UCL

Doctoral Thesis

Computer Network Optimisation with Artificial Intelligence and Optics

Author:

Christopher W. F. Parsonson

Supervisor:

Georgios Zervas

A thesis submitted for the degree of

Doctor of Philosophy

Optical Networks Group

Electronic and Electrical Engineering Department

2023

"*Science is the belief in the ignorance of experts.*"

Richard Feynman

Abstract

Christopher W. F. Parsonson

Computer Network Optimisation with Artificial Intelligence and Optics

The last decade has seen a proliferation in data-intensive compute applications such as artificial intelligence (AI), genome sequencing, and the internet-of-things. The ever-growing throughput demand of these big-data jobs has coincided with a slow down in the development of powerful computer chips. Consequently, there has been a shift away from local computation with general-purpose CPUs towards remote pooling of specialised high-bandwidth processors in cloud data centres (DCs) and high-performance compute (HPC) clusters. Such computation relies on a *computer network* to facilitate data querying and parallel processing. The traditional Moore's Law approach of evaluating compute power and cost purely in terms of individual end points is therefore no longer appropriate. Instead, compute must now be thought of as a *system* of interconnected resources which can be orchestrated to perform a task.

However, there has been a lack of development in next-generation computer networks, leading to the performance bottleneck of these systems moving away from the end point processors themselves and into the network connecting them. Optical networking is a technology which can offer orders-of-magnitude improvement in computer network performance. For optical networks to be widely used in DCs and HPCs, several obstacles related to physical optical device characteristics and resource management must be overcome. In this thesis, we develop and evaluate novel AI approaches for addressing these challenges.

The first part of the thesis looks at *optimising the physical plane's* devices in an optical computer network. Concretely, three gradient-free AI signal control approaches (ant colony optimisation, a genetic algorithm, and particle swarm optimisation) are proposed to enable high-bandwidth, low-power optical switching technologies to operate on the sub-nanosecond timescales required to realise an optical circuit switched data centre network.

The second part of the thesis considers the problem of *optimising the orchestration plane's* resource management methods used to control optical computer networks. A novel algorithm, *retro branching*, is proposed to improve the solve time performance of the canonical branch-and-bound exact solver using a graph neural network (GNN) trained with reinforcement learning (RL). State-of-the-art RL-for-branching results are achieved, opening the possibility for branch-and-bound to be applied to large NP-hard discrete optimisation problems such as those found in computer network resource management. We also propose another algorithm, *PAC-ML* (**p**artitioning for **a**synchronous **c**omputing with **m**achine **l**earning), which trains a GNN with RL to automatically decide *how much* to distribute deep learning jobs in an optical HPC architecture in order to meet user-defined run time requirements, minimise the blocking rate, and maximise system throughput under dynamic scenarios; the first of its kind to consider such a problem setting.

So far we have have considered optimising the devices in the physical plane and the resource managers in the orchestration plane of the computer network. These areas have both received research attention in prior works. However, what has not received much consideration is the underlying test bed in which physical and orchestration plane research and optimisation is typically conducted. Real DC and HPC environments are generally not available for research due to their proprietary nature and expensive cost of deployment. Consequently, researchers rely on simulated computer networks during novel system development. The fidelity, reproducibility, and flexibility of these simulations is therefore at least as

important as the development and optimisation of the physical and orchestration systems for which they are used. Poor simulations will lead to the misguided development of network systems which do not perform as expected when deployed in real production environments. With this motivation, the third part of this thesis considers how to design and *optimise the simulator* used for computer network system research and development. A novel open source traffic generation framework and library, *TrafPy*, is presented, as well as a subsequent update to the generation algorithm to make it scalable to computer networks with thousands of nodes.

Acknowledgements

I would like to thank my supervisor, Georgios Zervas, for taking me on as as a Ph.D. student in his group. Georgios gave me the space to explore a variety of topics across a broad range of research areas and provided invaluable guidance both in relation to the Ph.D. and the additional extra-curricula experiences I put myself forward for. I am also thankful to the wider Optical Networks Group, headed by Polina Bayvel, for providing a light-hearted but focused and academically rigorous environment in which I could develop as a researcher. In addition, I am grateful to Lee Heagney, the Department's IT & Systems Manager, for providing critical technical help throughout my Ph.D., even outside of office hours. I also thank EPSRC and the Cambridge-UCL Integrated Photonic and Electronic Systems centre for doctoral training for funding both my M.Res. and Ph.D. studies. Moreover, I thank the ConceptionX team for giving me the chance to learn more about how to convert blue sky Ph.D. research into practical and useful solutions, as well as for all of the amazing people and companies which the programme connected me with. Furthermore, I'd like to thank Thomas Barrett and Alexandre Laterre for giving me the opportunity to undertake an internship at InstaDeep, for supervising my project, and for patiently teaching me an enormous amount about how to methodically and persistently conduct novel machine learning research. On that note, I also thank everyone at InstaDeep for being so welcoming and for enabling me to experience how an exceptional research and engineering team operates in industry. I am also grateful to The Alan Turing Institute for admitting me as an Enrichment Student for a six-month placement, during which I met many fantastic people from a variety of

backgrounds and research topics and with whom I had numerous interesting conversations and fun experiences.

I would also like to thank all of the researchers, commentators, and authors whose work I cite throughout this thesis for being the source of many of my ideas and projects. I am especially grateful to Hongzi Mao, David Silver, Jure Leskovec, and Jakob Foerster, whose research I found particularly inspiring and who had a significant influence on my work. I also thank Mark Oxborrow and Julian Jones whose lectures sparked my interest in academic research, as well as my undergraduate tutor Fionn Dunne who engaged with this interest and guided me through my studies. I am also grateful to my early teachers at Barnardiston, particularly Paul Whittles, Caroline Blake, and Keith Boulter, whose never-ending patience, encouragement, and generosity was most helpful.

Finally, I would like to thank my friends and family for supporting me during my academic studies, especially Yasmin who chose to put up with me all these years. I am particularly thankful to my parents, Richard and Sharon Parsonson, for having provided me with all of the opportunities I could hope for. I am also enormously grateful to my grandparents, Stuart and Pamela Parsonson, whose infectious passion for knowledge and maths in part motivated my decision to pursue academic research, and whose many hours of teaching in the preceding years enabled me to do so.

Contents

List of Figures

List of Tables

List of Abbreviations

ACO ant colony optimisation

AI artificial intelligence

BFS breadth-first search

B&B branch-and-bound

CO combinatorial optimisation

CPU central processing unit

DAG directed acyclic graph

DCN data centre network

DFS depth-first search

DNN deep neural network

DQN deep Q-network

FLOP floating point operation

FMSTS fitting for minimising the sub-tree size

FPGA field-programmable gate array

GA genetic algorithm

GCN graph convolutional network

GNN graph neural network

GPU graphics processing unit

HPC high-performance computing

IL imitation learning

ILP integer linear programming

JCT job completion time

JSD Jensen-Shannon distance

LP linear programme

MDP Markov decision process

MILP mixed integer linear programme

MISIC multi-impulse step injection current

ML machine learning

MPI message passing interface

MSE mean squared error

MT-NLG Megatron-Turing natural language generation

NN neural network

OBS optical burst switching

OCS optical circuit switching

OEO optical-electrical-optical

OPS optical packet switching

PB pseudocost branching

PID proportional-integral-derivative

PISIC pre-impulse step injection current

POMDP partially observable Markov decision process

PPO proximal policy optimisation

PSO particle swarm optimisation

RL reinforcement learning

SB strong branching

SOA semiconductor optical amplifier

SOTA state-of-the-art

SVM support vector machine

TPU tensor processing unit

List of Units

B bytes

bn billion

Gbps gigabits per second

m metre

ms millisecond

nm nanometre

ns nanosecond

ps picosecond

Tbps terabits per second

μ**s** microsecond

To my family

Chapter 1

Introduction

1.1 The Information Revolution & Computer Networks

The information revolution began with the invention of the transistor in the mid-20^{th} century [Riordan, 2004]. Unlike the agricultural and industrial revolutions whose effects took millennia and centuries to be felt across the globe, the information revolution has transformed all facets of society within a single generation [Davidson and Rees-Mogg, 1999]. The speed of its proliferation is testament to its importance to the human condition; information technology is now used everywhere, from the economy and politics to healthcare and education.

Due to this new-found dependence on information technology, society now allocates a significant amount of capital and resources towards its advancement. Consequently, as reflected by Moore's Law, the 1965 observation that the cost-per-FLOP of a central processing unit (CPU) halves every 18 months [Thompson and Spanuth, 2018], we have enjoyed decades of exponentially more powerful compute at ever-lower costs. This has facilitated a range of ubiquitous technologies, from the internet and video streaming to personal computers and smart phones. Twenty years ago when computers were 1000× less powerful, none of these technologies in their present form would have been possible. It is difficult to imagine what future innovations humanity might miss out on over

the next twenty years were advances in information technology to slow.

And yet, current trends do indeed suggest that things are slowing. From 1985 to 2005, compute performance increased by 52% per annum. Since 2005, this rate has fallen to 22% [Hennessy and Patterson, 2017]. The fundamental reason behind this slow down has been the difficulty of manufacturing transistors on the nanometre (nm) scale, with the cost of building a chip fabrication plant having increased 13% annually to reach $16 billion (bn) in 2022 [Rotman, 2020].

In order to circumvent this slow down in the cost-per-FLOP reduction of CPUs, the last decade has seen three trends emerging (visualised in Fig. 1.1[1]).

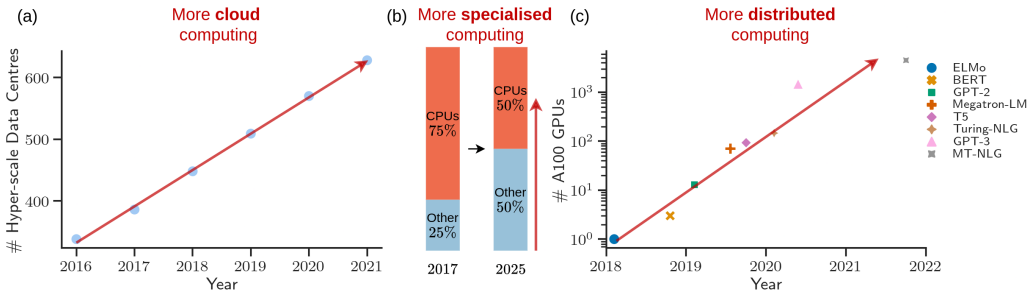


FIGURE 1.1: Visualisation of the three current trends in computing. (a) The number of hyper-scale cloud data centres world-wide almost doubled in a five year period [Cisco, 2016]. (b) By 2025, the majority of processors in data centres will no longer be general-purpose CPUs, but rather specialised high-bandwidth processors such as GPUs and FPGAs [McKinsey, 2019]. (c) The number of distributed A100 GPUs needed to train the state-of-the-art natural language processing models released between 2018 – 2022 has grown by over 1000× [Kharya and Alvi, 2021].

The first is the proliferation of cloud data centres. Rather than everyone needing to possess their own compute resources, a host provides compute-as-a-service to multiple users. In doing so, users avoid the need to directly pay for, implement, and manage the latest hardware themselves in order to use state-of-the-art compute. Pooling resources in this way can offset the increasing relative cost of improving compute performance. Consequently, cloud computing has become abundantly popular. Today, almost everyone, everywhere, everyday uses the cloud, be it through video streaming and AI assistants or instant messaging

[1]Fig. 1.1c assumes it takes 4480 A100 GPUs to train the 530 bn parameter Megatron-Turing natural language generation model [Wiggers, 2021], and that there is a linear relationship between the number of model parameters and the number of A100 GPUs needed to train it.

and data backups. Over 95% of global information traffic now exists in the cloud [Cisco, 2016], and with the appeal of pooling resources to increase the accessibility of powerful compute unlikely to wane, our dependence on data centres is likely to persist.

The second trend is the shift away from general purpose CPUs towards specialised processors such as graphics processing units (GPUs), tensor processing units (TPUs), and field-programmable gate arrays (FPGAs) in order to facilitate new big data computational jobs such as AI and genome processing [McKinsey, 2019]. These processors do fewer things than CPUs but can perform significantly better at their designated task. While Moore's Law appears to have ended for CPUs, the processing power of GPUs increased by over 25× from 2012 to 2018 [Perry, 2018]. However, the rate of this performance improvement was largely due to the low hanging fruit of tackling the parts of the compute ecosystem not well served by CPUs (primarily data-parallel computation, which has become critical to the neural network (NN) architectures widely used today). In the five years since 2018, the cost-per-FLOP of state-of-the-art GPUs has only halved every 3 years [Tamay, 2022]; 2× slower than Moore's Law. As articulated by Stoica [2020], this slow down is coinciding with a huge increase in computational demand by applications such as AI whose resource requirements have been doubling every 3.4 months since 2012; 50× faster than Moore's Law [OpenAI, 2018].

Consequently, a third trend has emerged; distributed computing. Rather than trying to fit a large computational job into the memory of a single device and sequentially running it, the job is instead split up and ran in parallel across multiple machines in a HPC system.

Both data centres and HPC systems require computers to communicate with one another, be it to query databases, synchronise the results of a parallel computation, and so on. This communication is done via a *computer network*, which is a system of connected end point processing nodes. The traditional

Moore's Law approach of evaluating compute power and cost purely in terms of individual end points is therefore no longer appropriate. Instead, compute must now be thought of as a *system* of interconnected resources which can be orchestrated to perform a task.

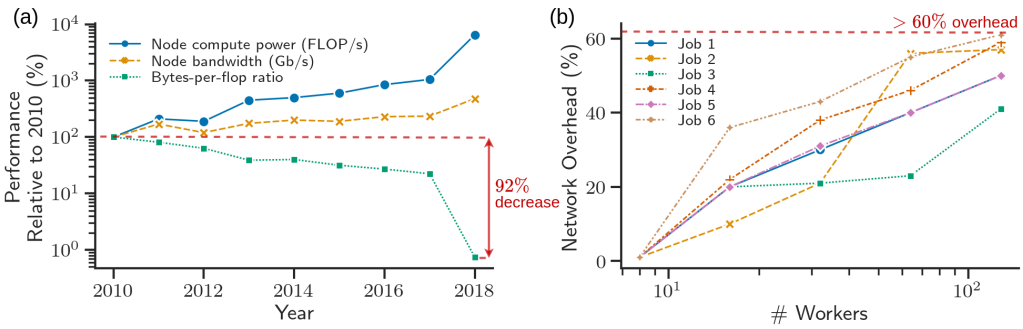


FIGURE 1.2: (a) From 2010 to 2018, the average compute performance improvement of the nodes in the top ten HPC computer network systems far outstripped the improvement in their communication bandwidth, leading to 92% fewer bytes being communicated per FLOP [Bergman, 2018]. (b) How consequently the network overhead - the fraction of the job completion time spent communicating information between workers when no computation is taking place - of distributed deep learning jobs increases with the number of machines used in Meta's GPU cluster, shifting the performance bottleneck into the network [Wang et al., 2022].

However, the ability with which we can increase compute power by increasing the number of machines we distribute across is also slowing. Fig. 1.2a visualises the average end point compute power and communication bandwidth of the top ten HPC systems from 2010 to 2018 [Bergman, 2018]. As shown, although end point compute power increased by a factor of 65, the network bandwidth only increased by 4.8×, leading to a 92% decrease in bytes communicated per FLOP over the eight year period and the maximum performance of these systems being less than 10% of what is theoretically possible. This has shifted the performance bottleneck of computer networks away from the end point nodes and into the network connecting them [Wang et al., 2022] (see Fig. 1.2b). Furthermore, the trend of using specialised parallel processors for executing ever-larger jobs is resulting in significantly more communication between machines being required. Improving the networks of modern HPC and data centre systems is therefore crucial if we are to continue to improve computational performance and cater to

next-generation applications such as AI, data science, and genome processing, and thus forms the focus of this thesis.

1.2 Artificial Intelligence for Optimisation

AI is a broad umbrella term with several definitions. In this thesis, AI is assumed to be the study and design of 'intelligent agents', where an intelligent agent is a system which perceives its environment and takes actions such that its chances of success are maximised [Poole et al., 1998].

The concept of AI has existed in the imaginations of humans for millennia. The oldest records date back to 400 B.C. with the legend of a 30 metre (m) tall brass robot, Talos, protecting Crete from pirates [Sparkes, 2013]. Its establishment as a scientific discipline, however, is relatively recent. The Church-Turing thesis [Turing, 1936, 1950] was the first to spark rigorous academic interest in AI.

The first half-century of AI research focused on its application to games; a domain which strikes a balance between complexity and ease of access. This began with the first analysis of chess playing as a search task [Shannon, 1950] and the study of AI for executing checkers strategies [Samuel, 1959]. Progress stalled in the 'AI winter' of the 1970s when funding for projects was sparse. In the late 1990s, interest in AI began to pick up again by leveraging increased computational power, focusing on specific tasks, establishing scientific standards, and forging links between AI and other fields such as mathematics, statistics, and economics.

The last decade has seen an explosion in the rate of AI progress, the cause of which can largely be attributed to AlexNet [Krizhevsky et al., 2012]; the winner of the 2012 ImageNet competition [Deng et al., 2009] to classify a database of 1.4 million images with 1000 possible classes. AlexNet was the first to achieve a step-change in performance at a widely applicable and important task through

the use of NNs. The realisation that NNs can be efficiently trained with GPUs to solve immensely challenging tasks *automatically* has led to significant progress in machine learning (ML), a particular branch of AI, and a virtuous cycle of both algorithmic and hardware development.

With AI having now achieved super-human performance in complex games such as Go [Silver et al., 2016], Poker [Brown and Sandholm, 2019], and Stratego [Perolat et al., 2022], recent years have seen a shift towards applying AI to real-world optimisation problems in a range of fields, from biology [Jumper et al., 2021] and physics [Wu and Tegmark, 2019] to recommendation algorithms [Afsar et al., 2022] and system management [Degrave et al., 2022].

This thesis considers optimisation problems which arise in computer networks, both at the device level and in terms of overall resource management. Classical approaches to solving these problems typically adopt the following workflow: (1) Construct a simplified model of the computer system; (2) deconstruct high-level design objectives (e.g. 'minimise end-user latency') into low-level tasks (e.g. 'minimise network packet queuing delay'); (3) manually handcraft a heuristic to optimise a problem within the simplified system model; and (4) meticulously test and tune the heuristic until acceptable real-world performance is achieved [Mao et al., 2020].

AI is particularly well suited to replace the above workflow and solve computer network optimisation problems for reasons here summarised into six key factors:

1. **Automatic optimisation**: Rather than requiring expert understanding of a given problem domain and then handcrafting a specific solution for it, AI optimisers can be applied often with little to no tuning and expert knowledge to *automatically* discover novel solutions without the costly overhead of human design.

2. **High-quality optimisation**: As shown by the super-human performance attained in games such as Go [Silver et al., 2016] and in the management

of systems such as nuclear reactors [Degrave et al., 2022], the solutions discovered by AI to complex optimisation problems often outperform heuristics designed by human experts.

3. **High-speed optimisation**: Many state-of-the-art handcrafted algorithms and heuristics (e.g. solvers based on branch-and-bound [Land and Doig, 1960]) require expensive computational steps to be performed, which can be detrimental to applications where fast decision making is critical. By contrast, many AI methods, particularly those which use NNs as function approximators, can make decisions on $O(ms)$ time scales or less [Shabka et al., 2022].

4. **High-fidelity optimisation**: AI methods can continuously adapt to handle real experiences when interacting with an environment, allowing them to directly optimise the actual computer network's workload and operating conditions in dynamic scenarios rather than relying on inaccurate system models.

5. **Handling of large search spaces**: Computers can search for solutions much faster than a human can think of them, often with parallel computation and the use of powerful function approximators such as NNs. This enables AI optimisers to be applied to problem domains with in excess of $O(10^{100})$ possible solutions without the need for exhaustive search or impractical solve times.

6. **Simple objectives**: Classical approaches to system optimisation often require the construction of low-level tasks in order to meet high-level design objectives, which inherently biases the resultant solution towards a potentially sub-optimal prior approach, and requires an expert-level understanding of the entire system stack. By contrast, AI methods can be

given simple high-level objectives, such as 'win a game of chess', and discover low-level policies and value functions which sufficiently optimise this objective. This further reduces the complexity of tackling the problem for practitioners, removes prior biases towards approaches which are assumed to be, but are not necessarily, performant, and allows designers to tackle problems without the need for an expert-level understanding of the whole system.

The above factors form the motivation for this thesis, which seeks to develop AI methods for optimising various components of computer networks at both the device and the resource management level. Furthermore, it is shown how optimisation with AI can enable the transition from electronic to optical networking with superior scalability, bandwidth, latency, and power consumption, and thereby address the shortcomings of modern cloud and HPC systems outlined in Section 1.1. It is therefore hoped that this thesis will aid in facilitating the development of next-generation compute applications, such as large-scale genome processing and ever-more complex AI systems, over the coming decades.

1.3 Structure of & Publications from this Thesis

This thesis is divided into a background section followed by three main parts, each of which addresses a different challenge in developing next-generation computer networks.

1.3.1 Background

Chapter 2 provides an introduction to optical networking and the algorithmic and conceptual tools common to multiple parts of this thesis. To aid the reader in digesting the necessary background information, concepts required only for a specific chapter are introduced within the corresponding chapter.

1.3.2 Part I: Optimising the Physical Plane

Computer communication networks are made up of *physical devices* such as fibre links, network switches, and end point processors. We refer to these devices collectively as the computer network's *physical plane*. The performances of the physical plane's components jointly determine the overall performance of the computer network in terms of key metrics such as throughput, cost, and energy consumption, and are therefore of critical importance.

Part I looks at optimising the physical devices in a computer communication network. Specifically, Chapter 3 proposes three gradient-free AI signal control approaches which enable high-bandwidth, low-power optical switching technologies to operate on the sub-nanosecond (ns) timescales that would be required in an optical data centre network.

The following papers have been published based on the work reported in Part I of this thesis:

- Hadi Alkharsan, **Christopher W. F. Parsonson**, Zacharaya Shabka, Xun Mu, Alessandro Ottino, and Georgios Zervas, 'Optimal and Low Complexity Control of SOA-Based Optical Switching with Particle Swarm Optimisation', *ECOC'22: Proceedings of the Forty-Eighth European Conference on Optical Communication*, 2022

- Thomas Gerard, **Christopher W. F. Parsonson**, Zacharaya Shabka, Benn Thomsen, Polina Bayvel, Domanic Lavery, and Georgios Zervas, 'AI-Optimised Tuneable Sources for Bandwidth-Scalable, Sub-Nanosecond Wavelength Switching', *Optics Express*, 2021

- **Christopher W. F. Parsonson**, Zacharaya Shabka, W. Konrad Chlupka, Bawang Goh, and Georgios Zervas, 'Optimal Control of SOAs with Artificial Intelligence for Sub-Nanosecond Optical Switching', *Journal of Lightwave Technology*, 2020

- Thomas Gerard, **Christopher W. F. Parsonson**, Zacharaya Shabka, Polina Bayvel, Domanic Lavery, and Georgios Zervas 'SWIFT: Scalable Ultra-Wideband Sub-Nanosecond Wavelength Switching for Data Centre Networks', *arXiv*, 2020

1.3.3 Part II: Optimising the Orchestration Plane

The physical devices in a computer network must all be orchestrated in order to perform a computational task. Poor orchestration of the physical devices can lead to under-utilised network resources and excessive operating costs and energy consumption. We refer to the collective *resource management* methods which perform physical device orchestration tasks as the computer network's *orchestration plane*.

Part II considers the resource management methods of the orchestration plane used to control cloud and HPC networks. Chapter 4 proposes a new algorithm which facilitates the integration of a GNN trained with RL to discover novel variable selection policies into a freely-available exact branch-and-bound solver which can be applied to generic NP-hard discrete optimisation problems such as those found in computer network management. Chapter 5 proposes a novel algorithm, also based on an RL-trained GNN, for automatically deciding how much to distribute a deep learning job in an HPC in order to meet user-defined run time requirements, minimise the blocking rate, and maximise system throughput under dynamic scenarios; the first of its kind to consider such a problem setting.

The following publications have come from Part II of this thesis:

- **Christopher W. F. Parsonson**, Zacharaya Shabka, Alessandro Ottino, and Georgios Zervas, 'Partitioning Distributed Compute Jobs with Reinforcement Learning and Graph Neural Networks', *arXiv*, 2023

- **Christopher W. F. Parsonson**, Alexandre Laterre, and Thomas D. Barrett, 'Reinforcement Learning for Branch-and-Bound Optimisation using Retrospective Trajectories', *AAAI'23: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence*, 2023

- Thomas D. Barrett, **Christopher W. F. Parsonson**, and Alexandre Laterre, 'Learning to Solve Combinatorial Graph Partitioning Problems via Efficient Exploration', *arXiv*, 2022

1.3.4 Part III: Optimising the Simulator

So far we have considered optimising the devices in the physical plane and the resource managers in the orchestration plane of computer networks. These are both areas which have received significant attention from the research community. However, what has not had much focus is the underlying test bed in which physical and orchestration plane research and optimisation is typically conducted.

Real production computer network systems such as data centre networks (DCNs) and HPCs are not generally available for researchers to build and test novel system components due to their proprietary nature and expensive cost of deployment. Consequently, many researchers resort to *simulating* computer networks in order to develop novel computer network systems. The fidelity, reproducibility, and flexibility of these simulations is therefore at least as important as the development and optimisation of the physical and orchestration systems for which they are used. Poor simulations will lead to the misguided development of network systems which do not perform as expected when deployed in real production environments.

With this motivation, Part III addresses a key problem faced by many computer network researchers, which is the reliance on low-fidelity, difficult-to-reproduce, and inflexible *computer network simulations* in the absence of access to real production systems. A novel open source traffic generation

framework and library is presented in Chapter 6, and a subsequent update to the generation algorithm to make it scalable to computer networks with $O(10^3)$ nodes is proposed in Chapter 7.

The following are publications which have come from Part III of this thesis:

- Joshua L. Benjamin, **Christopher W. F. Parsonson**, and Georgios Zervas, 'Data Scheduling Unit for Nanosecond Optical Data Center Networks', *arXiv*, 2023
- Yanwu Liu, Joshua L. Benjamin, **Christopher W. F. Parsonson**, and Georgios Zervas, 'A Hybrid Beam Steering Free-Space and Fiber Based Optical Data Center Network', *arXiv*, 2023
- **Christopher W. F. Parsonson**, Joshua L. Benjamin, and Georgios Zervas, 'A Vectorised Packing Algorithm for Efficient Generation of Custom Traffic Matrices', *OFC'23: Optical Fiber Communications Conference and Exhibition*, 2023
- **Christopher W. F. Parsonson**, Joshua L. Benjamin, and Georgios Zervas, 'Traffic generation for benchmarking data centre networks', *Optical Switching and Networking*, 2022
- Joshua L. Benjamin, Alessandro Ottino, **Christopher W. F. Parsonson**, and Georgios Zervas, 'Traffic Tolerance of Nanosecond Scheduling on Optical Circuit Switched Data Center Network', *OFC'22: Optical Fiber Communications Conference and Exhibition*, 2022
- Joshua L. Benjamin, **Christopher W. F. Parsonson**, and Georgios Zervas, 'Benchmarking Packet-Granular OCS Network Scheduling for Data Center Traffic Traces', *Photonic Networks and Devices*, 2021

Chapter 2

Background

2.1 Computer Networks

A computer network is a system of processors (a.k.a. 'workers' or 'servers'), such as CPUs, GPUs, and/or FPGAs, interconnected via a communication network. Computer networks are typically represented as graphs. The nodes in a computer network are either end point processors, which perform computational jobs, or intermediary network switches, which forward data being communicated between end points around the network. The edges of the graph are communication links along which data can be passed between nodes. The computer network types considered in this thesis include data centres, such as commercial, university, and social media data centres, and HPCs, such as compute systems dedicated to performing large deep learning tasks.

Computer networks have two facets; the *physical plane* and the *orchestration plane* (see Figure 2.1). The physical plane encompasses any *physical device* in the computer network, such as the end point processors, the intermediary switches, and the communication links. The orchestration plane refers to the *resource orchestration schemes* which determine how the physical plane's resources are utilised. These schemes include tasks such as placement (which devices to use) and scheduling (in which order to use the devices). Both the physical and orchestration planes are critical to determining the network's overall performance

in terms of throughput, latency, power consumption, and cost, and both are studied in this thesis.

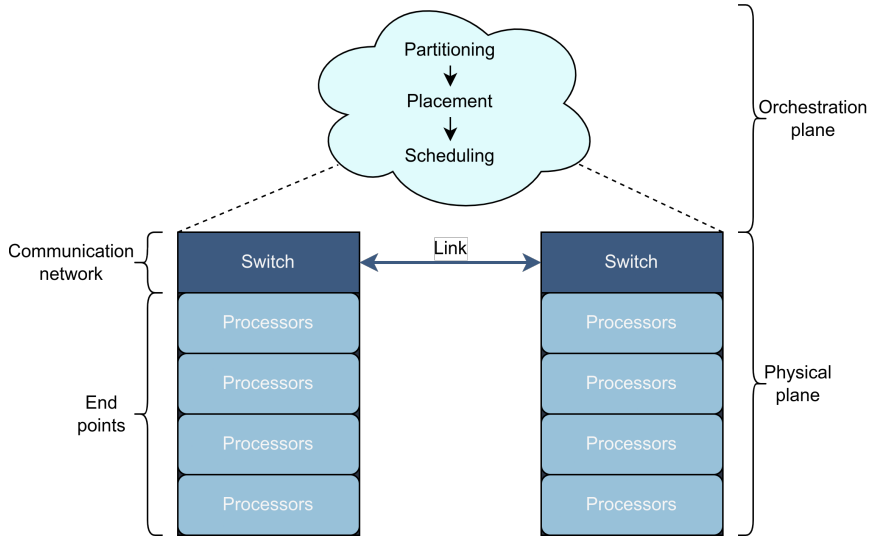


FIGURE 2.1: Visualisation of a computer network divided into the *physical plane* made up of physical devices such as switches, end-point processors, and communication links, and the *orchestration plane* made up of resource management schemes such as job partitioning, scheduling, and placement algorithms.

2.2 Packet vs. Circuit Switching

To communicate information, modern computer networks typically encode data into light by modulating the light's phase or amplitude and transmitting it along a glass fibre. In a single glass fibre, no two messages with the same wavelength and polarisation can be transmitted at the same time. There are two predominant paradigms for addressing this constraint; *circuit switching* and *packet switching*.

Circuit switching. In the vanilla circuit switching paradigm, a direct transmission line is established between source and destination. This line cannot be interrupted, broken, or changed for the duration of the transmission. The data are then streamed as one large block until all data have arrived at the destination. Consequently, as shown in Fig. 2.2a, no other end points can communicate along the same physical transmission line already in use; their

messages must instead be queued until the line is free. Circuit switched networks have the advantage that they can guarantee throughput and latency quality once communication begins, but have the disadvantage that there may be long queuing delays whilst waiting for a suitable transmission line to be free.

Packet switching. In the packet switching paradigm, rather than sending a message as one large block of data, the message is instead split up into smaller 'packets', usually around 500 to 1500 bytes (B) in size. Each packet is labelled with a 'header' indicating its source and destination, and can then be sent via any path through the network to arrive at its destination. A single message's packets do not necessarily need to arrive at their destination via the same route or as a constant stream; as shown in Fig. 2.2b, each packet can be adaptively routed to avoid conflicts. Packet switched networks have the advantage that they can provide low queue times before transmission begins, but have the disadvantages that they cannot provide service guarantees, since buffering may be required at the intermediary switches where the links are fully occupied, and that they may be forced to take inefficient routes around the network.

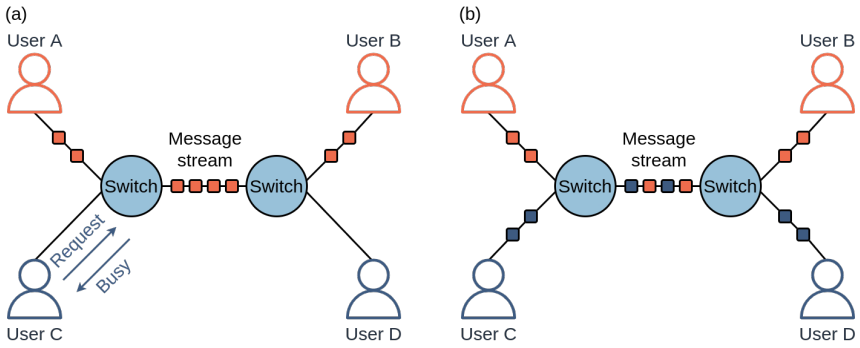


FIGURE 2.2: A visual comparison of the difference between packet and vanilla circuit switching. (a) In vanilla circuit switching, once a physical transmission line is established between source and destination, the line cannot be interrupted or facilitate the transfer of any other data. (b) In packet switching, the data of a single message is split up into multiple 'packets' labelled with the message's source and destination. This allows each packet to take a number of routes along different transmission lines and to be time-interleaved with other messages' packets in order to get to its destination.

Ultra-fast circuit switching. To combine the low latency and high service guarantee benefits of both packet and circuit switching, recent work has

considered ultra-fast circuit switching [Benjamin, 2020]. As with vanilla circuit switching, a communication line is established between source and destination. However, the data are split up into packets and interleaved on short timescales with other transmission requests to enable multiple messages to be transmitted along the same link via time-division multiplexing [Ralph, 1959]. This interleaving is done by rapidly reconfiguring the circuit each time a packet is communicated. The key remaining difference between packet and circuit switching is the decision process of where and when to send messages. In packet switching, this is done on a per-hop basis at each intermediary switch, whereas in circuit switching these decisions are made in advance. Ultra-fast circuit switching therefore retains the benefit of guaranteeing network performance by reserving communication resources for the duration of data transmission whilst also achieving low queuing delays by interleaving data from multiple different sources and destinations along the same physical communication line.

However, a key challenge in realising ultra-fast circuit switching is reconfiguring the logical circuits each time a new packet is to be communicated. For example, as shown in Fig. 2.3, large-scale cloud data centres typically have 91% of packets being $\leq$ 576 B in size (see Fig. 2.3a), which takes $\leq$ 43 ns to transmit along a 100 Gbps link (see Fig. 2.3b). The reconfiguration process must therefore occur on ns timescales in order to achieve acceptable network latency overhead; far lower than the millisecond (ms) scale speeds of classical electronic network switches and software-based schedulers [Benjamin, 2020]. This problem is further addressed in Chapter 3.

2.3 Electronic vs. Optical Networking

Most current computer networks use optic fibre communication links, but the switch devices which interconnect the network are usually electronic. Such networks are hereby referred to as *electronic networks*, as opposed to *optical*

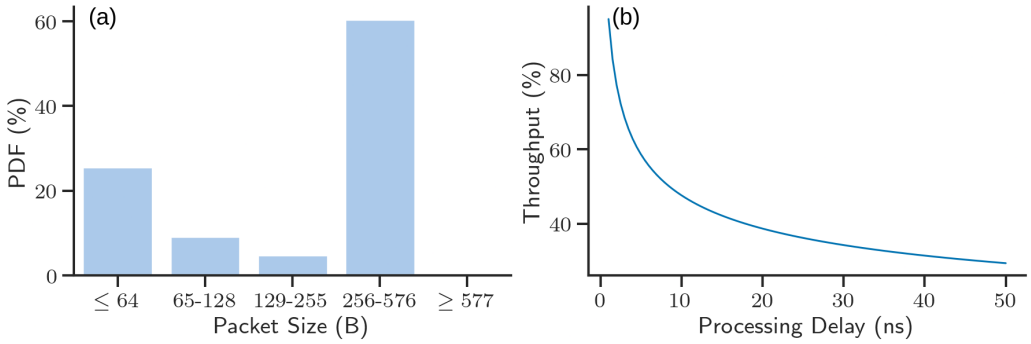


FIGURE 2.3: Meta's cloud data centre (a) packet size distribution and (b) throughput as a function of latency assuming 100 Gbps links [Clark et al., 2018].

networks which are interconnected by optical switches. For further details on how one form of optical switch works, see Chapter 3.

The limitations of electronic networking. Electronic networks have poor scalability, bandwidth, latency, and power consumption. Concretely, the 'Moore's law for networking', that electrical switches double their bandwidth every two years for a fixed power and cost [Ballani et al., 2018], lags behind the annual doubling of cloud traffic bandwidth demands [Shi et al., 2019]. Consequently, rather than upgrading to higher bandwidth switches, electronic networks are typically scaled by adding more switches to build a hierarchy of switch layers (see Fig. 2.4a). This increases the 'oversubscription ratio', which is the ratio of the bandwidth of all servers connected to a switch port to the bandwidth of the port itself. Since the per-port bandwidth of an electronic switch is limited and the power consumption required to cool active electronic devices is expensive, the amount of oversubscription achievable in an electronic network is restricted, thus hampering the network's overall scalability. As shown in Fig. 2.4b, electronic networks have a 'scale tax' where the power, cost, and latency of the network worsens as the network scales [Ballani et al., 2020].

Compounding these current limitations, things are expected to get worse for electronic networks. The 'Moore's law for networking' is expected to undergo a significant slow down beyond 2024 [Ballani et al., 2018]. Furthermore, with the growing requirements of large computational jobs such as training deep learning

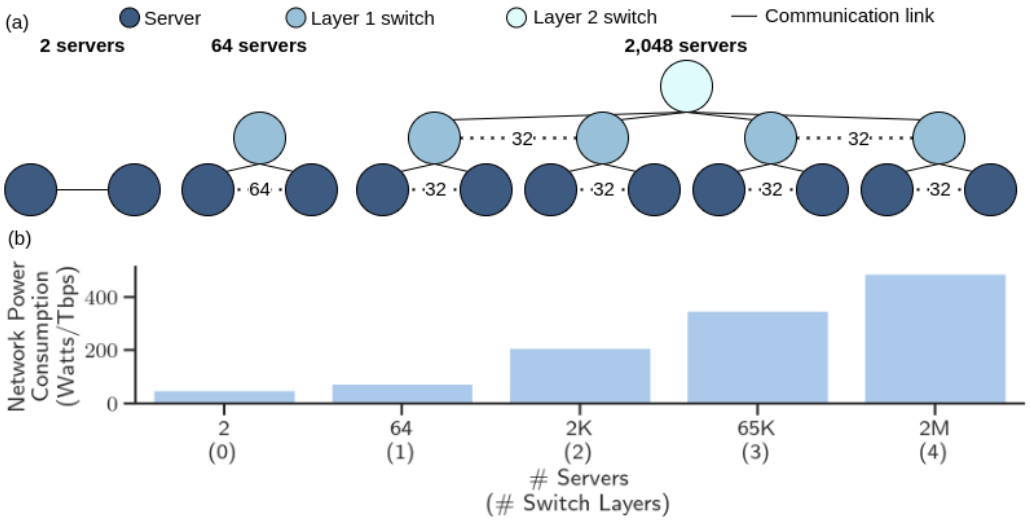


FIGURE 2.4: (a) Visualisation of how an electronic network with 64-port switches is typically scaled; layers are added to the switch hierarchy in order to accommodate more servers, leading to a larger oversubscription ratio. (b) Assuming 400-Gbps per port, how the total power consumed by the electronic network per unit of information communicated increases significantly as the number of switch layers (shown in brackets on the x-axis) is increased [Ballani et al., 2020].

models and processing biological genomes, cloud data centre and HPC demands are not only becoming more distributed and therefore more communication intensive, but are also moving away from being low bandwidth, software-driven CPU workloads towards becoming high bandwidth, hardware-driven GPU, TPU, and FPGA tasks with ultra-low latency requirements [Andreades et al., 2019]. For example, while CPUs rarely saturate 100 Gbps links, GPUs today can process in excess of 2.4 terabits per second (Tbps) of network traffic; a number which is increasing year-on-year [Ballani et al., 2020]. By 2025, the proportion of cloud requests being serviced by CPUs will have decreased by 75% in 2019 to $< 50\%$ in 2025 [McKinsey, 2019]. These factors are creating a perfect storm where the ever-worsening latency, power, cost, and scalability performance of electronic switching is coinciding with an abrupt increase in demand for low latency, high bandwidth computer networks.

The benefits of optical networking. Computer networks with optical switches have the potential to offer significant performance improvements over electronic networks. With a circuit switching implementation, since links are

reserved and unchanged for the duration of a message being communicated from source to destination (see Section 2.2), there is no need for packet inspection, optical buffering, optical-electrical-optical (OEO) conversion for in-switch processing, or mid-transmission contention and blocking, leading to significantly lower latency and power consumption than their electronic and packet switched counterparts [Liu et al., 2015]. Unlike optical packet switching (OPS), optical circuit switching (OCS) networks do not require optical buffering, queuing, or addressing, and are therefore more simple to implement [Benjamin, 2020]. Furthermore, the lack of OEO conversion overhead, the transparency to signal modulation format, and the lower heat generation reduces the number of expensive transceiver components needed, the hardware changes required when new transmission protocols are adopted, and the overall network power consumption compared to electronic networks, making OCS networks lower cost and more energy efficient to operate and upgrade. Additionally, optical switches have significantly higher bandwidth, enabling optical networks to retain low oversubscription ratios and thereby allow more servers to be connected to the same switch without increasing queue times as more switches and switch layers are added to scale the network. Moreover, optical switches are much more physically compact than their electronic counterparts. Having a small 'footprint' is a key design criterion in data centres and HPCs, where it is beneficial to have components close together at high density for the lowest latency. For these reasons, a core theme throughout this thesis is the development of AI-driven optimisation methods to help realise OCS computer networks.

2.4 Computational Complexity

Computational complexity is a key concept in computer science. It describes how much of a given resource is required to run a given algorithm. A problem instance Π has size n, where n might be, for example, the number of binary

digits needed to encode the instance. An algorithm used to solve the problem has a 'worst-cast' time complexity function $O(\cdot)$ which maps the instance size n to the maximum time needed for the algorithm to find a solution to the problem. Common $O(\cdot)$ complexity functions include $O(1)$ (constant time), $O(\log_2(n))$ (logarithmic time), $O(n)$ (linear time), $O(n^k)$ (polynomial time), $O(k^n)$ (exponential time), and $O(n!)$ (factorial time), where k is a constant $k > 1$. Fig. 2.5 visualises how these common complexity functions scale with n.

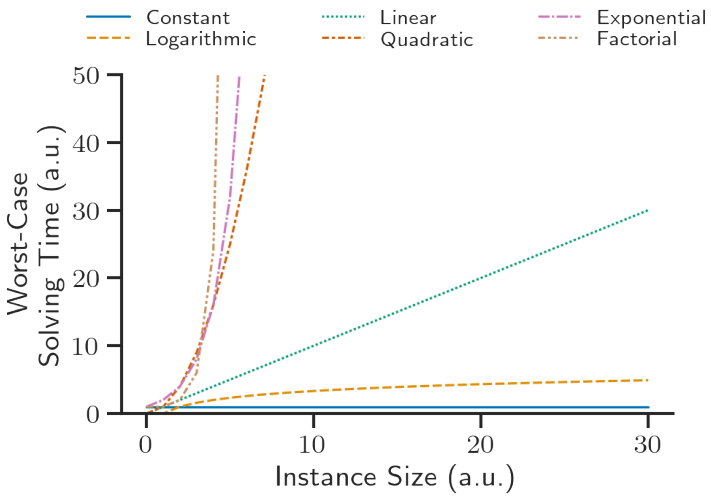


FIGURE 2.5: Visualisation of how common time complexities scale with problem instance size.

A problem for which there exists no known solution algorithm with a polynomial complexity function may take a prohibitively long time to solve. If only brute force solutions exist where every possible solution is explored, the problem is said to be *intractable*. Although most problems can be theoretically solved by brute force algorithms, if the search is not bound by polynomial time, such problems cannot be 'exactly' (provably optimally) solved in practical time frames when scaled to larger instance sizes. How such problems can be solved in practice is explored in Section 2.6.

Building on the notion of computational complexity, decision problems (those with a 'yes' or 'no' answer) can be categorised by *complexity class*, as visualised in Fig. 2.6. A complexity class is a set of problems that a machine can solve

given sufficient time resources. There are many complexity classes, with the four main ones being:

1. **P-problems**: the set of decision problems which can be solved in polynomial time;

2. **NP-problems**: the set of decision problems where the solution's validity can be verified in polynomial time, but where the solution itself cannot be guaranteed to be found in polynomial time;

3. **NP-complete problems**: the set of decision problems X in NP for which there is a polynomial time algorithm to reduce any other NP problem Y to X in polynomial time (therefore if you can solve Y quickly, then you can also solve X quickly); and

4. **NP-hard problems**: the set of decision problems which are at least as hard as NP-complete problems, but which are not necessarily in NP and therefore may not be verifiable in polynomial time.

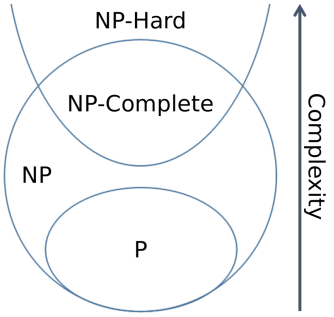


FIGURE 2.6: Euler diagram for the P, NP, NP-complete, and NP-hard complexity classes, assuming $P \neq NP$.

Goldreich [2008] provides a comprehensive overview of computational complexity. NP-complete and NP-hard problems constitute the most difficult to solve problems in computer science, since it is widely thought that $P \neq NP$ [Goldreich, 2010]. Many real-world discrete optimisation problems, such as the computer network optimisation problems considered in this thesis, turn out to be NP-complete or NP-hard.

2.5 Discrete Optimisation

Overview. Optimisation problems are a form of *search* where an optimal solution is being sought amongst some finite or infinite search space. There are two families of optimisation problem; *continuous variable* problems searching for an optimal set of real numbers or a function, and *discrete variable* problems searching for an optimal object (such as an integer, a set, a graph, and so on) from a finite (or countably infinite) set of possible objects. The latter category is colloquially referred to as CO, which is any optimisation problem with at least one decision variable which is subject to the *integrality constraint* (i.e. that its value must be an integer). There are many real-world examples of CO problems such as the travelling salesman problem [Laporte, 1992], finding the shortest path between two nodes in a graph [Johnson, 1973], routing data packets optimally in the internet [Johnson and Maltz, 1996], allocating flight crews to planes [Graf et al., 2020], and many more. All of the computer network optimisation problems addressed in this thesis fall under the category of CO, which also encompasses problems where some of the variables are continuous and some are discrete, and will be mathematically formulated as such.

Problem formulation. An instance of a CO problem Π is a triple (S, f, Ω), where S is a set of candidate solutions to Π, f is the objective function which assigns an objective function value $f(s)$ to each candidate solution $s \in S$, and Ω is a set of problem-specific constraints. Each solution $s \in S$ is made up of a series of m components (variables) $C = \{c_1, c_2, ..., c_m\}$. $\tilde{S}$ is the sub-set of feasible solutions which satisfy Ω, where $\tilde{S} \subseteq S$. CO problems are either maximisation or minimisation problems where the goal is to find the optimal solution $s^* \in \tilde{S}$ which either maximises $(f(s^*) \geq f(s) \forall s \in \tilde{S})$ or minimises $(f(s^*) \leq f(s) \forall s \in \tilde{S})$ the solution's objective function value $f(s)$. For a comprehensive introduction to CO, refer to Papadimitriou and Steiglitz [1982].

Why discrete optimisation problems are difficult. Optimising an

objective function with variables subject to the integrality constraint often turns out to be significantly more difficult than the same problem with the integrality constraints relaxed.

Linear and convex continuous functions are easy to optimise efficiently with algorithms such as simplex [Horen, 1985]. This is because the optimal solution will always reside on the corner points of the boundary of the feasible region (see Fig. 2.7a), and the equations for the feasible boundary are known.

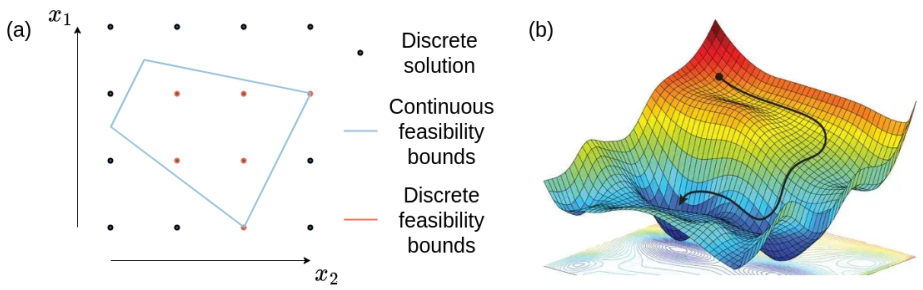


FIGURE 2.7: a) Diagram showing the whole solution space of two decision variables, x_1 and x_2, for a linear convex optimisation problem. In the continuous case, the equations bounding the feasible solution space are known and the optimal solution is guaranteed to reside on one of the boundary's corners. In the discrete case where the two variables must be integers, the equations of the feasibility bounds are unknown and the optimal solution may not necessarily be at corner points. (b) Illustration of how non-convex continuous optimisation tasks can, although by no means trivially, be solved with the use of gradient descent..

As soon as the integrality constraint is introduced, the optimal solution can instead reside anywhere within the feasible region, making methods such as simplex, which only navigate the feasibility region's corner points, inadequate.

Non-linear and non-convex continuous problems can also be inherently easier to solve than the same problem with integrality constraints added (see Fig. 2.7b). This is because the discrete version is non-differentiable, meaning that there is no easy way to know that a step in a given direction of the solution space is (1) feasible and (2) more optimal. Consequently, many combinatorial optimisation problems fall into the NP-complete and NP-hard complexity classes; the most difficult-to-solve problems in computer science. Such problems have no known

algorithm which can deterministically find an optimal solution in polynomial time.

2.6 Solving NP-Hard Problems

There are three classes of algorithms for solving NP-hard problems; *heuristic algorithms*, *approximation algorithms*, and *exact algorithms* (see Table 2.1 for a summary of their characteristics).

Heuristic algorithms. Heuristics are 'rules of thumb'. They generate solutions to combinatorial problems based on principles which are thought to be performant *a priori*. Typically, although not necessarily, heuristics can be scaled to large optimisation problems and can generate solutions quickly. Historically, heuristics have been handcrafted by human experts, but recent years have seen a surge of interest in the application of ML to *learn* heuristics automatically, as done in Chapters 4 and 5. General-purpose heuristic frameworks, termed *metaheuristics*, can be applied to many different problems with little to no tuning or problem-specific adaptations. Examples of metaheuristics include the AI evolutionary and swarm intelligence algorithms developed in Chapter 3. However, heuristics and metaheuristics provide no guarantee on how far the generated solution is from the optimal solution, which can be detrimental in applications where high solution quality is important.

Approximation algorithms. Approximation algorithms apply domain-specific mathematical tricks to certain problems in order to approximate the original complex problem into a simpler version which can be solved exactly. Approximation algorithms are able to provide an optimality bound guarantee on the worst-case distance of the generated solution from the optimal one, such as 'this solution is within at least 92% of the optimal solution', even when the optimal solution itself is not known. Although this guarantee is useful, these approximations are only applicable to certain problem settings and are therefore

not generalisable, and approximation algorithms are typically not scalable to large instances.

Exact algorithms. Exact algorithms are algorithms which, if left to run for long enough, are guaranteed to eventually find the provably optimal solution to a combinatorial problem. Although they generate optimal solutions and are generalisable to many different problems, exact algorithms typically scale poorly and are therefore unable to cope with large instances. As explored in Chapter 4, ML approaches can be integrated into exact solvers in order to improve their scalability whilst retaining their optimality guarantee and generality.

	Approximation Algorithms	Heuristics	Exact Algorithms
Quality guarantee	Yes	No	Yes
Generalisable	No	Yes	Yes
Scalable	No	Yes	No

TABLE 2.1: Summary of the typical characteristics of approximation algorithms, heuristics, and exact algorithms when solving combinatorial optimisation problems.

2.7 Artificial Intelligence

AI is the study and design of 'intelligent agents', where an intelligent agent is a system which perceives its environment and takes actions such that its chances of success are maximised [Poole et al., 1998]. Fig. 2.8 provides a visualisation of some of the branches of AI which are most popular at the time of writing [Mata et al., 2018], with the methods used in this thesis highlighted.

AI can be applied to solving NP-hard discrete optimisation problems via integration into the heuristic, approximation, or exact algorithmic paradigms outlined in Section 2.6. The power of AI techniques stems from the general principle of machine-powered automated problem solving. They are useful for complex problems which cannot be solved either analytically or in a practical time frame, and can often be applied without an expert knowledge of the specific problem domain.

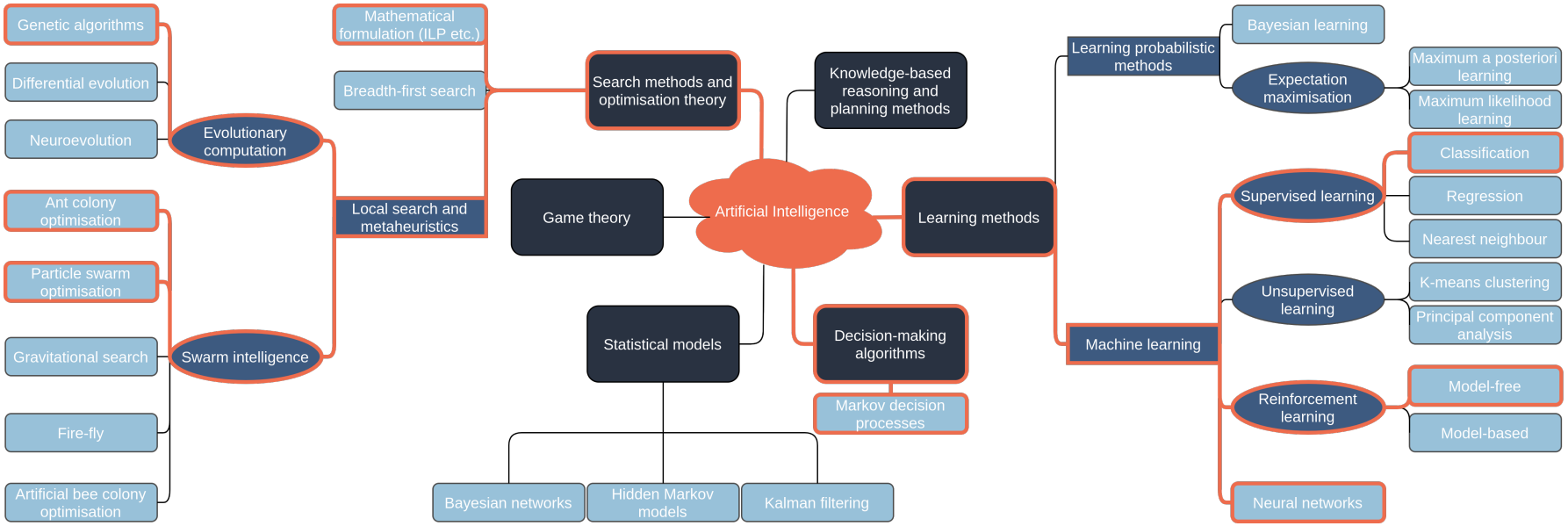


FIGURE 2.8: Main AI branches and sub-categories, with the methods explored and used in this thesis highlighted. This diagram is far from comprehensive, but gives a rough overview of where the methods used in this thesis fit in to the broader AI paradigm.

There are no hard rules specifying which problems are most suited to which AI techniques. The designer must therefore combine their high-level knowledge and intuition about both the problem being solved and the AI techniques available. They must then explore the efficacy and possible adaptation of the narrowed-down techniques for the problem being solved to determine which is best.

This thesis considers optimisation problems in computer networks. The most simple networking environments are those which are deterministic, observable, static, and completely known. Such scenarios are well-suited to search algorithms and classical optimisation theory methods such as breadth-first search and integer linear programming (ILP), which have been applied to simple small-scale network routing and planning problems [Simmons, 2008]. However, when these ideal conditions are relaxed or when the network is scaled to larger sizes, such techniques become inapplicable. Instead, they can be complemented or replaced by more flexible and scalable AI local search algorithms and metaheuristics as shown in Chapter 3. For highly dynamic environments, learning agents which can adapt to new conditions and unforeseen scenarios that could not have been anticipated at the design stage become powerful alternative or complementary problem solving tools. Such learning techniques have been applied to a range of networking problems, including quality of transmission estimation [Jiménez et al., 2012], modulation format recognition [Gonzalez et al., 2010], and task scheduling [Mao et al., 2019a], and are used in Chapters 4 and 5.

2.8 Machine Learning

ML is a sub-field of AI. A machine is said to have ‘learned‘ from experience E to execute a set of tasks T if its performance, as measured by P, improves with experience E [Mitchell, 1997]. The motivation for ML stems from the desire to

have systems which can solve dynamic problems without the need for human intervention or explicit instruction.

There are three broad learning paradigms. (1) **Supervised learning**: 'Labelled' data sets containing the inputs and the corresponding desired outputs are fed into the model, enabling the model to learn to map unseen inputs to seen labels (e.g. classification, regression, and so on). (2) **Unsupervised learning**: 'Unlabelled' data sets containing *only* inputs are fed into the model, enabling the model to learn underlying structures, patterns, or features of the data and thereby reduce unseen data into these learned structures (e.g. clustering, association, and so on). (3) **Reinforcement learning**: Input data is in the form of an observation from a dynamic sequential decision making environment which returns a reward signal to the model based on the actions it chooses to perform, enabling the model to learn how to take actions which maximise its expected long-term reward (e.g. playing chess, controlling complex systems, and so on).

This thesis focuses on the reinforcement learning paradigm in Chapters 4 and 5, although Chapter 4 also implements a supervised learning algorithm as a baseline. For a detailed overview of ML, refer to Russell and Norvig [2009].

2.9 Function Approximation with Neural Networks

In many cases, such as the RL setting where a value and/or policy function is being learned (see Section 2.11), tabular approaches which map inputs (e.g. states) to their exact outputs (e.g. optimal actions) are infeasible because they would require excessive amounts of memory. Instead, the true function mapping inputs to outputs can be *approximated* with a NN, and this has now become common practice in many applications of ML.

Layers. NNs are a composition of linear transforms and non-linear (*activation*) functions connected in a chain to form a directed acyclic computation

graph. Each function in the chain is a *layer* in the NN, although sometimes linear and non-linear function pairs are referred to as a single layer.

Parameterisation. Each NN layer is parameterised by a set of weights and biases. The weights of a layer form a matrix, and the bias values form a vector. Each vector dimension of the weight matrix and each scalar element of the bias vector operate on either all ('fully connected') or some of ('sparsely connected') the input's dimensions being passed to the layer, thus transforming the input into some output (see Fig. 2.9). The weights and bias value operating on a particular (set of) input dimension(s) are collectively termed a 'unit' or 'neuron' in the NN layer. The exact values of the weights and biases of each unit are what determine how the input is mapped to an output.

Learning. Training a NN constitutes learning (i.e. optimising) the set of weights and bias values of each layer to take an input and produce a desired output. For example, a simple learning task might be to find the values of the weights and biases in a NN which accurately map house characteristics (number of bedrooms, surface area, year built, and so on) to the house's price. This learning process is done by defining a *loss function*, such as the mean squared error between the predicted price and the actual price, and minimising the loss by adjusting the model's parameters using an appropriate optimisation algorithm during training. The optimisation algorithm used is typically gradient-based, such as the Adam optimiser [Kingma and Ba, 2015], but may also be gradient-free, such as the AI optimisation methods considered in Chapter 3.

Hidden layers. During training, only the target outputs of the final output layer are given; all intermediary layers between the input and output layers have 'hidden' targets determined by the learning framework being used. Hence, layers between the input and output layers are referred to as 'hidden layers'.

Deep neural networks. NNs with multiple hidden layers are referred to as DNNs. It has been shown that, given enough hidden units (parameters), a NN with only one hidden layer can approximate any continuous function with

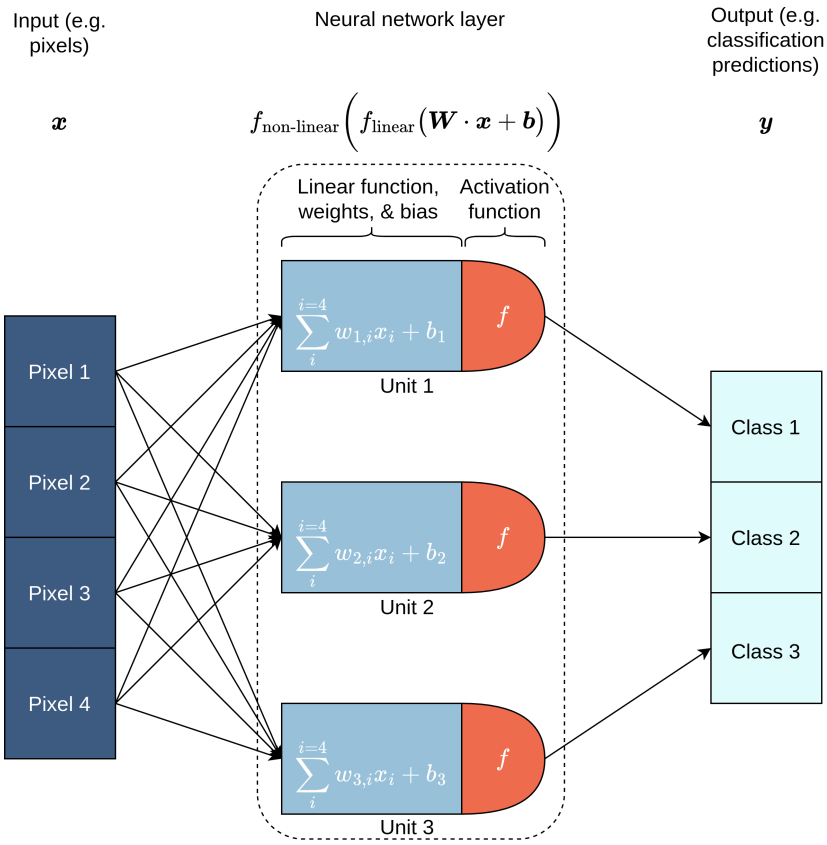


FIGURE 2.9: Visualisation of a typical layer in a fully connected feedforward NN. Each layer is composed of units, which in turn are composed of a linear transform on a set of weights and a bias value followed by a non-linear 'activation' function. In the specific example drawn here, a four-pixel image is flattened into a vector and passed into a single NN layer with three units ('dimensions'). Each unit outputs a single scalar whose value depends on the values of the units' weights and biases. Each unit's output corresponds to the NN's confidence that the image belongs to one of three possible image classes (e.g. dog, cat, or horse). During training, the values of the weights and biases are optimised until the NN successfully maps images to the correct corresponding image class.

a finite number of hidden units [Hornik et al., 1989], therefore making NNs *universal function approximators*. However, to approximate high-dimensional functions, one hidden layer would require exponentially more units as the number of dimensions in the function increases. The power of DNNs lies in their ability to assign feature labels by transforming high dimensional features into linearly separable regions [Osindero, 2018]; indeed, Montufar et al. [2014] showed that as the number of hidden layers is increased, the number of linearly separable regions of the DNNs increases exponentially, whereas increasing the number of hidden

units only increases the assignable feature label count polynomially. Therefore, much more powerful representations are attained with deep and narrow NNs as opposed to shallow and wide models.

The state of neural networks today. The last decade has seen the successful application of the function approximation power of DNNs to solve a variety of problems, from natural language processing [Goldberg and Hirst, 2017] to image recognition [Deng et al., 2009]. In this thesis, DNNs trained within the reinforcement learning paradigm are used to solve difficult combinatorial optimisation problems. For a detailed introduction to deep learning, refer to Goodfellow et al. [2016].

2.10 Graph Neural Networks

This thesis focuses on graph-based combinatorial optimisation problems and computer networks. However, traditional NN architectures cannot easily handle graph-structured data because graphs have no fixed node ordering or reference point, an arbitrarily varying size, a complex topological structure (i.e. no regular spatial locality, as vectors and grids do), and data points (nodes) with multiple features intricately related to other nodes via relationships (edges). Whereas standard NNs are restricted to handling only vector- and grid-structured inputs, such as sentences and images, GNNs are generalised NN architectures which can handle graph-structured data, such as networks and molecular structures (see Fig. 2.10). This thesis therefore uses GNNs for the graph-based ML problems considered in Chapters 4 and 5.

Most current GNNs use the *message passing* paradigm to map each node and edge onto a vector embedding space which captures neighbourhood relationships before performing additional graph-level embeddings and readouts if desired. Concretely, each GNN layer usually performs four stages (see Fig. 2.11): (1) **Message passing**: On each edge in the input graph, use a *message function*

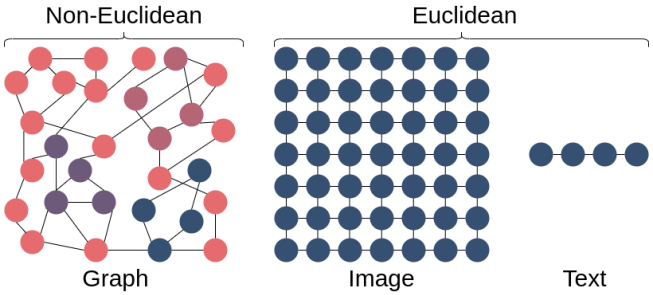


FIGURE 2.10: Visualisation comparing non-Euclidean graph structures, such as networks and molecules, with Euclidean-structured data, such as sentences and images.

to generate a message (representation) to pass from a source node to a set of destination nodes, where each node stores the message(s) it receives in its *mailbox*. (2) **Message aggregation**: On each node in the input graph, apply an aggregate function (a vanilla reduce operation such as mean, sum, max, min, and so on, or a trainable function) to the messages in its mailbox to generate an *intermediate* aggregate representation of its neighbourhood. (3) **Node-level embedding**: Pass the intermediate aggregate representation through a trainable function to produce a *final* vector embedding for each node, which is an aggregated representation of the node and its neighbourhood. (4) **Graph-level embedding** (optional): If desired, at the end of the final GNN layer, pass the node embeddings through a trainable function to produce a graph-level representation.

To include information from k hops away in a given node's embedding and therefore capture k-distance dependencies between nodes across the graph in the final GNN representation, k of these GNN layers can be used. Crucially, the parameters of all message, aggregation, and forward pass functions are shared across nodes, enabling GNNs to be *inductive* in that they can generalise to unseen nodes and graphs.

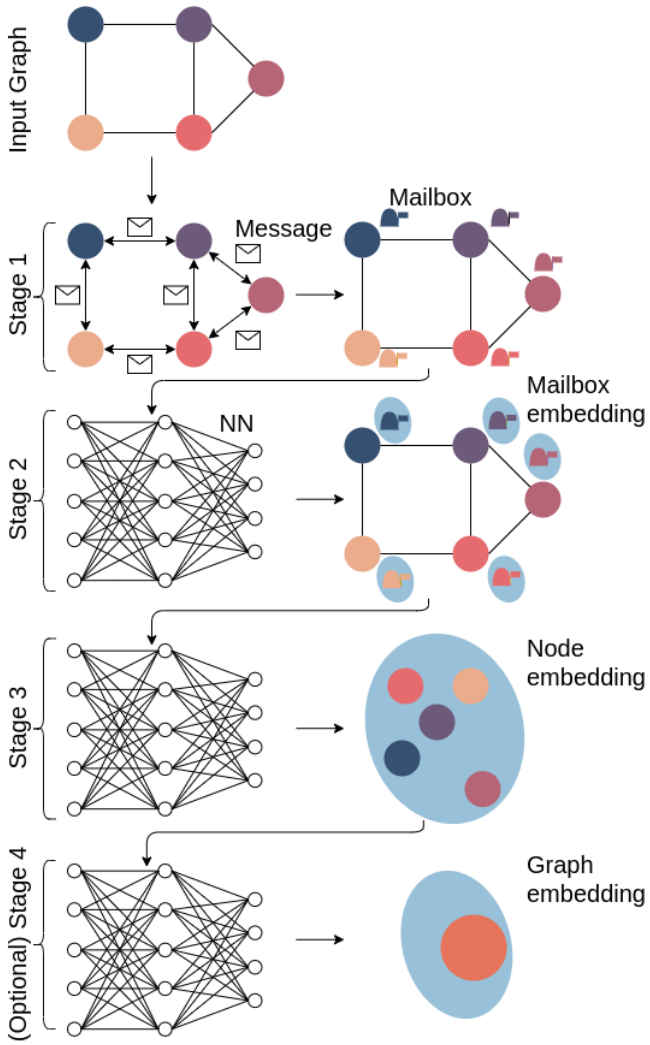


FIGURE 2.11: The stages performed by each layer in a typical GNN. Note that if the optional graph-level readout in stage 4 is performed, it is only done in the final GNN layer.

2.11 Reinforcement Learning

RL is an ML learning paradigm. It is the study of optimal decision making in natural and artificial systems [Silver, 2009]. In the general RL setting shown in Fig. 2.12, an *agent* interacts with an *environment* at each sequential time step t. The environment can be described by tuple $\langle T, R \rangle$, where T is a state transition probability matrix defining the transition probabilities from all states s to all successor states s' taking action u where $T^u_{ss'} = \mathbb{P}(S^{t+1} = s' | S^t = s, U^t = u)$, and R is a scalar reward function giving the expected immediate (next state) reward

given current state s and chosen action u where $R_s^u = \mathbb{E}(R^{t+1}|S^t = s, U^t = u)$.

FIGURE 2.12: A reinforcement learning setting, showing the iterative environment interaction feedback loop used by the agent to learn strategies which maximise the reward signal.

Markov decision process. The environment is usually assumed to have the *Markov property* whereby $\mathbb{P}(s^{t+1}|s^t) = \mathbb{P}(s^{t+1}|h^t)$; that is to say that the probability of the next state being s^{t+1} given the current state s^t is the same as the equivalent probability given all previous states in history $h^t = \{s^1, ..., s^t\}$. As such, this RL setting is usually assumed to be a MDP described by tuple $\langle S, U, T, R, \gamma \rangle$ where S is a finite set of possible environment states, U is either a discrete (finite) or continuous (infinite) set of possible actions, and $\gamma \in [0, 1]$ is a discount factor specifying the factor by which to multiply future expected rewards to discount their present value. Since Markov states are stochastic, future rewards are never fully certain and are therefore expressed as an *expectation*.

Agent goal. The agent's goal is to learn to maximise its expected total discounted future reward, termed the 'value' or 'return' $G^t = \sum_{k=0}^{\infty} \gamma^k R^{t+k+1}$, over the course of an *episode* (a sequence of decision steps which may or may not terminate at some point). To do so, the agent can use *model-free* RL to avoid explicitly modelling the environment by only using its *policy function* and/or its *value function* to make decisions. The policy function π maps an observed state s^t to a corresponding action u^t such that some estimated score objective is maximised. The value function estimates the expected return G_t from being in state s^t and following policy π (the *state value function* v) or from being in state s^t, taking action u^t, and following policy π (the *action value function* q).

Crucially, value and policy functions can be approximated and learned with NNs, enabling RL to be scaled to large problem instances (see Section 2.9 for background information on NNs).

Prediction and control. There are two aspects to maximising G_t; *prediction* and *control*. The prediction task is to, given a policy π and an MDP, find the value function v_π which correctly evaluates how well the agent would do by following π in the MDP. The control task is to, given an MDP, find the optimum value function v_* which maximises the value function over all policies, $v_*(s) = \max_\pi v_\pi(s)$, and the corresponding optimal policy π_* which achieves the optimum value function, $\pi_* \geq \pi, \forall\pi$. Note that in order to solve the control problem (finding the optimum value function and optimal policy), the agent must first solve the prediction problem (finding the value function which correctly evaluates a given policy). There are three classes of model-free RL algorithms for addressing the prediction and control tasks; *value-based*, *policy-based*, and *actor-critic* RL.

Value-based RL. Value-based methods involve learning a value function which *implicitly* defines a policy by following a policy, such as ϵ-greedy [Tokic and Palm, 2011], which is based on the expected returns predicted by the value function. Examples of common value-based algorithms include SARSA [Rummery and Niranjan, 1994] (*on-policy* learning), Q-learning [Watkins and Dayan, 1992] (*off-policy* learning), and DQN [Mnih et al., 2015] (Q-learning with *experience replay*), and are explored in Chapters 4 and 5. Value-based methods can be advantageous in environments with small action spaces since value function updates tend to be large and therefore achieve rapid convergence. However, since a value function must be defined, the maximum value of all possible state-action pairs must be found. This is often inefficient, since usually the agent would only want the best action rather than knowing the value of all state-action pairs, and cannot be used for large or continuous (infinite) action spaces. Furthermore, the policy typically sampled deterministically at test time

with the action with the highest value being preferred, which is disadvantageous for certain scenarios such as rock-paper-scissors where a random policy beats any deterministic policy [Bowling and Veloso, 2001] or in some partially observable Markov decision process (POMDP) problems such as the aliased grid-world problem [Crook and Hayes, 2003] which benefit from having stochastic elements in the policy.

Policy-based RL. Policy-based methods such as REINFORCE [Williams, 1992] do not consider a value function. Instead, they *explicitly* define a policy and directly learn the policy which maximises their expected return. Policy updates are often small and therefore converge more smoothly on an optimum policy. Furthermore, since there is no value function, the agent does not need to consider the value of every possible state-action pair, therefore policy-based algorithms can be scaled to very large or continuous action spaces; this is the most common motivation for researchers to use policy-based methods. Additionally, they can learn stochastic policies, which can be good for certain POMDPS. On the other hand, due to the small policy updates at each iteration, policy-gradient methods are vulnerable to long convergence times and stagnation at local minima rather than finding the global optimum policy. Furthermore, policy-based algorithms typically use Monte-Carlo methods without bootstrapping and therefore the agent does not get a reward until the end of an episode, thereby potentially introducing high variance and making learning difficult.

