Network System Optimization with Reinforcement Learning: Methods and Applications

by

Hongzi Mao

B.S., The Hong Kong University of Science and Technology (2015)
S.M., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 24, 2020

Certified by .....
Mohammad Alizadeh
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Network System Optimization with Reinforcement Learning: Methods and Applications

by

Hongzi Mao

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Networked systems rely on many control and decision-making algorithms. Classical approaches to designing and optimizing these algorithms, developed over the last four decades, are poorly suited to the diverse and demanding requirements of modern networks and applications. In the classical paradigm, the system designer assumes a simplified model of the system, specifies some low-level design goals, and develops a fixed algorithm to solve the problem. However, as networks and applications have grown in complexity and heterogeneity, designing fixed algorithms that work well across a variety of conditions has become exceedingly difficult. As a result, classical approaches often sacrifice performance for universality (e.g., TCP congestion control), or force designers to develop point solutions and specialized heuristics for each environment and application.

In this thesis, we investigate a new paradigm for solving challenging system optimization problems. Rather than design fixed algorithms for each problem, we develop systems that can learn to optimize the performance on their own using modern reinforcement learning. In the proposed approach, the system designer does not develop specialized heuristics for low-level design goals using simplified models. Instead, the designer architects a framework for data collection, experimentation, and learning that discovers the low-level actions that achieve high-level resource management objectives automatically.

We use this approach to build a series of practical network systems for important applications, including context-aware control protocols for adaptive video streaming, and schedulers for data-parallel and large-scale data processing workloads. We also use the insights from these systems to identify common problem structures and develop new reinforcement learning techniques for designing robust data-driven network systems.

Thesis Supervisor: Mohammad Alizadeh
Title: Associate Professor of Electrical Engineering and Computer Science

To my family

Acknowledgments

When I started my PhD study at MIT, little did I anticipate all the unknowns and uncertainties along the journey. All I knew was an incredible adventure ahead, and in a hindsight, what a ride! There were up times when new interesting ideas arrived; and there were definitely a lot of down times when nothing seemed to work or all our good work just kept being rejected. But all the bitter and sweet made the past five years a really rewarding experience for me to learn and grow. Looking back, I could have never gone through this journey without all the kindest help I received throughout the way. I am hugely indebted to all the amazing people I get to know over the years.

First, I am immensely grateful to my advisor, Mohammad Alizadeh. I cannot adequately stress how fortunate I am to have Mohammad as an ideal advisor. At a very early stage of my PhD study, Mohammad was creative and brave enough to push the two of us into an unknown and fresh research area that merges networking and machine learning. In this new territory, his ingenuity, discipline and candor were the torch that lit our way. Facing so many unknowns, no one knew how to work out all the details and I could have given up at so many points in time but it was Mohammad that kept me forward. Throughout the journey, Mohammad always kept me reaching for higher goals, but also diving deep to understand all the details to breathe with a working system. He taught me not to be afraid of risks, but also to take risks in a calculated and planned manner. He reminded me not to fixate my mind only on the low hanging fruits but to be free and bold to hunt for the next big thing. For all the lessons and good times, Mohammad, I am forever grateful.

I would also like to express my utmost gratitude to Dina Katabi and Fadel Adib, who advised my undergraduate intern research and led me to the wonderland of MIT for the first time. They taught me the spirit of hard working and they inspired me to never stop imagining the possibility and impact. I couldn't have even dreamed that one day I could contribute my code to demo a project live in front of the President of the United States. I am also tremen-

dously indebted to Dina for encouraging me to explore and pursue the broader research areas in computer systems and machine learning.

I want to thank my committee members, Sam Madden, Pulkit Agrawal and Ion Stoica. I am really lucky to have these phenomenal professors from different areas to join my thesis committee. They have brought unique insights from their domains to shepherd my thesis and provided invaluable suggestions for future research directions.

Next, I thank my internship mentors in industry, Srikanth and Ishai from Microsoft Research, Shannon and Eytan from Facebook and Chenjie from Google DeepMind. They brought me into a whole different world of engineering in production. They taught me what it meant to develop robust software systems with the intention to scale to billions of people. They opened my eyes to what impact is possible with the access to oceans of users and compute resource. I am lucky to get to know the best values and beliefs from all these great companies.

Also, I am grateful to my many other collaborators over the years: Zach, Chen-Yu, Robert, Fredo, Dinesh, Kanthi, Sandeep, Sachin, Ravi, Malte, Shaileshh, Zili, Drew, Shaun, Yuandong, Ryan, Pari, Tim, Nesime, Olga, Vikram, Jialin, Akshay, Hanrui, Jiacheng, Haonan, Ravichandra, Mehrdad, Songtao, Frank, Wei-Hung, Song, Hao, Lujing. Each one of them inspired me in a different way and they have played a monumental role to bring our work to a different level. In particular, I would like to extend my extra gratitude to Ravi, Malte and Shaileshh. They were senior graduate students or postdocs when I was a noob just starting to know how to do the basics of research. Yet, they treat me equally and kindly guide me through all the nuts and bolts both in elegant system building and rigorous mathematical derivation. They are the role models that set me the quality of research.

Equally importantly, I am humble and privileged to mentor some truly exceptional undergraduate students at MIT either through UROP or summer internship. Calvin extended the Pensieve project by introducing human-in-the-loop reward feedback and he executed the project end-to-end in just a short semester. His creativity and motivation really set a high bar for the ideal type of collaborators. Zili started with finding the optimal baseline for Decima. We mostly collaborated remotely but Zili made the whole process smooth and efficient through his determination. I am fortunate to continue collaborating with him on project Metis for interpreting RL systems. Haonan first helped contribute to one of the most important environments in our Park project. It was my pleasure to then saw him led the RL caching project where he invented a new subsampling technique for a family of RL problems.

I am thankful to all the members I got to befriend with in the NMS group: Hari, Anirudh, Ravi, Amy, Jonathan, Peter, Tiffany, Vikram, Songtao, Prateesh, Mehrdad, Deepti, Venkat, Pari, Lei, Akshay, Frank, Vibhaa, Ravichandra, Shaileshh, Radhika, Arjun, Inho, Manya, Kshiteej, Pouya, Arash, Seo Jin, James, Ahmed. I want to also specially thank the wonderful people I shared the office with: Peter, Prateesh, Mehrdad, Ravichandra and Lei. You guys are the everyday fuel of joy and fun inside and outside research.

I also want to thank all my other friends at Cambridge, around the States and across the oceans: Anbang, Changchen, Chao, Deepak, Dong, Ezz, Favyen, Feihu, Ge, Guo, Haitham, Hao, Hejin, Hongge, Hongzhou, Jiaming, Jiayue, Junchen, Kexin, Lei, Lijie, Lixin, Manxi, Mingmin, Omid, Pan, Qiao, Qinyu, Rahul, Renze, Ronghui, Rui, Ruizhi, Shichao, Shumo, Sicheng, Swarun, Tao, Tian, Tien-Ju, Wenbo, Xinhao, Yi, Yifei, Yifeng, Yongjia, Yu, Yue, Yueshen, Yunfei, Yuqian, Yutao, Yuzhe, Zeyuan, Zhi, Zhihong, Zhiqian, Zhiwei, Zhonglu. I lost count how many times I share the laughters and tears with the good friends after big paper deadlines, holiday and events, or just some photography expeditions around the New England area. You really make here a home when we are all away from home.

Lastly, I turn to my family. Words cannot express my gratitude towards my parents, Jianbo and Liyu, for the unconditional love and perpetual support in my entire life. You encouraged and fully trusted in me when I chose an unfamiliar career out of your expectation. My beloved fiancée, Yujie, we merged our path despite all the difficulties and uncertainties. You have sacrificed too much to hold up my dreams; I am forever indebted and I will do everything I can to cherish yours. Also my parents-in-law, Yin and Jinyi, thank you for the support and trust that transcend our distance apart. My family is the ultimate force that keeps me going even when I sometimes seem to lose all the directions and hope. This thesis is dedicated to you.

Previously Published Material

Chapter 3 revises a previous publication [204]: H. Mao, R. Netravali, M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. SIGCOMM, 2017.

Chapter 4 revises a previous publication [206]: H. Mao, M. Schwarzkopf, S. Venkatakrishnan, Z. Meng, M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. SIGCOMM, 2019.

Chapter 5 revises a previous publication [207]: H. Mao, S. Venkatakrishnan, M. Schwarzkopf, M. Alizadeh. ICLR, 2019.

Chapter 6 revises a previous publication [203]: H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, R. Addanki, M. Khani, S. He, V. Nathan, F. Cangialosi, S. Venkatakrishnan, W.-H. Weng, S. Han, T. Kraska, M. Alizadeh. Park: An Open Platform for Learning Augmented Computer Systems. NeurIPS, 2019.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Action through inaction. — Lao Tzu

Modern network systems rely on many control and resource management algorithms. Resource management refers broadly to the methods used to determine how to allocate compute and communication resources (e.g., CPU cycles, memory blocks, network bandwidth, etc.) to different applications, and to manage the contention for resources among applications. Resource management problems are ubiquitous and appear in all kinds of networks and systems. Examples include job scheduling in compute clusters [121, 123, 305], bitrate adaptation for video streaming [146, 329], network congestion control [320, 319, 86], relay selection for Internet telephony [330], virtual machine allocations in cloud computing [139] and more.

These problems have a long history and solutions draw upon many areas across computer science and applied mathematics. However, in practice, solutions often devolve into meticulously designed heuristics. Perusing recent research in the field, the typical design flow is: (1) construct a simplified model of the system optimization problem; (2) break down high-level optimization objectives (e.g., minimize user perceived application delay) into low-level design goals (e.g., minimize network packet queuing delay); (3) come up with heuristics to achieve these design goals under the simplified model and extensively tune the heuristics to reach good performance in actual systems.

This whole design philosophy heavily involves human engineers in the loop: as humans, we tend to focus on solving fixed, well-defined and semantically self-contained problems. Unfortunately, it is becoming increasingly difficult to design highly performant and robust systems for modern networks using this approach. First, the underlying components in many modern applications interact in complex, non-linear ways and are often extremely difficult

to model accurately. For example, in cluster scheduling, the runtime of a task varies with data locality, server characteristics, interactions with other tasks, and interference on shared resources such as memory caches, network bandwidth, etc [79, 121]. Second, practical optimization algorithms must operate in a wide range of heterogeneous and potentially unknown conditions which are impossible to captured by fixed and simplified models completely. A video streaming client, for instance, has to choose the bitrate for future video chunks based on noisy forecasts of available bandwidth [328], and operate well for different codecs, screen sizes and available network types. Third, many networks have complicated structures and requirements; it is often infeasible to find a correct set of low-level design goals that perfectly add up to the high-level application performance. As a result, practical solutions must combine and tune several heuristics to optimize performance — a tedious process that may need to be repeated when some aspect of the workload or the deployment environment changes. Therefore, state-of-the-art network systems often sacrifice performance for simplicity and universality, or force designers to develop point solutions and specialized heuristics for each environment and application.

In this thesis, we take a step back and ask what is the most natural way for *machines* to optimize complex networking systems. Rather than explicitly design and tune fixed algorithms for each problem, we seek to enable systems to learn to efficiently optimize the performance *on their own*. In our proposed approach, the system operator does not design specialized heuristics for low-level design goals using a simplified model of the system. Instead, she architects a framework for *data collection, experimentation, and learning* to discover the low-level actions that achieve a high-level optimization objective automatically.

1.1 General Methodology

The broad vision of this thesis is to combat heterogeneity across networks and applications without compromising performance using data-driven optimizations. Network capabilities, particularly at the "edge", can vary by orders of magnitude across different geographical regions and technologies (e.g., variable-bandwidth cellular networks to high-speed datacenter networks). Modern networked applications are also very diverse: large-scale cloud services, live/on-demand video streaming, distributed data processing, video analytics, large-scale AI training, and many more. The days when applications simply needed point-to-point, best-effort communication service from the network are long gone. Different networks and appli-

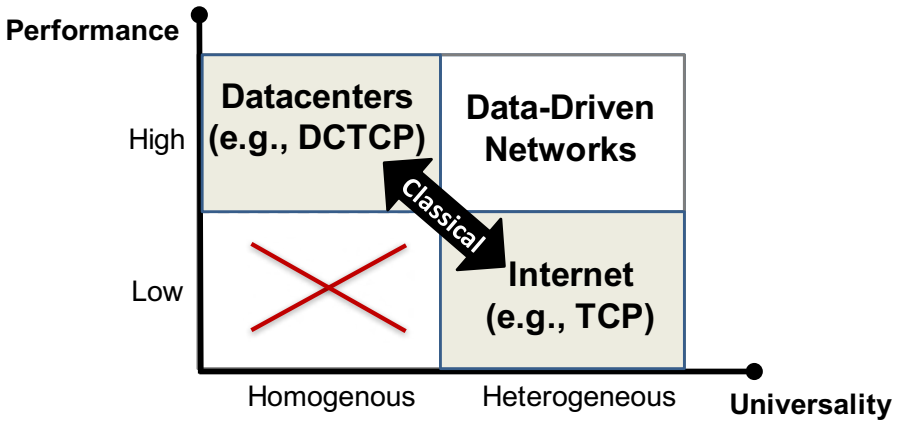


Figure 1-1: The tension between performance and universality. Data-driven network systems optimization can resolve the tension between performance and universality that plagues classical approaches.

cations would clearly benefit from better-tailored optimization strategies. However, classical approaches to network systems optimization, developed over the last four decades, are poorly suited to this task. Approaches that rely on fixed algorithms face a fundamental tension between performance and universality (Figure 1-1). Therefore, algorithms designed for heterogeneous environments often sacrifice performance for universality. A prominent example is TCP congestion control, which operates across a wide variety of networks but is not optimal in any of them. While the classical paradigm has been successful in some homogeneous settings, where we can model the network quite accurately and design algorithms with strong performance (e.g., datacenter transport [16, 18, 15]), we currently lack design techniques for high-performance networked systems across heterogeneous environments. This work seeks to resolve this tension and build networks, augmented with data-driven learning techniques, that are both highly adaptive to heterogeneous conditions and provide strong performance.

We are inspired by the recent success of applying machine learning to other domains that involve complex decision-making problems [220, 277, 2]. In particular, reinforcement learning (RL) deals with agents that learn to make decisions directly from the experience of interacting with the environment. The agent starts knowing nothing about the task at hand and learns by reinforcement — a reward signal that it receives based on how well it is doing on the task. RL has a long history [285], but recently the combination of RL with modern deep learning techniques for training large neural networks [180] has produced impressive successes in applications such as playing video games [220], Computer Go [277, 279, 278, 266], robotics control [186], datacenter cooling [94], etc.

At a high level, we believe that several benefits of RL are particularly well-suited to system optimization problems. These benefits leverage many inherent system properties and provide a data-driven solution for optimization problems that traditional approach struggle to tackle:

1. By continuously learning from the real experiences of interacting with a system environment, RL agents directly optimize for the actual workload and operating conditions as opposed to relying on inaccurate system models.
2. With the use of general purposed and powerful function approximators such as deep neural networks [180], RL agents can incorporate a rich collection of raw observations (e.g., application structure, performance statistics, resource usage patterns, etc.) to improve decisions across heterogeneous workloads and environments.
3. RL agents can learn to optimize a variety of high-level optimization objectives (e.g., user-level perceived video playback delay) without prior knowledge of how low-level metrics (e.g., transport-layer queueing delay, CDN cache hit ratio, backend video server utilization, etc.) impact the objective.
4. The underlying problem structures of many systems involve combinatorial optimization problems (e.g., some variants of the knapsack packing problem for resource packing or job scheduling), which generally lack a generic optimal solution. RL can help automating the process of improving the optimization solution for individual systems.
5. System operation decisions are often highly repetitive, making it easy to collect an abundance of training data to train RL models. This fact largely negates one of the potential drawbacks of RL approaches in practice — their high sample-complexity (i.e., need for a large amount of training data).

In practice, however, there are several important challenges to solve when applying modern RL to system optimization problems. In many cases, off-the-shelf RL methods are often insufficient to deal with the complexity and scale of the system optimization problems. To develop this thesis, we had to design efficient RL problem formulation for different system problems, develop scalable representations for the learning models and invent new RL algorithms to efficiently train systems with long sequences of stochastic workload patterns. In the following, we will overview these research challenges and new learning techniques we discovered with concrete system optimization problems.

1.2 Overview: Systems and Methods Developed

Our goal of this thesis is to develop the systems and algorithmic foundations for building practical data-driven network systems using RL principles. As an overview, we first build a series of practical systems with increasing levels of sophistication, and then use the insights from these systems to identify core principles and approaches for designing practical, robust, and high-performance data-driven networking systems.

1.2.1 Video Streaming

Video streaming has become the predominant application on today's Internet, contributing to over 60% of all Internet traffic [263]. On average, each Internet user spends around 7 hours per week watching videos online [229]. Concurrent with this growth has been a steady rise in user demands on video quality. Many studies have shown that users will quickly abandon video sessions if the quality is not sufficient (e.g., the video fails to play within 2 seconds in the initial loading), leading to significant losses in revenue for content providers [174, 84]. As a primary tool to optimize the video quality (e.g., higher resolution, fewer rebufferings, etc.), content providers deploy adaptive bitrate (ABR) algorithms, which run on client-side video players and dynamically choose a bitrate for each video *chunk* (e.g., 4-second block). based on observations such as the estimated network throughput and playback buffer occupancy. Their goal is to maximize the user's quality of experience (QoE) by adapting the video bitrate to the underlying network conditions.

However, selecting the optimal bitrates can be very challenging due to (1) the variability of network throughput [145, 328]; (2) the conflicting video QoE requirements (e.g., high bitrate vs minimal rebuffering); (3) the cascading effects of bitrate decisions (e.g., selecting a high bitrate may drain the playback buffer to a dangerous level and cause rebuffering in the future); and (4) the coarse-grained nature of ABR decisions. Despite the abundance of recently proposed schemes, state-of-the-art ABR algorithms suffer from a key limitation: they use fixed control rules based on simplified or inaccurate models of the deployment environment. As a result, existing schemes inevitably fail to achieve optimal performance across a broad set of network conditions and QoE objectives.

We developed Pensieve, a system that automatically generates strong ABR algorithms using modern RL. Pensieve is the first system to train a neural network model that directly selects bitrates based on observations collected by client video players. Pensieve does not

rely on pre-programmed models or assumptions about the environment. Instead, it mines information about the actual experience of past bitrate choices to optimize its control policy for the characteristics of the network. As a result, Pensieve automatically learns ABR algorithms that adapt to a wide range of environments and QoE metrics. In trace-driven and real world experiments, Pensieve outperforms state-of-the-art approaches in all cases, improving the average quality of experience (QoE) by 12%–25%. Also, in a deployment study at Facebook's production web-based video platform, a variant of Pensieve outperforms the existing ABR controller, especially on challenging and highly variant network conditions [202].

1.2.2 Workload Scheduling

In the era of cloud computing, efficient utilization of expensive compute clusters matters immensely for enterprises: even small improvements in utilization can save millions of dollars at scale [32, §1.2]. Cluster schedulers are key to realizing these savings. A good scheduler packs work tightly to reduce resource fragmentation [123, 121, 304], prioritizes jobs according to high-level metrics such as user-perceived latency [305], and avoids inefficient configurations [100]. At the current production, however, most cluster schedulers rely on simple heuristics (e.g., fair resource sharing) that prioritize generality, ease of understanding, and straightforward implementation over achieving the ideal performance on a specific workload. These systems forego potential performance optimizations because equipping schedulers with workload-specific information require expert knowledge and significant effort to devise, implement, and validate. For many organizations, these skills are either unavailable, or uneconomic as the engineer labor cost exceeds potential savings.

To side-step this trade-off, we developed Decima, an RL-based scheduling service for data processing jobs. Decima demonstrates, for the first time, the feasibility of learning workload-specific scheduling policies for complex, graph-structured jobs entirely through experience. To successfully learn such scheduling policies, Decima had to tackle several challenges that off-the-shelf RL techniques cannot readily handle:

1. Cluster schedulers must scale to thousands of machines and hundreds of jobs (each may contain dozens of computation stages). This leads to much larger problem sizes compared to conventional RL applications (e.g., game-playing [220, 277], robotics control [191, 268]). Moreover, the online job arrival makes the problem incompatible to most static RL algorithms that require fixed-sized vectors as inputs. We designed a scalable neural network architecture that uses a speical *graph neural network* [170, 35]

to process job and cluster information without manual feature engineering. Our neural networks reuse a small set of building block operations to process job DAGs, irrespective of their sizes and shapes, and to make scheduling decisions, irrespective of the number of jobs or machines.

2. To express the scheduling action, the scheduler must map potentially thousands of runnable stages to available servers. The exponentially large space of mappings poses a challenge for RL algorithms, which require adequate "exploration" (i.e., trying out different scheduling action sequences in order to observe their empirical outcome) to learn a good policy. We leverage an event-driven scheduling logic to ydevelop a two-dimensional action representation that determine both the job and degree of parallelism (i.e., how many servers to run) in one concise scheduling action. This representation substantially reduces model complexity compared to naive encodings of the scheduling problem, which is key to efficient learning.
3. Standard RL algorithms struggle to "kickstart" the learning with continuous streaming job arrivals, as the initial RL model makes poor decisions in early stages of training. With an unbounded stream of incoming jobs, the initial model inevitably accumulates an insurmountable backlog of jobs from which it can never recover, which prohibits further training. We develop a curriculum learning scheme to terminate training "episodes" early in the beginning, and gradually grow the episode length. This allows the policy to learn to handle simple, short job sequences first, and to then graduate to more challenging arrival sequences.

In our experiment, we have built a Decima prototype that integrates with Spark [333] on a 25-node cluster. Empirical evaluation shows that Decima improves average job completion time by at least 21% over hand-tuned scheduling heuristics, achieving up to 2× improvement during periods of high cluster load. Also, Decima extends to multi-resource scheduling of CPU and memory, where it improves average job completion time by 32-43% over prior state-of-the-art schemes.

1.2.3 RL in Input-Driven Environments

Over the course of building these data-driven systems, we have also identified some common underlying problem structures that fundamentally require building new RL algorithms. Specifically, many resource management and system optimization problems involve an ex-

ogenous, stochastic input process that affects the dynamics of the system. Queuing systems [171, 166] are an example; their dynamics are governed by not only the decisions made within the system (e.g., scheduling, load balancing, congestion control) but also the arrival process that brings work (e.g., jobs, network packets, customers) into the system. We found that this input process creates huge variance that is hard to handle for existing RL approaches. At a high level, the randomness in the input process (e.g., sequence of large or small jobs) can make it impossible for RL algorithms to tell whether the observed outcome of two decisions differs due to differences in the input process, or due to the quality the policy's decisions.

To reduce this variance, we derive a bias-free, input-dependent baseline technique for RL in such environments [207]. During training, the idea is to condition the reward feedback signal on the observed input process when assessing the impact of an action. For example, an RL-based scheduler can now distinguish whether the positive reward feedback results from a good scheduling decision (hence to reinforce the decisions) or from an easy-to-schedule job sequence (hence to ignore the noise). Therefore, this conditioning technique effectively isolates the contribution of the RL decision from the input process noise. Using a baseline for variance reduction is common in RL, but input-dependent baselines are unusual because they depend not only the current state of the environment but also on the entire *future* input sequence.

We formally define the set of problems applicable to this new RL technique and analytically show its benefits over existing approaches. We also present efficient algorithms to compute input-dependent baselines. Our experiments show that this new method not only consistently improves training stability and eventual policy performance across computer system environments such as job scheduling and network control, it also benefits robotic applications such as MuJoCo locomotion [294] in the presence of stochastic disturbances.

1.2.4 An Open Platform for Learning-Augmented Systems

RL-based systems research is inherently interdisciplinary and creates abundant opportunities to draw intellectual connections between the networking, systems, and machine learning areas. The landscape of building learning-based systems is vast, ranging from centralized control problems (e.g., a scheduling agent responsible for an entire computer cluster) to distributed multi-agent problems where multiple entities with partial information collaborate to optimize system performance (e.g., network congestion control with multiple connections sharing bottleneck links). Further, the control tasks manifest at a variety of timescales,

from fast, reactive control systems with sub-second response-time requirements (e.g., admission/eviction algorithms for caching objects in memory) to longer term planning problems that consider a wide range of signals to make decisions (e.g., VM allocation/placement in cloud computing).

A key obstacle for research on learning-augmented systems is the lack of good benchmarks for evaluating solutions, and the absence of an easy-to-use platform for experimenting with RL algorithms in systems. Conducting research on learning-based systems currently requires significant expertise to implement solutions in real systems, collect suitable real-world traces, and evaluate solutions rigorously. To lower this barrier of entry for future machine learning researchers to innovate in computer systems, we developed Park, an open, extensible platform that uses a common RL interface to connect to a suite of computer system environments [203]. For each environment, Park defines the control formulation, e.g., the events that triggers an interaction step, the state and action spaces and the reward function. This allows researchers to focus on the core algorithmic and learning challenges, without having to deal with low-level system implementation issues. At the same time, Park makes it easy to compare different proposed learning agents on a common benchmark (e.g., the optimization performance, learning efficiency, etc.), similar to how OpenAI Gym [53] has standardized RL benchmarks for robotics control tasks. Lastly, Park defines a standardized RPC interface [283] between the RL agent and the backend system, making it easy to extend to more environments in the future.

Park includes 12 representative environments that span a wide variety of problems across networking, databases, and distributed systems, and range from centralized planning problems to distributed fast reactive control tasks. In the backend of these environments, the systems are powered by both real systems and high fidelity simulators. With Park and its easy-to-compare benchmarks, we hope to help foster more interaction across research communities and enable researchers to evaluate different AI approaches on real-world networking and systems problems.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 provides a brief primer on the necessary RL background and related work for this thesis. Chapter 3 describes the problem formulation, system design and real-world deployment study of Pensieve, an RL-based

bitrate adaptation system for video streaming. Chapter 4 describes the challenges and techniques of Decima, which develops an RL-based method for efficiently scheduling complex, graph-based jobs with online arrival in data processing clusters. Chapter 5 defines the class of "input-driven" environments, a common problem structure underlying many systems, and develops a general variance reduction technique for RL in these environments. Chapter 6 describes an open platform for developing and comparing future learning-augmented computer systems. In Chapter 7, we conclude the thesis with a list of suggestions for future research directions.

Chapter 2

Background

2.1 Primer on Reinforcement Learning

We provide a brief review on the reinforcement learning (RL) techniques that we use in this thesis; for a detailed survey and rigorous derivations, see e.g., Sutton and Barto's book [285].

Reinforcement learning. Consider the general setting in Figure 2-1, where an RL *agent* interacts with an *environment*. At each step t, the agent observes some state s_t, and takes an action a_t. Following the action, the state of the environment transitions to s_{t+1} and the agent receives a reward r_t as feedback. The state transitions and rewards are stochastic and assumed to be a Markov process: the state transition to s_{t+1} and the reward r_t depend only on the state s_t and the action a_t at step t (i.e., they are conditionally independent of the past).

In the general "model-free" RL setting, the agent only controls its actions: it has no a priori knowledge of the state transition probabilities or the reward function. However, by interacting with the environment, the agent can learn these quantities during training.

For training, RL proceeds in *episodes*. Each episode consists of a sequence of (state, action, reward) observations — i.e., (s_t,a_t,r_t) at each step $t \in [0,1,...,T]$, where T is the episode length . The goal of RL is to maximize the total discounted reward $\mathbb{E}\left[\sum_{t=0}^{T}\gamma^t r_t\right]$, where γ is the discount factor that downweights the reward in the future.

