Michelini, and E R Palmer. Managing sensitive applications in the public cloud. *IBM Journal of Research and Development*, 60(2-3):4:1–4:13, mar 2016. ISSN 0018-8646. doi: 10.1147/JRD.2015.2513720.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, 2(3):207–227, 2011.

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

Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco AS

Netto, et al. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)*, 51(5):1–38, 2018.

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

Xiaomeng Chen, Hui Zhang, Hanli Bai, Chunming Yang, Xujian Zhao, and Bo Li. Runtime prediction of high-performance computing jobs based on ensemble learning. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, HP3C 2020, page 56–62, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376914. doi: 10.1145/3407947.3407968. URL https://doi.org/10.1145/3407947.3407968.

Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 103–127. Springer, 2002.

Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar. PowerAI DDL. *arXiv preprint arXiv:1708.02188*, 2017.

Renato L F Cunha, Marcos D. Assunção, Carlos Cardonha, and Marco A S Netto. Exploiting user patience for scaling resource capacity in cloud services. In *IEEE International Conference on Cloud Computing, CLOUD*, pages 448–455. IEEE Computer Society, 2014.

R.L.F. Cunha, E.R. Rodrigues, L.P. Tizzei, and M.A.S. Netto. Job placement advisor based on turnaround predictions for HPC hybrid clouds. *Future Generation Computer Systems*, 67, 2017. ISSN 0167739X. doi: 10.1016/j.future.2016.08.010.

Alexandre da Silva Veith, Felipe Rodrigo de Souza, Marcos Dias de Assunção, Laurent Lefèvre, and Julio Cesar Santos dos Anjos. Multi-objective reinforcement learning for reconfiguring data stream analytics on edge computing. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

Marcos Dias De Assunção, Carlos H. Cardonha, Marco A S Netto, and Renato L F Cunha. Impact of user patience on auto-scaling resource capacity for cloud services. *Future Generation Computer Systems*, 55:41–50, 2016.

Renato Luiz de Freitas Cunha and Luiz Chaimowicz. Towards a Common Environment for Learning Scheduling Algorithms. In *Proceedings of the 2020 IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*. IEEE Computer Society, November 2020. in press.

Renato Luiz de Freitas Cunha and Luiz Chaimowicz. On the impact of mdp design for reinforcement learning agents in resource management. In *Brazilian Conference on Intelligent Systems*, pages 79–93. Springer, 2021.

Renato Luiz de Freitas Cunha and Luiz Chaimowicz. An SMDP approach for Reinforcement Learning in HPC cluster schedulers. Accepted for publication at Future Generation Computer Systems on September 26th, 2022, 2022.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

Giacomo Domeniconi, Eun Kyung Lee, and Alessandro Morari. Cush: Cognitive scheduler for heterogeneous high performance computing system. In *Proceedings of DRL4KDD 19: Workshop on Deep Reinforcement Learning for Knowledge Discovery (DRL4KDD). Reproduced by: Vanamala Venkataswamy, Swaroopa Dola*, volume 12, 2019.

Yuping Fan and Zhiling Lan. Exploiting multi-resource scheduling for hpc. *SC Poster*, 2019.

Yuping Fan, Paul Rich, William E. Allcock, Michael E. Papka, and Zhiling Lan. Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 530–540, 2017. doi: 10.1109/CLUSTER.2017.11.

Yuping Fan, Zhiling Lan, Taylor Childers, Paul Rich, William Allcock, and Michael E Papka. Deep reinforcement agent for scheduling in hpc. *arXiv preprint arXiv:2102.06243*, 2021.

Dror G Feitelson. Metrics for parallel job scheduling and their convergence. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 188–205. Springer, 2001.

Dror G Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with the parallel workloads archive. *The Hebrew University and the Israel Institute of Technology*, 2012.

Keke Gai and Meikang Qiu. Optimal resource allocation using reinforcement learning for IoT content-centric services. *Applied Soft Computing Journal*, 70:12–21, 2018. ISSN 15684946. doi: 10.1016/j.asoc.2018.03.056. URL `https://doi.org/10.1016/j.asoc.2018.03.056`.

Piotr Gawłowicz and Anatolij Zubow. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, November 2019.

Paul Glasserman and David D Yao. Some guidelines and guarantees for common random numbers. *Management Science*, 38(6):884–908, 1992.