Actor-critic RL. Actor-critic methods are a new class of RL algorithms which are becoming increasingly popular. They explicitly define both a policy function (the *actor*) and a value function (the *critic*) and learn to optimise them both in order to try to get the advantages of both value- and policy-based methods. Common actor-critic algorithms include natural policy gradient [Kakade, 2001], A2C, A3C and Q-actor-critic [Mnih et al., 2016], deep deterministic policy gradient [Lillicrap et al., 2016], and proximal policy optimisation (PPO) [Schulman et al., 2017], and are considered in Chapters 4 and 5.

Advantages of RL. Using traditional RL has several advantages over heuristics and other ML paradigms such as supervised learning. First, no external data from human-designed or computationally expensive heuristics is required, enabling an agent to learn super-human policies without potentially sub-optimal initial biases towards a certain strategy or a costly expert example collection-and-labelling phase [Silver et al., 2016]. Second, a DNN with a finite number of layers and neurons will have its expressivity constrained [Dong et al., 2020], restricting the complexity of the set of functions it is capable of approximating. Because the objective of an RL agent is to maximise its expected future return which, under the assumption that a suitable reward function has been crafted, is equivalent to maximising performance on a given task, RL agents are able to maximise task performance given DNN expressivity constraints. Third, since RL agents maximise the *future* return, they are capable of learning sophisticated non-myopic policies which sacrifice short-term reward in exchange for higher long-term return [Sutton and Barto, 2018].

The state of reinforcement learning today. RL has been an established field for a long time. However, recent breakthroughs over the last decade in the development of large DNN models have yielded impressive results when trained in the RL paradigm. Mnih et al. [2015] showed that a DNN trained to approximate the value function via Q-learning can be used to play Atari games with super-human performance, and many 'deep RL' algorithms have since been developed. AlphaGo represented a significant milestone, being the first Go computer programme to beat professional human players on a full 19×19 board [Silver et al., 2016], combining deep reinforcement learning with Monte Carlo tree search [Browne et al., 2012]. Recently, RL has surpassed humans at complex real-time partially observable strategy games such as StarCraft [Vinyals et al., 2019], Dota [OpenAI et al., 2019], and Poker [Brown and Sandholm, 2019]. There have also been several breakthroughs in real-world applications, with OpenAI training a robot to solve a physical rubik's cube [Agostinelli et al.,

2019a] and Kiran et al. [2022] demonstrating a self-driving learning agent. This progress in RL presents a ripe opportunity for its application in discrete problem solving and computer network optimisation, as explored in Chapters 4 and 5. For a detailed introduction to RL, refer to Sutton and Barto [2018].

2.12 Deep Q-Learning

Deep Q-network (DQN) (also known as 'deep Q-learning') is a state-of-the-art value-based RL algorithm [Mnih et al., 2013, 2015], and some of its variants are used in Chapers 4 and 5. This section breaks down the components of this popular RL method.

Q-learning. Q-learning [Watkins, 1989] is the canonical value-based algorithm which can be applied to a sequential decision making process formalised as an MDP (see Section 2.11). It is an off-policy temporal difference algorithm whose goal is to learn the action value function mapping state-action pairs to their expected discounted future return when following a policy π; $Q^\pi(s,u) = \mathbb{E}_\pi\Big[\sum_{t'=t+1}^{\infty} \gamma_{t'-1} r(s_{t'}) | s_t{=}s, u_t{=}u\Big]$. By definition, an optimal policy π_* will select an action which maximises the true Q-value $Q_*(s,u)$, $\pi_*(s) = \arg\max_{u'} Q_*(s,u')$.

Concretely, the classical Q-learning algorithm maintains an action value look-up table $Q(s,u)$ mapping all possible state-action pairs to their predicted discounted return. The return is the sum of future rewards over the remainder of the episode. During training, Q-learning follows an exploration-exploitation policy. The simplest such policy is ϵ-greedy, where a random action is sampled with probability $\epsilon \in [0,1]$ and the best action, according to the current Q table, is sampled with probably $1-\epsilon$. At each time step t, the agent in state s_t uses this policy to select an action u_t which it performs in the environment to transition to the next state s_{t+1} and receive a reward r_{t+1}. $Q(s,u)$ is then updated according to:

$$Q(s_t, u_t) \leftarrow Q(s_t, u_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_{u'} Q(s_{t+1}, u') - Q(s_t, u_t) \right). \tag{2.1}$$

On the right-hand side of Eq. 2.1, $Q(s_t, u_t)$ is the agent's estimate of the discounted return of taking action u_t in state s_t, α is the learning rate, γ is the factor by which to discount future rewards to their present value, and $\max_{u'} Q(s_{t+1}, u')$ is an estimate of the future value of being in state s_{t+1} and taking an 'optimal' action according to Q. The $r_t + \gamma \cdot \max_{u'} Q(s_{t+1}, u')$ term is called the *temporal difference target*, and the collective $r_t + \gamma \cdot \max_{u'} Q(s_{t+1}, u') - Q(s_t, a_t)$ term the *temporal difference error*. As such, the $\max_{u'} Q(s_{t+1}, u')$ term treats Q as an oracle from which optimal actions can be sampled. Although Q is usually randomly initialised and changes at each update step, the general idea is that, with stable learning and sufficient exploration, Q will converge on the true Q_* function.

As a side note, Q-learning is a *temporal difference* algorithm because, rather than using the actual returns to update Q in Eq. 2.1 as done by *Monte Carlo* methods, it uses a bootstrapped estimate of the future returns $\max_{u'} Q(s_{t+1}, u')$. Furthermore, it is an *off-policy* algorithm because the policy used to select the action u_t at the current time step, such as ϵ-greedy sampling of Q, is different to the policy used to select the next-state action u' when evaluating the temporal difference target, such as greedy sampling of Q. This is as opposed to *on-policy* temporal difference algorithms, such as SARSA, which use the same action selection policy for both the current time step and for future time steps when bootstrapping.

Deep Q-learning. Many practical problems have an extremely large number of possible state-action combinations. For example, the game of Go has over 10^{700} possible sequences; far more than the number of atoms in the universe [Silver et al., 2016]. As such, modelling the action value function with a tabular

approach is intractable given practical memory constraints. To enable Q-learning to be scaled to complex problems, DQN [Mnih et al., 2013] approximates the true Q-function with a DNN parameterised by θ such that $Q_\theta(s,u) \approx Q_*(s,u)$.

Concretely, during training at each time step t, $Q_\theta(s,u)$ is used with an exploration strategy such as ϵ-greedy to select an action and add the observed transition $T = (s_t, u_t, r_{t+1}, \gamma_{t+1}, s_{t+1})$ to a replay memory buffer [Lin, 1992]. The network's parameters θ are then optimised with stochastic gradient descent to minimise the mean squared error loss between the *online* network's predictions and a bootstrapped estimate of the Q-value,

$$J_{DQN}(Q) = \Big[r_{t+1} + \gamma_{t+1} \max_{u'} Q_{\bar{\theta}}(s_{t+1}, u') - Q_\theta(s_t, u_t)\Big]^2, \tag{2.2}$$

where t is a time step uniform randomly sampled from the buffer and $Q_{\bar{\theta}}$ a *target* network with parameters $\bar{\theta}$ which are periodically copied from the acting online network. The target network is not directly optimised, but is used to provide the bootstrapped Q-value estimates for the loss function. Only periodically updating the target network rather than at each learning step leads to lower variance in the bootstrapped targets at each step. This helps helps to stabilise learning and leads to better convergence [Mnih et al., 2013].

Double DQN. In the traditional Q-learning update rule of Eq. 2.1 and the DQN loss of Eq. 2.2, the Q-function used to select and evaluate an action for the temporal difference target is the same; $\max_{u'} Q(s_{t+1}, u')$ for Eq. 2.1, and $\max_{u'} Q_{\bar{\theta}}(s_{t+1}, u')$ for Eq. 2.2. However, this can lead to an *overestimation bias* where the chosen action u' is incorrectly over-valued because the same function which perceives u' as being best is also being asked to evaluate it. This can lead to high variance updates, unstable learning, and convergence on local minima. Double DQN [van Hasselt et al., 2015] reduces overestimation by decomposing the `max` operation in the temporal difference target into *action selection* and *action evaluation* and performing these two tasks with two separate networks.

Concretely, action u' is greedily selected according to the online network Q_θ and evaluated with the separate target network $Q_{\bar{\theta}}$. The loss term from Eq. 2.2 then becomes:

$$J_{DDQN}(Q) = \Big[r_{t+1} + \gamma_{t+1} Q_{\bar{\theta}}(s_{t+1}, \max_{u'} Q_\theta(s_{t+1}, u')) - Q_\theta(s_t, u_t)\Big]^2. \tag{2.3}$$

Prioritised experience replay. Vanilla DQN replay buffers are sampled uniformly to obtain transitions for network updates. A preferable approach is to more frequently sample transitions from which there is much to learn. Prioritised experience replay [Schaul et al., 2016] deploys this intuition by sampling transitions with probability p_t proportional to the last encountered absolute temporal difference error,

$$p_t \propto |r_{t+1} + \gamma_{t+1} \max_{u'} Q_{\bar{\theta}}(s_{t+1}, u') - Q_\theta(s_t, u_t)|^\omega, \tag{2.4}$$

where ω is a tuneable hyperparameter for shaping the probability distribution. New transitions are added to the replay buffer with maximum priority to ensure all experiences will be sampled at least once to have their errors evaluated.

n-step Q-learning. Traditional Q-learning uses the target network's greedy action at the next step to bootstrap a Q-value estimate for the temporal difference target. Alternatively, to improve learning speeds and help with convergence [Sutton and Barto, 2018, Hessel et al., 2017], forward-view *multi-step* targets can be used [Sutton and Barto, 2018], where the n-step discounted return from state s is

$$r_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} r_{t+k+1}, \tag{2.5}$$

resulting in an n-step DQN loss of

$$J_{DQN_n}(Q) = \left[r_t^{(n)} + \gamma_t^{(n)} \max_{u'} Q_{\bar{\theta}}(s_{t+n}, u') - Q_\theta(s_t, u_t)\right]^2. \tag{2.6}$$

Dueling DQN. Traditional DQN approaches use a DNN architecture which is not specific to RL. Consequently, when learning the Q-function, the entire DNN architecture must learn to estimate the state value *and* the action advantage for each action in order to learn the state-action function $Q^\pi(s, u)$ of being in state s, taking action u, and following policy π. However, in many problems where bootstrapped Q-learning is applied, the most important objective is to learn to estimate the value of each state rather than the effect of each action for each state. This is especially true in environments and individual states where future transitions are mainly influenced by factors other than the agent's actions.

Leveraging the insight that in many states it is unnecessary to estimate the value of each action choice, Wang et al. [2015] developed a new DNN architecture, termed 'dueling DQN', which is better suited to the Q-learning task. Concretely, the dueling architecture uses the same core DNN as standard DQN. However, rather than following the initial encoding with a single sequence of fully connected layers to get a Q-value for each possible action in the current state, dueling DQN uses two separate streams of fully connected layers. One stream, parameterised by β, estimates the state value function $V_{\theta,\beta}(s)$ (the estimated future discounted return of the current state regardless of future actions taken), and the other stream, parameterised by α, estimates the relative action advantage function $A_{\theta,\alpha}(s, u)$ (the relative difference in the future discounted return of each action).

The outputs of the two streams are then combined via a special aggregation function to recover the state-action value function Q. Crucially, $V(s)$ and $A(s, u)$ must be combined into $Q(s, u)$ in such a way that they are independently identifiable from the output Q values alone in order for backpropagation to be able to calculate the appropriate loss and weight updates for the separate $V(s)$ and $A(s, u)$ streams. As such, a simple $Q(s, u) = V(s) + A(s, u)$ aggregation

function to get the Q-values from the two streams does not suffice. Instead, the authors tried two different aggregation schemes.

The first aggregation method subtracted the advantage of the *maximum* advantage action from all advantages to make the `argmax` action's advantage 0 and the rest < 0,

$$Q_{\theta,\alpha,\beta} = V_{\theta,\beta}(s) + \left(A_{\theta,\alpha}(s,u) - \max_{u'} A_{\theta,\alpha}(s,u') \right), \tag{2.7}$$

thus enabling $V(s)$ to be recovered at the `argmax` action's Q-value.

The second aggregation method subtracted the *mean* advantage from all action advantages to centre the advantage values around 0 (i.e. to have a mean of 0),

$$Q_{\theta,\alpha,\beta} = V_{\theta,\beta}(s) + \left(A_{\theta,\alpha}(s,u) - \frac{1}{|A|} \sum_{u'} A_{\theta,\alpha}(s,u') \right). \tag{2.8}$$

This makes $V(s)$ recoverable from $Q(s,u)$ by estimating the $V(s)$ value which, when subtracted from each $A(s,u)$ value, leads to a set of $A(s,u)$ values which have a mean of 0. In practice, this second approach of using the mean was found to lead to more stable learning since using a `mean` operation resulted in lower variance targets between learning steps compared to when a `max` operation was used.

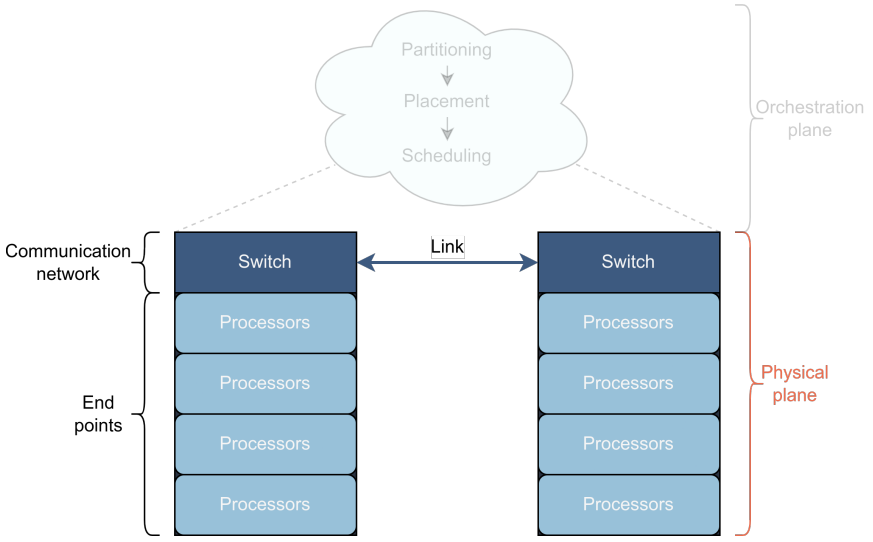
As with standard Q-learning, the output of the dueling network is a set of Q-values (one for each action), therefore no change to the underlying algorithm other than a slight adjustment of the network architecture was required. By decomposing the Q-function approximator in this way, dueling DQN is able to attain superior policy evaluation in the presence of many similar-value actions, and the authors demonstrated their architecture achieving state-of-the-art performance on the Atari 2600 games.

Ape-X DQN. Noting that state-of-the-art ML performance is often achieved with more computation, more powerful models, and larger training data sets,

Horgan et al. [2018] proposed Ape-X; a parallelisation approach to off-policy experience replay RL. Concretely, rather than using a single actor-learner setup, Ape-X decouples acting from learning. It distributes many actors across a set of CPU cores each with their own instance of the environment. Each actor retains a copy of a DNN shared across actors which it uses for action selection to accumulate experiences in parallel with other actors. These experiences are then communicated to a central shared replay buffer, where a single learner mounted on a GPU uses prioritised experience replay to sample the most important experiences for learning. Learner sampling, gradient computation, and network updates are done asynchronously with one another on separate threads, as are the periodic updates made to the actors' networks with the latest shared learner network. By using multiple actors in parallel, not only can orders of magnitude more transition data be attained for learning, but also a broader diversity of experiences can be collected by allocating a different exploration strategy to each actor and thereby avoid local optima in difficult exploration and large state-action space settings. For N_{actors} distributed actors, Horgan et al. [2018] used a per-actor ϵ-greedy exploration strategy whereby each actor i had a fixed exploration probability $\epsilon_i = \epsilon^{1+\frac{i}{N_{actors}-1}\cdot\alpha}$ where $\epsilon = 0.4$ and $\alpha = 0.7$. The authors demonstrated their approach achieving new state-of-the-art results on Atari in a fraction of the training time of prior works.

Part I

Optimising the Physical Plane



Chapter 3

SOA Control for Sub-Nanosecond Optical Switching

Abstract

Novel approaches to switching ultra-fast semiconductor optical amplifiers using artificial intelligence algorithms (particle swarm optimisation, ant colony optimisation, and a genetic algorithm) are developed and applied both in simulation and experiment. Effective off-on switching (settling) times of 542 ps are demonstrated with just 4.8% overshoot, achieving an order of magnitude improvement over previous attempts described in the literature and standard dampening techniques from control theory.

Publications related to this work (contributions indented):

- Hadi Alkharsan, **Christopher W. F. Parsonson**, Zacharaya Shabka, Xun Mu, Alessandro Ottino, and Georgios Zervas, 'Optimal and Low Complexity Control of SOA-Based Optical Switching with Particle Swarm Optimisation', *ECOC'22: Proceedings of the Forty-Eighth European Conference on Optical Communication*, 2022
  - PSO code
- Thomas Gerard, **Christopher W. F. Parsonson**, Zacharaya Shabka, Benn Thomsen, Polina Bayvel, Domanic Lavery, and Georgios Zervas, 'AI-Optimised Tuneable Sources for Bandwidth-Scalable, Sub-Nanosecond Wavelength Switching', *Optics Express*, 2021
  - PSO code, simulation & lab experiments, plots
- **Christopher W. F. Parsonson**, Zacharaya Shabka, W. Konrad Chlupka, Bawang Goh, and Georgios Zervas, 'Optimal Control of SOAs with Artificial Intelligence for Sub-Nanosecond Optical Switching', *Journal of Lightwave Technology*, 2020
  - ACO/PSO/GA algorithm selection, PSO code, simulation & lab experiments, paper writing, plots
- Thomas Gerard, **Christopher W. F. Parsonson**, Zacharaya Shabka, Polina Bayvel, Domanic Lavery, and Georgios Zervas 'SWIFT: Scalable Ultra-Wideband Sub-Nanosecond Wavelength Switching for Data Centre Networks', *arXiv*, 2020
  - PSO code, simulation & lab experiments, paper writing, plots

3.1 Introduction

The challenge of all-optical switching in DCNs stems from the short bursty nature of DCN traffic and the lack of an all-optical memory alternative to traditional storage techniques during buffering and contention resolution. The small packets which dominate DCN traffic (90% $<$ 576 bytes [Clark et al., 2018]) take only O(microsecond (μs)) to transfer. Therefore, to avoid an inefficient network with a switching time that is comparable to or greater than the forwarding time, OCS networks must be switched at O(ns) packet timescales [Benjamin et al., 2020]. However, current state-of-the-art commercial optical switches have $O(> 100\mu s)$ switching times [Farrington et al., 2010, Hamedazimi et al., 2014, Gray et al., 2015, Mellette et al., 2017, Webster, 2022].

A promising technology for high-speed all-optical switching is the SOA. SOAs have high and relatively flat optical gain bandwidths and can therefore be used for both space and wavelength switching [Assadihaghi et al., 2010]. They also have inherently fast switching times (theoretically limited only by their 100 picosecond (ps) carrier recombination lifetimes [Connelly, 2003]), a high extinction/optical contrast ratio, and a relatively compact design. These characteristics make SOAs an ideal candidate for low latency-, scalability-, and footprint-constrained DCN switching.

However, SOAs have an intrinsic optical overshoot and oscillatory response to electronic drive currents due to exciton density variations and spontaneous emission in the gain region [Paradisi, 2019]. This results in the key advantage of SOA DCN switching (rapid switching times) being negated, preventing sub-ns switching. Prior attempts to fix these faults have failed to achieve sub-ns switch times and cannot be scaled (see Section 3.3).

In this chapter, we propose a novel and scalable approach to optimising the SOA driving signal in an automated fashion with three AI techniques; GA, ACO, and PSO. These algorithms were chosen on the basis that they had previously

been applied to PID tuning in control theory [Kusuma et al., 2016]. Moreover, AI techniques propose the benefit of not requiring prior knowledge of the SOA and therefore provide a means of developing an optimisation method that is generalisable to any SOA-based switch. All algorithms were shown to reduce the settling and rise times to the $O(100\text{ ps})$ scale, and we experimentally demonstrate an order of magnitude improvement over the previous switching speed world record. The algorithms' hyperparameters were tuned in an SOA equivalent circuit simulation environment and their efficacy was demonstrated in an experimental setup. AI performance was compared to that of both standard and state-of-the-art literature approaches to optimising oscillating and overshooting systems, all of which the AI algorithms outperformed. Of the AI algorithms, PSO was found to have both the best performance and generalisability due to the additional hyperparameters and search space restrictions that were required for GA and ACO. All code and plotted data are freely available at Parsonson et al. [2020a] and Parsonson et al. [2020b] respectively.

3.2 Background

3.2.1 Semiconductor Optical Amplifiers

SOA physics. The basic principle of an SOA's operation is shown in Fig. 3.1. A *gain* (or *active*) region is sandwiched between an *n-type* material, which has many electrons in its *conduction band*, and a *p-type* material, which has many holes in its *valence band*. A driving pump current is applied via metal electrodes to supply excesses of electrons and holes to the n- and p-type materials respectively. The excess holes and electrons pass into the gain region's valence and conduction bands respectively, forming electron-hole pairs called *excitons*. An optical input signal is pumped into the gain region, stimulating electrons to recombine with holes by *stimulated emission* (see Fig. 3.2). This relaxation causes a photon

of energy equal to the gain region's *band gap* to be released. This band gap is chosen to be equal to the input optical signal's wavelength, therefore the SOA's stimulated emission process increases the intensity of the optical signal, thus creating an *amplified* optical output signal. Refer to Connelly [2003] for more details on SOAs.

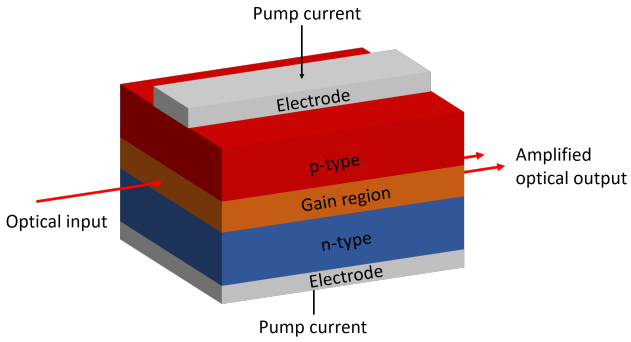


FIGURE 3.1: Schematic of an optical amplification device, such as a semiconductor optical amplifier (SOA). An optical input signal is amplified in the gain region by the process of stimulated emission, thereby outputting an optical signal with higher intensity. This is an 'all-optical' process.

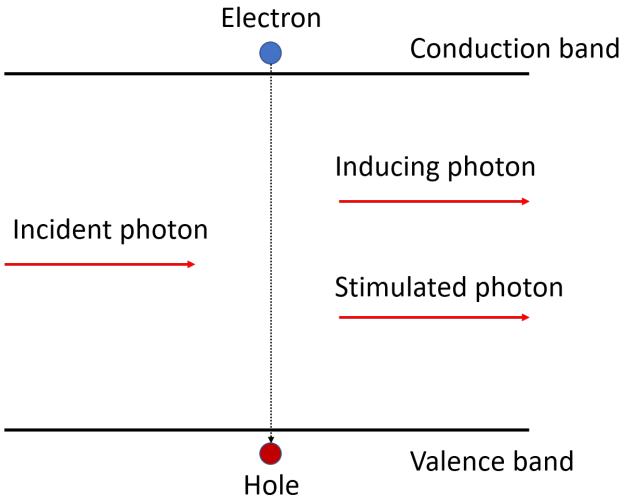


FIGURE 3.2: Schematic of the process of stimulated emission; the key phenomenon behind optical amplification. An incident 'inducing' photon stimulates the relaxation and recombination of an electron and a hole, thus stimulating the emission of a photon of energy equal to the bandgap across which the electron relaxed.

Optical switching with SOAs. SOAs have great potential as an ultra-fast all-optical switch technology [Connelly, 2003, Assadihaghi et al., 2010]. Consider

a simple $n \times n = 2 \times 2$ switch module, as shown in Fig. 3.3. In this case, there are $n = 2$ input fibres and $n = 2$ output fibres, therefore a switch is needed that can switch either of the two input fibres to either of the two output fibres. To create such a switch, $n^2 = 4$ SOAs are needed, with $n^2 = 4$ optic fibres connecting $n = 2$ input fibres to $n = 2$ output fibres having one SOA each. The light from each input fibre is split into $\frac{1}{n} = \frac{1}{2}$ intensity beams using a decoupler (in this case, a 50:50 decoupler). Each SOA is either 'on' (an electrical pump current is applied) or 'off' (no electrical pump current applied). In the 'on' state, there is a sufficient density of excited states in the active region for stimulated emission to be the dominant phenomenon occurring in the SOA. The light passes through the SOA, and the gain region amplifies the light signal by n times, thus re-amplifying the signal to its original intensity and compensating for the intensity loss incurred by splitting the signal upon entry to the switch. In the 'off' state, there are not enough excited states in the active region for stimulated emission to dominate. As such, the SOA has a high *extinction ratio* (the ratio of photons absorbed to photons emitted), and the majority of the photons are absorbed by the process of *stimulated absorption* (see Fig. 3.4). The end result is an all-optical switch that can route light to any output port from any input port by turning on the corresponding SOA.

Advantages of SOA switching. There are three primary advantages of using SOAs for optical switching. (1) *Ultra-fast switching:* In theory, the switching speed of an SOA is limited only by the *carrier recombination lifetime* (how long, on average, it takes a hole and an electron to recombine), which is O(ps). As such, SOA switches open up the possibility of ultra-fast sub-ns switching, thereby reducing network latency and enabling OCS architectures. (2) *Scalability:* The loss in signal intensity across each line in the switch with its corresponding SOA is constant across all lines/SOAs, and does not increase as the number of input/output ports is increased, thus enabling significant port

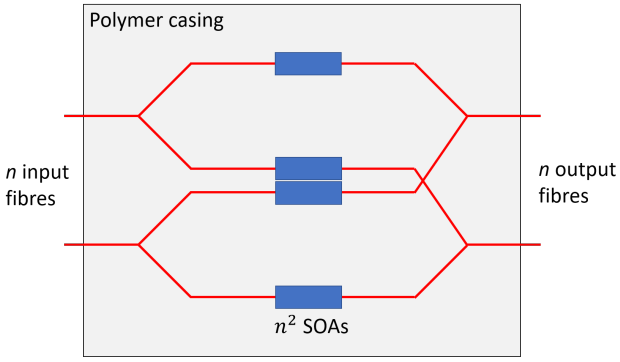


FIGURE 3.3: Schematic of how SOAs can be used to create an all-optical switch. Input light signals are split up and passed along all the possible routing paths. The SOA along the routing path corresponding to the desired output fibre that the input signal should be routed to is switched on, and all other SOAs are switched off. The SOA that is switched on re-amplifies the split signal by stimulated emission and allows it through to the output port. The SOAs that are switched off absorb the signal by stimulated absorption. All the fibres and SOAs are held in a polymer casing.

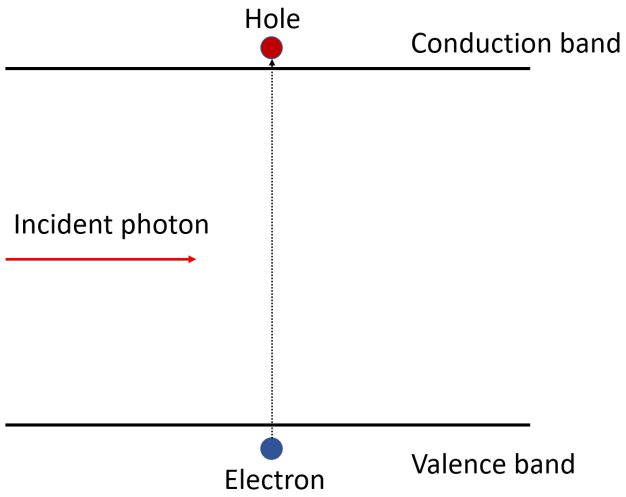


FIGURE 3.4: Schematic of the process of stimulated absorption. An incident photon passes its energy on to an electron in the valence band, exciting an electron to the conduction band.

scalability. (3) *Low power consumption:* Since SOA switching is a a passive all-optical process which generates little to no heat, therefore the amount of power needed to cool the switches is reduced. This significantly reduces the power consumption and running cost of the network relative to electronic switches and other optical switching technologies.

Challenges of SOA switching. Despite these advantages, SOAs present two key challenges preventing their application to switching; *power overshoot* and *power ringing* (see Fig. 3.5 for a visualisation of these phenomena). (1) *Power overshoot:* An intrinsic SOA response to an injection of current is to rapidly create excitons. The density of these excited states is initially high before falling to a more constant level. Therefore, there is an initial 'power overshoot' in the optical output power of an SOA switch; rather than a signal being a perfect off-on step function, it will initially overshoot the 'on' power state. This can lead to non-linearity problems related to the optical power in the transmission fibre. (2) *Power ringing:* When the SOA is turned 'on' by applying a pulse of electrical current, after the initial power overshoot, some excited carriers in the gain region begin to relax by *spontaneous emission* (see Fig. 3.6). This lowers the carrier density, therefore lowering the level of stimulated emission and causing a reduction in SOA gain, thus reducing the power output of the SOA. The carriers are then re-excited by the present current pulse, therefore increasing the gain again, but with the aforementioned power overshoot. Eventually after oscillations/ringing in the power output, the SOA gain settles on a constant level, but this initial ringing (which occurs for a period of time known as the *settling time*) leads to distortions in the signal being transmitted, and therefore cannot be used. As a result, although the on-off time of an SOA is quick (on the order of picoseconds), the effective on-off time (which must account for the settling time) is much slower (on the order of nanoseconds).

In this chapter, we address these two challenges by applying AI techniques to optimising the driving signal to the SOA such that the negative effects of power overshoot and ringing are minimised.

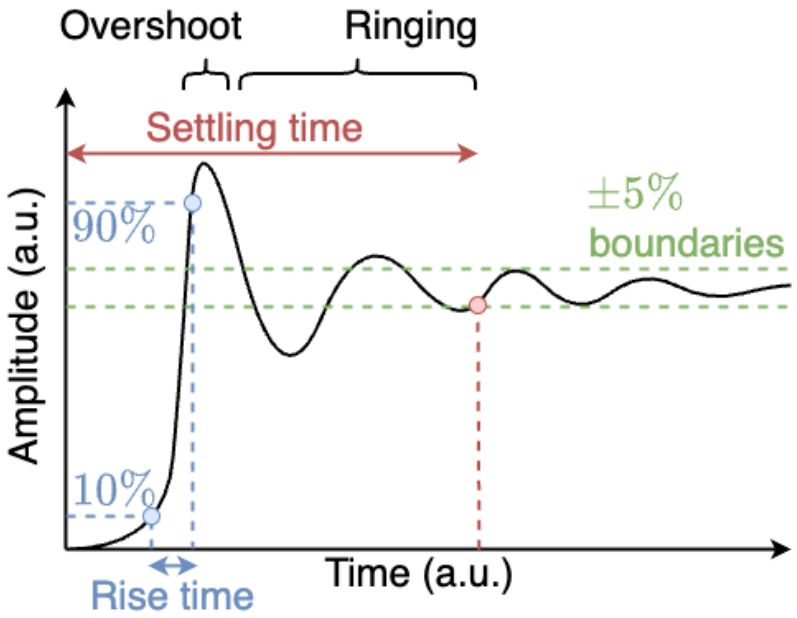


FIGURE 3.5: Visualisation of a typical SOA response when amplifying an optical signal. The SOA's optical output will overshoot the target settling point, and then ring for some period of time before settling within ±5% of the steady state.

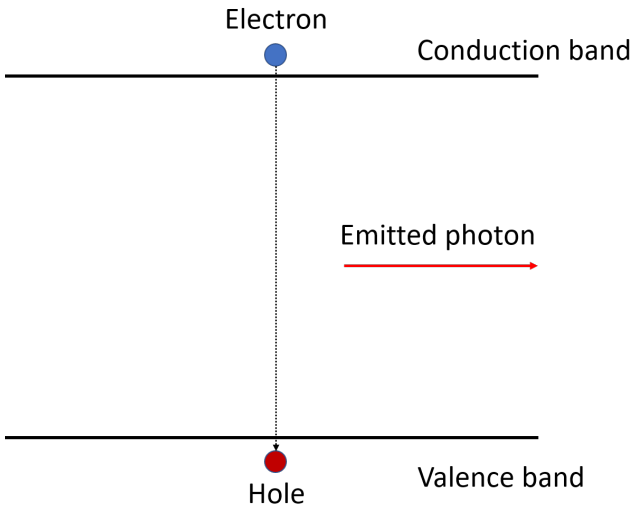


FIGURE 3.6: Schematic of the process of spontaneous emission. An electron in an excited state spontaneously recombines with a hole, emitting a photon equal to the energy across which the electron relaxed.

3.2.2 Evolutionary & Swarm Algorithms

Swarm and evolutionary algorithms are similar in that the methods in both categories are inspired by natural biological phenomena, and both are composed of a *population* of agents searching for near-optimal solutions. In fact, prior

to the mid-1990s, swarm algorithms were categorised as evolutionary methods [Bansal, 2019]. Their modern distinction is due to their differing philosophies; swarm algorithms utilise the emergent collective behaviour of their population, retaining the individual identity of the agents over time, whereas evolutionary algorithms rely on the concepts of natural selection and genetics, replacing individual identities with new generations of agents over time.

Swarm algorithms. Swarm intelligence algorithms are composed of simple agents which cooperate in a self-organised and decentralised manner such that a so-called 'intelligent' collective search strategy emerges [Bonabeau et al., 1999, Bansal, 2019]. Each agent constitutes a possible solution to the problem being solved. At every iteration, each solution is stochastically updated according to the 'fitness' (efficacy) of both the individual solution and the collective population of solutions following various update rules and communication strategies. Unlike many subsequent nature-inspired swarm methods, the first swarm algorithm, stochastic diffusion search [Bishop, 1989], was built entirely from a comprehensive mathematical framework. Later swarm algorithms such as ACO [Dorigo, 1992] and PSO [Kennedy et al., 1995], which today are among the most widely used [Bansal, 2019], were inspired by the natural social phenomena exhibited by ant colonies, flocks of birds, and schools of fish.

Evolutionary algorithms. Evolutionary computation algorithms are also composed of simple agents but, rather than cooperating, these agents typically compete and get replaced by new generations of agents with advantageous characteristic updates following the 'survival of the fittest' principle [De Jong et al., 1997]. Evolutionary algorithms have been around for several decades, with the first use of evolutionary programming thought to be in by Fraser [1958]. They generally work by randomly initialising a population of possible solutions, iteratively modifying them to generate a new set of solutions via a series of selection, crossover and mutation operations, and stochastically discarding poor solutions while evolving fit solutions into the next generation. The evolved

solutions should therefore improve generation-by-generation until a near-optimal solution is converged on.

Swarm & evolutionary algorithms for optimisation. There are several drawbacks evolutionary and swarm algorithms. Their efficacy often heavily depends on the hyperparameters used to control their behaviour and exploration-exploitation trade-offs, hence computationally costly tuning is often required. Furthermore, their fundamental reliance on stochastic behaviour creates problems with reproducibility and guaranteed global optimum convergence. As such, quadratic optimisation problems with linear constraints or linear programming problems of a reasonable size are best solved by classical methods with rigorous mathematical frameworks and good convergence properties. However, many real-world problems are unstructured, have complex, non-differentiable and non-convex objective functions and, at scale, have too many variables to be solved by classical techniques. Such problems are where the use of evolutionary and swarm intelligence algorithms can become advantageous.

3.2.3 Genetic Algorithms

GAs are a family of evolutionary algorithms. They mimic the mechanism of *Darwinian evolution*. In nature, each physical property (*phenotype*) of a living organism, such as eye and hair colour, is determined by a set of rules or instructions called *genes*. The genes are strung together into structures called *chromosomes*. As such, it is the genes within the chromosomes which determine the decoded phenotypic traits of the organism. These core concepts and terminologies are used by GAs. GAs are local search metaheuristics which, in the discrete case, can find near-optimal solutions to NP-hard CO problems.

Problem formulation. Given a problem Π described by triple (S, f, Ω), a chromosome is a candidate solution s to Π. The string of genes making up the chromosome are the solution's variables. Each gene is usually encoded by

a binary (0s and 1s) string representation (for example, a variable with 256 possible values can be encoded by an 8-bit string), although for problems with continuous variables, real-valued genes can also be used. The string of genes (variables) making up the chromosome (solution) have a real phenotypic value (solution output). The fitness of the resultant phenotype is determined by a fitness (objective) function f (for example, the difference between the actual and the target solution outputs, such as network latency vs. 0 latency). The task is therefore to find the optimum set of gene values s^* which result in the most favourable performance as determined by the fitness function $f(s)$. The fitness function used is always problem-dependent. In nature, this function corresponds to the organism's ability to operate and survive in a given environment, and the probability of selecting a given gene to pass on to future chromosomes during reproduction is proportional to the chromosome's fitness.

Optimisation process. As described by Kiranyaz [2014], the general process for GA optimisation via Darwinian evolution is as follows. (1) *Initialisation*: Generate a population of chromosomes whose gene strings are randomly initialised to yield a comprehensive range of possible solutions across the *search space* (set of possible solutions) S. (2) *Selection*: For each successive generation, first stochastically select chromosomes to use to breed into the next generation, granting higher selection probabilities to chromosomes with greater fitness values. (3) *Reproduction*: Second, use genetic operators (such as 'crossover' and 'mutation') to breed children of the chosen chromosomes whose genes differ but are similar to the original parent chromosomes. (4) *Evaluation*: Evaluate the fitness of the child chromosomes and substitute them for any chromosomes in the previous generation's population with poorer fitness values so as to keep the population size constant generation-to-generation. (5) *Termination*: If a target fitness is achieved, if the maximum number of generations has been reached, or if incremental fitness improvements have converged across multiple generations, terminate the GA process. Otherwise, repeat steps 2-4.

Algorithm 1 formulates this GA procedure for a population C of n chromosomes where each chromosome c occupies a phenotype state s in a search space (set of possible solutions) S with fitness function f.

Algorithm 1 Genetic Algorithm

Require: $n > 0, \forall s \in S$
  Repeat (for each generation):
    Select a set of parent chromosomes $C_p \in C$
    Apply genetic operator(s) to reproduce a set of child chromosomes C_c
    for c_c in C_c **do**
      $c_c^{fitness} = f(c_c)$
      if $c_c^{fitness} > min(C^{fitness})$ **then**
        $C[min(C^{fitness})] \leftarrow c_c$
      end if
    end for
  until termination

Crossover & mutation. Given a pair of chromosomes from which to breed, the key genetic operators in the above GA process are *crossover* and *mutation*. Crossover is applied with probability P_X, and is where the genetic information of each chromosome is swapped about some string position L to create two new chromosomes. For example, given two 16-bit chromosomes each made up of two 8-bit genes, [01001001 10001010] and [10100101 00001111], and given that L has been sampled at $L = 10$, the crossover operator would produce the child chromosomes [01001001 10001111] and [10100101 00001010]. Crossover allows for accelerated search early on in the GA process and enables exploration of sub-solution combinations for different chromosomes. However, as always with AI techniques, there is a trade-off. Set P_X too high and risk unstable fitness values oscillating about the global optimum; set P_X too low and suffer long convergence times and stagnation at local optima. Similarly, mutation is applied to the child chromosomes with probability P_M, and is where a single randomly chosen bit in the chromosome is flipped ($0 \Rightarrow 1$ or $1 \Rightarrow 0$). Mutation increases the ergodicity of GAs, compromising short-term fitness in exchange for greater exploration and therefore improved long-term performance. As with P_X, P_M

must also be carefully tuned. Too low and non-ergodic genetic drift without sufficient exploration will occur, resulting in convergence on local optima; too high and the volatile stochastic nature of the mutations will prevent convergence on the global optimum.

There are many different flavours of GA which generally differ according to the exact operator(s) and problem formulation used, but all fundamentally work as described above. For a comprehensive overview of GAs, refer to Kiranyaz [2014].

3.2.4 Ant Colony Optimisation

ACO is a swarm intelligence metaheuristic. It mimics the process by which ants forage for food. In nature, ants coordinate their activities via *stigmergy*; a form of indirect communication via modification of the surrounding environment by the deposition of *pheromone* chemicals. The *trail pheromone* is a specific type of pheromone used to mark paths on the ground, such as from a nest to a food source. When walking along a path, an ant deposits trail pheromones. Other ants can smell these pheromones and tend to choose, stochastically, paths with strong pheromone concentrations. When foraging begins, there are initially no pheromone trails, hence the path chosen by each ant is uniformly random. However, because pheromones have an *evaporation rate*, the first ant to find food and return to the nest will leave the strongest pheromone trail, thereby marking the shortest path. This in turn will bias the stochastic path choice of other ants, increasing the pheromone concentration on the shortest path via an autocatalytic process until most ants converge on the optimum path, with a few random fluctuations in path choice retaining some level of exploration. Hence, ants have a built-in optimisation capability; by using stochastic rules based on local information, they can collectively find the shortest path between two points.

These core concepts form the basis of ACO. They can be abstracted beyond shortest path identification to a variety of discrete optimisation problems via a mapping. ACO is a constructive metaheuristic which can be used to find near-optimal solutions to NP-hard CO problems.

Problem formulation. Consider a CO minimisation problem Π described by the triple (S, f, Ω). The search space constitutes a finite set of discrete components (variable values) $C = \{c_1, c_2, ..., c_m\}$, where m is the number of components in the search space. Each solution s is a finite sequence $s = \langle c_i, c_j, ..., c_h, ...\rangle$ over the elements in C. To solve (S, f, Ω), artificial ants iteratively build a solution by constructing a path on the *construction graph* $G = (C, L)$ whose edges L fully connect the nodes C. Components $c_i \in C$ and connections $l_{i,j} \in L$ have an associated *pheromone trail* τ, which is the long-term memory of the ant search process and which is updated by the ants themselves. They also have a *heuristic value* η, which captures *a priori* or run-time information about the problem and which is from a source other than the ants, such as the estimated cost of adding a new component or connection to the partial solution under construction.

Optimisation process. Each ant k uses $G = (C, L)$ to search for the optimal solution $s^* \in S$ for (S, f, Ω). It has a memory M^k in which to store information about the path followed (i.e. the partial solution built) so far. When initialised at its start node, the ant selects a node j amongst its neighbourhood of nodes N^k to move to. This selection is made by applying a stochastic decision rule. The decision rule is a function of the available local τ and η values associated with each possible action, the ant's private memory M^k storing its current partial solution, and the problem constraints Ω which prevent invalid selections. Once the ant has iteratively added its selected component actions c_j to M^k until ≥ 1 *termination conditions* e^k are met, the ant re-traces its path in reverse and updates the pheromone trail values τ at each component.

The above ACO process can be deconstructed as follows [Dorigo and Stützle, 2004]. (1) *Construct solutions*: Move each ant through G by applying a stochastic

local decision process to incrementally build a solution to the optimisation problem being solved until a termination condition is reached. (2) *Update pheromones*: Update the pheromone values τ in G based on which components were in each ant's solution and on the evaporation rate (which forms a useful 'forgetting' mechanism by which rapid convergence on local optima can be avoided). (3) *Daemon actions*: Implement custom centralised actions which cannot be performed by single ants (e.g. collect global information to decide which particular paths found by the ants were especially good and therefore warrant additional pheromone deposits on their respective components).

Algorithm 2 formulates this ACO process for a colony K of n ants. Dorigo and Stützle [2004] provide a more comprehensive overview of ACO.

Algorithm 2 Ant Colony Optimisation

Require: $n > 0, \forall s \in S$
  Repeat (for each iteration):
    for k in K **do**
      Repeat (for each step):
        Stochastically choose an action j from neighbourhood N^k
        $M^k \leftarrow M^k, j$
      until termination
    end for
    Update τ values in G using ants' solutions and any daemon actions
  until convergence on near-optimal solution

3.2.5 Particle Swarm Optimisation

PSO is a swarm intelligence algorithm. It mimics the way in which bird flocks and fish schools exhibit self-organised decentralised collective adaption to their surrounding environment. PSO is a local search metaheuristic which can be used to find near-optimal solutions to NP-hard CO problems.

Problem formulation. To solve a problem Π described by triple (S, f, Ω), a swarm of n particles are initialised at random positions. Each particle position represents a potential solution $s \in S$ to Π and has m components (dimensions). At iteration i, the position (solution) represented by particle j is denoted $s_j(i)$.

The goal is for the swarm of particles to collectively navigate through the search space S to find the optimal solution s^* according to the fitness function f.

Optimisation process. At each sequential iteration in the PSO process, the particle positions are updated by adding a velocity term $v_j(i)$, as in Eq. 3.1. This velocity term is what drives the PSO process. It contains the personal knowledge of the particle (the 'cognitive component', which is proportional to the distance between $s_j(i)$ and the particle's historic 'personal best' position p_{best_j}) and the socially exchanged information of the particle's neighbours (the 'social component', which is proportional to the distance between $s_j(i)$ and the whole swarm's historic 'global best' position g_{best}). At each iteration i, the velocity of particle j at the t^{th} dimension in position $s_j(i)$ is updated according to Eq. 3.2, where $C = \{c_1, c_2, ..., c_m\}$, conf_{1j} and conf_{2j} are the personal and social 'confidence acceleration constants' of particle j used to scale the contributions of the cognitive and social components respectively, r_{1j} and r_{2j} are random values in range $[0, 1]$ sampled from a uniform distribution in order to introduce stochastic exploration, and w_j is the 'momentum' of particle j used to control the exploration-exploitation inclinations of the particle.

$$s_j(i+1) = s_j(i) + v_j(i+1) \tag{3.1}$$

$$v_{jc}(i{+}1) = w_j{\cdot}v_{jc}(i){+}\text{conf}_{1j}{\cdot}r_{1j}(t){\cdot}\Big[p_{best_{jc}}(i) - s_{jc}(i)\Big]{+}\text{conf}_{2j}{\cdot}r_{2j}(i){\cdot}[g_{best_c} - s_{jc}] \tag{3.2}$$

A higher conf_1 encourages the particle to be more confident in itself and explore more positions, but possibly take longer to converge on the optimum g_{best}. On the other hand, a higher conf_2 encourages the particle to trust the social knowledge of its neighbours and converge faster on g_{best}, but be less inclined to explore new positions. As such, conf_1 and conf_2 are critical in controlling the

exploration-exploitation trade-off of PSO. It has been shown that so long as these variables satisfy Eq. 3.3, PSO is guaranteed to converge on some solution rather than unstably oscillate around objective function minima [Van Den Bergh and Engelbrecht, 2001].

$$0 \leq \frac{1}{2}\left(\text{conf}_1 + \text{conf}_2\right) - 1 < w < 1 \tag{3.3}$$

In a common variant of PSO, known as *dynamic* PSO [Clerc, 1999], w, conf_1 and conf_2 can be dynamically updated at the start of each iteration according to Eqs. 3.4, 3.5 and 3.6 respectively, thereby avoiding being either too exploitative or too exploratory at local iterations in the PSO process. $w(0)$ is an 'initial inertia weight' constant ($0 \leq w(0) < 1$), $w(n_i)$ is the 'final inertia weight' constant ($w(0) > w(n_i)$), $h_j(i)$ is the relative fitness improvement of particle j at iteration i, and conf_{max} and conf_{min} are the maximum and minimum values for the acceleration constants respectively.

$$w_j(i+1) = w(0) + \left[\left(w(n_i) - w(0)\right) \cdot \left(\frac{e^{h_j(i)} - 1}{e^{h_j(i)} + 1}\right)\right] \tag{3.4}$$

$$h_j(i) = \frac{p_{best_j}(i) - s_j(i)}{p_{best_j}(i) + s_j(i)} \tag{3.5}$$

$$\text{conf}_{1,2}(i) = \frac{\text{conf}_{min} + \text{conf}_{max}}{2} + \frac{\text{conf}_{max} - \text{conf}_{min}}{2} + \frac{e^{-h_j(i)} - 1}{e^{-h_j(i)} + 1} \tag{3.6}$$

f is used to evaluate a given particle position s_j, and in the case of a minimisation problem instance, can be used to update p_{best_j} and g_{best} at each iteration according to Eqs. 3.7 and 3.8 respectively.

$$p_{best_j}(i+1) = \begin{cases} p_{best_j}(i), & \text{if } f(s_j(i+1)) \geq f(p_{best_j}(i)) \\ s_j(i+1), & \text{otherwise} \end{cases} \tag{3.7}$$

$$g_{best}(i+1) = \begin{cases} g_{best}(i), & \text{if } f(s_j(i+1)) \geq f(g_{best}(i)) \\ s_j(i+1), & \text{otherwise} \end{cases} \tag{3.8}$$

Kiranyaz [2014] gives a more in-depth overview of the PSO metaheuristic.

3.3 Related Work

Various alternatives to OCS solutions have been proposed as a means to enable all-optical DCNs. These include optical loop memory [Srivastava et al., 2009], optical burst switching (OBS) [Chen, 2005], and OPS [Benjamin et al., 2017, Wang et al., 2018]. However, such techniques require more complex, expensive and unscalable architectures than provided by OCS [Benjamin, 2020]. The drawback of OCS is the aforementioned lack of a viable ultra-fast all-optical switch.

PISIC techniques. There have been a range of previous attempts to bring SOA switching speeds closer to their theoretical 100 ps optimum. A previous study looking to optimise SOA output applied a PISIC driving signal to the SOA [Gallep and Conforti, 2002]. This PISIC signal pre-excited carriers in the SOA's gain region, increasing the charge carrier density and the initial rate of stimulated emission to reduce the 10% to 90% rise time from 2 ns to 500 ps. However, this technique only considered rise time when evaluating SOA off-on switching times. A more accurate off-on time is given by the settling time, which is the time taken for the signal to settle within ±5% of the 'on' steady state. Before settling, bits experience a variable signal to noise ratio, which impacts the bit error rate (BER) and makes the signal unusable until settled, therefore the switch is effectively 'off' during this period.

MISIC techniques. Later attempts looked at applying a MISIC driving signal to remedy the SOA oscillatory and overshoot behaviour [Conforti and

Gallep, 2006, Ribeiro et al., 2009, Figueiredo et al., 2015]. As well as a pre-impulse, the MISIC signal included a series of subsequent impulses to balance the oscillations, reducing the rise time to 115 ps and the overshoot by 50% [Figueiredo et al., 2015]. However, the method for generating an appropriate pulse format was trial-and-error. Since each SOA has slightly different properties and parasitic elements, the same MISIC format cannot be applied to different SOAs, therefore a different format must be generated through this inefficient manual process for each SOA, of which there will be thousands in a real DCN. As such, MISIC is not scalable. Furthermore, the MISIC technique did not consider the settling time, therefore the effective off-on switching time was still several ns.

Surrounding component optimisation. More recent works used the MISIC technique, but focused on closer integration between SOA microwave elements to decrease rise time, instability, and non-linear behaviour [Figueiredo et al., 2017]. Taglietti et al. [2018] adopted this principle but also applied a Wiener filter. The filter was determined by the steady state value of the SOA response. The mean squared error (MSE) between the output and the filter is minimised by finding the optimal set of weight coefficients for the filter. The work accomplished a 60% reduction in guard time, with the goal of reducing guard time as much as possible such that the BER of the output did not exceed a particular level. However, the study did not consider the settling time, which is crucial for optimising the practical switching speed. Similarly, Sutili et al. [2019] explored the optimisation of an SOA by means of both modification of the driving signal and optimisation of the SOA's microwave mounting. A best case of 33% reduction in guard time was accomplished with an improved microwave mounting architecture and a step driving signal, where various MISIC and PISIC driving signals were also tested. This work demonstrated that significant improvements in guard time could be derived exclusively from improvements made to the microwave mounting of the SOA and that the improvement of

the SOA's output by optimisation of the driving signal did not preclude the simultaneous improvement by optimisation of the microwave mounting. It is therefore complementary to the results we present in this chapter, since we do not consider microwave component optimisation, but rather only focus on optimising the SOA drive signal.

The previous solutions discussed so far have had a design flow of first manually coming up with a heuristic for a simplified model of an SOA, followed by meticulous testing and tuning of the heuristic until good real world performance is achieved. If some aspect of the problem is changed such as the SOA type used or the desired shape of the output signal, this process must be repeated. In this chapter we present AI as a fully automated optimisation technique for any SOA, and experimentally demonstrate an order of magnitude improvement in switch speed over the previous world record. A comparison of our work and that of the literature is presented in Table 3.1.

3.4 Method

Problem formulation. Finding a near-optimal driving signal for an SOA can be formulated using the CO framework presented in Section 2.5. Π can be described by triple (S, f, Ω). Here, the set of candidate solutions S to Π is the set of possible electrical signals (which are voltage vs. time functions) available to drive the SOA with, where the constraints Ω are defined by the equipment used (which determines signal resolution, frequency response, and so on). Each solution $s \in S$ is therefore made up of a series of m components $C = \{c_1, c_2, ..., c_m\}$ where each component c_t corresponds to a point in the driving signal at time t. Each component can take one of 2^u possible voltage values, where u is the bit resolution of the driving signal generator. f is the the objective function which takes the optical signal output by the SOA (an intensity vs. time function with p points, here termed the 'process variable')

TABLE 3.1: Comparison of SOA Optimisation Techniques. (Best in bold).

- [a] Though exact value not reported in Sutili et al. [2019], it is referred to as being 'below 500 ps'.
- [b] Comparison of the ASM mounting against the commercial STF mounting.
- [c] Exact value not reported in Sutili et al. [2019] so percentage improvement is (approximately) inferred from a graph presented in Sutili et al. [2019]. Comparison made at bias current value corresponding to the best case performance of the best performing ASM mount + drive combination and is compared against the STF mount + drive at the same bias and for the same drive (step was best performing in the reported metrics).
- [d] Comparison is made between the best and worst cases presented in Taglietti et al. [2018].
- [e] Several variants of the 'MISIC' format were tested in Figueiredo et al. [2015] and the best is used here for comparison.
- [f] Comparison made with respect to the performance of the STEP driving signal presented in Figueiredo et al. [2015].

Method (Technique)	Reference	Rise Time, ps (Reduction, %)	Settling Time, ps (Reduction, %)	Overshoot, % (Reduction, %)	Guard Time, ps (Reduction, %)
PSO (Signal Optimisation)	This work	454 ps (35%)	**547 ps (85%)**	5% -	- -
ACO (Signal Optimisation)	This work	413 ps (41%)	560 ps (85%)	4.8% -	- -
GA (Signal Optimisation)	This work	**340 ps (51%)**	825 ps (78%)	10.3% -	- -
PISIC (Signal Optimisation)	This work	502 ps (28%)	4350 ps (-17%)	40.5% -	- -
MISIC1 (Signal Optimisation)	This work	502 ps (28%)	4020 ps (-8%)	undershot -	- -
Raised Cosine (Signal Optimisation)	This work	921 ps (-32%)	4690 ps (-26%)	undershot -	- -
PID Control (Signal Optimisation)	This work	501 ps (28%)	4020 ps (-8%)	2.3% -	- -
ASM Mounting + STEP Drive (Microwave Mounting Optimisation)	Sutili et al. [2019]	- -	- -	**≈ 5%** [c] **(≈ 75%** [b,c]**)**	≈ 500 ps [a] (≈ 33% [b,c])
STEP Drive + Wiener Filtering (Signal Optimisation + Filtering)	Taglietti et al. [2018]	- -	- -	- -	**286 ps (60%** [d]**)**
PISIC Drive (Signal Optimisation)	Figueiredo et al. [2015]	115 ps (34% [f])	- -	25% (-56% [f])	- -
MISIC-6 Drive [e] **(Signal Optimisation)**	Figueiredo et al. [2015]	115 ps (34% [f])	- -	12.5% (22% [f])	- -

when driven with the electrical signal $s \in S$ (the 'input control signal' with m points, here termed OP) and evaluates the driving signal's performance by assigning it a scalar objective function value. We define f (Eq. 3.9) as the MSE between the PV and a target 'set point' (SP), where SP is an ideal optical output with 0 rise time, settling time and overshoot. Thus, this CO problem is a minimisation problem where the goal is to find the optimal driving signal $s^* \in \tilde{S}$ which minimises the objective function value $f(s)$ such that $f(s^*) \leq f(s) \forall s \in \tilde{S}$.

$$f(PV) = \frac{1}{p} \sum_{i=1}^{p} (PV_i - SP_i)^2 \tag{3.9}$$

Using the above CO formulation, GA, ACO and PSO can all be applied to the CO problem of finding a near-optimal driving signal for a given SOA, as will now be described. The hyperparameter tuning process and settings used for each AI technique are given in Section 3.7.1.

Genetic algorithm. A chromosome represented a driving signal and a gene represented the voltage value for a given point in the driving signal. For a u-bit driving signal, each gene was encoded with a u-bit binary string. The phenotype resulting from the string of genes making up the chromosome was the corresponding SOA optical output whose fitness was evaluated by the objective function in Eq. 3.9. At each generation, $n_{\text{tournsize}}$ individuals were stochastically selected to breed into the next generation from a population of n chromosomes using the mutation and crossover genetic operators. Mutation was implemented using Gaussian mutation [Hinterding, 1995] where normally distributed noise of mean μ and standard deviation σ was applied to change a gene's value with mutation probability P_M. Crossover between two individuals in $n_{\text{tournsize}}$ was applied with probability P_X. Through the process of evolution, the population of n chromosomes would eventually converge on a near-optimal driving signal s^* for the SOA.

Ant colony optimisation. A graph G with m clusters (one cluster for

each component) where each cluster had 2^u nodes (one node for each possible component value) was initialised. A population of n ants started at the first cluster (point in the signal), chose a node (signal voltage value) in the cluster, and then moved to the next cluster. In this way, each ant could travel through G and iteratively build a driving signal solution s by choosing nodes in G. Defining the strength of the pheromone trail using the objective function in Eq. 3.9, an associated evaporation rate, and a probability of random path selection, the ants could iteratively converge on a near-optimal driving signal s^* for the SOA.

Particle swarm optimisation. A visualisation of how PSO was applied to SOA optimisation is given in Fig. 3.7. A population of n particles were initialised at random positions, where each position was an m-component driving signal. Since experimental results showed spurious overshoots after the rising edge and therefore an increase in the settling time, the PSO search space was bounded by a PISIC-shaped 'shell' beyond which the particle dimensions could not assume values. An added benefit of the shell was a reduction in the complexity of the problem and therefore also the convergence time. The shell area was a PISIC signal with a leading edge whose width was defined as some fraction of the 'on' period of the signal. The particles could then be flown through the search space, updating p_{best} for each particle and g_{best} for the population at each i^{th} iteration according to Eq. 3.9 until the particles converged on a near-optimal driving signal s^* for the SOA.

3.5 Simulation Setup

To enable rapid hyperparameter tuning, rather than relying on laboratory experiments, it was useful to first simulate an SOA and use this simulation environment to tune the AI hyperparameters and to test novel ideas such as the PISIC shell.

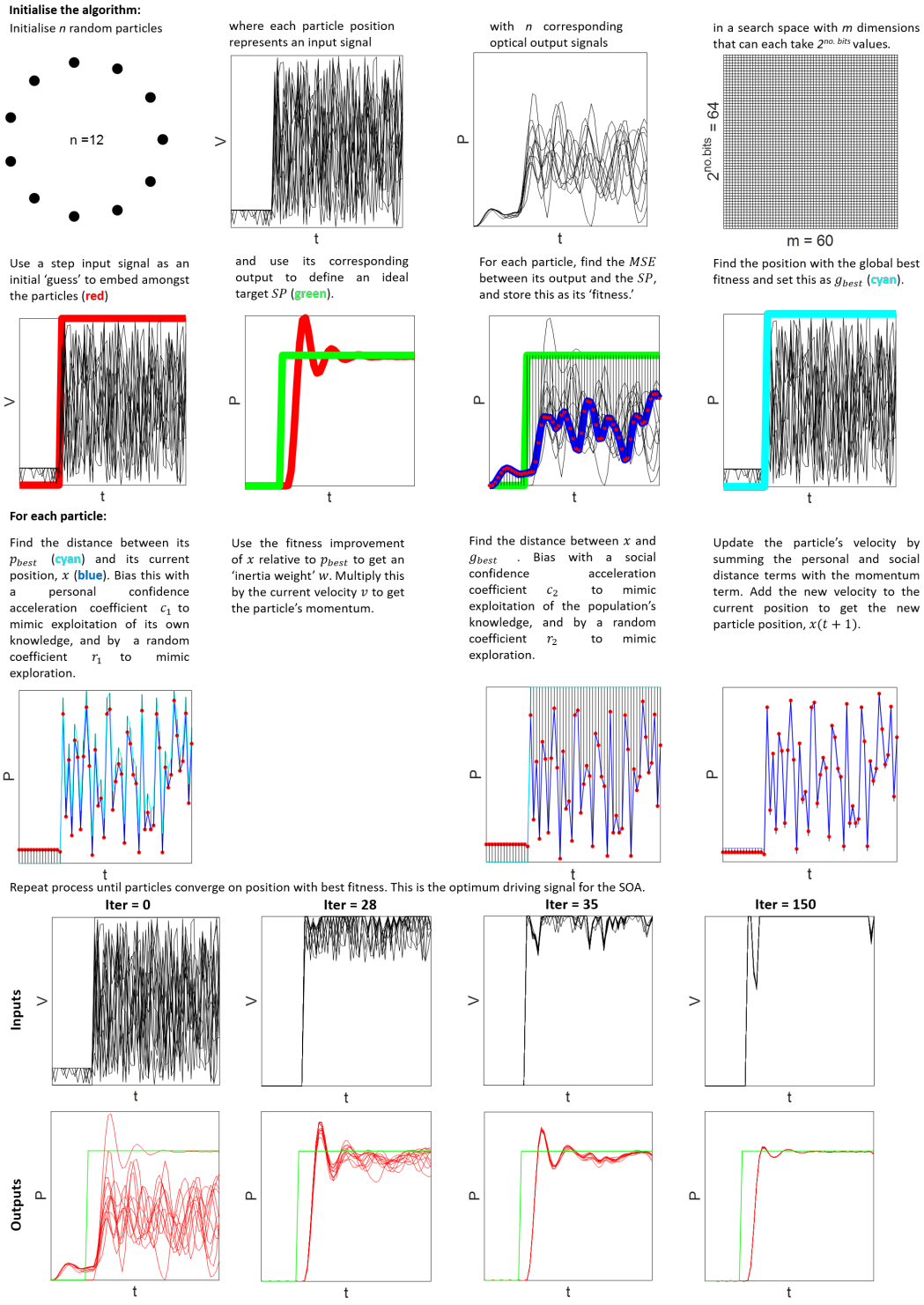


FIGURE 3.7: Visualisation of how PSO was applied to SOA optimisation.

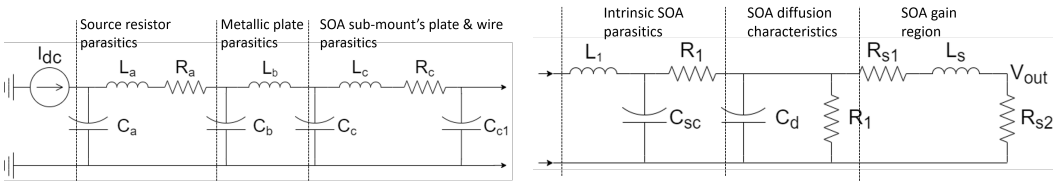
The shortcomings of rate equations. SOAs are typically modelled using simple rate equations. However, as shown by Ghafouri-Shiraz [2004], the electrical parasitics of an SOA and its surrounding packaging degrade optical signals by broadening the output optical pulse width, reducing the peak optical

power (thereby reducing optical contrast), and causing a slight time delay in the emitted optical pulse. Additionally, they alter the relaxation frequency of the SOA output oscillations. As such, modelling the electrical parasitics was crucial to building a simulation environment in which to optimise switching. As described by Ghafouri-Shiraz [2004], Figueiredo et al. [2011], and Tucker et al. [1984], assuming a small circuit model, microwave equivalent circuits can be used to more accurately simulate semiconductor diodes by accounting for these electrical parasitics. Therefore, equivalent circuits were the chosen approach to SOA modelling for this chapter.

Equivalent circuit modelling. The electrical parasitics were split into two categories; the parasitics from the microwave injection current components (the 50Ω resistor of the source, the metallic plate, and the SOA sub-mount plate and wire parasitics) and the SOA's intrinsic parasitics [Figueiredo et al., 2015]. Fig. 3.8a shows the microwave injection current parasitics modelled as an equivalent circuit, and Fig. 3.8b shows the SOA's intrinsic parasitics, diffusion characteristics, and gain region. These two equivalent circuits were connected to form a single SOA model that accounted for all the electrical parasitics. Since at low voltages ($< 0.8V$) the current (I) - voltage (V) relationship can be described by Equation 3.10, the ideality factor η and the saturation current I_s could be calculated using the semi-logarithmic I-V curve of the SOA in Fig. 3.9. Defining the threshold current (I_{TR}) as the bias current needed for stimulated emission to become more dominant than spontaneous emission, the SOA small signal model was split into two parts; below I_{TR} (2-50 mA) and above I_{TR} (75-110 mA). The equivalent circuits used to model the gain region of these two parts are shown in Fig. 3.10.

$$ln\Big(I\Big) = ln\Big(I_s\Big) + \left(\frac{1}{\eta}\right)\left(\frac{qV}{K_bT}\right) \tag{3.10}$$

Simulating below and above the threshold current. For the below



(A) (B)

FIGURE 3.8: Equivalent circuit diagrams of an SOA's (a) microwave injection current parasitics and (b) intrinsic parasitics, diffusion characteristics and gain region.

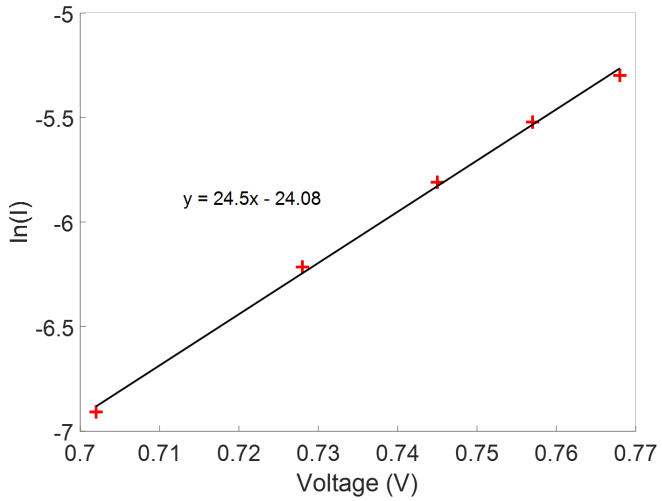
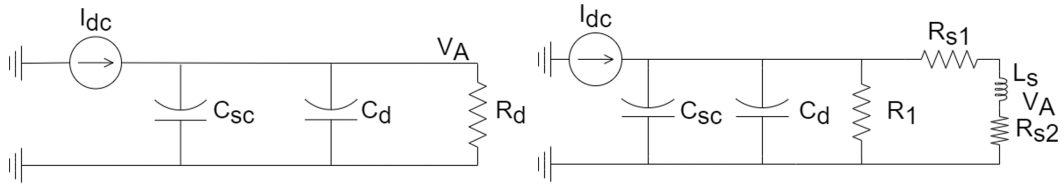


FIGURE 3.9: Semi-logarithmic I-V plot for the SOA used to calculate η and I_s.



(A) (B)

FIGURE 3.10: Equivalent circuit diagram of the SOA gain region (a) below I_{TR} and (b) above I_{TR}.

I_{TR} simulation, Equations 3.11 - 3.13 were used to calculate the space-charge capacitance C_{sc}, the diffusion capacitance C_d, and the Schockley diode resistance R_d (where V_{dc} = applied voltage). Taking these values and extrapolating them to higher bias currents, the SOA was modelled for above I_{TR}. Equations 3.14 - 3.16 were used to calculate the space-charge capacitance (C_{sc}), the diffusion capacitance (C_d), the resistances R_1, R_{s1} and R_{s2} which together modelled the SOA optical output oscillation dampening due to spontaneous and stimulated

emission, and the inductance (L_s). The SOA oscillations that arise from the dynamic exchange of energy between photons and SOA active region carriers were modelled by the charge-discharge effect between the capacitance $C_{sc} + C_d$ and the inductance L_s. The internal and external constants for these equations were taken from the literature values for a typical silicon laser diode [Ghafouri-Shiraz, 2004, Figueiredo et al., 2011, Tucker et al., 1984]. All constant values used are summarised in Tables 3.2 and 3.3.

$$R_d = \frac{\eta K_b T}{q I_s} \frac{1}{e^{\frac{qV_{dc}}{\eta K_b T}}} \tag{3.11}$$

$$C_d = \frac{\tau_n}{R_d} \tag{3.12}$$

$$C_{sc} = C_{sc0}\left(1 - \frac{V_{dc}}{V_D}\right)^{-\frac{1}{2}} \tag{3.13}$$

$$R_1 = \frac{R_d}{1 + \gamma \tau_n S_0} \tag{3.14}$$

$$R_{s1} = \frac{\epsilon R_d}{\gamma \tau_n} \tag{3.15}$$

$$R_{s2} = \frac{\beta \Gamma R_d \tau_p I_{tA}}{\alpha \gamma \tau_n S_0^2} \tag{3.16}$$

$$L_s = \frac{R_d \tau_p}{\gamma \tau_n S_0} \tag{3.17}$$

Obtaining an input-output model. It was found that the SOA in the experimental setup had the optimum trade-off between gain and signal noise at a bias current of 75 mA, therefore the simulated SOA was biased at this current. Using MATLAB's Simulink tool, a transfer function (TF) for the SOA equivalent circuit was obtained and simplified as shown in Eq. 3.18 with the constants

TABLE 3.2: Internal parameters used to model the SOA as an equivalent circuit.

Name	Symbol	Value	Units
Ideality factor	η	1.59	–
Saturation current	I_s	3.48e−11	A
Threshold current	I_{TR}	70	mA
Boltzmann constant	K_b	1.381e−23	JK^{-1}
Temperature	T	298.15	K
Electron charge	q	1.602e−19	C
Charge carrier lifetime	τ_n	3	ns
Zero-bias space-charge capacitance	C_{sc0}	1	pF
Built-in potential	V_D	1.3	V
Gain compression factor	ϵ	4.5e−12	m^3
Boltzmann constant	K_b	1.381e−12	JK^{-1}
Gain coefficient × group velocity	γ	2.4e−12	m^3s^{-1}
Spontaneous emission factor	β	e−4	–
Optical confinement factor	Γ	0.4	–
Photon lifetime	τ_p	1	ps
Active region volume	V	4e−16	m^3
Charge × active region volume	α	6.41e−25	Cm^3
Active region carrier density	N_{TR}	e24	m^{-3}
SOA leakage current	I_F	15	mA

TABLE 3.3: External parameters used to model the SOA's chip and packaging parasitics as an equivalent circuit.

Origin	Symbol	Value	Units
Source	C_a	0.25	pF
Source	L_a	0.34	nH
Source	R_a	50	Ω
Metallic plate	C_b	81	pF
Metallic plate	L_b	1.38	nH
Sub-mount	C_c	1.2	pF
Sub-mount	L_c	2.5	nH
Sub-mount	R_c	0.9	Ω
Sub-mount	C_{c1}	30	pF

TABLE 3.4: Constants used in the equivalent circuit transfer function.

a_9	1.65	a_4	1.37×10^{52}
a_8	4.56×10^{10}	a_3	2.82×10^{62}
a_7	3.05×10^{21}	a_2	9.20×10^{71}
a_6	4.76×10^{31}	a_1	1.69×10^{81}
a_5	1.70×10^{42}	a_0	2.40×10^{90}

defined in Table 3.4. This allowed for custom drive signals to be generated, sent to the biased SOA equivalent circuit, and an optical output measured. We note that the exponents in the TF of Table 3.4 are unusually high. To verify that the TF accurately modelled the SOA, Section 3.7.1 shows that the theoretical and experimental optical response frequency of the modelled and the real SOA closely matched oneanother. One possible reason for the high exponents could be overfitting to the complex relationships of the simulation. Another explanation could be that the setup of the MATLAB simulation was not conducive to easy interpretation by a TF model. However, given that the obtained TF accurately modelled the optical response of the experimental device, the TF model's fidelity was deemed sufficient for the purposes of this chapter. A full investigation of the high exponents and the TF model is left for further work.

$$TF = \frac{2.01 \times 10^{85}}{\sum_{i=0}^{9} a_i s^i} \tag{3.18}$$

Analytically optimising SOAs is difficult. As illustrated by the simulation methodology outlined above, the difficulty with SOA modelling, and subsequently also SOA switching, is that there are many variables whose values are difficult to experimentally measure, and which vary significantly even for same-specification SOAs due to parasitics introduced during manufacturing and packaging. Re-measuring these constants for a new SOA would be cumbersome, difficult, and unfruitful since broad assumptions would still need to be made. Furthermore, scaling this bespoke-modelling to 1,000s of SOAs in a single DCN

would be unrealistic. As such, analytical solutions to SOA switching are not beneficial. Additionally, different driving circuit setups with different amplifiers, bias tees, cabling, and so on influence the shape of the driving signal that arrives at the SOA, thereby requiring more manual tuning every time the equipment surrounding the SOA is changed. This highlights the need for the partially 'model-free' AI approaches proposed in this chapter, which neither make or require any assumptions about the SOA or the surrounding driving circuit they are optimising, resulting in their optimised driving signals being superior both in terms of performance and scalability relative to traditional analytical and/or manual methods. Here, we borrow the term 'model-free' from the field of reinforcement learning, meaning an algorithm that does not initially know anything about the environment in which it must perform its optimisation [Sutton and Barto, 2018]. In Section 3.7.1, we justify the claim that tuning AI algorithms in a single simulation environment enables the same AI hyperparameters to be transferred to unseen SOAs.