Generally speaking, there are two families to RL algorithms: policy-based methods and value-based methods. At a high level, policy-based methods allow the the agent to directly map the state to a probabilistic distribution over different actions. Value-based method estimates the outcome (i.e., expected discounted total reward) following different actions and the agent selects the action based on the predicted outcome. In different settings, these two

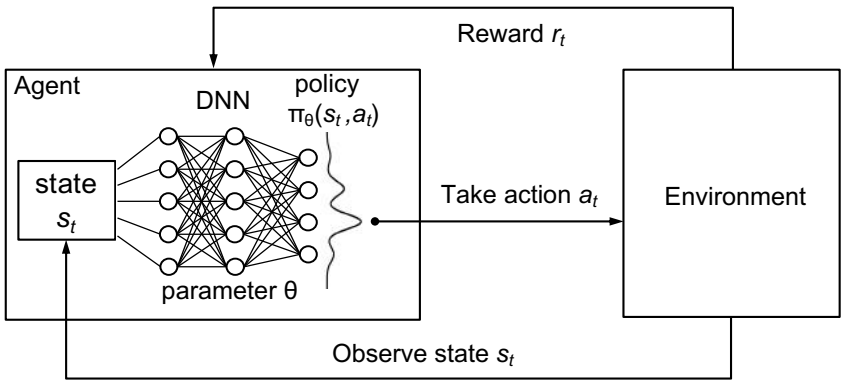


Figure 2-1: A reinforcement learning setting with neural networks [285, 201]. The policy is parameterized using a neural network and is trained iteratively via interactions with the environment that observe its state and take actions.

approaches have different advantages, which we compare next.

Policy. In policy-based RL methods, the agent picks actions based on a *policy* $\pi(s_t,a_t)$, defined as a probability of taking action a_t at state s_t. For most practical problems, the number of possible $\{$state, action$\}$ pairs is too large to store the policy in a lookup table (i.e., tabular form). It is therefore common to use *function approximators* [45, 214], with a manageable number of adjustable parameters, θ, to represent the policy as $\pi_\theta(s_t, a_t)$. Many forms of function approximators can be used to represent the policy. Popular choices include linear combinations of features of the state/action space (i.e., $\pi_\theta(s_t, a_t) = \theta^T \phi(s_t, a_t)$), and, recently, neural networks [130] for solving large-scale RL tasks [220, 277]. An advantage of neural networks is that they do not need hand-crafted features, and that they are end-to-end differentiable for training.

Value. Value-based methods are conceptually more "indirect", since they do not directly map states into actions. A representative value-based method is Q learning: for each state action pair (s_t, a_t), the agent estimates a Q function $Q^\pi(s_t, a_t)$, which aims to predict the total discounted reward after taking action a_t at state s_t and following policy π for the future steps. Large Q values correspond to the actions that likely lead to higher total rewards. The agent can then choose actions based on the values (e.g., by greedily picking the max value) to optimize its policy [285, §5, §6]. Coupled with neural networks (for value prediction), this line of approach was the first deep RL algorithm that achieved super-human performance on complex tasks such as Atari games [220].

Why favoring policy-based approaches? Policy-based methods are usually better suited for the system applications in this thesis. There are two main reasons for making this design

choice. First, the policy π expresses a direct mapping between the states and actions, which conceptually adheres to the current way human engineers design algorithms to control the systems. Given a state, we extract some useful information (e.g., queue sizes, server processing rate estimation, etc.) and we use some heuristics to decide an action based on the information (e.g., rank the queue size normalized by server speed and then join the shortest queue). Importantly, this policy abstraction allows us to verify whether the agent *can* express policies that are (at least) as sophisticated as the existing heuristic. This sanity check can be crucial for figuring out the proper neural network architecture (see §4.5.1 for an example in practice). By contrast, reasoning about the value of each state-action pair can be extremely difficult, as it is determined by many steps of state transitions and actions in the future.

Second, the convergence behavior is different between the two families of RL methods. Value-based methods iteratively optimize its policy — by following the action with the best value — and updates its value model — by estimate the outcome following the new sequences of actions. In essence, it aims to find a fixed point of the Bellman equations [40]. However, if the underlying neural network cannot express the optimal value function, then a value-based method can have difficulty converging because the algorithm is trying to converge to a fixed point that the neural network cannot express. By contrast, with policy-based methods, this issue does not arise, because regardless of the policy network's expressive power, the policy gradient algorithm will optimize for the reward objective over the space of policies that the neural network *can* express.

Policy gradient methods. We focus on a class of RL algorithms that perform training by using *gradient-descent* on the policy parameters [286]. Recall that the objective is to maximize the expected discounted total reward; the gradient of this objective is given by:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(s_t,a_t) Q^{\pi_\theta}(s_t,a_t)\right], \tag{2.1}$$

where $Q^{\pi_\theta}(s_t,a_t)$ is the expected total discounted reward from (deterministically) choosing action a_t in state s_t, and subsequently following policy π_θ [285, §13.2]. The key idea in policy gradient methods is to estimate the gradient using the trajectories of execution with the current policy. Following the *Monte Carlo Method* [136], the agent samples multiple trajectories and uses the empirical total discounted reward, v_t, as an unbiased estimate of

$Q^{\pi_\theta}(s_t,a_t)$. Operationally, it then updates the policy parameters via gradient descent:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(s_t,a_t) v_t, \tag{2.2}$$

where α is the learning rate. This equation results in the REINFORCE algorithm [317]. The intuition of REINFORCE is that the direction $\nabla_\theta \log \pi_\theta(s_t,a_t)$ indicates how to change the policy parameters in order to increase $\pi_\theta(s_t,a_t)$ (i.e., increase the probability of action a_t at state s_t). Equation 2.2 takes a step in this direction; the size of the step depends on the magnitude of the return v_t. The net effect is to reinforce actions that empirically lead to better returns.

In practice, there are two important ingredients to add when estimating the policy gradient. First, a key challenge is the high variance in the gradient estimates, as such variance increases sample complexity and can impede effective learning [269, 219]. A standard approach to reduce variance is to subtract a "baseline" $b(s_t)$ from the total discounted reward v_t [316]. The baseline serves as an unbiased estimation of agent's the average outcome after observing state s_t. Common choices of a baseline include time-based baseline [125], which aligns multiple trajectories on the same time step and slices out the total reward at each step to compute the average; or *value function* [219], which uses another function approximator, partially or entirely different from the policy function approximator, to estimate the expected total discounted reward. As a result, the policy gradient estimation becomes

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(s_t,a_t) A(s_t,a_t), \tag{2.3}$$

where $A(s_t,a_t) = v_t - b(s_t)$ is the "advantage" that compares the empirical total discounted reward with the average expectation. Intuitively, the advantage estimation explicitly reinforce the agent to favor actions that empirically lead to better-than-average outcome.

Second, we must ensure that the RL agent *explores* the action space adequately during training to discover good policies. One common practice to encourage exploration is to add an entropy regularization term to the actor's update rule [219]; this is critical in helping the learning agent converge to a good policy [323]. Concretely, we further modify Equation 2.3 to be

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(s_t,a_t) A(s_t,a_t) + \beta \nabla_\theta H(\pi_\theta(\cdot|s_t)), \tag{2.4}$$

where $H(\cdot)$ is the entropy of the policy (the probability distribution over actions) at each time step. This term encourages exploration by pushing θ in the direction of higher entropy (i.e., higher probability to try different actions, hence more exploration). The parameter β is set to a large value at the start of training (to encourage exploration) and decreases over time to emphasize improving rewards [219].

2.2 Related Work

2.2.1 Classical Network Resource Management

Congestion control. A huge body of research work has been developed for TCP congestion control [153] after the infamous "congestion collapse" events in the 1980s. Numerous designs modified TCP's control policies for network paths with high bandwidth-delay product [102, 167, 129]. Vegas [51] pioneered the idea of delay-based congestion control, followed by schemes like FAST [160] and Compound TCP [288]. DECbit [254] was one of the earliest designs to involve routers in congestion control, an idea emerged from a series development in active queue management [104, 99, 238, 103, 142, 29, 242, 178] and explicit feedback protocols [164, 90, 324].

In the modern age, congestion control continues to attract substantial research interest. A lot of recent work develops specialized algorithms for specific deployment environments. For example, researchers have proposed low latency [16, 17] and deadline-aware [318, 143] congestion control protocols for datacenter networks. New protocols have also been proposed for cellular networks [320, 334, 120], multi-path scenarios [105], and challenging networks where packet loss is a poor indicator of congestion [56]. Also, researchers have developed congestion control schemes that greedily search and optimize control rules based on the feedback [86, 87] or trace emulation [319, 280] of the deployed networks.

Scheduling. Scheduling techniques appear broadly in many distributed systems. A large number of approaches have been proposed for datacenter cluster scheduling: greedy scheduling (e.g., Mesos [141], Borg [305]), min-cost max-flow optimization (e.g., Quincy [151], Firmament [91]), mixed integer-linear programming (e.g., Tetrisched [297]), collaborative filtering (e.g., Paragon [78], Quasar [80]). Sparrow [240], Tarcil [81], and Mercury [163] are distributed schedulers for scheduling sub-second tasks with low latency. Tetris [121] and Graphene [123] develop heuristics for scheduling data-parallel jobs with multiple re-

source requirements and dependencies. Packet scheduling also underlies many datacenter transport designs [18, 126, 31, 224]. Several "coflow" scheduling heuristics [66, 85, 65] have been proposed for typical communication patterns in data-parallel workloads. Related problems appear in multi-tenant distributed systems with an emphasis on isolation and fairness [115, 198, 199], geo-distributed analytics systems with an emphasis on WAN bandwidth costs [311, 310, 250], video analytics [336], distributed machine learning [1], and more.

2.2.2 Machine Learning in Networking and Systems

Anomaly detection. An extensive literature has applied machine learning techniques to detecting anomalies in computer systems and networks. Researchers have applied anomaly detection techniques to network intrusion detection both at the network [82, 338, 144, 298, 223] and system call levels [106, 92, 327] (see [243] for a survey). Machine learning techniques have also been applied to detecting software misconfigurations [241, 331, 36] and vulnerabilities [50, 159].

Performance prediction. Many systems benefit from predictive models of performance. Paragon [78] and Quasar [80] use collaborative filtering to classify incoming workloads based on their anticipated performance on different hardware platforms and their interference profiles across different resources. Ernest [303] models the performance of data analytics jobs based on their behavior for small samples of input data. Wang et al. [315] use CART models to predict the performance of a storage device on an input workload.

Prediction also plays an important role in adaptive video streaming. CFA [156] learns critical features that impact video QoE from historical data and builds models to predict video QoE and suggest the best parameter settings (e.g., CDN, initial bitrate) for a video session. CS2P [328] clusters video sessions by critical features to predict initial throughput and develops a Hidden-Markov-Model predictor to model throughput dynamics. Pytheas [158] goes beyond prediction and casts QoE optimization as an exploration-exploitation task performed at the level of groups of similar sessions. Such *bandits* techniques are related to RL but cannot handle problems where control actions change the state of the system and rewards can be delayed.

System tuning. A variety of approaches have been developed to aid with tuning configuration parameters in different systems. Meta optimization [284] automatically fine-tunes compiler heuristics using evolutionary algorithms. OtterTune [299] uses a combination of supervised learning and nearest neighbor search techniques to suggest configuration knobs

for a database management system for a given workload. CherryPick [14] uses Bayesian Optimization to build application performance models to recommend a cost efficient cloud configuration using a small number of test runs. Jockey [100] and Omega [271] use "auto-scaling" to tune system capacity based on the load, in order to meet job deadlines or opportunistically accelerate jobs using spare resources.

Decision making. A less explored area is the application of machine learning to decision-making tasks within computer systems. For complex scheduling, there exists some learning-based prior work focused on constrained or simplified problems. As ab example, Zhang and Dietterich [337] used RL to schedule human resources for NASA shuttle missions. The formulation in this work assumes all tasks (missions) are known upfront whereas most scheduling problems that we consider require online decision making. An early work applies RL to decentralized packet routing [49] for small problem instances were neural network machinery was not needed. Dai et al. [73] merged graph neural networks with RL to train generalizable agents for graph related NP hard problems such as minimum vertex cover and traveling salesman problems that in some scenario outperformed existing heuristics.

2.2.3 Deep Reinforcement Learning

Reinforcement learning has a long history [285, 45], but the field has seen a surge of interest in recent years, fueled by a series of impressive success stories. The most crucial enabler of these successes is the advances in systems and algorithms that have made training large neural network models possible [180]. Minh et al. [220] showed that a deep neural network model trained via Q-learning can learn to play Atari games from raw pixel input data, with super-human performance in many cases. A large number of Deep RL algorithms have since been developed [191, 267, 301, 265, 219, 186]. Along this line of research, a significant milestone was AlphaGo [277, 279, 278, 266], which became the first computer Go program to beat the best professional human Go players on a full-sized 19×19 board. AlphaGo combines Deep RL with Monte Carlo tree search [55] techniques. Most recently, similar fundamental RL approach equipped with warehouses of parallel compute power has enabled researchers to train strong agents that surpass human champions in real time strategic games like Starcraft [309] and Dota [234]. Beyond well simulated games, most real-world applications of deep RL have been mostly in robotics domain [186, 113]. For example, OpenAI has trained a robotic hand with high degree of dexterity to solve a rubik's cube in simulation and generalized the control to real-world environment, with the ability to survive a wide range of

disturbance, such as object partial occlusion or robotic finger malfunction [235].

The applications of RL in networking and system prior to this thesis has been mostly restricted to simple, tabular setting, where the control rules only apply to a limited, pre-defined space of settings. For example, some existing work has proposed applying tabular RL to bitrate adaptation in video streaming [63, 300, 69, 70], where they explicitly map each bitrate decision under different settings to a value function. Without using function approximators (e.g., neural networks), these schemes do not scale to the large state spaces necessary for good performance in real networks, and their evaluation has been limited to simulations with synthetic network models. As we will describe in the following chapters, decision-making problems in networks and systems contain unique challenges and opportunities for deep RL approaches. We believe unearthing the problem structure and designing new RL approach has tremendous potential in a wide range of network resource management problems.

Chapter 3

Neural Adaptive Video Streaming

3.1 Introduction

Recent years have seen a rapid increase in the volume of HTTP-based video streaming traffic [67, 262]. Concurrent with this increase has been a steady rise in user demands on video quality. Many studies have shown that users will quickly abandon video sessions if the quality is not sufficient, leading to significant losses in revenue for content providers [174, 84]. Nevertheless, content providers continue to struggle with delivering high-quality video to their viewers.

Adaptive bitrate (ABR) algorithms are the primary tool that content providers use to optimize video quality. These algorithms run on client-side video players and dynamically choose a bitrate for each video *chunk* (e.g., 4-second block). ABR algorithms make bitrate decisions based on various observations such as the estimated network throughput and playback buffer occupancy. Their goal is to maximize the user's quality of experience (QoE) by adapting the video bitrate to the underlying network conditions. However, selecting the right bitrate can be very challenging due to (1) the variability of network throughput [145, 328, 339, 320, 335]; (2) the conflicting video QoE requirements (high bitrate, minimal rebuffering, smoothness, etc.); (3) the cascading effects of bitrate decisions (e.g., selecting a high bitrate may drain the playback buffer to a dangerous level and cause rebuffering in the future); and (4) the coarse-grained nature of ABR decisions. We elaborate on these challenges in §3.2.

The majority of existing ABR algorithms (§3.7) develop fixed control rules for making bitrate decisions based on estimated network throughput ("rate-based" algorithms [157, 328]),

playback buffer size ("buffer-based" schemes [146, 282]), or a combination of the two signals [188]. These schemes require significant tuning and do not generalize to different network conditions and QoE objectives. The state-of-the-art approach, MPC [329], makes bitrate decisions by solving a QoE maximization problem over a horizon of several future chunks. By optimizing directly for the desired QoE objective, MPC can perform better than approaches that use fixed heuristics. However, MPC's performance relies on an accurate model of the system dynamics—particularly, a forecast of future network throughput. As our experiments show, this makes MPC sensitive to throughput prediction errors and the length of the optimization horizon (§3.3).

In this thesis, we propose *Pensieve*,[1] a system that learns ABR algorithms automatically, without using any pre-programmed control rules or explicit assumptions about the operating environment. Pensieve uses modern reinforcement learning (RL) techniques [219, 285, 201] to learn a control policy for bitrate adaptation *purely through experience*. During training, Pensieve starts knowing nothing about the task at hand. It then gradually learns to make better ABR decisions through reinforcement, in the form of reward signals that reflect video QoE for past decisions.

Pensieve's learning techniques mine information about the actual performance of past choices to optimize its control policy *for the characteristics of the network*. For example, Pensieve can learn how much playback buffer is necessary to mitigate the risk of rebuffering in a specific network, based on the network's inherent throughput variability. Or it can learn how much to rely on throughput versus buffer occupancy signals, or how far into the future to plan its decisions automatically. By contrast, approaches that use fixed control rules or simplified network models are unable to optimize their bitrate decisions based on all available information about the operating environment.

Pensieve represents its control policy as a neural network that maps "raw" observations (e.g., throughput samples, playback buffer occupancy, video chunk sizes) to the bitrate decision for the next chunk. The neural network provides an expressive and scalable way to incorporate a rich variety of observations into the control policy.[2] Pensieve trains this neural network using A3C [219], a state-of-the-art actor-critic RL algorithm. We describe the basic training algorithm and present extensions that allow a single neural network model to gen-

[1] A pensieve is a device used in Harry Potter [260] to review memories.

[2] A few prior schemes [63, 300, 69, 70] have applied RL to video streaming. But these schemes use basic "tabular" RL approaches [285]. As a result, they must rely on simplified network models and perform poorly in real network conditions. We discuss these schemes further in §3.5.4 and §3.7.

eralize to videos with different properties, e.g., the number of encodings and their bitrates (§3.4).

To train its models, Pensieve uses simulations over a large corpus of network traces. Pensieve uses a fast and simple *chunk-level* simulator. While Pensieve could also train using packet-level simulations, emulations, or data collected from live video clients (§**??**), the chunk-level simulator is much faster and allows Pensieve to "experience" 100 hours of video downloads in only 10 minutes. We show that Pensieve's simulator faithfully models video streaming with live video players, provided that the transport stack is configured to achieve close to the true network capacity (§3.4.1).

We evaluate Pensieve using a full system implementation (§3.4.4). Our implementation deploys Pensieve's neural network model on an *ABR server*, which video clients query to get the bitrate to use for the next chunk; client requests include observations about throughput, buffer occupancy, and video properties. This design removes the burden of performing neural network computation on video clients, which may have limited computation power, e.g., TVs, mobile devices, etc. (§**??**).

We compare Pensieve to state-of-the-art ABR algorithms using a broad set of network conditions (both with trace-based emulation and in the wild) and QoE metrics (§3.5.2). We find that in all considered scenarios, Pensieve rivals or outperforms the best existing scheme, with average QoE improvements ranging from 12%–25%. Additionally, our results show Pensieve's ability to generalize to unseen network conditions and video properties. For example, on both broadband and HSDPA networks, Pensieve was able to outperform all existing ABR algorithms by training solely with a synthetic dataset. Finally, we present results which highlight Pensieve's low overhead and lack of sensitivity to system parameters, e.g., in the neural network (§3.5.4).

3.2 Background

HTTP-based adaptive streaming (standardized as DASH [9]) is the predominant form of video delivery today. By transmitting video using HTTP, content providers are able to leverage existing CDN infrastructure and maintain simplified (stateless) backends. Further, HTTP is compatible with a multitude of client-side applications such as web browsers and mobile applications.

In DASH systems, videos are stored on servers as multiple chunks, each of which repre-

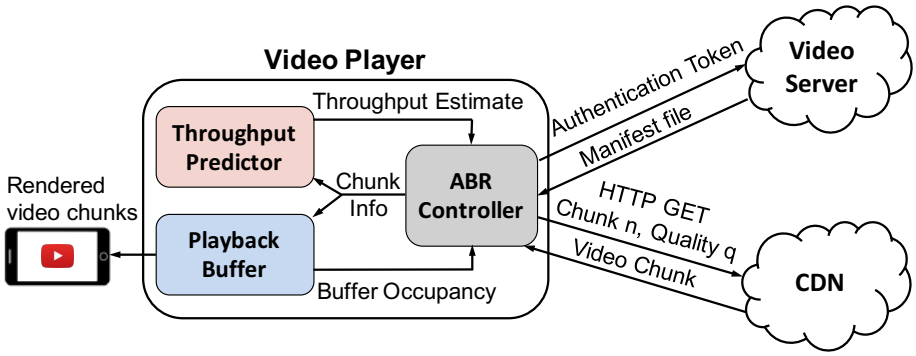


Figure 3-1: An overview of HTTP adaptive video streaming.

sents a few seconds of the overall video playback. Each chunk is encoded at several discrete bitrates, where a higher bitrate implies a higher quality and thus a larger chunk size. Chunks across bitrates are aligned to support seamless quality transitions, i.e., a video player can switch to a different bitrate at any chunk boundary without fetching redundant bits or skipping parts of the video.

Figure 3-1 illustrates the end-to-end process of streaming a video over HTTP today. As shown, a player embedded in a client application first sends a token to a video service provider for authentication. The provider responds with a manifest file that directs the client to a CDN hosting the video and lists the available bitrates for the video. The client then requests video chunks one by one, using an *adaptive bitrate (ABR) algorithm*. These algorithms use a variety of different inputs (e.g., playback buffer occupancy, throughput measurements, etc.) to select the bitrate for future chunks. As chunks are downloaded, they are played back to the client; note that playback of a given chunk cannot begin until the entire chunk has been downloaded.

Challenges: The policies employed by ABR algorithms heavily influence video streaming performance. However, these algorithms face four primary practical challenges:

1. Network conditions can fluctuate over time and can vary significantly across environments. This complicates bitrate selection as different scenarios may require different weights for input signals. For example, on time-varying cellular links, throughput prediction is often inaccurate and cannot account for sudden fluctuations in network bandwidth—inaccurate predictions can lead to underutilized networks (lower video quality) or inflated download delays (rebuffering). To overcome this, ABR algorithms must prioritize more stable input signals like buffer occupancy in these scenarios.
2. ABR algorithms must balance a variety of QoE goals such as maximizing video qual-

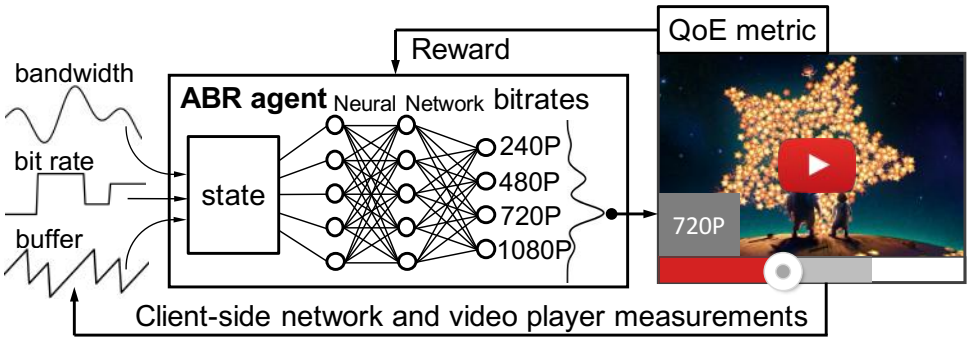


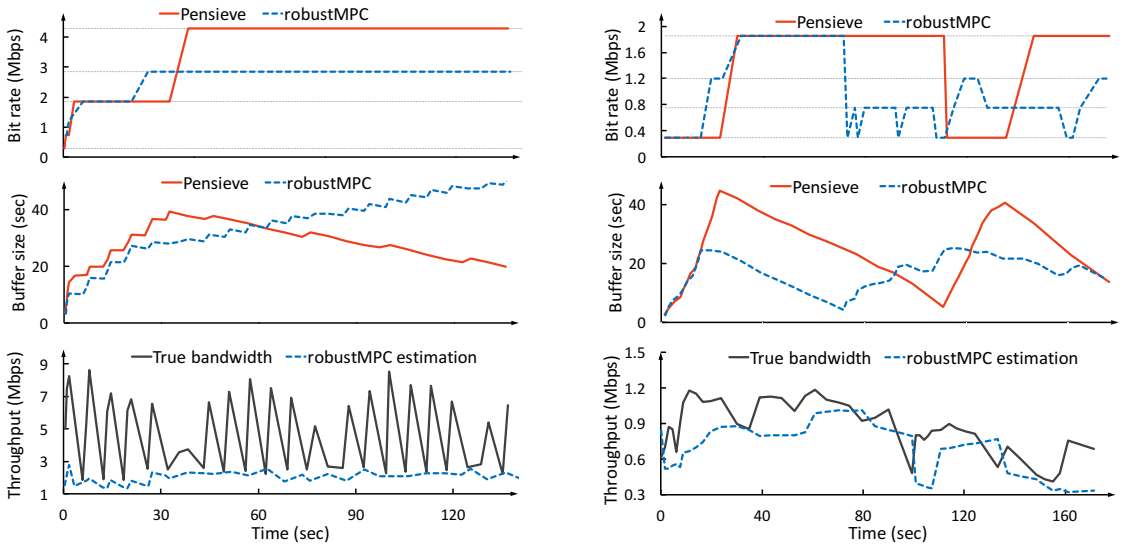
Figure 3-2: Applying reinforcement learning to bitrate adaptation.

ity (i.e., highest average bitrate), minimizing rebuffering events (i.e., scenarios where the client's playback buffer is empty), and maintaining video quality smoothness (i.e., avoiding constant bitrate fluctuations). However, many of these goals are inherently conflicting [10, 145, 157]. For example, on networks with limited bandwidth, consistently requesting chunks encoded at the highest possible bitrate will maximize quality, but may increase rebuffer rates. Conversely, on varying networks, choosing the highest bitrate that the network can support at any time could lead to substantial quality fluctuation, and hence degraded smoothness. To further complicate matters, preferences among these QoE factors vary significantly across users [168, 222, 221, 246].

3. Bitrate selection for a given chunk can have cascading effects on the state of the video player. For example, selecting a high bitrate may deplete the playback buffer and force subsequent chunks to be downloaded at low bitrates (to avoid rebuffering). Additionally, a given bitrate selection will directly influence the next decision when smoothness is considered—ABR algorithms will be less inclined to change bitrates.
4. The control decisions available to ABR algorithms are coarse-grained as they are limited to the available bitrates for a given video. Thus, there exist scenarios where the estimated throughput falls just below one bitrate, but well above the next available bitrate. In these cases, the ABR algorithm must decide whether to prioritize higher quality or the risk of rebuffering.

3.3 Learning ABR Algorithms

In this thesis, we consider a learning-based approach to generating ABR algorithms. Unlike approaches which use preset rules in the form of fine-tuned heuristics, our techniques



(a) Synthetic network. **(b)** HSDPA network.

Figure 3-3: Profiling bitrate selections, buffer occupancy, and throughput estimates with robustMPC [329] and Pensieve.

attempt to learn an ABR policy from observations. Specifically, our approach is based on reinforcement learning (RL). RL considers a general setting in which an *agent* interacts with an *environment*. At each time step t, the agent observes some *state* s_t, and chooses an *action* a_t. After applying the action, the state of the environment transitions to s_{t+1} and the agent receives a *reward* r_t. The goal of learning is to maximize the expected cumulative discounted reward: $\mathbb{E}[\sum_{t=0}^{\infty}\gamma^t r_t]$, where $\gamma \in (0,1]$ is a factor discounting future rewards.