Robert Glaubius, Terry Tidwell, Christopher Gill, and William D. Smart. Real-time scheduling via reinforcement learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, page 201–209, Arlington, Virginia, USA, 2010. AUAI Press. ISBN 9780974903965.

Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

Swetha Hariharan, Prakash Murali, Abhishek Pasari, and Sathish Vadhiyar. End-to-end predictions-based resource management framework for supercomputer jobs. *arXiv preprint arXiv:2008.08292*, 2020.

Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Steven Hotovy. Workload evolution on the cornell theory center ibm sp2. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 27–40. Springer, 1996.

Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022. doi: 10.32473/flairs.v35i.130584. URL https://doi.org/10.32473/flairs.v35i.130584.

Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010.

Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059, 2019.

Athanassios M Kintsakis, Fotis E Psomopoulos, and Pericles A Mitkas. Reinforcement learning based scheduling in a workflow management system. *Engineering Applications of Artificial Intelligence*, 81:94–106, 2019.

B. A. Kumar and T. Ravichandran. Instance and value (ivh) algorithm and dodging dependency for scheduling multiple instances in hybrid cloud computing. In *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pages 408–411, 2013.

Mohit Kumar, S.C. Sharma, Anubhav Goel, and S.P. Singh. A comprehensive survey for scheduling techniques in cloud computing. *Journal of Network and Computer Applications*, 143(April):1–33, 2019. ISSN 10848045. doi: 10.1016/j.jnca.2019.06.006. URL https://linkinghub.elsevier.com/retrieve/pii/S1084804519302036.

Rajath Kumar and Sathish Vadhiyar. Identifying quick starters: towards an integrated framework for efficient predictions of queue waiting times of batch parallel jobs. In *Proceedings of the 16th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2013.

Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 253–263. Springer, 2004.

Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proc. VLDB Endow.*, 11(6):705–718, February 2018. ISSN 2150-8097. doi: 10.14778/3199517.3199521. URL https://doi.org/10.14778/3199517.3199521.

Sisheng Liang, Zhou Yang, Fang Jin, and Yong Chen. Data centers job scheduling with deep reinforcement learning. *arXiv preprint arXiv:1909.07820*, 2019.

Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael Bowling. State of the Art Control of Atari Games Using Shallow Reinforcement Learning. In *AAMAS*, 2016.

Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003. ISSN 0743-7315. doi: https://doi.org/10.1016/S0743-7315(03)00108-4. URL https://www.sciencedirect.com/science/article/pii/S0743731503001084.

N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y. Liang, and D. I. Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys Tutorials*, 21(4):3133–3174, 2019.

Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and

open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

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. ISBN 9781450346610. doi: 10.1145/3005745.3005750.

Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Bojja Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. Park: An Open Platform for Learning-Augmented Computer Systems. *Nips*, (NeurIPS):61–63, 2019a. URL `https://github.com/park-project/park`.

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. 2019b.

Aniruddha Marathe, Rachel Harris, David K Lowenthal, Bronis R de Supinski, Barry Rountree, Martin Schulz, and Xin Yuan. A comparative study of high-performance computing on the cloud. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013.

Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 2–13. ACM, 2004.

Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society, 2010.

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, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE transactions on parallel and distributed systems*, 12(6):529–543, 2001.

Graham R Nudd, Darren J Kerbyson, Efstathios Papaefstathiou, Stewart C Perry, John S Harper, and Daniel V Wilcox. PACE-A toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3): 228–251, 2000.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. Dl2: A deep learning-driven scheduler for deep learning clusters. *arXiv preprint arXiv:1909.06040*, 2019.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 2021.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

Gonzalo P Rodrigo, P-O Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards understanding hpc users and systems: a nersc case study. *Journal of Parallel and Distributed Computing*, 111:206–221, 2018.

Eduardo R. Rodrigues, Renato L. de F. Cunha, Marco A. S. Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *Workshop on HPC User Support Tools*, Salt Lake City, Utah, 2016.

Iman Sadooghi, J Hernandez Martin, Tonglin Li, Kevin Brandstatter, Y Zhao, K Maheshwari, T Pais Pitta de Lacerda Ruivo, Steven Timm, G Garzoglio, and Ioan Raicu. Understanding the performance and potential of cloud computing for scientific applications. *IEEE Transaction on Cloud Computing*, PP(99):1–1, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

W. Smith. Prediction services for distributed computing. In *Proceeding of the 21th International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

Warren Smith, Valerie Taylor, and Ian Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Workshop on Job scheduling strategies for Parallel Processing*, pages 202–219. Springer, 1999.

Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 64(9):1007–1016, 2004.

Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 21–21. IEEE, 2002.

Stelios Sotiriadis, Nik Bessis, and Rajkumar Buyya. Self managed virtual machine scheduling in cloud systems. *Information Sciences*, 433-434:381 – 400, 2018. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2017.07.006. URL http://www.sciencedirect.com/science/article/pii/S0020025517308277.

Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. Selective reservation strategies for backfill job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 55–71. Springer, 2002.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018a.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, chapter 17: Frontiers, pages 469–472. MIT press, 2018b.

Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999. URL http://www-all.cs.umass.edu/pubs/1999/sutton{_}ps{_}AI99.pdf.

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.

Wei Tang, Zhiling Lan, Narayan Desai, Daniel Buettner, and Yongen Yu. Reducing fragmentation on torus-connected supercomputers. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 828–839. IEEE, 2011.

Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

Philip S Thomas and Emma Brunskill. Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines. *arXiv preprint arXiv:1706.06643*, 2017.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Zhao Tong, Zheng Xiao, Kenli Li, and Keqin Li. Proactive scheduling in distributed computing - A reinforcement learning approach. *Journal of Parallel and Distributed Computing*, 74(7): 2662–2672, 2014. ISSN 07437315. doi: 10.1016/j.jpdc.2014.03.007. URL http://dx.doi.org/10.1016/j.jpdc.2014.03.007.

Zhao Tong, Hongjian Chen, Xiaomei Deng, Kenli Li, and Keqin Li. A scheduling scheme in the cloud computing environment using deep Q-learning. *Information Sciences*, 512:1170–1191, 2020. ISSN 00200255. doi: 10.1016/j.ins.2019.10.035.

Dan Tsafrir and Dror G Feitelson. Instability in parallel job scheduling simulation: the role of workload flurries. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. Modeling user runtime estimates. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–35. Springer, 2005.

Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

José R Vázquez-Canteli, Jérôme Kämpf, Gregor Henze, and Zoltan Nagy. Citylearn v1.0: An openai gym environment for demand response with deep reinforcement learning. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pages 356–357, 2019.

Lan Wang and Erol Gelenbe. Adaptive dispatching of tasks in the cloud. *IEEE Transactions on Cloud Computing*, 6, 01 2015. doi: 10.1109/TCC.2015.2474406.

Yuandou Wang, Hang Liu, Wanbo Zheng, Yunni Xia, Yawen Li, Peng Chen, Kunyin Guo, and Hong Xie. Multi-objective workflow scheduling with deep-q-network-based multi-agent reinforcement learning. *IEEE access*, 7:39974–39982, 2019.

Yi Wei, L. Pan, Shijun Liu, L. Wu, and Xiangxu Meng. Drl-scheduling: An intelligent qos-aware job scheduling framework for applications in clouds. *IEEE Access*, 6:55112–55125, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 366–387. Springer-Verlag New York, Inc., 2008.

Miguel G Xavier, Marcelo Veiga Neves, Fabio D Rossi, Tiago C Ferreto, Tobias Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2013.

Minxian Xu, Chenghao Song, Huaming Wu, Sukhpal Singh Gill, Kejiang Ye, and Chengzhong Xu. Esdnn: Deep neural network based multivariate workload prediction in cloud computing environments. *ACM Trans. Internet Technol.*, mar 2022. ISSN 1533-5399. doi: 10.1145/3524114. URL https://doi.org/10.1145/3524114. Just Accepted.

Leo T. Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC)*, 2005.

Di Zhang, Dong Dai, Youbiao He, and Forrest Sheng Bao. Rlscheduler: Learn to schedule hpc batch jobs using deep reinforcement learning. *arXiv preprint arXiv:1910.08925*, 2019.

Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: an automated hpc batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer, 1995.

Yu Zhang, Jianguo Yao, and Haibing Guan. Intelligent Cloud Resource Management with Deep Reinforcement Learning. *IEEE Cloud Computing*, 4(6), 2017. ISSN 23256095. doi: 10.1109/MCC.2018.1081063.

Dinitry Zotkin and Peter J Keleher. Job-length estimation and performance in backfilling schedulers. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 236–243. IEEE, 1999.