3.6 Laboratory Setup

The experimental setup is shown in Fig. 3.11. An INPhenix-IPSAD1513C-5113 SOA with a 3 dB bandwidth of 69 nm, a small signal gain of 20.8 dB, a 0-140 mA bias current range, a saturation output power of 10 dBm, a response frequency of 0.6 GHz, and a noise figure of 7.0 dB was used. An SHF 100 BP RF amplifier was selected by calculating the amplified MSE relative to the direct signal for different amplifiers, enabling a full dynamic range peak-to-peak voltage of 7V. A 50Ω resistor was placed before the SOA, allowing for the maximum allowed dynamic current range of 140 mA to be applied across the SOA.

The 70 mA optimum SOA bias current was found by measuring how MSE, optical signal-to-noise ratio (OSNR), rise time, overshoot, and optical gain varied with current. A 70 mA bias using a -2.5 dBm SOA input laser power produced

the lowest rise time and MSE. The SOA was therefore driven between 0 and 140 mA centred at 70 mA. The other equipment used included a Lightwave 7900b lasing system, an Agilent 8156A optical attenuator, an LDX-3200 Series bias current source, a Tektronix 7122B AWG with 12 GSPS sampling frequency, an Anritsu M59740A optical spectrum analyser (OSA), and an Agilent 86100C oscilloscope (OSC) with an embedded photodiode. The RF signal going into the SOA had a rise time of 180 ps, therefore this was the best possible rise time (and settling time) that the SOA could have achieved. Throughout the experiments, a wavelength of 1,545 nm was used.

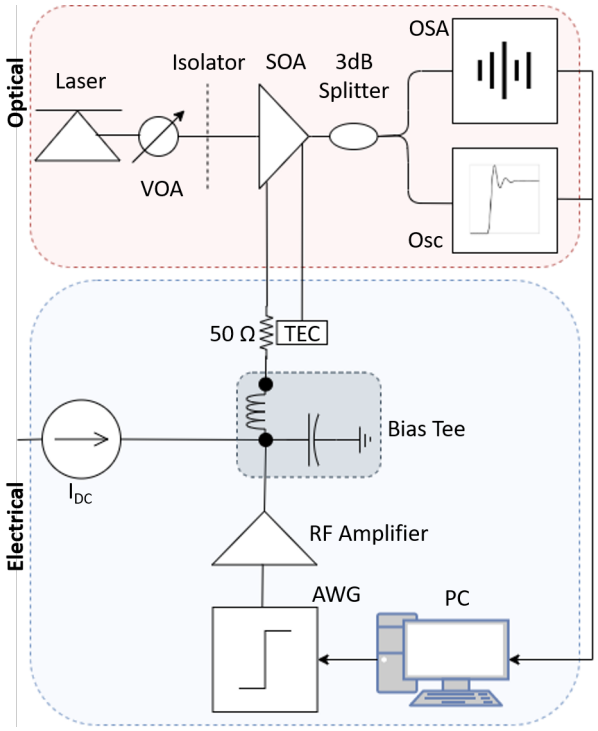


FIGURE 3.11: Diagram of the SOA experimental setup used.

3.7 Results & Discussion

3.7.1 Hyperparameter Tuning & Generality Testing in Simulation

Verifying simulation fidelity. To verify that the equivalent circuit was accurately simulating the SOA, Fig. 3.12 compares the frequency response of the theoretical TF with the experimental SOA. The TF had a -3dB bandwidth of 0.5 GHz (around 700 ps rise time) compared to the experimental SOA's 0.6 GHz (around 550 ps rise time). These values were similar to one another and consistent with both the theoretical and experimental optical responses. The differences between the responses were due to the use of equivalent circuit parameters from the literature which did not exactly match those of our SOA.

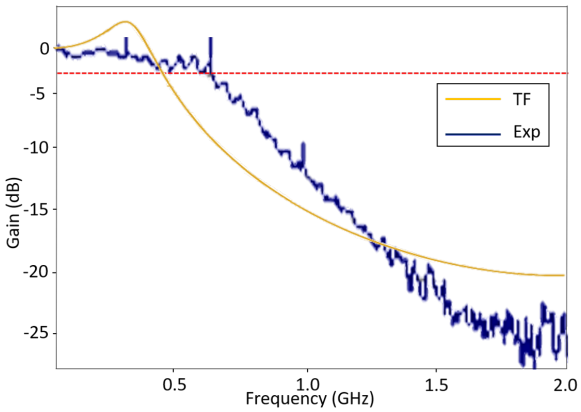


FIGURE 3.12: Frequency responses of the theoretical transfer function (TF) and the experimental SOA (Exp).

PSO hyperparameter tuning. The simulation environment enabled the PSO hyperparameters to be rapidly tuned by plotting the PSO learning curve (MSE vs. number of iterations). Since the same PSO algorithm ran multiple times may converge on different minima, each PSO version with its unique hyperparameters was ran 10 times and the 10 corresponding learning curves plotted on the same graph to get a 'cost spread' (i.e. how much the converged solution's MSE varied between PSO runs). A lower cost spread gave greater

reliability that PSO had converged on the best solution that it could find rather than getting stuck in a local minimum.

To help with convergence time and performance, some additional hyperparameters were defined:

- $iter_{max}$ = Maximum number of iterations that PSO could evolve through before termination. Higher gives more time for convergence but longer total optimisation time.
- max_v_f = Factor controlling the maximum velocity a particle could move with at each iteration. Higher can improve convergence time but, if too high, particles may oscillate around the optimum and never converge.
- on_s_f and off_s_f = 'On' and 'off' suppression factors used to set the minimum and maximum driving signal amplitudes the particle positions could take when the step signal was 'on' and 'off' respectively. Lower will restrict the particle search space to make the problem tractable for the algorithm, but too low will impact the generalisability of the algorithm to any SOA.
- $shell\_w\_f$ = Factor by which to multiply the 'on' time of the signal to get the width of the leading edge of the PISIC shell. Higher (wider) value will give the algorithm more freedom to rise over a longer period at the leading edge of the signal and improve generalisability, but will also increase the size of the search space and impact convergence.

To begin with, it was found that using dynamic PSO whereby w, c_1 and c_2 were adapted at the beginning of each generation led to multiple advantages. First, the solution found by 10 dynamic particles had the same MSE as that found by 2,560 static particles, reducing the computation time by a factor of 256. Second, the final driving signal found by adaptive PSO was significantly less noisy since it was less prone to local minima. Third, the final MSE found

was 63% lower. Fourth, although the relative cost spread of dynamic PSO was 72% compared to 50% due to the lower MSE, the absolute cost spread was just 8.7×10^{-13} compared to 140×10^{-13}.

Pursuing with dynamic PSO, it was found that placing a 'PISIC shell' on the search space (with $shell\_w\_f = 0.1$) beyond which the particles could not travel led to an absolute cost spread of 6.9×10^{-13} and a further 14% reduction in the final cost (despite initial costs being higher due to the fact that PISIC signals lead to greater overshoot and subsequently also greater oscillations). It was also found that initialising one of the n particle positions as a step driving signal improved the convergence time by a factor of two.

Using dynamic PSO, a PISIC shell and an embedded step, the following hyperparameter values were found to give the best spread, final cost and convergence time: $iter_{max} = 150$, $n = 160$, $max\_v\_f = 0.05$, $w(0) = 0.9$, $w(n_t) = 0.5$, $c_{min} = 0.1$, $c_{max} = 2.5$, $on\_s\_f = 2.0$, and $off\_s\_f = 0.2$. This final tuning resulted in a cost spread of just 1.8%. The evolution of this PSO tuning process is summarised in Fig. 3.13, where the learning curves for the above sets of hyperparameters have been plotted in red, orange, blue and green respectively.

The final PSO SOA output, shown in Fig. 3.13, had a rise time, settling time and overshoot of 669 ps, 669 ps and 3.7% respectively. Fig. 3.13 also shows the optical response to a step driving signal, showing a rise time, settling time and overshoot of 669 ps, 4.85 ns and 31.1% respectively. Thus, the simulations indicated that the settling time (and therefore the effective off-on switching time) could be reduced by a factor of 7.2 and the overshoot by a factor of 8.4 compared to a step. Although rise time remained unimproved, the laboratory results in Section 3.7.2 show that, for a real SOA with optical drift, PSO improves all three parameters.

ACO hyperparameter tuning. The important hyperparameters with respect to ACO (specifically the Ant Colony System algorithm used here) were the pheromone exponent (where higher values encourage more exploitation

of previously found paths), the evaporation exponent (where higher values discourage exploitation of previously found paths), the probability of an ant travelling along a randomly selected path, and the number of ants n.

Parameters were tuned by means of running optimisation routines with one hyperparameter varying across a range of values and the rest kept constant. For each MSE value, the learning curve from 10 different runs were plotted against each other. Just as with PSO, parameter values were selected to prioritise the minimisation of cost spread to ensure that the optimisation technique could give consistent results when used on different occasions. Firstly, it was found that beyond 200 ants, the cost spread did not improve significantly. Similarly, regardless of the spread, the ACO routine was typically converging after between 60 and 75 generations, so a generation cap of 100 was imposed since this was sufficient to guarantee convergence. The values for the other parameters were the pheromone constant $\alpha = 0.25$, the evaporation constant $\rho = 0.5$ and the exploration probability $p = 0.1$. It was also found that minimising the search space by reducing the dynamic range of the signal to $\pm 25\%$ centred at 50% of the maximum shortened convergence time without degradation of the final signal, which had the advantage of making matrices memory sizes manageable. No further hyperparameters, such as the PISIC shell applied with the PSO method, were utilised, which is more desirable since fewer hyperparameters simplify the tuning process.

As seen in Fig. 3.13, the spread of the ACO routine was reduced from 23% to 14.9% through tuning, but was still less consistent than the 1.8% spread of the PSO algorithm. Fig. 3.13 shows the convergence of the Ant Colony System algorithm for various hyperparameter combinations (described in the figure's caption). While the spread in the early iterations of the routine is explained by the embedding of a square signal in the PSO routine described above (since it is extremely unlikely to randomly initialise a signal better than a square and the ACO does not use any sort of initial signal embedding), the spread in the later

stages is thought to be due to some practical limitations of the ACO optimisation method. For N parameters with M values each, the ACO routine requires 2 $(N^2 \times M^2)$ matrices (point-wise multiplied to make a third). A 100 point signal with 100 possible values per point gives a matrix with $100,000,000$ elements. Implemented with the popular NumPy Python library, a minimum of 8 bytes per floating point means such a matrix is on the order of gigabytes. Given the relatively low power PC used in the experiment, restrictions on the state space had to be imposed due to memory limitations. This meant that rather than optimising each point on the signal (240) with the maximum resolution allowed by the AWG (8 bit = 256 points), only 180 points (those in the HIGH state of the initial driving step signal) were optimised with a resolution of 50 points. This meant that the state space viewed by the ACO routine was more strongly discretised than that viewed by a method (such as PSO) with lower memory requirements, limiting how optimum the generated signal can be and how well ACO could generalise to other SOAs. Nevertheless, as will be seen, ACO still produced driving signals that improved upon previous methods. The final ACO tuning output, shown in Fig. 3.13, had a rise time, settling time and overshoot of 753 ps, 1.58 ns and 9.1% respectively.

GA hyperparameter tuning. GA often uses a range of different hyperparameters (e.g. *tournsize* for Tournament Selection; or μ, σ, and P_M for Gaussian Mutation). This results in an overall high number of hyperparameters which might significantly impact the probability of the GA getting stuck in a local minimum as well as the speed of convergence. The high number of hyperparameters also meant that there were more values to fine-tune, which made tuning both more complex and time consuming, thereby reducing its generalisability. Since the high number of hyperparameters already impacted generalisability, we refrained from restricting the search space (as done with ACO and with the PSO PISIC shell) to try to still allow for as much generalisability as possible, but this would have the knock-on effect of poorer convergence and a lesser settled signal.

However, as demonstrated in Fig. 3.14, GA was still able to generalise fairly well to 10 different SOAs.

The DEAP Python library was used to implement GA, and came with a set of suggested default hyperparameter values. These were varied using grid search over 61 optimisations. A limit on the number of generations was set to 500, which was found to be sufficient for convergence.

Mutation was implemented using Gaussian Mutation, which has a probability P_M of changing each of an individual's points by applying normally distributed noise of mean μ and standard deviation σ. Using a negative μ led to a solution with lower values, while a positive μ did the opposite - each leading to a lower overall performance, so μ was set to 0. Decreasing P_M or σ slowed down the process as it reduced the overall mutation speed, but increasing either one too much led to the GA getting stuck at local minima. By performing grid search on the hyperparameters, the optimal values were found to be 0.06 and 0.15 respectively. A population size of 60 led to the fastest initial convergence speed (per number of fitness function evaluations), however, the higher number of 100 individuals in a population led to a better overall solution after many generations. Additionally, both P_X and P_M were increased significantly from 0.6 to 0.9 and from 0.05 to 0.3 respectively. Increasing $n_{\text{tournsize}}$ above 4 did not have an impact on the convergence, whereas using the values of 2 and 3 significantly slowed down the process. Most hyperparameters did not change by much from the DEAP library's default values since the initial values were almost optimal and changing them led to a slower convergence.

Fig. 3.13 shows the 10 learning curves for the default hyper parameters (red) and the optimised parameters (green), where the cost spread was reduced from 58.6% to 10.8%. Fig. 3.13 also shows the simulated SOA output of the tuned GA algorithm with a rise time, settling time and overshoot of 799 ps, 2.55 ns, and 9.0% respectively.

Generalising to Unseen SOAs. The hyperparameters of the AI algorithms

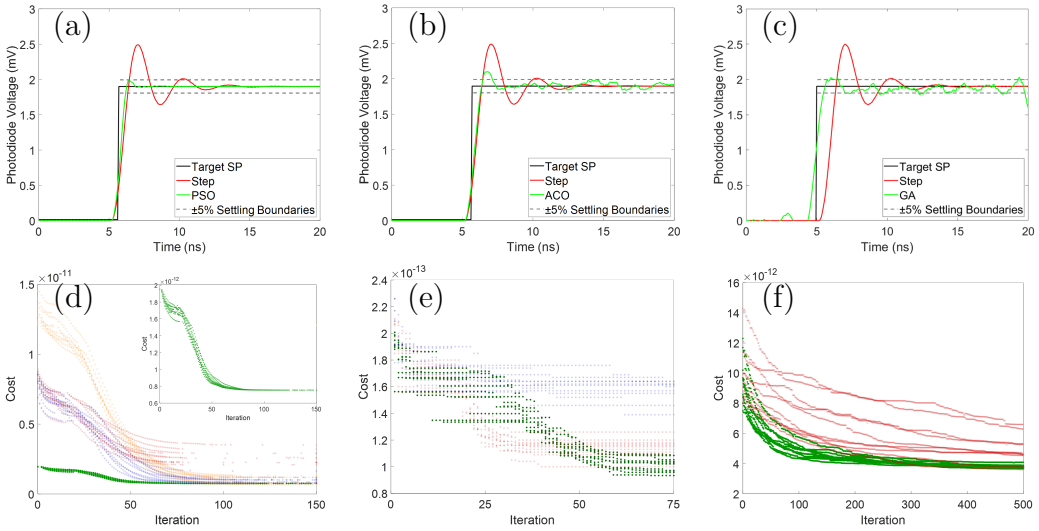


FIGURE 3.13: Simulated SOA optical response to (a) PSO, (b) ACO, and (c) GA driving signals relative to a standard step input. For reference, the target SPs used have also been plotted. Learning curves showing how both the cost spread and the optimum solution improved as the (d) PSO, (e) ACO, and (f) GA algorithms were tuned, showing 10 learning curves for each set of hyperparameters. The curves for the optimum hyperparameters have been plotted in green. For PSO in (d), some additional information has been plotted: i) No dynamic PSO, PISIC shell, or embedded step (red), ii) no PISIC shell or embedded step (blue), iii) no embedded step (orange), and iv) the final PSO algorithm (green, also plotted on separate graph (inserted)). For GA, the i) default DEAP library constants (red) and ii) optimised (green) hyperparameter learning curves have been plotted. For ACO, the blue curve is for a run with a larger pheromone exponent (0.5) value than the optimum, and the red is for a larger dynamic range on the signal search space (±50%).

can be used to address the general problem of 'SOA optimisation'. This is because the hyperparameters are only for restricting the search space to reduce the size of the problem, and restricting how much the algorithm can change its solution between iterations; they are specific to the general SOA optimisation problem, but *not* to a specific SOA. The equivalent circuit simulation environment provided a useful test bed in which to tune the algorithm hyperparameters and allow optimisation of any SOA (even though drive signal solutions derived from simulations are not directly transferable to experiment).

To test the above claim that these algorithms can in theory be generalised to any SOA, we generated 10 different TFs each modelling a different SOA. These were generated by multiplying the coefficients in Table 3.4 by various factors (summarised in Table 3.6 so as to be reproducible), thereby simulating SOAs with different characteristics. The optical outputs of these different SOAs in response to the same step driving signal are shown in Fig. 3.14. Using the PSO and GA algorithms with the *same hyperparameters*, all 10 of these SOAs were able to be optimised with no changes to the algorithms, as shown in Fig. 3.14 (where the AI electrical drive signals have been included for reference). Due to search space restrictions, ACO could not generalise. For all 10 SOAs, a common target set point was chosen. The set point was defined as a perfect 0 overshoot, rise time and settling time step response based on the steady states of the initial step response of one of the simulated SOA's. However, the target can be arbitrarily defined by the user if a different optical response is required, demonstrating the flexibility of the AI algorithms to optimise optical outputs with respect to specific problem requirements. Relative to this target set point, the performances are summarised in Table 3.5. Signals that did not settle have been marked as '-' and excluded from performance summary metrics. PSO had the greatest generalisability to optimising the settling times of different SOAs. Researchers in our field should therefore be able to black box our PSO AI approach and optimise their SOAs even though they will have different equivalent

TABLE 3.5: Performance summary for the techniques applied to the 10 different simulated SOAs, given in the format min | max | mean | standard deviation (best in bold).

- Signals marked '-' never settled.

Technique	Rise Time (ps)	Settling Time (ns)	Overshoot (%)
Step	**502**, **753**, 653, 86.4	3.1, -, 5.8, 3.0	16.5, 70.4, 39.2, 14.1
PSO	669, 837, 703, **58.5**	**0.67**, **1.3**, **0.87**, **0.20**	**2.51**, **6.01**, **4.46**, **1.22**
ACO	**502**, **753**, **644**, 79.4	1.6, -, 2.6, 0.82	11.1, 70.4, 32.6, 17.0
GA	760, 930, 793, **58.5**	1.0, 1.5, 1.3, 1.5	4.31, 9.36, 7.04, 1.54

circuit components from the specific device(s) optimised in this chapter.

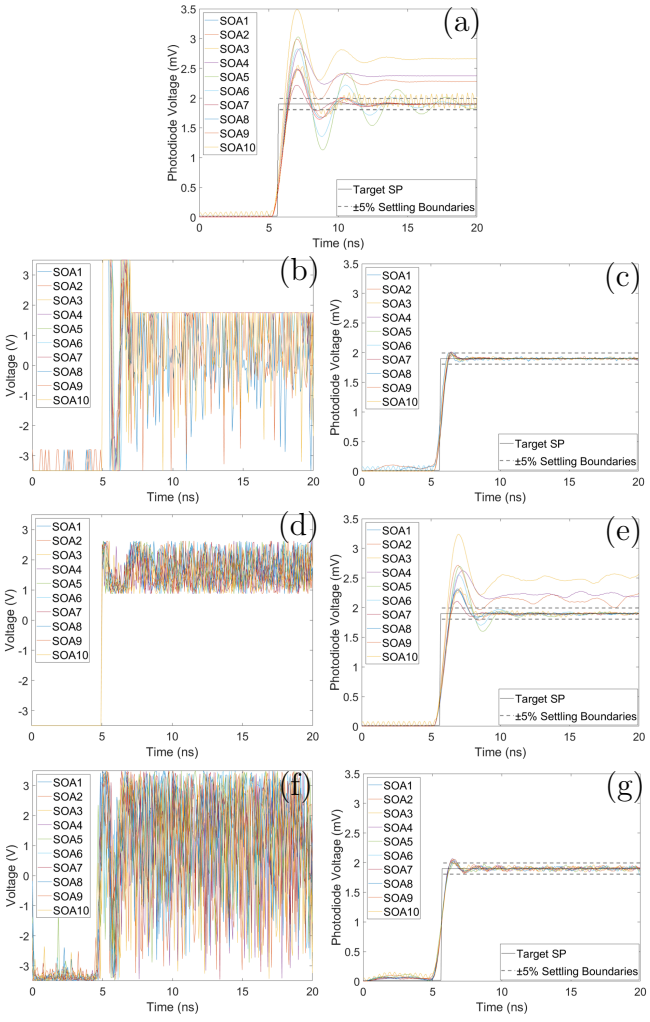


FIGURE 3.14: Simulated SOA optical responses of 10 different SOAs (each with a different transfer function) to (a) step, (c) PSO, (e) ACO, and (g) GA, and the corresponding driving signals for (b) PSO, (d) ACO, and (f) GA. All AI optimisations were done with the same hyperparameters and a common target SP.

TABLE 3.6: Factor(s) used on the EC transfer function coefficients to simulate different SOAs (factor = 1 unless stated otherwise).

TF Component:	Numerator	a_0	a_1	a_2
Factor(s):	1.0, 1.2, 1.4	0.8	0.7, 0.8, 1.2	1.05, 1.1, 1.2

3.7.2 Optimising an SOA in the Laboratory

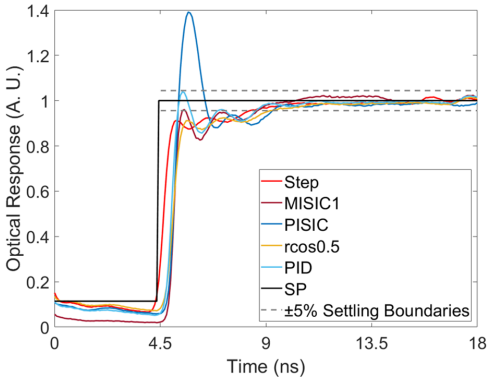


FIGURE 3.15: Experimental SOA responses to the step, PISIC, MISIC1, raised cosine and PID driving signals.

In this section the experimental results for the SOA responses to step, PISIC, MISIC, raised cosine, PID and AI driving signals have been compared. The objective was to reduce the off-on switching time and power oscillations (measured by the settling time and overshoot metrics).

Step. A step driving signal was the simplest format used to drive the SOA. Fig. 3.15 (which has been normalised with respect to the steady state value as done by Figueiredo et al. [2015] for easy comparison) shows the SOA optical response to a step driving signal, resulting in a rise time, settling time and overshoot of 697 ps, 3.72 ns and 0.0% (since it undershot the steady state) respectively.

PISIC. The PISIC format proposed by Gallep and Conforti [2002] was applied to the SOA with 2.95V step + 4.05V impulse, and the response is shown in Fig. 3.15 with a rise time, settling time and overshoot of 502 ps, 4.35 ns and 40.5% respectively. The form of the PISIC pulse used was optimised for the SOA in use, where different step-impulse voltage combinations (as done by

Figueiredo et al. [2015]) were tested, as well as varying widths of the pre-impulse section of the PISIC signal as a percentage of the total signal length centered at the percentage used by Figueiredo et al. [2015]. It was found that a 500ps pulse width gave the best results.

MISIC. The MISIC 1-6 bit-sequences proposed by Figueiredo et al. [2015] were applied with 2.95V step + 4.05V impulse, where the same step-impulse voltage combinations were tested as for PISIC. The format with the best performance was MISIC1, whose response is shown in Fig. 3.15 with a rise time, settling time and overshoot of 502 ps, 4.02 ns and 0.0% (undershot) respectively.

Raised cosine. A popular approach to optimising oscillating systems in control theory is the raised cosine approach, whereby the rising step for a signal of period T is adapted to a rising cosine defined by the frequency-domain piecewise function in (3.19). As β increases ($0 \leq \beta \leq 1$), the rate of signal rise decreases. The best performing raised cosine was $\beta = 0.5$, whose response is shown in Fig. 3.15 and whose rise time, settling time and overshoot were 921 ps, 4.69 ns and 0.0% (undershot) respectively.

$$H(f) = \begin{cases} 1, & \text{if } f \leq \frac{1-\beta}{2T} \\ \frac{1}{2}\left[1 + cos\left(\frac{\pi T}{\beta}\left[f - \frac{1-\beta}{2T}\right]\right)\right], & \text{if } \frac{1-\beta}{2T} < f \leq \frac{1+\beta}{2T} \\ 0, & \text{otherwise} \end{cases} \tag{3.19}$$

PID control. Another popular approach in control theory is the PID controller. The optical response of the PID control signal is shown in Fig. 3.15, with a rise time, settling time and overshoot of 501 ps, 4.02 ns and 2.3% respectively. In order to quickly obtain values for the 3 PID parameters, K_c, K_i and K_d, a First Order Plus Dead Time (FOPDT) model was applied to the SOA, where the key parameters for this model (K_p, τ_p and θ_p) can be measured directly from the step response of the device. The PID tuning parameter, τ_c, which is inversely proportional to the magnitude of the response to offset, was tested with values between that of an 'aggressive' tuning regime ($\tau_c \approx 0.1$) and a 'conservative' one

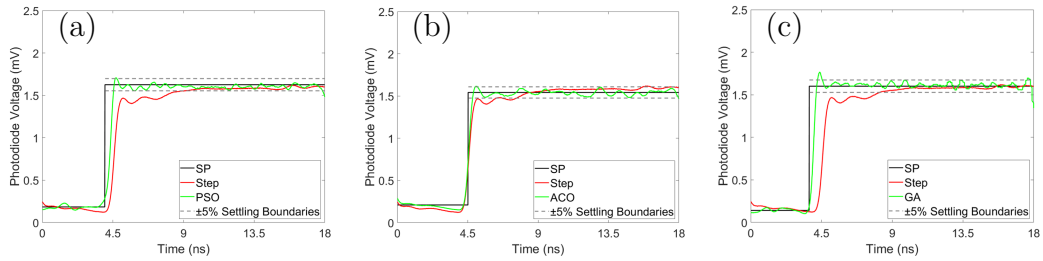


FIGURE 3.16: Experimental results showing the optimised SOA optical outputs for (a) PSO, (b) ACO, and (c) GA.

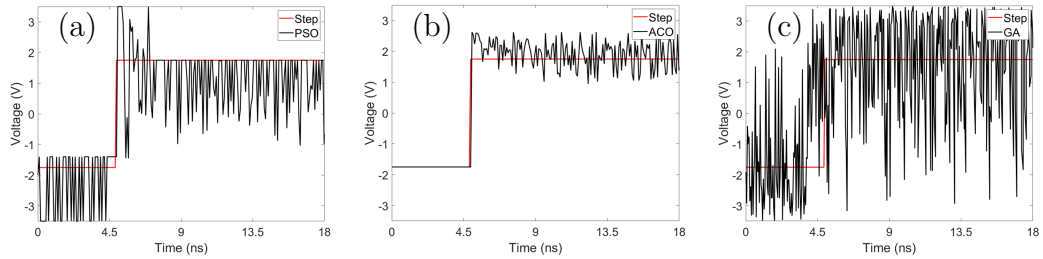


FIGURE 3.17: Experimental results showing the optimised SOA electrical driving signal inputs for (a) PSO, (b) ACO, and (c) GA.

($\tau_c \approx 10.0$). The results shown in Fig. 3.15 are with $\tau_c = 5.0$ which was found to be the best performing value.

PSO. The PSO algorithm used in the simulation environment was applied to the real SOA. The SP and the PSO response are shown in Fig. 3.16, with a rise time, settling time and overshoot of 454 ps, 547 ps and 5.0% respectively.

ACO. An ACO run with 200 ants accomplished a rise time, settling time and overshoot of 413 ps, 560 ps and 4.8% respectively, performing similarly well to the PSO algorithm. The ACO result is shown in Fig. 3.16

GA. Similarly, the GA result shown in Fig. 3.16 had a rise time, settling time, and overshoot of 340 ps, 825 ps, and 10.3% respectively. The rise times of the AI algorithms were an order of magnitude improvement on the step's, and the settling times (and therefore the effective off-on switching time) were several factors faster than the previous MISIC1 optimum from the literature, bringing SOA switching times truly down to the hundred ps scale. A scatter plot comparing these data is shown in Fig. 3.18.

Switching comparison of AI methods. By comparison, PSO had the

lowest settling time and therefore the lowest overall switch time. We hypothesise that this was due to the fact that PSO, being less memory-hungry than ACO and having superior convergence properties compared to GA as a result of having fewer hyperparameters to fine-tune and a smaller search space with the PISIC shell, was able to be given a better search space-hyperparameter tuning trade-off, and therefore was able to find a more optimum driving signal. This larger search space also enabled PSO to explore a wider variety of drive signal solutions without needing a large number of hyperparameters tuned (which adds complexity), allowing PSO to generalise to a more diverse set of SOAs than either ACO or GA were able to. Therefore, although in theory all AI algorithms used were powerful and generalisable, due to the number of hyperparameters and search space restrictions that were required in practice, PSO had both the best performance and generalisability, although GA came close to matching PSO.

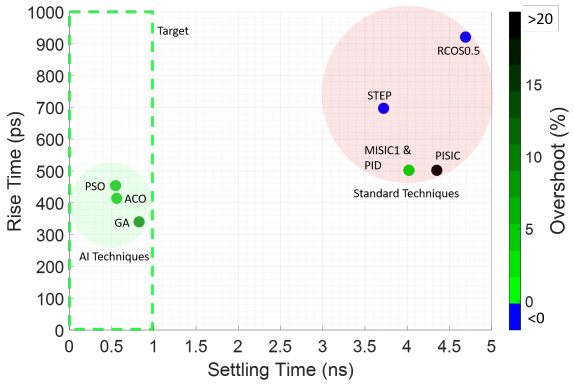


FIGURE 3.18: Scatter plot comparing the experimental rise times, settling times and overshoots of all the driving signals tested. The outlined target region highlights the performance required for truly sub-nanosecond optical switching.

Switching comparison of all methods. Table 3.1 shows results (both absolute and relative improvement for cross comparison) of the rise time, settling time, overshoot and guard time for all methods implemented in this work, as well as a variety from the literature. The rows associated with Figueiredo et al. [2015] are the results for the optimised PISIC and MISIC-6 signals defined and implemented in this work.

Optimised drive signal analysis. Finally, Fig. 3.17 shows the electrical drive signals found by each algorithm. Whilst we stress that the main focus of this chapter is the *method* rather than the *specific drive signal*, the drive signal is important for real-World implementation and general understanding of the search space restrictions used. As Fig. 3.17 shows, the derived driving signals are noisy despite a smooth resultant optical output. This is likely because the AWG (arbitrary waveform generator using an 8-bit digital to analogue converter) drive signal frequency was 6 GHz offering 12 GSa/s whereas the SOA used had a -3dB frequency response of 0.6 GHz, therefore we over-sampled the drive signal by approximately 10×. In a real DCN scenario, to implement our algorithms' driving signals in practice, we would likely use an FPGA or ASIC with an embedded on-chip DAC for multilevel signal generation, and there are already existing FPGAs (a.k.a. RF System on Chip (RFSoC)) that support multiple DACs at 6 GSa/s. Therefore in practice the search space would be lower (fewer dimensions/number of points to optimise) than assumed in this chapter, and we would expect this to improve the AI convergence characteristics. Further experiments using fewer points in the drive signal/a slower AWG are necessary to see what the true effects are on the AI algorithms. This is beyond the scope of this chapter, and we intend to further investigate it in our future work.

Signal noise analysis. Within the context of a DCN implementation of the presented methods, some considerations were made with respect to the effect that the algorithms have on the signal to noise ratio (SNR). Namely, it should be considered if the oscillations caused by the algorithms (all of which are of the order of 5%) have a negative effect on the SNR of the 'on' period of the output, particularly in comparison to the output of a step driving signal, where the 'on' period considered is defined as starting when the signal enters the ±5% (with respect to the steady state) region for a 20 ns pulse length. Following from the model of amplifier noise given by Agrawal [2002] and accounting for Shot noise, intrinsic amplifier noise (the noise figure of the SOA) and the additional

noise due to the fluctuations in the output, we consider the penalty on the noise figure (as defined by Agrawal [2002]) due to the deviations of the output from its steady state value throughout the duration of its 'on' period. Assuming (based on intrinsic and Shot noise contributions) a base noise figure (i.e. if the driving method caused no deviations at all) of 7.1dB, the measured noise figure penalties for ACO, PSO, GA and step were 1.05 dB, 0.65 dB, 1.12 dB and 0.53 dB with SNR values of 28.52 dB, 28.90 dB, 28.54 dB and 29.06 dB respectively, showing that the additional noise figure penalty due to the AI methods ranges between 0.08 dB (PSO) and 0.59 dB (GA) compared to a step in the case of the best performing algorithm (PSO).

3.8 Conclusions, Limitations, & Further Work

In this chapter, simulation and experimental results of SOA off-on switching were presented for various driving signal formats. The chapter outlined a novel approach to SOA driving signal generation with AI algorithms which made no assumptions about the SOA and therefore were general, required no historic data collection, and could be scaled to any SOA-based switch, opening up the possibility of rapid all-optical switching in real data centres. World-record settling times (and therefore effective off-on times) of 547 ps were achieved using PSO, offering an order of magnitude performance improvement with respect to settling time over our implementation of the PISIC and MISIC techniques from the literature and thereby establishing a new state-of-the-art. Additionally, the standard PID control and raised cosine techniques from control theory were shown to be inadequate for the problem of ultra-fast SOA switching. Although ACO and GA demonstrated slightly faster rise times than PSO, PSO had a faster settling time and also a significantly lower 1.8% cost spread, giving greater reliability that any given PSO run had found the optimum solution. Furthermore, due to the fewer restrictions placed on the search space and the lower number

of fine-tuned hyperparameters compared to ACO and GA, PSO was found to be more easy to generalise to unseen SOAs.

While this is good progress, there is much further work needed to make this technology viable for production DCNs.

Robustness to external noise. In DCN systems operating over long periods of time, environmental factors external to the SOA such as the temperature and the driving current may fluctuate. An interesting area of further work would be to test the optimised driving signal's robustness to these external fluctuations. To mitigate their impact, new methods could be developed which stochastically sample different temperatures and bias currents during the optimisation process to see whether the AI algorithms can account for these varying inputs in their final optimised driving signal. Alternatively, a lookup table could be created mapping external conditions such as temperature and bias current to the corresponding optimal driving signal found by the AI algorithm under those specific conditions.

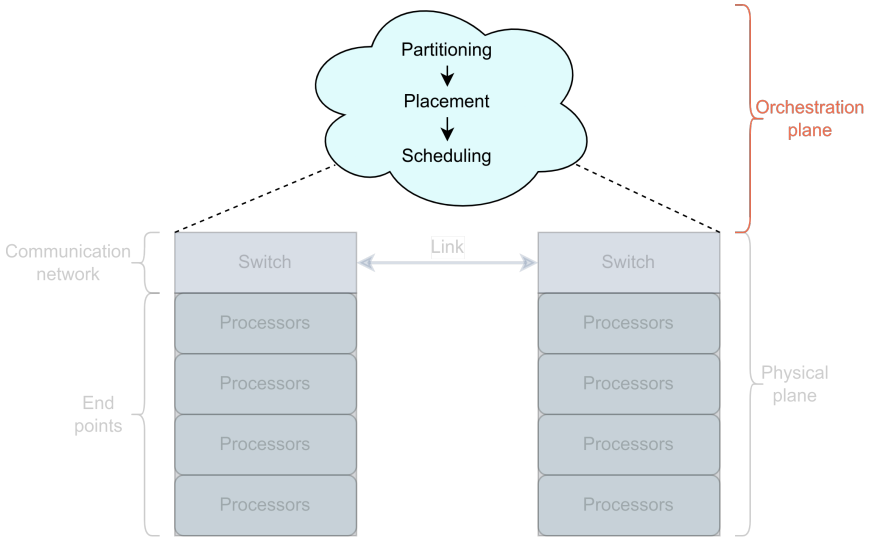
Lower resolution drive signal. As previously discussed, the frequency of the signal driving the SOA was 6 GHz whereas the SOA's frequency response was around 0.6 GHz, leading to 10× unnecessary oversampling. Future works might therefore consider reducing the sampling rate of the drive signal to (1) reduce the optimisation search space and thus improve AI convergence, and (2) reduce the complexity of the hardware needed to drive the SOAs in a production environment. On this latter point, it might be useful to develop approaches which *undersample* the SOA drive signal to enable the use of cheap and low-complexity hardware such as FPGAs and specialised ASICs.

Real data transmission. Although Gerard et al. [2021] took the work developed in this chapter and built an end-to-end tuneable light source, the system has not yet been used to transmit real data from source to destination. This would be a necessary step to measure the true BER and usefulness of an OCS communication network using the SOA switching method proposed here.

Cascaded SOAs. In this chapter we considered the simple setting of either blocking or amplifying a single light source with a single SOA. Gerard et al. [2021] extended this to a setting with two SOAs, however in practice a single switch device might contain many SOAs which might be cascaded in order to facilitate more complex network routing. Interesting research questions include whether or not the same optimised signal could be applied to each SOA in a cascade, or if the optimisation algorithm could collectively optimise the whole cascade simultaneously and how this might make the optimisation problem more difficult with a larger search space with more complex inter-SOA dependencies.

Part II

Optimising the Orchestration Plane



Chapter 4

Solving NP-Hard Discrete Optimisation Problems

Abstract

Combinatorial optimisation problems framed as mixed integer linear programmes (MILPs) are ubiquitous across a range of real-world applications. The canonical branch-and-bound algorithm seeks to exactly solve MILPs by constructing a search tree of increasingly constrained sub-problems. In practice, its solving time performance is dependent on heuristics, such as the choice of the next variable to constrain ('branching'). Recently, machine learning (ML) has emerged as a promising paradigm for branching. However, prior works have struggled to apply reinforcement learning (RL), citing sparse rewards, difficult exploration, and partial observability as significant challenges. Instead, leading ML methodologies resort to approximating high quality handcrafted heuristics with imitation learning (IL), which precludes the discovery of novel policies and requires expensive data labelling. This chapter proposes *retro branching*; a simple yet effective approach to RL for branching. By retrospectively deconstructing the search tree into multiple paths each contained within a sub-tree, we enable the agent to learn from shorter trajectories with more predictable next states. In experiments on four combinatorial tasks, our approach enables learning-to-branch without any expert guidance or pre-training. We outperform the current

state-of-the-art RL branching algorithm by 3-5× and come within 20% of the best IL method's performance on MILPs with 500 constraints and 1000 variables, with ablations verifying that our retrospectively constructed trajectories are essential to achieving these results.

Publications related to this work (contributions indented):

- **Christopher W. F. Parsonson**, Alexandre Laterre, and Thomas D. Barrett, 'Reinforcement Learning for Branch-and-Bound Optimisation using Retrospective Trajectories', *AAAI'23: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence*, 2023
  - Algorithm, code, experiments, paper writing, plots
- Thomas D. Barrett, **Christopher W. F. Parsonson**, and Alexandre Laterre, 'Learning to Solve Combinatorial Graph Partitioning Problems via Efficient Exploration', *arXiv*, 2022
  - Baseline comparison experiments, abstract/introduction/related work/background paper writing, plots

4.1 Introduction

A plethora of real-world problems fall under the broad category of CO (vehicle routing and scheduling [Korte and Vygen, 2012]; protein folding [Perdomo-Ortiz et al., 2012]; fundamental science [Barahona, 1982]). Many CO problems can be formulated as MILPs whose task is to assign discrete values to a set of decision variables, subject to a mix of linear and integrality constraints, such that some objective function is maximised or minimised. The most popular method for finding exact solutions to MILPs is B&B [Land and Doig, 1960]; a collection of heuristics which increasingly tighten the bounds in which an optimal solution can reside (see Section 4.2). Among the most important of these heuristics is *variable selection* or *branching* (which variable to use to partition the chosen node's search space), which is key to determining B&B solve efficiency [Achterberg and Wunderling, 2013].

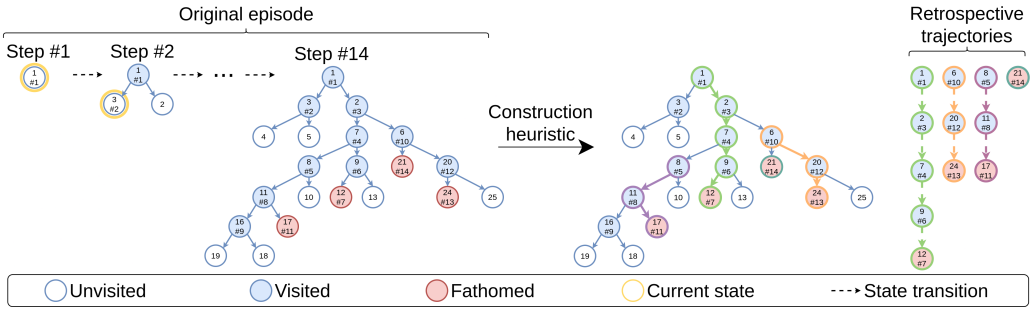


FIGURE 4.1: The proposed retro branching approach used during training. Each node is labelled with: Top: The unique ID assigned when it was added to the tree, and (where applicable); bottom: The step number (preceded by a '#') at which it was visited by the brancher in the original MDP. The MILP is first solved with the brancher and the B&B tree stored as usual (forming the 'original episode'). Then, ignoring any nodes never visited by the agent, the nodes are added to trajectories using some 'construction heuristic' (see Sections 4.4 and 4.6) until each eligible node has been added to one, and only one, trajectory. Crucially, the order of the sequential states within a given trajectory may differ from the state visitation order of the original episode, but all states within the trajectory will be within the same sub-tree. These trajectories are then used for training.

State-of-the-art (SOTA) learning-to-branch approaches typically use the IL paradigm to predict the action of a high quality but computationally expensive

human-designed branching expert [Gasse et al., 2019]. Since branching can be formulated as a MDP [He et al., 2014], RL seems a natural approach. The long-term motivations of RL include the promise of learning novel policies from scratch without the need for expensive expert data, the potential to exceed expert performance without human design, and the capability to maximise the performance of a policy parameterised by an expressivity-constrained DNN.

However, branching has thus far proved largely intractable for RL for reasons we summarise into three key challenges. (1) *Long episodes:* Whilst even random branching policies are theoretically guaranteed to eventually find the optimal solution, poor decisions can result in episodes of tens of thousands of steps for the 500 constraint 1000 variable MILPs considered by Gasse et al. 2019. This raises the familiar RL challenges of reward sparsity [Trott et al., 2019], credit assignment [Harutyunyan et al., 2019], and high variance returns [Mao et al., 2019b]. (2) *Large state-action spaces*: Each branching step might have hundreds or thousands of potential branching candidates with a huge number of unique possible sub-MILP states. Efficient exploration to discover improved trajectories in such large state-action spaces is a well-known difficulty for RL [Agostinelli et al., 2019b, Ecoffet et al., 2021]. (3) *Partial observability*: When a branching decision is made, the next state given to the brancher is determined by the next sub-MILP visited by the node selection policy. Jumping around the B&B tree without the brancher's control whilst having only partial observability of the full tree makes the future states seen by the agent difficult to predict. Etheve et al. 2020 therefore postulated the benefit of keeping the MDP within a sub-tree to improve observability and introduced the SOTA FMSTS RL branching algorithm. However, in order to achieve this, FMSTS had to use a DFS node selection policy which, as we demonstrate in Section 4.6, is highly sub-optimal and limits scalability.

In this chapter, we present *retro branching*; a simple yet effective method to overcome the above challenges and learn to branch via reinforcement. We follow

the intuition of Etheve et al. [2020] that constraining each sequential MDP state to be within the same sub-tree will lead to improved observability. However, we posit that a branching policy taking the 'best' actions with respect to only the sub-tree in focus can still provide strong overall performance *regardless of the node selection policy used*. This is aligned with the observation that leading heuristics such as SB and PB also do not explicitly account for the node selection policy or predict how the global bound may change as a result of activity in other sub-trees. Assuming the validity of this hypothesis, we can discard the DFS node selection requirement of FMSTS whilst retaining the condition that sequential states seen during training must be within the same sub-tree.

Concretely, our retro branching approach (shown in Fig. 4.1 and elaborated on in Section 4.4) is to, during training, take the search tree after the B&B instance has been solved and *retrospectively* select each subsequent state (node) to construct multiple trajectories. Each trajectory consists of sequential nodes within a single sub-tree, allowing the brancher to learn from shorter trajectories with lower return variance and more predictable future states. This approach directly addresses challenges (1) and (3) and, whilst the state-action space is still large, the shorter trajectories implicitly define more immediate auxiliary objectives relative to the tree. This reduces the difficulty of exploration since shorter trajectory returns will have a higher probability of being improved upon via stochastic action sampling than when a single long MDP is considered, thereby addressing (2). Furthermore, retro branching relieves the FMSTS requirement that the agent must be trained in a DFS node selection setting, enabling more sophisticated strategies to be used which are better suited for solving larger, more complex MILPs.

We evaluate our approach on MILPs with up to 500 constraints and 1000 variables, achieving a 3-5$\times$ improvement over FMSTS and coming within $\approx 20\%$ of the performance of the SOTA IL agent of Gasse et al. [2019]. Furthermore, we demonstrate that, for small instances, retro branching can uncover policies

superior to IL; a key motivation of using RL. Our results open the door to the discovery of new branching policies which can scale without the need for labelled data and which could, in principle, exceed the performance of SOTA handcrafted branching heuristics.

4.2 Background

4.2.1 Mixed Integer Linear Programming

An MILP is an optimisation task where values must be assigned to a set of n *decision variables* subject to a set of m linear *constraints* such that some linear *objective function* is minimised. MILPs can be written in the standard form

$$\arg\min_{\mathbf{x}} \left\{ \mathbf{c}^\top \mathbf{x} | \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \right\}, \tag{4.1}$$

where $\mathbf{c} \in \mathbb{R}^n$ is a vector of the objective function's coefficients for each decision variable in $\mathbf{x}$ such that $\mathbf{c}^\top \mathbf{x}$ is the objective value, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a matrix of the m constraints' coefficients (rows) applied to n variables (columns), $\mathbf{b} \in \mathbb{R}^m$ is the vector of variable constraint right-hand side bound values which must be adhered to, and $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ are the respective lower and upper variable value bounds. MILPs are hard to solve owing to their *integrality constraint(s)* whereby $p \leq n$ decision variables must be an integer. If these integrality constraints are relaxed, the MILP becomes a linear programme (LP), which can be solved efficiently using algorithms such as simplex [Nelder and Mead, 1965]. The most popular approach for solving MILPs exactly is B&B.

4.2.2 Branch-and-Bound

B&B is an algorithm composed of multiple heuristics for solving MILPs. It uses a search tree where nodes are MILPs and edges are partition conditions (added constraints) between them. Using a *divide and conquer* strategy, the MILP

is iteratively partitioned into sub-MILPs with smaller solution spaces until an optimal solution (or, if terminated early, a solution with a worst-case optimality gap guarantee) is found. The task of B&B is to evolve the search tree until the provably optimal node is found.

Concretely, as summarised in Fig. 4.2, at each step in the algorithm, B&B: (1) Selects an open (unfathomed leaf) node in the tree whose sub-tree seems promising to evolve; (2) selects ('branches on') a variable to tighten the bounds on the sub-MILP's solution space by adding constraints either side of the variable's LP solution value, generating two child nodes (sub-MILPs) beneath the focus node; (3) for each child, i) solve the relaxed LP (the *dual problem*) to get the *dual bound* (a bound on the best possible objective value in the node's sub-tree) and, where appropriate, ii) solve the *primal problem* and find a feasible (but not necessarily optimal) solution satisfying the node's constraints, thus giving the *primal bound* (the worst-case feasible objective value in the sub-tree); and (4) fathom any children (i.e. consider the sub-tree rooted at the child 'fully known' and therefore excluded from any further exploration) whose relaxed LP solution is integer-feasible, is worse than the incumbent (the globally best feasible node found so far), or which cannot meet the non-integrality constraints of the MILP. This process is repeated until the *primal-dual gap* (global primal-dual bound difference) is 0, at which point a provably optimal solution to the original MILP will have been found.

Note that the heuristics (i.e. primal, branching, and node selection) at each stage jointly determine the performance of B&B. More advanced procedures such as cutting planes [Mitchell, 2009] and column generation [Barnhart et al., 1998] are available for enhancement, but are beyond the scope of this work. Note also that solvers such as SCIP 2022 only store 'visitable' nodes in memory, therefore in practice fathoming occurs at a feasible node where a branching decision led to the node's two children being outside the established optimality bounds, being infeasible, or having an integer-feasible dual solution, thereby

closing the said node's sub-tree.

4.3 Related Work

Classical branching heuristics. PB [Benichou et al., 1971] and strong branching (SB) [Applegate et al., 1995, 2007] are two canonical branching algorithms. PB selects variables based on their historic branching success according to metrics such as bound improvement. Although the per-step decisions of PB are computationally fast, it must initialise the variable pseudocosts in some way which, if done poorly, can be particularly damaging to overall performance since early B&B decisions tend to be the most influential. SB, on the other hand, conducts a one-step lookahead for all branching candidates by computing their potential local dual bound gains before selecting the most favourable variable, and thus is able to make high quality decisions during the critical early stages of the search tree's evolution. Despite its simplicity, SB is still today the best known policy for minimising the overall number of B&B nodes needed to solve the problem instance (a popular B&B quality indicator). However, its computational cost renders SB infeasible in practice.

Learning-to-branch. Recent advances in deep learning have led ML researchers to contribute to exact CO (surveys provided by Lodi and Zarpellon 2017, Bengio et al. 2021, and Cappart et al. 2021). Khalil et al. 2016 pioneered the community's interest by using IL to train a support vector machine (SVM) to imitate the variable rankings of SB after the first 500 B&B node visits and thereafter use the SVM. Alvarez et al. 2017 similarly imitated SB, but learned to predict the SB scores directly using Extremely Randomized Trees [Geurts et al., 2006]. These approaches performed promisingly, but their per-instance training and use of SB at test time limited their scalability.

These issues were overcome by Gasse et al. 2019, who took as input a bipartite graph representation capturing the current B&B node state and predicted the

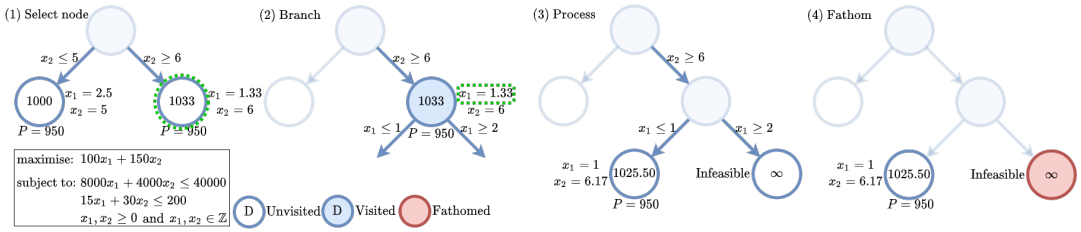


FIGURE 4.2: Typical 4-stage procedure iteratively repeated by B&B to solve an MILP. Each node represents an MILP derived from the original MILP being solved, and each edge represents the constraint added to derive a new child node (sub-MILP) from a given parent. Each node is labelled with the decision variable values of the solved LP relaxation on the right hand side, the corresponding dual bound in the centre, and the established primal bound beneath. Each edge is labelled with the introduced constraint to generate the child node. Green dotted outlines are used to indicate which node and variable were selected in stages (1) and (2) to lead to stages (3) and (4). The global primal (P) and dual (D) bounds are increasingly constrained by repeating stages 1-4 until P and D are equal, at which point a provably optimal solution will have been found. Note that for clarity we only show the detailed information needed at each stage, but that this does not indicate any change to the state of the tree.

corresponding action chosen by SB using a graph convolutional network (GCN). This alleviated the reliance on extensive feature engineering, avoided the use of SB at inference time, and demonstrated generalisation to larger instances than seen in training. Works since have sought to extend this method by introducing new observation features to generalise across heterogeneous CO instances [Zarpellon et al., 2021] and designing SB-on-a-GPU expert labelling methods for scalability [Nair et al., 2021].

Etheve et al. 2020 proposed FMSTS which, to the best of our knowledge, is the only published work to apply RL to branching and is therefore the SOTA RL branching algorithm. By using a DFS node selection strategy, they used the DQN approach [Mnih et al., 2013] to approximate the Q-function of the B&B sub-tree size rooted at the current node; a local Q-function which, in their setting, was equivalent to the number of global tree nodes. Although FMSTS alleviated issues with credit assignment and partial observability, it relied on using the DFS node selection policy (which can be far from optimal), was fundamentally limited by exponential sub-tree sizes produced by larger

instances, and its associated models and data sets were not open-accessed.

4.4 Retro Branching Methodology

We now describe our retro branching approach for learning-to-branch with RL.

States. At each time step t the B&B solver state is comprised of the search tree with past branching decisions, per-node LP solutions, the global incumbent, the currently focused leaf node, and any other solver statistics which might be tracked. To convert this information into a suitable input for the branching agent, we represent the MILP of the focus node chosen by the node selector as a bipartite graph. Concretely, the n variables and m constraints are connected by edges denoting which variables each constraint applies to. This formulation closely follows the approach of Gasse et al. 2019, with a full list of input features at each node detailed in Appendix A.5.

Actions. Given the MILP state s_t of the current focus node, the branching agent uses a policy $\pi(u_t|s_t)$ to select a variable u_t from among the p branching candidates.

Original full episode transitions. In the original full B&B episode, the next node visited is chosen by the node selection policy from amongst any of the open nodes in the tree. This is done independently of the brancher, which observes state information related only to the current focus node and the status of the global bounds. As such, the transitions of the 'full episode' are partially observable to the brancher, and it will therefore have the challenging task of needing to aggregate over unobservable states in external sub-trees to predict the long-term values of states and actions.

Retrospectively constructed trajectory transitions (retro branching). To address the partial observability of the full episode, we retrospectively construct multiple trajectories where all sequential states in a given trajectory are within the same sub-tree, and where the trajectory's terminal state is chosen

from amongst the as yet unchosen fathomed sub-tree leaves. A visualisation of our approach is shown in Fig. 4.1. Concretely, during training, we first solve the instance as usual with the RL brancher and any node selection heuristic to form the 'original episode'. When the instance is solved, rather than simply adding the originally observed MDP's transitions to the DQN replay buffer, we retrospectively construct multiple trajectory paths through the search tree. This construction process is done by starting at the highest level node not yet added to a trajectory, selecting an as yet unselected fathomed leaf in the sub-tree rooted at said node using some 'construction heuristic' (see Section 4.6), and using this root-leaf pair as a source-destination with which to construct a path (a 'retrospective trajectory'). This process is iteratively repeated until each eligible node in the original search tree has been added to one, and only one, retrospective trajectory. The transitions of each trajectory are then added to the experience replay buffer for learning. Note that retrospective trajectories are only used during training, therefore retro branching agents have no additional inference-time overhead.

Crucially, retro branching determines the sequence of states in each trajectory (i.e. the transition function of the MDP) such that the next state(s) observed in a given trajectory will *always* be within the same sub-tree (see Fig. 4.1) regardless of the node selection policy used in the original B&B episode. Our reasoning behind this idea is that the state(s) beneath the current focus node within its sub-tree will have characteristics (bounds, introduced constraints, etc.) which are strongly related with those of the current node, making them more observable than were the next states to be chosen from elsewhere in the search tree, as can occur in the 'original B&B' episode. Moreover, by correlating the agent's maximum trajectory length with the depth of the tree rather than the total number of nodes, reconstructed trajectories have orders of magnitude fewer steps and lower return variance than the original full episode, making learning tractable on large MILPs. Furthermore, because the sequential nodes visited

are chosen retrospectively in each trajectory, unlike with FMSTS, any node selection policy can be used during training. As we show in Section 4.6, this is a significant help when solving large and complex MILPs.

Rewards. As demonstrated in Section 4.6, the use of reconstructed trajectories enables a simple distance-to-goal reward function to be used; a $r = -1$ punishment is issued to the agent at each step except when the agent's action fathomed the sub-tree, where the agent receives $r = 0$. This reward was chosen because it provides an incentive for the the branching agent to reach the terminal state as quickly as possible. This auxiliary objective is desirable because, when aggregated over all trajectories in a given sub-tree, it corresponds to fathoming the whole sub-tree (and, by extension, solving the MILP) in as few steps as possible. This is because the only nodes which are stored by SCIP 2022 and which the brancher will be presented with will be feasible nodes which *potentially* contain the optimal solution beneath them. As such, any action chosen by the brancher which provably shows either the optimal solution to not be beneath the current node or which finds an integer feasible dual solution (i.e. an action which fathoms the sub-tree beneath the node) will be beneficial, because it will prevent SCIP from being able to further needlessly explore the node's sub-tree.

A note on partial observability. In the above retrospective formulation of the branching MDP, the primal, branching, and node selection heuristics active in other sub-trees will still influence the future states and fathoming conditions of a given retrospective trajectory. We posit that there are two extremes; DFS node selection where future states are fully observable to the brancher, and non-DFS node selection where they are heavily obscured. As shown in Section 4.6, our retrospective node selection setting strikes a balance between these two extremes, attaining sufficient observability to facilitate learning while enabling the benefits of short, low variance trajectories with sophisticated node selection strategies which make handling larger MILPs tractable.

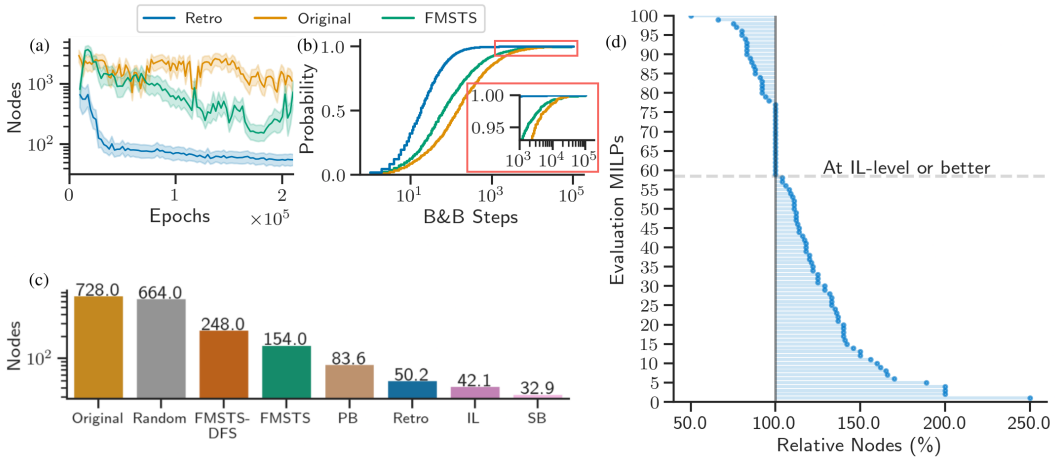


FIGURE 4.3: Performances of the branching agents on the 500×1000 set covering instances. (a) Validation curves for the RL agents evaluated in the same non-DFS setting. (b) CDF of the number of B&B steps taken by the RL agents for each instance seen during training. (c) The best validation performances of each branching agent. (d) The instance-level validation performance of the retro branching agent relative to the IL agent, with RL matching or beating IL on 42% of test instances.

4.5 Experimental Setup

All code for reproducing the experiments and links to the generated data sets are provided at `https://github.com/cwfparsonson/retro_branching`.

Network architecture and learning algorithm. We used the GCN architecture of Gasse et al. 2019 to parameterise the DQN value function with some minor modifications which we found to be helpful (see Appendix A.2.1). We trained our network with n-step DQN [Sutton, 1988, Mnih et al., 2013] using prioritised experience replay [Schaul et al., 2016], soft target network updates [Lillicrap et al., 2019], and an epsilon-stochastic exploration policy (see Appendix A.1.1 for a detailed description of our RL approach and the corresponding algorithms and hyperparameters used).

B&B environment. We used the open-source Ecole [Prouvost et al., 2020] and PySCIPOpt [Maher et al., 2016] libraries with SCIP 7.0.1 [SCIP, 2022] as the backend solver to do instance generation and testing. Where possible, we used the training and testing protocols of Gasse et al. [2019].

MILP Problem classes. In total, we considered four NP-hard problem benchmarks: set covering [Balas et al., 2018], combinatorial auction [Leyton-Brown et al., 2000], capacitated facility location [Litvinchev and Ozuna Espinosa, 2012], and maximum independent set [Bergman et al., 2016].

Baselines. We compared retro branching against the SOTA FMSTS RL algorithm of Etheve et al. [2020] (see Appendix A.6 for implementation details) and the SOTA IL approach of Gasse et al. [2019] trained and validated with 100 000 and 20 000 strong branching samples respectively. For completeness, we also compared against the SB heuristic imitated by the IL agent, the canonical PB heuristic, and a random brancher (equivalent in performance to most infeasible branching [Achterberg et al., 2004]). Note that we have ommited direct comparison to the SOTA tuned commercial solvers, which we do not claim to be competitive with at this stage. To evaluate the quality of the agents' branching decisions, we used 100 validation instances which were unseen during training, reporting the total number of tree nodes and LP iterations as key metrics to be minimised. As shown in Appendix A.3, 100 instances was a sample size with sufficient statistical significance to confidently draw conclusions about the relative performance between the algorithms being evaluated.

4.6 Results & Discussion

4.6.1 Performance of Retro Branching

Comparison to the SOTA RL branching heuristics. We considered set covering instances with 500 rows and 1000 columns. To demonstrate the benefit of the proposed retro branching method, we trained a baseline 'Original' agent on the original full episode, receiving the same reward as our retro branching agent (-1 at each non-terminal step and 0 for a terminal action which ended the episode - see Section 4.4 for details). We also trained the SOTA RL FMSTS

branching agent in a DFS setting and, at test time, validated the agent in both a DFS ('FMSTS-DFS') and non-DFS ('FMSTS') environment to fairly compare the policies. Note that the FMSTS agent serves as an ablation to analyse the influence of training on retrospective trajectories, since it uses our auxiliary objective but without retrospective trajectories, and that the Original agent further ablates the auxiliary objective since its 'terminal step' is defined as ending the B&B episode (where it receives $r_t = 0$ rather than $r_t = -1$). As shown in Fig. 4.3a, the Original agent was unable to learn on these large instances, with retro branching achieving 14× fewer nodes at test time. FMSTS also performed poorly, with highly unstable learning and a final performance 5× and 3× poorer than retro branching in the DFS and non-DFS settings respectively (see Fig. 4.3c). We posit that the cause of the poor FMSTS performance is due to its use of the sub-optimal DFS node selection policy, which is ill-suited for handling large MILPs and results in $\approx 10\%$ of episodes seen during training being on the order of 10-100k steps long (see Fig. 4.3b), which makes learning significantly harder for RL.

Comparison to non-RL branching heuristics. Having demonstrated that the proposed retro branching method makes learning-to-branch at scale tractable for RL, we now compare retro branching with the baseline branchers to understand the efficacy of RL in the context of the current literature. Fig. 4.3c shows how retro branching compares to other policies on large 500×1000 set covering instances. While the agent outperforms PB, it only matches or beats IL on 42% of the test instances (see Fig. 4.3d) and, on average, has a $\approx 20\%$ larger B&B tree size. Therefore although our RL agent was still improving and was limited by compute (see Appendix A.1.2), and in spite of our method outperforming the current SOTA FMSTS RL brancher, RL has not yet been able to match or surpass the SOTA IL agent at scale. This will be an interesting area of future work, as discussed in Section 4.7.

4.6.2 Analysis of Retro Branching

Verifying that RL can outperform IL. In addition to not needing labelled data, a key motivation for using RL over IL for learning-to-branch is the potential to discover superior policies. While Fig. 4.3 showed that, at test-time, retro branching matched or outperformed IL on 42% of instances, IL still had a lower average tree size. As shown in Table 4.1, we found that, on small set covering instances with 165 constraints and 230 variables, RL could outperform IL by $\approx 20\%$. While improvement on problems of this scale is not the primary challenge facing ML-B&B solvers, we are encouraged by this demonstration that it is possible for an RL agent to learn a policy better able to maximise the performance of an expressivity-constrained network than imitating an expert such as SB without the need for pre-training or expensive data labelling procedures (see Appendix A.8).

For completeness, Table 4.1 also compares the retro branching agent to the IL, PB, and SB branching policies evaluated on 100 unseen instances of four NP-hard CO benchmarks. We considered instances with 10 items and 50 bids for combinatorial auction, 5 customers and facilities for capacitated facility location, and 25 nodes for maximum independent set. RL achieved a lower number of tree nodes than PB and IL on all problems except combinatorial auction. This highlights the potential for RL to learn improved branching policies to solve a variety of MILPs.

Table 4.1: Test-time comparison of the best agents on the evaluation instances of the four NP-hard small CO problems considered.

	Set Covering		Combinatorial Auction		Capacitated Facility Location		Maximum Independent Set	
Method	# LPs	# Nodes	# LPs	# Nodes	# LPs	# Nodes	# LPs	# Nodes
SB	184	6.76	13.2	4.64	28.2	10.2	19.2	3.80
PB	258	12.8	22.0	7.80	**28.0**	10.2	25.4	5.77
IL	244	10.5	**16.0**	**5.29**	**28.0**	10.2	20.1	4.08
Retro	**206**	**8.68**	18.1	5.73	28.4	**10.1**	**19.1**	**4.01**

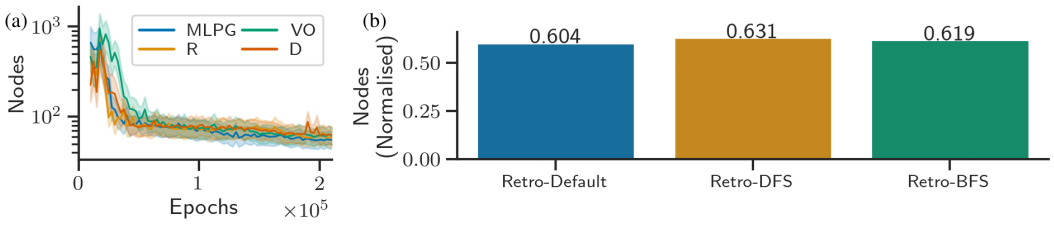


FIGURE 4.4: 500×1000 set covering performances. (a) Validation curves for four retro branching agents each trained with a different trajectory construction heuristic: Maximum LP gain (MLPG); random (R); visitation order (VO); and deepest (D). (b) The performances of the best retro branching agent deployed in three different node selection environments (default SCIP, DFS, and BFS) normalised relative to the performances of PB (measured by number of tree nodes).

Demonstrating the independence of retro branching to future state selection. As described in Section 4.4, in order to retrospectively construct a path through the search tree, a fathomed leaf node must be selected. We refer to the method for selecting the leaf node as the *construction heuristic*. The future states seen by the agent are therefore determined by the construction heurisitc (used in training) and the node selection heuristic (used in training and inference).

During our experiments, we found that the specific construction heuristic used had little impact on the performance of our agent. Fig. 4.4a shows the validation curves for four agents trained on 500×1000 set covering instances each using one of the following construction heuristics: Maximum LP gain ('MLPG': Select the leaf with the largest LP gain); random ('R': Randomly select a leaf); visitation order ('VO': Select the leaf which was visited first in the original episode); and deepest ('D': Select the leaf which results in the longest trajectory). As shown, all construction heuristics resulted in roughly the same performance (with MLPG performing only slightly better). This suggests that the agent learns to reduce the trajectory length regardless of the path chosen by the construction heuristic. Since the specific path chosen is independent of node selection, we posit that the relative strength of an RL agent trained with retro branching will also be independent of the node selection policy used.

To test this, we took our best retro branching agent trained with the default SCIP node selection heuristic and tested it on the 500×1000 validation instances in the default, DFS, and BFS SCIP node selection settings. To make the performances of the brancher comparable across these settings, we normalised the mean tree sizes with those of PB (a branching heuristic independent of the node selector) to get the performance relative to PB in each environment. As shown in Fig. 4.4b, our agent achieved consistent relative performance regardless of the node selection policy used, indicating its indifference to the node selector.

4.7 Conclusions, Limitations, & Further Work

We have introduced retro branching; a retrospective approach to constructing B&B trajectories in order to aid learning-to-branch with RL. We posited that retrospective trajectories address the challenges of long episodes, large state-action spaces, and partially observable future states which otherwise make branching an acutely difficult task for RL. We empirically demonstrated that retro branching outperforms the current SOTA RL method by 3-5$\times$ and comes within 20% of the performance of IL whilst matching or beating it on 42% of test instances. Moreover, we showed that RL can surpass the performance of IL on small instances, exemplifying a key advantage of RL in being able to discover novel performance-maximising policies for expressivity-constrained networks without the need for pre-training or expert examples. However, retro branching was not able to exceed the IL agent at scale. In this section we outline the limitations of this chapter and areas of further work.

Partial observability. A limitation of our proposed approach is the remaining partial observability of the MDP, with activity external to the current sub-tree and branching decision influencing future bounds, states, and rewards. In this and other studies, variable and node selection have been considered

in isolation. An interesting approach would be to combine node and variable selection, giving the agent full control over how the B&B tree is evolved.

Reward function. The proposed trajectory reconstruction approach can facilitate a simple RL reward function which would otherwise fail were the original 'full' tree episode used. However, assigning a -1 reward at each step in a given trajectory ignores the fact that certain actions, particularly early on in the B&B process, can have significant influence over the length of multiple trajectories. This could be accounted for in the reward signal, perhaps by using a retrospective backpropagation method (similar to value backpropagation in Monte Carlo tree search [Silver et al., 2016, 2017]).

Exploration. The large state-action space and the complexity of making thousands of sequential decisions which together influence final performance in complex ways makes exploration in B&B an acute challenge for RL. One reason for RL struggling to close the 20% performance gap with IL at scale could be that, at some point, stochastic action sampling to explore new policies is highly unlikely to find trajectories with improved performance. As such, more sophisticated exploration strategies could be promising, such as novel experience intrinsic reward signals [Burda et al., 2018, Zhang et al., 2021b], reverse backtracking through the episode to improve trajectory quality [Salimans and Chen, 2018, Agostinelli et al., 2019b, Ecoffet et al., 2021], and avoiding local optima using auxiliary distance-to-goal rewards [Trott et al., 2019] or evolutionary strategies [Conti et al., 2018].

Chapter 5

Partitioning Distributed Compute Jobs

Abstract

From natural language processing to genome sequencing, large-scale machine learning models are bringing advances to a broad range of fields. Many of these models are too large to be trained on a single machine, and instead must be distributed across multiple devices. This has motivated the research of new compute and network systems capable of handling such tasks. In particular, recent work has focused on developing management schemes which decide *how* to allocate distributed resources such that some overall objective, such as minimising the job completion time (JCT), is optimised. However, such studies omit explicit consideration of *how much* a job should be distributed, usually assuming that maximum distribution is desirable. In this work, we show that maximum parallelisation is sub-optimal in relation to user-critical metrics such as throughput and blocking rate. To address this, we propose PAC-ML (**p**artitioning for **a**synchronous **c**omputing with **m**achine **l**earning). PAC-ML leverages a graph neural network and reinforcement learning to learn how much to partition computation graphs such that the number of jobs which meet arbitrary user-defined JCT requirements is maximised. In experiments with five real deep learning computation graphs on a recently proposed optical architecture

across four user-defined JCT requirement distributions, we demonstrate PAC-ML achieving up to 56.2% lower blocking rates in dynamic job arrival settings than the canonical maximum parallelisation strategy used by most prior works.

Publications related to this work (contributions indentend):

- **Christopher W. F. Parsonson**, Zacharaya Shabka, Alessandro Ottino, and Georgios Zervas, 'Partitioning Distributed Compute Jobs with Reinforcement Learning and Graph Neural Networks', *arXiv*, 2023

  – Algorithm, code, experiments, paper writing, plots

5.1 Introduction

The last decade has seen an exponential increase in the amount of compute demanded by big data jobs such as AI and genome processing, with resource requirements doubling every 3.4 months since 2012; 50× faster than Moore's Law [OpenAI, 2018]. This trend is showing no sign of slowing down. The fundamental relationship between neural network accuracy and scale [Kaplan et al., 2020] provides a strong incentive for practitioners seeking performance improvement to further increase their resource requirements. Moreover, brain-scale AI will require at least as many parameters as the ≈1 000 trillion synapses present in the human brain [Furber, 2016]; several orders of magnitude more than the largest models used today.

The compute time and memory requirements of state-of-the-art big data applications already far outstrip the capabilities of any single hardware device. For example, one of the current largest DNNs, Megatron-Turing natural language generation (MT-NLG) [Smith et al., 2022], contains 530 billion parameters. These parameters alone occupy ≈1 000 GB, exceeding the capacity of the largest A100 GPU by over an order of magnitude, and the parameter loss gradients tracked during training occupy several times more. Even if the model could be fitted onto a single device, the training time would be ≈900 years[1]. To address these compute time and memory demands, rather than using a single device, big data jobs must be distributed and parallelised across a cluster of machines. For example, the Selene supercomputing cluster [NVIDIA, 2020] consists of 358 400 A100 GPU tensor cores, bringing the MT-NLG training time from 900 years down to the order of days[2].

However, parallelising jobs across ever-more machines brings its own challenges. With any parallelisation strategy, at some point the output of each

[1] Assuming it takes 8 V100 GPUs 36 years to train a 175 billion parameter model [NVIDIA, 2022] and extrapolating.

[2] Assuming a linear parallelisation speedup and 0 communication overhead.

'worker' (a single device processing at least part of a job) must be collected and synchronised to get the overall result of the parallelised computation. This synchronisation requires *communication* between the workers. As discussed in Chapter 1, as the number of workers used to execute a job is increased, the per-worker computation demands decrease, but the overall communication overhead between workers grows (see Fig. 1.2b). This shifts the performance bottleneck away from the workers themselves and into the network connecting them, and brings additional challenges with managing varying traffic characteristics for different job types and parallelisation strategies [Wang et al., 2022, Parsonson et al., 2022, Benjamin et al., 2021, 2022].