Figure 3-2 summarizes how RL can be applied to bitrate adaptation. As shown, the decision policy guiding the ABR algorithm is not handcrafted. Instead, it is derived from training a neural network. The ABR agent observes a set of metrics including the client playback buffer occupancy, past bitrate decisions, and several raw network signals (e.g., throughput measurements) and feeds these values to the neural network, which outputs the action, i.e., the bitrate to use for the next chunk. The resulting QoE is then observed and passed back to the ABR agent as a reward. The agent uses the reward information to train and improve its neural network model. More details about the specific training algorithms we used are provided in §3.4.2.

To motivate learning-based ABR algorithms, we now provide two examples where existing techniques that rely on fixed heuristics can perform poorly. We choose these examples for illustrative purposes. We do not claim that they are indicative of the performance gains with learning in realistic network scenarios. We perform thorough quantitative evaluations

comparing learning-generated ABR algorithms to existing schemes in §3.5.2.

In these examples, we compare RL-generated ABR algorithms to MPC [329]. MPC uses both throughput estimates and observations about buffer occupancy to select bitrates that maximize a given QoE metric across a future chunk horizon. Here we consider robustMPC, a version of MPC that is configured to use a horizon of 5 chunks, and a conservative throughput estimate which normalizes the default throughput prediction with the max prediction error over the past 5 chunks. As the MPC paper shows, and our results validate, robustMPC's conservative throughput prediction significantly improves performance over default MPC, and achieves a high level of performance in most cases (§3.5.2). However, heuristics like robustMPC's throughput prediction require careful tuning and can backfire when their design assumptions are violated.

Example 1: The first example considers a scenario in which the network throughput is highly variable. Figure 3-3a compares the network throughput specified by the input trace with the throughput estimates used by robustMPC. As shown, robustMPC's estimates are overly cautious, hovering around 2 Mbps instead of the average network throughput of roughly 4.5 Mbps. These inaccurate throughput predictions prevent robustMPC from reaching high bitrates even though the occupancy of the playback buffer continually increases. In contrast, the RL-generated algorithm is able to properly assess the high average throughput (despite fluctuations) and switch to the highest available bitrate once it has enough cushion in the playback buffer. The RL-generated algorithm considered here was trained on a large corpus of real network traces (§3.5.1), not the synthetic trace in this experiment. Yet, it was able to make the appropriate decision.

Example 2: In our second example, both robustMPC and the RL-generated ABR algorithm optimize for a new QoE metric which is geared towards users who strongly prefer HD video. This metric assigns high reward to HD bitrates and low reward to all other bitrates (details in Table 3.1), while still favoring smoothness and penalizing rebuffering. To optimize for this metric, an ABR algorithm should attempt to build the client's playback buffer to a high enough level such that the player can switch up to and maintain an HD bitrate level. Using this approach, the video player can maximize the amount of time spent streaming HD video, while minimizing rebuffering time and bitrate transitions. However, performing well in this scenario requires long term planning since at any given instant, the penalty of selecting a higher bitrate (HD or not) may be incurred many chunks in the future when the buffer cannot support multiple HD downloads.

Figure 3-3b illustrates the bitrate selections made by each of these algorithms, and the effects that these decisions have on the playback buffer. Note that robustMPC and the RL-generated algorithm were both configured to optimize for this new QoE metric. As shown, robustMPC is unable to apply the aforementioned policy. Instead, robustMPC maintains a medium-sized playback buffer and requests chunks at bitrates that fall between the lowest level (300 kbps) and the lowest HD level (1850 kbps). The reason is that, despite being tuned to consider a horizon of future chunks at every step, robustMPC fails to plan far enough into the future. In contrast, the RL-generated ABR algorithm is able to actively implement the policy outlined above. It quickly grows the playback buffer by requesting chunks at 300 kbps, and then immediately jumps to the HD quality of 1850 kbps; it is able to then maintain this level for nearly 80 seconds, thereby ensuring quality smoothness.

Summary: robustMPC has difficulty (1) factoring throughput fluctuations and prediction errors into its decisions, and (2) choosing the appropriate optimization horizon. These deficiencies exist because MPC lacks an accurate model of network dynamics—thus it relies on simple and sub-optimal heuristics such as conservative throughput predictions and a small optimization horizon. More generally, any ABR algorithm that relies on fixed heuristics or simplified system models suffers from these limitations. By contrast, RL-generated algorithms learn from actual performance resulting from different decisions. By incorporating this information into a flexible neural network policy, RL-generated ABR algorithms can automatically optimize for different network characteristics and QoE objectives.

3.4 Design

In this section, we describe the design and implementation of Pensieve, a system that generates RL-based ABR algorithms and applies them to video streaming sessions. We start by explaining the training methodology (§3.4.1) and algorithms (§3.4.2) underlying Pensieve. We then describe an enhancement to the basic training algorithm, which enables Pensieve to support different videos using a single model (§3.4.3). Finally, we explain the implementation details of Pensieve and how it applies learned models to real streaming sessions (§3.4.4).

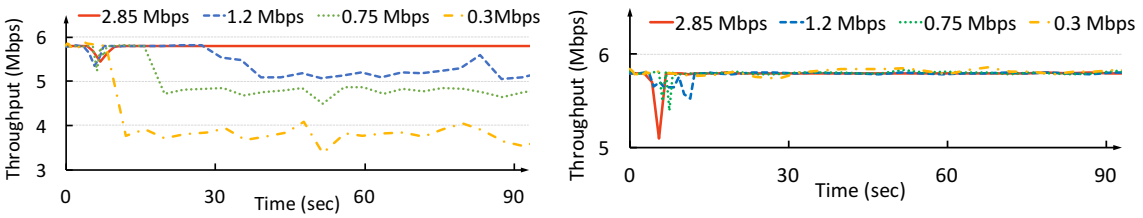
3.4.1 Training Methodology

The first step of Pensieve is to generate an ABR algorithm using RL (§3.3). To do this, Pensieve runs a training phase in which the learning agent explores a video streaming environment. Ideally, training would occur using actual video streaming clients. However, emulating the standard video streaming environment entails using a web browser to continually download video chunks. This approach is slow, as the training algorithm must wait until all of the chunks in a video are completely downloaded before updating its model.

To accelerate this process, Pensieve trains ABR algorithms in a simple simulation environment that faithfully models the dynamics of video streaming with real client applications. Pensieve's simulator maintains an internal representation of the client's playback buffer. For each chunk download, the simulator assigns a download time that is solely based on the chunk's bitrate and the input network throughput traces. The simulator then drains the playback buffer by the current chunk's download time, to represent video playback during the download, and adds the playback duration of the downloaded chunk to the buffer. The simulator carefully keeps track of rebuffering events that arise as the buffer occupancy changes, i.e., scenarios where the chunk download time exceeds the buffer occupancy at the start of the download. In scenarios where the playback buffer cannot accommodate video from an additional chunk download, Pensieve's simulator pauses requests for 500 ms before retrying.[3] After each chunk download, the simulator passes several state observations to the RL agent for processing: the current buffer occupancy, rebuffering time, chunk download time, size of the next chunk (at all bitrates), and the number of remaining chunks in the video. We describe how this input is used by the RL agent in more detail in §3.4.2. Using this chunk-level simulator, Pensieve can "experience" 100 hours of video downloads in only 10 minutes.

Though modeling the application layer semantics of client video players is straightforward, faithful simulation is complicated by intricacies at the transport layer. Specifically, video players may not request future chunks as soon as a chunk download completes, e.g., because the playback buffer is full. Such delays can trigger the underlying TCP connection to revert to slow start, a behavior known as *slow-start-restart* [19]. Slow start may in turn prevent the video player from fully using the available bandwidth, particularly for small chunk sizes (low bitrates). This behavior makes simulation challenging as it inherently ties network throughput to the ABR algorithm being used, e.g., schemes that fill buffers quickly will experience more slow start phases and thus less network utilization.

[3] This is the default request retry rate used by DASH players [9].



(a) TCP slow start restart enabled

(b) TCP slow start restart disabled

Figure 3-4: Profiling the throughput usage per-chunk of commodity video players with and without TCP slow start restart.

To verify this behavior, we loaded the test video described in §3.5.1 over an emulated 6 Mbps link using four ABR algorithms, each of which continually requests chunks at a single bitrate. We loaded the video with each scheme twice, both with slow-start-restart enabled and disabled.[4] Figure 3-4 shows the throughput usage *during chunk downloads* for each bitrate in both scenarios. As shown, with slow-start-restart enabled, the throughput depends on the bitrate of the chunk; ABR algorithms using lower bitrates (smaller chunk sizes) achieve less throughput per chunk. However, throughput is consistent and matches the available bandwidth (6 Mbps) for different bitrates if we disable slow-start-restart.

Pensieve's simulator assumes that the throughput specified by the trace is entirely used by each chunk download. As the above results show, this can be achieved by disabling slow-start-restart on the video server. Disabling slow-start-restart could increase traffic burstiness, but recent standards efforts are tackling the same problem for video streaming more gracefully by pacing the initial burst from TCP following an idle period [96, 132].

While it is possible to use a more accurate simulator (e.g., packet-level) to train Pensieve, in the end, no simulation can capture all real world system artifacts with 100% accuracy. However, we find that Pensieve can learn very high quality ABR algorithms (§3.5.2) using imperfect simulations, as long as it experiences a large enough variety of network conditions during training. This is a consequence of Pensieve's strong generalization ability (§3.5.3).

3.4.2 Basic Training Algorithm

We now describe our training algorithms. As shown in Figure 3-5, Pensieve's training algorithm uses A3C [219], a state-of-the-art actor-critic method which involves training two neural networks. The detailed functionalities of these networks are explained below.

[4] In Linux, the `net.ipv4.tcp_slow_start_after_idle` parameter can be used to set this configuration.

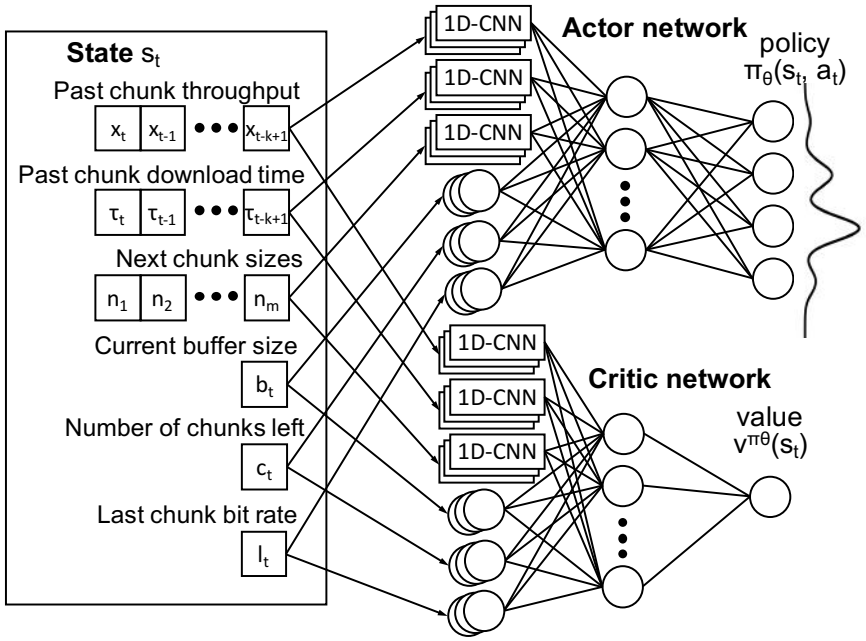


Figure 3-5: The Actor-Critic algorithm that Pensieve uses to generate ABR policies (described in §3.4.4).

Inputs: After the download of each chunk t, Pensieve's learning agent takes state inputs $s_t = (\vec{x_t}, \vec{\tau_t}, \vec{n_t}, b_t, c_t, l_t)$ to its neural networks. $\vec{x_t}$ is the network throughput measurements for the past k video chunks; $\vec{\tau_t}$ is the download time of the past k video chunks, which represents the time interval of the throughput measurements; $\vec{n_t}$ is a vector of m available sizes for the next video chunk; b_t is the current buffer level; c_t is the number of chunks remaining in the video; and l_t is the bitrate at which the last chunk was downloaded.

Policy: Upon receiving s_t, Pensieve's RL agent needs to take an action a_t that corresponds to the bitrate for the next video chunk. The agent selects actions based on a *policy*, defined as a probability distribution over actions $\pi : \pi(s_t, a_t) \rightarrow [0,1]$. $\pi(s_t, a_t)$ is the probability that action a_t is taken in state s_t. In practice, there are intractably many {state, action} pairs, e.g., throughput estimates and buffer occupancies are continuous real numbers. To overcome this, Pensieve uses a neural network (NN) [130] to represent the policy with a manageable number of adjustable parameters, θ, which we refer to as policy parameters. Using θ, we can represent the policy as $\pi_\theta(s_t, a_t)$. NNs have recently been applied successfully to solve large-scale RL tasks [220, 277, 201]. An advantage of NNs is that they do not need hand-crafted features and can be applied directly to "raw" observation signals. The *actor network* in Figure 3-5 depicts how Pensieve uses an NN to represent an ABR policy. We describe how we design the specific architecture of the NN in §3.5.3.

Policy gradient training: After applying each action, the simulated environment provides the learning agent with a reward r_t for that chunk. Recall from §3.3 that the primary goal of the RL agent is to maximize the expected cumulative (discounted) reward that it receives from the environment. Thus, the reward is set to reflect the performance of each chunk download according to the specific QoE metric we wish to optimize. See §3.5 for examples of QoE metrics.

The actor-critic algorithm used by Pensieve to train its policy is a *policy gradient method* [286]. We highlight the key steps of the algorithm, focusing on the intuition. The key idea in policy gradient methods is to estimate the gradient of the expected total reward by observing the trajectories of executions obtained by following the policy. The gradient of the cumulative discounted reward with respect to the policy parameters, θ, can be computed as [219]:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)\right]. \tag{3.1}$$

$A^{\pi_\theta}(s,a)$ is the *advantage* function, which represents the difference in the expected total reward when we *deterministically* pick action a in state s, compared with the expected reward for actions drawn from policy π_θ. The advantage function encodes how much better a specific action is compared to the "average action" taken according to the policy.

In practice, the agent samples a trajectory of bitrate decisions and uses the empirically computed advantage $A(s_t,a_t)$, as an unbiased estimate of $A^{\pi_\theta}(s_t,a_t)$. Each update of the actor network parameter θ follows the policy gradient,

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t,a_t) A(s_t,a_t), \tag{3.2}$$

where α is the learning rate. The intuition behind this update rule is as follows. The direction $\nabla_\theta \log \pi_\theta(s_t,a_t)$ specifies how to change the policy parameters in order to increase $\pi_\theta(s_t,a_t)$ (i.e., the probability of action a_t at state s_t). Equation 3.2 takes a step in this direction. The size of the step depends on the value of the advantage for action a_t in state s_t. Thus, the net effect is to reinforce actions that empirically lead to better returns.

To compute the advantage $A(s_t,a_t)$ for a given experience, we need an estimate of the *value function*, $v^{\pi_\theta}(s)$—the expected total reward starting at state s and following the policy π_θ. The role of the *critic network* in Figure 3-5 is to learn an estimate of $v^{\pi_\theta}(s)$ from empirically observed rewards. We follow the standard *Temporal Difference* method [285] to train

the critic network parameters θ_v,

$$\theta_v \leftarrow \theta_v - \alpha' \sum_t \nabla_{\theta_v} \left(r_t + \gamma V^{\pi_\theta}(s_{t+1};\theta_v) - V^{\pi_\theta}(s_t;\theta_v) \right)^2, \tag{3.3}$$

where $V^{\pi_\theta}(\cdot;\theta_v)$ is the estimate of $v^{\pi_\theta}(\cdot)$, output by the critic network, and α' is the learning rate for the critic. For an experience (s_t, a_t, r_t, s_{t+1}) (i.e., take action a_t in state s_t, receive reward r_t, and transition to s_{t+1}), the advantage $A(s_t, a_t)$ can now be estimated as $r_t + \gamma V^{\pi_\theta}(s_{t+1};\theta_v) - V^{\pi_\theta}(s_t;\theta_v)$. See [172] for more details.

It is important to note that the critic network merely helps to train the actor network. Post-training, only the actor network is required to execute the ABR algorithm and make bitrate decisions.

Finally, we must ensure that the RL agent explores the action space adequately during training to discover good policies. One common practice to encourage exploration is to add an entropy regularization term to the actor's update rule [219]; this can be critical in helping the learning agent converge to a good policy [323]. Concretely, we modify Equation 3.2 to be,

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t,a_t) A(s_t,a_t) + \beta \nabla_\theta H(\pi_\theta(\cdot|s_t)), \tag{3.4}$$

where $H(\cdot)$ is the entropy of the policy (the probability distribution over actions) at each time step. This term encourages exploration by pushing θ in the direction of higher entropy. The parameter β is set to a large value at the start of training (to encourage exploration) and decreases over time to emphasize improving rewards (§3.4.4).

The detailed derivation and pseudocode can be found in [219] (§4 and Algorithm S3).

Parallel training: To further enhance and speed up training, Pensieve spawns multiple learning agents in parallel, as suggested by the A3C paper [219]. By default, Pensieve uses 16 parallel agents. Each learning agent is configured to experience a different set of input parameters (e.g., network traces). However, the agents continually send their {state, action, reward} tuples to a central agent, which aggregates them to generate a single ABR algorithm model. For each sequence of tuples that it receives, the central agent uses the actor-critic algorithm to compute a gradient and perform a gradient descent step (Equations (3.3) and (3.4)). The central agent then updates the actor network and pushes out the new model to the agent which sent that tuple. Note that this can happen asynchronously among all agents, i.e., there is no locking between agents [258].

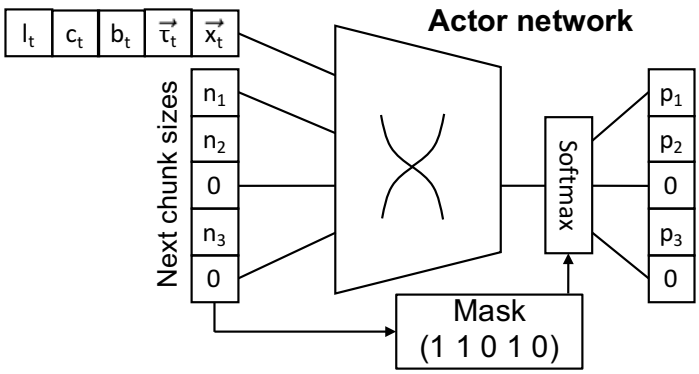


Figure 3-6: Modification to the state input and the softmax output to support multiple videos.

Choice of algorithm: A variety of different algorithms could be used to train the learning agent in the abstract RL framework described above (e.g., DQN [220], REINFORCE [286], etc.). In our design, we chose to use A3C [219] because (1) to the best of our knowledge, it is the state-of-art and it has been successfully applied to many other concrete learning problems [323, 306, 154]; and (2) in the video streaming application, the asynchronous parallel training framework supports online training in which many users concurrently send their experience feedback to the agent. We also compare Pensieve with previous tabular Q-learning schemes [63] in §3.5.4.

3.4.3 Enhancement for multiple videos

The basic algorithm described in §3.4.2 has some practical issues. The primary challenge is that videos can be encoded at different bitrate levels and may have diverse chunk sizes due to variable bitrate encoding [282], e.g., chunk sizes for 720p video are not identical across videos. Handling this variation would require each neural network to take a variable sized set of inputs and produce a variable sized set of outputs. The naive solution to supporting a broad range of videos is to train a model for each possible set of video properties. Unfortunately, this solution is not scalable. To overcome this, we describe two enhancements to the basic algorithm that enable Pensieve to generate a *single* model to handle multiple videos (Figure 3-6).

First, we pick canonical input and output formats that span the maximum number of bitrate levels we expect to see in practice. For example, a range of 13 levels covers the entire DASH reference client video list [75]. Then, to determine the input state for a specific video, we take the chunk sizes and map them to the index which has the closest bitrate. The remain-

ing input states, which pertain to the bitrates that the video does not support, are zeroed out. For example, in Figure 3-6, chunk sizes (n_1,n_2,n_3) are mapped to the corresponding indices, while the remaining input values are filled with zeroes.

The second change pertains to how the output of the actor network is interpreted. For a given video, we apply a mask to the output of the final softmax [47] layer in the actor network, such that the output probability distribution is only over the bitrates that the video actually supports. Formally, the mask is presented by a 0-1 vector $[m_1,m_2,...,m_k]$, and the modified softmax for the NN output $[z_1,z_2,...,z_k]$ will be

$$p_i = \frac{m_i e^{z_i}}{\sum_j m_j e^{z_j}}, \tag{3.5}$$

where p_i is the normalized probability for action i. With this modification, the output probabilities are still a continuous function of the network parameters. The reason is that the mask values $\{m_i\}$ are independent of the network parameters, and are only a function of the input video. As a result, the standard back-propagation of the gradient in the NN still holds and the training techniques established in §3.4.2 can be applied without modification. We evaluate the effectiveness of these modifications in more detail in §3.5.4.

3.4.4 Implementation

To generate ABR algorithms, Pensieve passes $k = 8$ past bandwidth measurements to a 1D convolution layer (CNN) with 128 filters, each of size 4 with stride 1. Next chunk sizes are passed to another 1D-CNN with the same shape. Results from these layers are then aggregated with other inputs in a hidden layer that uses 128 neurons to apply the softmax function (Figure 3-5). The critic network uses the same NN structure, but its final output is a linear neuron (with no activation function). During training, we use a discount factor $\gamma = 0.99$, which implies that current actions will be influenced by 100 future steps. The learning rates for the actor and critic are configured to be 10^{-4} and 10^{-3}, respectively. Additionally, the entropy factor β is controlled to decay from 1 to 0.1 over 10^5 iterations. We keep all these hyperparameters fixed throughout our experiments. While some tuning is useful, we found that Pensieve performs well for a wide range of hyperparameter values. Thus we did not use sophisticated hyperparameter tuning methods [118]. We implemented this architecture using TensorFlow [1]. For compatibility, we leveraged the TFLearn deep learning library's TensorFlow API [291] to declare the neural network during both training and testing.

Once Pensieve has generated an ABR algorithm using its simulator, it must apply the model's rules to real video streaming sessions. To do this, Pensieve runs on a standalone ABR server, implemented using the Python BaseHTTPServer. Client requests are modified to include additional information about the previous chunk download and the video being streamed (§3.4.2). By collecting information through client requests, Pensieve's server and ABR algorithm can remain stateless while still benefitting from observations that can solely be collected in client video players. As client requests for individual chunks arrive at the video server, Pensieve feeds the provided observations through its actor NN model and responds to the video client with the bitrate level to use for the next chunk download; the client then contacts the appropriate CDN to fetch the corresponding chunk. It is important to note that Pensieve's ABR algorithm could also operate directly inside video players. We evaluate the overhead that a server-side deployment has on video QoE in §3.5.4, and discuss other deployment models in more detail in §**??**.

3.5 Evaluation

In this section, we experimentally evaluate Pensieve. Our experiments cover a broad set of network conditions (both trace-based and in the wild) and QoE metrics. Our results answer the following questions:

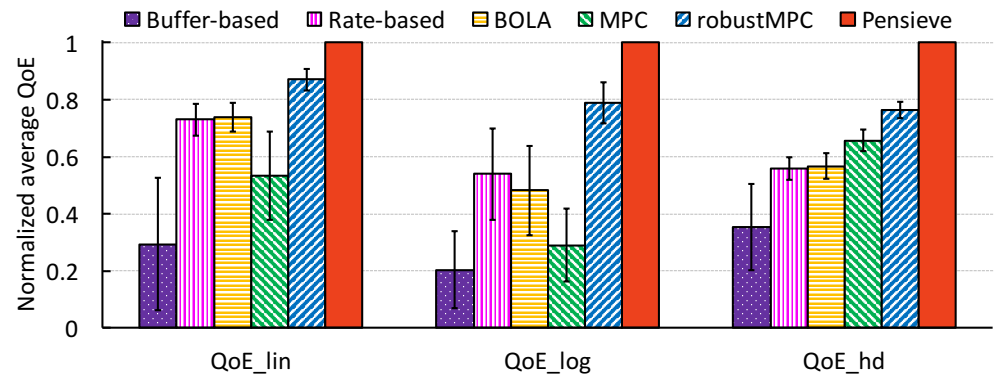
1. How does Pensieve compare to state-of-the-art ABR algorithms in terms of video QoE? We find that, in all of the considered scenarios, Pensieve is able to rival or outperform the best existing scheme, with average QoE improvements ranging from 13.1%–25.0% (§3.5.2); Figure 3-7 provides a summary.
2. Do the models learned with Pensieve generalize to new network conditions and videos? We find that Pensieve's ABR algorithms are able to maintain high levels of performance both in the presence of new network conditions and new video properties (§3.5.3).
3. How sensitive is Pensieve to various system parameters such as the neural network architecture and the latency between the video client and ABR server? Our experiments suggest that performance is largely unaffected by these parameters (Tables 3.2 and 3.3). For example, applying 100 ms RTT values between clients and the Pensieve server reduces average QoE by only 3.5% (§3.5.4).

3.5.1 Methodology

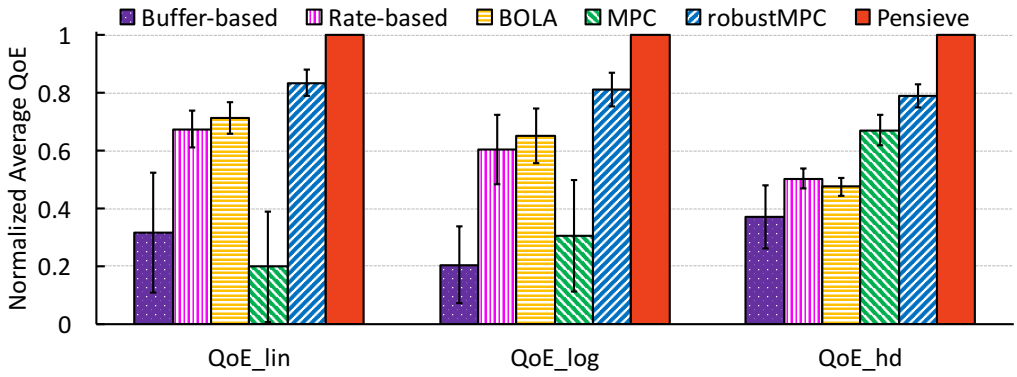
Network traces: To evaluate Pensieve and state-of-the-art ABR algorithms on realistic network conditions, we created a corpus of network traces by combining several public datasets: a broadband dataset provided by the FCC [97] and a 3G/HSDPA mobile dataset collected in Norway [259]. The FCC dataset contains over 1 million throughput traces, each of which logs the average throughput over 2100 seconds, at a 5 second granularity. We generated 1000 traces for our corpus, each with a duration of 320 seconds, by concatenating randomly selected traces from the "Web browsing" category in the August 2016 collection. The HSDPA dataset comprises 30 minutes of throughput measurements, generated using mobile devices that were streaming video while in transit (e.g., via bus, train, etc.). To match the duration of the FCC traces included in our corpus, we generated 1000 traces (each spanning 320 seconds) using a sliding window across the HSDPA dataset. To avoid scenarios where bitrate selection is trivial, i.e., situations where picking the maximum bitrate is always the optimal solution, or where the network cannot support any available bitrate for an extended period, we only considered original traces whose average throughput is less than 6 Mbps, and whose minimum throughput is above 0.2 Mbps. We reformatted throughput traces from both datasets to be compatible with the Mahimahi [230] network emulation tool. Unless otherwise noted, we used a random sample of 80% of our corpus as a training set for Pensieve; we used the remaining 20% as a test set for all ABR algorithms. All in all, our test set comprises of over 30 hours of network traces.

Adaptation algorithms: We compare Pensieve to the following algorithms which collectively represent the state-of-the-art in bitrate adaptation:

1. Buffer-Based (BB): mimics the buffer-based algorithm described by Huang et al. [146] which uses a reservoir of 5 seconds and a cushion of 10 seconds, i.e., it selects the highest bitrate that is predicted to keep the buffer occupancy above 5 seconds, and automatically chooses the highest available bitrate if the buffer occupancy exceeds 15 seconds.
2. Rate-Based (RB): predicts throughput using the harmonic mean of the experienced throughput for the past 5 chunk downloads. It then selects the highest available bitrate that is below the predicted throughput.
3. BOLA [282]: uses Lyapunov optimization to select bitrates solely considering buffer occupancy observations. We use the BOLA implementation in dash.js [9].
4. MPC [329]: uses buffer occupancy observations and throughput predictions (com-



(a) FCC broadband dataset



(b) Norway HSDPA dataset

Figure 3-7: Comparing Pensieve with existing ABR algorithms on broadband and 3G/HSDPA networks. The QoE metrics considered are presented in Table 3.1. Results are normalized against the performance of Pensieve. Error bars span $\pm$ one standard deviation from the average.

puted in the same way as RB) to select the bitrate which maximizes a given QoE metric over a horizon of 5 future chunks.

5. robustMPC [329]: uses the same approach as MPC, but accounts for errors seen between predicted and observed throughputs by normalizing throughput estimates by the max error seen in the past 5 chunks.

Note: MPC involves solving an optimization problem for each bitrate decision which maximizes the QoE metric over the next 5 video chunks. The MPC [329] paper describes a method, fastMPC, which precomputes the solution to this optimization problem for a quantized set of input values (e.g., buffer size, throughput prediction, etc.). Because the implementation of fastMPC is not publicly available, we implemented MPC using our ABR server as follows. For each bitrate decision, we solve the optimization problem exactly on the ABR server by enumerating all possibilities for the next 5 chunks. We found that the computation takes at most 27 ms for 6 bitrate levels and has negligible impact on QoE.

Name	bitrate utility ($q(R)$)	rebuffer penalty (μ)
QoE_{lin}	R	4.3
QoE_{log}	$\log(R/R_{min})$	2.66
QoE_{hd}	0.3→1, 0.75→2, 1.2→3 1.85→12, 2.85→15, 4.3→20	8

Table 3.1: The QoE metrics we consider in our evaluation. Each metric is a variant of Equation 3.6.

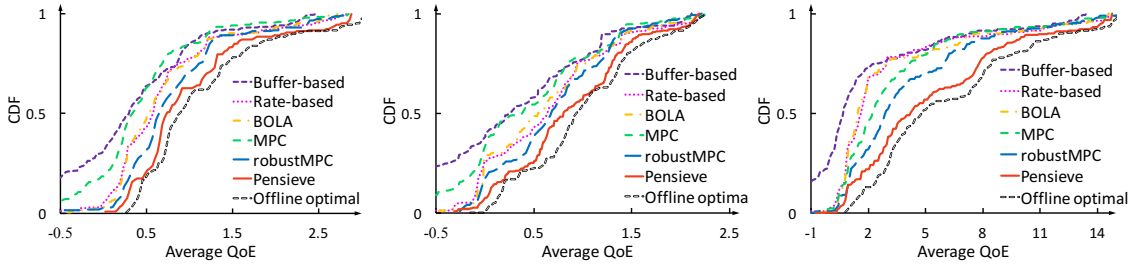
Experimental setup: We modified dash.js (version 2.4) [9] to support each of the aforementioned state-of-the-art ABR algorithms. For Pensieve and both variants of MPC, dash.js was configured to fetch bitrate selection decisions from an ABR server that implemented the corresponding algorithm. ABR servers ran on the same machine as the client, and requests to these servers were made using `XMLHttpRequests`. All other algorithms ran directly in dash.js. The DASH player was configured to have a playback buffer capacity of 60 seconds. Our evaluations used the "EnvivioDash3" video from the DASH-246 JavaScript reference client [75]. This video is encoded by the H.264/MPEG-4 codec at bitrates in $\{300, 750, 1200, 1850, 2850, 4300\}$ kbps (which pertain to video modes in $\{240,360,480,720,1080,1440\}$p). Additionally, the video was divided into 48 chunks and had a total length of 193 seconds. Thus, each chunk represented approximately 4 seconds of video playback. In our setup, the client video player was a Google Chrome browser (version 53) and the video server (Apache version 2.4.7) ran on the same machine as the client. We used Mahimahi [230] to emulate the network conditions from our corpus of network traces, along with an 80 ms RTT, between the client and server. Unless otherwise noted, all experiments were performed on Amazon EC2 t2.2xlarge instances.

QoE metrics: There exists significant variance in user preferences for video streaming QoE [168, 222, 221, 246]. Thus, we consider a variety of QoE metrics. We start with the general QoE metric used by MPC [329], which is defined as

$$QoE = \sum_{n=1}^{N} q(R_n) - \mu \sum_{n=1}^{N} T_n - \sum_{n=1}^{N-1} \left| q(R_{n+1}) - q(R_n) \right| \tag{3.6}$$

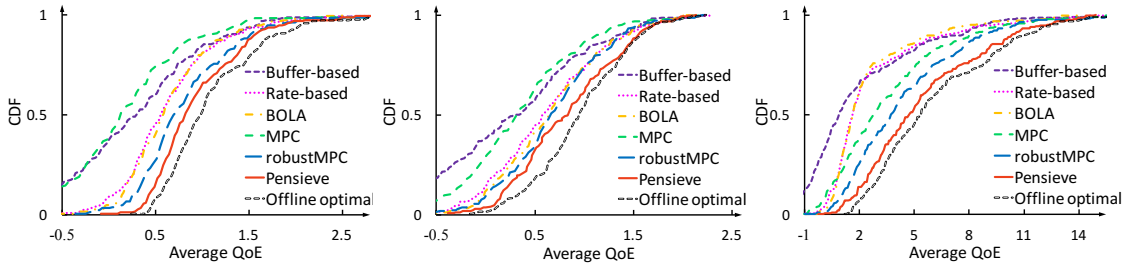
for a video with N chunks. R_n represents the bitrate of $chunk_n$ and $q(R_n)$ maps that bitrate to the quality perceived by a user. T_n represents the rebuffering time that results from downloading $chunk_n$ at bitrate R_n, while the final term penalizes changes in video quality to favor smoothness.

We consider three choices of $q(R_n)$:



(a) QoE_{lin} **(b)** QoE_{log} **(c)** QoE_{hd}

Figure 3-8: Comparing Pensieve with existing ABR algorithms on the QoE metrics listed in Table 3.1. Results were collected on the FCC broadband dataset. Average QoE values are listed for each ABR algorithm.



(a) QoE_{lin} **(b)** QoE_{log} **(c)** QoE_{hd}

Figure 3-9: Comparing Pensieve with existing ABR algorithms on the QoE metrics listed in Table 3.1. Results were collected on the Norway HSDPA dataset. Average QoE values are listed for each ABR algorithm.

1. QoE_{lin}: $q(R_n) = R_n$. This metric was used by MPC [329].
2. QoE_{log}: $q(R_n) = \log(R/R_{min})$. This metric captures the notion that, for some users, the marginal improvement in perceived quality decreases at higher bitrates. This metric was used by BOLA [282].
3. QoE_{hd}: This metric favors High Definition (HD) video. It assigns a low quality score to non-HD bitrates and a high quality score to HD bitrates.

The exact values of $q(R_n)$ for our baseline video are provided in Table 3.1. In this section, we report the average QoE per chunk, i.e., the total QoE metric divided by the number of chunks in the video.

3.5.2 Pensieve vs. Existing ABR algorithms

To evaluate Pensieve, we compared it with state-of-the-art ABR algorithms on each QoE metric listed in Table 3.1. In each experiment, Pensieve's ABR algorithm was trained to optimize

for the considered QoE metric, using the entire training corpus described in §3.5.1; both MPC variants were also modified to optimize for the considered QoE metric. Figure 3-7 shows the average QoE that each scheme achieves on our entire test corpus. Figures 3-8 and 3-9 provide more detailed results in the form of full CDFs for each network. As a comparison, we compute the offline[5] optimal using dynamic programing with future throughput information.

There are two key takeaways from these results. First, we find that Pensieve either matches or exceeds the performance of the best existing ABR algorithm on each QoE metric and network considered. The closest competing scheme is robustMPC; this shows the importance of tuning, as without robustMPC's conservative throughput estimates, MPC can become too aggressive (relying on the playback buffer) and perform worse than even a naive rate-based scheme. For QoE_{lin}, which was considered in the MPC paper [329], the average QoE for Pensieve is 13.1% higher than robustMPC on the FCC broadband network traces. The gap between Pensieve and robustMPC widens to 18.5% and 30.4% for QoE_{log} and QoE_{hd}. The results are qualitatively similar for the Norway HSDPA network traces.

Second, we observe that the performance of existing ABR algorithms is sensitive to different QoE objectives. The reason is that these algorithms employ fixed control laws, even though optimizing for different QoE objectives requires inherently different ABR strategies. For example, unlike QoE_{lin}, the optimal strategy for QoE_{log} is to make small increases in bitrate since the marginal improvement in user-perceived quality diminishes at higher bitrates. With this strategy, video players avoid jumping to high bitrate levels when the risk of rebuffering is high. However, to optimize for QoE_{lin}, the ABR algorithm needs to be more aggressive. Pensieve is able to automatically learn these policies (without explicit tuning) and thus, performance with Pensieve remains consistently high as conditions change.

The results for QoE_{hd} further illustrate this point. Recall that QoE_{hd} favors HD video, assigning the highest utility to the top three bitrates available for our test video (see Table 3.1). As discussed in §3.3, optimizing for QoE_{hd} requires significantly more long-term planning than the other two QoE metrics. When network bandwidth is inadequate, the ABR algorithm should build the playback buffer as quickly as possible using the lowest available bitrate. Once the buffer is large enough, it should then make a direct transition to the lowest HD quality (bypassing intermediate bitrates). However, building buffers to a level which circumvents rebuffering and maintains sufficient smoothness requires a lot of foresight. As

[5]Notice that the offline optimal is not realistic as it has full knowledge of the future. It only serves as an upper bound of QoE obtained by any possible sequence of decisions. In §3.5.4, we perform detailed analysis of the practical optimality gap with an online optimal scheme.

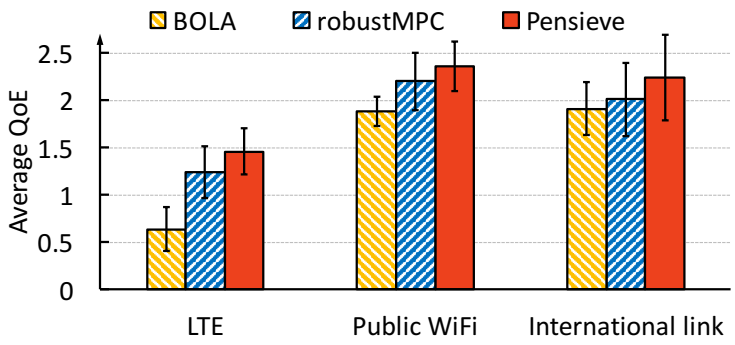


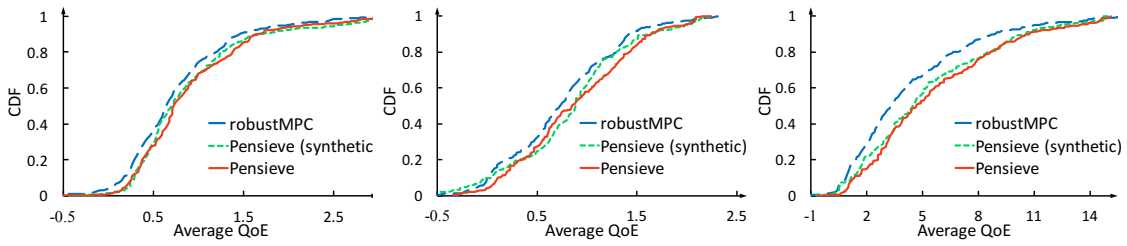
Figure 3-10: Comparing Pensieve with existing ABR algorithms in the wild on the QoE_{hd} metric. Results were collected using a public WiFi network and the Verizon LTE cellular network. Bars list averages and error bars span $\pm$ one standard deviation from the average.

illustrated by the example in Figure 3-3b, Pensieve is able to learn such a policy, while robustMPC's conservative throughput predictions and 5 chunk horizon prevent it from doing so. It may be possible to tune robustMPC to better cater to QoE_{hd}, e.g., by increasing the horizon length and reducing the conservatism of the throughput predictor. However, such tweaks may not perform well on other QoE metrics. In contrast, Pensieve learns a good ABR policy purely from experience, with zero tuning or designer involvement.

3.5.3 Generalization

In the experiments above, Pensieve was trained with a set of traces collected on the same networks that were used during testing; note that no test traces were directly included in the training set. However, in practice, Pensieve's ABR algorithms could encounter new networks, with different conditions (and thus, with different optimal strategies). To evaluate Pensieve's ability to generalize to new network conditions, we conduct two experiments. First, we evaluate Pensieve in the wild on two real networks. Second, we show how Pensieve can be trained to perform well across multiple environments using a purely synthetic dataset.

Real world experiments: We evaluated Pensieve and several state-of-the-art ABR algorithms in the wild using two networks: a public WiFi network at a local coffee shop, and the Verizon LTE cellular network. In these experiments, a client, running on a Macbook Pro laptop, contacted a video server running on a nearby desktop machine. We considered a subset of the ABR algorithms listed in §3.5.1: BOLA, robustMPC, and Pensieve. On each network, we loaded our test video five times with each scheme, randomly selecting the order among them. The Pensieve ABR algorithm evaluated here was solely trained using the broadband



(a) QoE_{lin} **(b)** QoE_{log} **(c)** QoE_{hd}

Figure 3-11: Comparing two ABR algorithms with Pensieve on the broadband and HSDPA networks: one algorithm was trained on synthetic network traces, while the other was trained using a set of traces directly from the broadband and HSDPA networks. Results are aggregated across the two datasets.

and HSDPA traces in our corpus. However, even on these new networks, Pensieve was able to outperform the existing ABR algorithms on the QoE_{hd} metric (Figure 3-10). The reason is the same as described above: because existing metrics were not manually tuned for QoE_{hd}, they fail to plan far enough into the future, requesting chunks at bitrates just below HD quality. In contrast, Pensieve's ABR algorithm automatically learned to generalize the strategy it developed for QoE_{hd} on the training networks to the new networks seen in the wild.

Training with a synthetic dataset: Can we train Pensieve without *any* real network data? Learning from synthetic data alone would of course be undesirable, but we use it as a challenging test of Pensieve's ability to generalize.

We design a data set to cover a relatively broad set of network conditions, with average throughputs ranging from 0.2 Mbps to 4.3 Mbps. Specifically, the dataset was generated using a Markovian model in which each state represented an average throughput in the aforementioned range. State transitions were performed at a 1 second granularity and followed a geometric distribution (making it more likely to transition to a nearby average throughput). Each throughput value was then drawn from a Gaussian distribution centered around the average throughput for the current state, with variance uniformly distributed between 0.05 and 0.5.

We then used Pensieve to compare two ABR algorithms on the test dataset described above (i.e., a combination of the HSDPA and broadband datasets): one trained solely using the synthetic dataset, and another trained explicitly on broadband and HSDPA network traces. Figure 3-11 illustrates our results for all three QoE metrics listed in Table 3.1. As shown, Pensieve's ABR algorithm that was trained on the synthetic dataset is able to generalize across these new networks, outperforming robustMPC and achieving average QoE

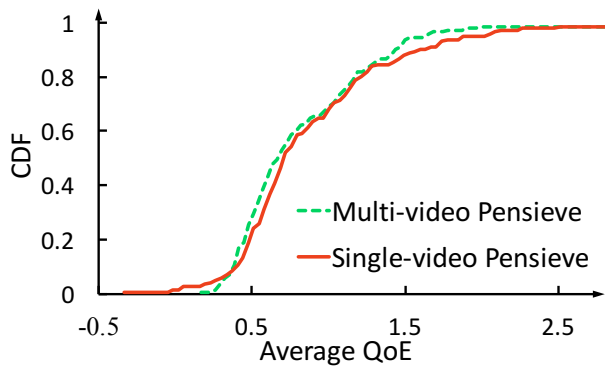


Figure 3-12: Comparing ABR algorithms trained across multiple videos with those trained explicitly on the test video. The measuring metric is QoE_{lin}.

values within 1.4%–11.9% of the ABR algorithm trained directly on the test networks. These results suggest that, in practice, Pensieve will likely be able to generalize to a broad range of network conditions encountered by its clients.

Multiple videos: As a final test of generalization, we evaluated Pensieve's ability to generalize across multiple video properties. To do this, we trained a single ABR algorithm on 1,000 synthetic videos using the techniques described in §3.4.3. Specifically, the number of bitrate is randomly selected from $[3,10]$ levels. The bitrates are then randomly chosen from $\{200,300,450,750,1200,1850,2350,2850,3500,4300\}$ kbps. The number of video chunks is randomly generated from $[20,100]$ chunks and the actual file size of each 4-second video chunk is multiplied with a Gaussian noise $\sim\mathcal{N}(0,0.1)$ to synthesize the variation of file size. Thus, these videos diverge on numerous properties including the bitrate options (both the number of options and value of each), number of chunks, chunk sizes and video duration. Additionally, none of the generated training videos overlaps the testing video on the bitrates. Unsurprisingly, the number of available bitrates for these videos represent the two ends of the spectrum for videos provided by the DASH reference client [75].

We compare this newly trained model to the original model, which is trained solely on the "EnvivioDash3" video described in §3.5.1. Our results measure QoE_{lin} on broadband and HSDPA network traces and are depicted in Figure 3-12. As shown, the generalized ABR algorithm from multi-video model is able to achieve average QoE_{lin} values within 3.22% of models trained explicitly on the test video. These results suggest that in practice, Pensieve servers can be configured to use a small number of ABR algorithms to improve streaming for a diverse set of videos.

Number of neurons and filters (each)	Average QoE_{hd}
4	3.850 ± 1.215
16	4.681 ± 1.369
32	5.106 ± 1.452
64	5.496 ± 1.411
128	5.489 ± 1.378

Table 3.2: Sweeping the number of CNN filters and hidden neurons in Pensieve's learning architecture.

Number of hidden layers	Average QoE_{hd}
1	5.489 ± 1.378
2	5.396 ± 1.434
5	4.253 ± 1.219

Table 3.3: Sweeping the number of hidden layers in Pensieve's learning architecture.

3.5.4 Pensieve Deep Dive

Pensieve's default implementation raises three practical concerns. How sensitive is Pensieve to the structure of its learning architecture? What is the overhead of training ABR algorithms and using the resulting models to guide client chunk downloads? What impact does the additional latency incurred by clients to retrieve bitrate suggestions from Pensieve's video servers have on client-perceived QoE? In this section, we describe fine-grained experiments that shed light on these challenges and explain the feasibility of using RL-generated ABR algorithms in real video streaming sessions. All experiments in this section used the experimental setup described in §3.5.1 and consider the QoE_{hd} metric.

Neural Network (NN) architecture: Starting with Pensieve's default learning architecture (Figure 3-5), we swept a range of NN parameters to understand the impact that each has on user-perceived QoE. First, using a fixed single hidden layer, we varied the number of filters in the 1D-CNN and the number of neurons in the hidden merge layer. These parameters were swept in tandem, i.e., when 4 filters were used, 4 neurons were used. Results from this sweep are presented in Table 3.2. As shown, performance begins to plateau once the number of filters and neurons each exceed 32. Additionally, notice that once these values reach 128 (Pensieve's default configuration), variance levels decrease while average QoE_{hd} values remain stable.

Next, after fixing the number of filters and hidden neurons to 128, we varied the number of hidden layers in Pensieve's architecture. The resulting QoE_{hd} values are listed in Table 3.3. Interestingly, we find that the shallowest network of 1 hidden layer yields the best

RTT (ms)	Average QoE_{hd}
0	5.407 ± 1.820
20	5.356 ± 1.768
40	5.309 ± 1.768
60	5.271 ± 1.773
80	5.217 ± 1.742
100	5.219 ± 1.748

Table 3.4: Average QoE_{hd} values when different RTT values are imposed between the client and Pensieve server.

performance; this represents the default value in Pensieve. Performance steadily degrades as we increase the number of hidden layers. However, it is important to note that our sweep used a fixed learning rate and number of training iterations. Tuning these parameters to cater to deeper networks may improve performance, as these networks generally take longer to train.

Training time: To measure the overhead of generating ABR algorithms using RL, we profiled Pensieve's training process. Training a single algorithm required approximately 50,000 iterations, where each iteration took 300 ms and corresponded to 16 agents updating their parameters in parallel (using the asynchronous training approach described in §3.4.2). Thus, in total, training took approximately 4 hours. We note that this cost is incurred offline and can be performed infrequently depending on environment stability.

Client-to-ABR server latency: Recall that with Pensieve, RL-generated ABR algorithms are applied to video streaming sessions by servers (not clients). Under this deployment model, clients must first query the Pensieve server to determine the bitrate to use for the next chunk, before downloading that chunk from a CDN server. To understand the overhead incurred by this additional round trip, we performed a sweep of the RTT between the client player and Pensieve server, considering values from 0 ms–100 ms. This experiment used the same setup described in §3.5.1, and measured the QoE_{hd} metric. Table 3.4 lists our results, highlighting that the latency from this additional RTT has minimal impact on QoE: the average QoE_{hd} with a 100 ms latency was within 3.5% of that when the latency was 0 ms. The reason is that the latency incurred from the additional round trip to Pensieve's server is masked by the playback buffer occupancy and chunk download times [145, 157].

Online and offline optimality: How is the performance of Pensieve compared with an optimal scheme? Notice that there still remains a sizable gap between Pensieve and *offline* optimal as shown in Figure 3-9 and 3-8. However, the offline optimal in §3.5.1 is obtained by running dynamic programming with the omniscient knowledge of the future bandwidth.

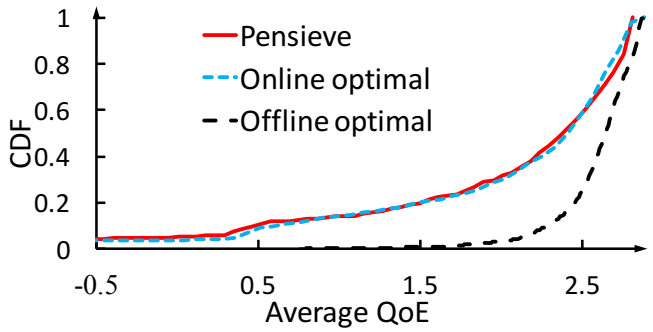


Figure 3-13: Comparing Pensieve with online and offline optimal with QoE_{lin} metric.

To reflect the realistic *online* optimal, it requires to know the underlining distribution of the future network throughput. Therefore, we conduct a controlled experiment where the chunk download time is generated following an known Markov process. Specifically, we simulate the download time T_n of chunk n as

$$T_n = T_{n-1}\frac{R_n}{R_{n-1}} + \epsilon, \tag{3.7}$$

where R_n is the bitrate of chunk n and ϵ is an additive Gaussian noise.

We can then write down the dynamic programming procedure for finding the online optimal decision,

$$\begin{aligned} QoE_n(B_n,T_n,R_n) = \max_{R_{n+1}} \Bigg\{ & q(R_{n+1}) - \mu \left[T_n \frac{R_{n+1}}{R_n} - B_n \right]_+ - \left| q(R_{n+1}) - q(R_n) \right| \\ & + \mathbb{E}_{T_{n+1}} \left[QoE_{n+1}(B_{n+1},T_{n+1},R_{n+1}) \right] \Bigg\}, \end{aligned} \tag{3.8}$$

$$B_{n+1} = \left[B_n - T_{n+1} \right]_+ + \delta, \tag{3.9}$$

$$T_{n+1} \sim \mathcal{N}\left(T_n \frac{R_{n+1}}{R_n}, \sigma^2 \right), \tag{3.10}$$

where B_n is the buffer occupancy right after chunk n is downloaded, which depends on the chunk download time T_n and the size of chunk δ. The chunk download time T_n yields a Gaussian distribution.

In our experiment, the evaluation metric follows QoE_{lin} in Table 3.1. The video chunk length δ is 4 seconds, using the setting of "EnvivioDash3" video described in §3.5.1. To generate network traces with similar range of the video bitrates, the initial download time T_0 is

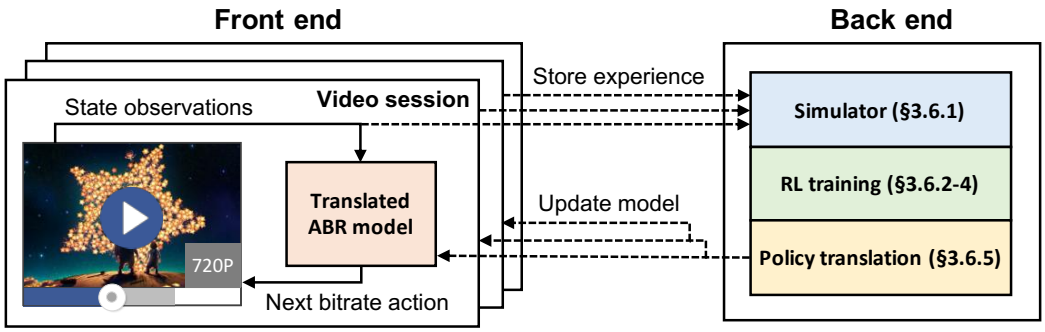


Figure 3-14: ABRL Design overview. For each video session in the production experiment, ABRL collects the experience of video watch time and the network bandwidth measurements and predictions. It then simulates the buffer dynamics of the video streaming using these experiences in the backend. After RL training, ABRL deploys the translated ABR model to the user front end.

set to 4 seconds for bitrate $R_0 = 2$kbps, and the standard deivation σ of the additive Guassian noise is configured to be 0.5. Additionally, Equations 3.9 and 3.10 are confined in $[0, 30]$ seconds to avoid unrealistic chunk download behavior (e.g., negative download time). The granularity of dynamic programming is 0.1 seconds for both buffer occupancy and download time.

We use the same setup in §3.5.1 to train a Pensieve agent in this simulated environment, and compare the performance with online and offline optimal. The results are depicted in Figure 3-13. Recall that the offline dynamic programing uses the *exact* future download time, whereas the online one only uses the corresponding *distribution*. Therefore, as expected, the offline optimal outperforms the online optimal by 9.1% on average, which is in the similar scale as the optimality gap observed in §3.5.2. Meanwhile, notice that the average QoE achieved by Pensieve is within 0.2% of the online optimal. This near-optimal performance implies that Pensieve is able to learn the underlining distribution of download time through experience, and it can learn an optimal online policy by interacting with the video streaming environment directly.