To address the communication bottleneck in distributed computing, recent works have sought to develop optical clusters [Benjamin et al., 2020, Ballani et al., 2020, Khani et al., 2021, Wang et al., 2022, Ottino et al., 2022]; machines interconnected by optical switches [Parsonson et al., 2020, Gerard et al., 2020b, 2021]. Compared to their electronic counterparts, optically switched networks offer orders of magnitude improvements in scalability, bandwidth, latency, and power consumption [Ballani et al., 2020, Zervas et al., 2018, Mishra et al., 2021] (see Section 5.2).

Optical clusters are typically operated under the OCS paradigm due to its non-blocking circuit configurations with high capacity and scalability [Raja et al., 2021]. OCS networks are fundamentally different from the electronic packet switched architectures used by most current clusters, resulting in entirely new communication patterns and resource demand characteristics. Consequently, new compute and network resource management schemes are needed in order to optimally allocate jobs and maximise performance.

Of the many resource management tasks which must be performed in a compute cluster, *job partitioning* (how to split a job up across how many devices) is key to overall performance. More partitioning can lead to lower compute times. However, it may also increase network overhead and occupancy of

cluster resources, possibly leading to future jobs being blocked upon arrival and consequently lower overall cluster throughput. Prior works such as SiP-ML [Khani et al., 2021] have introduced simple partitioning heuristics for optical networks which have notably improved cluster performance. However, they have not been designed under the more realistic setting of dynamic and stochastic job arrivals, have not considered the state of the cluster in a 'network-aware' manner when making partitioning decisions, and have been crafted to optimise for the sub-optimal objective of minimising JCT.

In this work, we first argue that simply minimising the JCT is a naive objective because it brazenly encourages more parallelisation of a job request without considering the effect this has on the ability of a cluster to service subsequent jobs. We then introduce a new more subtle formulation of the optimisation metric, the *user-defined blocking rate*, which more aptly encompasses the desires of cluster users. Next, we propose a simple modification of the quantised SiP-ML partitioner which, rather than maximally parallelising all jobs, minimally parallelises them such that they meet the user-defined maximum acceptable completion time. Then, we propose a novel network-aware partitioning strategy (see Fig. 5.4 and Section 5.5) called PAC-ML (**p**artitioning for **a**synchronous **c**omputing with **m**achine **l**earning) which utilises RL and a GNN to flexibly meet the demands of the user in an arbitrary manner given the current state of the cluster network. Finally, we demonstrate our method in simulation on the recently propsed RAMP optical architecture [Ottino et al., 2022], achieving up to 56.2% lower blocking rates than the best heuristic baseline. We show that different user-defined demand environments require different partitioning strategies for optimal results, and that a key advantage of PAC-ML is that it is able to discover performant strategies *automatically* without the need for handcrafted heuristics or environment-specific tuning.

5.2 Background

5.2.1 Parallelisation

Types of parallelism. Parallelisation is the process of distributing a computational job across multiple devices. This is done in order to reduce the time and/or physical memory needed to complete the job. There are three main types of deep learning parallelism; *data parallelism*, *model parallelism*, and *hybrid parallelism* (see below). Although today the most common method for DNN training parallelisation is data parallelism for its simplicity and limited network overhead, we focus on the less common but more desirable model parallelism paradigm for its strong scaling capabilities [Khani et al., 2021]. Our proposed partitioning methods are applicable to hybrid and pipeline parallelism, but these require additional simulation complexity and are therefore beyond the scope of this chapter.

Data parallelism. Data parallelism [Slotnick et al., 1962] is where an identical copy of the DNN model is sent to each worker. The input training data is parallelised by sampling a training batch, splitting it into non-overlapping micro-batches, training each worker on its own micro-batch, and updating the workers' local model parameters using some method to synchronise the gradients of the parameters with respect to the training loss after each training iteration. This synchronisation step is commonly referred to as *AllReduce*, and can be performed using various techniques. Data parallelism can be applied to any DNN model regardless of its architecture, enables the use of large data sets (which are crucial for scaling model performance [Hoffmann et al., 2022]), and facilitates the use of large training batch sizes which can lead to smoother and faster convergence. This is a form of *weak scaling*, where the JCT is decreased by reducing the total number of training iterations needed via increasing the amount of data processed per iteration as the number of workers is increased [Khani et

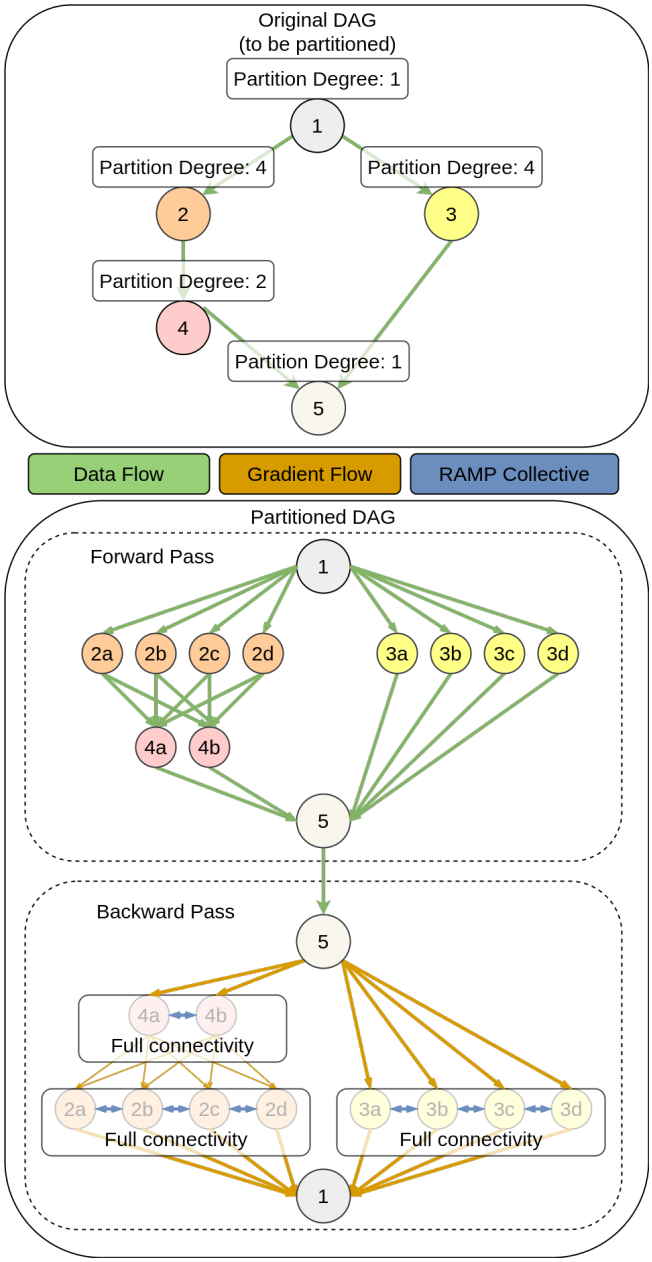


FIGURE 5.1: Diagram showing a DNN job DAG being partitioned. Top: A forward pass DAG where each node has an associated partition degree (how many times it will be divided when partitioned). Bottom: A partitioned DAG with forward and backward passes handled consecutively. Green edges in the graph represent data flow (i.e. output to input) between consecutive operations in the forward pass. Orange edges represent gradient exchanges processed in the backward pass (backpropagation). Blue edges represent full connectivity collective operations to synchronise weight updates across partitioned components of an operation. Note that, for brevity, the top unpartitioned DAG only shows the forward pass (since, before partitioning, the graph structure is identical to the backward pass), whereas the bottom partitioned DAG shows both the forward and backwards passes (since, after partitioning, the graph structures are different).

al., 2021]. However, it scales poorly for large models with many parameters since all parameters must fit onto a single worker and then be synchronised at the end of each training step, and has the constraint that the training data must be i.i.d. in order for parameter updates to be computed and summed across workers to attain the updated model parameters.

Model parallelism. Model parallelism [Karakus et al., 2021] is where the DNN model is partitioned (split) and a part of the model is sent to each worker. In the DNN forward pass, a training batch is sampled, copied, and sent to each worker which holds layer-1 of the DNN. The layer-1 worker(s) then compute the layer-1 output(s) and forward them to the worker(s) which hold layer-2, and so on. In the backward pass, the gradients of the model parameters with respect to the training loss are computed by starting at the worker(s) which hold the final layer and propagating these gradients back to the layer-1 workers, after which the partitioned model will be globally synchronised. Layer outputs, gradients, and activations are exchanged during the training iteration using a synchronisation step commonly referred to as *AllGather*. Model parallelism facilitates the use of very large models which otherwise would not fit onto a single worker and caters for time-efficient parallelisation of computational operations where possible. This is a form of *strong scaling*, where the JCT and per-worker memory utilisation are decreased via increasingly partitioning different parts the job across more workers as the number of workers is increased [Khani et al., 2021]. However, passing gradients between workers during training can create a large communication overhead [Mirhoseini et al., 2017, 2018], and expert domain knowledge of the specific model architecture is needed to know how to split the model across multiple workers.

Hybrid parallelism. Hybrid parallelism [Dean et al., 2012] is where a combination of data and model parallelism is used to strive for the benefits of both. This can be extended to include *pipeline parallelism* [Huang et al., 2019, Narayanan et al., 2019], where intra-batch parallelism (data and model

parallelism) are combined with inter-batch parallelism (pipelining) where multiple micro-batches are processed simultaneously where possible. Hybrid parallelism can result in higher worker utilisation and the advantages of both model and data parallelism, but requires complex bidirectional pipelining across different inputs, careful model parameter versioning to ensure correct computations of the gradients during the backward pass, and each stage allocated across workers must be load-balanced to ensure roughly equivalent computational times between workers in order to maximise peak pipeline throughput.

Computational jobs. A computational job is a directed acyclic graph (DAG) whose nodes are *operations* and edges are *dependencies*. Operations are computational tasks (e.g. some mathematical reduction, a database query, etc.). Dependencies are either *control dependencies*, where the child operation can only begin once the parent operation has been completed, or *data dependencies*, where at least one tensor is output from the parent operation and required as input to the child operation. In the context of DNNs, a job DAG is a sequence of forward pass, backward pass, and parameter update operations which need to be performed on data exchanged between operations. Whether or not this data passes through a communication network is determined by how the operations are partitioned, placed across a cluster of workers, and parallelised.

Job partitioning. Job partitioning refers to the process of splitting the operations of a job DAG into u (the *partition degree*) smaller sub-operations which can in turn be placed across u workers, thus reducing their run time and memory requirements. Partitioning is used in the model, hybrid, and pipeline parallelisim paradigms. More partitioning can decrease compute time and memory requirements, but requires more inter-worker communication, complex intra-worker operation scheduling, and greater resource utilisation, therefore potentially increasing overall completion time, cluster complexity, and subsequent job blocking rates. Fig. 5.1 visualises how an initial DAG for some arbitrary neural network architecture, where each operation has a partitioning degree, can

be re-represented in terms of its partitioned form. Both forward and backward passes are explicitly represented since inter-operation information dependencies (i.e. the edges in the graph) are not the same in each pass.

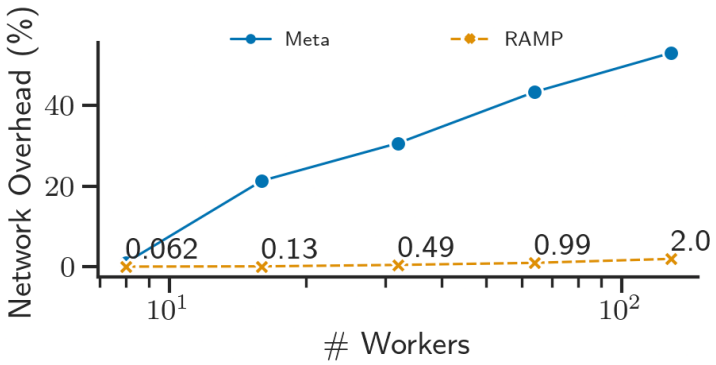


FIGURE 5.2: The mean network overhead of the 6 distributed deep learning jobs reported by [Wang et al., 2022] in Meta's GPU cluster compared to that of RAMP as reported by Ottino et al. [2022] on the 5 jobs considered in our work. Note that this is an approximate comparison, and that the important takeaway is that RAMP retains low network overheads as jobs become increasingly distributed.

5.2.2 RAMP

Overview. RAMP is a state-of-the-art OCS architecture designed specifically for cloud data centres and distributed deep learning systems [Ottino et al., 2022]. RAMP networks are parameterised by N_C communication groups, N_R racks per communication group, and N_S servers per rack, resulting in a $N_W = N_C \times N_R \times N_S$ worker cluster with a colloquially termed 'RAMP shape' defined by tuple $\langle N_C, N_R, N_S \rangle$. At its core, RAMP proposes a novel set of message passing interfaces (MPIs) for performing the synchronisation steps (AllReduce, AllGather, etc.) required by distributed DNN training jobs. These will be referred to as *collective operations*. These MPIs are designed to take full advantage of the high bandwidth provided by optical network architectures. Consequently, as shown in Fig. 5.2, the network overhead of RAMP remains remarkably low as the number of workers used to execute a job increase (see Section 5.6 for experimental details). The RAMP authors showed that this low

network overhead enables unprecedented scalability with up to 65 536 worker nodes capable of training O(trillion) parameter DNN models.

RAMP placement rules. As detailed in Ottino et al. [2022], a group of workers in a RAMP shape can only undergo collective operations if they are selected with respect to certain rules, loosely termed here 'symmetry' rules. For shape $\langle N_C, N_R, N_S \rangle$, these rules are as follows: (1) N_S workers per rack spread over N_R racks requires that the set of workers on each rack span N_R distinct communication groups. These N_R distinct communication groups do not have to be the same set across racks. (2) N_S workers on $N_R = 1$ rack must span N_S communication groups. (3) N_S workers spread over N_R racks ($N_S = 1$ worker per rack) must span N_S distinct communication groups.

In our simulations, we use a simple first-fit operation placement heuristic which conforms to these rules (refer to Appendix C.3.4 for further details).

5.3 Related Work

Recent years have seen a surge of interest in developing methods to distribute ML tasks across multiple devices [Ben-Nun and Hoefler, 2019, Mayer and Jacobsen, 2020]. One approach has been to optimise the *physical plane* of the distributed cluster such as its compute and network devices and architectures [Parsonson et al., 2020, Khani et al., 2021, Wang et al., 2022, Ottino et al., 2022]. In this work, we instead focus on optimising the *orchestration plane*, which determines how physical layer resources are allocated to execute a job. We divide the orchestration plane into three sub-components: Job (1) partitioning (*how many* devices to use); (2) placement (*which* devices to use); and (3) scheduling (*in which order* to use the devices). Many prior orchestration plane works have considered (2) and (3) (*how* to distribute), whereas we focus on (1) (*how much* to distribute). However, in this section we comment on recent progress across all these fields, since we leverage this progress throughout our work.

ML for discrete optimisation. Many CO problems turn out to be NP-hard, rendering exhaustive search techniques intractable for practical application [Bengio et al., 2021]. Consequently, practitioners rely on either approximate algorithms, which give restricted performance guarantees and poor scalability [Williamson and Shmoys, 2011], or heuristics, which have limited solution efficacy [Halim and Ismail, 2019]. Since the first application of neural networks to CO by Hopfield and Tank [1985], the last decade has seen a resurgence in ML-for-CO [Bello et al., 2016, Dai et al., 2017, Barrett et al., 2019, Gasse et al., 2019, Barrett et al., 2022, Parsonson et al., 2022]. The advantages of ML-for-CO over approximation algorithms and heuristics include handling complex problems at scale, learning either without external input and achieving super-human performance or imitating strong but computationally expensive solvers, and (after training) leveraging the fast inference time of a DNN forward pass to rapidly generate solutions. Since almost all cluster resource management tasks can be reduced to canonical CO problems [Bengio et al., 2021], many state-of-the-art resource management methods utilise recent advances in ML-for-CO.

Job placement. Mirhoseini et al. [2017] were the first to apply ML to the task of deciding which operations in a computation graph to place on which devices in a cluster. They used a sequence-to-sequence model consisting of an LSTM DNN with an attention mechanism trained with the simple REINFORCE policy gradient RL algorithm [Williams, 1992] such that the JCT of a deep learning job was minimised, outperforming handcrafted heuristics when training the Inception-V3 computer vision and LSTM natural language processing models. Gao et al. [2018] furthered this work by replacing REINFORCE with the more advanced PPO RL algorithm [Schulman et al., 2017] with lower variance and reduced training hardware demands. They demonstrated their method beating Mirhoseini et al. [2017] on the CIFAR-10 image recognition benchmark in terms of JCT. Mirhoseini et al. [2018] proposed a novel hierarchical model which decomposed the job placement task into a joint group-and-place problem,

reducing the JCT of Inception-V3, ResNet, LSTM, and NMT models by up to 60% relative to the state-of-the-art.

All works up to this point used DNN architectures restricted to Euclidean-structured input data. Consequently, in order to handle non-Euclidean graph-structured data such as computation graphs and cluster networks, they had to be re-trained each time a new graph structure was considered. Addanki et al. [2019] were the first to instead leverage a GNN, as well as the grouping scheme of Mirhoseini et al. [2018], to learn to generalise across different job types with varying computation graph structures, demonstrating device placement schemes which were on par with or better than prior approaches on Inception-V4, NASNet, and NMT after 6.1× fewer training steps. Khadka et al. [2021] furthered the use of GNNs for job placement by combining GNNs, RL, and population-based evolutionary search with the hierarchical group-and-place scheme of Mirhoseini et al. [2018]. Concretely, they replaced the manually-designed operation grouping heuristic with a learned policy capable of superior scaling and JCT performance.

Job scheduling. Bao et al. [2018] addressed the job scheduling problem (the order in which to execute operations placed across a set of devices) using a primal-dual framework for online job scheduling. They represented the problem as an ILP which their proposed algorithm could solve in polynomial time in an online fashion such that the cluster resources were maximally utilised and the JCT minimised. Li et al. [2021] proposed a placement-aware scheme which leveraged the pre-determined device placement allocation to decide on a job schedule which could reduce the average JCT by up to 25% relative to other scheduling methods. Paliwal et al. [2020] went further by utilising an RL-trained GNN and a genetic algorithm to jointly optimise both job placement and scheduling, demonstrating both lower JCT and peak memory usage than other strategies when distributing TensorFlow computation graphs across a cluster.

Job partitioning. To the best of our knowledge, Khani et al. [2021] are the only ones to have explicitly considered the question of *how much* to distribute a

computation graph in the context of an optical network. Like other works, they assumed that a maximum parallelisation strategy (i.e. partition the job across as many workers as possible) is a desirable objective, and then focused on how best to design the physical layer such that the JCT could be minimised.

All works discussed in this section have assumed that the JCT is the key objective to minimise. Consequently, where the question of partitioning is considered, prior works have assumed that more parallelisation is desirable. However, we posit that user-critical metrics such as throughput and blocking rate are compromised by prioritising optimisation of the JCT in a cluster setting with dynamic job arrivals. To address this shortcoming, we propose a new ML-based resource management scheme which explicitly addresses the partitioning question. Concretely, our work leverages the emergent trend from these other orchestration plane fields, namely utilising an RL-trained GNN, to decide how much to partition different jobs in a dynamic setting with arbitrary user-defined completion time requirements.

5.4 User-Defined Blocking Rate

To motivate our work, we first explore the key metrics to consider when evaluating a job partitioning strategy with the help of an experiment on 32 GPU workers, and then introduce a new formulation of the *user-defined blocking rate*. All experimental details are given in Section 5.6.

The inadequacy of optimising the job completion time. As discussed in Section 5.3, most prior works researching management schemes for distributed computing aim to minimise JCT; the time taken to complete a given job. If a job j begins running at wall clock time $t^{\text{start}}_{wc,j}$ and is completed at time $t^{\text{end}}_{wc,j}$, researchers usually record the completion time as $\text{JCT}_j = t^{\text{end}}_{wc,j} - t^{\text{start}}_{wc,j}$. Consequently, most systems maximise the degree to which they parallelise jobs in order to minimise JCT. While it is true that end users undoubtedly want this JCT metric to be

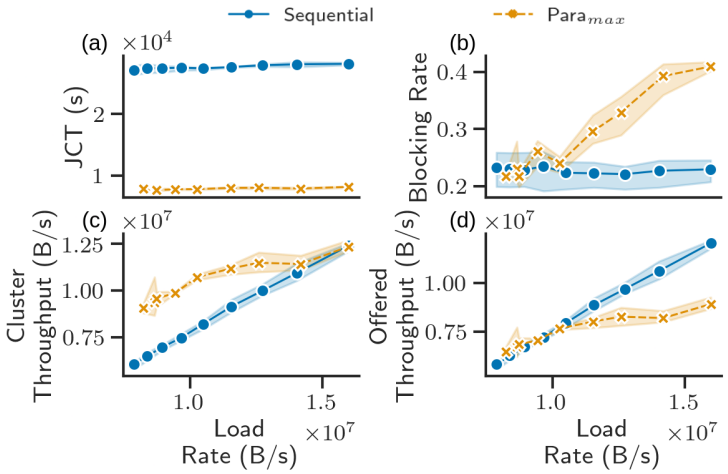


FIGURE 5.3: (a-b) Demonstration of how more partitioning can lead to a lower JCT than no partitioning (i.e. sequentially running the job on a single device), but this may be at the cost of a higher blocking rate since more cluster resources are occupied when subsequent jobs arrive. (c-d) Demonstration of how optimising for the cluster throughput leads to an unfair bias towards more partitioning, because more parallelism creates more work for the cluster and therefore artificially increases cluster throughput even though, from the perspective of the user, the original offered throughput may be lower.

minimised, it fails to quantify when a job was *blocked*, which occurs when no cluster resources were available to service it. While more parallelism will often lead to a lower JCT for a given job, it will also use up more of the cluster's compute and network resources, potentially blocking future job arrivals (see Fig. 5.3). Therefore in practice, end-users wish to minimise both the JCT and the overall *blocking rate* (the fraction of jobs blocked over a given time period). While maximum parallelisation will lead to a minimised JCT, we posit that a balance between these two extreme parallelisation strategies can more aptly optimise for both the JCT and blocking rate.

Alternative optimisation objectives. One metric which encapsulates both the JCT and blocking rate is *throughput*; the information processed per unit time. There are two issues with using throughput as an optimisation objective. (1) Operators must be careful how they measure the throughput to be optimised. If they measure the *cluster throughput* (the total *cluster information* processed per unit time), they will be biased towards more parallelisation,

because when a job is partitioned and parallelised, the edge dependencies coming in to and out of the partitioned operation node(s) must be replicated (see Fig. 5.1). This artificially creates more information for the cluster to process even though, from the end users' perspective, the total information processed of their original demand is the same. Therefore, the *offered throughput* (the total original demand information (i.e. before partitioning was applied) processed per unit time) is a more suitable throughput metric to optimise. Figure 5.3 shows an example of how a 'maximum partitioning' strategy, such as that used by SiP-ML [Khani et al., 2021], can have superior cluster throughput when compared to a 'no partitioning' strategy (sequentially running the job on a single device) despite having lower offered throughput. However, offered throughput is still an inadequate optimisation metric, because (2) in practice, different jobs being serviced by the cluster originating from different client users have different priorities and job completion time requirements. For example, two identical machine learning training jobs might be submitted to the cluster, but one from a user who intends to deploy the model commercially and requires it to be completed overnight, and the other from a user who is employing the model for research and has less stringent completion time requirements. Ideally, operators would direct their clusters to meet flexible user-defined per-job completion time requirements.

The user-defined blocking rate. To enable users to dynamically determine the completion time on a per-job basis whilst also maximising the number of job demands satisfied, we introduce a new formulation of the *user-defined blocking rate* objective for the partitioning algorithm to optimise. Given a job which, if executed sequentially on one device, would be completed in $\text{JCT}_j^{\text{seq}}$, we define the *maximum acceptable JCT* as $\text{JCT}_j^{\text{acc}} = \beta \cdot \text{JCT}_j^{\text{seq}}$, where $\{\beta \in \mathbb{R} : 0 < \beta \leq 1\}$. Here, β is a parameter chosen by the user which determines how quickly the job must be completed. If $\text{JCT}_j > \beta \cdot \text{JCT}_j^{\text{seq}}$, then the cluster will have failed to complete the job within the required time and the job will be recorded as

having been blocked. The user-defined blocking rate is therefore the fraction of jobs which failed to meet the $\text{JCT}_j \leq \beta \cdot \text{JCT}_j^{\text{seq}}$ requirement over a given period of time. Note that rather than brazenly optimising for either the JCT or the blocking rate, the user-defined blocking rate enables the cluster operator to instead dynamically specify their desired completion time on a per-job basis, and the performance of the cluster is evaluated according to how well it was able to meet the requirements of the user. Furthermore, the β parameter corresponds to the speed-up factor being requested by the user and, since $\{\beta \in \mathbb{R} : 0 < \beta \leq 1\}$, can be given directly as input to a DNN.

5.5 PAC-ML Partitioning Methodology

RL agents can learn general policies without the need for human guidance. An RL job partitioner therefore has the potential to take an arbitrary maximum acceptable JCT provided by the user and *automatically* decide how much to distribute the job such that, over a period of time, the number of jobs which meet the JCT requirements specified by the user is maximised. Such an agent would therefore be able to minimise the blocking rate whilst also accounting for the flexible and dynamic JCT specifications of the user. Following this logic, we now describe our PAC-ML (**p**artitioning for **a**synchronous **c**omputing with **m**achine **l**earning) approach for learning to partition computation jobs with RL and a GNN.

5.5.1 Markov Decision Process Formulation

Since allocating cluster resources for jobs arriving dynamically in time is a sequential decision making process, formulating problems such as job partitioning as an MDP is a natural approach and facilitates the application of many traditional and state-of-the-art RL algorithms [Mao et al., 2016, Addanki et al., 2019, Paliwal et al., 2020].

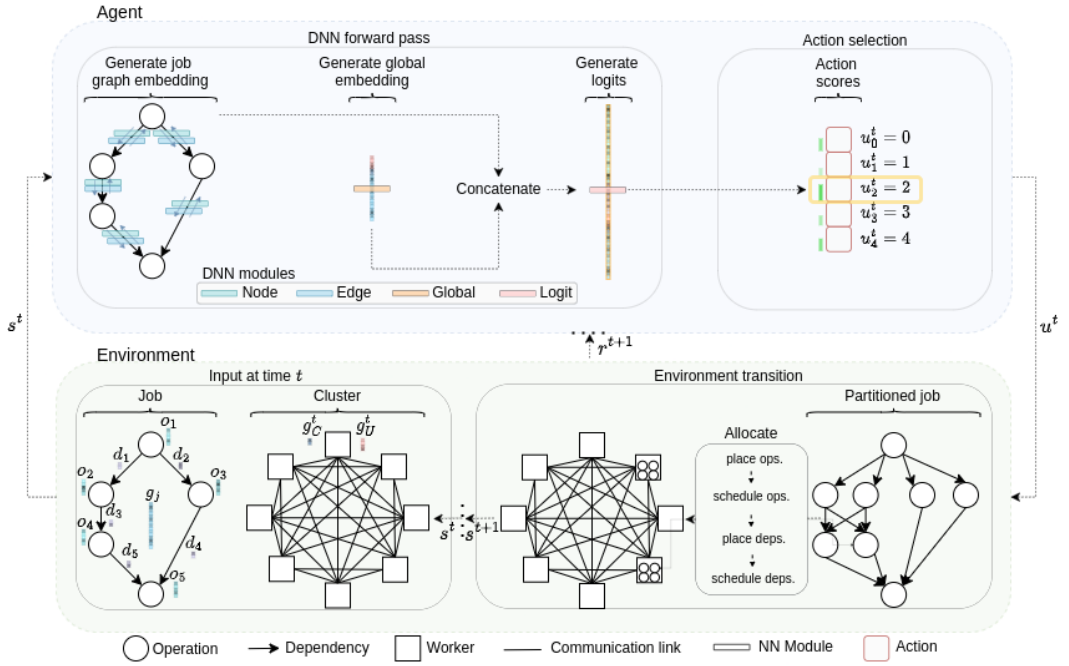


FIGURE 5.4: An overview of our PAC-ML approach transitioning from step $t \rightarrow t+1$. At each time step t when there is a new job to be placed on the cluster, we: (i) Use a GNN to generate an embedded representation of the node and edge features in the job's computation graph, and a standard feedforward DNN to do the same for the global job and cluster features; (ii) concatenate the outputs of (i) and use another feedforward DNN to generate a logit for each action $u^t \in U^t$; (iii) pass the chosen action u^t to the environment and partition the job accordingly; (iv) apply any internal environment allocation heuristics (operation and dependency placement and scheduling, etc.) to attempt to host the job on the cluster; (v) if accepted onto the cluster, perform a lookahead to evaluate the job's completion time; (vi) fast-forward the environment's wall clock time t_{wc} to when the next job arrives, and return the corresponding reward r^{t+1} and updated state s^{t+1}.

States. A new job j arriving at time step t is comprised of a DAG $G(O, D, g_j)$ with node operations O, edge dependencies D, and any other job statistics which might be recorded g_j. Similarly, the state of the cluster at time t is made up of the number of workers available, the jobs currently running on the cluster, and so on. To compress the state of the cluster and the job requesting to be placed into a representation suitable as input for a neural network at time step t, we encode this information into five feature vectors:

1. **Per-operation features** $o_i \forall i \in \{1, ..., |O|\}$ (5 features): (i) The compute cost (run time in seconds on an A100 GPU); (ii) a binary variable indicating whether the operation has the greatest compute cost in the job; (iii) the

memory cost (byte occupancy); (iv) a binary variable indicating whether the operation has the greatest memory cost in the job; and (v) the node depth with respect to the source node. The compute and memory costs are normalised by the highest compute and memory cost operations in the job, and the node depth is normalised by the depth of the deepest node.

2. **Per-dependency features** $d_i \forall i \in \{1, ..., |D|\}$ (2 features): (i) The size (in bytes) of the edge dependency normalised by the largest dependency in the job; and (ii) a binary indicator of whether the dependency is the largest in the job.

3. **Global job features** g_j (15 features): (i) The number of operations; (ii) the number of dependencies; (iii) the sequential job completion time; (iv) the maximum acceptable job completion time; the maximum acceptable job completion time fraction β both (v) raw and (vi) normalised; (vii) the total memory cost of all of the operations; (viii) the total size of all of the dependencies; (ix) the number of training steps which need to be performed; the (x) mean and (xi) median of the operation compute costs; the (xii) mean and (xiii) median of the operation memory costs; and (xiv) the mean and (xv) median of the dependency sizes. Each feature is normalised by the highest respective value of the feature across all job types.

4. **Global cluster features** g_C^t (2 features): (i) The number of occupied workers; and (ii) the number of jobs running. Both features are normalised by the total number of workers in the cluster N_W.

5. **Global action features** g_U^t ($\frac{N_W}{2}$ features): A binary vector indicating the validity of each possible partitioning decision given the state of the cluster and the RAMP rules defined by [Ottino et al., 2022].

Actions. Given the state s^t encapsulating both the job requesting to be placed and the current state of the cluster, the partitioning agent uses a policy $\pi(s^t)$ to select a number of times u^t up to which to partition each operation in the job's computation graph (using a similar minimum operation run time quantum discretisation scheme to Khani et al. [2021]), where $u_i^t \forall i \in \{0, 1, ..., \frac{N_W}{2}\}$ (i.e. there are $\left(\frac{N_W}{2} + 1\right)$ possible discrete actions). Note that $u^t = 0$ enables the agent to reject a job without placing it, $u^t = 1$ places the job onto one worker and runs it sequentially, and $1 < u^t \leq \frac{N_W}{2}$ attempts to distribute the job's operations across up to u^t workers. In our setting and given the RAMP rules of Ottino et al. [2022], an invalid partitioning action is one which is at least one of: (i) An odd number (except $u^t = 1$), or either (ii) greater than the number of workers available or (iii) has no valid RAMP placement shape given the current state of the cluster (see Section 5.2).

Rewards. As a consequence of the RAMP rules defined by Ottino et al. [2022], which require that the worker and network resources allocated to a given job are reserved exclusively for that job for the duration of its run time, we are able to perform a deterministic lookahead to evaluate what the overall completion time, JCT_j, of the job will be as soon as it is placed. Subsequently, when a job j arrives at time step t, we can immediately determine whether or not the cluster met the $\text{JCT}_j^{\text{acc}}$ specified by the user. This enables the use of a simple per-step $+1/-1$ reward scheme,

$$r^{t+1} = \begin{cases} 1, & \text{if } \text{JCT}_j \leq \beta \cdot \text{JCT}_j^{\text{seq}} \\ -1, & \text{otherwise} \end{cases}, \tag{5.1}$$

which when aggregated and maximised over the course of an episode corresponds to maximally meeting the specified per-job completion time requirements and therefore minimising the user-defined blocking rate.

Transitions. In our hybrid time- and event-drive simulation, when the agent

makes a partitioning decision at time step t, the environment transitions to the next step $t+1$ by fast-forwarding its internal simulated wall clock time, t_{wc}, to when the next job arrives and requests to be placed, updating the states of any running and completed jobs and their corresponding compute and network resources as necessary. The episode terminates when $t_{\text{wc}} = T_{\text{wc}}^{\text{max}}$.

5.5.2 PAC-ML Learning Setup

Reinforcement learning algorithm. To find a policy which maximises the expected return when partitioning jobs, we used the state-of-the-art Ape-X DQN [Horgan et al., 2018] RL algorithm; a distributed and highly scalable value-based method (see Appendix C.6 for algorithm details and hyperparameters).

Neural network architecture. To make the learning of value and policy functions tractable in large state-action spaces, we approximated them with a custom-built message passing GNN implemented using the open-source PyTorch [Paszke et al., 2019] and DGL [Wang et al., 2019] libraries. Refer to Appendix C.5 for further architectural details.

5.6 Experimental Setup

All code for reproducing the experiments and links to the generated data sets are provided at `https://github.com/cwfparsonson/ddls`.

Simulation environment. We built an open-source Gym environment [Brockman et al., 2016] to simulate the RAMP OCS system of Ottino et al. [2022] in an RL-compatible manner. We used a hybrid time- and event- driven simulation approach where we kept track of the internal simulation wall clock time t_{wc}, enabling the measurement of time-based metrics, but only took a partitioning decision when needed (i.e. when a new job demand arrived at the cluster), aiding efficiency since no discrete steps were needlessly simulated. All our experiments used similar cluster parameters to Ottino et al. [2022]. We used

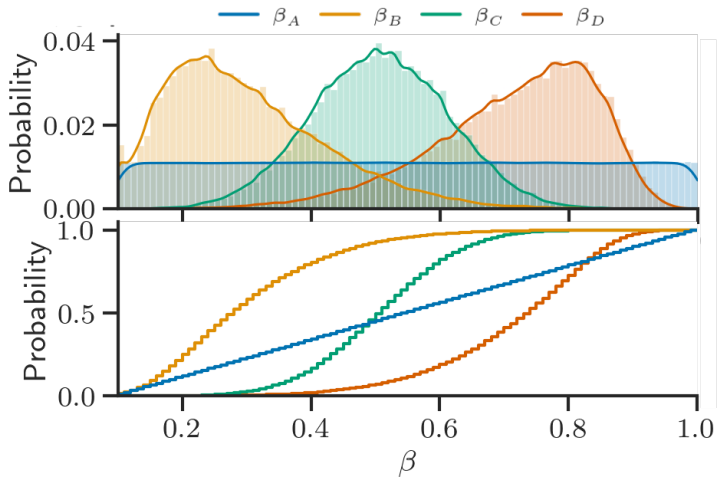


FIGURE 5.5: The four β distributions used in our experiments in order to measure the capability of each partitioner to cater to different user-defined maximum acceptable completion time requirement settings. In each β_X experiment setting, each new job generated was assigned a β value sampled from β_X in order to get the maximum acceptable job completion time, $\beta \cdot \text{JCT}^{\text{seq}}$ (see Section 5.4).

$N_W = 32$ ($N_C = 4, N_R = 4, N_S = 2$) A100 GPUs with 80 GB memory capacity, 2 THz memory frequency, and a peak computational power of 130 Tflop/s. We assumed an intra-GPU propagation latency of 50 ns, a negligible OCS circuit reconfiguration latency of 1 ns, a worker input-output latency of 100 ns, and a total worker communication capacity of 1.6 TB/s (resulting in a per-transceiver bandwidth of $\frac{1.6\times10^{12}}{N_C}$ B/s). All experiments were run up to a simulated wall clock time of $T_{\text{wc}}^{\max} = 10^6$ s (around 12 days) of continuous cluster operation with dynamic job arrivals and were repeated across 3 random seeds, with the subsequent min-max confidence intervals for each measurement metric reported. More details of the simulation environment are provided in Appendix C.3.

Compute jobs. We used the computation graph time and memory profiles of five real deep learning job types open-accessed with Microsoft's PipeDream research [Narayanan et al., 2019, 2021] (see Appendix C.4 for details). These jobs encompassed image classification (AlexNet [Krizhevsky et al., 2012], ResNet-18 [He et al., 2016], SqueezeNet-10 [Iandola et al., 2016], and VGG-16 [Simonyan and Zisserman, 2014]) and natural language processing (GNMT [Wu et al., 2016]) tasks, thereby testing the generality of the approaches we considered.

All jobs arrived to the cluster dynamically and stochastically throughout the simulation period, with the inter-arrival time fixed at 1000 s to control the load rate. Each job was ran for $N_{iter} = 50$ training iterations, where one training iteration consists of one forward and backward pass through the neural network.

Partitioning. When partitioning the operations in a job's computation graph, we allowed the partitioning agents to split each operation up to $\frac{N_W}{2}$ times (the environment's 'maximum partitioning degree'). We followed Khani et al. [2021] by (1) assuming a linear dependency between the total number of operation splits and each split's compute time; and (2) choosing a minimum quantum of computation time, τ, and splitting operations up to a number of times which would result in sub-operations with a compute time no smaller than τ in order to maximise GPU utilisation. We set $\tau = 10$ ms. As such, a given partitioning action u^t set the maximum partitioning degree of the job, but individual operations within the job could be split fewer times depending on their initial compute time and τ. Note that although this restricts each operation to be distributed across a maximum of u^t servers, the total number of workers used by all operations in the job can still be greater than u^t depending on the operation placement heuristic's choices.

Maximum acceptable job completion times. In our setting, a partitioner would ideally be able to take an arbitrary job with an arbitrary maximum acceptable job completion time, $\beta \cdot \text{JCT}^{\text{seq}}$, and partition the job such that the completion time requirement is satisfied for as many dynamically arriving jobs as possible (thereby minimising the user-defined blocking rate; see Section 5.4). To test each partitioner's ability to do this, we ran experiments using four β distributions ($\beta_A, \beta_B, \beta_C,$ and β_D; see Fig. 5.5). For each β_X experiment, when one of the five possible jobs was randomly generated to arrive at the cluster, a β value, discretised to two decimal places, was randomly sampled from the experiment's β_X distribution and assigned to the job. By sampling a broad range of β values from a selection of β_X distributions, we ensured that

we could analyse the performance of each partitioning agent under different completion time requirement settings and subsequently measure the capability of each method to cater for different user-defined requirements.

	Heuristics			RL
	Random	Para$_{max}$	Para$_{min}$	PAC-ML
β_A	$0.517^{+0.015}_{-0.015}$	$0.262^{+0.002}_{-0.003}$	$0.309^{+0.014}_{-0.015}$	$\mathbf{0.203^{+0.007}_{-0.009}}$
β_B	$0.601^{+0.007}_{-0.008}$	$0.263^{+0.006}_{-0.004}$	$0.396^{+0.006}_{-0.003}$	$\mathbf{0.258^{+0.007}_{-0.003}}$
β_C	$0.505^{+0.016}_{-0.012}$	$0.267^{+0.004}_{-0.006}$	$0.307^{+0.015}_{-0.012}$	$\mathbf{0.117^{+0.003}_{-0.003}}$
β_D	$0.465^{+0.004}_{-0.006}$	$0.263^{+0.006}_{-0.004}$	$0.142^{+0.027}_{-0.046}$	$\mathbf{0.099^{+0.008}_{-0.007}}$

TABLE 5.1: Blocking rate performance of the partitioning agents on the four β distributions (best in **bold**). Results are given as the mean across 3 seeds, and error bars denote the corresponding min-max confidence intervals.

Partitioner baselines. We considered three heuristic baseline partitioning strategies. (1) Most prior works partition a given job across as many workers as are available up to a pre-defined environment maximum partition degree [Khani et al., 2021, Wang et al., 2022]. We refer to this strategy as 'Para$_{max}$'. (2) Given the low network overhead (see Fig. 5.2) and contentionless nature of RAMP, and given the operations' linear split-compute time dependency of our environment, a reasonable estimate for the completion time of a job with sequential run time JCT$^{\text{seq}}$ distributed across u^t workers would be JCT $\approx \frac{\text{JCT}^{\text{seq}}}{u^t}$. Therefore, in light of our objective to minimise the user-defined blocking rate, we introduce a new partitioning strategy, 'Para$_{min}$', which partitions the job up to the estimated minimum amount of parallelisation needed to satisfy the job's completion time requirements, $u^t = \lceil \frac{1}{\beta} \rceil$ (i.e. the estimated speed-up factor needed). (3) For completeness, we also ran a 'Random' partitioning baseline, which selects a partitioning degree randomly from amongst the number of available workers.

Metrics recorded. To measure the performance of our partitioning agents, we recorded the following key metrics. (1) *User-defined blocking rate* (which we abbreviate to 'blocking rate'): The fraction of arrived jobs which had their completion time requirements met by the cluster. (2) *Offered throughput*: The

total 'information size' of the original jobs (i.e. before partitioning was applied) processed per unit time. Since the open-access PipeDream job profiles used in our experiments did not contain per-operation flop/s (computational load) information, we summed the jobs' operation and dependency sizes (measured in bytes (B)) to get the total 'information size' of each job. The *load rate* could then be defined as the rate of job information arriving at the cluster per unit time, and the corresponding offered throughput as the rate at which this total job information was processed by the cluster. For a full list of metric definitions, refer to Appendix C.1.

5.7 Results & Discussion

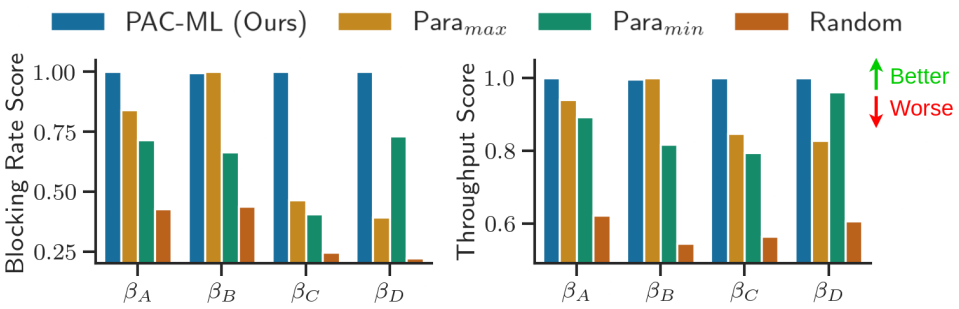


FIGURE 5.6: Validation performances (higher is better) of each partitioning agent evaluated across three seeds normalised with respect to the best-performing partitioner in each B_X environment.

5.7.1 Performance of the PAC-ML Partitioner

Comparison to the baseline partitioners. To test the performance of each partitioning agent under different completion time requirement settings, we ran our experiments across four different β distributions (see Section 5.6). We visualise the relative blocking rate and throughput performance differences between the agents in Fig. 5.6, where an agent's 'score' is its normalised performance relative to the best-performing agent with respect to a given

metric. We evaluate these scores as $\text{score}_{\text{blocking}} = \left(\frac{\text{best\_blocking\_rate}}{\text{blocking\_rate}}\right)$, and $\text{score}_{\text{throughput}} = \left(\frac{\text{throughput}}{\text{best\_throughput}}\right)$ for each agent (refer to Appendix C.7 for all raw metric values). As shown in Table 5.1 and Fig. 5.6, our PAC-ML agent achieved the best blocking rate across all four β distributions, beating its nearest rival by $22.5\%, 1.90\%, 56.2\%,$ and 30.3% for $\beta_{A,B,C,D}$ respectively.

Comparison amongst the baseline partitioners. Fig. 5.6 visualises the performance of the best PAC-ML agents on each of the four β distribution environments compared to the baseline heuristic performances. Interestingly, the best baseline in terms of blocking rate for $\beta_{A,B,C}$ is Para_{max}, but this switches to Para_{min} for β_D. On β_B, PAC-ML achieved roughly equivalent performance to Para_{max} by learning that, on this β demand distribution, maximum parallelisation led to the lowest blocking rates. This shows that different partitioning strategies have varying relative performances under different cluster settings. A key advantage of PAC-ML is therefore that the question of which partitioning strategy is best for a given environment need not be addressed by sub-optimal hand-crafted heuristics or environment-specific hyperparameter tuning. Instead, we have demonstrated in Table 5.1 and Fig. 5.6 that PAC-ML can *automatically* learn performant partitioning strategies in arbitrary environment settings.

5.7.2 Analysis of the PAC-ML Partitioner

Offered throughput analysis. One risk of optimising only for the blocking rate when training the PAC-ML agent is that it maximises the number of jobs accepted by prioritising small low-information jobs at the cost of a sub-optimal offered throughput; a key metric when measuring a cluster's quality of service to users. Fig. 5.6 shows that the offered throughput improves with the blocking rate, with the PAC-ML agent ultimately achieving the best throughput across all four β distributions.

Bias analysis. An important question is whether there is any bias in the

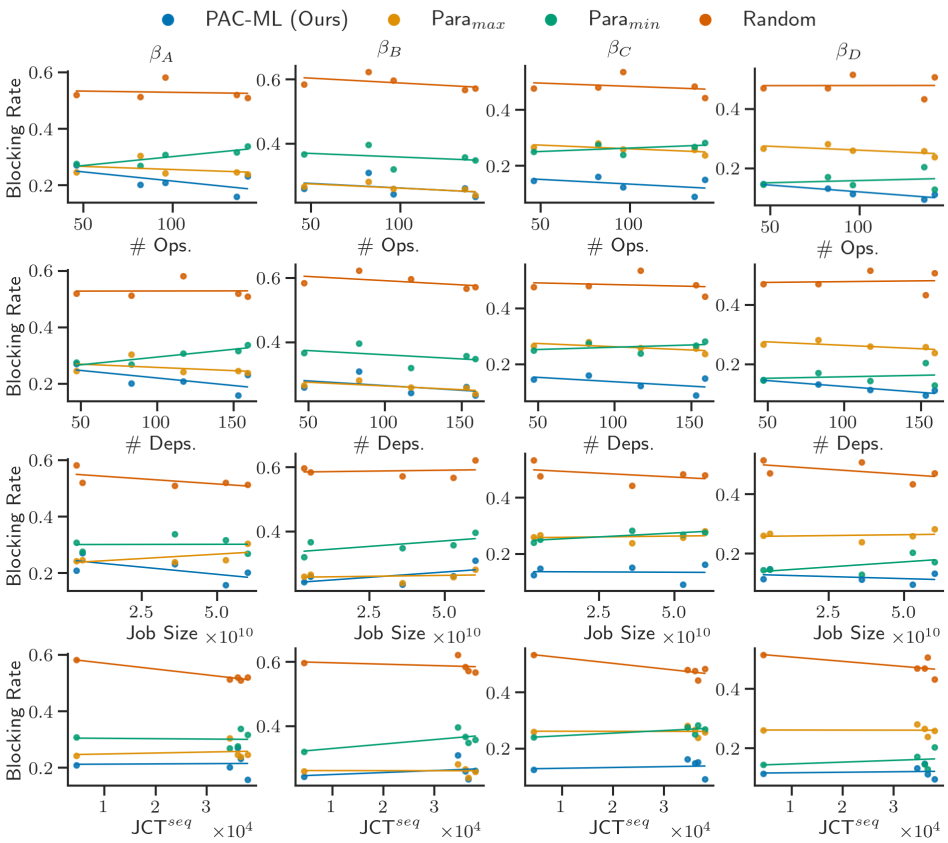


FIGURE 5.7: Mean per-job blocking rates of the five job types considered for each partitioning agent under each β_X setting plotted against the number of operations (ops.), number of dependencies (deps.), the total job information size, and the sequential run time of the job were it ran on a single device (JCT^{seq}).

kinds of jobs the PAC-ML agent learns to prioritise in order to minimise the blocking rate. To investigate this, Fig. 5.7 shows the blocking rate vs. the original characteristics for each of the five jobs considered (see Appendix C.4 for a summary of these characteristics) for each β_X distribution environment. The PAC-ML agent had little to no bias across the jobs relative to the other partitioners, with all jobs attaining approximately the same blocking rate. There was a slight bias towards the larger jobs with greater sequential completion times and more information to process, which is likely due to the fact that larger jobs occupy more resources and therefore inherently become favoured over smaller jobs.

5.8 Conclusions, Limitations, & Further Work

In conclusion, we have introduced a new partitioning strategy called PAC-ML. Leveraging RL and a GNN, PAC-ML learns to partition computation jobs dynamically arriving at a cluster of machines such that the number of jobs which meet arbitrary user-defined completion time requirements is maximised without the need for hand-crafted heuristics or environment-dependent hyperparameter tuning. We tested our partitioner on the recently proposed RAMP optical architecture [Ottino et al., 2022] across four distributions of user-defined completion time requirements, demonstrating up to 56.2% lower blocking rates relative to the canonical maximum parallelisation strategies used by most prior works when partitioning five real deep learning jobs. We hope that our work will spur a new avenue of research into developing partitioning strategies for distributed computing. In this section, we outline some limitations of the work done in this chapter and potentially interesting areas of further work.

Exceeding completion time expectations. In this work, we rewarded PAC-ML with $+1$ for completing a job within the user-defined maximum acceptable completion time and -1 for failing to do so. Although minimising the blocking rate is crucial for users, it would also be desirable to minimise the JCT as much as possible. An interesting area of further study would therefore be to incorporate this objective into the reward function, perhaps by combining the JCT speed-up factor or offered throughput with the blocking rate via multi-objective RL [Hayes et al., 2022].

Real-world experiments. Our work has considered real open-access deep learning computation graph profiles but on a simulated optical architecture. A natural but significant next step would be to implement PAC-ML in a real distributed cluster. An important question would be whether an agent trained in a simulated environment would be capable of inferring in a real cluster at test time, or if real-world training would be needed.

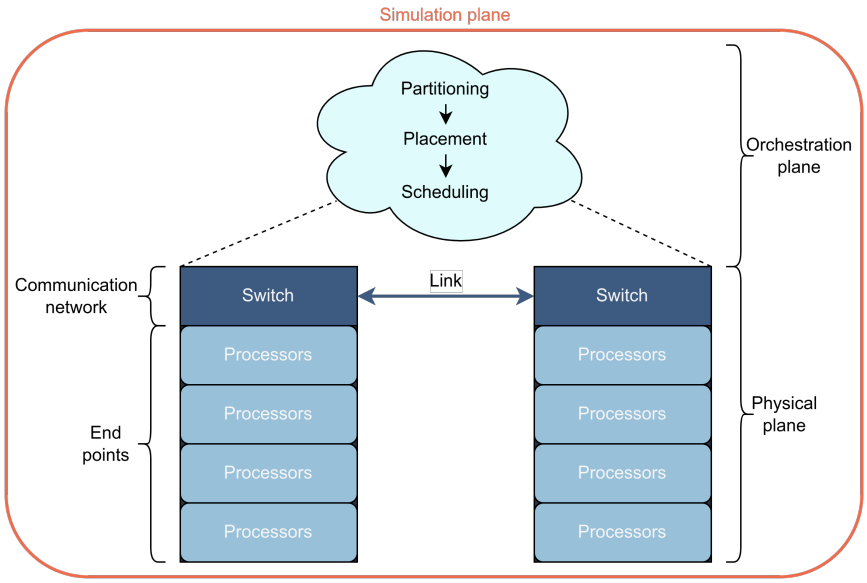
Generalisation to unseen environments. This study ran PAC-ML in an environment which had the same load rate, β distribution, cluster network size, and job computation graphs at train and test time. An interesting research question would be whether PAC-ML would be able to learn on one set (or a distribution) of these parameters and then generalise to a new set at test time, or if it would need to leverage existing or new state-of-the-art methods in GNN [Knyazev et al., 2019, Garg et al., 2020, Fan et al., 2021] and RL [Cobbe et al., 2019, Wang et al., 2020, Kirk et al., 2021] generalisation.

Robustness to stochastic inter-arrival times. In our experiments, we fixed the inter-arrival rate in order to fix the load rate. However, real clusters have variable inter-arrival times [Parsonson et al., 2022]. Handling highly stochastic environments is a known challenge for RL [Mao et al., 2019b], and therefore presents an interesting future research avenue for PAC-ML.

Combining the orchestration plane. In our work, we have considered the job partitioning task in isolation of the job placement and scheduling tasks. However, prior works have found the merging of the latter sub-tasks into a single resource management problem beneficial to performance [Paliwal et al., 2020]. An interesting area of further work would be to combine PAC-ML into a a single algorithm which handled job partitioning, placement, and scheduling via methods such hierarchical RL [Barto and Mahadevan, 2003, Vezhnevets et al., 2017, Mirhoseini et al., 2018, Paliwal et al., 2020, Zhang et al., 2021a] or multi-agent RL [Foerster, 2018].

Part III

Optimising the Simulator



Chapter 6

A Framework for Generating Custom and Reproducible Network Traffic

Abstract

Benchmarking is commonly used in research fields, such as computer architecture design and machine learning, as a powerful paradigm for rigorously assessing, comparing, and developing novel technologies. However, the data centre network (DCN) community lacks a standard open-access and reproducible traffic generation framework for benchmark workload generation. Driving factors behind this include the proprietary nature of traffic traces, the limited detail and quantity of open-access network-level data sets, the high cost of real world experimentation, and the poor reproducibility and fidelity of synthetically generated traffic. This is curtailing the community's understanding of existing systems and hindering the ability with which novel technologies, such as optical DCNs, can be developed, compared, and tested.

This chapter presents TrafPy; an open-access framework for generating both realistic and custom DCN traffic traces. TrafPy is compatible with any simulation, emulation, or experimentation environment, and can be used for

standardised benchmarking and for investigating the properties and limitations of network systems such as schedulers, switches, routers, and resource managers. We give an overview of the TrafPy traffic generation framework, and provide a brief demonstration of its efficacy through an investigation into the sensitivity of some canonical scheduling algorithms to varying traffic trace characteristics in the context of optical DCNs. TrafPy is open-sourced via GitHub [Parsonson and Zervas, 2021a] and all data associated with this manuscript via RDR [Parsonson and Zervas, 2021b].

Publications related to this work (contributions indented):

- **Christopher W. F. Parsonson**, Joshua L. Benjamin, and Georgios Zervas, 'Traffic generation for benchmarking data centre networks', *Optical Switching and Networking*, 2022

  – Algorithms, code, experiments, paper writing, plots

- Joshua L. Benjamin, Alessandro Ottino, **Christopher W. F. Parsonson**, and Georgios Zervas, 'Traffic Tolerance of Nanosecond Scheduling on Optical Circuit Switched Data Center Network', *OFC'22: Optical Fiber Communications Conference and Exhibition*, 2022

  – Code, traffic generation

- Joshua L. Benjamin, **Christopher W. F. Parsonson**, and Georgios Zervas, 'Benchmarking Packet-Granular OCS Network Scheduling for Data Center Traffic Traces', *Photonic Networks and Devices*, 2021

  – Code, traffic generation

6.1 Introduction

A benchmark is a series of experiments performed within some standard framework to measure the performance of an object. Researching data centre network (DCN) systems and objects such as networks, resource managers, and topologies involves understanding which types of mechanisms, principles or architectures are generalisable, scalable and performant when deployed in real-world environments. Benchmarking is a powerful paradigm for investigating such questions, and has proved to be a strong driving force behind innovation in a variety of fields [Weber et al., 2019]. A famous example of a successful benchmark is the ImageNet project [Deng et al., 2009], which has facilitated a range of significant discoveries in the field of deep learning over the last decade.

In order to benchmark a DCN system, a traffic trace with which to load the network is required. This presents several challenges. (1) Data related to DCNs are often considered privacy-sensitive and proprietary to the owner, therefore few DCN traffic traces are openly available. (2) When a real DCN trace is made available, it is often specific to a particular DCN and possibly not representative of current and future systems, too limited for cutting-edge data-hungry applications such as reinforcement learning, and not sufficient for stress-testing different loads in networks with arbitrary capacities to understand system limitations and vulnerabilities to future workloads. (3) Even if an attempt is made to make a real DCN available for live testing, deploying experimental systems in such large-scale production environments is often too expensive and time consuming. (4) Reducing or approximating DCN traffic down to small-scale experiments is often unfruitful since many DCN application traffic patterns only emerge at large scales.

For these reasons, most DCN researchers revert to simulating DCN traffic in order to conduct their experiments. However, synthetic DCN traffic generation is often plagued by numerous inadequacies. A common simplification approach is

to assume uniform or 'named' (Gaussian, Pareto, log-normal, etc.) distributions from which to sample DCN traffic characteristics. However, such distributions often ignore fluctuations caused by the short bursty nature of real DCN traffic, rendering the simulation unrealistically simple. Sometimes researchers will try to implement their own unique distributions to better describe real DCN traffic, however this brings difficulties with trying to reproduce and benchmark against literature reports since there is no standard framework for doing so. Another common approach is to only focus on the temporal (arrival time) dependence of DCN traffic characteristics and assume uniform spatial (server-to-server) dependencies. However, this fails to capture the spatial variations in server-to-server communication which are needed to accurately mimic real traffic. Works by Alizadeh et al. [2012, 2013] and Bai et al. [2016] introduced important DCN systems, but the traffic generators released with their papers fall short of addressing the issues of fidelity, reproducibility, and compatibility with generic network architectures (see Section 6.2).

These difficulties with simulating DCN traffic have meant that no traffic generation framework, and subsequently no universal DCN system benchmark, has emerged as the networking research field's tool-of-choice. The lack of a rigorous benchmarking framework has been a major issue in DCN literature since individual researchers have often used their own tests without adhering to the aforementioned requirements. This has limited reproducibility, stifled network object prototype benchmarking, and hindered training data supply for novel machine learning systems. Without benchmarking, it is difficult to systematically and consistently test and validate new heuristics for specific tasks such as flow scheduling. Furthermore, without sufficient training data, state-of-the-art machine learning models are less able to replace existing heuristics.

To address the lack of openly available traffic data sets, the aforementioned problems with simulation, and the absence of a system benchmark, a common

DCN traffic generation framework is needed. We introduce TrafPy: An open-source Python API for realistic and custom DCN traffic generation for any network under arbitrary loads, which can in turn be used for investigating a variety of network objects such as networks, schedulers, buffer managers, switch/route architectures, and topologies. TrafPy contributes two key novel ideas to traffic generation, which we detail throughout this chapter:

1. **Reproducibility guarantee** A novel method for providing a distribution reproducibility guarantee when generating traffic based on the Jensen-Shannon distance metric (see Section 6.3.3).

2. **Traffic generation algorithm**: A novel method for efficiently creating reproducible flow-level traffic with granular control over both spatial and temporal characteristics (see Section 6.3.5).

In addition to the above, TrafPy also contains the following features which, when combined with these novel aspects, make TrafPy a useful tool for benchmark workload generation:

- **Interactivity**: A distribution shaping tool for rapid creation of complex distributions which accurately mimic realistic workloads given only high-level characteristic descriptions (see Appendix B.3).

- **Compatibility**: Compatibility with any simulation, emulation, or experimentation environment by exporting traffic into universally compatible file formats.

- **Accessibility**: Open-source code and documentation with a low barrier to entry.

6.2 Background & Related Work

While there is limited literature on DCN traffic generation, data sets, and benchmarking for the reasons outlined in Section 6.1, there have been notable works striving towards their creation.

Real workloads. There are a collection of publicly available DCN workload traces and job computation graph data sets [Yahoo, 2015, Google, 2015, Facebook, 2014, OpenCloud, 2012, Ren et al., 2012, Eucalyptus, 2015, Pucher et al., 2015, Wolski and Brevik, 2017, Delft, 2015, Shen et al., 2015, JSSPP, 2017, Klusáček and Parák, 2017, Azure, 2017, Cortez et al., 2017, Alibaba, 2017, Lu et al., 2017, LANL and TwoSigma, 2018, Amvrosiadis, 2018, Amvrosiadis et al., 2018, NCSA, 2018, Jha et al., 2019, Jha et al., 2020]. However, almost all of these stem from Hadoop clusters and are limited to data mining applications [Pucher et al., 2015], therefore their use is primarily suited to application-specific testing and evaluation rather than as a generic tool for generating arbitrary loads and testing and designing DCN systems as TrafPy is proposed for. Additionally, many of them lack flow-level data, which is needed to accurately benchmark network systems.

Real workload characteristics. There is a limited body of work, primarily from private corporations, aiming to characterise real DCN workloads without open-accessing the underlying proprietary raw data. Benson et al. [2010a] built on work done by Kandula et al. [2009] and Benson et al. [2010b] by characterising DCN traffic into one of three categories; university, private enterprise, and commercial cloud DCNs. They identified that each of these categories serviced different applications and therefore had different traffic patterns. University DCNs serviced applications such as database backups, distributed file system hosting (e.g. email servers, web services for faculty portals, etc.), and multicast video streams. Private enterprise hosted the same applications as university DCNs but additionally serviced a significant number of custom applications and

development test beds. Commercial cloud DCNs focused more on internet-facing applications (e.g. search indexing, webmail, video, etc.), and intensive data mining and MapReduce-style jobs. They also went further than prior works by quantifying the number of hot spots and characterising the flow-level properties of DCN traffic.

The above cloud DCN studies came almost exclusively from Microsoft, who primarily service MapReduce-style applications. Roy et al. [2015] broke this homogeneous view of cloud traffic by reporting the traffic characteristics of Facebook's DCNs, thereby introducing a fourth DCN category; social media cloud DCNs. Social media cloud applications include generating responses to web requests (email, messenger, etc.), MySQL database storage and cache querying, and newsfeed assembly. This results in network traffic being more uniform and, in contrast to Microsoft's commercial cloud DCNs, having a much lower proportion (12.9%) of traffic being intra-rack.

Note that the above examples did not open-access the full data sets, but rather provided quantitative characterisations of their nature for other researchers to inform their own traffic generators.

Traffic generators. In their seminal pFabric work, Alizadeh et al. [2013] provided open-access traffic generation code which loosely replicated web search and data mining DCN workloads by following a Poisson flow inter-arrival time distribution whose arrival rate was adjusted to meet a required target load and with a mix of small and large characteristically heavy-tailed flow sizes. Additionally, the same authors [Alizadeh et al., 2012] released a simple generator which used a server application to create many-to-one flow requests from 9 servers, again following a load-adjustable Poisson arrival time distribution with 80% of flows having a size of 1 kB (a single packet) and 20% being 10 MB. As the authors noted themselves, these workloads were not intended to be realistic, but rather were designed to demonstrate clear impact comparisons between different DCN design schemes and the small latency-sensitive and large

bandwidth-sensitive flows. TrafPy, on the other hand, can facilitate the shaping of complex inter-arrival and flow size distributions with one-to-one, many-to-one, and one-to-many non-uniform server-server distributions with ease. Furthermore, TrafPy enables the generation of traffic with the same characteristics as Alizadeh et al. [2013, 2012], but for any network topology with an arbitrary number of servers and link capacities, allowing for the straightforward comparison of novel DCN fabrics with pre-established benchmark workloads.

Similarly, Bai et al. [2016] conducted an extensive experiment into the trade-off between throughput, latency, and weighted fair sharing in scenarios where each switch port had multiple queues. Alongside their study they released an open-access traffic generator which could take a configuration file as input and generate both uniform and non-uniform server-server flow requests from pre-defined discrete probability distributions. However, to produce traffic, users had to manually enter numbers into a configuration file, which made the code difficult to use. Furthermore, the generator of Bai et al. [2016] had no mechanism for ensuring distribution reproducibility when sampling from a pre-defined probability distribution; a feat achieved by TrafPy via the Jensen-Shannon distance method (see Section 6.3.3).

The key objective of TrafPy is to augment DCN research projects such as those cited above [Alizadeh et al., 2013, 2012, Bai et al., 2016]. Unlike our work, the primary focus of such projects was not on the traffic generator itself, but rather on using traffic generation as a means of benchmarking innovative ideas. We posit that the fidelity, generality, reproducibility, and compatibility of TrafPy, achieved by generating custom server-level flow traffic, would make such works easier to conduct and to compare against as baselines in future projects.

6.3 Method

6.3.1 Design Objectives

Designing successful network object benchmarks requires a flexible, modular, and reproducible traffic generation framework. The framework should enable fair comparisons between different systems whilst maintaining a rigorous experimental setting. In light of the issues highlighted in Section 6.1, the following criteria are required of such a framework:

1. *Fidelity*: Generate demands which represent realistic DCN traffic.
2. *Generality*: Generate traffic for arbitrary DCN applications and topologies.
3. *Scalability*: Efficiently scale to large networks.
4. *Reproducibility*: Reliably reproduce traffic traces to run multiple test repeats or to reproduce other researchers' traffic conditions.
5. *Repeatability*: Summarise traffic distributions such that, given just a few parameters, other researchers can repeat the demand data set for cross-validation and comparison.
6. *Replicability*: Interactively shape characteristic distributions visually to replicate realistic data given only a plot or written description (i.e. in the absence of raw data).
7. *Compatibility*: Export generated demands into universally compatible data formats such that they can be imported into any simulation, emulation, or experimentation test bed.
8. *Comparability*: Compare a set of standardised performance metrics across different studies.

6.3.2 TrafPy Overview

An overview of the TrafPy API user experience is given in Fig. 6.1 and further elaborated on throughout this chapter, with Table B.1 summarising the notation used and some API examples given in Appendix B.3. The core component of TrafPy is the *Generator*, which can be used for generating custom, literature, or standard benchmark network traffic traces. These traces can be saved in standard formats (e.g. JSON, CSV, pickle, etc.) and imported into any script or network simulator. Researchers can therefore design their systems and experiments independently of TrafPy and use their own programming languages, making TrafPy compatible with already-developed research projects and future network objects. This also means that TrafPy can be used with any simulation, emulation, or experimentation test bed. The Generator has an optional interactive visual tool for shaping and reproducing distributions, therefore little to no programming experience is required to use it to generate and save traffic data in standard formats. As the nature of DCN traffic changes, new traffic distributions can be generated with TrafPy and state-of-the-art benchmarks established.

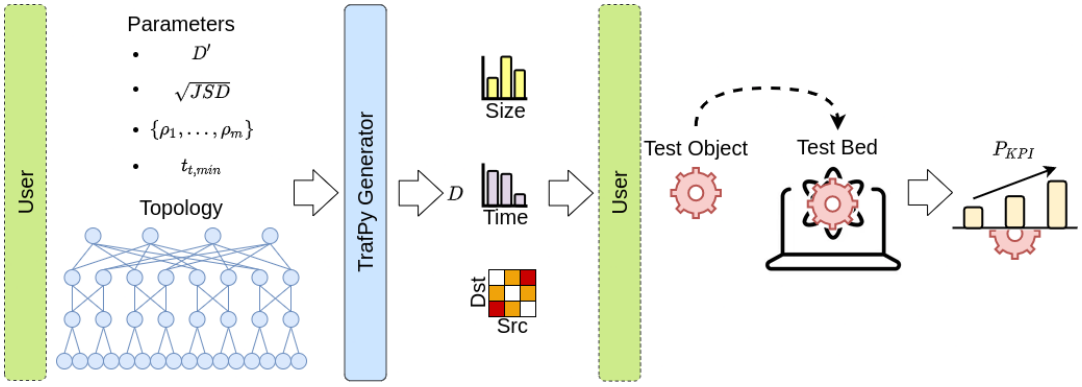


FIGURE 6.1: TrafPy API user experience for using custom or benchmark TrafPy parameters D' to make flow traffic trace D with maximum Jensen-Shannon distance threshold $\sqrt{JSD}$ and minimum flow arrival duration $t_{t,min}$ for m loads $\{\rho_1, ..., \rho_m\}$. The generated trace D can then be used to benchmark a DCN system test object (e.g. a scheduler) in a test bed (a simulation, emulation, or experimentation environment) to measure the key performance indicators P_{KPI}. The user need only use TrafPy to generate the traffic; all other tasks can be done externally to TrafPy in any programming language.

Flow traffic. The flow-centric paradigm considers a single demand as a *flow*, which is a task demanding some information be sent from a source node to a destination node in the network. Flow characteristics include *size* (how much information to send), *arrival time* (the time the flow arrives ready to be transported through the network, as derived from the network-level *inter-arrival time* which is the time between a flow's time of arrival and its predecessor's), and *source-destination node pair* (which machine the flow is queued at and where it is requesting to be sent). Together, these characteristics form a network-level *source-destination node pair distribution* ('how much' (as measured by either probability or load) each machine tends to be requested by arriving flows). When a new flow arrives at a source and requests to be sent to a destination, it can be stored in a buffer until completed (all information fully transferred) or, if the buffer is full, dropped. Once dropped or completed, the flow is not re-used.

TrafPy distributions. At the heart of TrafPy are two key notions; that no raw data should be required to produce network traffic, and that every aspect of the API should be parameterised for reproducibility. To achieve the first, rather than using clustering and autoregressive models to fit distributions to data [Li, 2010, Feitelson, 2003], TrafPy provides an interactive tool for visually shaping distributions. This way, researchers need only have either a written (e.g. 'the data followed a Pareto distribution with 90% of the values less than 1') or visual description of a traffic trace's characteristics in order to produce it. To achieve the second, all distributions are parameterised by a handful of parameters (termed D'; see Appendix B.2 for an example of the parameters used in this chapter), and a third party need only see D' in order to reproduce the original distribution. As such, TrafPy traces are discrete distributions in the form of hash tables, which can be sampled at run-time to generate flows. These tables map each possible value taken by all flow characteristics to fractional values which represent either the 'probability of occurring' for size and time distributions, or the 'fraction of the overall traffic load requested' for node

distributions. This enables traffic traces to be generated from common TrafPy benchmarks for custom network systems in a reproducible manner without needing to reformat the original data in order to make it compatible with new systems and topologies, as would be needed if the benchmarks were hard-coded request data sets instead of distributions.

6.3.3 Distribution Accuracy and Reproducibility

All TrafPy distributions are summarised by a set of parameters D'. Once D' has been established (by e.g. the community as a benchmark or a researcher as a custom stress-test or future workload trace), TrafPy must be able to reliably and accurately reproduce (via sampling) the 'original' distribution parameterised by D' each time a new set of traffic data is generated. Therefore, a guarantee that the sampled distribution will be close to the original is required to ensure reproducibility. TrafPy utilises the *Jensen-Shannon Divergence* (JSD) [Rao, 1982, Lin, 1991] to quantify how *distinguishable* discrete probability distributions are from one another. Given a set of n probability distributions $\{\mathbb{P}_1, ..., \mathbb{P}_n\}$, a corresponding set of weights $\{\pi_1, ..., \pi_n\}$ to quantify the contribution of each distribution's entropy to the overall similarity metric, and the entropy $H(\mathbb{P}_i)$ of a discrete distribution with m random variables $X_i = \{x_1^i, ..., x_m^i\}$ which occur with probability $\mathbb{P}_i = \{\mathbb{P}_i(x_1^i), ..., \mathbb{P}_i(x_m^i)\}$ where $H(X_i) = -\sum_{j=1}^{m} \mathbb{P}_i(x_j^i) \log \mathbb{P}_i(x_j^i)$, the JSD between the distributions can be calculated as in Eq. 6.1. In the context of TrafPy, the $\mathbb{P}_i$ distributions are the hash tables of variable value-fraction pairs and the weights are simply set to 1.

$$\mathrm{JSD}_{\pi_1,...,\pi_n}(\mathbb{P}_1, ..., \mathbb{P}_n) = H\Big(\sum_{i=1}^{n} \pi_i \mathbb{P}_i\Big) - \sum_{i=1}^{n} \pi_i H(\mathbb{P}_i) \tag{6.1}$$

The square root of the Jensen-Shannon Divergence gives the *Jensen-Shannon distance* [Lin, 1991], which is a metric between 0 and 1 used to describe the similarity between distributions (0 being exactly the same, 1 being completely

different). The TrafPy API enables users to specify their own maximum $\sqrt{JSD}$ threshold, $\sqrt{JSD_{\text{threshold}}}$, when sampling data from a set of original distributions to create their own data set(s). A lower distance requires that the sampled distributions be more similar to the original distributions. TrafPy will automatically sample more demands until, by the law of large numbers, the user-specified $\sqrt{JSD}$ threshold is met.

Fig. 6.2 shows how, for an example benchmark's flow size and inter-arrival time distribution, the $\sqrt{JSD}$ between the original and the sampled distributions changes with the number of samples (number of demands). As shown, most characteristic parameters (mean, minimum, maximum, and standard deviation) of the sampled distributions converge at $\sqrt{JSD} \approx 0.1$; a threshold reached after 137,435 demands for the flow size distribution and 27,194 for the inter-arrival times. The greater the number of possible random variable values and complexity in the original distribution, the more demands which will be needed to lower the $\sqrt{JSD}$. The distribution which requires the most demands to meet the $\sqrt{JSD}$ threshold will determine the minimum number of demands needed for the generated flow data set to accurately reproduce the original set from which it is sampled.