3.6 Real-World Deployment Study

In this section, we customize Pensieve, codenamed ABRL, to perform a deployment study at Facebook's web-based video streaming platform. Real-world ABR contains several challenges that require techniques beyond those in Pensieve — we implement a scalable neural network architecture that supports videos with arbitrary bitrate encodings (§3.6.2); we de-

sign a training method to cope with the variance resulting from the stochasticity in network conditions (§3.6.3); we leverage constrained Bayesian optimization for reward shaping in order to optimize the conflicting QoE objectives (§3.6.4); and we translate the learned ABR policy to deploy in the front end (§3.6.5). Figure 3-14 shows an overview. In a week-long worldwide deployment with more than 30 million video streaming sessions, our RL approach outperforms the existing human-engineered ABR algorithms in production.

3.6.1 Simulator

To train the ABR agent with RL, we first build a simulator that models the playback buffer dynamics during video streaming, similar to §3.4.1. The simulator uses sampled traces collected from the actual video playback sessions from the user frontend. At each video chunk download event, we log to the backend a tuple of (1) network bandwidth estimation, (2) bandwidth measurement for the previous chunk download, (3) the elapsed time of downloading the previous chunk and (4) the file sizes corresponding to different bitrate encodings of the video chunk. The bandwidth estimation is an output from a Facebook networking module. Note that the length of the trace varies naturally across different video sessions due to the difference in the watch time. In our training, we use more than 100,000 traces from production video streaming sessions.

3.6.2 ABR Agent Architecture

Upon downloading each video chunk at each step t, the RL agent observes the state $s_t = (x_t, o_t, \vec{n}_t)$, where x_t is the bandwidth prediction for the next chunk, o_t is the current buffer occupancy and $\vec{n}_t$ is a vector of the file sizes for the next video chunk. As shown in Figure 3-15, the agent samples the next bitrate action a_t based on its parametrized policy: $\pi_\theta(a_t|s_t) \rightarrow [0,1]$. In practice, since the number of bitrate encodings (and thus the length of $\vec{n}_t$) varies across different videos [181], we architect the policy network to take an arbitrary number of file sizes as input. Specifically, for each bitrate, the input to the corresponding policy network consists of the predicted bandwidth and buffer occupancy, concatenated with the corresponding file size. We then copy the *same* neural network for each of the bitrate encodings (i.e., the neural networks shown in Figure 3-15 share the same weights θ). Each copy of the policy network outputs a "priority" value q_t^i for selecting the corresponding bitrate i. Afterwards, we use a softmax [47] operation to map these priority values into a probability

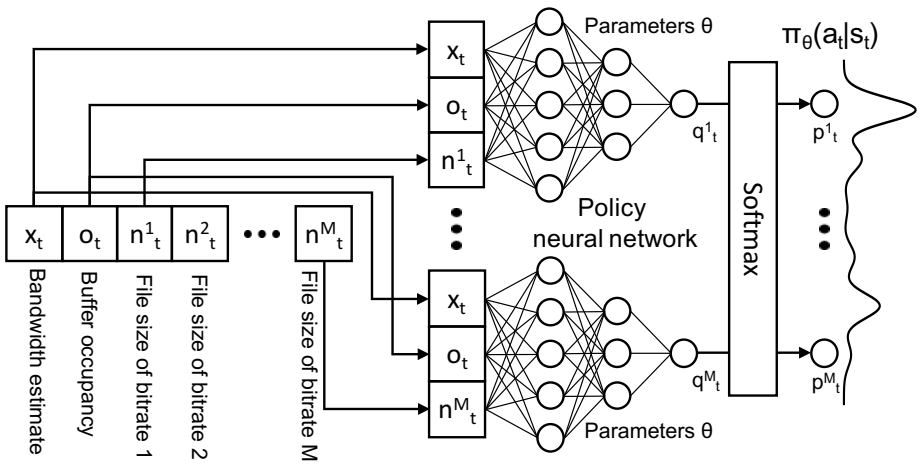


Figure 3-15: Policy network architecture. For each bitrate, the input is fed to a copy of the *same* policy neural network. We then apply a (parameter-free) softmax operator to compute the probability distribution of the next bitrate. This architecture can scale to arbitrary number of bitrate encodings.

distribution p^i_t over each bitrate: $p^i = \exp(q^i_t)/\sum_{i=1}^{M}[\exp(q^i_t)]$. Importantly, the whole policy network architecture is end-to-end differentiable and can be trained with the policy gradient algorithms [286].

Training. We use the policy gradient method [286, 285, 293] to update the policy neural network parameters in order to optimize for the objective. Consider a simulated video streaming session of length T, where the agent collects (state, action, reward) experiences, i.e., (s_t,a_t,r_t) at each step t. The policy gradient method updates the policy parameter θ using the estimated gradient of the cumulative reward:

$$\theta \leftarrow \theta + \alpha \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(s_t,a_t) \left(\sum_{t'=t}^{T} r_{t'} - b_t \right), \tag{3.11}$$

where α is the learning rate and b_k is a *baseline* for reducing the variance of the policy gradient [316].

Notice that the estimation of the advantage over the average case relies on the accurate estimation of the average. For this problem, the standard baselines, such as the time-based baseline [125, 317] or value function [219], suffer from large variance due to the stochasticity in the traces [207]. We further describe the details of this variance in §3.6.3 and our approach to reducing it.

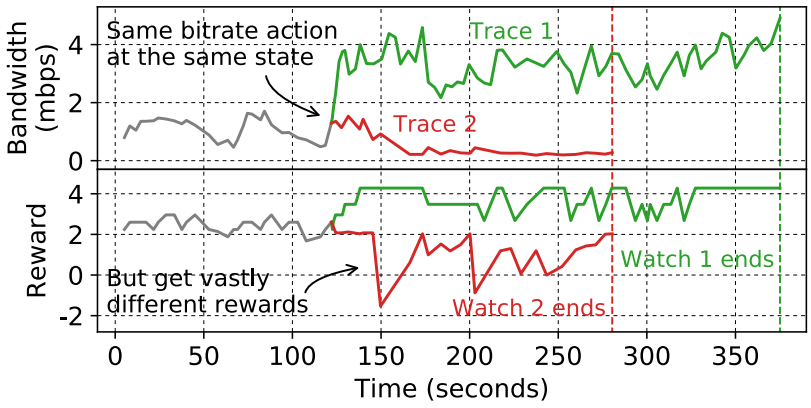


Figure 3-16: Illustrative example of how the difference in the traces of network bandwidth and video watch time creates significant variance for the reward feedback.

3.6.3 Variance Reduction

ABRL's RL training on the simulator is powered by a large number of network traces collected from the front end video platform (§3.6.1). During training, ABRL must experience a wide variety of network conditions and video watches in order to generalize its ABR policy well. However, this creates a challenge for training: different traces contain very different network bandwidth and video duration, which significantly affects the total reward observed by the RL agent. Consider an illustrative example shown in Figure 3-16, where we use a *fixed* buffer-based ABR policy [146] to pick the bitrate action at time τ. Even for this fixed policy, if the future trace happens to contain large bandwidth (e.g., Trace 1), the reward feedback will naturally be large, since the network can support high bitrate without stalls. By contrast, if the future network condition becomes poor (e.g., Trace 2), the reward will likely be lower than average. More importantly, the video duration determines the possible length of ABR interactions, which dictates the *total* reward the RL agent can receive for training (e.g., the longer watch time in Trace 1 leads to larger total reward). The key problem is that the difference across the traces is independent of the bitrate action at time τ — e.g., the future bandwidth might fluctuate due to the inherent stochasticity in the network; or a user might stop watching a video regardless of the quality. As a result, this creates large variance in the reward feedback used for estimating the policy gradient in Equation (5.1).

To solve this problem, we adopt a new baselining technique (§5) for handling an exogenous, stochastic process in the environment when training RL agents [207]. The key idea is to modify the baseline in Equation (3.11) to an "input-dependent" one that takes the input process (e.g., the trace in this problem) into account explicitly. In particular, for this problem, we

implement the input-dependent baseline by loading the *same* trace (i.e., the same time-series for network bandwidth and the same video watch time) multiple times and computing the average total reward at each time step among these video sessions. Essentially, this uses the time-based baseline [125] for Equation (3.11) but computes the average return conditional on the specific instantiation of a trace. During training, we repeat this procedure for a large number of randomly-sampled network traces. As a result, this approach entirely removes the variance caused by the difference in future network condition or the video duration. Since the difference in the reward feedback is only due to the difference in the actions, this enables the RL agent to assess the quality of different actions much more accurately. In Figure 3-19, we show how this approach helps improve the training performance. Chapter 5 describes this variance reduction technique formally and generalizes it to a broad class of "input-driven" environments.

3.6.4 Reward Shaping with Bayesian Optimization

The goal of ABRL is to outperform the existing ABR policy according to multiple production objectives (i.e., increasing the video quality while reducing the stall time). However, these objectives have an inherent trade-off: optimizing one dimension (by tuning up the corresponding reward weight) diminishes the performance in another dimension (e.g., high video quality increases the risk of stalls). Specifically, as a feedback for the bitrate action a_t, the agent receives a reward r_t constructed as a weighted combination of selected bitrate b_t and stall time of the past chunk d_t:

$$r_t = w_b b_t^{v_b} - w_d d_t^{v_d} + w_c[\mathbb{1}(d_t > 0)], \tag{3.12}$$

where $\mathbb{1}(\cdot)$ is an indicator function counting for stall events, and w_b, w_d, w_c, v_b, v_d are the tuning weights for the reward. Notice that these weights cannot be predetermined, because the goal of RL-based ABR is to outperform the existing ABR algorithm in *every* dimension of the metric (i.e., higher bitrate, less stall time and less stall count), which does not amount to a quantitative objective.

To determine the proper combination of the reward weights, we treat ABRL's RL training module (§3.6.2, §3.6.3) as a black box function $f(\vec{w}) \rightarrow (q, l)$ that maps the reward weights $\vec{w} \triangleq (w_b, w_d, w_c, v_b, v_d)$ from Equation (3.12) to a noisy estimate of the average video quality q and stall rate l in unseen test video sessions.

Then, we use Bayesian optimization [273] to efficiently search for the weight combinations that leads to better (q, l), with only a few invocations of the RL training module. This procedure of tuning the weights in the reward function is a realization of reward shaping [231]. We formulate the multi-dimensional optimization problem as a constrained optimization problem:

$$\text{argmax}_{\vec{w}}\, q(\vec{w}), \text{subject to } \frac{l(\vec{w})}{l_s} \leq C, \tag{3.13}$$

where $q(\vec{w})$ and $l(\vec{w})$ are the quality and stall rate evaluated at $\vec{w}$, l_s is the stall rate of the existing policy (non-RL based) used in production at Facebook, and C is a constant.

Notice that the function $q(\cdot)$ and $l(\cdot)$ can only be observed by running the RL training module — a computationally intensive procedure. We solve this constrained optimization problem with Bayesian optimization. Bayesian optimization uses a Gaussian process (GP) [256] surrogate model to approximate the results of the RL training procedure using a limited number of training runs. Gaussian processes are flexible non-parametric Bayesian models representing a posterior distribution over possible smooth functions compatible with the data. We find that GPs are excellent models of the output of the RL training module, as small changes to the reward function will result in small changes in the overall outcomes. Furthermore, GPs are known to produce good estimates of uncertainty.

Using Bayesian optimization, we start from an initial set of M design points $\{\vec{w}_i\}_{i=1}^{M}$, and iteratively test new points on the RL module according to an acquisition function that navigates the explore/exploit tradeoff based on the GP surrogate model.

A popular acquisition function for Bayesian optimization is expected improvement (EI) (see [107, §4.1]). The basic version of EI simply computes the expected value of improvement at each point relative to the best observed point: $\alpha_{EI}(\vec{x}|f^*) = \mathbb{E}_{y\sim g(\vec{x}|\mathcal{D})}[\max(0, f(y) - f^*)]$, where $\mathcal{D} \triangleq \{\vec{w}_i, q(\vec{w}_i)\}_{i=1}^{N}$ represents N runs of data points, f^* is the current best observed value and $g(\vec{x}|\mathcal{D})$ denotes the the posterior distribution of f value from the surrogate.

We use a variant of EI — Noisy Expected Improvement (NEI) — which supports optimization of noisy, constrained function evaluations [185]. While EI and its constrained variants (e.g., [185]), are designed to optimize deterministic functions (which have a known best feasible values), NEI integrates over the uncertainty in which observed points are best, and weights the value of each point by the probability of feasibility.

NEI is a natural fit for our optimization task, since the training procedure is stochastic

(e.g., it depends on the random seed). We therefore evaluate the ABRL training module with a given $\vec{w}$ multiple times and compute its standard error, which are then passed into the NEI algorithm. NEI supports batch updating, allowing us to evaluate multiple reward parameterizations in parallel.

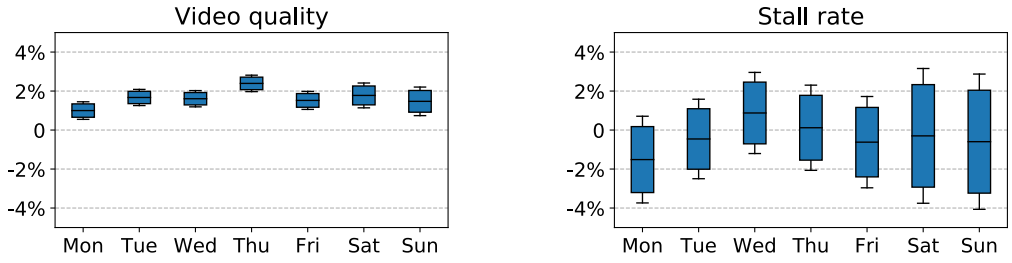
3.6.5 Policy Translation

In practice, most video players execute the ABR algorithms in the front end to avoid the extra latency for connecting to the back end [10, 281, 6, 146]. Therefore, we need to deploy the learned ABR policy to the users directly — i.e., the design of an ABR server in the back end hosting the requests from all users is not ideal [204]. To deploy the learned policy, we make use of the web-based video platform at Facebook, where the front end service (if uncached) fetches the most up-to-date video player (including the ABR policy) from the back end server at the beginning of a video streaming session.

For ease of understanding and maintenance in deployment, we translate the neural network ABR policy to an interpretable form. In particular, we found that the learned ABR policies approximately exhibit a linear structure — the bitrate decision boundaries are approximately linear and the distances between the boundaries are constant in part of the decision space. As a result, we approximate the learned ABR policy with a deterministic linear fitting function. Specifically, we first randomly pick N tuples of bandwidth prediction x and buffer occupancy o (see the inputs in Figure 3-15). Then, for each tuple values (x,o) and for each of the M equally spaced bitrates with file sizes $n^1, n^2, \cdots, n^M$, we invoke the policy network to compute the probability of selecting the corresponding bitrate: $\pi(a^1|x,o,n^1), \pi(a^2|x,o,n^2), \cdots, \pi(a^M|x,o,n^M)$. Next, we determine the "intended" bitrate using a weighted sum: $\bar{n} = \sum_{i=1}^{M} n^i \pi(a^i|x,o,n^i)$. This serves as the target bitrate for the output of the linear fitting function. Finally, we use three parameters a,b, and c, to fit a linear model of bandwidth prediction and buffer occupancy, which minimizes the mean squared error over all N points:

$$\sum_{i=1}^{N} \left| a x_i + b o_i + c - \bar{n}_i \right|^2. \tag{3.14}$$

Here, we use the standard least square estimator for the model fitting, which is the optimal unbiased linear estimator [340]. At inference time, the front end video player uses the fitted



(a) Video quality. **(b)** Stall rate.

Figure 3-17: A week-long performance comparison with production ABR policy. The comparison is sampled from over 30 million video streaming session. The box spans 95% confidence intervals and the bars spans 99% confidence intervals.

linear model to determine the intended bitrate and then selects the maximum available bitrate that is below the intended bitrate.

Translating the neural network ABR policy provides interpretability for human engineers but it is also a compromise in terms of ABR performance (§3.6.7 empirically evaluates this trade-off). Also, adding more contextual-based features would likely require a non-linear policy encoded directly in a neural network. It is worth noting that directly using RL to train a linear policy is a natural choice. However, to our surprise, training ABRL with a linear policy function leads to worse ABR performance than the existing heuristics. We hypothesize this is because policy gradient with a weak function approximator such as a linear one has difficulty converging to the optimal, even though the optimal policy can be simple [195, 108, 95, 4].

3.6.6 Overall Performance in Production

In a week-long deployment on Facebook's production video platform, we compare the performance of ABRL's translated ABR policy (§3.6.5) with that of the existing heuristic-based ABR algorithm. The experiment includes over 30 million worldwide video playback sessions. Figure 3-17 shows the relative improvement of ABRL in terms of video quality and stall rate.

Overall, ABRL achieves a 1.6% increase in average bitrate and a 0.4% decrease in stall rate. Most notably, ABRL consistently selects higher bitrate through the whole week (99% confidence intervals all positive). However, choosing higher bitrates does *not* sacrifice stall rate — ABRL rivals or outperforms the default scheme on the average stall rate every day, even on Thursday when gains in video quality are highest. This shows ABRL uses the output from the bandwidth prediction module better than the fine-tuned heuristic. By directly inter-

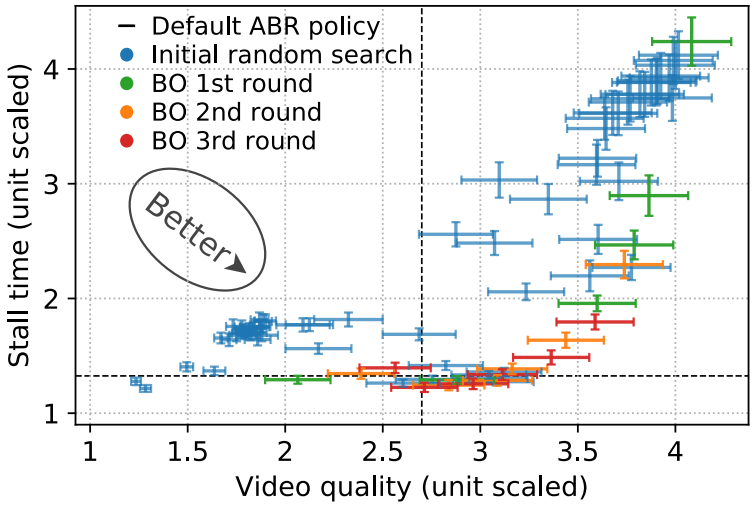


Figure 3-18: Reward shaping via Bayesian optimization using the ABRL simulator. The initial round has 64 random initial parameters. Successive batches of Bayesian optimization converge to optimal weightings that improve video quality while reducing stall rate. The performance is tested on held out network traces.

acting with the observed data, ABRL learns quantitatively how conservative or aggressive the ABR should be with different predicted bandwidths. As a result, this also leads to a 0.2% improvement in the end-user video watch time.

These improvement numbers may look modest compared to the those reported by recent academic papers [146, 282, 329, 204]. This is mostly because we only experiment with web-based videos, which primarily consist of well-connected desktop or laptop traffic, different from the prior schemes that mostly concern cellular and unstable networks. Indeed, the improvement is more substantial in the user group with poor network conditions (Figure 3-21), in which case the ABR problem becomes more challenging. Nonetheless, any non-zero improvement is significant given the massive volume of Facebook videos. In the following, we perform detailed analysis to quantify the performance gains at a more granular level.

3.6.7 Detailed Analysis of RL Pipeline

Reward shaping. To optimize the multi-dimensional objective, we use a Bayesian Optimization approach for reward shaping (§3.6.4). The goal is to tune the weights in the reward function in order to train a policy that operates on the Pareto frontier of video quality and stall (and, ideally, outperform the existing policy in both dimensions). Figure 3-18 shows the performance from different reward weights during the reward shaping procedure. At each

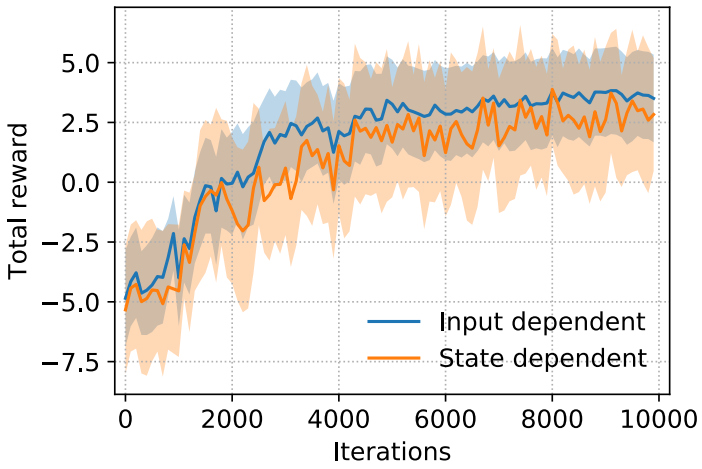
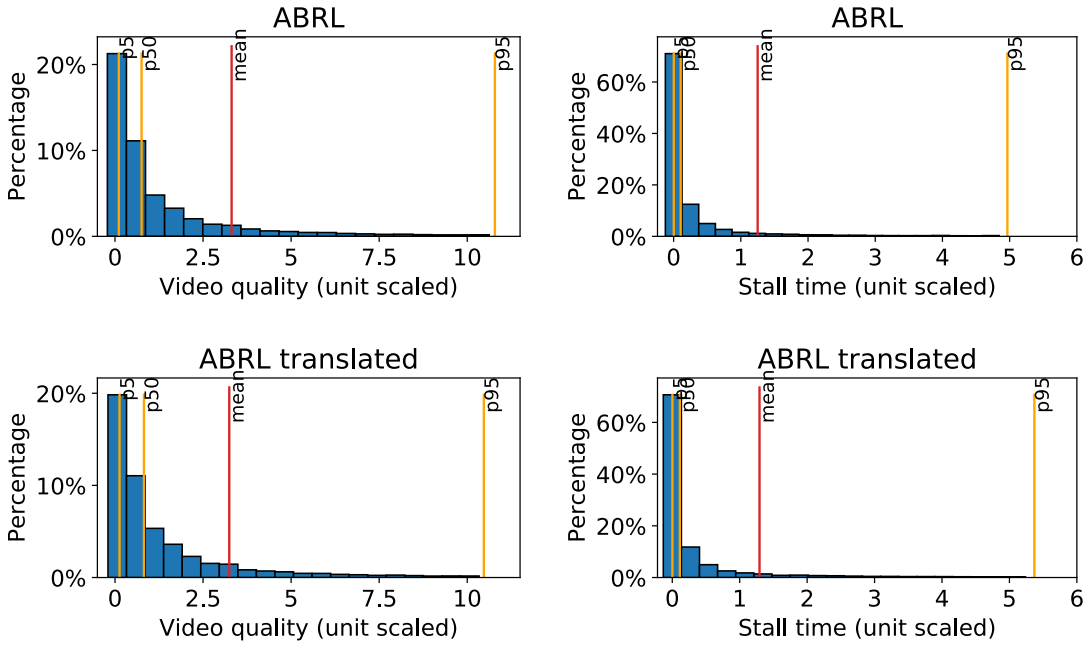


Figure 3-19: Improvements learning performance due to variance reduction. The network condition and watch time in different traces introduces variance in the policy gradient estimation. The input-dependent baseline helps reduce such variance and improve training performance. Shaded area spans $\pm$ std.

iteration, we set the reward weights using the output from the Bayesian optimization module, and treat ABRL's RL module as a black box, in which the policy is trained until convergence according to the chosen reward weights. The Bayesian optimization module then observes the testing outcomes (both video quality and stall) and sets the search criteria for the next iteration to be "expected improvement in video quality such that stall time degrades no more than 5%". As shown, within three iterations, ABRL is able to home in on the empirical Pareto frontier. In this search space, there are many more weight configurations that lead to better video quality (i.e., right of the dashed line) than the configurations leading to fewer stalls (i.e., lower than the dashed line). Compared to the existing ABR scheme, ABRL finds a few candidate reward weights that lead to better ABR policy both in terms of video quality and stalls (i.e., lower and to the right of the existing policy). For the production experiment in §3.6.6, we deploy the policy within the region that shows the largest improvement in stall. After this search procedure, engineers on the video team can pick policies based on different deployment objectives as well.

Variance reduction. To reduce the variance introduced by the network and the watch time across different the traces, we compute the baseline for policy gradient by averaging over the cumulative rewards from the same trace (in all the parallel rollouts) at each iteration, effectively achieving the input-dependent baseline (§3.6.3). For comparison, we also train an agent with the regular state-dependent baseline (i.e., output from a value function that only takes the state observation as input). Figure 3-19 evaluates the impact of variance reduction

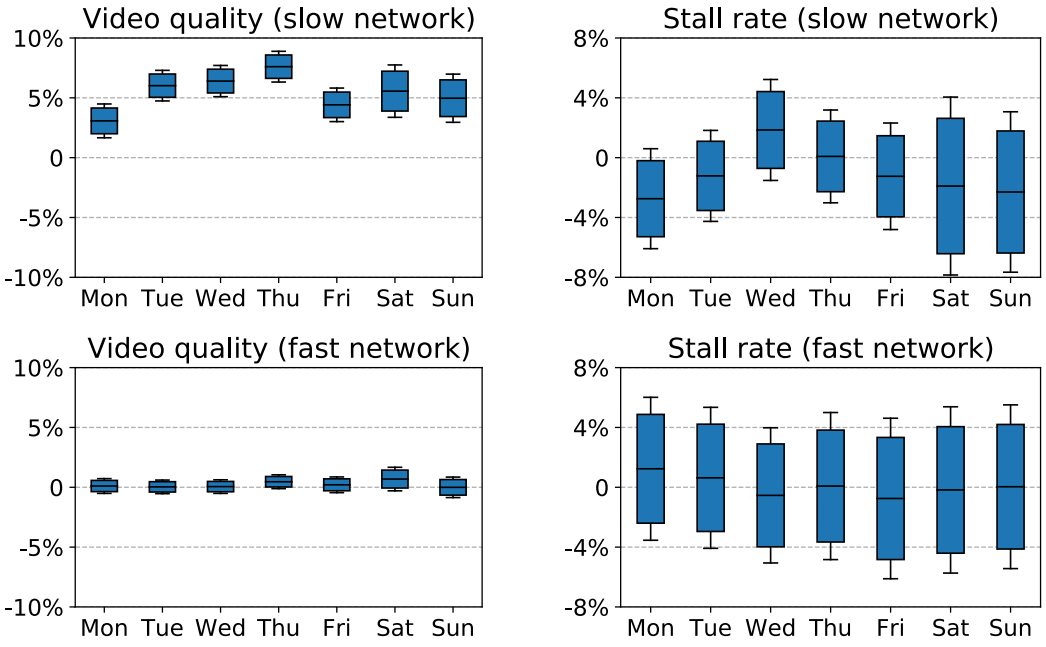


(a) Video quality. **(b)** Stall.

Figure 3-20: Performance comparison of ABRL and its linear approximated variant. The agents are tested with unseen traces in simulation. Translating the policy degrades the average performance by 0.8% in stall and 0.6% in quality.

by comparing the learning curve trained with the input-dependent baseline to that with the state-dependent baseline. As shown, the agent with the input-dependent baseline achieves about 12% higher eventual total reward (i.e., the direct objective of RL training). Moreover, we find that the agent with input-dependent baseline converges faster in terms of the entropy of the policy, which is also indicated by the narrower shaded area in Figure 3-19. At each point in the learning curve, the standard deviation of rewards is around half as large under the input-dependent baseline. This is expected because of the large variance in the policy gradient estimation given the uncertain network throughput in the trace. Fixing the trace at each training iteration removes the variance introduced by the external input process, making the training significantly more stable.