6.3.4 Node Distributions

'Node distributions' are a mapping of how much each machine (network node) pair tends to be requested by arriving flows, as measured by the pair's load (flow information arriving per unit time), to form a source-destination pair matrix. These distributions can be defined *explicitly* on a per-node basis. However, explicit mappings would result in D' being defined for a specific topology (since each topology might have a different number of machines and/or a different machine labeling convention). Therefore, TrafPy node distributions can also be *implicitly* defined by high-level parameters. These parameters are the fraction

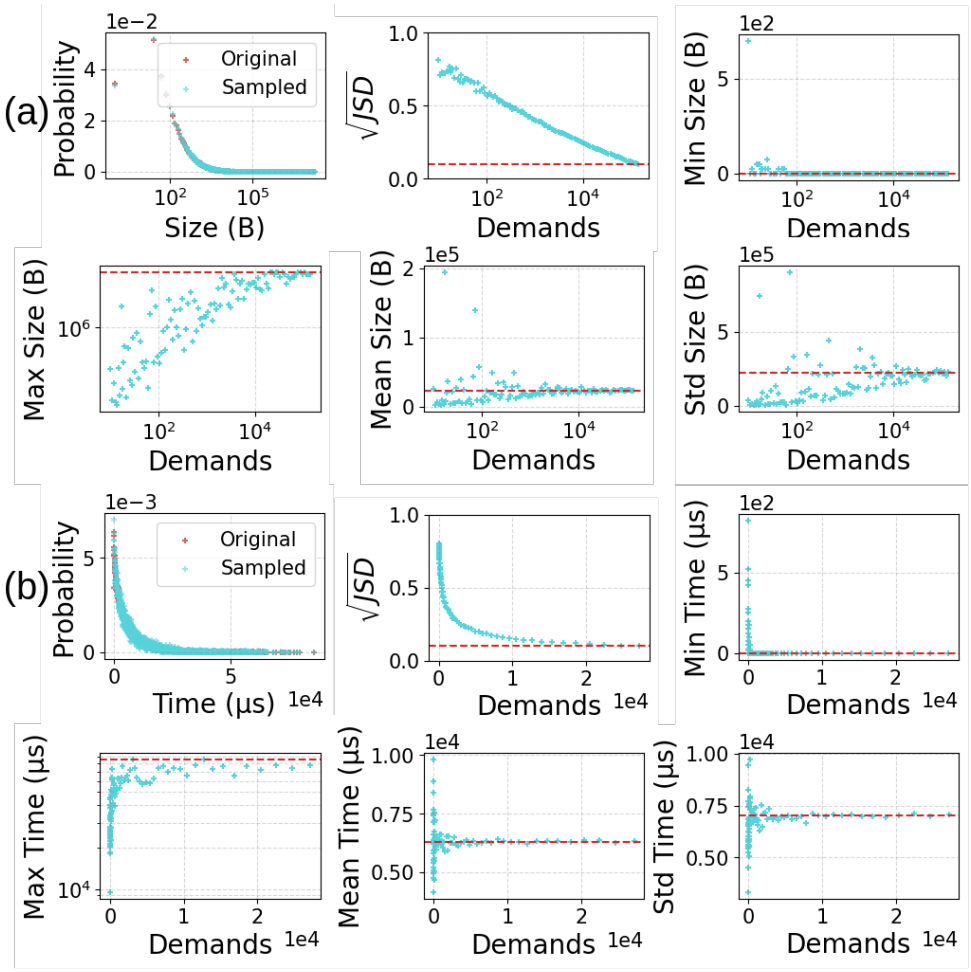


FIGURE 6.2: How the Jensen-Shannon distances between the original (red) and sampled (cyan) distributions and the sampled distributions' characteristic parameters (target from original distribution plotted as red dotted line) vary with the number of demands for (a) flow size and (b) inter-arrival time. Note that the first sub-plots of (a) and (b) are plotting the probability distribution of the flow characteristic in question, whereas the other sub-plots are plotting various metrics ($\sqrt{JSD}$, minimum value, maximum value, etc.) of the generated traffic as a function of the number of demands (flows) generated.

of the nodes and/or node pairs which account for some proportion of the overall traffic load and, optionally, the fraction of the traffic which is intra- vs. inter-cluster (where 'clusters' are usually considered as 'racks' in the context of DCNs). In this way, node distributions can be defined independently of the network topology, enabling greater generality and the use of custom topologies with traffic traces and benchmarks parameterised by D', even if D' was originally defined for a different topology. Furthermore, this allows individual or groups of network nodes to be set as 'hot', 'cold', or any combination of hot and cold as desired

by the user. Note that this formalism also enables both in-cast (many-to-one) and out-cast (one-to-many) traffic patterns, since any node(s) can have multiple out-cast and in-cast flow demands generated at a given point in time when sampling from the node distribution.

6.3.5 Traffic Generation Methodology

Algorithm 3 TrafPy traffic generation process.

Input: $\mathbb{P}(B^s)$, $\mathbb{P}(B^t)$, $\mathbb{P}(B^n)$, $\sqrt{JSD_{\text{threshold}}}$ ρ_{target}, $\langle n_n, n_c, C_c \rangle$, $t_{t,min}$
Output: $\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\}$
Initialise: n_f, $\{\boldsymbol{b}^s, \boldsymbol{b}^t\}$ empty arrays

Step 1: Partially initialise n_f flows $\{\boldsymbol{b}^s, \boldsymbol{b}^a\}$
while $\sqrt{JSD(\mathbb{P}(B^s), \mathbb{P}(\boldsymbol{b}^s))} \leq \sqrt{JSD_{\text{threshold}}}$ **do**
  $\mathbf{b}^s \leftarrow$ Sample b^s from $\mathbb{P}(B^s)$ n_f times
  $n_f := \lceil 1.1 \times n_f \rceil$
end while
while $\sqrt{JSD(\mathbb{P}(B^t), \mathbb{P}(\boldsymbol{b}^t))} \leq \sqrt{JSD_{\text{threshold}}}$ **do**
  $\mathbf{b}^t \leftarrow$ Sample b^t from $\mathbb{P}(B^t)$ n_f times
  $n_f := \lceil 1.1 \times n_f \rceil$
end while
$n_f = \textbf{max}(\{\textbf{length}(\boldsymbol{b}^s), \textbf{length}(\boldsymbol{b}^t)\})$
Resample so that $\textbf{length}(\boldsymbol{b}^s) = \textbf{length}(\boldsymbol{b}^t) = n_f$
Initialise $\boldsymbol{b}^a$ zero array of length n_f
for i in $[2, ..., n_f]$ **do**
  $\boldsymbol{b}^a_i := \boldsymbol{b}^a_{i-1} + \boldsymbol{b}^t_{i-1}$
end for
$\varrho = \frac{\sum_{i=1}^{n_f} \boldsymbol{b}^s_i}{\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0} \rightarrow \rho = \frac{\varrho}{\frac{n_n \cdot C_c \cdot n_c}{2}} \rightarrow \alpha_t = \frac{\rho}{\rho_{\text{target}}}$
for i in $[1, ..., n_f]$ **do**
  $\boldsymbol{b}^a_i := \alpha_t \times \boldsymbol{b}^a_i$
end for
$\varrho := \frac{\sum_{i=1}^{n_f} \boldsymbol{b}^s_i}{\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0} \rightarrow \rho := \frac{\varrho}{\frac{n_n \cdot C_c \cdot n_c}{2}}$

Step 2: 'Pack the flows' $\rightarrow$ fully initialise n_f flows $\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\}$
Initialise $\boldsymbol{b}^p$ and $\boldsymbol{b}^n$ from $\mathbb{P}(B^n)$ with $n_n^2 - n_n$ elements
$\boldsymbol{d} = \varrho \cdot \boldsymbol{b}^n \cdot (\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0)$
for i in $[1, ..., n_f]$ **do**
  Sort pairs in descending $\boldsymbol{d}_p$ order and randomly self-shuffle equal $\boldsymbol{d}_p$ pairs
  First pass: Attempt $\boldsymbol{d}_p \approx 0 \forall p \in [1, ..., n_n^2 - n_n]$
  for p in $[1, ..., n_n^2 - n_n]$ **do**
    if $\boldsymbol{d}_p - \boldsymbol{b}^s_i \geq 0$ **then**
      $\boldsymbol{b}^p_i = p$
      $\boldsymbol{d}_p := \boldsymbol{d}_p - \boldsymbol{b}^s_i$
      break
    end if
  end for
  if first pass unsuccessful **then**
    Second pass: Ensure no link capacity exceeds $\frac{C_c}{2}$
    for p in $[1, ..., n_n^2 - n_n]$ **do**
      if capacity not exceeded **then**
        $\boldsymbol{b}^p_i = p$
        $\boldsymbol{d}_p := \boldsymbol{d}_p - \boldsymbol{b}^s_i$
        break
      end if
    end for
  end if
end for

Step 3: Ensure $\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0 \geq t_{t,min}$
if $\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0 < t_{t,min}$ **then**
  $\beta = \left\lceil \frac{\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0}{t_{t,min}} \right\rceil$
  $\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\} := \textbf{double}(\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\})$ β times
end if

Given the distributions of flow sizes, inter-arrival times, and node pairs $\mathbb{P}(B^s)$, $\mathbb{P}(B^t)$, and $\mathbb{P}(B^n)$ of a benchmark B, TrafPy can generate traffic at a (optionally) specified target load fraction (fraction of overall network capacity being requested for a given time period) $\rho_{\text{target}} \in [0, 1]$ with maximum Jensen-Shannon distance threshold $\sqrt{JSD_{\text{threshold}}}$ for an arbitrary topology T with n_n server nodes, n_c channels (light paths) per communication link, and C_c capacity per server node link channel (divided equally between the source and destination ports such that each machine may simultaneously transmit and receive data), forming tuple $\langle n_n, n_c, C_c \rangle$ with total network capacity per direction (maximum information units transported per unit time) $C_t = \frac{n_n \cdot C_c \cdot n_c}{2}$. Since load rate is defined as information arriving per unit time, in order to generate traffic at arbitrary loads, either the amount of information (flow sizes) or the rate of arrival (flow inter-arrival times) must be adjusted in order to change the load rate. Since DCNs tend to handle particular types of applications and jobs which result in particular flow sizes, we posit that a reasonable assumption is that changing loads are the result of changing rates of demand arrivals rather than changing flow sizes (which remain fixed for a given application type). Therefore, if a target load is specified, TrafPy automatically adjusts the scale of the inter-arrival time distribution values in $\mathbb{P}(B^t)$ by a constant factor to meet the target load whilst keeping the same general shape of the $\mathbb{P}(B^t)$ distribution that was initially input to the generator. The following 3-step traffic generation process (summarised in Algorithm 3) is used to achieve the above:

Step 1 (generate n_f flows with size and arrival time characteristics $\{\boldsymbol{b}^s, \boldsymbol{b}^a\}$): First, n_{b^s} flow sizes and n_{b^t} inter-arrival times are independently sampled from $\mathbb{P}(B^s)$ and $\mathbb{P}(B^t)$ to form vectors $\boldsymbol{b}^s$ and $\boldsymbol{b}^t$ respectively, where n_{b^s} and n_{b^t} are incrementally increased by a constant factor until $\sqrt{JSD(\mathbb{P}(B^s), \mathbb{P}(\boldsymbol{b}^s))} \leq \sqrt{JSD_{\text{threshold}}}$ and $\sqrt{JSD(\mathbb{P}(B^t), \mathbb{P}(\boldsymbol{b}^t))} \leq \sqrt{JSD_{\text{threshold}}}$ by the law of large numbers. Whichever distribution needed fewer samples to meet $\sqrt{JSD} \leq \sqrt{JSD_{\text{threshold}}}$ is then continually sampled such that there are n_f flow sizes

and inter-arrival times, where $n_f = \mathbf{max}(\{n_{b^s}, n_{b^t}\})$. Then, $\boldsymbol{b}^t$ (whose order is arbitrary from the previous random sampling process) can be converted to an equivalent arrival time vector $\boldsymbol{b}^a$ by initialising a zero array of length n_f and setting $\boldsymbol{b}^a_i := \boldsymbol{b}^a_{i-1} + \boldsymbol{b}^t_{i-1} \forall i \in [2, ..., n_f]$, resulting in a total time duration of $t_t = \boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0$ over which the flows arrive. Next, the load rate ϱ is evaluated with $\varrho = \frac{\sum_{i=1}^{n_f} \boldsymbol{b}^s_i}{t_t}$, converted to a load fraction $\rho = \frac{\varrho}{C_t}$, and adjusted to meet ρ_{target} by multiplying the elements of $\boldsymbol{b}^t$ by a constant factor $\alpha_t = \frac{\rho}{\rho_{\text{target}}}$. Then, $\boldsymbol{b}^a$ can be re-initialised with the updated $\boldsymbol{b}^t$ as before, and a set $\{\boldsymbol{b}^s, \boldsymbol{b}^a\}$ of n_f flows can be partially initialised each with size b^s and arrival time b^a and an overall load $\rho = \rho_{\text{target}}$ on network T.

Step 2 ('pack the flows' $\rightarrow$ generate n_f flows with size, arrival time, and source-destination node pair characteristics $\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\}$): Next, to meet the source-destination node pair load fractions specified by $\mathbb{P}(B^n)$, the flows are packed into node pairs with a simple packing algorithm. First, a vector of $n_n^2 - n_n$ node pairs $\boldsymbol{b}^p$ (which do not include self-similar pairs) and their corresponding load pair fractions $\boldsymbol{b}^n$ are extracted from $\mathbb{P}(B^n)$. Next, these 'target' load pair fractions $\boldsymbol{b}^n$ are converted into a hash table mapping each pair p of the $[1, ..., n_n^2 - n_n]$ pairs to their current 'distance' from their respective target total information request magnitudes $\boldsymbol{d} = \varrho \cdot \boldsymbol{b}^n \cdot t_t$. In other words, we take the load fractions (fraction of overall information requested) of each node pair $\boldsymbol{b}^n$ and multiply them by the total simulation load rate (information units arriving per unit time) ϱ and the total simulation time t_t to create a vector $\boldsymbol{d}$ which, when first initialised, represents the total amount of information which is requested by each source-destination pair across the whole simulation. The task of the packer is therefore to assign source-destination pairs to each flow such that $\boldsymbol{d}_p \approx 0 \forall p \in [1, ..., n_n^2 - n_n]$. For each sequential i^{th} flow $\forall i \in [1, ..., n_f]$, after sorting the pairs in descending $\boldsymbol{d}_p$ order (with any pairs with equal $\boldsymbol{d}_p$ randomly shuffled amongst one-another), the packer will try to 'pack the flow' (given its size $\boldsymbol{b}^s_i$) into a source-destination pair in two passes. For the first pass the packer

loops through each sorted p^{th} pair $\forall p \in [1, ..., n_n^2 - n_n]$ and checks that assigning the flow to this pair would not result in $\boldsymbol{d}_p < 0$. If this condition is met, the packer sets $\boldsymbol{b}_i^p = p$ and $\boldsymbol{d_p} := \boldsymbol{d_p} - \boldsymbol{b}_i^s$ before moving to the next flow. However, if the condition is violated for all pairs, the packer moves to the second pass, where it again loops through each sorted pair p but now, rather than ensuring $\boldsymbol{d}_p \geq 0$, only ensures that assigning the pair would not exceed the maximum server link's source/destination port capacity $\frac{C_c}{2}$ before setting $\boldsymbol{b}_i^p = p$ and $\boldsymbol{d}_p := \boldsymbol{d}_p - \boldsymbol{b}_i^s$. In other words, the first pass attempts to achieve $\boldsymbol{d}_p \approx 0 \forall p \in [1, ..., n_n^2 - n_n]$ to try to match $\mathbb{P}(B^n)$ but, failing that, the second pass ensures that no server link load exceeds 1.0 of the link capacity. Consequently, as ρ_{target} approaches 1.0, so too will the resultant packed node distribution's server links, thereby converging on a uniform distribution no matter what the original skewness was of $\mathbb{P}(B^n)$ as shown in Fig. 6.3 and further elaborated on in Appendix B.5. Once this packing process is complete, a set $\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\}$ of n_f flows each with size b^s, arrival time b^a, and source-destination node pair b^p, an overall load ρ_{target} on network T, and a flow size, inter-arrival time, and node distribution of approximately $\mathbb{P}(B^s)$, $\mathbb{P}(B^t)$, and $\mathbb{P}(B^n)$ will have been fully initialised.

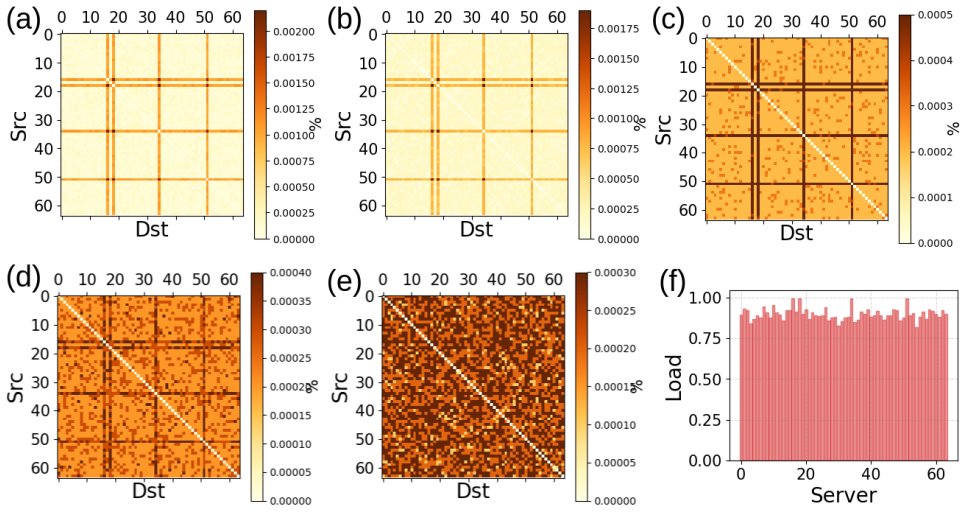


FIGURE 6.3: Visualisation of the packed flow nodes converging on uniform distributions as the total network load approaches 1.0 regardless of how skewed the original target node distribution is. The plotted distributions are for overall network loads (a) 0.1, (b) 0.3, (c) 0.5, (d) 0.7, and (e) 0.9, and (f) the final demonstrably uniform endpoint loads on each server at 0.9 overall load.

Step 3 (ensure $\boldsymbol{b}^a_{n_f} - \boldsymbol{b}^a_0 \geq t_{t,min}$): The final stage of the flow generation process is then to ensure that the flow arrival duration t_t is greater than or equal to some minimum duration $t_{t,min}$ (a parameter often required for test bed measurement reliability) specified by either the user. This is done by simply doubling the set $\{\boldsymbol{b}^s, \boldsymbol{b}^a, \boldsymbol{b}^p\}$ of flows $\beta = \lceil \frac{t_t}{t_{t,min}} \rceil$ times to make an updated set of $n_f := \beta \cdot n_f$ flows with $t_t \geq t_{t,min}$ and the same distribution and load statistics as before.

6.3.6 Stipulating Traffic Generation Guidelines

Given a user- or benchmark-specified set of distribution parameters D', TrafPy generates traffic trace D. As such, whenever using TrafPy to generate D, D' should always be reported to help others reproduce the same trace (as done in Table B.2 of Appendix B.2 for this chapter). For the same reason, all traffic traces D generated from D' should have a maximum $\sqrt{JSD_{\text{threshold}}}$ of 0.1 as outlined in Section 6.3.3. Enough demands should be generated so as to have a last demand arrival time t_t larger than the time needed to complete the largest demands in the user-defined network T under the test conditions used; not doing so would result in all large flows being dropped regardless of what decisions were made. This would unfairly punish systems optimised for large demands, since such systems would allocate network resources to requests which ultimately could never be completed during the experiment. TrafPy conveniently generates and saves traffic data sets in a range of formats including JSON, CSV, and pickle. Therefore if desired, users may generate traffic in TrafPy and then use their own custom test bed and analysis scripts written in any programming language thereafter by simply importing the TrafPy-generated traffic. For result reliability, each trace D should be generated R times from D' and used to test the network object, where R should be sufficiently large enough so as to have a satisfactory confidence interval (which might vary between projects but should

be reported regardless).

6.4 Experimental Setup

Here we conduct a brief experiment into the sensitivity of four schedulers to different traffic traces. Specifically, we look at shortest remaining processing time (SRPT) [Cai et al., 2016, Alizadeh et al., 2013, Hong et al., 2012], fair share (FS) [Cai et al., 2016], first fit (FF) [Al-Fares et al., 2010], and random DCN flow scheduling. We note that while TrafPy can be used to rigorously investigate and understand different scheduling systems and topologies, the purpose of the experiments ran here is to illustrate how TrafPy can be used to benchmark systems. A deep analysis and investigation of the scheduling algorithms, topologies, and other state-of-the-art systems beyond those considered here is left for further work.

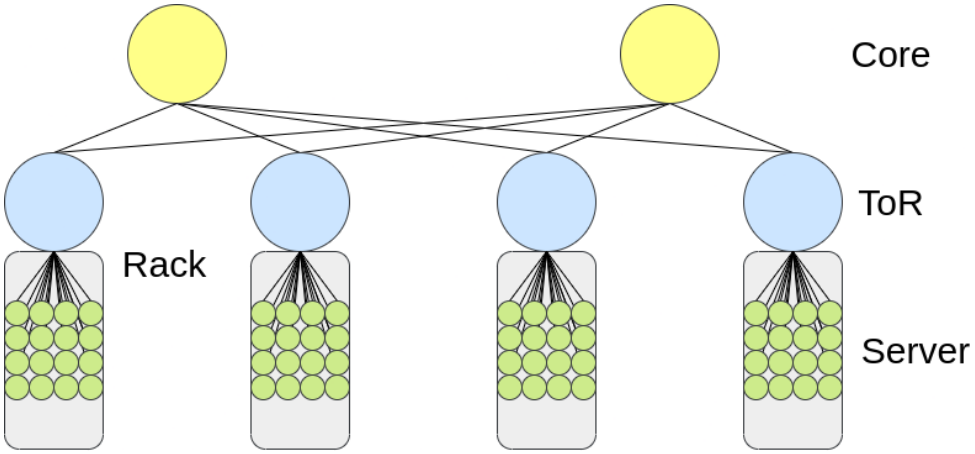


FIGURE 6.4: 2-layer spine-leaf topology used with 64 end point (server) nodes, 10 Gbps server-to-ToR links, and 80 Gbps ToR-to-core links (1:1 subscription ratio, 640 Gbps total network capacity).

6.4.1 Network

All experiments assume an optical TDM-based circuit switched network architecture with a 64-server folded clos (spine-leaf) topology made up of 2 core switches,

4 top-of-the-rack (ToR) switches, and 64 servers (16 servers per rack) with bidirectional links, as shown in Fig. 6.4. The server-to-rack and ToR-to-core links each have one channel with 10 Gbps and 80 Gbps capacity respectively, leading to a 1:1 subscription ratio and a total network capacity of 640 Gbps (320 Gbps bisection bandwidth). Flows are mapped to TDM circuits, and we assume ideal server-level time multiplexing of the flows' packets such that the bandwidth of each channel can be fully utilised. The core switch performs link/fiber switching. There are various ways to perform packet/TDM aggregation of flows at the server and to realise such networks, but neither are the focus of this work.

6.4.2 Traffic Traces

We use TrafPy to generate two categories of traffic with which to investigate our schedulers; DCN traces based on real-world application data, and custom skewed node and rack data for testing system performance under extreme conditions. We use a maximum $\sqrt{JSD_{\text{threshold}}}$ of 0.1, setting $t_{t,min}$ =3.2e5 μs (≈10 times larger than the time taken to complete the largest ≈20e6 B flow amongst our benchmarks), and generating traffic of loads 0.1-0.9 for each data set. We generate each set $R = 5$ times to run five repeats of our experiments and therefore ensure reliability. All TrafPy parameters D' used to generate the traffic are reported in Table B.2 of Appendix B.2 for reproducibility.

'Realistic' DCN traces. Four types of Data Centers and their network flow demand distributions are explored; *University* [Benson et al., 2010a], *Private Enterprise* [Benson et al., 2011], *Commercial Cloud* [Kandula et al., 2009], and *Social Media Cloud* [Roy et al., 2015]. Each DCN type services different applications and therefore has a different traffic pattern. Using TrafPy, flow distributions for each of these categories were generated to established a set of open-source traffic traces for the *DCN benchmark*. The tuned TrafPy parameters D' of each flow characteristic have been summarised in Table B.2. The resultant

distributions are shown in Fig. 6.5, and the subsequent quantitative summary of each distribution's characteristics is given in Table B.3 of Appendix B.2.

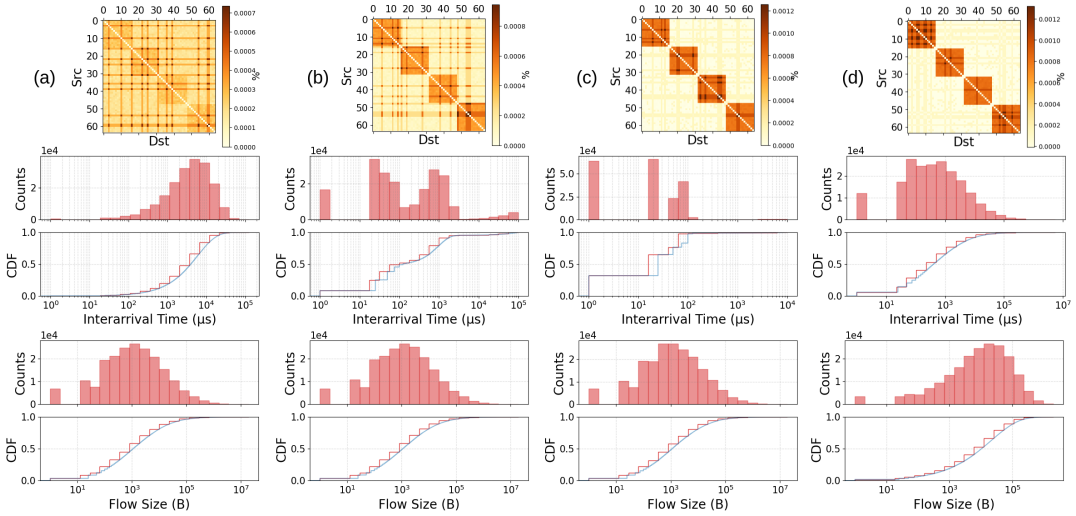


FIGURE 6.5: TrafPy distribution plots for the *DCN benchmark* containing the (a) University [Benson et al., 2010a], (b) Private Enterprise [Benson et al., 2011], (c) Commercial Cloud [Kandula et al., 2009], and (d) Social Media Cloud [Roy et al., 2015] data sets. Each plot contains (i) the end point node load distribution matrix and (ii) the flow size and inter-arrival time histogram and CDF distributions.

'Extreme' skewed node and rack sensitivity traces. We generated two additional traces; the *skewed nodes sensitivity* benchmark and the *rack sensitivity* benchmark. These were not based on realistic data, but rather designed to test and better understand our systems under extreme conditions. Both use the same flow size and inter-arrival time distributions as the commercial cloud data set in Fig. 6.5, however the node distribution is adjusted. Specifically, the skewed nodes benchmark is made up of five sets with uniform, 5%, 10%, 20%, and 40% of the server nodes being 'skewed' by accounting for 55% of the total overall traffic load, named *skewed_nodes_sensitivity_uniform*, *0.05*, *0.1*, *0.2*, and *0.4* respectively (see Appendix B.5 for further justification and analysis of these values). Similarly, the rack distribution benchmark is made up of 5 sets with uniform, 20%, 40% , 60%, and 80% of the traffic being intra-rack (and the rest inter-rack) named *rack_sensitivity_uniform*, *0.2*, *0.4*, *0.6*, and *0.8* respectively. Therefore, these distributions allow for investigations into DCN

system sensitivity to i) the number of skewed nodes and ii) the ratio of intra- vs. inter-rack traffic. They have been plotted in Fig. 6.6.

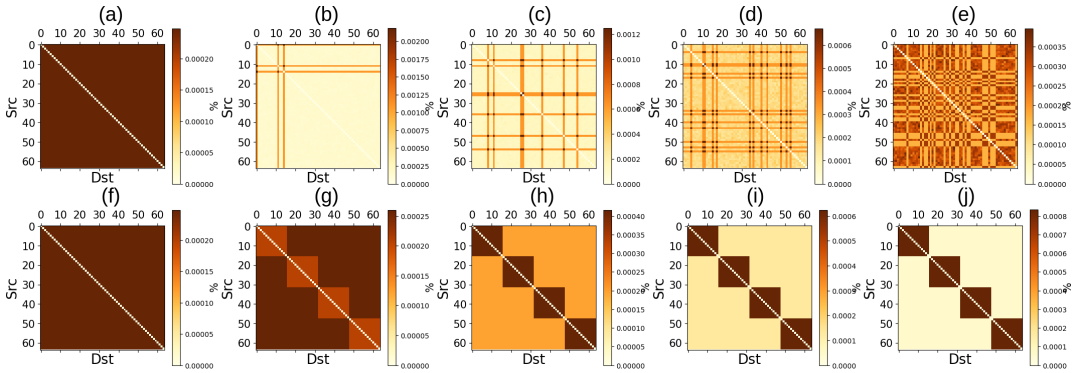


FIGURE 6.6: TrafPy node distribution plots for the *skewed nodes sensitivity* benchmark with (a) uniform, (b) 5%, (c) 10%, (d) 20%, and (e) 40% of nodes accounting for 55% of the overall traffic load, and for the *rack sensitivity* benchmark with (f) uniform, (g) 20%, (h) 40%, (i) 60%, and (j) 80% traffic being intra-rack and the rest inter-rack.

6.4.3 Simulation Details

We use a time-driven simulator where scheduling decisions are made at fixed intervals. The time between decisions is the 'slot size'; smaller slot sizes result in greater scheduling decision and measurement metric granularity, but at the cost of longer simulation times and the need for scheduler and switch hardware optimisation [Benjamin, 2020, Parsonson et al., 2020, Gerard et al., 2020a, 2021, Benjamin et al., 2021]. We use a slot size of 1 ms. We assume perfect packet time-multiplexing whereby the scheduler is allowed to schedule as many flow packets for the next time slot as the channel bandwidth of its rate-limiting link in its chosen path will allow. We run 9 simulations (loads 0.1-0.9) for each benchmark data set, terminating the simulation when the last demand arrives at $t = t_t$ (which is $\geq t_{t,min}$ =3.2e5 μs). We set the warm-up time as being 10% of the simulation time t_t before which no collected data contribute to the final performance metrics. Similarly, since the simulation is terminated at t_t, we exclude any cool-down period from measurement. For each experiment, we then record: (1) mean flow completion time (FCT); (2) 99^{th} percentile (p99)

FCT; (3) maximum (max) FCT; (4) absolute throughput (total number of information units transported per unit time); (5) relative throughput (fraction of arrived information successfully transported); and (6) fraction of arrived flows accepted. We report each of these metrics' mean across the $R = 5$ runs and their corresponding 95% confidence intervals.

6.5 Results & Discussion

To begin the investigation into the sensitivity of different schedulers, we first input TrafPy-generated traffic with heavily skewed nodes and racks (see Section 6.4.2) into our simulator to understand how the four schedulers considered behave at the extremes. We then test the same schedulers under traces for different DCN types to see how the results from the 'extreme' condition investigation translate into more realistic scenarios. For brevity, we provide the full results in Appendix B.6 and a summary in this section.

Extreme rack conditions. As shown in Table B.17, as the rack distribution becomes heavily skewed to intra-rack, the completion time metrics of FS become increasingly superior to SRPT. This suggests that real DCNs which have heavy intra-rack traffic (e.g. social media cloud DCNs) would benefit from deploying pure FS scheduling policies, at least at higher loads, whereas DCNs with heavy inter-rack traffic (e.g. university DCNs) would benefit from deploying FS at medium loads and SRPT at low and high loads.

In terms of throughput and demands accepted, FF is competitive with SRPT and FS at low intra-rack traffic levels, but as the DCN becomes more heavily intra-rack (e.g. social media cloud DCNs), SRPT and FS are preferable, with FS achieving the best performances at higher loads. Again, a preferable strategy would likely be to utilise SRPT strategies at low loads before switching to FS at loads about 0.3 to 0.5 (depending on the level of intra-rack traffic).

Extreme node conditions. As shown in Table B.18, at the two extremes of heavily skewed and uniform traffic, scheduler completion time performances are similar in that SRPT outperforms FS at low and high loads, but FS performs well at medium loads. However, in between these two extremes (around 40% of nodes requesting 55% of overall traffic), there is a point where FS becomes the dominant scheduler in terms of completion time.

In terms of throughput and demands accepted, under heavily skewed conditions (5% nodes requesting 55% of traffic), FF and/or Rand beat SRPT and FS across all 0.1-0.9 loads in terms of throughput and fraction of information accepted. This suggests that FF and SRPT are strained under high skews with respect to these two metrics. However, as observed with the uniform distribution, this comes at the cost of the fraction of arrived flows accepted, where SRPT and FS outperform FF and Rand across all loads. As the proportion of nodes requesting 55% of traffic is increased to 10%, 20%, and 40%, relative scheduler performances converge to those seen with the uniform distribution, with FS and SRPT being mostly dominant except at high 0.8 and 0.9 loads, where FF often has the better throughput and fraction of information accepted.

Realistic conditions. Table B.19 summarises the results for the four schedulers on each of the four 'realistic' DCN benchmarks considered. As shown, the SRPT scheduler tends to achieve the best completion time metrics when loads are low (≤ 0.7) and where traffic is primarily inter-rack (the University and Private Enterprise DCNs). This is to be expected, since a policy which prioritises completion of the smallest flows as soon as possible will keep its completion time averages low. However, as traffic reaches higher loads (> 0.7), the fair share policy achieves the best completion time metrics. This indicates that networks would benefit from scheduling policies which can dynamically adapt to changing traffic loads. Moreover, for networks with characteristically intra-rack traffic (the Commercial Cloud and Social Media Cloud DCNs), the fair share policy attains the best completion time and throughput metrics. These

results therefore validate the predictions made by the rack distribution sensitivity analysis study; namely that completion time metrics in real DCN traces with heavily intra-rack (e.g. Commercial Cloud and Social Media Cloud) traffic benefit from FS scheduling strategies. On the other hand, at least for low loads, low intra-rack DCN traces (e.g. University and Private Enterprise) benefit from SRPT scheduling strategies.

These results suggest that not only should scheduling policies be adapted to changing traffic loads, but also to changing characteristics such as the level of inter- vs. intra-rack communication. Note that, as expected, the fair share policy provides the best worst-case completion time (max FCT), the greatest network utilisation (throughput), and the strongest service guarantee (number of flow requests satisfied) across most loads and DCN types.

6.6 Conclusions, Limitations, & Further Work

In conclusion, we have introduced TrafPy; an API for generating custom and realistic DCN traffic and a standardised protocol for benchmarking DCN systems which is compatible with any simulation, emulation, or experimentation test bed. These systems can be any combination of networked devices or methods such as schedulers, switches, routers, admission control policies, management protocols, topologies, buffering methods, and so on. TrafPy has been developed with a focus on having a high level of *fidelity*, *generality*, *scalability*, *reproducability*, *repeatability*, *replicability*, *compatbility*, and *comparability* in the context of DCN research, which in turn will aid in accelerating innovation.

We have demonstrated the efficacy of TrafPy by briefly investigating the sensitivity of four canonical schedulers to varying traffic loads and characteristics. The scheduler performances were heavily dependent on the level of intra-rack traffic and overall network load. We found that SRPT was generally the dominant scheduler for low intra-rack traffic (particularly at low loads), but that FS became

superior across all loads at high intra-rack levels. These insights were then found to translate into realistic DCN traces, with low intra-rack users such as University and Private Enterprise DCNs benefiting from SRPT policies at low and medium loads and high intra-rack traces such as Commercial Cloud and Social Media Cloud being more suited to FS strategies. This shows that there is no 'one size fits all' strategy for scheduling different types of DCNs, and that there would be great value in the development of traffic-informed and dynamic DCN systems. With its standardised traffic generation and benchmark protocol, TrafPy is an ideal tool for developing such systems via the benchmark paradigm described throughout this chapter.

The space of potential research areas from this work is vast. We hope presently unforeseeable avenues will be pursued with the support of TrafPy's standardised traffic generation and rigorous benchmarking framework. For example, based on the preliminary results of scheduler sensitivity to varying load conditions and traffic trace characteristics, we expect new scheduling heuristics and learning algorithms to be developed which can dynamically adapt to network traffic states and outperform literature baselines in open-source TrafPy benchmarks. The 2.5 TB of open-access simulation data from this chapter enable some interesting offline reinforcement learning opportunities.

With regards to how future research might enhance TrafPy itself, here we outline some of the limitations of this chapter and interesting avenues of further work.

More benchmark traffic traces. TrafPy has been built to enable users to easily establish and share new benchmark traffic traces. Therefore, in addition to the initial benchmark traffic distributions introduced in this chapter, future works could develop new benchmarks. These might consider modelling proprietary traffic traces which cannot be open sourced directly but which can be legally synthetically mimicked. Alternatively, they could be presently unrealistic traces designed to test systems under extreme conditions or to model systems in

environments which are expected to exist in the future.

Automatic characterisation & imitation of real data. Although TrafPy can generate traces without any raw data given whatever characteristic distributions the user provides, it would be useful to be able to input real data (e.g. [Bai et al., 2016]) and have TrafPy automatically characterise the traffic in order to generate realistic data. An interesting extension to this would be to build a tool that learns to generate synthetic data from a limited sample of real data, possibly with the use of generative ML.

Expansion to the job-centric traffic paradigm. In this chapter, we have considered the flow-centric traffic paradigms where DCN demands are considered as network flows. However, as discussed in Appendix B.7, in practice network flows arise from jobs being submitted to the DCN. A useful project would therefore be to extend TrafPy's functionality to generate DCN jobs from which flow traffic would arise. This would not only more accurately mimic real DCN job tasks, but also bridge the gap between the computer science and networking communities, which usually consider the job- and flow-centric paradigms in isolation.

Establishing TrafPy-powered leaderboards. The flourishing of AI over the last decade can be attributed to the winner of the 2012 ImageNet competition. Without ImageNet, it would have been difficult to demonstrate the primary of neural networks trained on GPUs at real world tasks such as image classification. Benchmarking with leaderboards which compare and evaluate systems on the same task under strict constraints can lead to the highly effective research and development of novel systems. Therefore, a useful project would be to use TrafPy to establish open-access and rigorous benchmarking leaderboards, similar to ImageNet, which evaluate systems such as flow schedulers on particular tasks.

Beyond data centre traffic. The version of TrafPy propsed here has been specifically designed for generating DCN traffic. However, there is no reason why TrafPy should be restricted in this way. Fundamentally, all traffic in any

network, from long-haul core networks to road transportation networks, can be modelled by flows which have a source, destination, arrival time, and some 'cost' of transport (size, time, fuel, and so on). Future works might therefore work on generalising the language and front-end interface of TrafPy to be generic to any form of network traffic generation.

Chapter 7

Accelerating Traffic Matrix Generation at Scale

Abstract

This chapter proposes a new algorithm for generating custom network traffic matrices which achieves 13×, 38×, and 70× faster generation times than the algorithm originally proposed for TrafPy traffic generation on networks with 64, 256, and 1024 nodes respectively.

Publications related to this work (contributions indented):

- **Christopher W. F. Parsonson**, Joshua L. Benjamin, and Georgios Zervas, 'A Vectorised Packing Algorithm for Efficient Generation of Custom Traffic Matrices', *OFC'23: Optical Fiber Communications Conference and Exhibition*, 2023
  - Algorithm, code, experiments, paper writing, plots

7.1 Introduction

Data centres have become critical tools for modern computational tasks. To meet the ever-increasing demands of data centres, recent years have seen a growth in the research and development of next-generation data centre optical systems [Khani et al., 2021]. However, most researchers rely on simulations, which require the generation of synthetic traffic. In doing so, they often make overly simplistic assumptions about the characteristics of their generated traffic and develop systems which, in practice, perform poorly under real-world conditions [Parsonson et al., 2022]. Furthermore, many works omit open-accessing their synthetic traffic or even the methodology used to generate it, bringing problems with reproducibility, benchmarking, and cross-validation. The lack of a reproducible and high-fidelity synthetic traffic generation tool has been a long-standing problem in the data centre research community.

Prior works [Alizadeh et al., 2013, Bai et al., 2016] have released traffic generators, but these were either intended to be unrealistic, were for specific network topologies, required the cumbersome use of inflexible configuration files, or lacked a reproducibility guarantee. To address this, Chapter 6 presented TrafPy; an open source tool for generating reproducible data centre traffic with custom distributions and characteristics [Parsonson et al., 2022]. However, Chapter 6 only demonstrated traffic generation for 64 network nodes; far smaller than the $O(1000)$ node data centres which are becoming increasingly common place.

In this chapter, we first show that the original flow source-destination assignment algorithm ('packing', see Section 7.2) presented in Chapter 6 is a major bottleneck when generating traffic with TrafPy because its time complexity scales poorly with the number of data centre nodes $|N|$ for which $|F|$ flows are being generated. This prevents the generation of traffic for large networks. Next, we propose a novel vectorised packing algorithm which fits in with the rest of the

TrafPy traffic generation framework. Finally, we demonstrate the new vectorised packer achieving 13×, 38×, and 70× faster generation times than originally reported in Chapter 6 on networks with 64, 256, and 1024 nodes respectively with up to ≈ 5M traffic flows, with the speed-up factor increasing with the network size. We expect this work to unlock a new realm of data centre research at scale and to further facilitate the development of next-generation systems and common platforms for benchmarking networks. We note that while here we focus on generating traffic for optical data centres, the same traffic generation scheme and vectorised packing algorithm could be re-purposed and applied to any network system.

7.2 Custom Traffic Matrix Generation

Problem statement. Data centre traffic is made up of *flows*. A flow f is fully described by its *size* f^s (how much information to send), *arrival time* f^a (when the flow requests to be transported through the data centre, thus giving rise to the *inter-arrival time* in a dynamic multi-flow setting), and *source-destination pair* f^p (which machines in the data centre the flow is requesting to be sent between). In the framework presented in Chapter 6, traffic generation is split into two stages. In the first stage ('shaping and sampling'), custom flow size and inter-arrival time distributions are generated and sampled to attain a set of sizes $\mathbf{b}^s$ and arrival times $\mathbf{b}^a$ for $f \in F$ flows which match the target distributions within some Jensen-Shannon distance (JSD) threshold[1]. In the second stage ('packing'), given $\mathbf{b}^s$ and $\mathbf{b}^a$, the task is to assign each flow $f \in F$ to a source-destination pair such that some target node distribution (a.k.a. traffic matrix heat map) $\mathbb{P}^N$ with nodes $n \in N$ and corresponding source-destination node pairs $p \in P$ is realised as closely as possible without exceeding the load capacity

[1]The JSD $\in [0, 1]$ is a measure of how similar two distributions are to one another (lower is more similar), and the JSD 'threshold' as defined in Chapter 6 is a constraint on how similar the generated traffic characteristics must be to the target distributions.

limitations of any node. Chapter 6 formulates this task by extracting the fraction of the overall load requested by each pair $p \in P$ into an array, multiplying each element by the overall data centre's target load rate to get the per-pair target load rate, and then again multiplying each element by the simulation duration (the time between the first and last flows' arrivals) to get the total amount of information to load onto each pair, $\mathbf{b}^{p,I}_{target}$, needed in order to achieve the desired target node distribution $\mathbb{P}^N$. The packing task is therefore reduced to finding the source-destination node pair assignments for each flow $f \in F$ such that the difference between the actual and the target per-pair total information loads, $\mathbf{b}^{p,I}_{target} - \mathbf{b}^{p,I}_{actual}$, is 0 *or*, where this is not possible given any incompatibility between the target node distribution $\mathbb{P}^N$ and the overall data centre load rate, to match $\mathbb{P}^N$ as closely as possible (see Chapter 6 for further details).

As shown in Fig. 7.1a, as $|N|$ is increased, stage two (packing) becomes a major bottleneck, taking ≈1 000 000 times longer than stage one for $|N|$ =1024. Therefore, in this chapter we focus on optimising stage two.

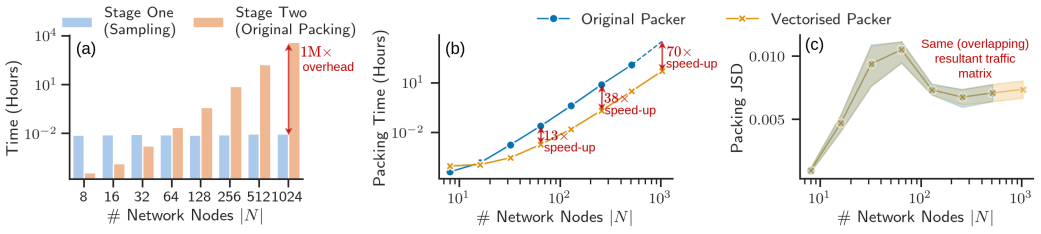


FIGURE 7.1: i) (a) The time for stages one (shaping and sampling) and two (packing) when generating flows with the original packing algorithm. ii) The packing (b) time and (c) Jensen-Shannon distance between the target and the generated node distributions for the original and vectorised packing algorithms when generating traffic for networks with different numbers of nodes. (a) shows that the original packing algorithm is the major traffic generation bottleneck of Chapter 6. (b) shows that as the number of network nodes is increased, the vectorised packer's speed-up factor over the original algorithm increases. (c) shows that both algorithms achieve the exact same resultant node distribution. Note that the original algorithm's time results for $|N| = 1024$ are extrapolations since it would have taken ≈ 200 days to run the packer.

Original packing algorithm. The original packing algorithm presented in Chapter 6 works by sequentially iterating through the set of flows and, for

each flow, conducting two passes through the candidate source-destination pairs. In the first pass, the packer attempts to match the target node distribution by looping through all pairs, sorted in descending order of the total size of flow information previously assigned, to find a pair which has not yet met its target information load given the target node distribution and total flow arrival duration provided. Failing to find such a node pair, the packer moves to the second pass, whereby it again loops through each sorted pair but now in search of a source-destination combination which, if allocated the flow in question, would not exceed either the source's or the destination's maximum load capacity given any prior flow allocations.

Algorithm 4 Vectorised packing algorithm pseudocode.

Input: F, P, $\mathbf{b}^{p,I}_{target}$

Output: $\mathbf{b}^{p,I}_{actual}$

Initialise: $\mathbf{b}^{p,I}_{actual} = 0(|P|)$, $\mathbf{b}^{p,c} = \frac{\text{node capacity}}{2}(|P|)$

for f in F **do**

$\mathbf{b}^{p,m}$ =`where`$(\mathbf{b}^{p,c} - f^s < 0, 0, 1)$ `// Generate boolean mask`

$\mathbf{b}^{p,I,m}_{target} = \mathbf{b}^{p,I}_{target}[\mathbf{b}^{p,m}], \mathbf{b}^{p,I,m}_{actual} = \mathbf{b}^{p,I}_{actual}[\mathbf{b}^{p,m}]$ `// Mask invalid pairs`

$\mathbf{p}^{max} =$ `argmax`$(2 \cdot \mathbf{b}^{p,I,m}_{target} - \mathbf{b}^{p.I,m}_{actual})$ `// Get furthest pairs`

$p^{chosen} =$ `random_choice`$(\mathbf{p}^{max})$ `// Randomly choose pair`

`update_trackers`(f, p^{chosen}) `// Assign flow to pair`

end for

Vectorised packing algorithm. We negate the need for separate first and second passes and for nested pair `for` loops by using vector array operations. We begin by initialising the per-pair remaining capacity vector as the maximum port capacity (half the per-node capacity, since it is split between the source and destination ports) $\mathbf{b}^{p,c}$. We then sequentially iterate through $f \in F$ and, for each flow f, we generate a boolean vector pairs mask $\mathbf{b}^{p,m}$ which masks out any pair indices $i \in [0, ..., |P|]$ which would exceed their load capacity were they to be allocated the flow in question:

$$\mathbf{b}_i^{p,m} = \begin{cases} 0 & \text{if } \mathbf{b}_i^{p,c} - f^s < 0 \\ 1, & \text{otherwise} \end{cases} \tag{7.1}$$

We then apply this pairs mask to filter out any invalid pairs, thus ensuring that any pair chosen from here on would meet the requirements of the second pass of the generation methodology in Chapter 6 and also reducing the time complexity of the `argmax` operation below in Eq. 7.2 (since the number of candidate pairs is now reduced). Next, we take the masked candidate pairs' current distances from the target information loads, $\mathbf{b}_{target}^{p,I,m} - \mathbf{b}_{actual}^{p,I,m}$, shift them by $\mathbf{b}_{target}^{p,I,m}$ in order to retain any skewness in $\mathbb{P}^N$ for as long as possible given the overall data centre load specified, and find the pairs in this masked subset which are furthest from their target information loads, $\mathbf{p}^{max}$:

$$\mathbf{p}^{max} = \text{argmax}\left(2 \cdot \mathbf{b}_{target}^{p,I,m} - \mathbf{b}_{actual}^{p,I,m}\right) \tag{7.2}$$

In order to avoid any bias towards smaller pair indices and create the fade phenomenon in the resultant traffic heat map (see Section 6.3.5), we randomly choose a pair $p_{chosen} \in \mathbf{p}^{max}$ to which to allocate the flow f, thus meeting the requirements of the first pass of Chapter 6. Finally, we update the current total information vector's element for the chosen pair, $\mathbf{b}_{actual}^{p_{chosen},I}$, and the remaining capacity vector elements $\mathbf{b}^{p,c}$ for any pairs $p \in P$ which share either a source or a destination with p_{chosen}. The pseudocode for this vectorised packer is summarised in Algorithm 4.

7.3 Experimental Setup

TrafPy enables the production of custom traffic distributions through the control of a handful of parameters. These include the flow size and inter-arrival time distribution parameters, the node distribution's inter- vs. intra-rack and skew

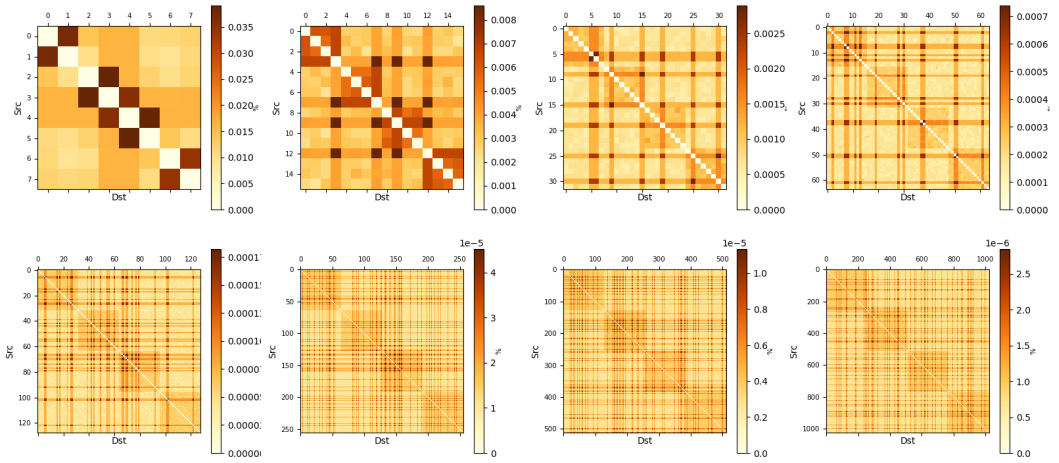


FIGURE 7.2: Custom traffic matrix distributions generated with 8, 16, 32, 64, 128, 256, 512, and 1024 nodes, where the colour of each source-destination pair corresponds to the fraction of the overall network load it requests.

node fractions, and the overall network load. To measure the packing times for the original and vectorised packing algorithms, we generated an assortment of custom traffic patterns typical for a 'university' data centre[2] as detailed in Chapter 6 for networks with 4 racks and $|N| = \{8, 16, 32, 64, 128, 256, 512, 1024\}$ nodes (see Fig. 7.2). We assumed an optical data centre network with with an overall network load rate of 50%. For each traffic matrix, we generated $|F| = 5 \cdot |N|^2$ flows to ensure non-sparse packing. Each packing algorithm was ran on a shared cluster with an Intel Xeon ES-2660 CPU across 4 seeds to ensure reliable packing times given the variance in use of the shared cluster, with the 95% confidence interval bands plotted for any metrics recorded.

7.4 Results & Discussion

Fig. 7.1b shows the packing times taken by the original and the vectorised packers when generating the distributions shown in Fig. 7.2. The vectorised packer achieved a $\approx 38\times$ speed-up over the original packer on the $|N| = 264$

[2]University data centres service applications such as database backups, distributed file system hosting, and multicast video streaming, with $\approx 70\%$ of traffic being inter-rack and $\approx 20\%$ of nodes requesting $\approx 55\%$ of the traffic load.

traffic matrix and $\approx 70\times$ on the $|N|$ =1024 matrix. Although the vectorised algorithm was slightly slower than the original packer on the smallest $|N| = 8$ network due to performing a `where` vector operation on all pairs, the absolute generation time was still $O(s)$ and this additional overhead quickly becomes negligible across $|N| > 8$ networks.

To verify that our proposed vectorised packing algorithm was generating the same node distributions as the original packer used in Chapter 6, we measured the JSD between the target and the generated node distributions for each algorithm (see Fig. 7.1c). As expected, both packers deterministically reach the same solution, but the Jensen-Shannon distance will not be exactly 0 for either due to the incompatibility between the 50% network load and the skewed target distribution (see Chapter 6 for more details).

7.5 Conclusions, Limitations, & Further Work

In conclusion, we have proposed a flow source-destination pair assignment algorithm which makes novel use of vector array operations to achieve orders of magnitude faster traffic generation times than the original algorithm used in Chapter 6 when generating custom traffic matrices. This work significantly improves the utility of an open source traffic generation framework in order to aid the production of high-fidelity traffic patterns and to test and develop network systems at scale. Here we outline the limitations of this chapter and areas of further work.

GPU implementation. The fundamental insight of this chapter is that the traffic generation task can be framed as a series of tensor operations. GPUs are particularly good at parallelising tensor operations, therefore implementing the algorithm proposed in this chapter on a GPU would likely improve the traffic generation speed by several factors and enable additional scalability. This

might enable traffic generation for networks with $> O(10^4)$ nodes without any adjustment needed to the generation algorithm.

Analysing network system scalability. With the ability to generate traffic for $O(10^3)$ node networks, an interesting project would be to take previously proposed network systems, such as the scheduler of Benjamin et al. [2021], and see whether the reported performance can be retained at scale.

Chapter 8

Afterword: Conclusions, Limitations, & Further Work

This thesis has motivated, proposed, and investigated a number of challenges facing the realisation of next-generation computer networks, with a particular focus on AI-driven optical solutions. These included AI methods to switch all-optical SOA switches on the sub-ns timescales required for a practical optical data centre, establishing a new switch speed record in Chapter 3. The thesis also proposed how key optical cluster resource management challenges, such as how much to partition computational jobs in Chapter 5, can be addressed automatically by AI methods which learn to optimise key user-defined performance criteria. Moreover, consideration was given as to how to solve generic NP-hard discrete optimisation problems such as those found in all manner of orchestration and physical layer computer network components by proposing a new RL-based branching algorithm which achieved state-of-the-art results in Chapter 4. Chapters 6 and 7 are the first to propose a general traffic generation framework for standardising DCN system testing and benchmarking, which will help future researchers to output reproducible, cross-validated, and performant novel ideas.

While this is good progress, there are many outstanding challenges which remain, and specific areas for further work have been outlined at the end of each chapter. An overarching theme of further work is to implement the ideas proposed

in this thesis in a practical laboratory setting. With a real implementation of an OCS network using the AI-driven SOA switching methodology of Chapter 3 and the resource management schemes proposed in Chapters 4 and 5, the practical efficacy with which the network could process information could be evaluated using the TrafPy-generated traffic of Chapters 6 and 7. Once verified with TrafPy, a small optical HPC architecture, such as RAMP [Ottino et al., 2022] from Chapter 5, could be implemented and augmented with the methods proposed in this thesis, and a real DNN model could be trained to demonstrate the practical benefits of an AI-driven OCS computer network.

Furthermore, this thesis has not comprehensively evaluated the robustness and resilience of the proposed AI methods to different scenarios in the real world. Methods which formally verify neural network performance [Tjeng et al., 2017, Albarghouthi, 2021] would be an interesting research direction.

Moreover, computer network practitioners may be reluctant to adopt new AI methods whose policies they do not fully understand. Further model interpretability research [Rudin et al., 2021] may be crucial for adoption.

These additional investigations will undoubtedly present formidable challenges. However, they will also offer the potential to realise computer network systems with dramatically improved processing power and, ultimately, unleash the next generation of big data jobs, from AI and genome sequencing to the internet of things and large-scale data science.

Appendix A

Solving NP-Hard Discrete Optimisation Problems

A.1 RL Training

A.1.1 Training Parameters

The RL training hyperparameters are summarised in Table A.1. We used n-step DQN [Sutton, 1988, Mnih et al., 2013] with prioritised experience replay [Schaul et al., 2016], with overviews of each of these approaches provided in Section 2.12. For exploration, we followed an ϵ-stochastic policy ($\epsilon \in [0, 1]$) whereby the probabilities for action selection were ϵ for a random action and $1 - \epsilon$ for an action sampled from the softmax probability distribution over the Q-values of the branching candidates. We also found it helpful for learning stability to clip the gradients of our network before applying parameter updates.

A.1.2 Training Time and Convergence

To train our RL agent, we had a compute budget limited to one A100 GPU which was shared by other researchers from different groups. This resulted in highly variable training times. On average, one epoch on the large 500×1000 set covering instances took roughly 0.42 seconds (which includes the time to act in the B&B environment to collect and save the experience transitions, sample from

Training Parameter	Value
Batch size	64 (128)
Actor steps per learner update	5 (10)
Learning rate	5e−5
Discount factor	0.99
Optimiser	Adam
Buffer size $\lvert\mathcal{M}\rvert_{\text{init}}$	20e3
Buffer size $\lvert\mathcal{M}\rvert_{\text{capacity}}$	100e3
Prioritised experience replay β_{init}	0.4
Prioritised experience replay β_{final}	1.0
$\beta_{\text{init}} \rightarrow \beta_{\text{final}}$ learner steps	5e3
Prioritised experience replay α	0.6
Minimum experience priority	1e−3
Soft target network update τ_{soft}	1e−4
Gradient clip value	10
n-step DQN n	3
Exploration probability ϵ	2.5e−2

TABLE A.1: Training parameters used for training the RL agent. All parameters were kept the same across CO instances except for the large 500×1000 set covering instances, which we used a larger batch size and actor steps per learner update (specified in brackets).

the buffer, make online vs. target network predictions, update the network, etc.). Therefore training for 200k epochs (roughly the amount needed to converge on a strong policy within $\approx 20\%$ of the imitation agent) took 5-6 days.

As shown in Fig. A.1, when we left our retro branching agent to train for $\approx$ 13 days ($\approx$ 500k epochs), although most performance gains had been made in the first $\approx$ 200k epochs, the agent never stopped improving (the last improved checkpoint was at 485k epochs). A potentially promising next step might therefore be to increase the compute budget of our experiments by distributing retro branching across multiple GPUs and CPUs and see whether or not the agent does eventually match or exceed the 500×1000 set covering performance of the IL agent after enough epochs.

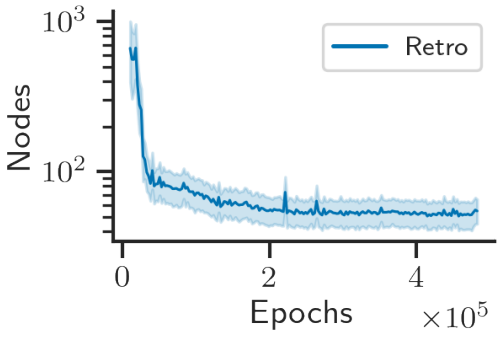


FIGURE A.1: Validation curve for the retro branching agent on the 500×1000 set covering test instances. Although most performance gains were made in the first $\approx$ 200k epochs, the agent did not stop improving, with the last recorded checkpoint improvement at 485k epochs.

A.2 Neural Network

A.2.1 Architecture

We used the same GCN architecture as Gasse et al. 2019 to parameterise our DQN value function with some minor modifications which we found to be helpful. Firstly, we replaced the ReLU activations with Leaky ReLUs which we inverted in the final readout layer in order to predict the negative Q-values of our MDP. Secondly, we initialised our linear layer weights and biases with a normal distribution ($\mu = 0, \sigma = 0.01$) and all-zeros respectively, and our layer normalisation weights and biases with all-ones and all-zeros respectively. Thirdly, we removed a network forward pass in the bipartite graph convolution message passing operation which we found to be unhelpfully computationally expensive. For clarity, Fig. A.2 shows the high-level overview of the neural network architecture. For a full analysis of the benefit of using GCNs for learning to branch, refer to Gasse et al. 2019.

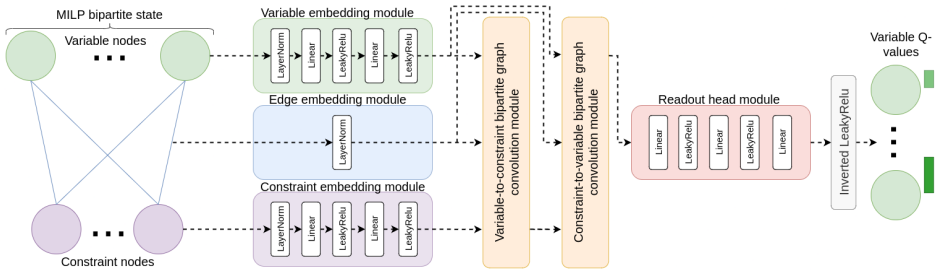


FIGURE A.2: Neural network architecture used to parameterise the Q-value function for our ML agents, taking in a bipartite graph representation of the MILP and outputting the predicted Q-values for each variable in the MILP.

A.2.2 Inference & Solving Times

The key performance criterion to optimise for any branching method is the reduction of the overall B&B solving time. However, accurate and precise solving time and primal-dual integral over time comparisons are difficult because they are hardware-dependent. This is particularly problematic in research settings where CPU/GPU resources are often shared between multiple researchers and therefore hardware performance (and consequently solving time) significantly varies. Consequently, as in other works Khalil et al. [2016], **?**], Etheve et al. [2020], we presented and optimised for the number of B&B tree nodes as this is hardware-independent and, in the context of prior work, can be used to infer the solving time.

Specifically, we use the same GCN-based architecture of Gasse et al. 2019 for all ML branchers, thus all ML approaches have the same per-step inference cost. Therefore the relative difference in the number of tree nodes is exactly the relative wall-clock times on equal hardware. When the per-step inference process is different (as for our non-ML baselines, such as SB), the number of tree nodes is not an adequate proxy for solving time. However, Gasse et al. 2019 have already demonstrated that the GCN-based branching policies of IL outperform the solving time of other branchers such as SB. As this ML speed-up has already been established, in this chapter we focus on improving the per-step ML decision quality using RL rather than further optimising network

architecture, or otherwise, for speed, which we leave to further work.

However, empirical solving times are of interest to the broader optimisation community. Therefore, Table A.2 provides a summary of the solving times of the branching agents on the large 500 × 1000 set covering instances under the assumption that they were ran on the same hardware as Gasse et al. 2019.

Method	Solving time (s)
SB	33.5
IL	2.1
Retro	2.5
FMSTS-DFS	12.2
FMSTS	7.6
Original	35.8

TABLE A.2: Inferred mean solving times of the branching agents on the large 500 × 1000 set covering instances under the assumption that they were ran on the same hardware as Gasse et al. 2019.

A.3 Data Set Size Analysis

As described in Section 4.5, we used 100 MILP instances unseen during training to evaluate the performance of each branching agent. This is in line with prior works such as Khalil et al. 2016 who used 84 instances and Gasse et al. 2019 who used 20. To ensure that 100 instances are a large enough data set to reliably compare branching agents, we also ran the agents on 1000 large 500 × 1000 set covering instances. The relative performance of each branching agent was approximately the same as when evaluated on 100 instances, with Retro scoring 65.3 nodes, FMSTS 250 (3.8× worse than Retro), IL 55.4 (17.8% better than Retro), and SB 43.3. In the interest of saving evaluation time and hardware demands and to make the development of and comparison to our work by future research projects more accessible, as well as for clarity in the per-instance Retro-IL comparison of Fig. 4.3d, we report the results for 100 evaluation instances in

the main chapter in the knowledge that the relative performances are unchanged as we scale the data set to a larger size.

A.4 SCIP Parameters

For all non-DFS branching agents we used the same SCIP 2022 B&B parameters as Gasse et al. 2019, as summarised in Table A.3.

SCIP Parameter	Value
`separating/maxrounds`	0
`separating/maxroundsroot`	0
`limits/time`	3600

Table A.3: Summary of the SCIP 2022 hyperparameters used for all non-DFS branching agents (any parameters not specified were the default SCIP 2022 values).

A.5 Observation Features

We found it useful to add 20 features to the variable nodes in the bipartite graph in addition to the 19 features used by Gasse et al. 2019. These additional features are given in Table A.4; their purpose was to help the agent to learn to aggregate over the uncertainty in the future primal-dual bound evolution caused by the partially observable activity occurring in sub-trees external to its retrospectively constructed trajectory.

A.6 FMSTS Implementation

Etheve et al. [2020] did not open-source any code, used the paid commercial CPLEX [2009] solver, and experimented with proprietary data sets. Furthermore, they omitted comparisons to any other ML baseline such as Gasse et al. [2019], further limiting their comparability. However, we have done a 'best effort' implementation of the relatively simple FMSTS algorithm, whose core idea is to

Variable Feature	Description
db_frac_change	Fractional dual bound change
pb_frac_change	Fractional primal bound change
max_db_frac_change	Maximum possible fractional dual change
max_pb_frac_change	Maximum possible fractional primal change
gap_frac	Fraction primal-dual gap
num_leaves_frac	# leaves divided by # nodes
num_feasible_leaves_frac	# feasible leaves divided by # nodes
num_infeasible_leaves_frac	# infeasible leaves divided by # nodes
num_lp_iterations_frac	# nodes divded by # LP iterations
num_siblings_frac	Focus node's # siblings divided by # nodes
is_curr_node_best	If focus node is incumbent
is_curr_node_parent_best	If focus node's parent is incumbent
curr_node_depth	Focus node depth
curr_node_db_rel_init_db	Initial dual divided by focus' dual
curr_node_db_rel_global_db	Global dual divided by focus' dual
is_best_sibling_none	If focus node has a sibling
is_best_sibling_best_node	If focus node's sibling is incumbent
best_sibling_db_rel_init_db	Initial dual divided by sibling's dual
best_sibling_db_rel_global_db	Global dual divided by sibling's dual
best_sibling_db_rel_curr_node_db	Sibling's dual divided by focus' dual

TABLE A.4: Descriptions of the 20 variable features we included in our observation in addition to the 19 features used by Gasse et al. 2019.

set the Q-function of a DQN agent as minimising the sub-tree size rooted at the current node and to use a DFS node selection heuristic. To replicate the DFS setting of Etheve et al. [2020] in SCIP [2022], we used the parameters shown in Table A.5. We will release the full re-implementation to the community along with our own code.

SCIP Parameter	Value
`separating/maxrounds`	0
`separating/maxroundsroot`	0
`limits/time`	3600
`nodeselection/dfs/stdpriority`	1 073 741 823
`nodeselection/dfs/memsavepriority`	536 870 911
`nodeselection/restartdfs/stdpriority`	−536 870 912
`nodeselection/restartdfs/memsavepriority`	−536 870 912
`nodeselection/restartdfs/selectbestfreq`	0
`nodeselection/bfs/stdpriority`	−536 870 912
`nodeselection/bfs/memsavepriority`	−536 870 912
`nodeselection/breadthfirst/stdpriority`	−536 870 912
`nodeselection/breadthfirst/memsavepriority`	−536 870 912
`nodeselection/estimate/stdpriority`	−536 870 912
`nodeselection/estimate/memsavepriority`	−536 870 912
`nodeselection/hybridestim/stdpriority`	−536 870 912
`nodeselection/hybridestim/memsavepriority`	−536 870 912
`nodeselection/uct/stdpriority`	−536 870 912
`nodeselection/uct/memsavepriority`	−536 870 912

TABLE A.5: Summary of the SCIP 2022 hyperparameters used the DFS FMSTS branching agent of Etheve et al. 2020 (any parameters not specified were the default SCIP 2022 values).

A.7 Pseudocode

A.7.1 Retrospective Trajectory Construction

Algorithm 5 shows the proposed 'retrospective trajectory construction' method, whereby fathomed leaf nodes not yet added to a trajectory are selected as the brancher's terminal states and paths to them are iteratively established using some construction method.

Algorithm 5 Retrospectively construct trajectories.

Input: B&B tree $\mathcal{T}$ from solving MILP
Output: Retrospectively constructed trajectories
Initialise: nodes_added, subtree_episodes = [$\mathcal{T}_{\text{root}-1}$], []
// Construct trajectories until all valid node(s) in $\mathcal{T}$ added
while True **do**
  // Root trajectories at highest level unselected node(s)
  subtrees = []
  for node in nodes_added **do**
    for child_node in $\mathcal{T}_{\text{node}}$.children **do**
      if child_node not in nodes_added **then**
        // Use depth-first-search to get sub-tree
        subtrees.append(dfs($\mathcal{T}$, root=child_node))
      end if
    end for
  end for
  // Construct trajectory episode(s) from sub-tree(s)
  if len(subtrees) > 0 **then**
    for subtree in subtrees **do**
      subtree_episode = construct_path(subtree) (6)
      subtree_episode[-1].done = True
      subtree_episodes.append(subtree_episode)
      for node in subtree_episode **do**
        nodes_added.append(node)
      end for
    end for
  else
    // All valid nodes in $\mathcal{T}$ added to a trajectory
    break
  end if
end while

A.7.2 Maximum Leaf LP Gain

Algorithm 6 shows the proposed 'maximum leaf LP gain' trajectory construction method, whereby the fathomed leaf node with the greatest change in the dual bound ('LP gain') is used as the terminal state of the trajectory.

Algorithm 6 Maximum leaf LP gain trajectory construction.

Input: Sub-tree $\mathcal{S}$
Output: Trajectory $\mathcal{S}_E$
Initialise: gains = {}
for leaf in $\mathcal{S}$.leaves **do**
  if leaf closed by brancher **then**
    gains.leaf = $|\mathcal{S}_{\text{root}}.\text{dual\_bound} - \mathcal{S}_{\text{leaf}}.\text{dual\_bound}|$
  end if
end for
terminal_node = max(gains)
$\mathcal{S}_E$ = shortest_path(source=$\mathcal{S}_{\text{root}}$, target=terminal_node)

A.8 Cost of Strong Branching Labels

As well as performance being limited to that of the expert imitated, IL methods have the additional drawback of requiring an expensive data labelling phase. Fig. A.3 shows how the explore-then-strong-branch labelling scheme of Gasse et al. 2019 scales with set covering instance size (rows × columns) and how this becomes a hindrance for larger instances. Although an elaborate infrastructure can be developed to try to label large instances at scale [Nair et al., 2021], ideally the need for this should be avoided; a key motivator for using RL to branch.

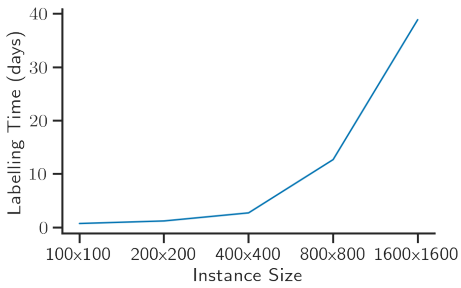


FIGURE A.3: How the explore-then-strong-branch data labelling phase of the strong branching imitation agent scales with set covering instance size (rows × columns) using an Intel Xeon ES-2660 CPU and assuming 120 000 samples are needed for each set.

Appendix B

A Framework for Generating Custom and Reproducible Synthetic Traffic

B.1 Table of Notation

B.2 TrafPy Distribution Parameters

TABLE B.2: Benchmark categories with their real traffic characteristics reported in the literature (where appropriate) and the corresponding TrafPy parameters D' needed to reproduce the distributions.

$\text{DCN}_{<i,ii,iii,iv>} \rightarrow$ <*university, private_enterprise, commercial_cloud, social_media_cloud*>

$\text{Skewed}_{<i,ii,iii,iv,v>} \rightarrow$ skewed_nodes_sensitivity_<*uniform, 0.05, 0.1, 0.2, 0.4*>

$\text{Rack}_{<i,ii,iii,iv,v>} \rightarrow$ rack_sensitivity_<*uniform, 0.2, 0.4, 0.6, 0.8*>

[a] Real traffic characteristics reported in the literature.

[b] Corresponding TrafPy parameters D'.

c = net.graph['rack_to_ep_dict'] → Network cluster (i.e. rack) configuration.

$d(u)$ = int(u * len(net.graph['endpoints'])) → Number of nodes to skew.

$e(u, v)$ = [$v/d(u)$ for _ in range($d(u)$)] → Fraction of overall traffic load to distribute amongst the skewed nodes.

r | r_d | p | n_s | n_p = rack_prob_config | 'racks_dict' | 'prob_inter_rack' | num_skewed_nodes | skewed_node_probs

Benchmark Category	Applications	Size, Bytes	Inter-arrival Time, μs	Inter- | Intra-Rack Traffic, %	Hot Nodes | Load Requested, %
DCN$_i$ Benson et al. [2010a], Benson et al. [2011]	Database backups, hosting distributed file systems (email, servers, web services for faculty portals etc.), multi-cast video streams	[a] 80% < 10,000 [b] 'lognormal', {μ: 7, σ: 2.5}, min_val=1, max_val=2e7, round=25	[a] 10% < 400, 80% < 10,000 [b] 'weibull', {α: 0.9, λ: 6,000}, min_val=1, round=25	[a] 70 | 30 [b] r={r_d: c, p: 0.7 }	[a]20|55[b]'multimodal', n_s=d(0.2), n_p=e(0.2, 0.55)

DCN$_{ii}$ **Benson et al. [2010a]**	University + 'custom' applications and development test beds	[a] 80% < 10,000 [b] 'lognormal', {μ: 7, σ: 2.5}, min_val=1, max_val=2e7, round=25	[a] 80% < 1,000 [b] 'multimodal', min_val=1, max_-val=100,000, locations=[40,1], skews=[-1,4], scales=[60,1000], num_skew_-samples =[10]e3, round=25, bg_factor=0.05	[a] 50 | 50 [b] r={r_d: c, p: 0.5}	[a]20|55[b]'multimodal', n_s=d(0.2), n_p=e(0.2, 0.55)
DCN$_{iii}$ **Benson et al. [2010a], Kandula et al. [2009]**	Internet-facing applications (search indexing, webmail, video, etc.), data mining and MapReduce-style applications	[a] 80% < 10,000 [b] 'lognormal', {μ: 7, σ: 2.5}, min_val=1, max_val=2e7, round=25	[a] Median 10 [b] 'multimodal', min_val=1, max_-val=100,000, locations =[10,20,100,1], skews=[0,0,0,100], scales=[1,3,4,50], num_skew_-samples =[10,7,5,20]e3, round=25, bg_factor=0.01	[a] 20 | 80 [b] r={r_d: c, p: 0.2}	[a]20|55[b]'multimodal', n_s=d(0.2), n_p=e(0.2, 0.55)
DCN$_{iv}$ **Roy et al. [2015]**	Web request response generation (mail, messenger, etc.), MySQL database storage & cache querying, newsfeed assembly	[a] 10% < 300, 90% < 100, 000 [b] 'weibull', {α: 0.5, λ: 21,000}, min_val=1, max_val=2e6, round=25	[a] 10% < 20, 90% < 10, 000 [b] 'lognormal', {μ: 6, σ: 2.3}, min_val=1, round=25	[a] 12.9 | 87.1 [b] r={r_d: c, p: 0.129}	[a]20|55[b]'multimodal', n_s=d(0.2), n_p=e(0.2, 0.55)
Skewed$_i$, **Rack**$_i$	-	[b]DCN$_{iii}$	[b]DCN$_{iii}$	[b]'uniform', r = None	[b]'uniform' $n_s = n_p =$ None
Skewed$_{ii}$	-	[b]DCN$_{iii}$	[b]DCN$_{iii}$	[b]'uniform', r = None	5|55[b]'uniform' $n_s = d(0.05)$ $n_p =$ $e(0.05, 0.55)$
Skewed$_{iii}$	-	[b]DCN$_{iii}$	[b]DCN$_{iii}$	[b]'uniform', r = None	5|55[b]'uniform' $n_s = d(0.1)$ $n_p =$ $e(0.1, 0.55)$
Skewed$_{iv}$	-	[b]DCN$_{iii}$	[b]DCN$_{iii}$	[b]'uniform', r = None	5|55[b]'uniform' $n_s = d(0.2)$ $n_p =$ $e(0.2, 0.55)$
Skewed$_{v}$	-	[b]DCN$_{iii}$	[b]DCN$_{iii}$	[b]'uniform', r = None	5|55[b]'uniform' $n_s = d(0.4)$ $n_p =$ $e(0.4, 0.55)$
Rack$_{ii}$	-	[b]DCN$_{iii}$	[b]DCN$_{iii}$	80|20[b]'uniform', $r = \{r_d : c, p : 0.8\}$	[b]'uniform' $n_s = n_p =$ None

$\mathbf{Rack}_{iii}$	-	${}^{b}\mathrm{DCN}_{iii}$	${}^{b}\mathrm{DCN}_{iii}$	$60\|40^{b}$'uniform', $r = \{r_d : c, p : 0.6\}$	^{b}'uniform' $n_s = n_p =$ None
$\mathbf{Rack}_{iv}$	-	${}^{b}\mathrm{DCN}_{iii}$	${}^{b}\mathrm{DCN}_{iii}$	$40\|60^{b}$'uniform', $r = \{r_d : c, p : 0.4\}$	^{b}'uniform' $n_s = n_p =$ None
$\mathbf{Rack}_{v}$	-	${}^{b}\mathrm{DCN}_{iii}$	${}^{b}\mathrm{DCN}_{iii}$	$20\|80^{b}$'uniform', $r = \{r_d : c, p : 0.2\}$	^{b}'uniform' $n_s = n_p =$ None

Symbol	Definition
D'	Set of parameters defining the TrafPy distributions
D	Traffic trace generated using the D' TrafPy parameters
$\mathbb{P}$	Probability distribution
X	Discrete random variables
H	Entropy
JSD	Jensen-Shannon divergence
$\sqrt{\text{JSD}}$	Jensen-Shannon distance
$\{\pi_1, ..., \pi_n\}$	Weightings for the JSD of n distributions
B^s, B^t, B^n	Flow size, inter-arrival time, and node pair random variables for benchmark workload B
$\boldsymbol{b^s}, \boldsymbol{b^t}, \boldsymbol{b^n}$	Flow sizes, inter-arrival times, and node pairs sampled from benchmark workload B
$\boldsymbol{b^a}$	Flow arrival times derived from inter-arrival times $\boldsymbol{b^t}$
T	DCN network topology
ρ	Load fraction (fraction of overall network capacity requested)
n_n	Number of server nodes
n_c	Number of channels per communication link
C_c	Capacity per server node link channel
C_t	Total network capacity per direction
n_f	Number of flows generated
t_t	Total time duration of simulation
ϱ	Load rate (information arriving per unit time)
α_t	Inter-arrival time adjustment factor
d_p	Difference between a node pair's current and target information request magnitude
β	Number of flows adjustment factor
R	Number of traffic traces to generate and simulate for a suitable confidence interval

Table B.1: Table summarising the symbol notation used throughout the paper.

Table B.3: Flow size, inter-arrival time, and node load distribution characteristics for the University (U), Private Enterprise (PE), Commercial Cloud (CC), and Social Media Cloud (SMC) data sets of the DCN benchmark after generating the distributions from TrafPy parameters D'.

Variable	DCN	# Modes	Min.	Max.	Mean	Variance	Skewness	Kurtosis
Size (B)	U	1	1	19.8e6	22.9e3	42e9	39.4	2.41e3
	PE	1	1	19e6	23.3e3	53.5e9	44.1	2.79e3
	CC	1	1	19.2e6	22.3e3	38.4e9	36.9	2.08e3
	SMC	1	1	3.17e6	42e3	8.87e9	6.20	66.4
Inter-arrival time (μs)	U	1	1	126e3	6.3e3	49.9e6	2.44	9.92
	PE	2	1	100e3	2.83e3	154e6	5.7	33.1
	CC	4	1	10e3	84.5	0.32e6	13	179
	SMC	1	1	54.6e5	5.51e3	2.11e9	47.8	3.75e3

Variable	DCN	% Hot Nodes	% Hot Node Traffic	% Inter-Rack
Node load distribution (%)	U	20	55	70
	PE	20	55	50
	CC	20	55	20
	SMC	20	55	12.9

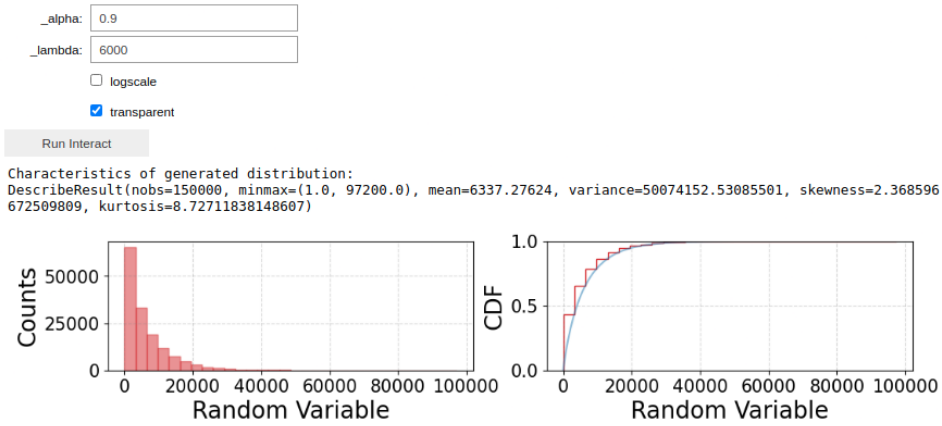


FIGURE B.1: Output of example code for interactively and visually shaping a 'named' distribution in a Jupyter Notebook.

B.3 TrafPy API Examples

B.3.1 Custom Distribution Shaping

Interactively & Visually Shaping a Custom 'Named' Distribution in a Jupyter Notebook. Example of interactively and visually shaping a weibull distribution's parameters to achieve a target distribution for some random variable in Jupyter Notebook (output in Fig. B.1):

```

import trafpy.generator as tpg

dist = tpg.gen_named_val_dist(dist='weibull',
                              interactive_plot=True,
                              min_val=1,
                              max_val=None,
                              size=15e4)


```

This same distibution can then be reproduced by using the same parameters:

```

dist = tpg.gen_named_val_dist(dist='weibull',
                              params={'_alpha': 0.9, '_lambda': 6000}
                              min_val=1,
                              max_val=None)

```

Interactively & Visually Shaping a Custom 'Multimodal' Distribution in a Jupyter Notebook. To generate a multimodal distribution, first shape each mode individually (output in Fig. B.2):

```

import trafpy.generator as tpg

data_dict = tpg.gen_skew_dists(min_val=1,
                               max_val=1e5,
                               num_modes=2)

```

Then combine the distributions, filling the distribution with a tuneable amount of 'background noise' (output in Fig. B.3):

```

multimodal_dist = tpg.combine_multiple_mode_dists(data_dict,
                                                  min_val=1,
                                                  max_val=1e5)

```

This same distribution can be reproduced using the same parameters:

```

multimodal_dist = tpg.gen_multimodal_val_dist(min_val=1,
                                              max_val=1e5,
                                              locations=[40, 1],
                                              skews=[-1, 4],
                                              scales=[60, 1000],
                                              num_skew_samples=[1e4, 1e4],
                                              bg_factor=0.05)

```

N.B. An equivalent function can be used for generating custom skew distributions with a single mode which also do not fall under one of the canonical 'named' distributions.

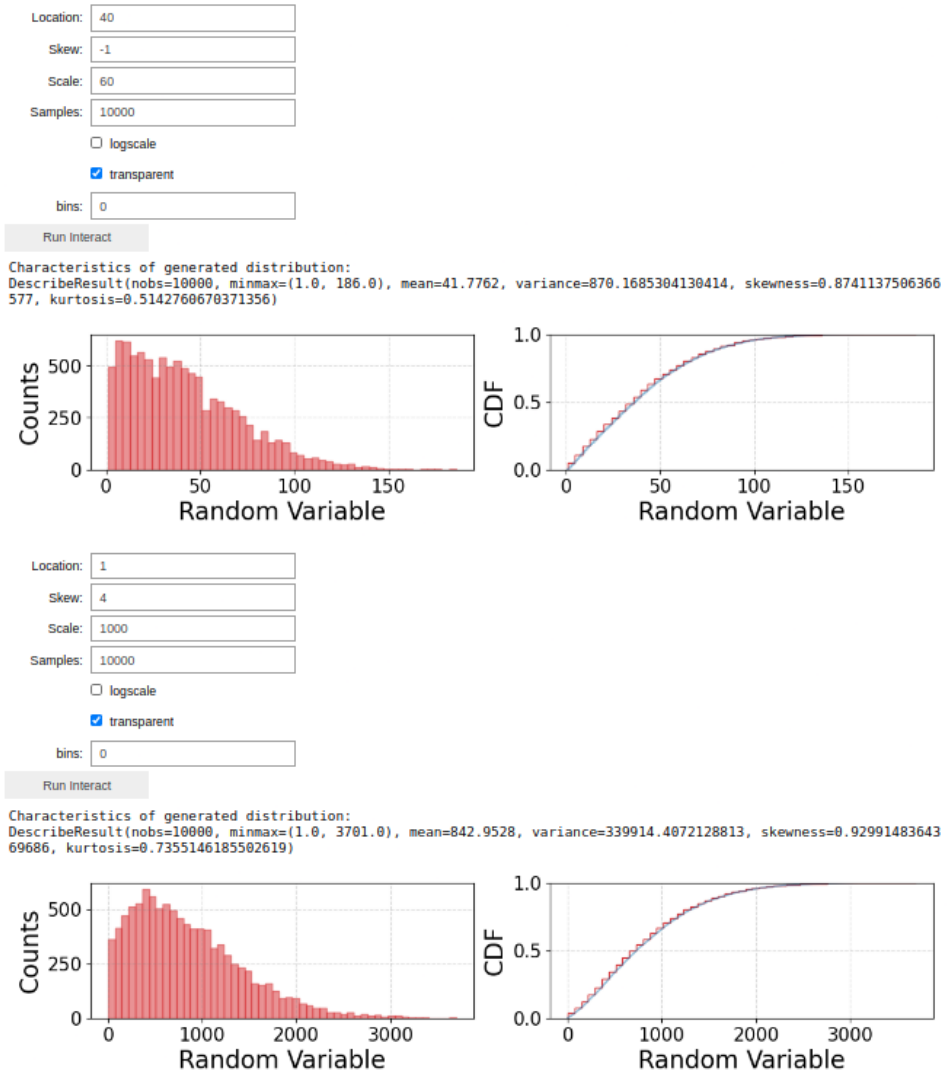


FIGURE B.2: Output for step 1 of example code for interactively and visually shaping a 'multimodal' distribution in a Jupyter Notebook, where you must first shape each mode individually.

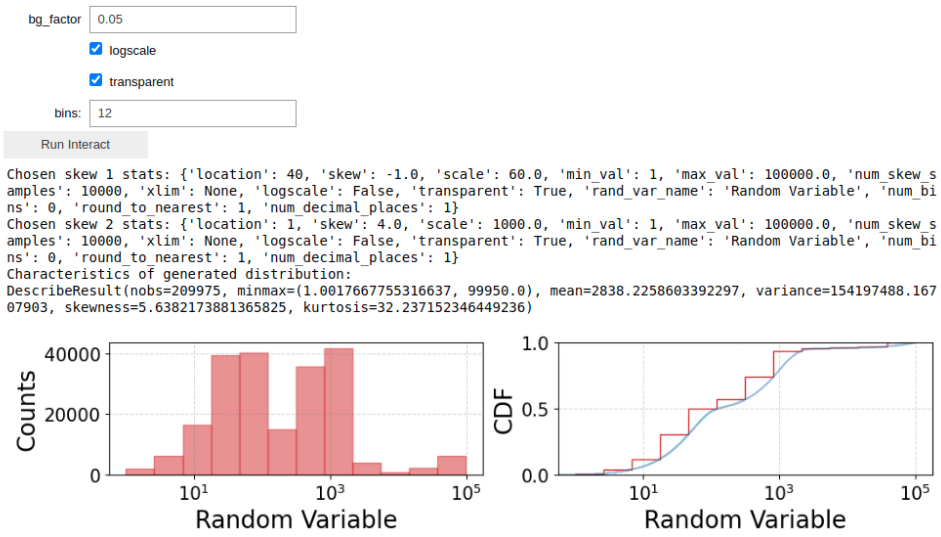


FIGURE B.3: Output for step 2 of example code for interactively and visually shaping a 'multimodal' distribution in a Jupyter Notebook, where you must combine your individually shaped modes into a single distribution.

B.3.2 Benchmark Importing & Flow Generation

Example code for generating and visualising a load 0.1 University benchmark data set of flows for a custom topology (output in Fig. B.4):

```
1
2  import trafpy.generator as tpg
3  from trafpy.benchmarker import BenchmarkImporter
4  from trafpy.generator import Demand, DemandsAnalyser, DemandPlotter
5
6  # set variables
7  min_duration = 1000
8  jsd_threshold = 0.1
9
10 # initialise network
11 net = tpg.gen_arbitrary_network(num_eps=64, ep_channel_capacity=1250)
12
13 # initialise benchmark distributions
14 importer = BenchmarkImporter(benchmark_version='0.01')
15 dists = importer.get_benchmark_dists(benchmark='university', eps=net.graph['endpoints'])
16
17 # generate flow-centric demand data set
18 network_load_config = {'network_rate_capacity': net.graph['max_nw_capacity'],
19                        'ep_channel_capacity': net.graph['ep_channel_capacity'],
20                        'target_load_fraction': 0.1}
21 flow_centric_demand_data = tpg.create_demand_data(eps=net.graph['endpoints'],
22                                                   node_dist=dists['node_dist'],
23                                                   flow_size_dist=dists['flow_size_dist'],
24                                                   interarrival_time_dist=dists['interarrival_time_dist'],
25                                                   network_load_config=network_load_config,
26                                                   jsd_threshold=jsd_threshold,
27                                                   min_duration=min_duration)
28
29 # print summary table
30 demand = Demand(flow_centric_demand_data, net.graph['endpoints'])
31 DemandsAnalyser([demand], net).compute_metrics(print_summary=True)
32
33 # visualise distributions
34 plotter = DemandPlotter(demand)
35 plotter.plot_flow_size_dist()
36 plotter.plot_interarrival_time_dist()
37 plotter.plot_node_dist()
38
```

	Name	Flows	1st	Last	Duration	Info	Load	Smallest	Largest
0	demand	135916	0.0	788907.126500	788907.126500	3.155629e+09	4000.0	1.0	17236750.0

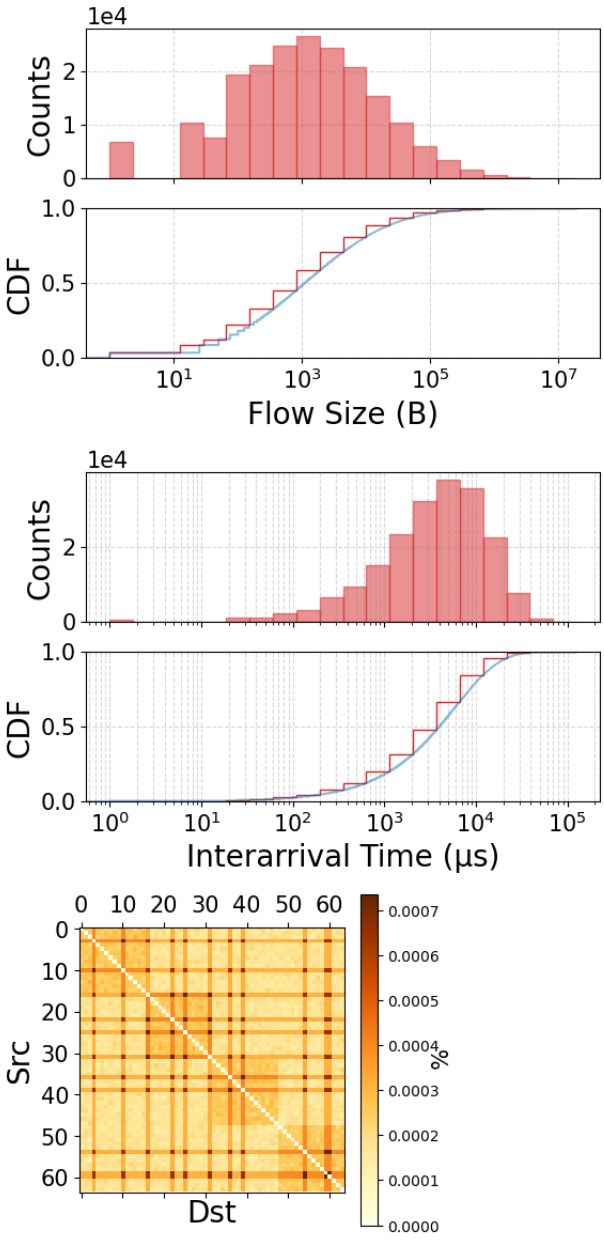


FIGURE B.4: Output of example code for generating a benchmark.

B.4 Pseudocode

B.4.1 Scheduling

The flow scheduling pseudocode is shown in Algorithm 7. First, information about the queued flows such as their characteristics (packets left, time of arrival, flow queue, destination node, etc.), the network links requested in the source-destination path, and the bandwidth requested, is collected. If the scheduler uses cost-based scheduling (e.g. SRPT uses flow completion time cost), a cost is also assigned to each flow. Next, for each link being requested by the flows, while the link in question has some available bandwidth left to allocate for the current time slot, the scheduler chooses flows until either there is no bandwidth left or there are no flows demanding the link which have not been chosen. Finally, for each flow in the set of these provisionally chosen flows, the smallest number of packets scheduled for the flow in question across all links is chosen as the flow's number of packets to schedule. Note that this simulation methodology considers bandwidth bottlenecks throughout all layers of the network. The pseudocode in Algorithm 8 is used to resolve any contentions and attempt to set up the flow, thus adding the flow to the ultimate set of flows chosen by the scheduler for the given time slot. The parts which are **scheduler**-specific have been marked in bold.

B.4.2 TrafPy Benchmark Protocol

Algorithm 7 Flow scheduling process.

```
Collect flow information
link_allocations = []
for link in links do
    while link bandwidth ≠ 0 do
        link_allocations.append(scheduler choose flow)
    end while
end for
chosen_flows = []
for flow in flows do
    if flow in link_allocations then
        flow_packets = min(packets allocated for flow in link_allocations)
        establish, removed_flows = scheduler resolve_contentions(flow, chosen_flows)
        if establish then
            chosen_flows.append(flow)
            chosen_flows.remove(removed_flows)
        end if
    end if
end for
```

Algorithm 8 Flow contention resolution process.

```
Require: flow, chosen_flows
removed_flows = []
while True do
    if no_contention(flow) then
        establish = True
        return establish, removed_flows
    else
        contending_flow = find_contending_flow()
        establish = scheduler resolve_contention(flow, contending_flow)
        if not establish then
            chosen_flows.append(removed_flows)
            return establish
        else
            chosen_flows.remove(contending_flow)
            continue
        end if
    end if
end while
```

Algorithm 9 TrafPy benchmark protocol.

```
for r in range(R) do
    for d in D do
        for ρ ← 0.1 to 0.9 step 0.1 do
            P_KPI = ϒ(χ, d, ρ)
        end for
    end for
end for
```

B.5 Traffic Skew Convergence

A constraint of any traffic matrix is that the load on each end point (the fraction of the end point's capacity being requested) cannot exceed 1.0. Consequently, certain traffic skews become infeasible at higher loads (for example, it is impossible for an $n > 1$ network to have 1 node requesting 100% of the traffic if the overall network is under a 1.0 load). As shown in Fig. 6.3, this results in all traffic matrices tending towards uniform (i.e. having no skew) as the overall network load tends to 1.0.

A question traffic trace generators may ask is: for a given load, what combination of i) number of skewed nodes, ii) corresponding fraction of the arriving network traffic the skewed nodes request, and iii) overall network load results in the traffic matrix being skewed or not skewed? To answer this question, we make the following assumptions:

- All network end points have equal bandwidth capacities.
- All end points are either 'skewed' or 'not skewed' by the same amount.
- 'Skew' is defined by a *skew factor*, which is the fractional difference between the load rate per skewed node and the load rate per non-skewed node (the highest being the numerator, and the lowest being the denominator).
- For a given combination of skewed nodes and the load rate they request of some overall network load, any excess load (exceeding 1.0) on a given end point is distributed equally amongst all other end points whose loads are < 1.0.

With the above assumptions, we can calculate the skew factor for each combination of skewed nodes, corresponding traffic requested, and overall network load. Doing this for 0-100% of the network nodes being skewed and requesting 0-100% of the overall network load under network loads 0.1-0.9, we can construct

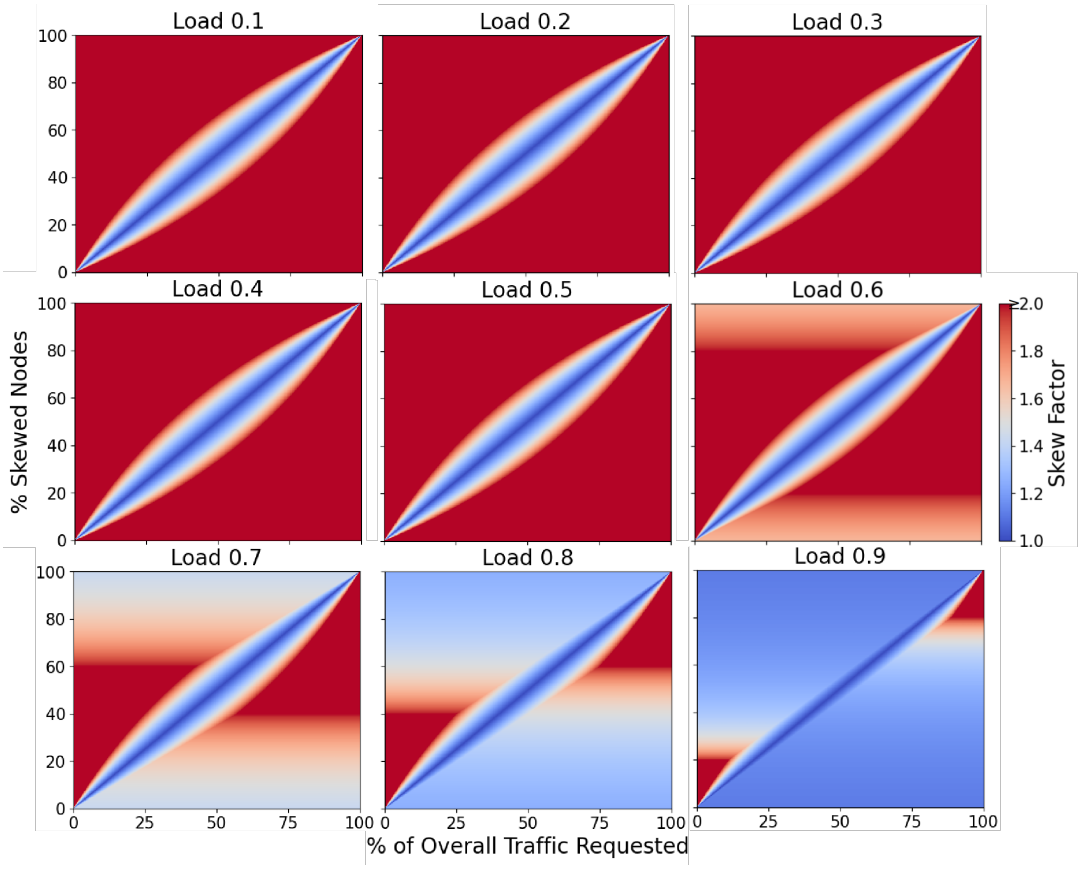


FIGURE B.5: Skew factor heat maps for 0-100% of network nodes requesting 0-100% of the overall network traffic across loads 0.1-0.9 plotted at 0.1% resolution. For clarity, combinations with skew factors ≥ 2 have been assigned the same colour.

a look-up table of skew factors for each of these combinations before generating any actual traffic. Fig. B.5 shows a high resolution (0.1%) heat map of these combinations, with any skew factors ≥ 2.0 set to the same colour for visual clarity. Fig. B.6 shows the corresponding plots with lower resolution (5%) but with the skew factors labelled. As expected, above 0.6 network loads, certain combinations of number of skewed nodes and traffic requested become restricted as to how much skew there can be in the matrix, with many combinations tending towards uniform (skew factor 1.0) at 0.9 loads.

Using the skew factor data from Figs. B.5 and B.6, we can be confident at 5%, 10%, 20%, and 40% of the network nodes requesting 55% of the overall network traffic that the skew factor will be > 1.0 across loads 0.1-0.9. Fig. B.7 shows the skew factor as a function of load for these combinations. Therefore,

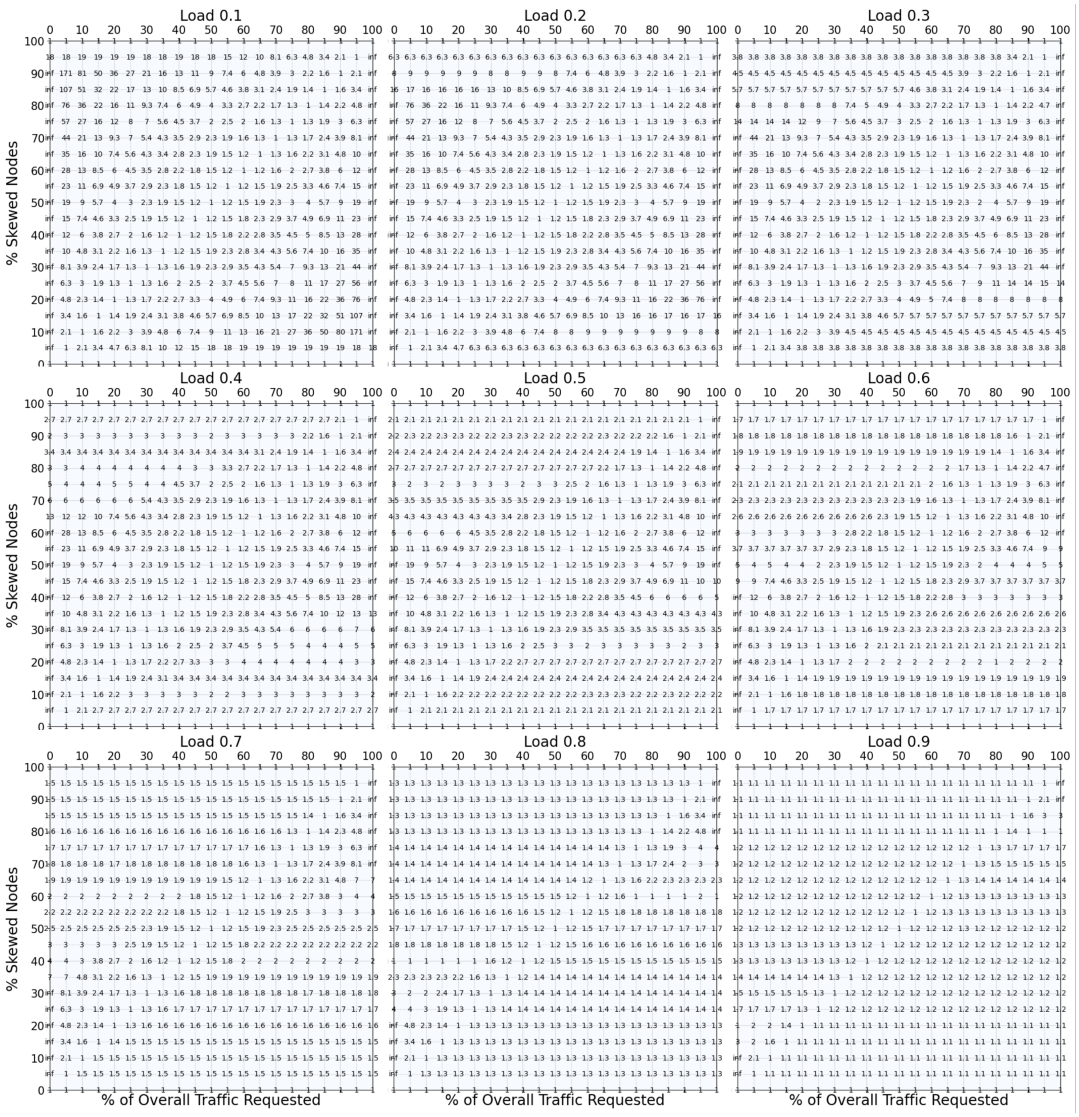


FIGURE B.6: Labelled skew factor tables for 0-100% of network nodes requesting 0-100% of the overall network traffic across loads 0.1-0.9 plotted at 5% resolution.

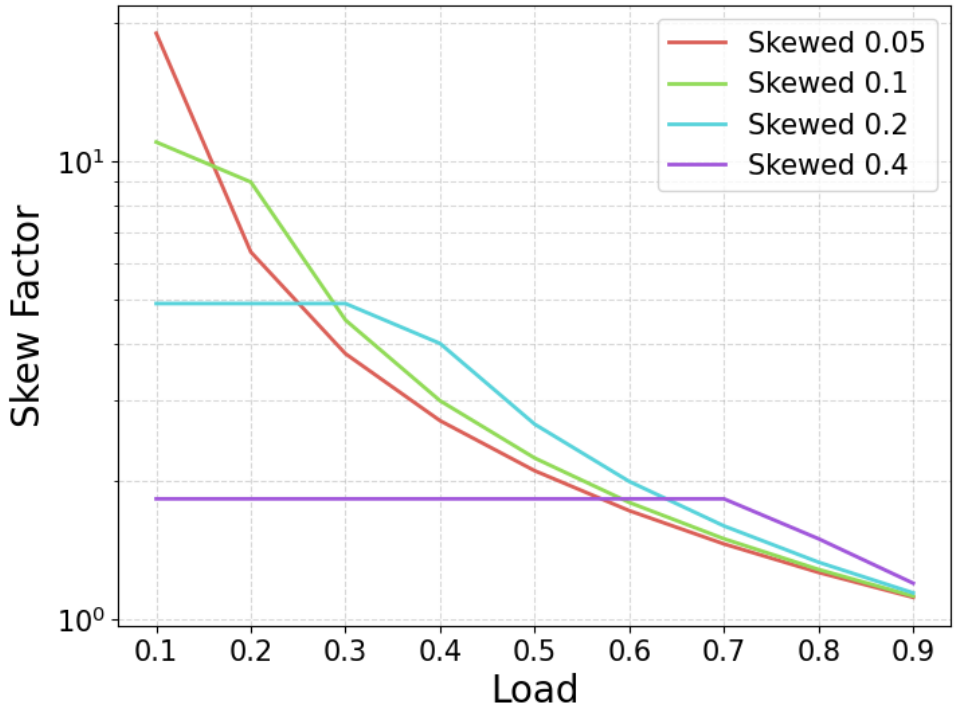


FIGURE B.7: Skew factor as a function of load for 5%, 10%, 20%, and 40% of the network nodes requesting 55% of the overall network traffic.

these were the combinations chosen for the skewed nodes sensitivity benchmark defined in Section 6.4 of this manuscript.

B.6 Scheduler Performance Summary

B.6.1 Completion Time Performance Plots

Plots showing the schedulers' completion performances are provided for the realistic DCN (Fig. B.8) uniform (Fig. B.9, extreme rack (Fig. B.10), and extreme nodes (Fig. B.11 traffic traces.

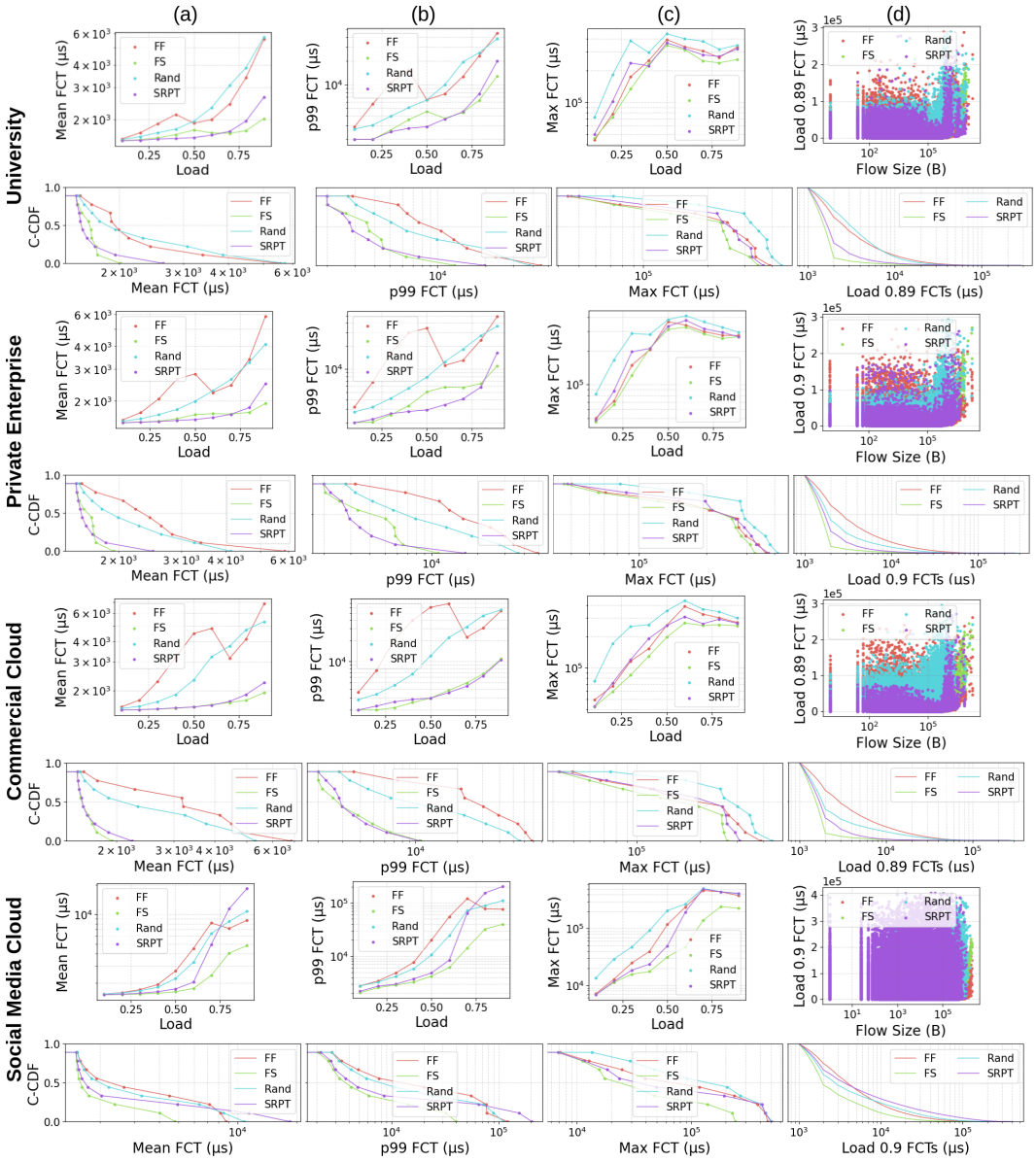


FIGURE B.8: The schedulers' (a) mean, (b) 99$^{\text{th}}$ percentile, and (c) maximum flow completion time metrics for the **DCN benchmark distributions** across loads 0.1-0.9, and (d) a scatter plot of flow completion time as a function of flow size for the same distribution at load 0.9.

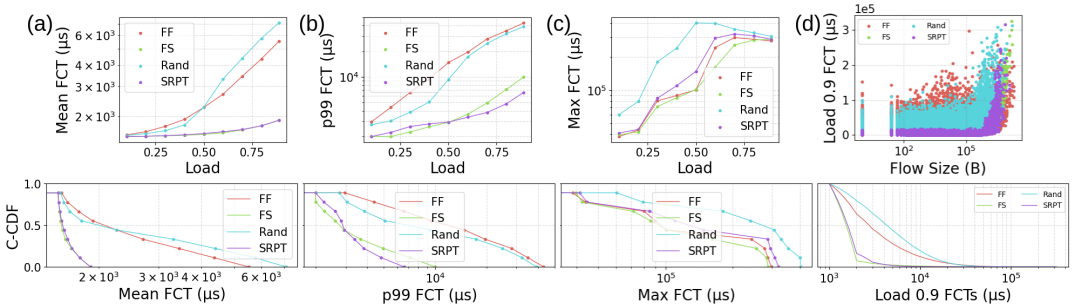


FIGURE B.9: The schedulers' (a) mean, (b) 99$^{\text{th}}$ percentile, and (c) maximum flow completion time metrics for the **uniform node distribution** across loads 0.1-0.9, and (d) a scatter plot of flow completion time as a function of flow size for the same distribution at load 0.9.

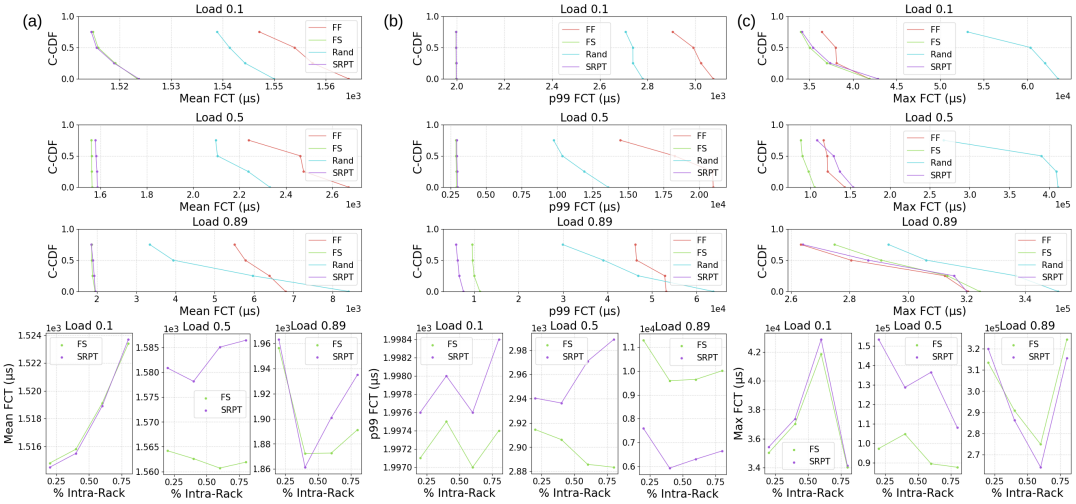


FIGURE B.10: Sensitivity of the schedulers' (a) mean, (b) 99$^{\text{th}}$ percentile, and (c) maximum flow completion times to the changing **intra-rack distribution** for loads 0.1, 0.5, and 0.9. The complementary CDF plots include data for all 4 schedulers, whereas the scatter plots contain the top 2 performing schedulers (SRPT and FS) for clarity.

B.6.2 Throughput and Flows Accepted Performance Plots

Plots showing the schedulers' throughput and accepted flow performances are provided for the realistic DCN (Fig. B.12, uniform (Fig. B.13), extreme rack (Fig. B.14), and extreme nodes (B.15) traffic traces.

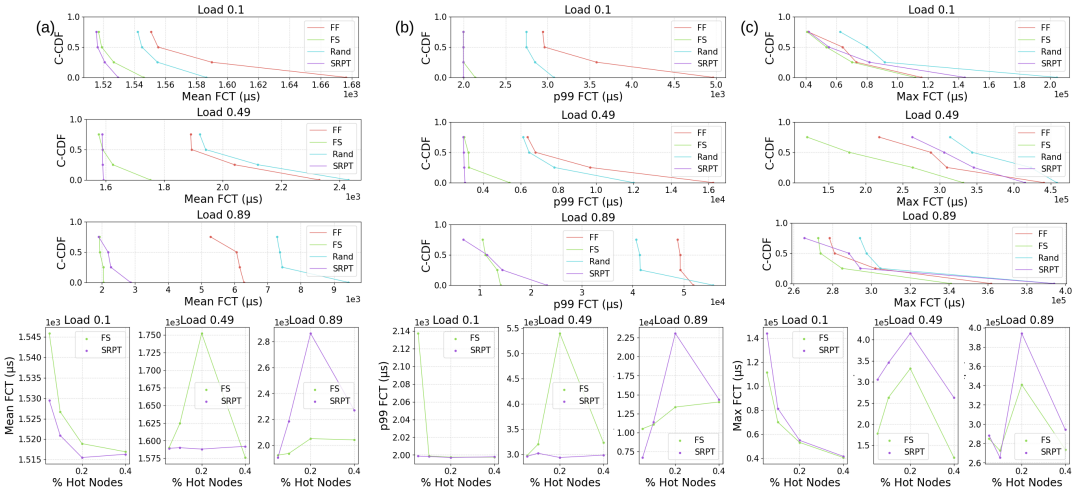


FIGURE B.11: Sensitivity of the schedulers' (a) mean, (b) 99$^{\text{th}}$ percentile, and (c) maximum flow completion times to the changing **skewed nodes distribution** for loads 0.1, 0.5, and 0.9. The complementary CDF plots include data for all 4 schedulers, whereas the scatter plots contain the top 2 performing schedulers (SRPT and FS) for clarity.

B.6.3 Performance Metric Tables

The below performance tables summarise the schedulers' mean performances (averaged across 5 runs, 95% confidence intervals reported) for each P_{KPI}, each load, and each benchmark.

DCN Benchmarks

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1557.2±0.19%	2903.2±0.77%	44249.8±8.9%	0.994±0.2%	1.0±0.0012%	0.994±0.2%
0.10	FS	1521.5±0.028%	1997.2±0.0059%	45984.4±11.0%	0.993±0.24%	1.0±0.00082%	0.993±0.24%
0.10	Rand	1543.5±0.051%	2708.2±0.38%	72316.3±9.1%	0.991±0.2%	1.0±0.00078%	0.991±0.2%
0.10	SRPT	1518.8±0.021%	1996.9±0.0039%	50036.6±11.0%	0.995±0.2%	1.0±0.00025%	0.995±0.2%
0.20	FF	1677.7±1.0%	5629.1±8.4%	77986.8±8.3%	0.985±0.39%	1.0±0.01%	0.985±0.39%
0.20	FS	1537.6±0.11%	1999.4±0.0039%	72962.6±5.9%	0.983±0.4%	1.0±0.0019%	0.983±0.4%
0.20	Rand	1600.8±0.18%	3050.2±1.3%	182454.6±11.0%	0.962±0.34%	1.0±0.0025%	0.962±0.34%
0.20	SRPT	1529.5±0.079%	2014.7±0.56%	102306.4±12.0%	0.985±0.32%	1.0±0.0019%	0.985±0.32%
0.30	FF	1887.8±0.78%	10474.4±4.9%	174541.8±16.0%	0.975±0.17%	0.999±0.0073%	0.975±0.17%
0.30	FS	1575.3±0.19%	2630.4±2.8%	134195.3±3.0%	0.97±0.12%	1.0±0.0013%	0.97±0.12%
0.30	Rand	1682.3±0.2%	3937.4±0.35%	381073.0±4.0%	0.857±0.87%	0.999±0.0063%	0.857±0.87%
0.30	SRPT	1551.2±0.099%	2500.5±0.29%	235811.0±5.7%	0.956±0.29%	1.0±0.00062%	0.956±0.29%
0.40	FF	2124.1±2.2%	15235.4±11.0%	247350.9±7.0%	0.939±0.38%	0.998±0.02%	0.939±0.38%
0.40	FS	1643.5±0.12%	3562.8±4.5%	230440.4±6.6%	0.926±0.58%	0.999±0.0025%	0.926±0.58%
0.40	Rand	1762.5±0.23%	5081.8±0.67%	295319.0±1.8%	0.816±0.75%	0.999±0.0092%	0.816±0.75%
0.40	SRPT	1561.9±0.08%	2771.3±0.31%	221163.5±5.0%	0.902±0.41%	1.0±0.0014%	0.902±0.41%
0.50	FF	1902.1±1.1%	6389.1±2.7%	391005.8±7.6%	0.909±0.94%	0.999±0.0067%	0.909±0.94%
0.50	FS	1740.5±1.2%	4533.5±12.0%	344343.1±7.9%	0.9±1.1%	0.999±0.0055%	0.9±1.1%
0.50	Rand	1947.7±1.8%	6365.3±4.5%	443976.4±11.0%	0.818±1.2%	0.998±0.0037%	0.818±1.2%
0.50	SRPT	1582.2±0.16%	2904.8±0.36%	363481.8±7.4%	0.875±0.76%	1.0±0.0012%	0.875±0.76%
0.60	FF	1989.3±1.0%	7602.7±4.6%	335234.2±5.2%	0.917±0.39%	0.999±0.0057%	0.917±0.39%
0.60	FS	1677.7±0.53%	3701.9±1.1%	314020.0±4.8%	0.912±0.31%	0.999±0.0036%	0.912±0.31%
0.60	Rand	2322.4±2.7%	9921.0±8.2%	398738.8±2.5%	0.805±0.48%	0.997±0.027%	0.805±0.48%
0.60	SRPT	1630.0±0.084%	3630.4±0.48%	322416.8±5.0%	0.879±0.47%	1.0±0.0022%	0.879±0.47%
0.70	FF	2434.1±1.8%	12649.6±5.0%	305610.1±2.9%	0.912±0.35%	0.998±0.033%	0.912±0.35%
0.70	FS	1672.2±0.4%	4415.8±1.9%	246486.9±2.9%	0.914±0.3%	0.999±0.0033%	0.914±0.3%
0.70	Rand	3083.8±1.4%	19421.0±4.0%	377667.2±2.1%	0.755±1.1%	0.993±0.048%	0.755±1.1%
0.70	SRPT	1712.6±0.28%	4502.1±1.8%	280418.9±5.9%	0.878±0.46%	0.999±0.008%	0.878±0.46%
0.79	FF	3394.1±2.1%	23179.1±3.5%	265525.7±5.5%	0.9±0.23%	0.995±0.033%	0.9±0.23%
0.79	FS	1724.5±0.31%	6302.9±1.9%	236377.1±3.3%	0.913±0.28%	0.999±0.004%	0.913±0.28%
0.79	Rand	3861.5±1.8%	25389.9±1.9%	317002.4±2.3%	0.731±0.83%	0.988±0.033%	0.731±0.83%
0.79	SRPT	1950.3±1.3%	7574.3±6.7%	271794.0±1.7%	0.848±0.36%	0.999±0.017%	0.848±0.36%
0.89	FF	5550.1±1.9%	44869.3±2.5%	333023.3±11.0%	0.87±0.62%	0.987±0.041%	0.87±0.62%
0.89	FS	2015.9±0.54%	12793.3±1.7%	254036.6±10.0%	0.873±0.9%	0.998±0.013%	0.873±0.9%
0.89	Rand	5718.1±7.5%	38174.0±8.7%	346773.2±12.0%	0.692±0.71%	0.979±0.045%	0.692±0.71%
0.89	SRPT	2645.0±5.0%	19839.5±12.0%	319581.9±11.0%	0.755±0.43%	0.993±0.12%	0.755±0.43%

TABLE B.4: Scheduler performance summary with 95% confidence intervals for the **University** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1576.7±0.34%	3207.9±3.5%	50143.5±5.5%	0.998±0.085%	1.0±0.00094%	0.998±0.085%
0.10	FS	1522.1±0.021%	1997.1±0.0079%	46335.0±4.4%	0.997±0.095%	1.0±0.0006%	0.997±0.095%
0.10	Rand	1550.9±0.053%	2765.2±0.49%	82610.5±7.8%	0.994±0.19%	1.0±0.00074%	0.994±0.19%
0.10	SRPT	1520.3±0.01%	1997.3±0.0079%	48062.1±5.8%	0.997±0.13%	1.0±0.00047%	0.997±0.13%
0.20	FF	1726.6±1.6%	6794.6±11.0%	70833.7±3.0%	0.983±0.29%	0.999±0.01%	0.983±0.29%
0.20	FS	1532.2±0.13%	2048.4±0.76%	66026.7±2.0%	0.983±0.22%	1.0±0.00072%	0.983±0.22%
0.20	Rand	1598.9±0.16%	3199.8±2.0%	166233.2±8.3%	0.946±0.6%	1.0±0.0044%	0.946±0.6%
0.20	SRPT	1529.5±0.11%	2214.8±1.6%	87532.1±7.1%	0.984±0.22%	1.0±0.00048%	0.984±0.22%
0.30	FF	2058.9±3.1%	16033.0±12.0%	149462.6±8.8%	0.98±0.19%	0.999±0.016%	0.98±0.19%
0.30	FS	1549.9±0.13%	2528.8±1.1%	121311.0±7.3%	0.981±0.24%	1.0±0.001%	0.981±0.24%
0.30	Rand	1684.2±0.39%	4149.8±1.9%	285851.7±4.8%	0.899±0.73%	0.999±0.0088%	0.899±0.73%
0.30	SRPT	1543.2±0.056%	2616.2±0.41%	196424.2±9.0%	0.978±0.22%	1.0±0.00089%	0.978±0.22%
0.40	FF	2638.3±4.1%	30026.6±9.2%	205182.9±8.2%	0.942±0.6%	0.997±0.036%	0.942±0.6%
0.40	FS	1599.4±0.25%	3333.2±1.9%	211188.7±4.4%	0.943±0.21%	1.0±0.002%	0.943±0.21%
0.40	Rand	1799.1±0.54%	5653.6±2.3%	280714.7±3.0%	0.84±1.1%	0.999±0.015%	0.84±1.1%
0.40	SRPT	1564.1±0.085%	2802.8±0.32%	210192.4±7.6%	0.937±0.46%	1.0±0.0017%	0.937±0.46%
0.50	FF	2824.6±5.9%	34301.5±14.0%	365468.3±13.0%	0.907±1.0%	0.994±0.11%	0.907±1.0%
0.50	FS	1682.6±0.72%	5048.5±3.6%	311288.1±9.1%	0.902±1.2%	0.999±0.0061%	0.902±1.2%
0.50	Rand	1993.9±1.9%	7870.4±4.9%	381296.9±10.0%	0.811±1.1%	0.998±0.019%	0.811±1.1%
0.50	SRPT	1582.9±0.26%	2938.1±0.38%	332134.3±13.0%	0.903±0.65%	1.0±0.0026%	0.903±0.65%
0.60	FF	2230.4±1.3%	11218.7±5.3%	339021.9±2.3%	0.915±0.44%	0.997±0.065%	0.915±0.44%
0.60	FS	1705.0±0.53%	5843.2±3.2%	326252.1±3.6%	0.907±0.43%	0.999±0.0044%	0.907±0.43%
0.60	Rand	2282.4±1.6%	12522.1±6.0%	412445.3±4.4%	0.782±1.4%	0.997±0.029%	0.782±1.4%
0.60	SRPT	1624.0±0.21%	3425.1±1.3%	375244.9±5.9%	0.898±0.38%	1.0±0.0028%	0.898±0.38%
0.70	FF	2449.3±0.71%	13110.2±2.3%	297091.8±4.2%	0.921±0.26%	0.998±0.02%	0.921±0.26%
0.70	FS	1696.4±0.49%	5751.0±4.4%	283512.5±4.4%	0.907±0.17%	0.999±0.003%	0.907±0.17%
0.70	Rand	2636.5±0.7%	18278.2±2.0%	363011.5±2.3%	0.74±1.0%	0.995±0.029%	0.74±1.0%
0.70	SRPT	1691.0±0.23%	4085.2±1.5%	315470.7±7.2%	0.892±0.36%	1.0±0.0026%	0.892±0.36%
0.79	FF	3400.0±0.81%	24127.3±1.5%	275964.6±3.9%	0.897±0.37%	0.994±0.03%	0.897±0.37%
0.79	FS	1732.4±0.24%	6508.5±1.6%	258779.8±3.2%	0.893±0.45%	0.999±0.0034%	0.893±0.45%
0.79	Rand	3264.4±1.7%	27586.4±3.0%	325223.7±2.3%	0.675±0.73%	0.989±0.04%	0.675±0.73%
0.79	SRPT	1841.9±0.58%	5834.0±2.5%	292946.1±3.7%	0.853±0.25%	0.999±0.011%	0.853±0.25%
0.90	FF	5851.8±1.9%	48861.6±2.7%	274329.9±2.0%	0.866±0.64%	0.983±0.08%	0.866±0.64%
0.90	FS	1940.3±0.35%	11084.7±2.1%	268340.1±3.0%	0.842±0.52%	0.998±0.0064%	0.842±0.52%
0.90	Rand	4124.7±1.4%	36647.1±2.2%	294642.1±0.58%	0.625±1.5%	0.983±0.055%	0.625±1.5%
0.90	SRPT	2492.0±5.5%	16474.5±15.0%	267699.8±2.0%	0.711±0.41%	0.994±0.2%	0.711±0.41%

TABLE B.5: Scheduler performance summary with 95% confidence intervals for the **Private Enterprise** benchmark.

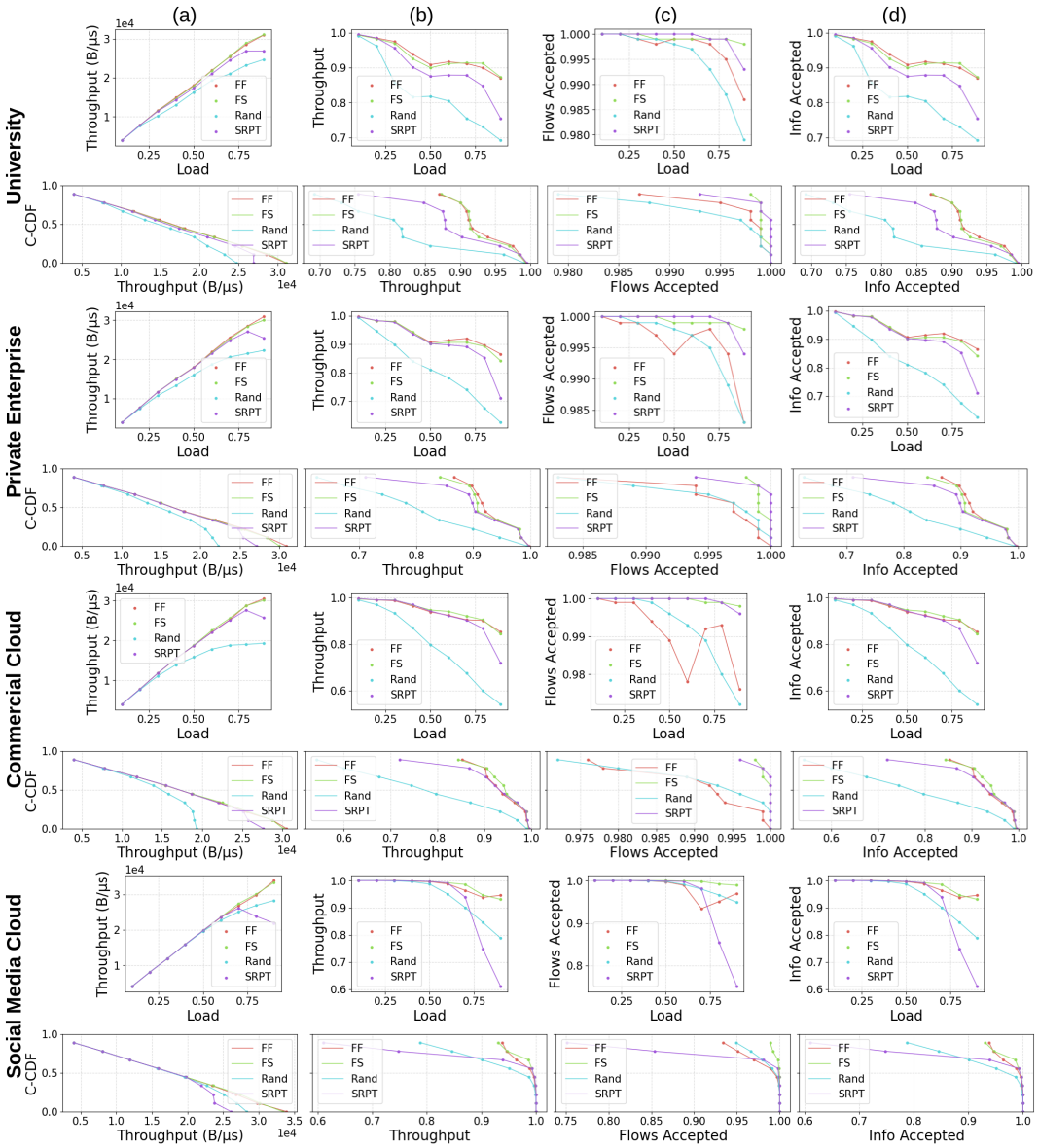


FIGURE B.12: The schedulers' (a) absolute throughput (information units transported per unit time), (b) relative throughput (fraction of arrived information successfully transported), (c) fraction of arrived flows accepted, and (d) fraction of arrived information accepted metrics for the **DCN benchmark distributions** across loads 0.1-0.9.

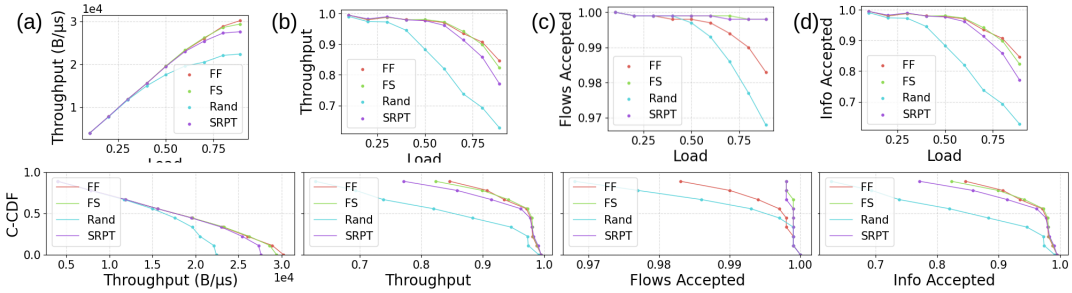


FIGURE B.13: The schedulers' (a) absolute throughput (information units transported per unit time), (b) relative throughput (fraction of arrived information successfully transported), (c) fraction of arrived flows accepted, and (d) fraction of arrived information accepted metrics for the **uniform node distribution** across loads 0.1-0.9.

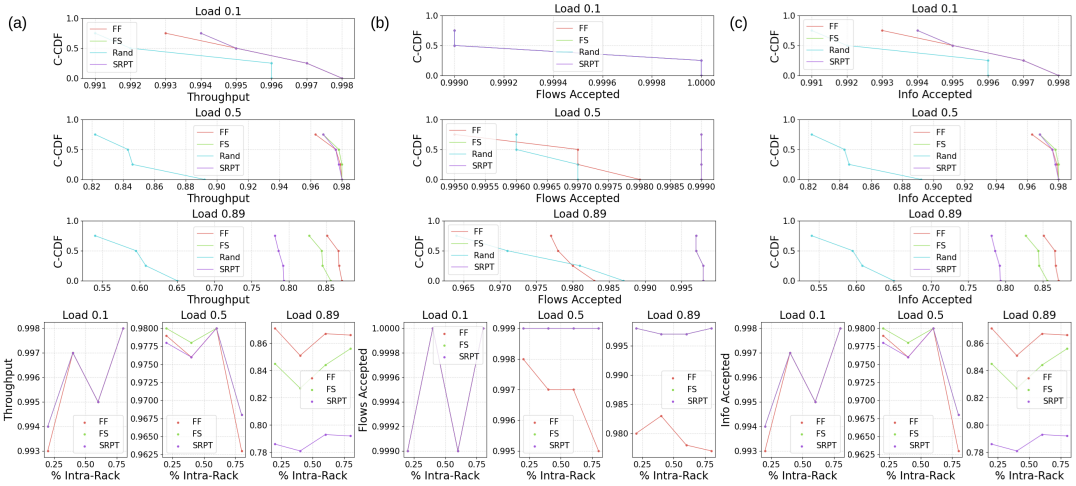


FIGURE B.14: Sensitivity of the schedulers' (a) relative throughput, (b) fraction of arrived flows accepted, and (c) fraction of arrived information accepted metrics to the changing **intra-rack distribution** for loads 0.1, 0.5, and 0.9. The complementary CDF plots include data for all 4 schedulers, whereas the scatter plots contain the top 3 performing schedulers (SRPT, FS, and FF) for clarity.

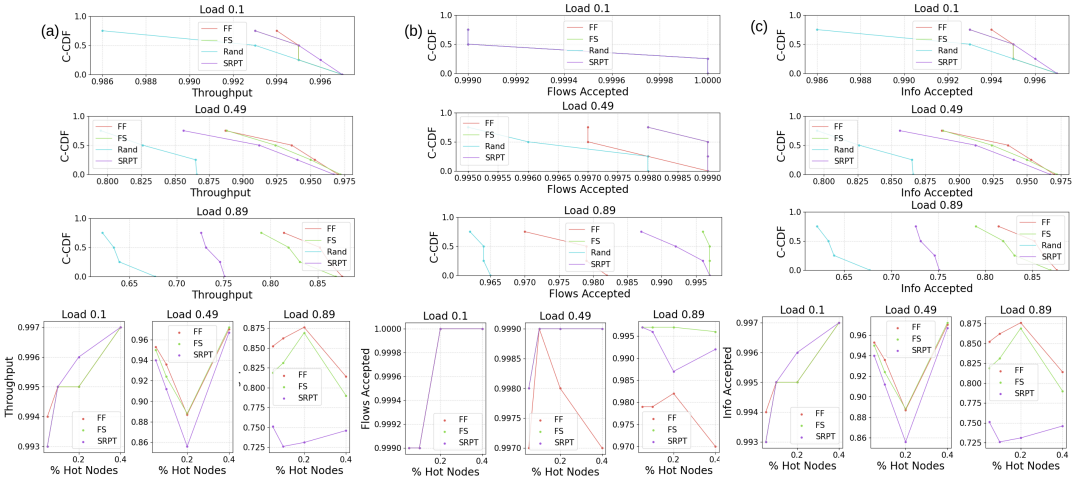


FIGURE B.15: Sensitivity of the schedulers' (a) relative throughput, (b) fraction of arrived flows accepted, and (c) fraction of arrived information accepted metrics to the changing **skewed nodes distribution** for loads 0.1, 0.5, and 0.9. The complementary CDF plots include data for all 4 schedulers, whereas the scatter plots contain the top 3 performing schedulers (SRPT, FS, and FF) for clarity.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1588.2±0.46%	3604.1±2.4%	49490.3±6.8%	0.996±0.052%	1.0±0.0019%	0.996±0.052%
0.10	FS	1520.1±0.083%	1997.1±0.0039%	42361.1±4.6%	0.994±0.16%	1.0±0.00059%	0.994±0.16%
0.10	Rand	1551.4±0.12%	2816.9±0.47%	75051.0±13.0%	0.99±0.2%	1.0±0.0023%	0.99±0.2%
0.10	SRPT	1519.3±0.077%	1997.7±0.0059%	42911.8±5.8%	0.996±0.08%	1.0±0.00037%	0.996±0.08%
0.20	FF	1747.8±1.2%	7437.1±6.9%	67090.9±3.3%	0.99±0.29%	0.999±0.018%	0.99±0.29%
0.20	FS	1524.9±0.14%	1998.8±0.0059%	59363.5±6.5%	0.991±0.3%	1.0±0.0013%	0.991±0.3%
0.20	Rand	1602.1±0.24%	3372.3±1.2%	171058.6±6.9%	0.97±0.64%	1.0±0.0033%	0.97±0.64%
0.20	SRPT	1525.8±0.13%	2276.5±0.57%	71962.2±7.8%	0.991±0.26%	1.0±0.0013%	0.991±0.26%
0.30	FF	2274.3±2.6%	21086.7±9.0%	116200.4±8.8%	0.987±0.06%	0.999±0.012%	0.987±0.06%
0.30	FS	1538.4±0.061%	2149.4±0.9%	85571.7±5.5%	0.99±0.066%	1.0±0.00071%	0.99±0.066%
0.30	Rand	1707.2±0.29%	4544.2±1.7%	249283.9±9.6%	0.933±0.58%	1.0±0.003%	0.933±0.58%
0.30	SRPT	1540.7±0.023%	2620.5±0.31%	119981.5±12.0%	0.989±0.092%	1.0±0.00056%	0.989±0.092%
0.40	FF	3203.2±3.6%	39373.6±7.1%	153040.1±4.9%	0.964±0.31%	0.994±0.11%	0.964±0.31%
0.40	FS	1557.2±0.17%	2559.2±0.57%	129399.8±9.6%	0.968±0.36%	1.0±0.00083%	0.968±0.36%
0.40	Rand	1889.3±0.56%	6600.9±3.0%	259317.4±3.9%	0.87±0.65%	0.999±0.012%	0.87±0.65%
0.40	SRPT	1564.1±0.13%	2830.9±0.38%	190613.2±10.0%	0.97±0.25%	1.0±0.00072%	0.97±0.25%
0.50	FF	4495.2±3.4%	60948.4±4.3%	255736.7±14.0%	0.939±0.64%	0.989±0.18%	0.939±0.64%
0.50	FS	1584.6±0.13%	2963.7±0.38%	196875.6±7.7%	0.947±0.84%	1.0±0.0039%	0.947±0.84%
0.50	Rand	2324.1±3.3%	12139.1±11.0%	353111.3±13.0%	0.797±0.74%	0.996±0.027%	0.797±0.74%
0.50	SRPT	1585.3±0.082%	2962.4±0.21%	254463.8±8.2%	0.942±0.56%	1.0±0.0022%	0.942±0.56%
0.60	FF	4837.1±5.1%	68328.0±3.3%	387525.7±2.3%	0.924±0.23%	0.978±0.2%	0.924±0.23%
0.60	FS	1639.9±0.14%	3835.1±0.83%	268943.4±3.6%	0.941±0.14%	1.0±0.0018%	0.941±0.14%
0.60	Rand	3236.8±0.65%	22198.9±0.66%	439374.7±1.1%	0.744±0.42%	0.993±0.015%	0.744±0.42%
0.60	SRPT	1628.1±0.15%	3565.0±0.8%	308435.8±4.8%	0.922±0.26%	1.0±0.0026%	0.922±0.26%
0.70	FF	3173.6±0.7%	22472.4±2.9%	327840.2±2.7%	0.905±0.31%	0.992±0.044%	0.905±0.31%
0.70	FS	1686.9±0.23%	4915.5±1.0%	254484.7±1.8%	0.921±0.44%	0.999±0.0024%	0.921±0.44%
0.70	Rand	3760.3±0.94%	31788.5±2.2%	365861.9±2.2%	0.675±0.25%	0.989±0.027%	0.675±0.25%
0.70	SRPT	1715.2±0.24%	4404.1±1.2%	264969.5±5.5%	0.903±0.33%	1.0±0.004%	0.903±0.33%
0.79	FF	4144.2±2.0%	30541.3±4.0%	301349.2±2.6%	0.902±0.18%	0.993±0.025%	0.902±0.18%
0.79	FS	1743.5±0.24%	6572.0±1.6%	259058.4±2.9%	0.905±0.18%	0.999±0.0026%	0.905±0.18%
0.79	Rand	4740.4±0.98%	46094.7±2.0%	344636.1±0.65%	0.6±0.65%	0.98±0.032%	0.6±0.65%
0.79	SRPT	1889.5±0.74%	6169.9±3.9%	292500.7±4.5%	0.868±0.038%	0.999±0.0052%	0.868±0.038%
0.89	FF	6856.2±0.89%	54158.7±2.0%	272757.7±1.4%	0.853±0.25%	0.976±0.14%	0.853±0.25%
0.89	FS	1940.3±0.16%	10891.2±0.75%	253250.9±1.6%	0.844±0.37%	0.998±0.0061%	0.844±0.37%
0.89	Rand	5320.7±1.0%	55646.5±1.6%	300652.9±0.86%	0.541±0.4%	0.972±0.051%	0.541±0.4%
0.89	SRPT	2234.9±1.5%	10623.8±4.9%	267587.5±2.8%	0.719±0.62%	0.996±0.025%	0.719±0.62%

TABLE B.6: Scheduler performance summary with 95% confidence intervals for the **Commercial Cloud** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1536.7±0.074%	2766.2±0.3%	7153.9±2.0%	1.0±0.0064%	1.0±0.00041%	1.0±0.0064%
0.10	FS	1513.9±0.062%	2053.4±0.84%	6892.3±3.1%	1.0±0.0062%	1.0±0.00024%	1.0±0.0062%
0.10	Rand	1536.5±0.054%	2762.4±0.35%	13551.4±11.0%	1.0±0.0079%	1.0±0.00056%	1.0±0.0079%
0.10	SRPT	1515.2±0.062%	2189.1±0.38%	6820.5±3.6%	1.0±0.0063%	1.0±0.00032%	1.0±0.0063%
0.20	FF	1591.6±0.11%	3410.3±1.2%	12773.2±12.0%	1.0±0.0045%	1.0±0.00091%	1.0±0.0045%
0.20	FS	1523.7±0.033%	2560.6±0.61%	11206.1±11.0%	1.0±0.0056%	1.0±0.00047%	1.0±0.0056%
0.20	Rand	1581.4±0.097%	3237.3±1.2%	29019.5±19.0%	1.0±0.0099%	1.0±0.0015%	1.0±0.0099%
0.20	SRPT	1532.6±0.054%	2720.7±0.49%	11620.8±11.0%	1.0±0.0052%	1.0±0.0005%	1.0±0.0052%
0.30	FF	1707.8±0.42%	4849.3±2.3%	24735.7±10.0%	1.0±0.011%	1.0±0.0032%	1.0±0.011%
0.30	FS	1539.5±0.056%	2859.5±0.27%	15729.0±8.7%	1.0±0.0089%	1.0±0.0006%	1.0±0.0089%
0.30	Rand	1660.7±0.13%	4184.2±1.1%	47524.1±19.0%	0.999±0.025%	1.0±0.0033%	0.999±0.025%
0.30	SRPT	1565.8±0.095%	2972.7±0.14%	18417.1±12.0%	1.0±0.0073%	1.0±0.00098%	1.0±0.0073%
0.40	FF	1924.6±0.8%	7639.7±3.0%	39600.9±9.7%	0.998±0.021%	0.999±0.0098%	0.998±0.021%
0.40	FS	1563.9±0.11%	3266.6±1.1%	17450.8±3.5%	0.999±0.019%	1.0±0.0023%	0.999±0.019%
0.40	Rand	1808.3±0.31%	5802.5±0.92%	92643.3±23.0%	0.996±0.042%	0.999±0.0058%	0.996±0.042%
0.40	SRPT	1622.6±0.19%	3731.7±0.93%	23635.4±7.2%	0.999±0.01%	1.0±0.0038%	0.999±0.01%
0.50	FF	2646.7±2.9%	20076.0±7.9%	117682.7±9.7%	0.996±0.066%	0.997±0.052%	0.996±0.066%
0.50	FS	1624.4±0.21%	4201.7±1.4%	31567.8±3.4%	0.997±0.058%	1.0±0.0047%	0.997±0.058%
0.50	Rand	2218.8±0.77%	10570.1±3.4%	207351.1±11.0%	0.987±0.15%	0.998±0.019%	0.987±0.15%
0.50	SRPT	1737.3±0.53%	4829.9±1.8%	49492.8±6.5%	0.997±0.045%	0.999±0.013%	0.997±0.045%
0.60	FF	4495.9±4.4%	55356.7±7.6%	237610.0±7.3%	0.988±0.16%	0.989±0.053%	0.988±0.16%
0.60	FS	1755.8±0.41%	6110.1±1.8%	47599.2±5.6%	0.992±0.15%	0.999±0.024%	0.992±0.15%
0.60	Rand	3262.0±1.6%	24348.0±2.2%	269243.0±2.0%	0.951±0.31%	0.991±0.049%	0.951±0.31%
0.60	SRPT	2034.5±2.0%	8447.1±8.2%	193698.4±8.9%	0.992±0.12%	0.998±0.071%	0.992±0.12%
0.69	FF	8175.5±2.7%	121246.2±3.5%	468538.0±5.4%	0.964±0.22%	0.934±0.82%	0.964±0.22%
0.69	FS	2384.6±1.7%	14253.5±3.9%	138806.7±6.4%	0.986±0.14%	0.998±0.026%	0.986±0.14%
0.69	Rand	6394.4±1.4%	72096.8±3.3%	507914.9±2.5%	0.901±0.29%	0.98±0.049%	0.901±0.29%
0.69	SRPT	4937.4±9.9%	64798.0±18.0%	500125.6±2.8%	0.939±0.81%	0.981±0.55%	0.939±0.81%
0.80	FF	7182.3±1.7%	77566.2±3.8%	443785.0±2.9%	0.938±0.13%	0.951±0.19%	0.938±0.13%
0.80	FS	4026.1±2.1%	32187.7±2.7%	243834.5±3.5%	0.947±0.19%	0.992±0.034%	0.947±0.19%
0.80	Rand	8489.0±2.2%	89488.4±1.5%	446095.0±1.4%	0.846±0.23%	0.966±0.06%	0.846±0.23%
0.80	SRPT	11412.4±4.1%	154590.0±3.9%	443708.3±1.7%	0.748±0.42%	0.854±0.88%	0.748±0.42%
0.90	FF	8731.6±1.5%	76236.3±1.8%	380339.7±2.7%	0.946±0.13%	0.97±0.2%	0.946±0.13%
0.90	FS	4809.9±1.4%	40007.0±2.0%	228118.7±2.1%	0.931±0.14%	0.989±0.038%	0.931±0.14%
0.90	Rand	10800.9±0.96%	110549.1±0.69%	407971.9±1.1%	0.788±0.41%	0.949±0.15%	0.788±0.41%
0.90	SRPT	18401.3±2.4%	204251.6±2.0%	416090.4±0.56%	0.61±0.78%	0.751±0.98%	0.61±0.78%

TABLE B.7: Scheduler performance summary with 95% confidence intervals for the **Social Media Cloud** benchmark.