Trade-off of performance for interpretability. Figure 3-20 shows how the testing performance of video quality and stall in simulation differ between ABRL's original neural network policy and the translated policy (§3.6.5). Most noticeably, making the ABR policy linear and interpretable incurs a 0.8% and 8.9% degradation in the mean and 95th percentile of stall rate. This accounts for the tradeoff to make the learned ABR policy fully interpretable. Also, we tried to train a linear policy directly from scratch (by removing hidden layers in the neural net-



(a) Video quality breakdown.

(b) Stall rate breakdown.

Figure 3-21: Breakdown the performance comparison with different network quality for the live experiment. "slow network" corresponds to $< 500K$ mbps measured network bandwidth, and "fast network" corresponds to $> 10M$ mbps bandwidth. The box spans 95% confidence intervals and the bars spans 99% confidence intervals.

work and removing all the non-linear transformations). However, the performance of the directly learned linear policy does not outperform the existing baseline. This in part is because over-parametrization in the policy network helps ABRL learn a more robust policy [195, 95].

Subgroup analysis. To better understand how ABRL outperforms the existing ABR scheme, we breakdown the performance gain in different network conditions. In Figure 3-21, we categorize the video sessions based on the average measured network bandwidths. As shown, ABRL overall achieves a higher bitrate while maintaining fewer stalls in both fast and slow networks. Moreover, ABRL performs significantly better in slow network conditions, where it delivers 5.9% higher bitrate with 2.4% fewer stalls on average. When the network connectivity is unstable, ABR is challenging — a controller must agilely switch to lower bitrate when the bandwidth prediction or buffer level is low, but must avoid being too conservative by persistently sticking with low bitrates (when is feasible to use higher bitrate without stalling). In the slow network condition, ABRL empirically uses the noisy network bandwith estimation better than the heuristic system in order to maintain better buffer levels. This drill down experiment reflects one of the crucial benefits of the learning-based approach: the RL

agent is able to tailor the ABR control policy based on the specific network condition automatically. This also indicates that ABRL optimizes algorithm performance under network conditions that existing schemes may overlook.

3.6.8 Remark

We intend to work on several directions to further enhance ABRL in the production systems. First, ABRL's training is only performed once offline with pre-collected network traces. To better incorporate with the updates in the backend infrastructure, we can set up a continual retraining routine weekly or daily. Prior studies have shown the benefit of continual training with ever updating systems [325].

Second, we primarily evaluate ABRL on Facebook's web-based video platform, because it has the fastest codebase update cycle (unlike mobile development, where the updates are batched in new version releases). However, the network conditions for cellular networks have larger variability and are more unpredictable, where the gain of an RL-based ABR scheme can be larger (e.g., we observed larger performance gain for ABRL when the network condition is poor in §3.6.7. Developing a similar learning framework for mobile clients can potentially lead to larger ABR improvements.

Third, the gains from using ABRL are rather modest, as they use only the same state variables (§3.6.2) as the current heuristic-based ABR algorithm. Given a fixed parameterization of a simple policy, other techniques such as Bayesian optimization currently serve as a more practical alternative to RL. However, ABRL can also we extend the state space to incorporate more contextual features, such as video streaming regions, temporal information, and the contents of the video itself (since categorizing and optimizing the video quality based on video content types can likely result in better perceptual quality), which engineers cannot easily fold into heuristics. We expect that RL methods provide more practical benefit when the state features become richer.

Lastly, there exists a discrepancy between simulated buffer dynamics and the real video streaming session in practice. Better bridging this gap can increase the generalizability of ABRL's learned policy. To this end, there is ongoing work addressing the discrepancy between simulation and reality with Bayesian optimization in reward shaping [184]. Furthermore, another viable approach is to directly perform RL training on the production system. The challenge for this is to construct a similarly safe training mechanism [205, 20] that prevents the initial RL trials from decreasing perceptual quality of a video (e.g., restricting the

initial RL policy from randomly select poor bitrates).

3.7 Related Work

The earliest ABR algorithms can be primarily grouped into two classes: rate-based and buffer-based. Rate-based algorithms [157, 328] first estimate the available network bandwidth using past chunk downloads, and then request chunks at the highest bitrate that the network is predicted to support. For example, Festive [157] predicts throughput to be the harmonic mean of the experienced throughput for the past 5 chunk downloads. However, these methods are hindered by the biases present when estimating available bandwidth on top of HTTP [156, 188]. Several systems aim to correct these throughput estimates using smoothing heuristics and data aggregation techniques [328], but accurate throughput prediction remains a challenge in practice [339].

In contrast, buffer-based approaches [146, 282] solely consider the client's playback buffer occupancy when deciding the bitrates for future chunks. The goal of these algorithms is to keep the buffer occupancy at a pre-configured level which balances rebuffering and video quality. The most recent buffer-based approach, BOLA [282], optimizes for a specified QoE metric using a Lyapunov optimization formulation. BOLA also supports chunk download abandonment, whereby a video player can restart a chunk download at a lower bitrate level if it suspects that rebuffering is imminent.

Each of these approaches performs well in certain settings but not in others. Specifically, rate-based approaches are best at startup time and when link rates are stable, while buffer-based approaches are sufficient and more robust in steady state and in the presence of time-varying networks [146]. Consequently, recently proposed ABR algorithms have also investigated combining these two techniques. The state-of-the-art approach is MPC [329], which employs model predictive control algorithms that use both throughput estimates and buffer occupancy information to select bitrates that are expected to maximize QoE over a horizon of several future chunks. However, MPC still relies heavily on accurate throughput estimates which are not always available. When throughput predictions are incorrect, MPC's performance can degrade significantly. Addressing this issue requires heuristics that make throughput predictions more conservative. However, tuning such heuristics to perform well in different environments is challenging. Further, as we observed in §3.3, MPC is often unable to plan far enough into the future to apply the policies that would maximize performance

in given settings.

A separate line of work has proposed applying RL to adaptive video streaming [63, 300, 69, 70]. All of these schemes apply RL in a "tabular form," which stores and learns the value function for all states and actions explicitly, rather than using function approximators (e.g., neural networks). As a result, these schemes do not scale to the large state spaces necessary for good performance in real networks, and their evaluation has been limited to simulations with synthetic network models. For example, the most recent tabular scheme [63] relies on the fundamental assumption that network bandwidth is Markovian, i.e., the future bandwidth depends only on the throughput observed in the last chunk download. This assumption confines the state space to consider only one past bandwidth measurement, making the tabular approach feasible to implement. As we saw in §3.5.4, the information contained in one past chunk is not sufficient to accurately infer the distribution of future bandwidth. Nevertheless, some of the techniques used in the existing RL video streaming schemes (e.g., Post-Decision States [63, 249]) could be used to accelerate learning in Pensieve as well.

3.8 Conclusion

We presented Pensieve, a system which generates ABR algorithms using reinforcement learning. Unlike ABR algorithms that use fixed heuristics or inaccurate system models, Pensieve's ABR algorithms are generated using observations of the resulting performance of past decisions across a large number of video streaming experiments. This allows Pensieve to optimize its policy for different network characteristics and QoE metrics directly from experience. Over a broad set of network conditions and QoE metrics, we found that Pensieve outperformed existing ABR algorithms by 12%–25%. In Facebook deployment, a customized implementation of Pensieve consistently outperforms the existing production ABR algorithm on over 30 million worldwide video streaming sessions. We open source Pensieve, our models, and our experimental infrastructure at `https://web.mit.edu/pensieve`.

Chapter 4

Learning Scheduling Algorithms for Data Processing Clusters

4.1 Introduction

Efficient utilization of expensive compute clusters matters for enterprises: even small improvements in utilization can save millions of dollars at scale [32, §1.2]. Cluster schedulers are key to realizing these savings. A good scheduling policy packs work tightly to reduce fragmentation [123, 121, 304], prioritizes jobs according to high-level metrics such as user-perceived latency [305], and avoids inefficient configurations [100]. Current cluster schedulers rely on heuristics that prioritize generality, ease of understanding, and straightforward implementation over achieving the ideal performance on a specific workload. By using general heuristics like fair scheduling [23, 115], shortest-job-first, and simple packing strategies [121], current systems forego potential performance optimizations. For example, widely-used schedulers ignore readily available information about job structure (i.e., internal dependencies) and efficient parallelism for jobs' input sizes. Unfortunately, workload-specific scheduling policies that use this information require expert knowledge and significant effort to devise, implement, and validate. For many organizations, these skills are either unavailable, or uneconomic as the labor cost exceeds potential savings.

In this thesis, we show that modern machine-learning techniques can help side-step this trade-off by *automatically learning* highly efficient, workload-specific scheduling policies. We present Decima,[1] a general-purpose scheduling service for data processing jobs with de-

[1] In Roman mythology, Decima measures threads of life and decides their destinies.

pendent stages. Many systems encode job stages and their dependencies as directed acyclic graphs (DAGs) [333, 290, 150, 58]. Efficiently scheduling DAGs leads to hard algorithmic problems whose optimal solutions are intractable [123]. Given only a high-level goal (e.g., minimize average job completion time), Decima uses existing monitoring information and past workload logs to automatically learn sophisticated scheduling policies. For example, instead of a rigid fair sharing policy, Decima learns to give jobs different shares of resources to optimize overall performance, and it learns job-specific parallelism levels that avoid wasting resources on diminishing returns for jobs with little inherent parallelism. The right algorithms and thresholds for these policies are workload-dependent, and achieving them today requires painstaking manual scheduler customization.

Decima learns scheduling policies through experience using modern reinforcement learning (RL) techniques. RL is well-suited to learning scheduling policies because it allows learning from actual workload and operating conditions without relying on inaccurate assumptions. Decima encodes its scheduling policy in a neural network trained via a large number of simulated experiments, during which it schedules a workload, observes the outcome, and gradually improves its policy. However, Decima's contribution goes beyond merely applying off-the-shelf RL algorithms to scheduling: to successfully learn high-quality scheduling policies, we had to develop novel data and scheduling action representations, and new RL training techniques.

First, cluster schedulers must scale to hundreds of jobs and thousands of machines, and must decide among potentially hundreds of configurations per job (e.g., different levels of parallelism). This leads to much larger problem sizes compared to conventional RL applications (e.g., game-playing [220, 277], robotics control [191, 268]), both in the amount of information available to the scheduler (the *state space*), and the number of possible choices it must consider (the *action space*).[2] We designed a scalable neural network architecture that combines a *graph neural network* [77, 73, 170, 35] to process job and cluster information without manual feature engineering, and a *policy network* that makes scheduling decisions. Our neural networks reuse a small set of building block operations to process job DAGs, irrespective of their sizes and shapes, and to make scheduling decisions, irrespective of the number of jobs or machines. These operations are parameterized functions learned during training, and designed for the scheduling domain — e.g., ensuring that the graph neural

[2]For example, the state of the game of Go [279] can be represented by $19 \times 19 = 361$ numbers, which also bound the number of legal moves per turn.

network can express properties such as a DAG's critical path. Our neural network design substantially reduces model complexity compared to naive encodings of the scheduling problem, which is key to efficient learning, fast training, and low-latency scheduling decisions.

Second, conventional RL algorithms cannot train models with continuous streaming job arrivals. The randomness of job arrivals can make it impossible for RL algorithms to tell whether the observed outcome of two decisions differs due to disparate job arrival patterns, or due to the quality the policy's decisions. Further, RL policies necessarily make poor decisions in early stages of training. Hence, with an unbounded stream of incoming jobs, the policy inevitably accumulates a backlog of jobs from which it can never recover. Spending significant training time exploring actions in such situations fails to improve the policy. To deal with the latter problem, we terminate training "episodes" early in the beginning, and gradually grow their length. This allows the policy to learn to handle simple, short job sequences first, and to then graduate to more challenging arrival sequences. To cope with the randomness of job arrivals, we condition training feedback on the actual sequence of job arrivals experienced, using a recent technique for RL in environments with stochastic inputs [207]. This isolates the contribution of the scheduling policy in the feedback and makes it feasible to learn policies that handle stochastic job arrivals.

We integrated Decima with Spark and evaluated it in both an experimental testbed and on a workload trace from Alibaba's production clusters [12, 193].[3] Our evaluation shows that Decima outperforms existing heuristics on a 25-node Spark cluster, reducing average job completion time of TPC-H query mixes by at least 21%. Decima's policies are particularly effective during periods of high cluster load, where it improves the job completion time by up to 2× over existing heuristics. Decima also extends to multi-resource scheduling of CPU and memory, where it improves average job completion time by 32-43% over prior schemes such as Graphene [123].

In summary, we make the following key contributions:

1. A scalable neural network design that can process DAGs of arbitrary shapes and sizes, schedule DAG stages, and set efficient parallelism levels for each job (§4.5.1–§4.5.2).
2. A set of RL training techniques that for the first time enable training a scheduler to handle unbounded stochastic job arrival sequences (§4.5.3).
3. Decima, the first RL-based scheduler that schedules complex data processing jobs and learns workload-specific scheduling policies without human input, and a prototype

[3] We used an earlier version of Alibaba's public `cluster-trace-v2018` trace.

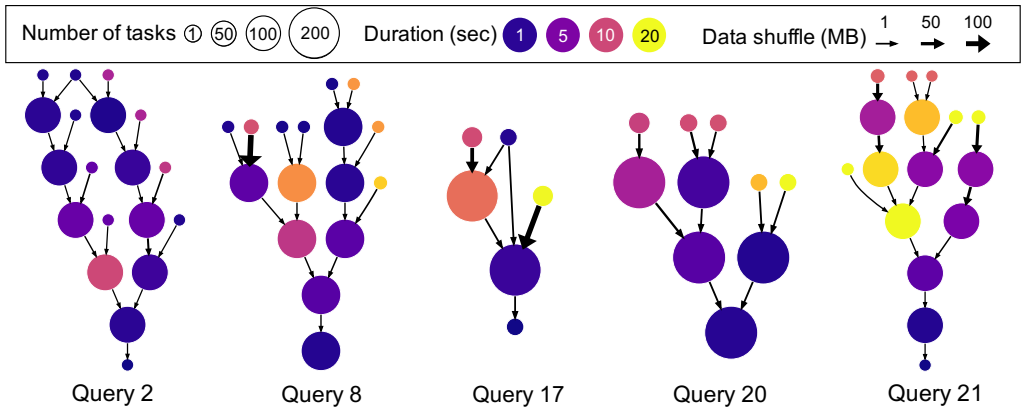


Figure 4-1: Data-parallel jobs have complex data-flow graphs like the ones shown (TPC-H queries in Spark), with each node having a distinct number of tasks, task durations, and input/output sizes.

implementation of it (§4.6).

4. An evaluation of Decima in simulation and in a real Spark cluster, and a comparison with state-of-the-art scheduling heuristics (§4.7).

4.2 Motivation

Data processing systems and query compilers such as Hive, Pig, SparkSQL, and DryadLINQ create *DAG-structured* jobs, which consist of processing stages connected by input/output dependencies (Figure 4-1). For recurring jobs, which are common in production clusters [7], reasonable estimates of runtimes and intermediate data sizes may be available. Most cluster schedulers, however, ignore this job structure in their decisions and rely on e.g., coarse-grained fair sharing [23, 115, 46, 116], rigid priority levels [305], and manual specification of each job's parallelism [271, §5]. Existing schedulers choose to largely ignore this rich, easily-available job structure information because it is difficult to design scheduling algorithms that make use of it. We illustrate the challenges of using job-specific information in scheduling decisions with two concrete examples: (1) dependency-aware scheduling, and (2) automatically choosing the right number of parallel tasks.

4.2.1 Dependency-Aware Task Scheduling

Many job DAGs in practice have tens or hundreds of stages with different durations and numbers of parallel tasks in a complex dependency structure. An ideal schedule ensures that independent stages run in parallel as much as possible, and that no stage ever blocks on a de-

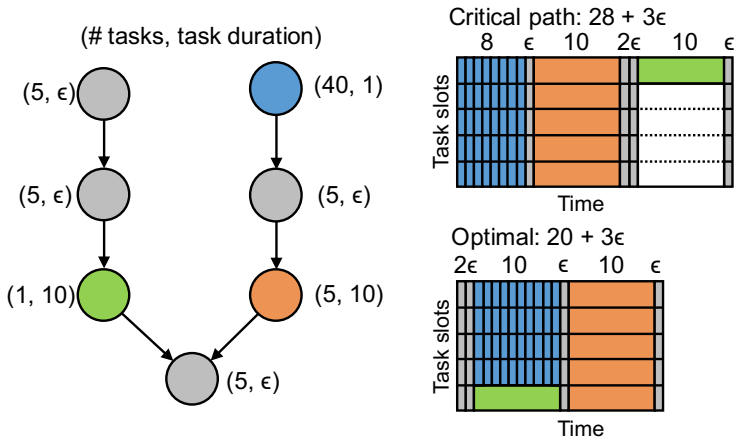


Figure 4-2: An optimal DAG-aware schedule plans ahead and parallelizes execution of the blue and green stages, so that orange and green stages complete at the same time and the bottom join stage can execute immediately. A straightforward critical path heuristic would instead focus on the right branch, and takes 29% longer to execute the job.

pendency if there are available resources. Ensuring this requires the scheduler to understand the dependency structure and plan ahead. This "DAG scheduling problem" is algorithmically hard [123, §2.2]. For example, Figure 4-2 shows a common scenario: a DAG with two branches that converge in a join stage. A simple critical path heuristic would choose to work on the right branch, which contains more aggregate work: 90 task-seconds vs. 10 task-seconds in the left branch. With this choice, once the orange stage finishes, however, the final join stage cannot run, since its other parent stage (in green) is still incomplete. Completing the green stage next, followed by the join stage — as a critical-path schedule would — results in an overall makespan of $28+3\epsilon$. The optimal schedule, by contrast, completes this DAG in $20+3\epsilon$ time, 29% faster. Intuitively, an ideal schedule allocates resources such that both branches reach the final join stage at the same time, and execute it without blocking.

Theoretical research [276, 48, 60, 182] has focused mostly on simple instances of the problem that do not capture the complexity of real data processing clusters (e.g., online job arrivals, multiple DAGs, multiple tasks per stage, jobs with different inherent parallelism, overheads for moving jobs between machines, etc.). For example, in a recent paper, Agrawal et al. [8] showed that two simple DAG scheduling policies (shortest-job-first and latest-arrival-processor-sharing) have constant competitive ratio in a basic model with one task per job stage. As our results show (§4.2.3, §4.7), these policies are far from optimal in a real Spark cluster.

Hence, designing an algorithm to generate optimal schedules for all possible DAG com-

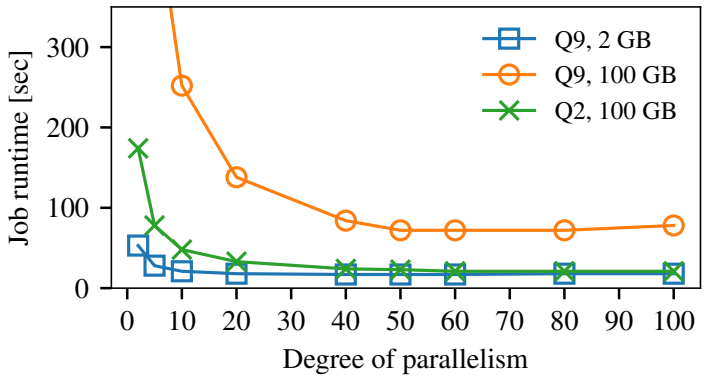


Figure 4-3: TPC-H queries scale differently with parallelism: Q9 on a 100 GB input sees speedups up to 40 parallel tasks, while Q2 stops gaining at 20 tasks; Q9 on a 2 GB input needs only 5 tasks. Picking "sweet spots" on these curves for a mixed workload is difficult.

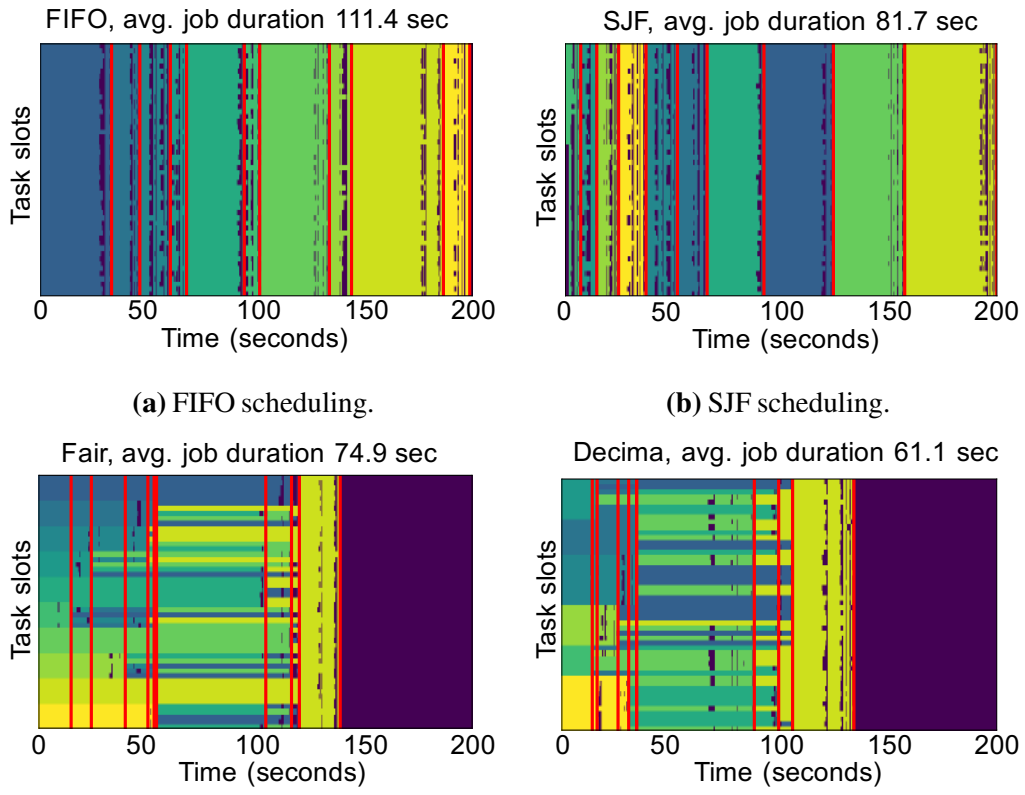
binations is intractable [211, 123]. Existing schedulers ignore this challenge: they enqueue tasks from a stage as soon as it becomes available, or run stages in an arbitrary order.

4.2.2 Setting the Right Level of Parallelism

In addition to understanding dependencies, an ideal scheduler must also understand how to best split limited resources among jobs. Jobs vary in the amount of data that they process, and in the amount of parallel work available. A job with large input or large intermediate data can efficiently harness additional parallelism; by contrast, a job running on small input data, or one with less efficiently parallelizable operations, sees diminishing returns beyond modest parallelism.

Figure 4-3 illustrates this with the job runtime of two TPC-H [295] queries running on Spark as they are given additional resources to run more parallel tasks. Even when both process 100 GB of input, Q2 and Q9 exhibit widely different scalability: Q9 sees significant speedup up to 40 parallel tasks, while Q2 only obtains marginal returns beyond 20 tasks. When Q9 runs on a smaller input of 2 GB, however, it needs no more than ten parallel tasks. For all jobs, assigning additional parallel tasks beyond a "sweet spot" in the curve adds only diminishing gains. Hence, the scheduler should reason about which job will see the largest marginal gain from extra resources and accordingly pick the sweet spot for each job.

Existing schedulers largely side-step this problem. Most burden the user with the choice of how many parallel tasks to use [271, §5], or rely on a separate "auto-scaling" component based on coarse heuristics [100, 24]. Indeed, many fair schedulers [151, 115] divide



(a) FIFO scheduling.

(b) SJF scheduling.

(c) Fair scheduling.

(d) Decima.

Figure 4-4: Decima improves average JCT of 10 random TPC-H queries by 45% over Spark's FIFO scheduler, and by 19% over a fair scheduler on a cluster with 50 task slots (executors). Different queries in different colors; vertical red lines are job completions; purple means idle.

resources without paying attention to their decisions' efficiency: sometimes, an "unfair" schedule results in a more efficient overall execution.

4.2.3 An Illustrative Example on Spark

The aspects described are just two examples of how schedulers can exploit knowledge of the workload. To achieve the best performance, schedulers must also respect other considerations, such as the execution order (e.g., favoring short jobs) and avoiding resource fragmentation [305, 121]. Considering all these dimensions together — as Decima does — makes a substantial difference. We illustrate this by running a mix of ten randomly chosen TPC-H [295] queries with input sizes drawn from a long-tailed distribution on a Spark cluster with 50 parallel task slots.[4] Figure 4-4 visualizes the schedules imposed by *(a)* Spark's

[4] See §4.7 for details of the workload and our cluster setup.

default FIFO scheduling; *(b)* a shortest-job-first (SJF) policy that strictly prioritizes short jobs; *(c)* a more realistic, fair scheduler that dynamically divides task slots between jobs; and *(d)* a scheduling policy learned by Decima. We measure average job completion time (JCT) over the ten jobs. Having access to the graph structure helps Decima improve average JCT by 45% over the naive FIFO scheduler, and by 19% over the fair scheduler. It achieves this speedup by completing short jobs quickly, as five jobs finish in the first 40 seconds; and by maximizing parallel-processing efficiency. SJF dedicates all task slots to the next-smallest job in order to finish it early (but inefficiently); by contrast, Decima runs jobs near their parallelism sweet spot. By controlling parallelism, Decima reduces the total time to complete all jobs by 30% compared to SJF. Further, unlike fair scheduling, Decima partitions task slots non-uniformly across jobs, improving average JCT by 19%.

Designing general-purpose heuristics to achieve these benefits is difficult, as each additional dimension (DAG structure, parallelism, job sizes, etc.) increases complexity and introduces new edge cases. Decima opens up a new option: using data-driven techniques, it *automatically* learns workload-specific policies that can reap these gains. Decima does so without requiring human guidance beyond a high-level goal (e.g., minimal average JCT), and without explicitly modeling the system or the workload.

4.3 The DAG Scheduling Problem in Spark

Decima is a general framework for learning scheduling algorithms for DAG-structured jobs. For concreteness, we describe its design in the context of the Spark system.

A Spark job consists of a DAG whose nodes are the execution *stages* of the job. Each stage represents an operation that the system runs in parallel over many shards of the stage's input data. The inputs are the outputs of one or more parent stages, and each shard is processed by a single *task*. A stage's tasks become runnable as soon as all parent stages have completed. How many tasks can run in parallel depends on the number of *executors* that the job holds. Usually, a stage has more tasks than there are executors, and the tasks therefore run in several "waves". Executors are assigned by the Spark master based on user requests, and by default stick to jobs until they finish. However, Spark also supports dynamic allocation of executors based on the wait time of pending tasks [24], although moving executors between jobs incurs some overhead (e.g., to tear down and launch JVMs).

Spark must therefore handle three kinds of scheduling decisions: (1) deciding how many

executors to give to each job; (2) deciding which stages' tasks to run next for each job, and (3) deciding which task to run next when an executor becomes idle. When a stage completes, its job's *DAG scheduler* handles the activation of dependent child stages and enqueues their tasks with a lower-level *task scheduler*. The task scheduler maintains task queues from which it assigns a task every time an executor becomes idle.

We allow the scheduler to move executors between job DAGs as it sees fit (dynamic allocation). Decima focuses on DAG scheduling (i.e., which stage to run next) and executor allocation (i.e., each job's degree of parallelism). Since tasks in a stage run identical code and request identical resources, we use Spark's existing task-level scheduling.