Skewed Nodes Distribution Benchmark

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.1	FF	1554.5±0.15%	2977.0±0.79%	38288.3±6.7%	0.995±0.11%	1.0±0.024%	0.995±0.11%
0.1	FS	1518.8±0.12%	1997.5±0.0039%	39693.2±4.6%	0.995±0.11%	1.0±0.024%	0.995±0.11%
0.1	Rand	1544.1±0.11%	2750.1±0.34%	60170.8±9.4%	0.991±0.15%	1.0±0.024%	0.991±0.15%
0.1	SRPT	1518.3±0.12%	1998.0±0.0039%	41190.0±5.2%	0.995±0.11%	1.0±0.024%	0.995±0.11%
0.2	FF	1620.8±0.34%	4398.3±4.0%	43732.0±4.4%	0.98±0.3%	0.999±0.054%	0.98±0.3%
0.2	FS	1524.3±0.1%	1999.6±0.016%	42196.8±4.5%	0.982±0.32%	0.999±0.055%	0.982±0.32%
0.2	Rand	1579.3±0.18%	3049.8±1.1%	79304.0±9.3%	0.974±0.25%	0.999±0.057%	0.974±0.25%
0.2	SRPT	1524.9±0.087%	2234.7±1.0%	44396.8±4.8%	0.983±0.28%	0.999±0.055%	0.983±0.28%
0.3	FF	1744.2±0.55%	6564.0±2.3%	80217.5±5.9%	0.988±0.18%	0.999±0.069%	0.988±0.18%
0.3	FS	1532.9±0.1%	2255.9±0.46%	71447.0±7.3%	0.989±0.16%	0.999±0.064%	0.989±0.16%
0.3	Rand	1643.6±0.22%	3856.5±0.36%	180283.0±6.6%	0.973±0.27%	0.999±0.066%	0.973±0.27%
0.3	SRPT	1537.1±0.071%	2612.7±0.59%	84911.1±7.0%	0.99±0.15%	0.999±0.064%	0.99±0.15%
0.4	FF	1917.3±0.82%	9481.8±2.8%	89676.1±6.4%	0.981±0.29%	0.998±0.057%	0.981±0.29%
0.4	FS	1544.5±0.079%	2602.7±0.7%	85476.6±5.7%	0.98±0.3%	0.999±0.049%	0.98±0.3%
0.4	Rand	1776.3±0.22%	5093.5±0.91%	239854.0±7.3%	0.946±0.38%	0.999±0.05%	0.946±0.38%
0.4	SRPT	1554.7±0.058%	2819.0±0.48%	109885.3±8.2%	0.98±0.25%	0.999±0.05%	0.98±0.25%
0.5	FF	2254.6±0.82%	14792.9±1.9%	100669.6±4.8%	0.978±0.24%	0.998±0.046%	0.978±0.24%
0.5	FS	1563.8±0.16%	2927.5±0.58%	101281.5±7.7%	0.981±0.23%	0.999±0.042%	0.981±0.23%
0.5	Rand	2259.1±1.4%	9368.2±3.4%	403534.7±12.0%	0.883±0.74%	0.997±0.041%	0.883±0.74%
0.5	SRPT	1580.4±0.069%	2948.5±0.13%	148065.0±4.9%	0.977±0.25%	0.999±0.04%	0.977±0.25%
0.6	FF	2696.5±1.4%	19574.0±3.3%	242541.4±13.0%	0.971±0.36%	0.997±0.051%	0.971±0.36%
0.6	FS	1595.6±0.15%	3652.1±0.98%	161242.9±14.0%	0.973±0.24%	0.999±0.051%	0.973±0.24%
0.6	Rand	3309.7±1.1%	17326.4±1.6%	401082.8±4.4%	0.82±0.87%	0.993±0.066%	0.82±0.87%
0.6	SRPT	1620.9±0.077%	3373.6±0.78%	294496.9±6.8%	0.962±0.35%	0.999±0.051%	0.962±0.35%
0.7	FF	3436.8±1.0%	27933.1±2.3%	297748.1±3.1%	0.935±0.51%	0.994±0.077%	0.935±0.51%
0.7	FS	1660.9±0.21%	4953.9±1.2%	255268.4±4.4%	0.942±0.36%	0.999±0.078%	0.942±0.36%
0.7	Rand	4393.5±1.1%	24778.6±1.8%	354839.4±2.2%	0.738±1.1%	0.986±0.082%	0.738±1.1%
0.7	SRPT	1668.4±0.15%	3827.3±0.71%	320957.0±4.4%	0.914±0.47%	0.998±0.077%	0.914±0.47%
0.8	FF	4361.4±2.1%	34817.0±2.7%	287276.9±3.8%	0.907±0.59%	0.99±0.15%	0.907±0.59%
0.8	FS	1758.1±0.5%	7135.0±1.9%	283104.7±1.3%	0.899±0.8%	0.998±0.1%	0.899±0.8%
0.8	Rand	5762.2±1.5%	32239.6±1.4%	329015.7±2.0%	0.693±1.1%	0.977±0.15%	0.693±1.1%
0.8	SRPT	1758.1±0.41%	4842.1±2.4%	309165.9±3.6%	0.858±0.63%	0.998±0.11%	0.858±0.63%
0.9	FF	5520.3±1.7%	43104.1±2.4%	278164.1±2.2%	0.846±0.61%	0.983±0.061%	0.846±0.61%
0.9	FS	1890.9±0.47%	9974.2±1.9%	287700.1±3.1%	0.823±0.74%	0.998±0.038%	0.823±0.74%
0.9	Rand	7095.9±1.3%	39006.4±1.4%	306075.6±1.6%	0.627±0.93%	0.968±0.041%	0.627±0.93%
0.9	SRPT	1890.8±0.89%	6584.6±4.1%	287161.6±2.9%	0.771±0.54%	0.998±0.036%	0.771±0.54%

TABLE B.8: Scheduler performance summary with 95% confidence intervals for the **skewed_nodes_sensitivity_uniform** and **rack_sensitivity_uniform** benchmarks.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1676.3±1.3%	4965.6±5.9%	115613.1±12.0%	0.994±0.17%	0.999±0.042%	0.994±0.17%
0.10	FS	1545.9±0.21%	2137.2±1.5%	111455.6±11.0%	0.993±0.15%	0.999±0.04%	0.993±0.15%
0.10	Rand	1586.2±0.22%	3071.9±1.5%	204371.5±6.6%	0.986±0.18%	0.999±0.04%	0.986±0.18%
0.10	SRPT	1529.5±0.14%	1998.7±0.0078%	144042.5±11.0%	0.993±0.16%	0.999±0.041%	0.993±0.16%
0.20	FF	1769.9±2.2%	4943.9±12.0%	281567.2±4.5%	0.922±0.66%	0.997±0.086%	0.922±0.66%
0.20	FS	1653.3±0.56%	3724.2±11.0%	264636.5±5.0%	0.896±0.51%	0.998±0.092%	0.896±0.51%
0.20	Rand	1691.1±0.83%	4168.9±5.4%	185373.4±3.9%	0.901±0.24%	0.998±0.091%	0.901±0.24%
0.20	SRPT	1547.1±0.16%	2306.6±1.5%	165611.6±5.4%	0.933±0.29%	0.999±0.093%	0.933±0.29%
0.30	FF	1697.5±0.24%	4419.9±2.0%	289568.9±7.6%	0.949±0.49%	0.999±0.037%	0.949±0.49%
0.30	FS	1612.6±0.46%	2501.7±1.3%	297525.1±2.3%	0.927±0.69%	0.999±0.035%	0.927±0.69%
0.30	Rand	1686.6±0.47%	3854.9±0.88%	210069.0±5.6%	0.911±0.71%	0.999±0.036%	0.911±0.71%
0.30	SRPT	1551.1±0.14%	2604.4±0.24%	228406.2±12.0%	0.943±0.35%	0.999±0.038%	0.943±0.35%
0.40	FF	1789.6±0.58%	6066.7±3.7%	257805.2±5.8%	0.955±0.24%	0.998±0.096%	0.955±0.24%
0.40	FS	1584.6±0.29%	2728.7±0.95%	201816.6±2.3%	0.938±0.16%	0.999±0.088%	0.938±0.16%
0.40	Rand	1783.5±0.32%	4928.8±0.76%	275464.8±7.1%	0.905±0.21%	0.998±0.092%	0.905±0.21%
0.40	SRPT	1561.7±0.11%	2830.3±0.32%	266258.6±11.0%	0.945±0.21%	0.999±0.09%	0.945±0.21%
0.50	FF	2040.0±1.8%	9688.9±8.6%	287779.7±19.0%	0.953±0.52%	0.997±0.15%	0.953±0.52%
0.50	FS	1589.9±0.25%	2981.8±0.49%	177708.6±6.5%	0.95±0.48%	0.998±0.13%	0.95±0.48%
0.50	Rand	2120.0±1.7%	7781.7±3.5%	314269.3±12.0%	0.866±0.72%	0.996±0.14%	0.866±0.72%
0.50	SRPT	1589.3±0.16%	2963.8±0.28%	306084.3±13.0%	0.94±0.52%	0.998±0.14%	0.94±0.52%
0.60	FF	2468.3±1.2%	14704.2±3.3%	311801.5±8.9%	0.956±0.3%	0.998±0.042%	0.956±0.3%
0.60	FS	1620.5±0.097%	3756.5±0.77%	197184.5±4.0%	0.954±0.23%	0.999±0.038%	0.954±0.23%
0.60	Rand	3082.2±1.8%	15591.2±3.3%	430919.2±2.4%	0.815±0.9%	0.995±0.056%	0.815±0.9%
0.60	SRPT	1633.7±0.19%	3493.7±1.2%	337388.2±4.1%	0.94±0.21%	0.999±0.038%	0.94±0.21%
0.70	FF	3267.8±2.8%	23735.1±7.2%	301004.1±4.5%	0.939±0.27%	0.995±0.062%	0.939±0.27%
0.70	FS	1659.3±0.14%	4784.4±0.97%	251399.3±4.4%	0.937±0.32%	0.999±0.052%	0.937±0.32%
0.70	Rand	4312.8±1.8%	23854.6±2.8%	362330.2±2.8%	0.751±1.2%	0.988±0.088%	0.751±1.2%
0.70	SRPT	1695.4±0.39%	4072.7±2.1%	320406.5±3.1%	0.918±0.35%	0.999±0.054%	0.918±0.35%
0.79	FF	4478.6±1.1%	36615.6±3.5%	307393.6±1.8%	0.905±0.4%	0.989±0.085%	0.905±0.4%
0.79	FS	1763.6±0.15%	7054.1±1.4%	269808.6±5.0%	0.896±0.29%	0.998±0.067%	0.896±0.29%
0.79	Rand	5939.0±1.5%	33275.3±2.1%	332125.7±1.8%	0.679±1.2%	0.977±0.087%	0.679±1.2%
0.79	SRPT	1792.6±0.76%	5219.3±3.7%	303203.6±3.1%	0.842±0.32%	0.998±0.071%	0.842±0.32%
0.90	FF	6062.3±2.5%	48771.8±3.1%	278389.8±2.2%	0.852±0.38%	0.979±0.075%	0.852±0.38%
0.90	FS	1924.4±0.59%	10517.0±2.1%	284887.0±4.4%	0.819±0.73%	0.997±0.1%	0.819±0.73%
0.90	Rand	7280.9±2.0%	40622.7±2.7%	304640.3±2.2%	0.621±0.53%	0.965±0.071%	0.621±0.53%
0.90	SRPT	1905.5±0.72%	6722.4±2.8%	288426.4±2.8%	0.751±0.27%	0.997±0.095%	0.751±0.27%

TABLE B.9: Scheduler performance summary with 95% confidence intervals for the **skewed_nodes_sensitivity_0.05** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1589.7±0.2%	3580.9±0.61%	72990.8±4.8%	0.995±0.13%	0.999±0.044%	0.995±0.13%
0.10	FS	1526.7±0.14%	1998.8±0.016%	70198.0±4.2%	0.995±0.14%	0.999±0.045%	0.995±0.14%
0.10	Rand	1554.9±0.18%	2849.2±0.33%	91598.0±5.9%	0.995±0.14%	0.999±0.045%	0.995±0.14%
0.10	SRPT	1520.9±0.11%	1998.2±0.0098%	81555.4±5.0%	0.995±0.13%	0.999±0.045%	0.995±0.13%
0.20	FF	1904.3±1.3%	11165.2±7.4%	170783.1±17.0%	0.966±0.67%	0.998±0.044%	0.966±0.67%
0.20	FS	1575.6±0.12%	2708.5±4.0%	172624.9±5.3%	0.957±0.77%	0.999±0.045%	0.957±0.77%
0.20	Rand	1641.1±0.21%	3799.1±0.95%	258243.8±8.6%	0.901±0.8%	0.999±0.047%	0.901±0.8%
0.20	SRPT	1542.8±0.11%	2384.0±1.5%	237546.1±11.0%	0.951±0.71%	0.999±0.046%	0.951±0.71%
0.30	FF	2110.4±5.3%	13637.6±22.0%	364074.5±7.9%	0.922±0.62%	0.997±0.057%	0.922±0.62%
0.30	FS	1695.1±0.62%	6015.8±11.0%	348982.1±3.3%	0.908±0.47%	0.999±0.031%	0.908±0.47%
0.30	Rand	1734.6±1.1%	5030.5±5.8%	329509.4±5.2%	0.871±0.9%	0.999±0.036%	0.871±0.9%
0.30	SRPT	1551.9±0.15%	2671.0±0.84%	347195.5±7.9%	0.911±0.58%	0.999±0.031%	0.911±0.58%
0.40	FF	1757.4±0.3%	5007.6±1.5%	232866.4±6.1%	0.933±0.32%	0.998±0.086%	0.933±0.32%
0.40	FS	1640.0±0.54%	2879.0±1.3%	290705.3±4.7%	0.903±0.38%	0.998±0.086%	0.903±0.38%
0.40	Rand	1738.4±0.63%	4569.2±1.4%	253293.4±7.2%	0.869±0.54%	0.998±0.093%	0.869±0.54%
0.40	SRPT	1564.4±0.083%	2821.2±0.2%	236239.4±6.3%	0.909±0.35%	0.999±0.087%	0.909±0.35%
0.50	FF	1890.3±0.82%	6780.9±4.0%	309771.9±13.0%	0.936±0.71%	0.999±0.036%	0.936±0.71%
0.50	FS	1624.6±0.76%	3202.1±3.5%	263314.3±5.5%	0.924±0.54%	0.999±0.03%	0.924±0.54%
0.50	Rand	1921.4±0.49%	6121.4±1.8%	344062.4±11.0%	0.865±0.87%	0.998±0.036%	0.865±0.87%
0.50	SRPT	1590.1±0.1%	3024.8±0.66%	345835.9±12.0%	0.912±0.45%	0.999±0.031%	0.912±0.45%
0.60	FF	2228.0±1.3%	11127.2±4.4%	325509.6±4.4%	0.941±0.43%	0.998±0.063%	0.941±0.43%
0.60	FS	1619.8±0.32%	3642.9±1.6%	278038.2±4.2%	0.935±0.29%	0.999±0.041%	0.935±0.29%
0.60	Rand	2611.0±3.2%	11568.8±5.7%	414642.6±4.4%	0.839±0.76%	0.996±0.064%	0.839±0.76%
0.60	SRPT	1634.8±0.17%	3676.2±1.4%	310853.9±6.2%	0.915±0.3%	0.999±0.044%	0.915±0.3%
0.70	FF	2875.4±1.1%	18523.2±3.0%	278140.2±6.7%	0.932±0.48%	0.996±0.055%	0.932±0.48%
0.70	FS	1653.5±0.21%	4697.6±0.6%	230876.0±2.8%	0.935±0.28%	0.999±0.042%	0.935±0.28%
0.70	Rand	4114.6±1.3%	22245.1±2.0%	368369.0±2.3%	0.784±0.92%	0.99±0.13%	0.784±0.92%
0.70	SRPT	1719.6±0.24%	4455.5±1.8%	254458.0±2.9%	0.904±0.36%	0.999±0.044%	0.904±0.36%
0.80	FF	4161.5±3.0%	32209.9±5.0%	290395.7±3.7%	0.908±0.36%	0.989±0.055%	0.908±0.36%
0.80	FS	1754.4±0.31%	7051.4±1.7%	270181.8±3.0%	0.9±0.21%	0.998±0.081%	0.9±0.21%
0.80	Rand	5293.0±2.7%	29396.4±3.3%	306156.3±1.7%	0.719±1.1%	0.978±0.095%	0.719±1.1%
0.80	SRPT	1862.6±0.57%	6197.4±2.1%	296322.5±3.7%	0.858±0.27%	0.997±0.088%	0.858±0.27%
0.89	FF	6157.7±1.4%	49317.2±1.9%	281000.4±3.5%	0.862±0.28%	0.979±0.12%	0.862±0.28%
0.89	FS	1936.4±0.45%	11082.2±2.2%	272673.6±2.7%	0.831±0.44%	0.997±0.071%	0.831±0.44%
0.89	Rand	7365.4±0.86%	41371.3±1.0%	297468.5±1.7%	0.639±0.88%	0.964±0.12%	0.639±0.88%
0.89	SRPT	2185.8±3.7%	11390.2±12.0%	265498.8±1.9%	0.726±0.49%	0.996±0.086%	0.726±0.49%

TABLE B.10: Scheduler performance summary with 95% confidence intervals for the **skewed_nodes_sensitivity_0.1** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1555.2±0.21%	2960.3±0.45%	63958.8±7.2%	0.995±0.078%	1.0±0.026%	0.995±0.078%
0.10	FS	1518.9±0.18%	1997.5±0.0059%	53307.4±6.3%	0.995±0.11%	1.0±0.025%	0.995±0.11%
0.10	Rand	1544.8±0.22%	2746.4±0.76%	80003.7±8.1%	0.993±0.11%	1.0±0.025%	0.993±0.11%
0.10	SRPT	1515.5±0.18%	1997.1±0.0039%	55035.6±6.0%	0.996±0.068%	1.0±0.025%	0.996±0.068%
0.20	FF	1653.6±0.43%	4948.5±3.1%	100796.4±7.0%	0.98±0.39%	0.999±0.063%	0.98±0.39%
0.20	FS	1538.1±0.21%	2078.9±1.6%	86478.5±8.6%	0.978±0.44%	0.999±0.059%	0.978±0.44%
0.20	Rand	1604.6±0.33%	3121.7±1.8%	210697.6±5.6%	0.937±0.7%	0.999±0.06%	0.937±0.7%
0.20	SRPT	1529.4±0.1%	2199.9±0.86%	107139.2±10.0%	0.979±0.48%	0.999±0.058%	0.979±0.48%
0.30	FF	1879.0±1.4%	9864.3±5.0%	219455.7±9.2%	0.968±0.39%	0.998±0.059%	0.968±0.39%
0.30	FS	1587.8±0.47%	2934.7±2.7%	165814.7±9.3%	0.965±0.38%	0.999±0.06%	0.965±0.38%
0.30	Rand	1688.9±0.27%	4259.1±1.6%	388493.8±6.6%	0.811±1.2%	0.998±0.063%	0.811±1.2%
0.30	SRPT	1555.3±0.14%	2580.5±0.5%	318733.6±8.1%	0.947±0.66%	0.999±0.061%	0.947±0.66%
0.40	FF	2047.8±1.9%	12996.0±8.4%	253351.7±3.1%	0.901±0.99%	0.997±0.077%	0.901±0.99%
0.40	FS	1656.3±0.3%	4469.8±0.9%	237826.1±6.6%	0.901±0.88%	0.999±0.058%	0.901±0.88%
0.40	Rand	1750.5±0.46%	5216.2±1.9%	272425.1±5.7%	0.774±0.66%	0.997±0.059%	0.774±0.66%
0.40	SRPT	1565.6±0.083%	2783.8±0.28%	235162.6±6.6%	0.88±0.78%	0.999±0.058%	0.88±0.78%
0.50	FF	1893.6±1.5%	6355.4±4.8%	440695.9±8.9%	0.887±0.41%	0.998±0.073%	0.887±0.41%
0.50	FS	1752.4±1.1%	5396.1±12.0%	331678.2±8.6%	0.888±1.2%	0.999±0.052%	0.888±1.2%
0.50	Rand	1941.6±1.1%	6437.1±3.0%	458290.7±9.6%	0.795±0.49%	0.998±0.056%	0.795±0.49%
0.50	SRPT	1588.2±0.059%	2940.4±0.42%	415335.3±7.7%	0.856±0.52%	0.999±0.056%	0.856±0.52%
0.61	FF	1981.7±0.88%	7326.2±3.0%	372958.5±3.3%	0.901±0.21%	0.998±0.043%	0.901±0.21%
0.61	FS	1692.9±0.47%	3992.5±2.3%	297476.6±4.3%	0.898±0.22%	0.999±0.037%	0.898±0.22%
0.61	Rand	2203.6±1.2%	8062.8±2.4%	407016.8±2.1%	0.801±0.43%	0.997±0.043%	0.801±0.43%
0.61	SRPT	1638.4±0.15%	3706.4±0.78%	327127.1±6.4%	0.863±0.25%	0.999±0.039%	0.863±0.25%
0.70	FF	2412.4±0.75%	12132.2±2.0%	307320.4±1.3%	0.897±0.33%	0.997±0.054%	0.897±0.33%
0.70	FS	1671.6±0.3%	4565.0±1.8%	292849.3±3.5%	0.906±0.18%	0.999±0.051%	0.906±0.18%
0.70	Rand	3156.7±0.98%	15098.8±1.5%	369120.8±2.5%	0.782±0.31%	0.993±0.057%	0.782±0.31%
0.70	SRPT	1756.4±0.26%	5157.5±1.5%	326751.6±4.6%	0.862±0.16%	0.999±0.053%	0.862±0.16%
0.80	FF	3541.7±0.85%	24415.3±1.9%	304075.1±3.6%	0.892±0.17%	0.993±0.075%	0.892±0.17%
0.80	FS	1731.9±0.23%	6430.1±1.5%	234881.5±4.1%	0.901±0.21%	0.999±0.061%	0.901±0.21%
0.80	Rand	5311.2±3.2%	30099.0±3.7%	329220.0±1.8%	0.728±1.2%	0.98±0.12%	0.728±1.2%
0.80	SRPT	2006.7±0.79%	8444.4±3.0%	291953.9±5.8%	0.833±0.37%	0.998±0.059%	0.833±0.37%
0.90	FF	6282.4±3.1%	51863.0±4.5%	361626.4±12.0%	0.876±0.5%	0.982±0.17%	0.876±0.5%
0.90	FS	2051.1±0.77%	13365.0±2.2%	340927.7±9.7%	0.869±1.1%	0.997±0.092%	0.869±1.1%
0.90	Rand	9434.3±8.7%	55751.7±9.7%	394053.6±12.0%	0.677±0.61%	0.962±0.14%	0.677±0.61%
0.90	SRPT	2864.2±4.1%	23077.6±10.0%	394099.5±12.0%	0.731±1.4%	0.987±0.23%	0.731±1.4%

TABLE B.11: Scheduler performance summary with 95% confidence intervals for the **skewed_nodes_sensitivity_0.2** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.10	FF	1550.7±0.14%	2940.7±0.61%	41390.8±6.7%	0.997±0.072%	1.0±0.018%	0.997±0.072%
0.10	FS	1516.9±0.13%	1997.5±0.0078%	40765.3±4.2%	0.997±0.07%	1.0±0.018%	0.997±0.07%
0.10	Rand	1542.2±0.16%	2746.4±0.23%	62404.4±12.0%	0.997±0.082%	1.0±0.017%	0.997±0.082%
0.10	SRPT	1516.3±0.13%	1997.9±0.0098%	41765.3±4.4%	0.997±0.07%	1.0±0.018%	0.997±0.07%
0.20	FF	1626.3±0.32%	4422.1±2.7%	55331.8±5.9%	0.98±0.11%	0.999±0.044%	0.98±0.11%
0.20	FS	1527.7±0.11%	2008.0±0.33%	46606.0±7.2%	0.981±0.11%	0.999±0.045%	0.981±0.11%
0.20	Rand	1582.2±0.082%	2999.5±0.33%	98692.2±7.2%	0.961±0.22%	0.999±0.044%	0.961±0.22%
0.20	SRPT	1528.6±0.12%	2280.1±0.98%	53343.4±7.0%	0.983±0.22%	0.999±0.045%	0.983±0.22%
0.30	FF	1748.0±0.97%	6884.9±6.0%	70468.9±9.3%	0.99±0.19%	0.999±0.084%	0.99±0.19%
0.30	FS	1534.4±0.1%	2385.5±0.78%	68968.2±8.7%	0.991±0.2%	0.999±0.084%	0.991±0.2%
0.30	Rand	1662.0±0.58%	3976.1±1.9%	232725.1±11.0%	0.972±0.47%	0.999±0.085%	0.972±0.47%
0.30	SRPT	1538.9±0.064%	2658.7±0.41%	85328.2±9.3%	0.991±0.18%	0.999±0.084%	0.991±0.18%
0.40	FF	1940.0±0.89%	9772.0±3.0%	88904.4±3.9%	0.981±0.23%	0.998±0.086%	0.981±0.23%
0.40	FS	1552.0±0.17%	2718.8±0.66%	81504.4±5.4%	0.983±0.25%	0.999±0.082%	0.983±0.25%
0.40	Rand	1836.8±0.6%	5756.4±1.2%	274773.4±3.7%	0.908±0.39%	0.998±0.085%	0.908±0.39%
0.40	SRPT	1561.4±0.11%	2844.9±0.29%	111871.6±4.0%	0.981±0.24%	0.999±0.082%	0.981±0.24%
0.51	FF	2329.1±1.4%	16228.3±6.2%	218249.3±18.0%	0.97±0.62%	0.997±0.1%	0.97±0.62%
0.51	FS	1576.2±0.19%	3237.0±1.5%	120960.8±6.1%	0.972±0.6%	0.999±0.078%	0.972±0.6%
0.51	Rand	2429.3±3.3%	11991.0±7.5%	422835.2±8.8%	0.826±0.52%	0.995±0.089%	0.826±0.52%
0.51	SRPT	1592.1±0.16%	2987.7±0.52%	263110.8±12.0%	0.967±0.66%	0.999±0.078%	0.967±0.66%
0.60	FF	2939.0±2.1%	23736.5±4.0%	343896.8±6.1%	0.948±0.36%	0.996±0.045%	0.948±0.36%
0.60	FS	1633.6±0.4%	4389.9±2.1%	258643.0±5.9%	0.959±0.32%	0.999±0.043%	0.959±0.32%
0.60	Rand	3201.2±2.1%	19085.7±4.6%	436718.7±1.5%	0.766±1.7%	0.993±0.055%	0.766±1.7%
0.60	SRPT	1632.6±0.094%	3514.9±1.3%	323235.8±4.6%	0.934±0.47%	0.999±0.044%	0.934±0.47%
0.71	FF	3837.7±1.9%	34431.1±4.0%	322903.8±2.5%	0.911±0.37%	0.992±0.084%	0.911±0.37%
0.71	FS	1730.6±0.38%	6601.9±1.2%	274442.8±2.9%	0.922±0.39%	0.999±0.05%	0.922±0.39%
0.71	Rand	3911.6±1.4%	24538.5±3.2%	381889.7±2.0%	0.731±1.2%	0.989±0.076%	0.731±1.2%
0.71	SRPT	1706.2±0.21%	4321.1±1.7%	365187.1±2.6%	0.886±0.35%	0.999±0.054%	0.886±0.35%
0.80	FF	4505.2±3.5%	40048.5±6.2%	297883.9±2.8%	0.854±0.38%	0.985±0.18%	0.854±0.38%
0.80	FS	1843.6±0.83%	9336.0±3.2%	284147.5±4.5%	0.856±0.62%	0.997±0.095%	0.856±0.62%
0.80	Rand	4761.7±2.7%	28060.7±2.3%	315479.6±2.5%	0.694±0.71%	0.982±0.12%	0.694±0.71%
0.80	SRPT	1807.6±0.51%	5691.1±2.9%	275652.6±5.5%	0.819±0.61%	0.998±0.1%	0.819±0.61%
0.89	FF	5277.0±2.7%	49286.4±4.5%	301906.9±1.8%	0.814±0.44%	0.97±0.43%	0.814±0.44%
0.89	FS	2042.1±0.48%	14036.0±2.1%	273754.2±3.0%	0.79±0.55%	0.996±0.11%	0.79±0.55%
0.89	Rand	7441.6±3.5%	41471.0±2.8%	294162.8±1.2%	0.633±1.2%	0.964±0.2%	0.633±1.2%
0.89	SRPT	2271.4±3.0%	14379.1±11.0%	294316.1±2.1%	0.746±0.67%	0.992±0.29%	0.746±0.67%

TABLE B.12: Scheduler performance summary with 95% confidence intervals for the **skewed_nodes_sensitivity_0.4** benchmark.

Rack Distribution Benchmark

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.1	FF	1547.1±0.2%	2905.6±0.53%	36420.2±3.8%	0.993±0.23%	0.999±0.047%	0.993±0.23%
0.1	FS	1514.8±0.14%	1997.1±0.0059%	35026.4±2.6%	0.994±0.23%	0.999±0.046%	0.994±0.23%
0.1	Rand	1538.9±0.15%	2708.2±0.35%	53118.6±2.9%	0.991±0.21%	0.999±0.046%	0.991±0.21%
0.1	SRPT	1514.5±0.14%	1997.6±0.0078%	35426.4±2.6%	0.994±0.22%	0.999±0.046%	0.994±0.22%
0.2	FF	1613.3±0.16%	4210.8±1.1%	43491.3±3.3%	0.985±0.34%	0.999±0.032%	0.985±0.34%
0.2	FS	1522.9±0.14%	1998.9±0.012%	38988.6±1.6%	0.986±0.37%	1.0±0.029%	0.986±0.37%
0.2	Rand	1575.6±0.2%	3009.0±0.77%	70182.4±2.6%	0.978±0.45%	0.999±0.029%	0.978±0.45%
0.2	SRPT	1524.5±0.14%	2252.2±1.0%	41095.6±1.6%	0.987±0.31%	1.0±0.028%	0.987±0.31%
0.3	FF	1751.4±0.79%	6744.9±3.8%	67480.0±9.5%	0.989±0.17%	0.999±0.042%	0.989±0.17%
0.3	FS	1534.7±0.13%	2247.8±0.56%	63424.2±7.2%	0.99±0.17%	0.999±0.042%	0.99±0.17%
0.3	Rand	1649.2±0.34%	3891.9±0.76%	148433.8±8.6%	0.977±0.19%	0.999±0.041%	0.977±0.19%
0.3	SRPT	1539.5±0.12%	2626.4±0.49%	83252.7±9.4%	0.989±0.2%	0.999±0.041%	0.989±0.2%
0.4	FF	1924.1±1.5%	9755.3±7.2%	88414.1±9.8%	0.977±0.23%	0.998±0.086%	0.977±0.23%
0.4	FS	1541.5±0.092%	2542.4±0.5%	74926.1±11.0%	0.98±0.19%	0.999±0.085%	0.98±0.19%
0.4	Rand	1795.0±0.49%	5339.0±1.4%	216058.0±7.5%	0.941±0.46%	0.998±0.089%	0.941±0.46%
0.4	SRPT	1552.2±0.035%	2802.5±0.34%	99179.4±14.0%	0.979±0.2%	0.999±0.085%	0.979±0.2%
0.5	FF	2239.7±2.0%	14440.3±7.6%	120877.0±5.2%	0.979±0.27%	0.998±0.048%	0.979±0.27%
0.5	FS	1564.2±0.13%	2914.5±0.6%	97264.9±6.6%	0.98±0.21%	0.999±0.05%	0.98±0.21%
0.5	Rand	2330.3±1.7%	9746.8±4.0%	408828.1±10.0%	0.892±1.0%	0.997±0.055%	0.892±1.0%
0.5	SRPT	1580.9±0.082%	2940.4±0.36%	153416.6±10.0%	0.978±0.3%	0.999±0.051%	0.978±0.3%
0.6	FF	2842.5±2.5%	22991.2±7.0%	308474.4±6.6%	0.967±0.28%	0.996±0.067%	0.967±0.28%
0.6	FS	1595.6±0.19%	3658.7±0.66%	137386.1±5.7%	0.972±0.28%	0.999±0.048%	0.972±0.28%
0.6	Rand	3265.3±0.75%	16613.3±1.2%	420951.4±3.7%	0.825±0.81%	0.994±0.065%	0.825±0.81%
0.6	SRPT	1619.1±0.097%	3390.5±1.8%	336922.0±5.1%	0.961±0.37%	0.999±0.049%	0.961±0.37%
0.7	FF	3465.2±0.49%	27554.1±1.9%	287240.4±6.0%	0.95±0.29%	0.994±0.066%	0.95±0.29%
0.7	FS	1648.7±0.15%	4775.0±1.0%	210756.3±2.1%	0.95±0.28%	0.999±0.063%	0.95±0.28%
0.7	Rand	4658.5±2.2%	25482.9±2.8%	345529.5±3.4%	0.755±0.84%	0.985±0.055%	0.755±0.84%
0.7	SRPT	1678.0±0.25%	3916.3±1.2%	307069.7±3.6%	0.927±0.29%	0.999±0.063%	0.927±0.29%
0.8	FF	4604.8±1.5%	37588.2±2.3%	287174.0±1.8%	0.904±0.52%	0.988±0.11%	0.904±0.52%
0.8	FS	1759.3±0.18%	7189.5±1.6%	278549.2±2.2%	0.886±0.5%	0.998±0.071%	0.886±0.5%
0.8	Rand	5891.2±0.77%	32310.3±1.1%	323761.6±2.1%	0.694±1.3%	0.977±0.11%	0.694±1.3%
0.8	SRPT	1757.8±0.7%	4908.0±3.1%	307367.1±4.4%	0.853±0.39%	0.998±0.073%	0.853±0.39%
0.9	FF	6385.1±2.0%	52863.8±3.2%	320436.3±9.5%	0.871±0.69%	0.98±0.041%	0.871±0.69%
0.9	FS	1956.4±1.1%	11288.2±3.1%	313425.0±11.0%	0.845±0.79%	0.998±0.034%	0.845±0.79%
0.9	Rand	8399.6±6.6%	46907.3±7.9%	336830.9±12.0%	0.65±0.81%	0.964±0.082%	0.65±0.81%
0.9	SRPT	1963.3±1.2%	7596.5±5.1%	320009.5±13.0%	0.786±0.3%	0.998±0.042%	0.786±0.3%

TABLE B.13: Scheduler performance summary with 95% confidence intervals for the **rack_sensitivity_0.2** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.1	FF	1553.9±0.15%	3023.9±1.1%	38020.6±5.6%	0.997±0.15%	1.0±0.029%	0.997±0.15%
0.1	FS	1515.8±0.057%	1997.5±0.0039%	37020.6±5.8%	0.997±0.15%	1.0±0.026%	0.997±0.15%
0.1	Rand	1541.3±0.058%	2739.1±0.31%	61994.1±11.0%	0.996±0.16%	1.0±0.026%	0.996±0.16%
0.1	SRPT	1515.5±0.052%	1998.0±0.0078%	37373.3±6.5%	0.997±0.15%	1.0±0.026%	0.997±0.15%
0.2	FF	1643.8±0.34%	4775.7±2.4%	52879.5±3.8%	0.986±0.2%	0.999±0.04%	0.986±0.2%
0.2	FS	1525.7±0.11%	1999.3±0.0059%	48949.2±3.9%	0.987±0.18%	0.999±0.043%	0.987±0.18%
0.2	Rand	1587.8±0.18%	3035.9±0.46%	126408.0±8.0%	0.976±0.14%	0.999±0.044%	0.976±0.14%
0.2	SRPT	1526.4±0.12%	2225.3±1.5%	51165.4±4.0%	0.988±0.17%	0.999±0.044%	0.988±0.17%
0.3	FF	1787.0±0.46%	7619.8±3.9%	66882.6±8.2%	0.988±0.16%	0.999±0.016%	0.988±0.16%
0.3	FS	1532.2±0.15%	2231.4±0.81%	57004.1±7.0%	0.989±0.21%	1.0±0.015%	0.989±0.21%
0.3	Rand	1671.2±0.57%	4113.2±2.9%	256001.0±7.5%	0.956±0.31%	0.999±0.015%	0.956±0.31%
0.3	SRPT	1536.4±0.17%	2610.9±0.51%	65648.0±10.0%	0.989±0.16%	1.0±0.016%	0.989±0.16%
0.4	FF	1997.7±0.57%	11546.2±2.4%	78798.6±6.6%	0.973±0.27%	0.998±0.054%	0.973±0.27%
0.4	FS	1542.8±0.11%	2588.6±0.51%	66608.7±3.7%	0.976±0.23%	0.999±0.065%	0.976±0.23%
0.4	Rand	1805.3±0.55%	6476.5±3.3%	287594.6±1.8%	0.882±0.51%	0.998±0.074%	0.882±0.51%
0.4	SRPT	1553.2±0.061%	2820.2±0.21%	85975.8±3.3%	0.977±0.22%	0.999±0.066%	0.977±0.22%
0.5	FF	2476.5±1.5%	20978.2±5.7%	115951.9±3.9%	0.976±0.45%	0.997±0.053%	0.976±0.45%
0.5	FS	1562.6±0.05%	2906.2±0.44%	104707.6±4.4%	0.978±0.38%	0.999±0.046%	0.978±0.38%
0.5	Rand	2104.8±2.3%	11901.7±9.0%	411058.5±11.0%	0.822±0.74%	0.996±0.041%	0.822±0.74%
0.5	SRPT	1578.2±0.12%	2936.4±0.35%	128711.3±3.1%	0.976±0.36%	0.999±0.044%	0.976±0.36%
0.6	FF	2880.0±1.5%	24414.7±5.0%	242585.7±3.1%	0.971±0.31%	0.997±0.044%	0.971±0.31%
0.6	FS	1592.5±0.094%	3616.1±0.56%	131921.2±7.3%	0.972±0.22%	0.999±0.031%	0.972±0.22%
0.6	Rand	2420.9±0.64%	17877.9±2.6%	417817.1±1.5%	0.778±1.0%	0.995±0.043%	0.778±1.0%
0.6	SRPT	1619.6±0.11%	3401.4±1.4%	235066.7±3.8%	0.966±0.35%	0.999±0.033%	0.966±0.35%
0.7	FF	3534.3±1.3%	33314.7±4.1%	311692.8±3.6%	0.935±0.27%	0.994±0.052%	0.935±0.27%
0.7	FS	1642.8±0.05%	4665.5±1.2%	245550.4±4.2%	0.937±0.3%	0.999±0.057%	0.937±0.3%
0.7	Rand	2652.7±1.1%	21768.7±1.6%	375157.8±2.3%	0.71±0.89%	0.993±0.071%	0.71±0.89%
0.7	SRPT	1660.0±0.2%	3780.8±1.1%	327831.0±2.0%	0.915±0.3%	0.999±0.059%	0.915±0.3%
0.8	FF	4311.5±0.84%	39028.2±2.3%	294072.3±2.3%	0.911±0.39%	0.99±0.056%	0.911±0.39%
0.8	FS	1731.1±0.25%	6579.1±1.4%	238819.8±2.4%	0.904±0.33%	0.999±0.047%	0.904±0.33%
0.8	Rand	2906.5±0.56%	24944.4±1.5%	320552.6±1.3%	0.665±1.4%	0.99±0.063%	0.665±1.4%
0.8	SRPT	1747.3±0.58%	4642.0±2.8%	280469.5±6.0%	0.865±0.3%	0.999±0.05%	0.865±0.3%
0.9	FF	5497.1±2.4%	46230.6±2.7%	280463.7±3.8%	0.851±0.43%	0.983±0.086%	0.851±0.43%
0.9	FS	1872.4±0.26%	9593.3±1.6%	290850.1±1.7%	0.827±0.43%	0.997±0.094%	0.827±0.43%
0.9	Rand	3347.1±1.2%	29914.9±0.63%	306219.5±1.8%	0.608±1.4%	0.987±0.095%	0.608±1.4%
0.9	SRPT	1861.4±0.99%	5931.5±4.3%	286319.1±1.6%	0.781±0.33%	0.997±0.095%	0.781±0.33%

TABLE B.14: Scheduler performance summary with 95% confidence intervals for the **rack_sensitivity_0.4** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.1	FF	1557.4±0.18%	2992.3±1.3%	41866.0±3.4%	0.995±0.1%	0.999±0.045%	0.995±0.1%
0.1	FS	1519.1±0.041%	1997.0±0.012%	41866.0±2.5%	0.995±0.1%	0.999±0.044%	0.995±0.1%
0.1	Rand	1544.3±0.069%	2738.7±0.46%	63511.4±5.0%	0.992±0.23%	0.999±0.044%	0.992±0.23%
0.1	SRPT	1518.9±0.044%	1997.6±0.0098%	42866.0±3.2%	0.995±0.1%	0.999±0.044%	0.995±0.1%
0.2	FF	1639.1±0.39%	4710.0±2.0%	48916.2±3.7%	0.989±0.26%	0.999±0.039%	0.989±0.26%
0.2	FS	1522.8±0.17%	1998.9±0.0039%	47869.5±2.2%	0.989±0.27%	0.999±0.043%	0.989±0.27%
0.2	Rand	1582.2±0.16%	3046.6±0.46%	105951.9±8.2%	0.981±0.28%	0.999±0.043%	0.981±0.28%
0.2	SRPT	1525.6±0.18%	2338.1±0.83%	46897.0±4.4%	0.99±0.23%	0.999±0.043%	0.99±0.23%
0.3	FF	1786.7±0.96%	7425.5±4.7%	64656.1±7.6%	0.986±0.29%	0.999±0.047%	0.986±0.29%
0.3	FS	1531.2±0.19%	2225.5±1.8%	57743.4±4.0%	0.987±0.32%	0.999±0.046%	0.987±0.32%
0.3	Rand	1663.9±0.3%	4087.7±1.3%	233890.5±4.2%	0.967±0.37%	0.999±0.046%	0.967±0.37%
0.3	SRPT	1538.9±0.17%	2680.8±0.29%	64343.4±5.0%	0.988±0.32%	0.999±0.046%	0.988±0.32%
0.4	FF	2070.3±1.5%	11972.7±4.4%	89212.8±6.2%	0.98±0.34%	0.997±0.09%	0.98±0.34%
0.4	FS	1543.2±0.094%	2555.3±1.1%	84834.8±6.3%	0.981±0.34%	0.999±0.085%	0.981±0.34%
0.4	Rand	1804.3±0.48%	6112.3±1.9%	210247.2±7.1%	0.917±0.59%	0.998±0.088%	0.917±0.59%
0.4	SRPT	1558.0±0.083%	2842.9±0.21%	108634.8±9.3%	0.981±0.28%	0.999±0.085%	0.981±0.28%
0.5	FF	2462.2±0.96%	18251.1±3.1%	121295.6±5.9%	0.98±0.15%	0.997±0.062%	0.98±0.15%
0.5	FS	1560.7±0.05%	2885.8±0.48%	89431.3±4.1%	0.98±0.18%	0.999±0.048%	0.98±0.18%
0.5	Rand	2236.8±1.9%	13576.8±6.8%	390033.5±9.3%	0.843±0.62%	0.997±0.048%	0.843±0.62%
0.5	SRPT	1585.1±0.042%	2970.9±0.11%	136545.8±11.0%	0.98±0.17%	0.999±0.05%	0.98±0.17%
0.6	FF	2956.1±1.6%	24090.4±4.4%	242220.9±13.0%	0.975±0.13%	0.996±0.065%	0.975±0.13%
0.6	FS	1586.0±0.16%	3517.5±0.65%	138093.6±7.7%	0.979±0.1%	0.999±0.063%	0.979±0.1%
0.6	Rand	2728.2±0.78%	22105.7±1.7%	432399.7±2.9%	0.771±0.97%	0.994±0.081%	0.771±0.97%
0.6	SRPT	1624.4±0.043%	3519.2±0.73%	284839.9±4.3%	0.97±0.16%	0.999±0.064%	0.97±0.16%
0.7	FF	3858.3±1.5%	35582.9±3.2%	272734.1±8.6%	0.951±0.16%	0.993±0.051%	0.951±0.16%
0.7	FS	1630.8±0.12%	4456.6±1.0%	225655.3±3.9%	0.953±0.13%	0.999±0.058%	0.953±0.13%
0.7	Rand	3035.8±0.58%	26654.0±1.4%	356900.5±1.4%	0.708±0.38%	0.991±0.072%	0.708±0.38%
0.7	SRPT	1680.3±0.17%	3938.0±0.64%	291331.3±5.2%	0.931±0.27%	0.999±0.058%	0.931±0.27%
0.8	FF	4501.2±1.5%	38457.9±2.2%	277274.4±2.6%	0.915±0.42%	0.988±0.16%	0.915±0.42%
0.8	FS	1713.2±0.17%	6223.9±1.2%	259604.2±3.1%	0.908±0.33%	0.998±0.095%	0.908±0.33%
0.8	Rand	3484.0±2.5%	32752.7±3.8%	321488.9±4.3%	0.644±1.1%	0.986±0.087%	0.644±1.1%
0.8	SRPT	1752.6±0.43%	4662.8±1.9%	279493.1±6.2%	0.869±0.31%	0.998±0.098%	0.869±0.31%
0.9	FF	5773.6±0.51%	46545.8±0.72%	263361.6±3.2%	0.867±0.22%	0.978±0.14%	0.867±0.22%
0.9	FS	1872.7±0.3%	9645.7±1.2%	274889.1±1.4%	0.844±0.17%	0.997±0.07%	0.844±0.17%
0.9	Rand	3943.0±1.1%	39082.1±1.2%	293317.7±1.1%	0.595±0.58%	0.981±0.062%	0.595±0.58%
0.9	SRPT	1900.9±0.69%	6304.4±3.1%	264047.5±2.1%	0.793±0.19%	0.997±0.066%	0.793±0.19%

TABLE B.15: Scheduler performance summary with 95% confidence intervals for the **rack_sensitivity_0.6** benchmark.

Load	Subject	Mean FCT (μs)	p99 FCT (μs)	Max FCT (μs)	Throughput (Frac)	Flows Accepted (Frac)	Info Accepted (Frac)
0.1	FF	1564.4±0.17%	3075.8±0.78%	38130.3±3.3%	0.998±0.055%	1.0±0.032%	0.998±0.055%
0.1	FS	1523.4±0.13%	1997.4±0.002%	34026.7±2.1%	0.998±0.055%	1.0±0.031%	0.998±0.055%
0.1	Rand	1549.9±0.14%	2779.9±0.22%	60347.0±7.3%	0.996±0.08%	1.0±0.032%	0.996±0.08%
0.1	SRPT	1523.7±0.13%	1998.4±0.0039%	34147.9±2.8%	0.998±0.055%	1.0±0.032%	0.998±0.055%
0.2	FF	1655.6±0.53%	4886.6±3.3%	48187.1±7.1%	0.991±0.17%	0.998±0.1%	0.991±0.17%
0.2	FS	1525.7±0.11%	1998.9±0.0078%	41674.6±5.2%	0.991±0.16%	0.999±0.099%	0.991±0.16%
0.2	Rand	1589.2±0.13%	3175.5±1.1%	91618.8±7.3%	0.983±0.13%	0.999±0.099%	0.983±0.13%
0.2	SRPT	1528.4±0.11%	2350.2±0.88%	43538.3±5.8%	0.992±0.16%	0.999±0.098%	0.992±0.16%
0.3	FF	1812.5±0.4%	7816.8±1.9%	68547.0±5.1%	0.986±0.17%	0.999±0.049%	0.986±0.17%
0.3	FS	1532.3±0.1%	2202.5±0.51%	64297.6±5.1%	0.987±0.2%	0.999±0.053%	0.987±0.2%
0.3	Rand	1657.7±0.13%	4051.6±0.72%	227634.8±3.2%	0.972±0.33%	0.999±0.051%	0.972±0.33%
0.3	SRPT	1541.1±0.11%	2701.4±0.27%	73297.6±7.1%	0.989±0.14%	0.999±0.053%	0.989±0.14%
0.4	FF	2211.4±1.3%	15442.4±5.6%	107060.7±14.0%	0.976±0.26%	0.997±0.065%	0.976±0.26%
0.4	FS	1546.2±0.12%	2605.9±0.68%	76600.8±12.0%	0.98±0.3%	0.999±0.046%	0.98±0.3%
0.4	Rand	1823.3±0.45%	6253.9±1.4%	256431.0±7.3%	0.918±0.43%	0.998±0.047%	0.918±0.43%
0.4	SRPT	1560.4±0.088%	2854.5±0.31%	86910.4±10.0%	0.98±0.22%	0.999±0.047%	0.98±0.22%
0.5	FF	2670.2±1.5%	20930.8±5.2%	142824.2±12.0%	0.963±0.51%	0.995±0.094%	0.963±0.51%
0.5	FS	1561.9±0.088%	2883.4±0.46%	87631.8±5.5%	0.968±0.45%	0.999±0.086%	0.968±0.45%
0.5	Rand	2097.1±1.6%	10368.7±5.3%	266946.9±14.0%	0.846±0.85%	0.996±0.093%	0.846±0.85%
0.5	SRPT	1586.5±0.14%	2988.9±0.73%	108035.3±3.4%	0.968±0.46%	0.999±0.087%	0.968±0.46%
0.6	FF	3437.5±0.59%	30455.8±3.2%	221359.1±14.0%	0.971±0.22%	0.995±0.097%	0.971±0.22%
0.6	FS	1589.9±0.079%	3541.0±0.89%	121075.9±9.3%	0.978±0.15%	0.999±0.066%	0.978±0.15%
0.6	Rand	3021.9±1.6%	24451.3±2.3%	412148.6±1.0%	0.771±0.67%	0.993±0.083%	0.771±0.67%
0.6	SRPT	1632.0±0.072%	3575.4±0.19%	219688.9±7.6%	0.97±0.28%	0.999±0.069%	0.97±0.28%
0.7	FF	4226.4±1.0%	37246.2±1.9%	250830.8±3.5%	0.955±0.43%	0.992±0.1%	0.955±0.43%
0.7	FS	1630.6±0.12%	4431.6±0.92%	200199.1±3.6%	0.961±0.23%	0.999±0.077%	0.961±0.23%
0.7	Rand	3899.6±1.9%	35618.2±3.2%	367726.5±1.9%	0.684±0.88%	0.988±0.14%	0.684±0.88%
0.7	SRPT	1694.3±0.1%	4009.5±0.56%	299390.8±5.7%	0.936±0.3%	0.999±0.079%	0.936±0.3%
0.8	FF	5264.1±1.5%	44602.4±1.7%	284358.6±6.4%	0.905±0.58%	0.985±0.066%	0.905±0.58%
0.8	FS	1721.5±0.35%	6287.8±2.1%	249298.1±4.5%	0.907±0.49%	0.998±0.076%	0.907±0.49%
0.8	Rand	4485.1±1.8%	44277.7±2.8%	331280.5±2.2%	0.59±0.66%	0.98±0.1%	0.59±0.66%
0.8	SRPT	1772.7±0.17%	4871.1±0.47%	308528.4±3.4%	0.871±0.35%	0.998±0.073%	0.871±0.35%
0.9	FF	6797.9±2.0%	53200.8±2.1%	312515.9±11.0%	0.866±0.59%	0.977±0.08%	0.866±0.59%
0.9	FS	1891.3±1.2%	10007.9±4.7%	324448.5±7.9%	0.856±0.93%	0.998±0.036%	0.856±0.93%
0.9	Rand	5968.6±7.1%	63779.3±11.0%	351222.1±11.0%	0.54±0.64%	0.971±0.097%	0.54±0.64%
0.9	SRPT	1935.1±0.79%	6647.8±2.6%	315660.8±11.0%	0.792±0.17%	0.998±0.049%	0.792±0.17%

TABLE B.16: Scheduler performance summary with 95% confidence intervals for the **rack_sensitivity_0.8** benchmark.

B.6.4 Winner Tables

The below 'winner tables' summarise the winning schedulers for each load and benchmark with their performance improvement relative to the worst performing baseline for each P_{KPI} averaged across 5 runs. These tables are useful for gaining an overarching view of the multi-faceted performance results which are often difficult to interpret through graphical means alone.

Load	Mean FCT	p99 FCT	Max FCT	Throughput	Flows Accepted
0.1	SRPT, -2.3%	FS, -33%	FF, -36%	FF+FS+SRPT, 0.40%	–
0.2	FS, -6.0%	FS, -55%	FS, -47%	SRPT, 0.92%	–
0.3	FS, -12%	FS, -66%	FS, -60%	SRPT, 1.7%	–
0.4	FS, -19%	FS, -73%	FS, -64%	FF, 3.7%	FS+Rand+SRPT, 0.10%
0.5	FS, -31%	FS, -80%	FF, -75%	FS, 11%	FS+SRPT, 0.21%
0.6	FS, -52%	SRPT, -83%	FS, -60%	FS, 19%	FS+SRPT, 0.60%
0.7	FS, -62%	SRPT, -86%	FS, -28%	FS, 28%	FS, 1.3%
0.8	FS+SRPT, -69%	SRPT, -86%	FS, -14%	FF, 31%	FS+SRPT, 2.1%
0.9	SRPT, -73%	SRPT, -85%	FF, -9.1%	FF, 35%	FS+SRPT, 3.1%
0.1	SRPT, -2.107%	FS, -31.27%	FS, -34.06%	FS+SRPT, 0.3027%	–
0.2	FS, -5.603%	FS, -52.53%	FS, -44.45%	SRPT, 0.9202%	FS+SRPT, 0.1001%
0.3	FS, -12.37%	FS, -66.67%	FS, -57.27%	FS, 1.331%	–
0.4	FS, -19.88%	FS, -73.94%	FS, -65.32%	FS, 4.145%	FS+SRPT, 0.1002%
0.5	FS, -32.88%	FS, -79.82%	FS, -76.21%	FS, 9.865%	FS+SRPT, 0.2006%
0.6	FS, -51.13%	SRPT, -85.25%	FS, -67.36%	FS, 17.82%	FS+SRPT, 0.503%
0.7	FS, -64.61%	SRPT, -85.79%	FS, -39.0%	FF+FS, 25.83%	FS+SRPT, 1.421%
0.8	SRPT, -70.16%	SRPT, -86.94%	FS, -13.96%	FF, 30.26%	FS+SRPT, 2.149%
0.9	FS, -76.71%	SRPT, -85.63%	FS, -6.949%	FF, 34.0%	FS+SRPT, 3.527%
0.1	SRPT, -2.471%	FS, -33.94%	FS, -40.28%	FF+FS+SRPT, 0.1004%	–
0.2	FS, -7.185%	FS, -58.14%	FS, -61.28%	SRPT, 1.23%	–
0.3	FS, -14.26%	FS, -70.72%	FS, -77.73%	FS+SRPT, 3.452%	FS+SRPT, 0.1001%
0.4	FS, -22.77%	FS, -77.58%	FS, -76.84%	SRPT, 10.77%	FS+SRPT, 0.1002%
0.5	FS, -36.9%	FS, -86.15%	FS, -74.53%	FS, 18.98%	FS+SRPT, 0.3012%
0.6	FS, -44.7%	SRPT, -86.07%	FS, -68.43%	FS, 24.94%	FS+SRPT, 0.402%
0.7	FS, -53.52%	SRPT, -88.65%	FS, -34.55%	FS, 31.97%	FS+SRPT, 0.6042%
0.8	FS, -59.85%	SRPT, -88.11%	FS, -25.5%	FF, 36.99%	FS+SRPT, 0.9091%
0.9	SRPT, -66.14%	SRPT, -87.17%	FF, -8.411%	FF, 39.97%	FS+SRPT, 1.424%
0.1	SRPT, -2.472%	FS, -33.26%	FF+FS, -34.08%	FF+FS+SRPT, 0.3024%	–
0.2	FS, -7.095%	FS, -57.56%	SRPT, -55.74%	SRPT, 0.9174%	–
0.3	FS, -14.3%	FS, -70.03%	FS, -75.31%	SRPT, 2.172%	–
0.4	FS, -25.46%	FS, -78.66%	FS, -59.65%	FS+SRPT, 6.979%	FS+SRPT, 0.2006%
0.5	FS, -36.61%	FS, -84.19%	FS, -77.07%	FF+FS+SRPT, 16.25%	FS+SRPT, 0.2006%
0.6	FS, -46.35%	FS, -85.4%	FS, -68.06%	FS, 26.98%	FS+SRPT, 0.503%
0.7	FS, -57.73%	SRPT, -88.93%	FS, -36.77%	FS, 34.6%	FS+SRPT, 0.8073%
0.8	FS, -61.94%	SRPT, -87.88%	FS, -19.25%	FF, 42.08%	FS+SRPT, 1.217%
0.9	FS, -67.56%	SRPT, -86.46%	FF, -10.21%	FF, 45.71%	FS+SRPT, 1.943%
0.1	FS, -2.621%	FS, -35.06%	FS, -43.61%	FF+FS+SRPT, 0.2008%	–
0.2	FS, -7.846%	FS, -59.09%	FS, -54.51%	SRPT, 0.9156%	FS+Rand+SRPT, 0.1002%
0.3	FS, -15.46%	FS, -71.82%	FS, -71.75%	SRPT, 1.749%	–
0.4	FS, -30.08%	FS, -83.13%	FS, -70.13%	FS+SRPT, 6.754%	FS+SRPT, 0.2006%
0.5	FS, -41.51%	FS, -86.22%	FS, -67.17%	FS+SRPT, 14.42%	FS+SRPT, 0.402%
0.6	FS, -53.75%	FS, -88.37%	FS, -70.62%	FS, 26.85%	FS+SRPT, 0.6042%
0.7	FS, -61.42%	SRPT, -89.24%	FS, -45.56%	FS, 40.5%	FS+SRPT, 1.113%
0.8	FS, -67.3%	SRPT, -89.08%	FS, -24.75%	FS, 53.73%	FS+SRPT, 1.837%
0.9	FS, -72.18%	SRPT, -89.58%	FF, -11.02%	FF, 60.37%	FS+SRPT, 2.781%

TABLE B.17: The winning schedulers' performances relative to the losing baselines for (from top to bottom) the 0 (uniform), $0.2, 0.4, 0.6$, and 0.8 rack sensitivity traces. For brevity, '–' indicates all schedulers' performances were equal.

Load	Mean FCT	p99 FCT	Max FCT	Throughput	Flows Accepted
0.1	SRPT, -2.329%	FS, -32.9%	FF, -36.37%	FF+FS+SRPT, 0.4036%	–
0.2	FS, -5.954%	FS, -54.54%	FS, -46.79%	SRPT, 0.924%	–
0.3	FS, -12.11%	FS, -65.63%	FS, -60.37%	SRPT, 1.747%	–
0.4	FS, -19.44%	FS, -72.55%	FS, -64.36%	FF, 3.7%	FS+Rand+SRPT, 0.1002%
0.5	FS, -30.78%	FS, -80.21%	FF, -75.05%	FS, 11.1%	FS+SRPT, 0.2006%
0.6	FS, -51.79%	SRPT, -82.76%	FS, -59.8%	FS, 18.66%	FS+SRPT, 0.6042%
0.7	FS, -62.2%	SRPT, -86.3%	FS, -28.06%	FS, 27.64%	FS, 1.318%
0.8	FS+SRPT, -69.49%	SRPT, -86.09%	FS, -13.95%	FF, 30.88%	FS+SRPT, 2.149%
0.9	SRPT, -73.35%	SRPT, -84.72%	FF, -9.119%	FF, 34.93%	FS+SRPT, 3.099%
0.10	SRPT, -8.757%	SRPT, -59.75%	FS, -45.46%	FF, 0.8114%	–
0.20	SRPT, -12.59%	SRPT, -53.34%	SRPT, -41.18%	SRPT, 4.129%	SRPT, 0.2006%
0.30	SRPT, -8.624%	FS, -43.4%	Rand, -29.39%	FF, 4.171%	–
0.40	SRPT, -12.73%	FS, -55.02%	FS, -26.74%	FF, 5.525%	FS+SRPT, 0.1002%
0.50	SRPT, -25.03%	SRPT, -69.41%	FS, -43.45%	FF, 10.05%	FS+SRPT, 0.2008%
0.60	FS, -47.42%	SRPT, -77.59%	FS, -54.24%	FF, 17.3%	FS+SRPT, 0.402%
0.70	FS, -61.53%	SRPT, -82.93%	FS, -30.62%	FF, 25.03%	FS+SRPT, 1.113%
0.79	FS, -70.3%	SRPT, -85.75%	FS, -18.76%	FF, 33.28%	FS+SRPT, 2.149%
0.90	SRPT, -73.83%	SRPT, -86.22%	FF, -8.617%	FF, 37.2%	FS+SRPT, 3.316%
0.10	SRPT, -4.328%	SRPT, -44.2%	FS, -23.36%	–	–
0.20	SRPT, -18.98%	SRPT, -78.65%	FF, -33.87%	FF, 7.214%	FS+Rand+SRPT, 0.1002%
0.30	SRPT, -26.46%	SRPT, -80.41%	Rand, -9.494%	FF, 5.855%	FS+Rand+SRPT, 0.2006%
0.40	SRPT, -10.98%	SRPT, -43.66%	FF, -19.9%	FF, 7.365%	SRPT, 0.1002%
0.50	SRPT, -17.24%	SRPT, -55.39%	FS, -23.86%	FF, 8.208%	FF+FS+SRPT, 0.1002%
0.60	FS, -37.96%	FS, -68.51%	FS, -32.95%	FF, 12.16%	FS+SRPT, 0.3012%
0.70	FS, -59.81%	SRPT, -79.97%	FS, -37.32%	FS, 19.26%	FS+SRPT, 0.9091%
0.80	FS, -66.85%	SRPT, -80.76%	FS, -11.75%	FF, 26.29%	FS, 2.045%
0.89	FS, -73.71%	FS, -77.53%	SRPT, -10.75%	FF, 34.9%	FS, 3.423%
0.10	SRPT, -2.553%	SRPT, -32.54%	FS, -33.37%	SRPT, 0.3021%	–
0.20	SRPT, -7.511%	FS, -57.99%	FS, -58.96%	FF, 4.589%	–
0.30	SRPT, -17.23%	SRPT, -73.84%	FS, -57.32%	FF, 19.36%	FS+SRPT, 0.1002%
0.40	SRPT, -23.55%	SRPT, -78.58%	SRPT, -13.68%	FF+FS, 16.41%	FS+SRPT, 0.2006%
0.50	SRPT, -18.2%	SRPT, -54.32%	FS, -27.63%	FS, 11.7%	FS+SRPT, 0.1002%
0.61	SRPT, -25.65%	SRPT, -54.03%	FS, -26.91%	FF, 12.48%	FS+SRPT, 0.2006%
0.70	FS, -47.05%	FS, -69.77%	FS, -20.66%	FS, 15.86%	FS+SRPT, 0.6042%
0.80	FS, -67.39%	FS, -78.64%	FS, -28.66%	FS, 23.76%	FS, 1.939%
0.90	FS, -78.26%	FS, -76.03%	FS, -13.49%	FF, 29.39%	FS, 3.638%
0.10	SRPT, -2.218%	FS, -32.07%	FS, -34.68%	–	–
0.20	FS, -6.063%	FS, -54.59%	FS, -52.78%	SRPT, 2.289%	–
0.30	FS, -12.22%	FS, -65.35%	FS, -70.36%	FS+SRPT, 1.955%	–
0.40	FS, -20.0%	FS, -72.18%	FS, -70.34%	FS, 8.26%	FS+SRPT, 0.1002%
0.51	FS, -35.12%	SRPT, -81.59%	FS, -71.39%	FS, 17.68%	FS+SRPT, 0.402%
0.60	SRPT, -49.0%	SRPT, -85.19%	FS, -40.78%	FS, 25.2%	FS+SRPT, 0.6042%
0.71	SRPT, -56.38%	SRPT, -87.45%	FS, -28.14%	FS, 26.13%	FS+SRPT, 1.011%
0.80	SRPT, -62.04%	SRPT, -85.79%	SRPT, -12.62%	FS, 23.34%	SRPT, 1.629%
0.89	FS, -72.56%	FS, -71.52%	FS, -9.325%	FF, 28.59%	FS, 3.32%

TABLE B.18: The winning schedulers' performances relative to the losing baselines for (from top to bottom) the 0 (uniform), $0.05, 0.1, 0.2$, and 0.4 skewed nodes sensitivity traces. For brevity, '–' indicates all schedulers' performances were equal.

Load	Mean FCT	p99 FCT	Max FCT	Throughput	Flows
0.10	SRPT, -2.466%	SRPT, -31.22%	FF, -38.81%	SRPT, 0.4036%	–
0.20	SRPT, -8.834%	FS, -64.48%	FS, -60.01%	FF+SRPT, 2.391%	–
0.30	SRPT, -17.83%	SRPT, -76.13%	FS, -64.78%	FF, 13.77%	FS+SRPT, 0.1001%
0.40	SRPT, -26.47%	SRPT, -81.81%	SRPT, -25.11%	FF, 15.07%	SRPT, 0.2004%
0.50	SRPT, -18.77%	SRPT, -54.54%	FS, -22.44%	FF, 11.12%	SRPT, 0.2004%
0.60	SRPT, -29.81%	SRPT, -63.41%	FS, -21.25%	FF, 13.91%	SRPT, 0.3009%
0.70	FS, -45.77%	FS, -77.26%	FS, -34.73%	FS, 21.06%	FS+SRPT, 0.6042%
0.79	FS, -55.34%	FS, -75.18%	FS, -25.43%	FS, 24.9%	FS+SRPT, 1.113%
0.89	FS, -64.75%	FS, -71.49%	FS, -26.74%	FS, 26.16%	FS, 1.941%
0.10	SRPT, -3.577%	FS, -37.74%	FS, -43.91%	FF, 0.4024%	–
0.20	SRPT, -11.42%	FS, -69.85%	FS, -60.28%	SRPT, 4.017%	FS+Rand+SRPT, 0.1001%
0.30	SRPT, -25.05%	FS, -84.23%	FS, -57.56%	FS, 9.121%	FS+SRPT, 0.1001%
0.40	SRPT, -40.72%	SRPT, -90.67%	FF, -26.91%	FS, 12.26%	FS+SRPT, 0.3009%
0.50	SRPT, -43.96%	SRPT, -91.43%	FS, -18.36%	FF, 11.84%	SRPT, 0.6036%
0.60	SRPT, -28.85%	SRPT, -72.65%	FS, -20.9%	FF, 17.01%	SRPT, 0.3009%
0.70	SRPT, -35.86%	SRPT, -77.65%	FS, -21.9%	FF, 24.46%	SRPT, 0.5025%
0.79	FS, -49.05%	SRPT, -78.85%	FS, -20.43%	FF, 32.89%	FS+SRPT, 1.011%
0.90	FS, -66.84%	FS, -77.31%	SRPT, -9.144%	FF, 38.56%	FS, 1.526%
0.10	SRPT, -4.338%	FS, -44.59%	FS, -43.56%	FF+SRPT, 0.6061%	–
0.20	FS, -12.75%	FS, -73.12%	FS, -65.3%	FS+SRPT, 2.165%	FS+Rand+SRPT, 0.1001%
0.30	FS, -32.36%	FS, -89.81%	FS, -65.67%	FS, 6.109%	FS+Rand+SRPT, 0.1001%
0.40	FS, -51.39%	FS, -93.5%	FS, -50.1%	SRPT, 11.49%	FS+SRPT, 0.6036%
0.50	FS, -64.75%	SRPT, -95.14%	FS, -44.25%	FS, 18.82%	FS+SRPT, 1.112%
0.60	SRPT, -66.34%	SRPT, -94.78%	FS, -38.79%	FS, 26.48%	FS+SRPT, 2.249%
0.70	FS, -55.14%	SRPT, -86.15%	FS, -30.44%	FS, 36.44%	SRPT, 1.112%
0.79	FS, -63.22%	SRPT, -86.61%	FS, -24.83%	FS, 50.83%	FS+SRPT, 1.939%
0.89	FS, -71.7%	SRPT, -80.91%	FS, -15.77%	FF, 57.67%	FS, 2.675%
0.10	FS, -1.484%	FS, -25.77%	SRPT, -49.67%	–	–
0.20	FS, -4.266%	FS, -24.92%	FS, -61.38%	–	–
0.30	FS, -9.855%	FS, -41.03%	FS, -66.9%	FF+FS+SRPT, 0.1001%	–
0.40	FS, -18.74%	FS, -57.24%	FS, -81.16%	FS+SRPT, 0.3012%	FS+SRPT, 0.1001%
0.50	FS, -38.63%	FS, -79.07%	FS, -84.78%	FS+SRPT, 1.013%	FS, 0.3009%
0.60	FS, -60.95%	FS, -88.96%	FS, -82.32%	FS+SRPT, 4.311%	FS, 1.011%
0.69	FS, -70.83%	FS, -88.24%	FS, -72.67%	FS, 9.434%	FS, 6.852%
0.80	FS, -64.72%	FS, -79.18%	FS, -45.34%	FS, 26.6%	FS, 16.16%
0.90	FS, -73.86%	FS, -80.41%	FS, -45.18%	FF, 55.08%	FS, 31.69%

TABLE B.19: The winning schedulers' performances relative to the losing baselines for (from top to bottom) the University, Private Enterprise, Commercial Cloud, and Social Media Cloud DCN traces. For brevity, '–' indicates all schedulers' performances were equal.

B.7 A Note on the Flow- vs. Job-Centric Traffic Paradigms

Common DCN *jobs* include search queries, generating social media feeds, and performing machine learning tasks such as inference and backpropagation. These jobs are directed acyclic graphs composed of *operations* (nodes) and *dependencies* (edges) [Paliwal et al., 2019]. The dependencies are either *control dependencies* (where the child operation can only begin once the parent operation has been completed) or *data dependencies* (where ≥ 1 tensors are output from the parent operation as required input for the child operation). In the context of DCNs, when a job arrives, each operation in the job is placed onto some machine to execute it. These operations might all be placed onto one machine or, as is often the case, distributed across different machines in the network [Shabka and Zervas, 2021]. The DCN is then used to pass the tensors around between machines executing the operations. Job data dependencies whose parent and child operations are placed onto different machines have their tensors become DCN *flows*.

There are therefore two paradigms when considering traffic demand generation in DCNs; the *flow-centric* paradigm, which is agnostic to the overall computation graph being executed in the DCN when servicing an application, and the *job-centric* paradigm, which does consider the computation graph when generating network flows. For this manuscript, we considered the flow-centric paradigm, where a single demand is a *flow*; a task demanding some information be sent from a source node to a destination node in the network. Flow characteristics include *size* (how much information to send), *arrival time* (the time the flow arrives ready to be transported through the network, as derived from the network-level *inter-arrival time* which is the time between a flow's time of arrival and its predecessor's), and *source-destination node pair* (which machine

the flow is queued at and where it is requesting to be sent). Together, these charactisterics form a network-level *source-destination node pair distribution* ('how much' (as measured by either probability or load) each machine tends to be requested by arriving flows).

In real DCNs, traffic flows can be correlated with one another since they may be part of the same job and therefore share similar characteristics. An interesting area of future work will be to develop TrafPy to support the job-centric paradigm and have this type of inter-flow correlation. However, this is beyond the scope of this manuscript.