4.4 Overview and Design Challenges

Decima represents the scheduler as an agent that uses a neural network to make decisions, henceforth referred to as the *policy network*. On *scheduling events* — e.g., a stage completion (which frees up executors), or a job arrival (which adds a DAG) — the agent takes as input the current *state* of the cluster and outputs a scheduling *action*. At a high level, the state captures the status of the DAGs in the scheduler's queue and the executors, while the actions determine which DAG stages executors work on at any given time.

Decima trains its neural network using RL through a large number of offline (simulated) experiments. In these experiments, Decima attempts to schedule a workload, observes the outcome, and provides the agent with a *reward* after each action. The reward is set based on Decima's high-level scheduling objective (e.g., minimize average JCT). The RL algorithm uses this reward signal to gradually improve the scheduling policy.

Decima's RL framework (Figure 4-5) is general and it can be applied to a variety of systems and objectives. In §4.5, we describe the design for scheduling DAGs on a set of identical executors to minimize average JCT. Our results in §4.7 will show how to apply the same design to schedule multiple resources (e.g., CPU and memory), optimize for other objectives like makespan [252], and learn qualitatively different polices depending on the underlying system (e.g., with different overheads for moving jobs across machines).

Challenges. Decima's design tackles three key challenges:

1. **Scalable state information processing.** The scheduler must consider a large amount of dynamic information to make scheduling decisions: hundreds of job DAGs, each with dozens of stages, and executors that may each be in a different state (e.g., assigned

to different jobs). Processing all of this information via neural networks is challenging, particularly because neural networks usually require fixed-sized vectors as inputs.

2. **Huge space of scheduling decisions.** The scheduler must map potentially thousands of runnable stages to available executors. The exponentially large space of mappings poses a challenge for RL algorithms, which must "explore" the action space in training to learn a good policy.
3. **Training for continuous stochastic job arrivals**. It is important to train the scheduler to handle continuous randomly-arriving jobs over time. However, training with a continuous job arrival process is non-trivial because RL algorithms typically require training "episodes" with a finite time horizon. Further, we find that randomness in the job arrival process creates difficulties for RL training due to the variance and noise it adds to the reward.

4.5 Design

This section describes Decima's design, structured according to how it addresses the three aforementioned challenges: scalable processing of the state information (§4.5.1), efficiently encoding scheduling decisions as actions (§4.5.2), and RL training with continuous stochastic job arrivals (§4.5.3).

4.5.1 Scalable State Information Processing

On each state observation, Decima must convert the state information (job DAGs and executor status) into features to pass to its policy network. One option is to create a flat feature vector containing all the state information. However, this approach cannot scale to arbitrary number of DAGs of arbitrary sizes and shapes. Further, even with a hard limit on the number of jobs and stages, processing a high-dimensional feature vector would require a large policy network that would be difficult to train.

Decima achieves scalability using a *graph neural network*, which encodes or "embeds" the state information (e.g., attributes of job stages, DAG dependency structure, etc.) in a set of *embedding* vectors. Our method is based on graph convolutional neural networks [170, 73, 35] but customized for scheduling. Table 4.1 defines our notation.

The graph embedding takes as input the job DAGs whose nodes carry a set of stage attributes (e.g., the number of remaining tasks, expected task duration, etc.), and it outputs

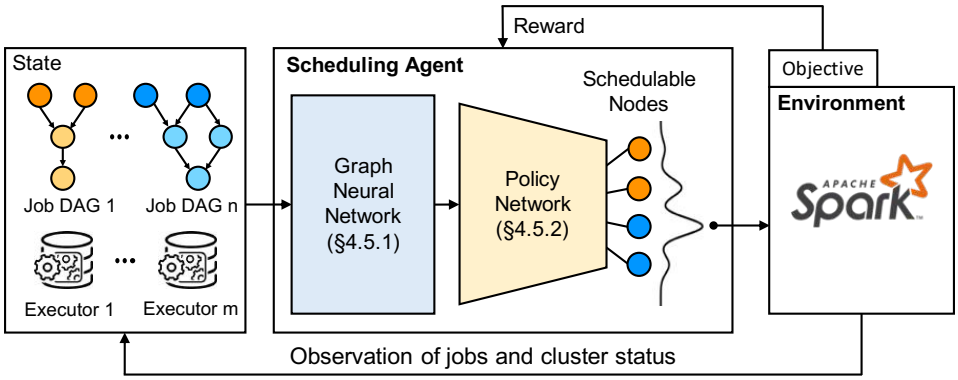


Figure 4-5: In Decima's RL framework, a *scheduling agent* observes the *cluster state* to decide a scheduling *action* on the cluster *environment*, and receives a *reward* based on a high-level objective. The agent uses a *graph neural network* to turn job DAGs into vectors for the *policy network*, which outputs actions.

entity	symbol	entity	symbol
job	i	per-node feature vector	$\mathbf{x}_v^i$
stage (DAG node)	v	per-node embedding	$\mathbf{e}_v^i$
node v's children	$\xi(v)$	per-job embedding	$\mathbf{y}^i$
job i's DAG	G_i	global embedding	$\mathbf{z}$
job i's parallelism	l_i	node score	q_v^i
non-linear functions	f,g,q,w	parallelism score	w_l^i

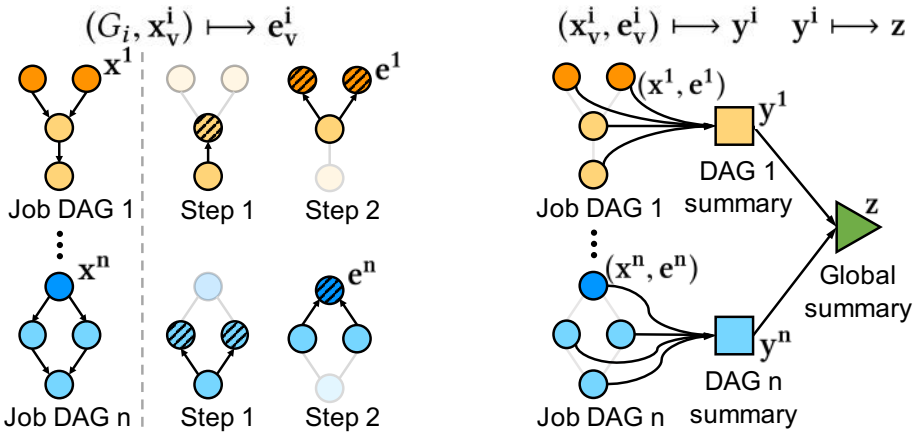
Table 4.1: Notation used throughout §4.5.

three different types of embeddings:

1. per-node embeddings, which capture information about the node and its children (containing, e.g., aggregated work along the critical path starting from the node);
2. per-job embeddings, which aggregate information across an entire job DAG (containing, e.g., the total work in the job); and
3. a global embedding, which combines information from all per-job embeddings into a cluster-level summary (containing, e.g., the number of jobs and the cluster load).

Importantly, what information to store in these embeddings is not hard-coded — Decima automatically learns what is statistically important and how to compute it from the input DAGs through end-to-end training. In other words, the embeddings can be thought of as feature vectors that the graph neural network learns to compute without manual feature engineering. Decima's graph neural network is scalable because it reuses a common set of operations as building blocks to compute the above embeddings. These building blocks are themselves implemented as small neural networks that operate on relatively low-dimensional input vectors.

Per-node embeddings. Given the vectors $\mathbf{x}_v^i$ of stage attributes corresponding to the nodes



(a) Per-node embedding. **(b)** Per-job and global embeddings.

Figure 4-6: A *graph neural network* transforms the raw information on each DAG node into a vector representation. This example shows two steps of local message passing and two levels of summarizations.

in DAG G_i, Decima builds a per-node embedding $(G_i, \mathbf{x}_v^i) \longmapsto \mathbf{e}_v^i$. The result $\mathbf{e}_v^i$ is a vector (e.g., in $\mathbb{R}^{16}$) that captures information from all nodes reachable from v (i.e., v's child nodes, their children, etc.). To compute these vectors, Decima propagates information from children to parent nodes in a sequence of *message passing* steps, starting from the leaves of the DAG (Figure 4-6a). In each message passing step, a node v whose children have aggregated messages from all of their children (shaded nodes in Figure 4-6a's examples) computes its own embedding as:

$$\mathbf{e}_v^i = g\left[\sum_{u \in \xi(v)} f(\mathbf{e}_u^i)\right] + \mathbf{x}_v^i, \tag{4.1}$$

where $f(\cdot)$ and $g(\cdot)$ are non-linear transformations over vector inputs, implemented as (small) neural networks, and $\xi(v)$ denotes the set of v's children. The first term is a general, non-linear aggregation operation that summarizes the embeddings of v's children; adding this summary term to v's feature vector ($\mathbf{x}_v$) yields the embedding for v. Decima reuses the same non-linear transformations $f(\cdot)$ and $g(\cdot)$ at all nodes, and in all message passing steps.

Most existing graph neural network architectures [170, 77, 73] use aggregation operations of the form $\mathbf{e_v} = \sum_{u \in \xi(v)} f(\mathbf{e_u})$ to compute node embeddings. However, we found that adding a second non-linear transformation $g(\cdot)$ in Eq. (4.1) is critical for learning strong scheduling policies. The reason is that without $g(\cdot)$, the graph neural network cannot compute certain useful features for scheduling. For example, it cannot compute the critical path [165] of a DAG, which requires a max operation across the children of a node dur-

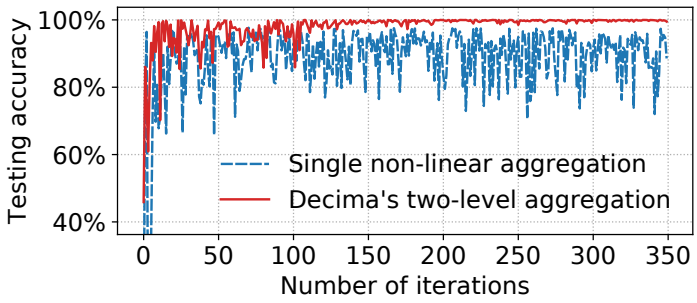


Figure 4-7: Trained using supervised learning, Decima's two-level non-linear transformation is able to express the max operation necessary for computing the critical path (§4.5.1), and consequently achieves near-perfect accuracy on unseen DAGs compared to the standard graph embedding scheme.

ing message passing.[5] Combining two non-linear transforms $f(\cdot)$ and $g(\cdot)$ enables Decima to express a wide variety of aggregation functions. For example, if f and g are identity transformations, the aggregation sums the child node embeddings; if $f \sim \log(\cdot/n)$, $g \sim \exp(n \times \cdot)$, and $n \to \infty$, the aggregation computes the max of the child node embeddings.

During development, we relied on a simple sanity check to test the expressiveness of a graph embedding scheme. We used supervised learning to train the graph neural network to output the critical path value of each node in a large number of random graphs, and then checked how accurately the graph neural network identified the node with the maximum critical path value. Figure 4-7 shows the testing accuracy that Decima's node embedding with two aggregation levels achieves on unseen graphs, and compares it to the accuracy achieved by a simple, single-level embedding with only one non-linear transformation. Decima's node embedding manages to learn the max operation and therefore accurately identifies the critical path after about 150 iterations, while the standard embedding is incapable of expressing the critical path and consequently never reaches a stable high accuracy.

Per-job and global embeddings. The graph neural network also computes a summary of all node embeddings for each DAG G_i, $\{(\mathbf{x}_v^i, \mathbf{e}_v^i), v \in G_i\} \longmapsto \mathbf{y}^i$; and a global summary across all DAGs, $\{\mathbf{y}^1, \mathbf{y}^2, \ldots\} \longmapsto \mathbf{z}$. To compute these embeddings, Decima adds a summary node to each DAG, which has all the nodes in the DAG as children (the squares in Figure 4-6b). These DAG-level summary nodes are in turn children of a single global summary node (the triangle in Figure 4-6b). The embeddings for these summary nodes are also computed using Eq. (4.1). Each level of summarization has its own non-linear transformations f and g; in

[5] The critical path from node v can be computed as: $\text{cp}(v) = \max_{u \in \xi(v)} \text{cp}(u) + \text{work}(v)$, where work$(\cdot)$ is the total work on node v.

other words, the graph neural network uses six non-linear transformations in total, two for each level of summarization.

4.5.2 Encoding Scheduling Decisions as Actions

The key challenge for encoding scheduling decisions lies in the learning and computational complexities of dealing with large action spaces. As a naive approach, consider a solution, that given the embeddings from §4.5.1, returns the assignment for all executors to job stages in one shot. This approach has to choose actions from an exponentially large set of combinations. On the other extreme, consider a solution that invokes the scheduling agent to pick one stage every time an executor becomes available. This approach has a much smaller action space (O(# stages)), but it requires long sequences of actions to schedule a given set of jobs. In RL, both large action spaces and long action sequences increase sample complexity and slow down training [285, 22].

Decima balances the size of the action space and the number of actions required by decomposing scheduling decisions into a series of two-dimensional actions, which output (1) a stage designated to be scheduled next, and (2) an upper limit on the number of executors to use for that stage's job.

Scheduling events. Decima invokes the scheduling agent when the set of runnable stages — i.e., stages whose parents have completed and which have at least one waiting task — in any job DAG changes. Such scheduling events happen when (1) a stage runs out of tasks (i.e., needs no more executors), (2) a stage completes, unlocking the tasks of one or more of its child stages, or (3) a new job arrives to the system.

At each scheduling event, the agent schedules a group of free executors in one or more actions. Specifically, it passes the embedding vectors from §4.5.1 as input to the policy network, which outputs a two-dimensional action $\langle v, l_i \rangle$, consisting of a stage v and the parallelism limit l_i for v's job i. If job i currently has fewer than l_i executors, Decima assigns executors to v up to the limit. If there are still free executors after the scheduling action, Decima invokes the agent again to select another stage and parallelism limit. This process repeats until all the executors have been assigned, or there are no more runnable stages. Decima ensures that this process completes in a finite number of steps by enforcing that the parallelism limit l_i is greater than the number of executors currently allocated to job i, so that at least one new executor is scheduled with each action.

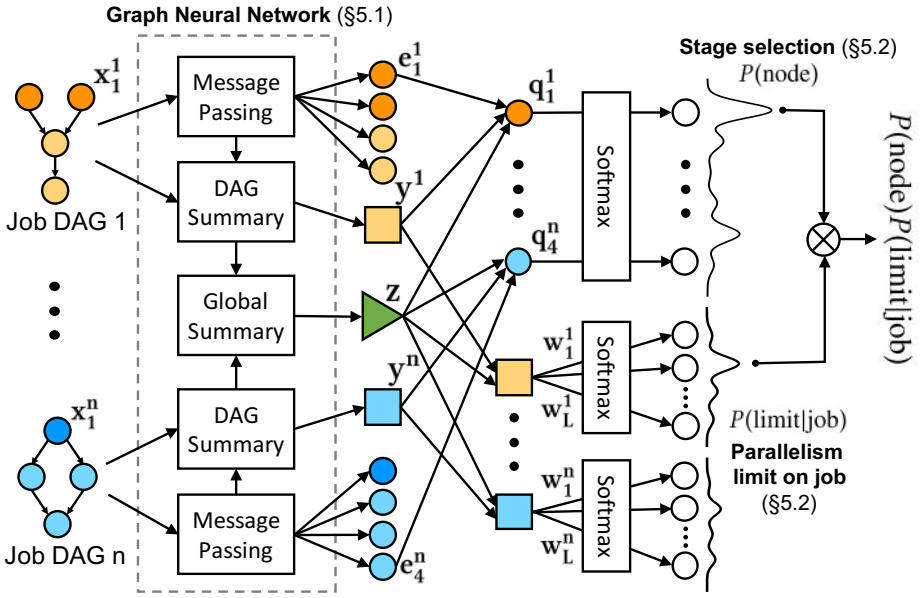


Figure 4-8: For each node v in job i, the *policy network* uses per-node embedding $\mathbf{e}_v^i$, per-job embedding $\mathbf{y}^i$ and global embedding $\mathbf{z}$ to compute (1) the score $\mathbf{q}_v^i$ for sampling a node to schedule and (2) the score $\mathbf{w}_l^i$ for sampling a parallelism limit for the node's job.

Stage selection. Figure 4-8 visualizes Decima's policy network. For a scheduling event at time t, during which the state is s_t, the policy network selects a stage to schedule as follows. For a node v in job i, it computes a score $q_v^i \triangleq q(\mathbf{e}_v^i, \mathbf{y}^i, \mathbf{z})$, where $q(\cdot)$ is a *score function* that maps the embedding vectors (output from the graph neural network in §4.5.1) to a scalar value. Similar to the embedding step, the score function is also a non-linear transformation implemented as a neural network. The score q_v^i represents the priority of scheduling node v. Decima then uses a softmax operation [47] to compute the probability of selecting node v based on the priority scores:

$$P(\text{node}=v) = \frac{\exp(q_v^i)}{\sum_{u\in\mathcal{A}_t}\exp(q_u^{j(u)})}, \tag{4.2}$$

where $j(u)$ is the job of node u, and $\mathcal{A}_t$ is the set of nodes that can be scheduled at time t. Notice that $\mathcal{A}_t$ is known to the RL agent at each step, since the input DAGs tell exactly which stages are runnable. Here, $\mathcal{A}_t$ restricts which outputs are considered by the softmax operation. The whole operation is end-to-end differentiable.

Parallelism limit selection. Many existing schedulers set a static degree of parallelism for each job: e.g., Spark by default takes the number of executors as a command-line argument on job submission. Decima adapts a job's parallelism each time it makes a scheduling deci-

sion for that job, and varies the parallelism as different stages in the job become runnable or finish execution.

For each job i, Decima's policy network also computes a score $w_l^i \triangleq w(\mathbf{y}^i, \mathbf{z}, l)$ for assigning parallelism limit l to job i, using another score function $w(\cdot)$. Similar to stage selection, Decima applies a softmax operation on these scores to compute the probability of selecting a parallelism limit (Figure 4-8).

Importantly, Decima uses the same score function $w(\cdot)$ for all jobs and all parallelism limit values. This is possible because the score function takes the parallelism l as one of its inputs. Without using l as an input, we cannot distinguish between different parallelism limits, and would have to use separate functions for each limit. Since the number of possible limits can be as large as the number of executors, reusing the same score function significantly reduces the number of parameters in the policy network and speeds up training (Figure 4-18a).

Decima's action specifies *job-level* parallelism (e.g., ten total executors for the entire job), as opposed fine-grained stage-level parallelism. This design choice trades off granularity of control for a model that is easier to train. In particular, restricting Decima to job-level parallelism control reduces the space of scheduling policies that it must explore and optimize over during training.

However, Decima still maintains the expressivity to (indirectly) control stage-level parallelism. On each scheduling event, Decima picks a stage v, and new parallelism limit l_i for v's job i. The system then schedules executors to v until i's parallelism reaches the limit l_i. Through repeated actions with different parallelism limits, Decima can add desired numbers of executors to specific stages. For example, suppose job i currently has ten executors, four of which are working on stage v. To add two more executors to v, Decima, on a scheduling event, picks stage v with parallelism limit of 12. Our experiments show that Decima achieves the same performance with job-level parallelism as with fine-grained, stage-level parallelism choice, at substantially accelerated training (Figure 4-18a).

4.5.3 Training

The primary challenge for training Decima is how to train with continuous stochastic job arrivals. To explain the challenge, we first describe the RL algorithms used for training.

RL training proceeds in *episodes*. Each episode consists of multiple scheduling events, and each scheduling event includes one or more actions. Let T be the total number of actions in an episode (T can vary across different episodes), and t_k be the wall clock time of the k^{th}

action. To guide the RL algorithm, Decima gives the agent a *reward* r_k after each action based on its high-level scheduling objective. For example, if the objective is to minimize the average JCT, Decima penalizes the agent $r_k = -(t_k - t_{k-1})J_k$ after the k^{th} action, where J_k is the number of jobs in the system during the interval $[t_{k-1}, t_k)$. The goal of the RL algorithm is to minimize the expected time-average of the penalties: $\mathbb{E}\left[1/t_T \sum_{k=1}^{T}(t_k - t_{k-1})J_k\right]$. This objective minimizes the average number of jobs in the system, and hence, by Little's law [62, §5], it effectively minimizing the average JCT.

Decima uses a policy gradient algorithm for training. The main idea in policy gradient methods is to learn by performing gradient descent on the neural network parameters using the rewards observed during training. Notice that all of Decima's operations, from the graph neural network (§4.5.1) to the policy network (§4.5.2), are differentiable. For conciseness, we denote all of the parameters in these operations jointly as θ, and the scheduling policy as $\pi_\theta(s_t, a_t)$ — defined as the probability of taking action a_t in state s_t.

Consider an episode of length T, where the agent collects (*state, action, reward*) observations, i.e., (s_k, a_k, r_k), at each step k. The agent updates the parameters θ of its policy $\pi_\theta(s_t, a_t)$ using the REINFORCE policy gradient algorithm [317]:

$$\theta \leftarrow \theta + \alpha \sum_{k=1}^{T} \nabla_\theta \log \pi_\theta(s_k, a_k) \left(\sum_{k'=k}^{T} r_{k'} - b_k \right). \tag{4.3}$$

Here, α is the learning rate and b_k is a *baseline* used to reduce the variance of the policy gradient [316]. An example of a baseline is a "time-based" baseline [201, 125], which sets b_k to the cumulative reward from step k onwards, averaged over all training episodes. Intuitively, $(\sum_{k'} r_{k'} - b_k)$ estimates how much better (or worse) the total reward is (from step k onwards) in a particular episode compared to the average case; and $\nabla_\theta \log \pi_\theta(s_k, a_k)$ provides a direction in the parameter space to increase the probability of choosing action a_k at state s_k. As a result, the net effect of this equation is to increase the probability of choosing an action that leads to a better-than-average reward.[6]

Challenge #1: Training with continuous job arrivals. To learn a robust scheduling policy, the agent has to experience "streaming" scenarios, where jobs arrive continuously over time, during training. Training with "batch" scenarios, where all jobs arrive at the beginning of an

[6] The update rule in Eq. (4.3) aims to maximize the sum of rewards during an episode. To maximize the time-average of the rewards, Decima uses the average reward formulation of this equation. See Appendix A for details.

episode, leads to poor policies in streaming settings (e.g., see Figure 4-17). However, training with a continuous stream of job arrivals is non-trivial. In particular, the agent's initial policy is very poor (e.g., as the initial parameters are random). Therefore, the agent cannot schedule jobs as quickly as they arrive in early training episodes, and a large queue of jobs builds up in almost every episode. Letting the agent explore beyond a few steps in these early episodes wastes training time, because the overloaded cluster scenarios it encounters will not occur with a reasonable policy.

To avoid this waste, we terminate initial episodes early so that the agent can reset and quickly try again from an idle state. We gradually increase the episode length throughout the training process. Thus, initially, the agent learns to schedule short sequences of jobs. As its scheduling policy improves, we increase the episode length, making the problem more challenging. The concept of gradually increasing job sequence length — and therefore, problem complexity — during training realizes curriculum learning [41] for cluster scheduling.

One subtlety about this method is that the termination cannot be deterministic. Otherwise, the agent can learn to predict when an episode terminates, and defer scheduling certain large jobs until the termination time. This turns out to be the optimal strategy over a fixed time horizon: since the agent is not penalized for the remaining jobs at termination, it is better to strictly schedule short jobs even if it means starving some large jobs. We found that this behavior leads to indefinite starvation of some jobs at runtime (where jobs arrive indefinitely). To prevent this behavior, we use a *memoryless* termination process. Specifically, we terminate each training episode after a time τ, drawn randomly from an exponential distribution. As explained above, the mean episode length increases during training up to a large value (e.g., a few hundreds of job arrivals on average).

Challenge #2: Variance caused by stochastic job arrivals. Next, for a policy to generalize well in a streaming setting, the training episodes must include many different job arrival patterns. This creates a new challenge: different job arrival patterns have a large impact on performance, resulting in vastly different rewards. Consider, for example, a scheduling action at the time t shown in Figure 4-9. If the arrival sequence following this action consists of a burst of large jobs (e.g., job sequence 1), the job queue will grow large, and the agent will incur large penalties. On the other hand, a light stream of jobs (e.g., job sequence 2) will lead to short queues and small penalties. The problem is that this difference in reward has nothing to do with the action at time t — it is caused by the randomness in the job arrival process. Since the RL algorithm uses the reward to assess the goodness of the action, such variance

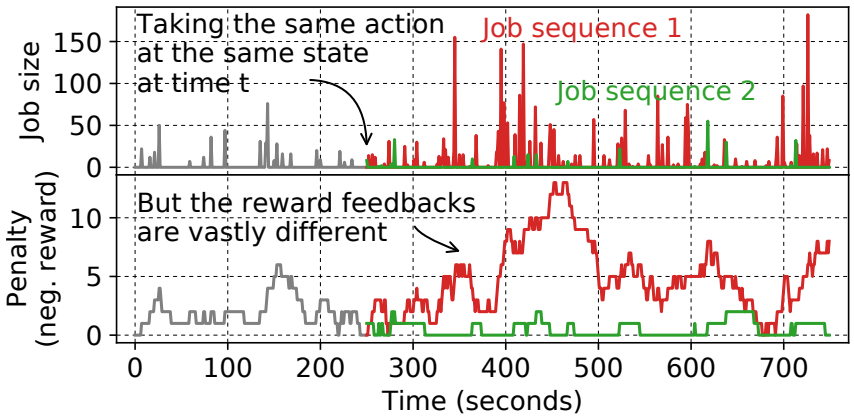


Figure 4-9: Illustrative example of how different job arrival sequences can lead to vastly different rewards. After time t, we sample two job arrival sequences, from a Poisson arrival process (10 seconds mean inter-arrival time) with randomly-sampled TPC-H queries.

adds noise and impedes effective training.

To resolve this problem, we build upon a recently-proposed variance reduction technique for "input-driven" environments [207], where an exogenous, stochastic input process (e.g., Decima's job arrival process) affects the dynamics of the system. The main idea is to fix the *same* job arrival sequence in multiple training episodes, and to compute separate baselines specifically for each arrival sequence. In particular, instead of computing the baseline b_k in Eq. (4.3) by averaging over episodes with different arrival sequences, we average only over episodes with the same arrival sequence. During training, we repeat this procedure for a large number of randomly-sampled job arrival sequences (§4.7.2 and §4.7.3 describe how we generate the specific datasets for training). This method removes the variance caused by the job arrival process entirely, enabling the policy gradient algorithm to assess the goodness of different actions much more accurately (see Figure 4-17). For the implementation details of our training and the hyperparameter settings used, see Appendix A.

4.6 Implementation

We have implemented Decima as a pluggable scheduling service that parallel data processing platforms can communicate with over an RPC interface. In §4.6.1, we describe the integration of Decima with Spark. Next, we describe our Python-based training infrastructure which includes an accurate Spark cluster simulator (§4.6.2).

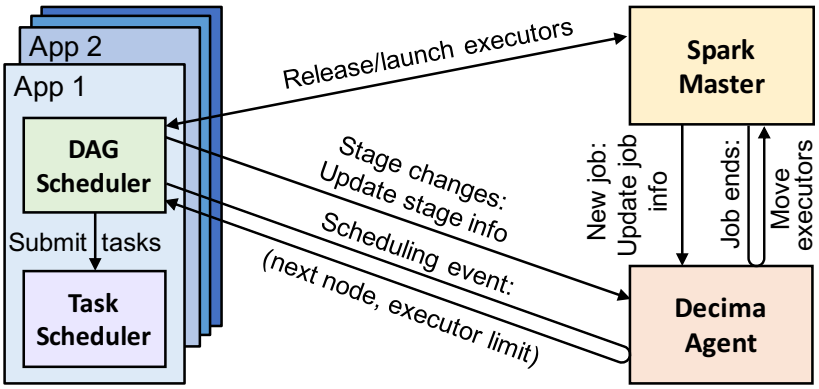


Figure 4-10: Spark standalone cluster architecture, with Decima additions highlighted.