Appendix C

Partitioning Distributed Compute Jobs

C.1 Metric Definitions

Table C.1 summarises the metric jargon used throughout the main chapter.

C.2 Experimental Hardware

All environment simulations were ran on Intel Xeon ES-2660 CPUs, and all learner network training and inference was done on either a V100 or an A100 GPU.

C.3 Additional Simulation Details

C.3.1 Code Structure

We built a core RAMP simulation environment which followed a Gym-like interface [Brockman et al., 2016] but without inheriting from a Gym environment object to allow additional flexibility. We then built a wrapper 'job partitioning' environment which did conform to the Gym interface but used our core RAMP simulation environment to perform the internal RAMP simulation logic. Our

Metric	Description
Job completion time	Time between job arriving and being completed.
Sequential job completion time	Time it would take to complete a job were its operations ran sequentially on a single device.
Maximum acceptable job completion time	Maximum time allowed to complete a job.
Speed-up factor	Factor difference between sequential job completion time and actual job completion time.
Network overhead	Fraction of the job completion time spent communicating information between workers when no computation was taking place.
Blocking rate	Fraction of the arrived jobs which were successfully serviced across a given period of time.
Job information size	Summed sizes (in bytes) of a job's operations and dependencies.
Cluster throughput	Total *partitioned job* information processed per unit time by the cluster.
Offered throughput	Total *original job* information processed per unit time by the cluster.
Load rate	Amount of job information arriving at the cluster per unit time.
Job inter-arrival time	Time between when two jobs arrived at the cluster.

Table C.1: Descriptions of the various metrics referred to throughout the main chapter.

code base is publicly available at https://github.com/cwfparsonson/ddls for further practical implementation details.

C.3.2 Job Allocation Procedure

When a job arrives at the cluster, our environment uses the following ordered sequence of task executions to allocate the job:

1. **Op. partitioning**: Partition the job DAG's operations to attain a 'partitioned' job DAG.
2. **Op. placement**: Place the operations in the partitioned job DAG onto a sub-set of cluster workers.
3. **Op. scheduling**: For each worker, schedule the priority of its placed operations to resolve conflicts where ≥ 2 operations are ready to be executed at the same time.
4. **Dep. placement**: Given the placed operations and the data dependencies which must be exchanged between operations, place the dependencies onto cluster communication links.
5. **Dep. scheduling**: For each communication link, schedule the priority of its placed dependencies to resolve conflicts where ≥ 2 dependencies are ready to be communicated at the same time.

C.3.3 Job Allocation Methods

Each of the above allocation procedure tasks can be performed by any algorithm, heuristic, or learning agent. In our work, we use the following methods:

1. **Op. partitioning**: PAC-ML, Para$_{max}$, Para$_{min}$, or Random. See the main chapter for details.

2. **Op. placement**: A first-fit heuristic customised for the requirements of RAMP. See Section C.3.4 below for details.

3. **Op. scheduling**: Shortest remaining processing time [Cai et al., 2016, Alizadeh et al., 2013, Hong et al., 2012]. Given a set of operations placed on a worker, the operation with the shortest remaining run time will have the highest priority and therefore be executed first wherever two operations on the same worker request to be executed at the same time.

4. **Dep. placement**: Shortest path & first-fit. Given a set of operation placements, for any dependencies which need to be transferred through the network (i.e. for dependencies with size > 0 and whose parent operation is placed on a separate worker from the child operation), (1) first-fit select a path from the $k-$shortest path with available light channel(s), and (2) first-fit select an available channel.

5. **Dep. scheduling**: Shortest remaining processing time. Given a set of dependencies placed on a communication link channel, the dependency with the shortest remaining processing time (i.e. the lowest amount of information left to be transferred) will have the highest priority and therefore be communicated first wherever two dependencies on the same link channel request to be transported at the same time.

C.3.4 First-Fit Operation Placement in RAMP

The original RAMP paper of Ottino et al. [2022] did not specify an operation placement heuristic which conformed to the RAMP placement rules (see Section 5.2). Here, we propose a simple first-fit heuristic which conforms to these rules whilst making the placement problem tractable for large cluster networks.

The basic idea behind partitioning and placement in the scenario described in this work is to exploit the network efficiencies of RAMP as much as possible. In

particular, this means maximising the use of RAMP's highly efficient collective operations. For a generic partitioned DAG, in the backward pass, collectives happen for each operation when weights/gradients are shared between sub-operations. If both a parent and child operation are placed on the same set of (RAMP symmetry adherent) workers, then when the parent communicates its output to the child's input in the forward pass this will also constitute a collective operation. As such the placement heuristic implemented here seeks to primarily maximise the amount that these two conditions are encountered. Given some operation, o, that has been partitioned into N equal sub-operations, o_i and needs to be placed, the placement is handled as:

1. If a parent of o has been partitioned and placed across N servers which adhere to the RAMP symmetry conditions, and if these servers each have enough memory to store o_i, then place o across this set of N servers. This ensures collective operations can happen in both the forward and backward pass.

2. Otherwise, check if a set of N workers can be found in the network that adheres to the RAMP symmetry requirements. This is achieved by sliding the various possible symmetric shapes over the topology until a suitable one (or none) is found. This ensures collective operations in the backward pass only.

Allocating in this way ensures that every partitioned operation can exploit RAMP's efficient collective operation process on the backward pass, and where possible can also exploit it on the forward pass when receiving information from (one of) its parents.

C.3.5 Evaluating the job completion time

The time to complete each operation was taken from the real computation job profiles of the DNN jobs considered (see Section C.4). To calculate the

communication time of point-to-point information transfers and of the MPI collectives, we used the equations and code of Ottino et al. [2022].

C.3.6 Possible Causes of a Job Being Blocked

A job is blocked when either JCT $> \beta \cdot \text{JCT}^{\text{seq}}$ (i.e. failing to meet user's chosen JCT requirement) or when the cluster does not have enough available resources to service the job. The possible causes of this latter form of blocking are:

- Prior jobs using up too many cluster resources when later jobs arrive;
- the minimum operation run time quantum not being low enough to partition the operations enough times to lead to the desired JCT;
- mounted worker operation scheduling conflicts for partitioned operations mounted on the same worker leading to longer run times, since one worker can only execute one operation at a time; and
- excessive communication overheads incurring from over-partitioning of the job.

C.4 Job Computation Graph Data Sets

All computation graphs used in our experiments were taken from the open-access PipeDream computation graph data set [Narayanan et al., 2019]. Fig. C.1 shows a visualisation of the key computation graph characteristics for each neural network model considered, where the numbers reported are for one training iteration (i.e. one forward and backward pass through the model). Table C.2 reports the same characteristics but in tabular form. Finally, for completeness, Fig. C.2 shows the actual job DAGs of the models used.

Model	# ops.	JCT^{seq}	Max. op. comp. time	Σ op. mem.	Max. op. mem.	Depth	# deps.	Σ dep. size	Max. dep. size
ResNet-18	142	36 668.35	473.625	17.258 66e9	0.822 121 2e9	60	159	18.733 29e9	0.822 083 6e9
VGG-16	82	34 525.35	113.330	30.625 30e9	1.644 315e9	80	83	29.467 06e9	1.644 167e9
GNMT	96	4470.80	15.88	2.368 447e9	3.269 491e8	30	117	1.027 801e9	0.194 437 1e9
SqueezeNet-10	136	38 000.15	474.637	24.962 62e9	1.168 007e9	102	153	27.910 09e9	1.167 950e9
AlexNet	46	36 061.15	635.902	3.046 234e9	0.198 339 6e9	44	47	2.422 161e9	0.198 246 4e9

TABLE C.2: Summary of the characteristics of the deep learning computation graphs used for our experiments before partitioning. The statistics shown are for the operations ('ops.') and dependencies ('deps.') which need to be executed and satisfied to conduct one training iteration. Therefore, to carry out N_{iter} training steps, the computation graph would need to be executed N_{iter} times. Computation ('comp.') time units are reported in seconds, and memory ('mem.') units in bytes.

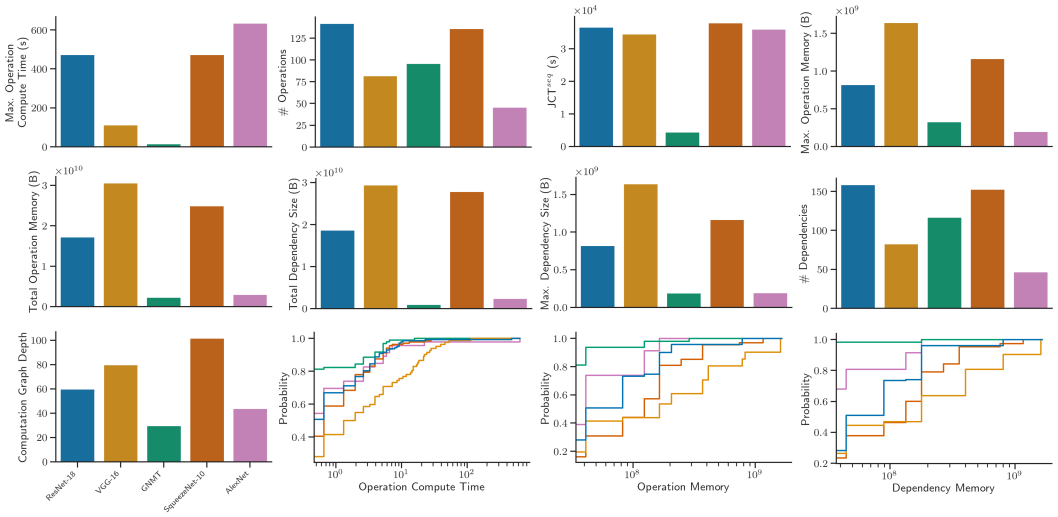


FIGURE C.1: Visualisation of the characteristics of the deep learning computation graphs used for our experiments before partitioning. The bottom left sub-figure contains the model colour code scheme for all other sub-figures. The statistics shown are for the operations and dependencies which need to be executed and satisfied to conduct one training iteration. Therefore, to carry out N_{iter} training steps, the computation graph would need to be executed N_{iter} times. Computation time units are reported in seconds, and memory units in bytes.

C.5 Neural Network Architecture

As shown in Fig. C.3, we used a message passing GNN similar to GraphSAGE with mean pooling [Hamilton et al., 2018] to parameterise the PAC-ML policy. Table C.3 summarises the hyperparameters used for the components of this DNN. We note that we did not perform extensive hyperparamter tuning on the GNN architecture. Below is a detailed explanation of this architecture.

GNN. First, the GNN layer takes in the DAG's node and edge features and generates an embedding for each node and edge in the graph. Then, each local node's nearest neighbour (1-hop away) sends the local node a message ('message passing') which is the neighbouring nodes' embeddings concatenated with their connected edges' embeddings. These messages are stored in the local node's 'mailbox', which now contains information about the node's neighbourhood. To ensure consistent dimensioning with the received messages, a dummy zero-padded edge embedding is concatenated with the local node's embedding. Next,

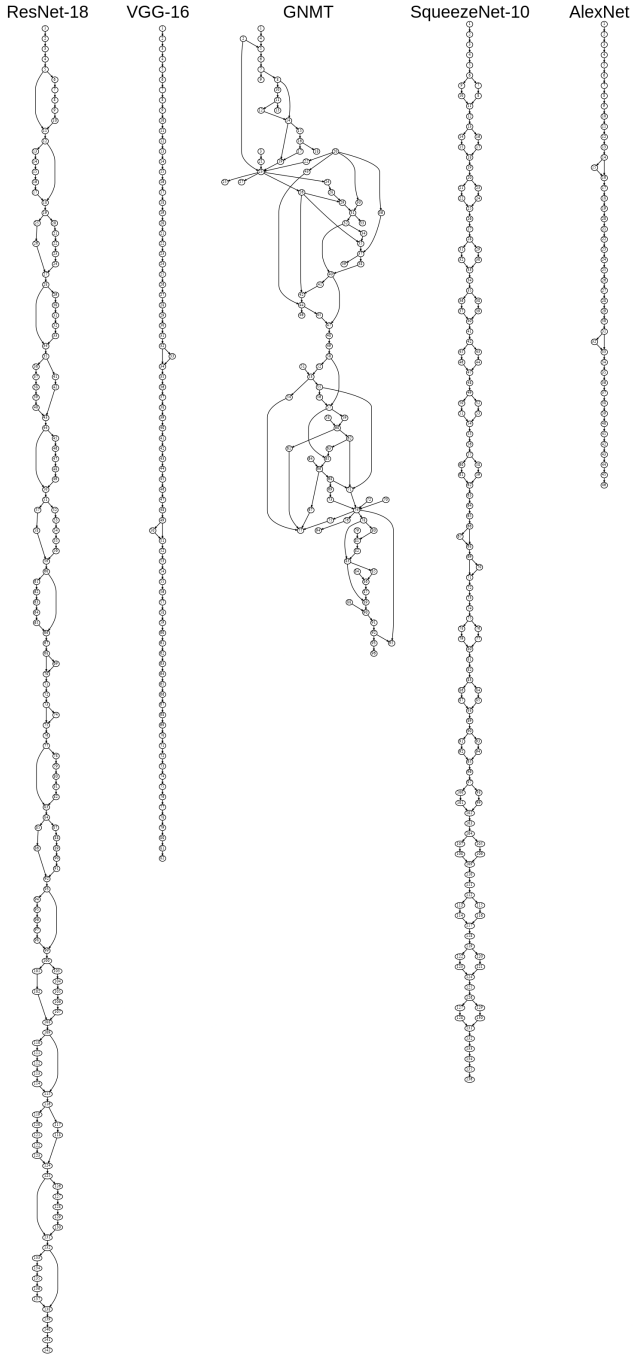


FIGURE C.2: Deep learning computation graphs used for our experiments before partitioning. Each computation graph represents the operations and dependencies which need to be executed and satisfied to conduct one forward and one backward pass through the neural network. Therefore, to carry out N_{iter} training steps, the computation graph would need to be executed N_{iter} times.

the reduce module takes the local and message embeddings and generates a reduced representation for each. Finally, to generate a layer-l output embedding for the local node, the element-wise mean of the reduced embeddings is taken

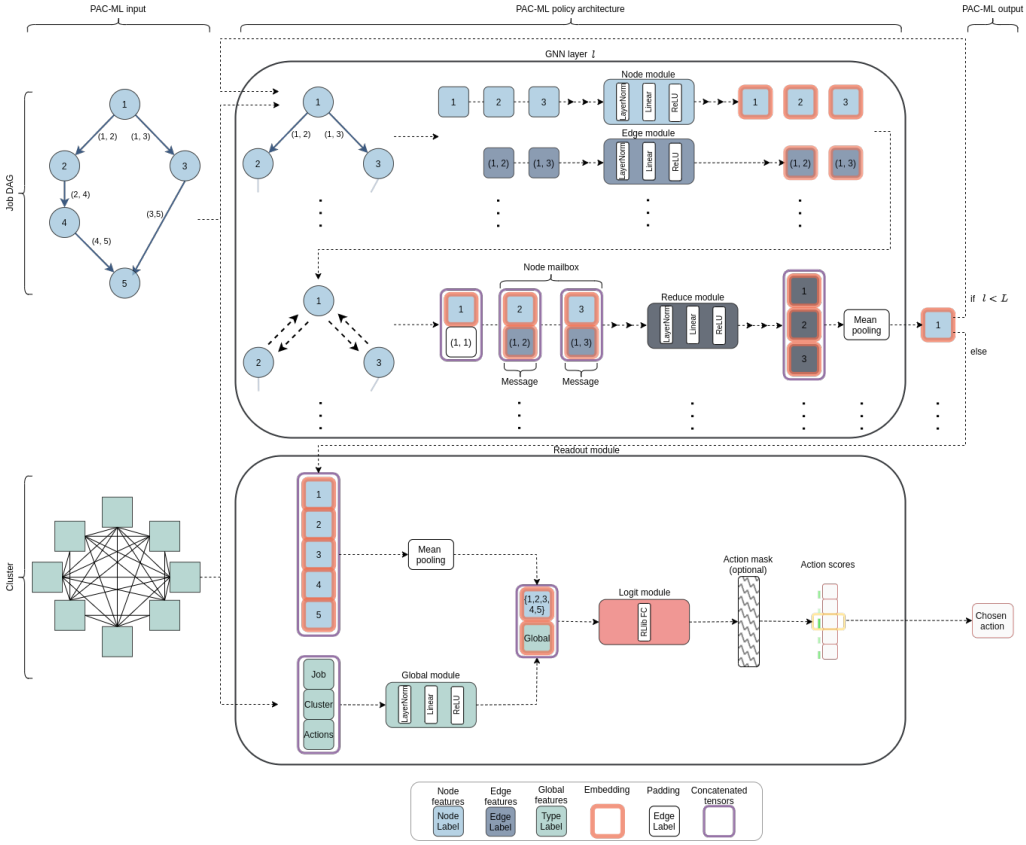


FIGURE C.3: Schematic of the DNN architecture with $|L|$ GNN layers used to parameterise the policy of PAC-ML. The GNN is similar to that of GraphSAGE with mean pooling [Hamilton et al., 2018]. Each GNN layer $l \in L$ contains a node, edge, and reduce DNN module and ultimately learns to create an embedded representation for each node in a given job DAG. These per-node embeddings are then passed, along with any global job, cluster, and action features, to a readout module. The readout module ultimately generates scores for each possible action, which enables an action to be selected following a given exploration-exploitation policy being followed. For clarity, this figure only shows the GNN embedding-generation process for node 1. See accompanying text for a detailed explanation of this architecture and the accompanying figure.

('mean pooling'). Note that this embedding process is done for each node in the DAG, but for clarity Fig. C.3 only follows node 1.

If $l < L$ (i.e. if this is not the last GNN layer), these final node embeddings are used as new features for the original DAG's nodes and are passed to the next GNN layer. If $l \equiv L$, then the node embeddings are passed to the readout module. Note that (1) the node, edge, and reduce modules are shared across the aforementioned operations within a given GNN layer when generating node embeddings, but not across different GNN layers, and (2) the l^{th}-layer's output

node embeddings will contain information about the node's neighbourhood from up to l hops away.

Readout. The readout module takes the GNN's node embeddings and the job's and cluster's global features as input. To convert the node-level embeddings of the GNN into a representation of the overall job DAG, their element-wise mean is taken. To generate an embedding capturing the global job, cluster, and action information, a global DNN module is used. The DAG and global embeddings are then concatenated and passed to a logit module, which in turn generates a vector of (optionally masked) scores for each possible action in the environment. Finally, based on these scores and the exploration-exploitation policy being followed, an action is selected.

Parameter	Value
Message passing # hidden dimensions	64
Message passing # output dimensions	32
Reduce module # hidden dimensions	64
Reduce module # output dimensions	64 if $l < L$, else 16
Global module # hidden dimensions	8
Global module # output dimensions	8
Logit module RLlib FC net # layers	1
Logit module RLlib FC net # hidden dimensions	256
All modules' activation	ReLU
GNN # layers L	2
Apply action mask	False

Table C.3: Hyperparamters used for the PAC-ML ApeX-DQN DNN policy architecture shown in Fig. C.3. Note that the 'message passing' dimensions refer to the dimensions of the concatenated node and edge modules' embeddings, so the dimensions of these modules' hidden and output embeddings will be half the corresponding 'message passing' dimension. Due to the RLlib implementation of Ape-X DQN, we did not apply an action mask, but instead included the action mask in the global features given to the model and used the reward signal to train the agent to avoid selecting invalid actions.

C.6 Reinforcement Learning Algorithm

Approach. Given the stochastic nature of our dynamic cluster environment setting, we hypothesised that a value-based RL method would be best suited

to our setting [Mao et al., 2019b]. We did try the PPO [Schulman et al., 2017] actor-critic method but found performance to be worse, although we leave a full analysis of alternative RL algorithms to future work.

As stated in the main chapter, we used the state-of-the-art value-based Ape-X DQN RL algorithm [Horgan et al., 2018] to attain the PAC-ML policy. Concretely, we used the Ape-X parallelisation approach with double Q-learning action selection-evaluation [van Hasselt et al., 2015] and multi-step bootstrapped learning targets [Sutton and Barto, 2018, Hessel et al., 2017], prioritised experience replay [Schaul et al., 2016], a dueling DQN network architecture [Wang et al., 2015], and a per-actor ϵ-greedy exploration algorithm. For a breakdown of each of these components, refer to Appendix 2.12.

Hyperparameters. To select the algorithm hyperparameters, we conducted a Bayesian search across the search space summarised in Table C.4, with simulations conducted in a light 32-worker RAMP environment with a maximum simulation run time of 2e5 seconds to speed up the search. We adopted similar search ranges to those used by Kurach et al. [2019], Hoffman et al. [2020], Parsonson et al. [2022]. For each set of hyperparameters, we ran the algorithm for 100 learner steps (a.k.a. training epochs), and performed a validation across 3 seeds at each learner step (see Fig. C.4). We selected the parameter set with the highest episode return across the 3 seeds (see Table C.4). We also report the importance of each parameter with respect to the total episode return. The importance is calculated by training a random forest with all algorithm hyperparameters as inputs and the episode return as the target output, with the per-feature (hyperparameter) importance values predicted by random forest reported accordingly [Fabros, 2018, Howard, 2018]. All our experiments used the same per-actor ϵ-greedy exploration as Horgan et al. [2018].

We note that our RL algorithms were implemented using the open-source RLlib library [Liang et al., 2018] and hyperparameter tuning was done using Weights & Biases [Biewald, 2020].

Parameter	Search Range	Best Value	Importance
Discount factor γ	$\{0.99, 0.993, 0.997, 0.999, 0.9999\}$	0.999	0.004
Learning rate	Log-uniform values (1e−7, 1e−3)	4.121e−7	0.045
v_{min}	$\{-1, -10, -100, -200, -1000\}$	−1000	0.01
v_{max}	$\{1, 10, 100, 200, 1000\}$	1000	0.004
Target network update frequency	{ 1e3, 1e4, 1e5 }	1e5	0.001
Prioritised replay α	$\{0.1, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$	0.9	0.04
Prioritised replay β	$\{0.1, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$	0.1	0.047
n-step	$\{1, 3, 5, 10\}$	3	0.227
# CPU workers	32	32	–
# GPU workers	1	1	–
Batch mode	Truncated episodes	Truncated episodes	–
Rollout length	50	50	–
Train batch size	512	512	–
Optimiser	Adam	Adam	–
Dueling	True	True	–
# atoms	1	1	–
Noisy	False	False	–
Double Q	True	True	–
Replay buffer capacity	100 000	100 000	–
Learning starts	10 000	10 000	–
Prioritised replay TD-error ϵ	1e−6	1e−6	–

TABLE C.4: Ape-X DQN training parameter sweep search range, best value found, and corresponding parameter importance.

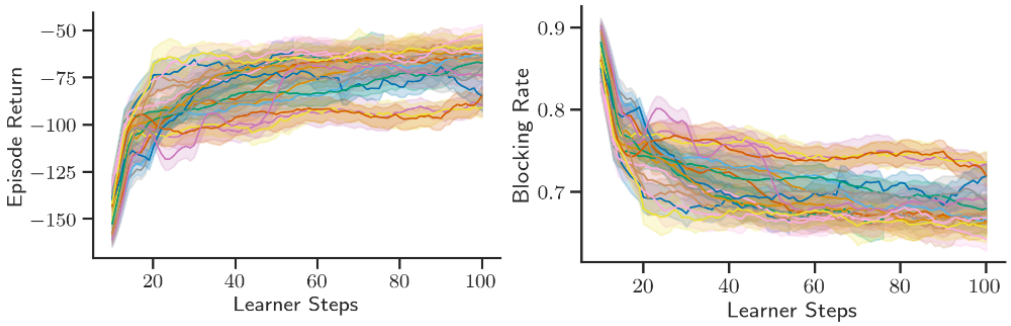


FIGURE C.4: Validation performance of the Ape-X DQN hyperparameter sweep. Each agent was trained for 100 learner steps, and at each learner step a validation was performed across 3 seeds - the mean metrics with their min-max interval bands are plotted for each hyperparameter set.

C.6.1 Final Learning Curves

For completeness, Fig. C.5 shows the learning curves of the tuned PAC-ML agents in each β_X environment superimposed on the baseline agents' performances. At each learner step, the PAC-ML agent was evaluated across three seeds in the validation environment.

C.7 Additional Experimental Results

Fig. C.6 shows the performance of the agents in terms of raw blocking rate, throughput, JCT, and JCT speed-up.

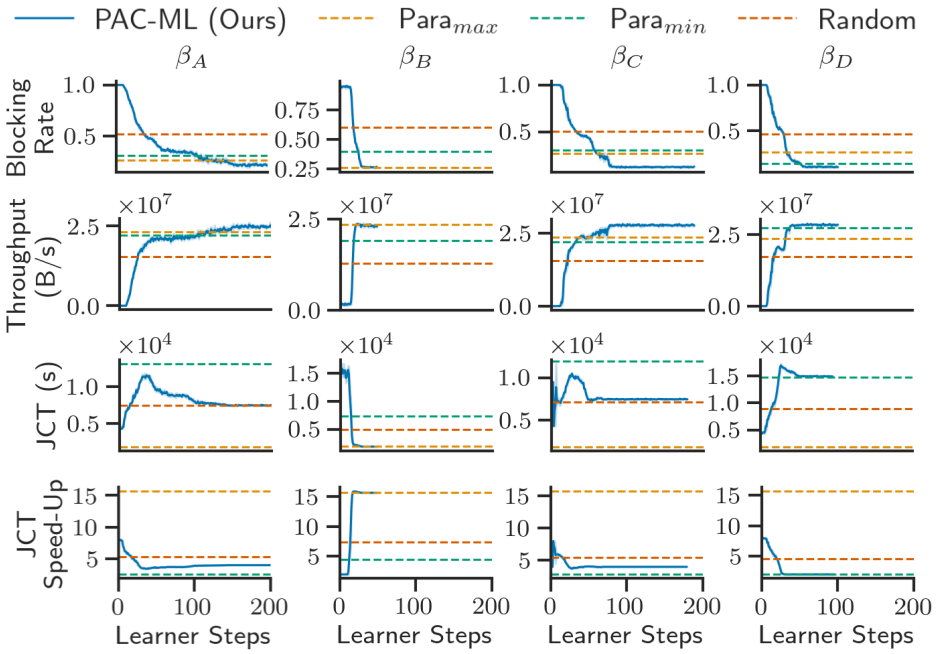


FIGURE C.5: Validation curves of the PAC-ML agent trained in four different β distribution environments. At each learner step (update to the GNN), the agent was evaluated across 3 seeds, with the mean blocking rate, offered throughput, JCT, and JCT speed-up (relative to the jobs' sequential run time JCT^{seq}) performance metrics reported as well as their min-max confidence intervals. For reference, the performances of the baseline heuristic partitioners are also plotted.

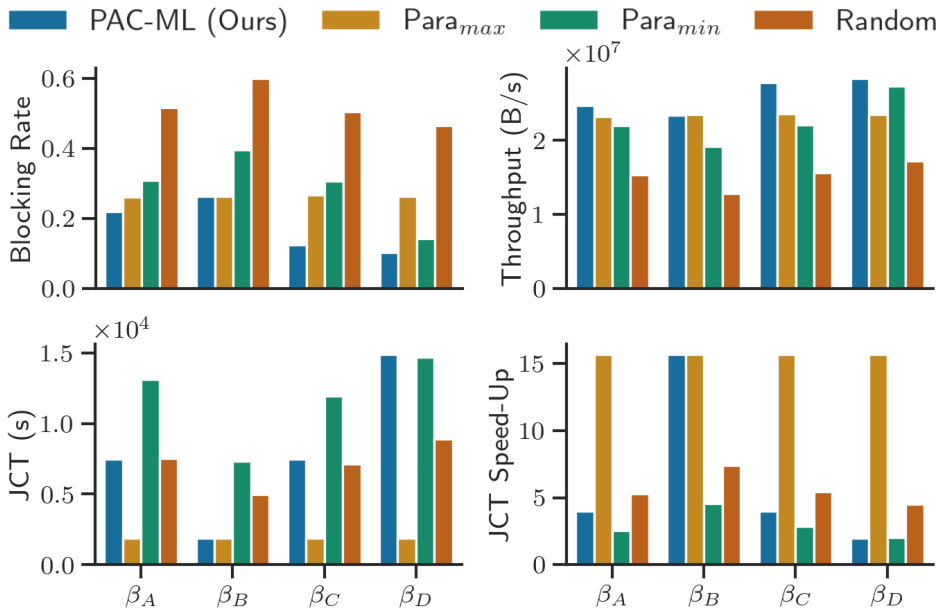


FIGURE C.6: Validation performances of each partitioning agent evaluated across three seeds, with the mean blocking rate, offered throughput, JCT, and JCT speed-up (relative to the jobs' sequential run time JCT^{seq}) performance metrics reported.

Bibliography

T Achterberg and R Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*. Facets of Combinatorial Optimization, 2013. doi: 10.1007/978-3-642-38189-818.

Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. Technical Report 04-13, ZIB, Takustr. 7, 14195 Berlin, 2004.

Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. *Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning*. Curran Associates Inc., Red Hook, NY, USA, 2019.

M. Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey. *ACM Comput. Surv.*, jun 2022. ISSN 0360-0300. doi: 10.1145/3543846. URL https://doi.org/10.1145/3543846. Just Accepted.

F. Agostinelli, S. McAleer, and A. Shmakov. Solving the Rubik's cube with deep reinforcement learning and search. *Nature*, 2019a.

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, Aug 2019b. ISSN 2522-5839. doi: 10.1038/s42256-019-0070-z. URL https://doi.org/10.1038/s42256-019-0070-z.

Gowind P. Agrawal. *Fiber-Optic Communication Systems*. John Wiley & Sons, 3rd edition, 2002.

Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 19, USA, 2010. USENIX Association.

Aws Albarghouthi. Introduction to neural network verification. *arXiv*, 2021.

Alibaba. Alibaba Cluster Trace. Technical report, 2017. URL https://github.com/alibaba/clusterdata.

Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for Ultra-Low latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, April 2012. USENIX Association. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/alizadeh.

Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, August 2013. ISSN 0146-4833. doi: 10.1145/2534169.2486031. URL https://doi.org/10.1145/2534169.2486031.

M Alizadeh et al. pfabric: Minimal near-optimal datacenter transport. *SIGCOMM*, 2013.

Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS J. Comput.*, 29:185–195, 2017.

George Amvrosiadis. The Atlas Cluster Trace Repository. *USENIX*, 43(4), 2018.

George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/amvrosiadis.

Paris Andreades, Kari Clark, Philip M. Watts, and Georgios Zervas. Experimental demonstration of an ultra-low latency control plane for optical packet switching in data center networks. *Optical Switching and Networking*, 32:51–60, 2019. ISSN 1573-4277. doi: https://doi.org/10.1016/j.osn.2018.11.005. URL https://www.sciencedirect.com/science/article/pii/S1573427718301577.

D. L. Applegate, R. E. Bixpy, V. Chvatal, and W. J. Cook. Finding cuts in the TSP (A preliminary report). Technical report, DIMACS, 1995.

D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

A Assadihaghi, H Teimoori, and T J Hall. SOA-Based Optical Switches. *Optical Switches - Materials and Design*, pages 158–180, 2010.

Azure. Azure Public Dataset. Technical report, 2017. URL https://github.com/Azure/AzurePublicDataset.

Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ecn in multi-service multi-queue data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 537–549, Santa Clara, CA, March 2016. USENIX Association. ISBN 978-1-931971-29-4. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/bai.

Wei Bai et al. Enabling ecn in multi-service multi-queue data centers. *NSDI*, 2016.

Egon Balas, Andrew Ho, and Carnegie Mellon UniversityDesign Research Center. Set covering algorithms using cutting planes, heuristics, and subgradient optimization : a computational study, Jun 2018. URL https://kilthub.cmu.edu/articles/journal_contribution/Set_covering_algorithms_using_cutting_planes_heuristics_and_subgradient_optimization_a_computational_study/6707945/1.

Hitesh Ballani, Paolo Costa, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, and Hugh Williams. Bridging the last mile for optical switching in data centers. In *Optical Fiber Communication Conference*, page W1C.3. Optica Publishing Group, 2018. doi: 10.1364/OFC.2018.W1C.3. URL https://opg.optica.org/abstract.cfm?URI=OFC-2018-W1C.3.

Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Benn Thomsen, Kai Shi, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *SIGCOMM*, SIGCOMM '20, page 782–797, New York, NY, USA, August 2020. ACM, Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3406221. URL https://www.microsoft.com/en-us/research/publication/sirius-a-flat-datacenter-network-with-nanosecond-optical-switching/.

Jagdish Chand Bansal. *Evolutionary and Swarm Intelligence Algorithms*, volume 779. 2019. ISBN 978-3-319-91339-1. URL http://link.springer.com/10.1007/978-3-319-91341-4.

Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online job scheduling in distributed machine learning clusters. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, page 495–503. IEEE Press, 2018.

doi: 10.1109/INFOCOM.2018.8486422. URL https://doi.org/10.1109/INFOCOM.2018.8486422.

Francisco Barahona. On the computational complexity of Ising spin glass models. *Journal of Physics A: Mathematical and General*, 15(10):3241, 1982.

Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Oper. Res.*, 46(3):316–329, mar 1998. ISSN 0030-364X. doi: 10.1287/opre.46.3.316. URL https://doi.org/10.1287/opre.46.3.316.

Thomas D Barrett, William R Clements, Jakob N Foerster, and AI Lvovsky. Exploratory combinatorial optimization with reinforcement learning. *Association for the Advancement of Artificial Intelligence*, 2019.

Thomas D. Barrett, Christopher W. F. Parsonson, and Alexandre Laterre. Learning to solve combinatorial graph partitioning problems via efficient exploration. *arXiv preprint arXiv:2205.14105*, 2022. doi: 10.48550/ARXIV.2205.14105. URL https://arxiv.org/abs/2205.14105.

Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13:341–379, 2003.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL http://arxiv.org/abs/1611.09940.

Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), aug 2019. ISSN 0360-0300. doi: 10.1145/3320060. URL https://doi.org/10.1145/3320060.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2020.07.063. URL http://www.sciencedirect.com/science/article/pii/S0377221720306895.

M Benichou, J M Gauthier, P Girodet, G Hentges, G Ribiere, and O Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971. ISSN 1436-4646. doi: 10.1007/BF01584074. URL https://doi.org/10.1007/BF01584074.

Joshua Benjamin. *Towards Sub-Microsecond Optical Circuit Switched Networks for Future Data Centres*. PhD thesis, UCL, 2020.

Joshua L Benjamin, Thomas Gerard, Domaniç Lavery, Polina Bayvel, and Georgios Zervas. PULSE: Optical Circuit Switched Data Center Architecture Operating at Nanosecond Timescales. *J. Lightwave Technol.*, 38(18):4906–4921, sep 2020. URL http://jlt.osa.org/abstract.cfm?URI=jlt-38-18-4906.

Joshua L Benjamin, Christopher W F Parsonson, and Georgios Zervas. Benchmarking Packet-Granular OCS Network Scheduling for Data Center Traffic Traces. In *OSA Advanced Photonics Congress 2021*, page NeW3B.3. Optica Publishing Group, 2021. doi: 10.1364/NETWORKS.2021.NeW3B.3. URL http://opg.optica.org/abstract.cfm?URI=Networks-2021-NeW3B.3.

Joshua L. Benjamin, Alessandro Ottino, Christopher W. F. Parsonson, and Georgios Zervas. Traffic tolerance of nanosecond scheduling on optical circuit switched data center network. In *2022 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3, 2022.

Joshua Lawrence Benjamin, Adam Funnell, Philip Michael Watts, and Benn Thomsen. A high speed hardware scheduler for 1000-port optical packet switches to enable scalable data centers. *Proceedings - 2017 IEEE 25th Annual*

Symposium on High-Performance Interconnects, HOTI 2017, pages 41–48, 2017. doi: 10.1109/HOTI.2017.22.

Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pages 267–280, 2010a. doi: 10.1145/1879141.1879175.

Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, January 2010b. ISSN 0146-4833. doi: 10.1145/1672308.1672325. URL https://doi.org/10.1145/1672308.1672325.

Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450310413. doi: 10.1145/2079296.2079304. URL https://doi.org/10.1145/2079296.2079304.

David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization*. Springer Publishing Company, Incorporated, 1st edition, 2016. ISBN 3319428470.

Keren Bergman. Empowering Flexible and Scalable High Performance Architectures with Embedded Photonics. *IPDPS*, 2018.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

J. M. Bishop. Stochastic searching networks. *IEE Conference Publication*, (313): 329–331, 1989. ISSN 05379989.

E Bonabeau, M Dorigo, and G Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

Michael Bowling and Manuela Veloso. Rational and convergent learning in stochastic games. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'01, page 1021–1026, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558608125.

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

Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019. doi: 10.1126/science.aay2400. URL https://www.science.org/doi/abs/10.1126/science.aay2400.

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. 2018.

C. X. Cai, S. Saeed, I. Gupta, R. H. Campbell, and F. Le. Phurti: Application and network-aware flow scheduling for multi-tenant mapreduce clusters. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 161–170, 2016. doi: 10.1109/IC2E.2016.21.

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. 2021.

Y. Chen. Optical Burst Switching: A New Area in Optical Networking Research. *IEEE*, pages 16–23, 2005.

Cisco. Cisco global cloud index: Forecast and methodology. Technical report, Cisco, 2016.

Kari Clark, Hitesh Ballani, Polina Bayvel, Daniel Cletheroe, Thomas Gerard, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, Philip Watts, Hugh Williams, Georgios Zervas, Paolo Costa, and Zhixin Liu. Sub-nanosecond clock and data recovery in an optically-switched data centre network. In *2018 European Conference on Optical Communication (ECOC)*, pages 1–3, 2018. doi: 10.1109/ECOC.2018.8535333.

Maurice Clerc. The swarm and the queen: Towards a deterministic and adaptive particle swarm optimization. *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999*, 3:1951–1957, 1999. doi: 10.1109/CEC.1999.785513.

Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1282–1289. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/cobbe19a.html.

E. Conforti and C. M. Gallep. A fast electro-optical amplified switch using a resistive combiner for multi-pulse injection. In *2006 IEEE MTT-S International Microwave Symposium Digest*, pages 1935–1938, 2006. doi: 10.1109/MWSYM.2006.249812.

Michael Connelly. Semiconductor Optical Amplifiers and their Applications. *Applications of Photonic Technology 5*, 4833(August 2003):974, 2003. doi: 10.1117/12.478235.

Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents.

In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 5032–5043, Red Hook, NY, USA, 2018. Curran Associates Inc.

Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132772. URL https://doi.org/10.1145/3132747.3132772.

IBM CPLEX. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

Paul A. Crook and Gillian Hayes. Learning in a state of confusion: Perceptual aliasing in grid world navigation. In *IN TOWARDS INTELLIGENT MOBILE ROBOTS 2003 (TIMR 2003), 4 TH BRITISH CONFERENCE ON (MOBILE) ROBOTICS*, 2003.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6351–6361, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

James Dale Davidson and William Rees-Mogg. *The Sovereign Individual; Mastering the Transition to the Information Age*. Simon & Schuster, Inc., USA, 1999. ISBN 0684832720.

K. De Jong, D. Fogel, and H.-P Schwefel. *Handbook of Evolutionary Computation*. CRC Press, 1997.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de las Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, Seb Noury, Federico Pesamosca, David Pfau, Olivier Sauter, Cristian Sommariva, Stefano Coda, Basil Duval, Ambrogio Fasoli, Pushmeet Kohli, Koray Kavukcuoglu, Demis Hassabis, and Martin Riedmiller. Magnetic control of tokamak plasmas through deep reinforcement learning. *NatureTY - JOUR*, 602 (7897):414–419, Feb 2022. ISSN 1476-4687. doi: 10.1038/s41586-021-04301-9. URL https://doi.org/10.1038/s41586-021-04301-9.

Delft. GWA-T-12 Bitbrains Trace. Technical report, 2015. URL http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains.

J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Kefan Dong, Yuping Luo, Tianhe Yu, Chelsea Finn, and Tengyu Ma. On the expressivity of neural networks for deep reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

M Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnicodi Milano, 1992.

Marco Dorigo and Thomas Stützle. *Ant colony optimization*. The MIT Press, 2004. ISBN 9781439802847. doi: 10.4249/scholarpedia.1461.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, Feb 2021. ISSN 1476-4687. doi: 10.1038/s41586-020-03157-9. URL http://dx.doi.org/10.1038/s41586-020-03157-9.

Marc Etheve, Zacharie Alès, Côme Bissuel, Olivier Juan, and Safia Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. *Lecture Notes in Computer Science*, page 176–185, 2020. ISSN 1611-3349. doi: 10.1007/978-3-030-58942-4_12. URL http://dx.doi.org/10.1007/978-3-030-58942-4_12.

Eucalyptus. Eucalyptus IaaS Cloud Workload. Technical report, 2015. URL https://sites.cs.ucsb.edu/{~}rich/workload/.

Melissa Fabros. Introduction to hyperparameters, Sept 2018. URL https://forums.fast.ai/t/wiki-lesson-thread-lesson-4/7540.

Facebook. Facebook Workload Repository. Technical report, 2014. URL https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

Shaohua Fan, Xiao Wang, Chuan Shi, Peng Cui, and Bai Wang. Generalizing graph neural networks on out-of-distribution graphs, 2021. URL https://arxiv.org/abs/2111.10657.

Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 40 (4):339–350, August 2010. ISSN 0146-4833. doi: 10.1145/1851275.1851223. URL https://doi.org/10.1145/1851275.1851223.

D. G. Feitelson. Metric and workload effects on computer systems evaluation. *Computer*, 36(9):18–25, 2003. doi: 10.1109/MC.2003.1231190.

R. C. Figueiredo, T. Sutili, N. S. Ribeiro, C. M. Gallep, and E. Conforti. Semiconductor optical amplifier space switch with symmetrical thin-film resistive current injection. *Journal of Lightwave Technology*, 35(2):280–287, 2017. doi: 10.1109/JLT.2016.2635202.

Rafael C. Figueiredo, Eduardo C. Magalhães, Napoleao S. Ribeiro, Cristiano M. Gallep, and Evandro Conforti. Equivalent circuit of a semiconductor optical amplifier chip with the bias current influence. *SBMO/IEEE MTT-S International Microwave and Optoelectronics Conference Proceedings*, pages 852–856, 2011. doi: 10.1109/IMOC.2011.6169263.

Rafael C. Figueiredo, Napoleao S. Ribeiro, Antonio Marcelo Oliveira Ribeiro, Cristiano M. Gallep, and Evandro Conforti. Hundred-Picoseconds Electro-Optical Switching With Semiconductor Optical Amplifiers Using Multi-Impulse Step Injection Current. *Journal of Lightwave Technology*, 33(1): 69–77, jan 2015. ISSN 0733-8724. doi: 10.1109/JLT.2014.2372893. URL http://ieeexplore.ieee.org/document/6963258/.

J Foerster. *Deep multi-agent reinforcement learning*. PhD thesis, University of Oxford, 2018.

A Fraser. Simulation of Genetic Systems by Automatic Digital Computers. *Australian Journal of Biological Sciences*, (2):87–94, 1958. doi: 10.1109/9780470544600.ch3.

Steve Furber. Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, 13(5):051001, aug 2016. doi: 10.1088/1741-2560/13/5/051001. URL https://doi.org/10.1088/1741-2560/13/5/051001.

C.M. Gallep and E. Conforti. Reduction of semiconductor optical amplifier switching times by preimpulse step-injected current technique. *IEEE Photonics Technology Letters*, 14(7):902–904, jul 2002. ISSN 1041-1135. doi: 10.1109/LPT.2002.1012379. URL http://ieeexplore.ieee.org/document/1012379/.

Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1676–1684. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/gao18a.html.

Vikas Garg, Stefanie Jegelka, and Tommi Jaakkola. Generalization and representational limits of graph neural networks. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3419–3430. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/garg20c.html.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. *Exact Combinatorial Optimization with Graph Convolutional Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Thomas Gerard, Christopher Parsonson, Zacharaya Shabka, Polina Bayvel, Domaniç Lavery, and Georgios Zervas. Swift: Scalable ultra-wideband sub-nanosecond wavelength switching for data centre networks. 2020a. doi: 10.48550/ARXIV.2003.05489. URL https://arxiv.org/abs/2003.05489.

Thomas Gerard, Christopher Parsonson, Zacharaya Shabka, Polina Bayvel, Domaniç Lavery, and Georgios Zervas. Swift: Scalable ultra-wideband sub-nanosecond wavelength switching for data centre networks, 2020b.

Thomas Gerard, Christopher Parsonson, Zacharaya Shabka, Benn Thomsen, Polina Bayvel, Domaniç Lavery, and Georgios Zervas. AI-optimised tuneable sources for bandwidth-scalable, sub-nanosecond wavelength switching. *Opt. Express*, 29(7):11221–11242, mar 2021. doi: 10.1364/OE.417272. URL http://opg.optica.org/oe/abstract.cfm?URI=oe-29-7-11221.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-6226-1. URL https://doi.org/10.1007/s10994-006-6226-1.

H. Ghafouri-Shiraz. *The principles of semiconductor laser diodes and amplifiers: Analysis and transmission line laser modeling*. Imperial College Press, 2004. ISBN 186094339X. URL https://books.google.co.uk/books/about/The{_}Principles{_}of{_}Semiconductor{_}Laser{_}Di.html?id=zYvZwgiPlKUC{&}redir{_}esc=y.

Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017. ISBN 1627052984.

Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, USA, 1 edition, 2008. ISBN 052188473X.

Oded Goldreich. *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press, USA, 1st edition, 2010. ISBN 0521122546.

N. G. Gonzalez, D. Zibar, and I. T. Monroy. Cognitive digital receiver for burst mode phase modulated radio over fiber links. In *36th European Conference and Exhibition on Optical Communication*, pages 1–3, 2010. doi: 10.1109/ECOC.2010.5621525.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618.

Google. Google Cluster Workload. Technical report, 2015. URL https://github.com/google/cluster-data.

Vojtech Graf, Dusan Teichmann, Jiri Horinka, and Michal Dorda. Dynamic Model for Scheduling Crew Shifts. *Mathematical Problems in Engineering*, 2020. doi: https://doi.org/10.1155/2020/5372567.

C. Gray, R. Ayre, K. Hinton, and R. S. Tucker. Power consumption of iot access network technologies. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 2818–2823, 2015. doi: 10.1109/ICCW.2015.7247606.

A. Hanif Halim and Idris Ismail. Combinatorial optimization: Comparison of heuristic algorithms in travelling salesman problem. *Archives of Computational Methods in Engineering*, 26:367–380, 2019.

Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 319–330, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328364.

doi: 10.1145/2619239.2626328. URL https://doi.org/10.1145/2619239.2626328.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv*, 2018.

Anna Harutyunyan, Will Dabney, Thomas Mesnard, Nicolas Heess, Mohammad G. Azar, Bilal Piot, Hado van Hasselt, Satinder Singh, Greg Wayne, Doina Precup, and Rémi Munos. *Hindsight Credit Assignment*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Conor F. Hayes, Roxana Radulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M. Zintgraf, Richard Dazeley, Fredrik Heintz, Enda Howley, Athirai A. Irissappane, Patrick Mannion, Ann Nowé, Gabriel de Oliveira Ramos, Marcello Restelli, Peter Vamplew, and Diederik M. Roijers. A practical guide to multi-objective reinforcement learning and planning. *Auton. Agents Multi Agent Syst.*, 36(1):26, 2022. doi: 10.1007/s10458-022-09552-y. URL https://doi.org/10.1007/s10458-022-09552-y.

He He, Hal Daumé, and Jason Eisner. Learning to search in branch-and-bound algorithms. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3293–3301, Cambridge, MA, USA, 2014. MIT Press.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055.

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. 2017.

R. Hinterding. Gaussian mutation and self-adaption for numeric genetic algorithms. In *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, volume 1, pages 384–, 1995. doi: 10.1109/ICEC.1995.489178.

Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stanczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, Leonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gomez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. 2020. doi: 10.48550/ARXIV.2006.00979. URL https://arxiv.org/abs/2006.00979.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. 2022. doi: 10.48550/ARXIV.2203.15556. URL https://arxiv.org/abs/2203.15556.

Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols*

for Computer Communication, SIGCOMM '12, page 127–138, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314190. doi: 10.1145/2342356.2342389. URL https://doi.org/10.1145/2342356.2342389.

J. J. Hopfield and D. W. Tank. "neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, July 1985. ISSN 0340-1200. doi: 10.1007/BF00339943.

Jeff Horen. Linear programming, by katta g. murty, john wiley & sons, new york, 1983, 482 pp. *Networks*, 15(2):273–274, 1985. URL http://dblp.uni-trier.de/db/journals/networks/networks15.html#Horen85.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1Dy---0Z.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

Jeremy Howard. Intro to machine learning: Lesson 4, Sept 2018. URL https://www.youtube.com/watch?v=0v93qHDqq_g.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv preprint arXiv:1602.07360*, 2016. URL http://arxiv.org/abs/1602.07360. cite arxiv:1602.07360Comment: In ICLR Format.

S. Jha, A. Patke, J. Brandt, A. Gentile, M. Showerman, E. Roman, Z. T. Kalbarczyk, B. Kramer, and R. K. Iyer. A study of network congestion in two supercomputing high-speed interconnects. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 45–48, 2019. doi: 10.1109/HOTI.2019.00024.

Saurabh Jha, Archit Patke, Jim Brandt, Ann Gentile, Benjamin Lim, Mike Showerman, Greg Bauer, Larry Kaplan, Zbigniew Kalbarczyk, William Kramer, and Ravi Iyer. Measuring congestion in high-performance datacenter interconnects. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 37–57, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/jha.

T Jiménez, J C Aguado, I de Miguel, R J Durán, N Fernández, M Angelou, D Sánchez, N Merayo, P Fernández, N Atallah, R M Lorenzo, I Tomkos, and E J Abril. A Cognitive System for Fast Quality of Transmission Estimation in Core Optical Networks. In *Optical Fiber Communication Conference*, page OW3A.5. Optical Society of America, 2012. doi: 10.1364/OFC.2012.OW3A.5. URL http://www.osapublishing.org/abstract.cfm?URI=OFC-2012-OW3A.5.

David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

Donald B. Johnson. A note on dijkstra's shortest path algorithm. *J. ACM*, 20 (3):385–388, July 1973. ISSN 0004-5411. doi: 10.1145/321765.321768. URL https://doi.org/10.1145/321765.321768.

JSSPP. JSSPP Workloads Archive. Technical report, 2017. URL https://jsspp.org/workload/.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, Aug 2021. ISSN 1476-4687. doi: 10.1038/s41586-021-03819-2. URL https://doi.org/10.1038/s41586-021-03819-2.

Sham Kakade. A Natural Policy Gradient. In *Adv. Neural Inf. Process Syst.*, volume 14, pages 1531–1538, 2001.

Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements &; analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, page 202–208, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587714. doi: 10.1145/1644893.1644918. URL https://doi.org/10.1145/1644893.1644918.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei.

Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Can Karakus, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Cavdar, Teng Xu, Haohan Chen, Arash Rahnama, and Luis Quintela. Amazon sagemaker model parallelism: A general and flexible framework for large model training. 2021. doi: 10.48550/ARXIV.2111.05972. URL https://arxiv.org/abs/2111.05972.

James Kennedy, Russell Eberhart, and Bls Gov. Particle Swarm Optimization. *International Conference on Neural Networks*, 11(1):111–117, 1995. ISSN 1598-8619. URL http://ci.nii.ac.jp/naid/10015518367.

Shauharda Khadka, Estelle Aflalo, Mattias Mardar, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. Optimizing memory placement using evolutionary graph reinforcement learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=-6vS_4Kfz0.

Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra N. Dilkina. Learning to branch in mixed integer programming. In *AAAI*, 2016.

Mehrdad Khani et al. Sip-ml: High-bandwidth optical network interconnects for machine learning training. *SIGCOMM*, 2021. doi: 10.1145/3452296.3472900.

Paresh Kharya and Ali Alvi. Using deepspeed and megatron to train megatron-turing nlg 530b, the world's largest and most powerful generative language model. https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-lar Oct 2021.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2022. doi: 10.1109/TITS.2021.3054625.

Serkan Kiranyaz. *Adaptation, Learning, and Optimization*, volume 15. Springer, 2014. doi: 10.1007/978-3-642-37846-1_3.

Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning, 2021. URL https://arxiv.org/abs/2111.09794.

Dalibor Klusáček and Boris Parák. Analysis of Mixed Workloads from Shared Cloud Infrastructure. In Dalibor Klusáček, Walfredo Cirne, and Narayan Desai, editors, *Job Scheduling Strategies for Parallel Processing*, pages 25–42, Cham, 2017. Springer International Publishing. ISBN 978-3-319-77398-8.

Boris Knyazev, Graham W. Taylor, and Mohamed R. Amer. *Understanding Attention and Generalization in Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA, 2019.

B. H. Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, New York, NY, 2012. ISBN 9783642244889 3642244882 3642244874 9783642244872. doi: 10.1007/978-3-642-24488-9.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira,

C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Karol Kurach, Anton Raichuk, Piotr Stanczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. Google research football: A novel reinforcement learning environment. 2019. doi: 10.48550/ARXIV.1907.11180. URL https://arxiv.org/abs/1907.11180.

D. H. Kusuma, M. Ali, and N. Sutantra. The comparison of optimization for active steering control on vehicle using pid controller based on artificial intelligence techniques. In *2016 International Seminar on Application for Technology of Information and Communication (ISemantic)*, pages 18–22, Aug 2016. doi: 10.1109/ISEMANTIC.2016.7873803.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):pp. 497–520, 1960. ISSN 00129682.

LANL and TwoSigma. ATLAS Traces Repository. Technical report, 2018. URL https://ftp.pdl.cmu.edu/pub/datasets/ATLAS/.

Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992. ISSN 0377-2217. doi: https://doi.org/10.1016/0377-2217(92)90138-Y. URL http://www.sciencedirect.com/science/article/pii/037722179290138Y.

Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, EC '00, page 66–76, New York,

NY, USA, 2000. Association for Computing Machinery. ISBN 1581132727. doi: 10.1145/352871.352879. URL https://doi.org/10.1145/352871.352879.

H. Li. Realistic workload modeling and its performance impacts in large-scale escience grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(4): 480–493, 2010. doi: 10.1109/TPDS.2009.99.

Qingping Li, Jingwei Xu, and Chun Cao. Scheduling distributed deep learning jobs in heterogeneous cluster with placement awareness. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, Internetware '20, page 217–228, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388191. doi: 10.1145/3457913.3457936. URL https://doi.org/10.1145/3457913.3457936.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3053–3062. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/liang18b.html.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. 2019.

J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991. doi: 10.1109/18.61115.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3–4):293–321, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992699. URL https://doi.org/10.1007/BF00992699.

Igor Litvinchev and Edith Lucero Ozuna Espinosa. Solving the two-stage capacitated facility location problem by the lagrangian heuristic. In Hao Hu, Xiaoning Shi, Robert Stahlbock, and Stefan Voß, editors, *Computational Logistics*, pages 92–103, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33587-7.

Shiyun Liu, Qixiang Cheng, Muhammad Ridwan Madarbux, Adrian Wonfor, Richard V. Penty, Ian H. White, and Philip M. Watts. Low latency optical switch for high performance computing with minimized processor energy load [invited]. *Journal of Optical Communications and Networking*, 7(3): A498–A510, 2015. doi: 10.1364/JOCN.7.00A498.

Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017. ISSN 1863-8279. doi: 10.1007/s11750-017-0451-6. URL https://doi.org/10.1007/s11750-017-0451-6.

C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017. doi: 10.1109/BigData.2017.8258257.

Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pages 301–307. Springer International Publishing, 2016. doi: 10.1007/978-3-319-42432-3_37.

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*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450346610. doi: 10.1145/3005745.3005750. URL https://doi.org/10.1145/3005745.3005750.

Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342080. URL https://doi.org/10.1145/3341302.3342080.

Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019b. URL https://openreview.net/forum?id=Hyg1G2AqtQ.

Hongzi Mao, Computer Science, Mohammad Alizadeh, Computer Science, Thesis Supervisor, Leslie A Kolodziejski, and Computer Science. *Network System Optimization with Reinforcement Learning : Methods and Applications*. PhD thesis, MIT, 2020.

Javier Mata, Ignacio de Miguel, Ramón J. Durán, Noemí Merayo, Sandeep Kumar Singh, Admela Jukan, and Mohit Chamania. Artificial intelligence (AI) methods in optical networks: A comprehensive survey. *Optical Switching and Networking*, 28:43–57, 2018. ISSN 15734277. doi: 10.1016/j.osn.2017.12.006.

Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Comput. Surv.*, 53

(1), feb 2020. ISSN 0360-0300. doi: 10.1145/3363554. URL https://doi.org/10.1145/3363554.

McKinsey. Artificial-intelligence hardware: New opportunities for semiconductor companies. Technical report, McKinsey, 2019. URL https://www.mckinsey.com/industries/semiconductors/our-insights/artificial-intelligence-hardware-new-opportunities-for-semiconductor-companie

William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 267–280, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098838. URL https://doi.org/10.1145/3098822.3098838.

Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2430–2439, International Convention Centre, Sydney, Australia, 2017. PMLR. URL http://proceedings.mlr.press/v70/mirhoseini17a.html.

Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=Hkc-TeZ0W.

Vaibhawa Mishra, Joshua L. Benjamin, and Georgios Zervas. Monet: heterogeneous memory over optical network for large-scale data center resource

disaggregation. *Journal of Optical Communications and Networking*, 13(5): 126–139, 2021. doi: 10.1364/JOCN.419145.

John E Mitchell. *Integer programming: branch and cut algorithmsInteger Programming: Branch and Cut Algorithms*, pages 1643–1650. Springer US, Boston, MA, 2009. ISBN 978-0-387-74759-0. doi: 10.1007/978-0-387-74759-0_287. URL https://doi.org/10.1007/978-0-387-74759-0{_}287.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *NeurIPS'13 Workshop*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 00280836. URL http://dx.doi.org/10.1038/nature14236.

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 Maria Florina Balcan and Kilian Q Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 2016. PMLR. URL http://proceedings.mlr.press/v48/mniha16.html.

Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the Number of Linear Regions of Deep Neural Networks. In

Z Ghahramani, M Welling, C Cortes, N Lawrence, and K Q Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27 of *NIPS'14*, pages 2924–2932, Cambridge, MA, USA, 2014. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2014/file/109d2dd3608f669ca17920c511c2a41e-Paper.pdf.

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks. 2021.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Granger, Phil Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM Symposium on Operating Systems Principles (SOSP 2019)*, October 2019. URL https://www.microsoft.com/en-us/research/publication/pipedream-generalized-pipeline-parallelism-for-dnn-training/.

Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning (ICML 2021)*, July 2021. URL https://www.microsoft.com/en-us/research/publication/memory-efficient-pipeline-parallel-dnn-training/.

NCSA. Blue Waters HPC Cluster Trace. Technical report, 2018. URL https://github.com/CSLDepend/monet.

John A. Nelder and Roger Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

NVIDIA. Nvidia selene: Leadership-class supercomputing infrastructure. https://www.nvidia.com/en-us/on-demand/session/supercomputing2020-sc2019/, Nov 2020.

NVIDIA. Nvidia ai platform delivers big gains for large language models. https://developer.nvidia.com/blog/nvidia-ai-platform-delivers-big-gains-for-large-language-models/, Jul 2022.

OpenAI. Ai and compute. https://openai.com/blog/ai-and-compute/, May 2018.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *arXiv*, 2019.

OpenCloud. OpenCloud Hadoop Workload. Technical report, 2012. URL http://ftp.pdl.cmu.edu/pub/datasets/hla/.

Simon Osindero. Deep Mind Lecture Series, Lecture 3, Neural Networks Foundations, 2018.

Alessandro Ottino, Joshua Benjamin, and Georgios Zervas. Ramp: A flat nanosecond optical network and mpi operations for distributed deep learning systems. *arXiv*, 2022. doi: 10.48550/ARXIV.2211.15226. URL https://arxiv.org/abs/2211.15226.

Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. *arXiv e-prints*, art. arXiv:1905.02494, May 2019.

Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020. URL https://openreview.net/forum?id=rkxDoJBYPB.

Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., USA, 1982. ISBN 0131524623.

Alberto Paradisi. *Optical Communications: Advanced Systems and Devices for Next Generation Networks*. 2019.

C. W. F. Parsonson, Z. Shabka, W. K. Chlupka, B. Goh, and G. Zervas. Optimal control of soas with artificial intelligence for sub-nanosecond optical switching. *Journal of Lightwave Technology*, 38(20):5563–5573, 2020. doi: 10.1109/JLT.2020.3004645.

Christopher Parsonson and Georgios Zervas. Trafpy, July 2021a. URL https://github.com/cwfparsonson/trafpy.

Christopher Parsonson and Georgios Zervas. Trafpy rdr data, July 2021b.

Christopher Parsonson, Zacharaya Shabka, Konrad Chlupka, Bawang Goh, and Georgios Zervas. https://github.com/cwfparsonson/soa_driving, May 2020a. URL https://doi.org/10.5281/zenodo.3865905.

Christopher Parsonson, Zacharaya Shabka, Konrad Chlupka, Bawang Goh, and Georgios Zervas. https://doi.org/10.5522/04/12356696.v1, May 2020b. URL https://rdr.ucl.ac.uk/articles/An_Artificial_

Intelligence_Approach_to_Optimal_Control_of_Sub-Nanosecond_SOA-Based_Optical_Switches/12356696/1.

Christopher W. F. Parsonson, Alexandre Laterre, and Thomas D. Barrett. Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. 2022. doi: 10.48550/ARXIV.2205.14345. URL https://arxiv.org/abs/2205.14345.

Christopher Parsonson et al. Traffic generation for benchmarking data centre networks. *Optical Switching and Networking*, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Alejandro Perdomo-Ortiz, Neil Dickson, Marshall Drew-Brook, Geordie Rose, and Alán Aspuru-Guzik. Finding low-energy conformations of lattice protein models by quantum annealing. *Scientific Reports*, 2:571, 2012.

Julien Perolat, Bart De Vylder, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer, Paul Muller, Jerome T. Connor, Neil Burch, Thomas Anthony, Stephen McAleer, Romuald Elie, Sarah H. Cen, Zhe Wang, Audrunas Gruslys, Aleksandra Malysheva, Mina Khan, Sherjil Ozair, Finbarr Timbers, Toby Pohlen, Tom Eccles, Mark Rowland, Marc Lanctot, Jean-Baptiste Lespiau, Bilal Piot, Shayegan Omidshafiei, Edward Lockhart, Laurent Sifre,

Nathalie Beauguerlange, Remi Munos, David Silver, Satinder Singh, Demis Hassabis, and Karl Tuyls. Mastering the game of stratego with model-free multiagent reinforcement learning. *Science*, 378(6623):990–996, 2022. doi: 10.1126/science.add4679. URL https://www.science.org/doi/abs/10.1126/science.add4679.

Tekla Perry. Move Over, Moore's Law: Get Ready for Huang's Law. Technical report, 2018.

D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020. URL https://openreview.net/forum?id=IVc9hqgibyB.

A. Pucher, E. Gul, R. Wolski, and C. Krintz. Using trustworthy simulation to engineer cloud schedulers. In *2015 IEEE International Conference on Cloud Engineering*, pages 256–265, 2015. doi: 10.1109/IC2E.2015.14.

Arslan Sajid Raja, Sophie Lange, Maxim Karpov, Kai Shi, Xin Fu, Raphael Behrendt, Daniel Cletheroe, Anton Lukashchuk, Istvan Haller, Fotini Karinou, Benn Thomsen, Krzysztof Jozwik, Junqiu Liu, Paolo Costa, Tobias Jan Kippenberg, and Hitesh Ballani. Ultrafast optical circuit switching for data centers using integrated soliton microcombs. *Nature Communications*, 12(1): 5867, Oct 2021. ISSN 2041-1723. doi: 10.1038/s41467-021-25841-8. URL https://doi.org/10.1038/s41467-021-25841-8.

Bown Ralph. Time division multiplex system for signals of different bandwidth, u.s. patent us2919308a, 1959.

C.Radhakrishna Rao. Diversity and dissimilarity coefficients: A unified approach. *Theoretical Population Biology*, 21(1):24–43, 1982. ISSN 0040-5809. doi: https://doi.org/10.1016/0040-5809(82)90004-1. URL https://www.sciencedirect.com/science/article/pii/0040580982900041.

Kai Ren, Garth Gibson, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's Adolescence; A Comparative Workloads Analysis from Three Research Clusters. In *High Performance Computing, Networking, Storage and Analysis (SCC)*, page 1452, 2012. ISBN 978-1-4673-6218-4. doi: 10.1109/SC.Companion.2012.253.

N. S. Ribeiro, A. L. Toazza, C. M. Gallep, and E. Conforti. Rise time and gain fluctuations of an electrooptical amplified switch based on multipulse injection in semiconductor optical amplifiers. *IEEE Photonics Technology Letters*, 21 (12):769–771, 2009. doi: 10.1109/LPT.2009.2017731.

Michael Riordan. The Lost History of the Transistor. *IEEE Spectrum*, 2004.

David Rotman. We're not Prepared for the End of Moore's Law. *MIT Technology Review*, 2020.

Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787472. URL https://doi.org/10.1145/2785956.2787472.

Cynthia Rudin, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong. Interpretable machine learning: Fundamental principles and 10 grand challenges. *ArXiv*, abs/2103.11251, 2021.

G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.

Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration. 2018.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, jul 1959. ISSN 0018-8646. doi: 10.1147/rd.33.0210. URL https://doi.org/10.1147/rd.33.0210.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17.

SCIP. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, 2022.

Zacharaya Shabka and Georgios Zervas. Nara: Learning network-aware resource allocation algorithms for cloud data centres, 2021.

Zacharaya Shabka, Michael Enrico, Nick Parsons, and Georgios Zervas. One-shot, offline and production-scalable pid optimisation with deep reinforcement learning. *arXiv*, 2022. doi: 10.48550/ARXIV.2210.13906. URL https://arxiv.org/abs/2210.13906.

Claude E. Shannon. Programming a computer for playing chess. 1950.

S. Shen, V. Van Beek, and A. Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474, 2015. doi: 10.1109/CCGrid.2015.60.

Kai Shi, Sophie Lange, Istvan Haller, Daniel Cletheroe, Raphael Behrendt, Benn Thomsen, Fotini Karinou, Krzysztof Jozwik, Paolo Costa, and Hitesh Ballani. System demonstration of nanosecond wavelength switching with burst-mode pam4 transceiver. In *45th European Conference on Optical Communication (ECOC 2019)*, pages 1–4, 2019. doi: 10.1049/cp.2019.1034.

David Silver. *Reinforcement Learning and Simulation-Based Search in Computer Go*. PhD thesis, University of Alberta, 2009.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587): 484–489, January 2016. ISSN 0028-0836. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017. ISSN 1476-4687. doi: 10.1038/nature24270. URL https://doi.org/10.1038/nature24270.

J Simmons. Optical Network Design and Planning. *Optical Network Design and Planning*, 2008. ISSN 1935-3839. doi: 10.1007/978-0-387-76476-4.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The solomon computer. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, AFIPS '62 (Fall), page 97–107, New York, NY, USA, 1962. Association for Computing Machinery. ISBN 9781450378796. doi: 10.1145/1461518.1461528. URL https://doi.org/10.1145/1461518.1461528.

Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv*, 2022. doi: 10.48550/ARXIV.2201.11990. URL https://arxiv.org/abs/2201.11990.

Brian A Sparkes. *The red and the black: studies in Greek pottery*. Routledge, 2013.

Rajiv Srivastava, Rajat Kumar Singh, and Yatindra Nath Singh. Design analysis of optical loop memory. *Journal of Lightwave Technology*, 27(21):4821–4831, 2009. ISSN 07338724. doi: 10.1109/JLT.2009.2026493.

Ian Stoica. The future of computing is distributed. https://www.datanami.com/2020/02/26/the-future-of-computing-is-distributed/, Feb 2020.

T. Sutili, P. Rocha, C. M. Gallep, and E. Conforti. Energy efficient switching technique for high-speed electro-optical semiconductor optical amplifiers. *Journal of Lightwave Technology*, 37(24):6015–6024, 2019. doi: 10.1109/JLT.2019.2945168.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988. ISSN 1573-0565. doi: 10.1007/BF00115009. URL https://doi.org/10.1007/BF00115009.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA, second edition, 2018. ISBN 0262039249. URL http://incompleteideas.net/book/the-book-2nd.html.

Bruno Taglietti, Tiago Sutili, Rafael C Figueiredo, Rafael Ferrari, and Evandro Conforti. Semiconductor optical amplifier space switch BER improvement and guard-time reduction through feed-forward filtering. *Optics Communications*, 426:295–301, 2018. ISSN 0030-4018. doi: https://doi.org/10.1016/j.optcom.2018.05.065. URL http://www.sciencedirect.com/science/article/pii/S0030401818304504.

Marius Hobbhahn Tamay. Trends in GPU Price-Performance. Technical report, 2022.

Neil Thompson and Svenja Spanuth. The Decline of Computers As a General Purpose Technology: Why Deep Learning and the End of Moore's Law are Fragmenting Computing. *SSRN Electronic Journal*, 2018. ISSN 1556-5068. doi: 10.2139/ssrn.3287769.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2017.

Michel Tokic and Günther Palm. Value-Difference Based Exploration: Adaptive Control between Epsilon-Greedy and Softmax. In Joscha Bach and Stefan Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence*, pages 335–346, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24455-1.

Alexander Trott, Stephan Zheng, Caiming Xiong, and Richard Socher. Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards. In *NeurIPS*, 2019.

Rodney S. Tucker, Rodney S. Tucker, and Ivan P. Kaminow. High-Frequency Characteristics of Directly Modulated InGaAsP Ridge Waveguide and Buried Heterostructure Lasers. *Journal of Lightwave Technology*, 2(4):385–393, 1984. ISSN 15582213. doi: 10.1109/JLT.1984.1073654.

A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950. ISSN 00264423, 14602113. URL http://www.jstor.org/stable/2251299.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

F. Van Den Bergh and A. P. Engelbrecht. Training product unit networks using cooperative particle swarm optimisers. *Proceedings of the International Joint Conference on Neural Networks*, 1(4):126–131, 2001. doi: 10.1109/ijcnn.2001.939004.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. FeUdal networks

for hierarchical reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3540–3549. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/vezhnevets17a.html.

O. Vinyals, I. Babuschkin, and W.M. Czarnecki. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 2019.

Kaixin Wang, Bingyi Kang, Jie Shao, and Jiashi Feng. Improving generalization in reinforcement learning with mixture regularization, 2020. URL https://arxiv.org/abs/2010.10814.

Lin Wang, Xinbo Wang, Massimo Tornatore, Kwang Joon Kim, Sun Me Kim, Dae Ub Kim, Kyeong Eun Han, and Biswanath Mukherjee. Scheduling with machine-learning-based flow detection for packet-switched optical data center networks. *Journal of Optical Communications and Networking*, 10(4):365–375, 2018. ISSN 19430620. doi: 10.1364/JOCN.10.000365.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. 2019. doi: 10.48550/ARXIV. 1909.01315. URL https://arxiv.org/abs/1909.01315.

Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Zhijao Jia, Dheevatsa Mudigere, Ying Zhang, Anthony Kewitsch, and Manya Ghobadi. Topoopt: Optimizing the network topology for distributed dnn training. *arXiv preprint arXiv:2202.00433*, 2022.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015. URL http://arxiv.org/abs/1511.06581. cite arxiv:1511.06581Comment: 15 pages, 5 figures, and 5 tables.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

Lukas M. Weber, Wouter Saelens, Robrecht Cannoodt, Charlotte Soneson, Alexander Hapfelmeier, Paul P. Gardner, Anne Laure Boulesteix, Yvan Saeys, and Mark D. Robinson. Essential guidelines for computational method benchmarking. *Genome Biology*, 20(1):1–12, 2019. ISSN 1474760X. doi: 10.1186/s13059-019-1738-8.

Joel Webster. "SERIES 7000 - 384x384 port Software-Defined Optical Circuit Switch. https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabl asp, 2022. URL https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabl asp. [Online]. Available: https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabl asp.

Kyle Wiggers. Nvidia makes massive language model available to enterprises. https://venturebeat.com/uncategorized/nvidia-makes-massive-language-model-available-to-enterprises/, Nov 2021.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011. ISBN 978-0-521-19527-0.

R. Wolski and J. Brevik. Qpred: Using quantile predictions to improve power usage for private clouds. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 179–187, 2017. doi: 10.1109/CLOUD.2017.31.

Tailin Wu and Max Tegmark. Toward an artificial intelligence physicist for unsupervised learning. *Phys. Rev. E*, 100:033311, Sep 2019. doi: 10.1103/PhysRevE.100.033311. URL https://link.aps.org/doi/10.1103/PhysRevE.100.033311.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016. URL http://arxiv.org/abs/1609.08144. cite arxiv:1609.08144.

Yahoo. Yahoo Computing Systems Data. Technical report, 2015. URL https://webscope.sandbox.yahoo.com/catalog.php?datatype=s{&}guccounter=1.

Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies, 2021.

Georgios Zervas, Hui Yuan, Arsalan Saljoghei, Qianqiao Chen, and Vaibhawa Mishra. Optically disaggregated data centers with minimal remote memory latency: Technologies, architectures, and resource allocation [invited]. *Journal of Optical Communications and Networking*, 10(2):A270–A285, 2018. doi: 10.1364/JOCN.10.00A270.

Jesse Zhang, Haonan Yu, and Wei Xu. Hierarchical reinforcement learning by discovering intrinsic options. In *International Conference on Learning Representations*, 2021a. URL https://openreview.net/forum?id=r-gPPHEjpmw.

Tianjun Zhang, Huazhe Xu, Xiaolong Wang, Yi Wu, Kurt Keutzer, Joseph E. Gonzalez, and Yuandong Tian. Noveld: A simple yet effective exploration criterion. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021b. URL https://openreview.net/forum?id=CYUzpnOkFJp.