4.6.1 Spark Integration

A Spark cluster[7] runs multiple parallel *applications*, which contain one or more jobs that together form a DAG of processing stages. The Spark master manages application execution and monitors the health of many *workers*, which each split their resources between multiple executors. Executors are created for, and remain associated with, a specific application, which handles its own scheduling of work to executors. Once an application completes, its executors terminate. Figure 4-10 illustrates this architecture.

To integrate Decima in Spark, we made two major changes:

1. Each application's **DAG scheduler** contacts Decima on startup and whenever a scheduling event occurs. Decima responds with the next stage to work on and the parallelism limit (§4.5.2).
2. The Spark **master** contacts Decima when a new job arrives to determine how many executors to launch for it, and aids Decima by taking executors away from a job once they complete a stage.

State observations. In Decima, the feature vector $\mathbf{x}_v^i$ (§4.5.1) of a node v in job DAG i consists of: (1) the number of tasks remaining in the stage, (2) the average task duration, (3) the number of executors currently working on the node, (4) the number of available executors, and (5) whether available executors are local to the job. We picked these features by attempting to include information necessary to capture the state of the cluster (e.g., the number of executors currently assigned to each stage), as well as the statistics that may help

[7] We discuss Spark's "standalone" mode of operation here (http://spark.apache.org/docs/latest/spark-standalone.html); YARN-based deployments can, in principle, use Decima, but require modifying both Spark and YARN.

in scheduling decisions (e.g., a stage's average task duration). These statistics depend on the information available (e.g., profiles from past executions of the same job, or runtime metrics) and on the system used (here, Spark). Decima can easily incorporate additional signals.

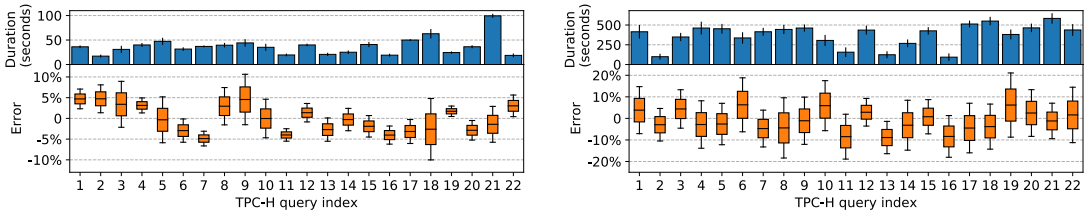
Neural network architecture. The graph neural network's six transformation functions $f(\cdot)$ and $g(\cdot)$ (§4.5.1) (two each for node-level, job-level, and global embeddings) and the policy network's two score functions $q(\cdot)$ and $w(\cdot)$ (§4.5.2) are implemented using two-hidden-layer neural networks, with 32 and 16 hidden units on each layer. Since these neural networks are reused for all jobs and all parallelism limits, Decima's model is lightweight — it consists of 12,736 parameters (50KB) in total. Mapping the cluster state to a scheduling decision takes less than 15ms (Figure 4-18b).

4.6.2 Spark Simulator

Decima's training happens offline using a faithful simulator that has access to profiling information (e.g., task durations) from a real Spark cluster (§4.7.2) and the job run time characteristics from an industrial trace (§4.7.3). To faithfully simulate how Decima's decisions interact with a cluster, our simulator captures several real-world effects:

1. The first "wave" of tasks from a particular stage often runs slower than subsequent tasks. This is due to Spark's pipelined task execution [239], JIT compilation [177] of task code, and warmup costs (e.g., making TCP connections to other executors). Decima's simulated environment thus picks the actual runtime of first-wave tasks from a different distribution than later waves.
2. Adding an executor to a Spark job involves launching a JVM process, which takes 2–3 seconds. Executors are tied to a job for isolation and because Spark assumes them to be long-lived. Decima's environment therefore imposes idle time reflecting the startup delay every time Decima moves an executor across jobs.
3. A high degree of parallelism can *slow down* individual Spark tasks, as wider shuffles require additional TCP connections and create more work when merging data from many shards. Decima's environment captures these effects by sampling task durations from distributions collected at different levels of parallelism if this data is available.

To validate the simulator's fidelity, we measured how simulated and real Spark differ in terms of job completion time for ten runs of TPC-H job sets (§4.7.2), both when jobs run alone and when they share a cluster with other jobs. Figure 4-11 shows the results: the



(a) Single job running in isolation.

(b) Mixture of jobs on a shared cluster.

Figure 4-11: Testing the fidelity of our Spark simulator with Decima as a scheduling agent. Blue bars in the upper part show the absolute real Spark job duration (error bars: standard deviation across ten experiments); the orange bars in the lower figures show the distribution of simulation error for a 95% confidence interval. The mean discrepancy between simulated and actual job duration is at most $\pm 5\%$ for isolated, single jobs, and the mean error for a mix of all 22 queries running on the cluster is at most $\pm 9\%$.

simulator closely matches the actual run time of each job, even when we run multiple jobs together in the cluster. In particular, the mean error of our simulation is within 5% of real runtime when jobs run in isolation, and within 9% when sharing a cluster (95^{th} percentile: $\leq 10\%$ in isolation, $\leq 20\%$ when sharing).

We found that capturing all first-order effects of the Spark environment is crucial to achieving this accuracy. For example, without modeling the delay to move an executor between jobs, the simulated runtime consistently underapproximates reality. Training in such an environment would result in a policy that moves executors more eagerly than is actually sensible (§4.7.4). Likewise, omitting the effects of initial and subsequent "waves" of tasks, or the slowdown overheads imposed with highDecima degrees of paralllism, significantly increases the variance in simulated runtime and makes it more difficult for Decima to learn a good policy.

4.7 Evaluation

We evaluated Decima on a real Spark cluster testbed and in simulations with a production workload from Alibaba. Our experiments address the following questions:

1. How does Decima perform compared to carefully-tuned heuristics in a real Spark cluster (§4.7.2)?
2. Can Decima's learning generalize to a multi-resource setting with different machine configurations (§4.7.3)?
3. How does each of our key ideas contribute to Decima's performance; how does Dec-

ima adapt when scheduling environments change; and how fast does Decima train and make scheduling decisions after training?

4.7.1 Existing Baseline Algorithms

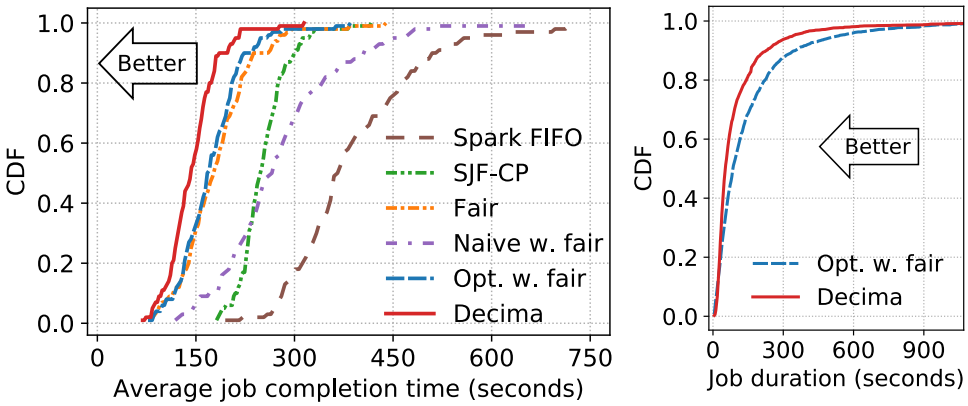
In our evaluation, we compare Decima's performance to that of seven baseline algorithms:

1. Spark's default FIFO scheduling, which runs jobs in the same order they arrive in and grants as many executors to each job as the user requested.
2. A shortest-job-first critical-path heuristic (SJF-CP), which prioritizes jobs based on their total work, and within each job runs tasks from the next stage on its critical path.
3. Simple fair scheduling, which gives each job an equal fair share of the executors and round-robins over tasks from runnable stages to drain all branches concurrently.
4. Naive weighted fair scheduling, which assigns executors to jobs proportional to their total work.
5. A carefully-tuned weighted fair scheduling that gives each job $T_i^\alpha / \sum_i T_i^\alpha$ of total executors, where T_i is the total work of each job i and α is a tuning factor. Notice that $\alpha = 0$ reduces to the simple fair scheme, and $\alpha = 1$ to the naive weighted fair one. We sweep through $\alpha \in \{-2, -1.9, ..., 2\}$ for the optimal factor.
6. The standard multi-resource packing algorithm from Tetris [121], which greedily schedules the stage that maximizes the dot product of the requested resource vector and the available resource vector.
7. Graphene*, an adaptation of Graphene [123] for Decima's discrete executor classes. Graphene* detects and groups "troublesome" nodes using Graphene's algorithm [123, §4.1], and schedules them together with optimally tuned parallelism as in (5), achieving the essence of Graphene's planning strategy. We perform a grid search to optimize for the hyperparameters (details in [206, Appendix F]).

4.7.2 Spark Cluster

We use an OpenStack cluster running Spark v2.2, modified as described in §4.6.1, in the Chameleon Cloud testbed. [8] The cluster consists of 25 worker VMs, each running two executors on an m1.xlarge instance (8 CPUs, 16 GB RAM) and a master VM on an m1.xxxlarge instance (16 CPUs, 32 GB RAM). Our experiments consider (1) *batched* arrivals, in which

[8] https://www.chameleoncloud.org



(a) Batched arrivals. **(b)** Continuous arrivals.

Figure 4-12: Decima's learned scheduling policy achieves 21%–3.1× lower average job completion time than baseline algorithms for batch and continuous arrivals of TPC-H jobs in a real Spark cluster.

multiple jobs start at the same time and run until completion, and (2) *continuous* arrivals, in which jobs arrive with stochastic interarrival distributions or follow a trace.

Batched arrivals. We randomly sample jobs from six different input sizes (2, 5, 10, 20, 50, and 100 GB) and all 22 TPC-H [295] queries, producing a heavy-tailed distribution: 23% of the jobs contain 82% of the total work. A combination of 20 random jobs (unseen in training) arrives as a batch, and we measure their average JCT.

Figure 4-12a shows a cumulative distribution of the average JCT over 100 experiments. There are three key observations from the results. First, SJF-CP and fair scheduling, albeit simple, outperform the FIFO policy by 1.6× and 2.5× on average. Importantly, the fair scheduling policies outperform SJF-CP since they work on multiple jobs, while SJF-CP focuses all executors exclusively on the shortest job.

Second, perhaps surprisingly, unweighted fair scheduling outperforms fair scheduling weighted by job size ("naive weighted fair"). This is because weighted fair scheduling grants small jobs *fewer* executors than their fair share, slowing them down and increasing average JCT. Our tuned weighted fair heuristic ("opt. weighted fair") counters this effect by calibrating the weights for each job *on each experiment* (§4.7.1). The optimal α is usually around -1, i.e., the heuristic sets the number of executors inversely proportional to job size. This policy effectively focuses on small jobs early on, and later shifts to running large jobs in parallel; it outperforms fair scheduling by 11%.

Finally, Decima outperforms all baseline algorithms and improves the average JCT by 21% over the closest heuristic ("opt. weighted fair"). This is because Decima prioritizes jobs

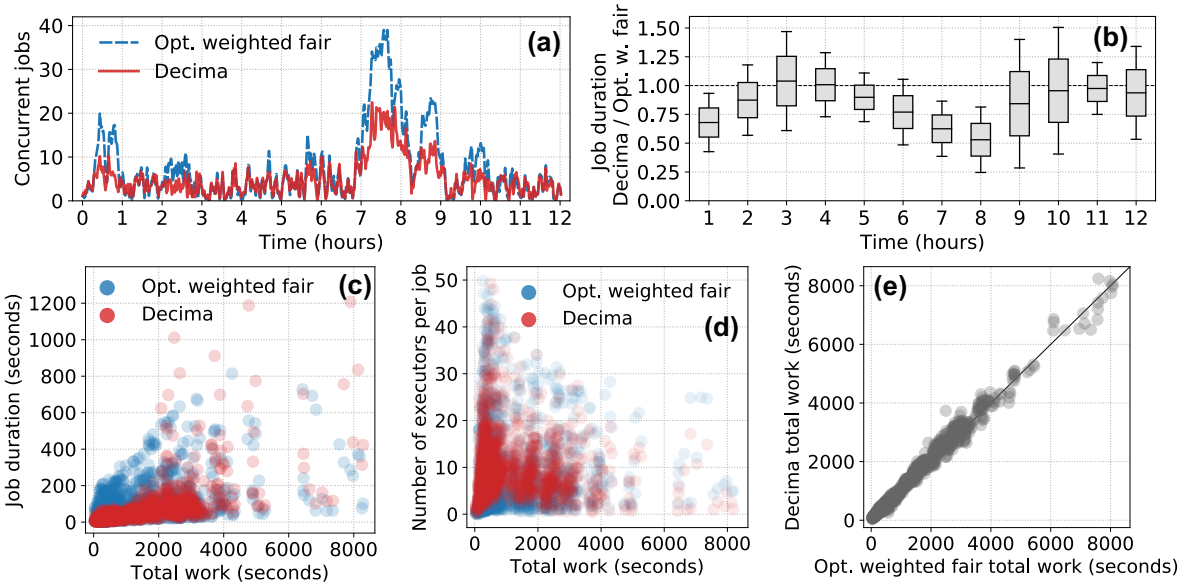
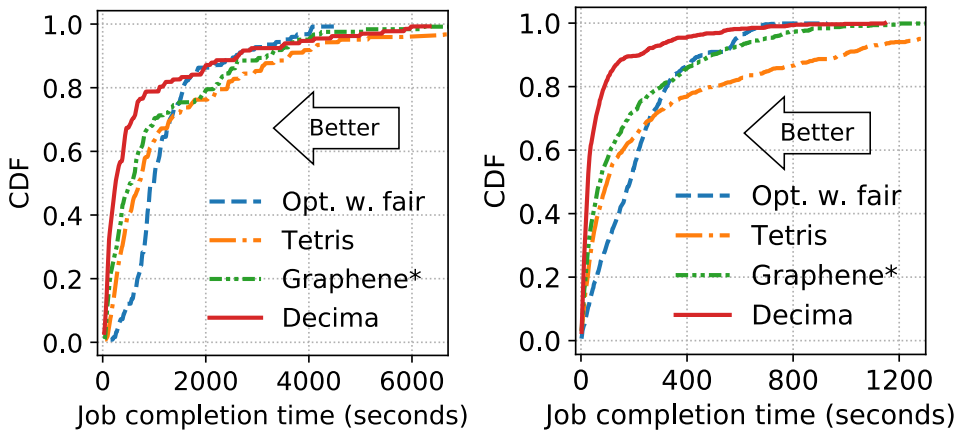


Figure 4-13: Time-series analysis (a, b) of continuous TPC-H job arrivals to a Spark cluster shows that Decima achieves most performance gains over heuristics during busy periods (e.g., runs jobs $2\times$ faster during hour 8), as it appropriately prioritizes small jobs (c) with more executors (d), while preventing work inflation (e).

better, assigns efficient executor shares to different jobs, and leverages the job DAG structure (§4.7.4 breaks down the benefit of each of these factors). Decima autonomously learns this policy through end-to-end RL training, while the best-performing baseline algorithms required careful tuning.

Continuous arrivals. We sample 1,000 TPC-H jobs of six different sizes uniformly at random, and model their arrival as a Poisson process with an average interarrival time of 45 seconds. The resulting cluster load is about 85%. At this cluster load, jobs arrive faster than most heuristic-based scheduling policies can complete them. Figure 4-12b shows that Decima outperforms the only baseline algorithm that can keep up ("opt. weighted fair"); Decima's average JCT is 29% lower. In particular, Decima shines during busy, high-load periods, where scheduling decisions have a much larger impact than when cluster resources are abundant. Figure 4-13a shows that Decima maintains a lower concurrent job count than the tuned heuristic particularly during the busy period in hours 7–9, where Decima completes jobs about $2\times$ faster (Figure 4-13b). Performance under high load is important for batch processing clusters, which often have long job queues [255], and periods of high load are when good scheduling decisions have the most impact (e.g., reducing the overprovisioning required for workload peaks).



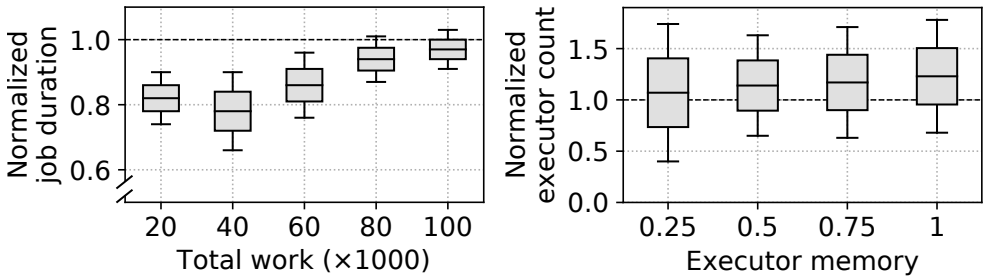
(a) Industrial trace replay. **(b)** TPC-H workload.

Figure 4-14: With multi-dimensional resources, Decima's scheduling policy outperforms Graphene* by 32% to 43% in average JCT.

Decima's performance gain comes from finishing small jobs faster, as the concentration of red points in the lower-left corner of Figure 4-13c shows. Decima achieves this by assigning more executors to the small jobs (Figure 4-13d). The right number of executors for each job is workload-dependent: indiscriminately giving small jobs more executors would use cluster resources inefficiently (§4.2.2). For example, SJF-CP's strictly gives all available executors to the smallest job, but this inefficient use of executors inflates total work, and SJF-CP therefore accumulates a growing backlog of jobs. Decima's executor assignment, by contrast, results in similar total work as with the hand-tuned heuristic. Figure 4-13e shows this: jobs below the diagonal have smaller total work with Decima than with the heuristic, and ones above have larger total work in Decima. Most small jobs are on the diagonal, indicating that Decima only increases the parallelism limit when extra executors are still efficient. Consequently, Decima successfully balances between giving small jobs extra resources to finish them sooner and using the resources efficiently.

4.7.3 Multi-Dimensional Resource Packing

The standalone Spark scheduler used in our previous experiments only provides jobs with access to predefined executor slots. More advanced cluster schedulers, such as YARN [302] or Mesos [141], allow jobs to specify their tasks' resource requirements and create appropriately-sized executors. Packing tasks with multi-dimensional resource needs (e.g., ⟨CPU, memory⟩) onto fixed-capacity servers adds further complexity to the scheduling problem [121, 123].



(a) Job duration grouped by total work, Decima normalized to Graphene*.

(b) Number of executors that Decima uses for "small" jobs, normalized to Graphene*.

Figure 4-15: Decima outperforms Graphene* with multi-dimensional resources by (a) completing small jobs faster and (b) use "oversized" executors for small jobs (smallest 20% in total work).

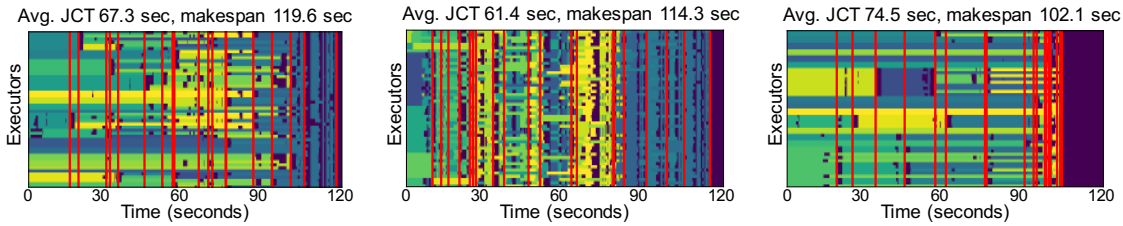
We use a production trace from Alibaba to investigate if Decima can learn good multi-dimensional scheduling policies with the same core approach.

Industrial trace. The trace contains about 20,000 jobs from a production cluster. Many jobs have complex DAGs: 59% have four or more stages, and some have hundreds. We run the experiments using our simulator (§4.6.2) with up to 30,000 executors. This parameter is set according to the maximum number of concurrent tasks in the trace. We use the first half of the trace for training and then compare Decima's performance with other schemes on the remaining portion.

Multi-resource environment. We modify Decima's environment to provide several discrete executor *classes* with different memory sizes. Tasks now require a minimum amount of CPU and memory, i.e., a task must fit into the executor that runs it. Tasks can run in executors larger than or equal to their resource request. Decima now chooses a DAG stage to schedule, a parallelism level, and an executor class to use. Our experiments use four executor types, each with 1 CPU core and $(0.25, 0.5, 0.75, 1)$ unit of normalized memory; each executor class makes up 25% of total cluster executors.

Results. We run simulated multi-resource experiments on continuous job arrivals according to the trace. Figure 4-14a shows the results for Decima and three other algorithms: the optimally tuned weighted-fair heuristic, Tetris, and Graphene*. Decima achieves a 32% lower average JCT than the best competing algorithm (Graphene*), suggesting that it learns a good policy in the multi-resource environment.

Decima's policy is qualitatively different to Graphene*'s. Figure 4-15a breaks Decima's improvement over Graphene* down by jobs' total work. Decima completes jobs faster than



(a) Average JCT objective. **(b)** Avg. JCT, with zero-cost executor motion. **(c)** Minimal makespan objective.

Figure 4-16: Decima learns qualitatively different policies depending on the environment (e.g., costly (a) vs. free executor migration (b)) and the objective (e.g., average JCT (a) vs. makespan (c)). Vertical red lines indicate job completions, colors indicate tasks in different jobs, and dark purple is idle time.

Graphene* for all job sizes, but its gain is particularly large for small jobs. The reason is that Decima learns to use "oversized" executors when they can help finish nearly-completed small jobs when insufficiently many right-sized executors are available. Figure 4-15b illustrates this: Decima uses 39% more executors of the largest class on the jobs with smallest 20% total work. In other words, Decima trades off memory fragmentation against clearing the job queue more quickly. This trade-off makes sense because small jobs (1) contribute more to the average JCT objective, and (2) only fragment resources for a short time. By contrast, Tetris greedily packs tasks into the best-fitting executor class and achieves the lowest memory fragmentation. Decima's fragmentation is within 4%–13% of Tetris's, but Decima's average JCT is 52% lower, as it learns to balance the trade-off well. This requires respecting workload-dependent factors, such as the DAG structure, the threshold for what is a "small" job, and others. Heuristic approaches like Graphene* attempt to balance those factors via additive score functions and extensive tuning, while Decima learns them without such inputs.

We also repeat this experiment with the TPC-H workload, using 200 executors and sampling each TPC-H DAG node's memory request from $(0,1]$. Figure 4-14b shows that Decima outperforms the competing algorithms by even larger margins (e.g., 43% over Graphene*). This is because the industrial trace lacks work inflation measurements for different levels of parallelism, which we provide for TPC-H. Decima learns to use this information to further calibrate executor assignment.

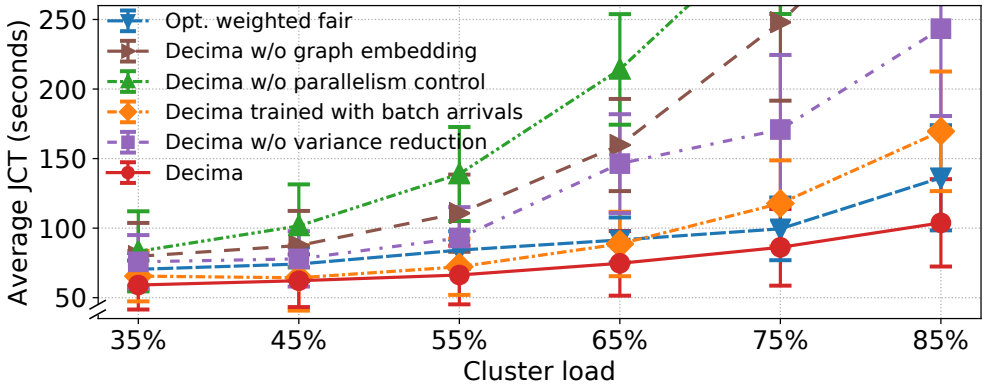


Figure 4-17: Breakdown of each key idea's contribution to Decima with continuous job arrivals. Omitting any concept increases Decima's average JCT above that of the weighted fair policy.

4.7.4 Decima Deep Dive

Finally, we demonstrate the wide range of scheduling policies Decima can learn, and break down the impact of our key ideas and techniques on Decima's performance. In addition, we further evaluate Decima's optimality via an exhaustive search of job orderings, the robustness of its learned policies to changing environments, and Decima's sensitivity to incomplete information.

Learned policies. Decima outperforms other algorithms because it can learn different policies depending on the high-level objective, the workload, and environmental conditions. When Decima optimizes for average JCT (Figure 4-16a), it learns to share executors for small jobs to finish them quickly and avoids inefficiently using too many executors on large jobs (§4.7.2). Decima also keeps the executors working on tasks from the same job to avoid the overhead of moving executors (§4.6.1). However, if moving executors between jobs is free — as is effectively the case for long tasks, or for systems without JVM spawn overhead — Decima learns a policy that eagerly moves executors among jobs (cf. the frequent color changes in Figure 4-16b). Finally, given a different objective of minimizing the overall *makespan* for a batch of jobs, Decima learns yet another different policy (Figure 4-16c). Since only the *final* job's completion time matters for a makespan objective, Decima no longer works to finish jobs early. Instead, many jobs complete together at the end of the batched workload, which gives the scheduler more choices of jobs throughout the execution, increasing cluster utilization.

Impact of learning architecture. We validate that Decima uses all raw information provided in the state and requires all its key design components by selectively omitting compo-

Setup (IAT: interarrival time)	Average JCT [sec]
Opt. weighted fair (best heuristic)	91.2 ± 23.5
Decima, trained on test workload (IAT: 45 sec)	65.4 ± 28.7
Decima, trained on anti-skewed workload (IAT: 75 sec)	104.8 ± 37.6
Decima, trained on mixed workloads	82.3 ± 31.2
Decima, trained on mixed workloads with interarrival time hints	76.6 ± 33.4

Table 4.2: Decima generalizes to changing workloads. For an unseen workload, Decima outperforms the best heuristic by 10% when trained with a mix of workloads; and by 16% if it knows the interarrival time from an input feature.

nents. We run 1,000 continuous TPC-H job arrivals on a simulated cluster at different loads, and train five different variants of Decima on each load.

Figure 4-17 shows that removing any one component from Decima results in worse average JCTs than the tuned weighted-fair heuristic at a high cluster load. There are four takeaways from this result. First, parallelism control has the greatest impact on Decima's performance. Without parallelism control, Decima assigns all available executors to a single stage at every scheduling event. Even at a moderate cluster load (e.g., 55%), this leads to an unstable policy that cannot keep up with the arrival rate of incoming jobs. Second, omitting the graph embedding (i.e., directly taking raw features on each node as input to the score functions in §4.5.2) makes Decima unable to estimate remaining work in a job and to account for other jobs in the cluster. Consequently, Decima has no notion of small jobs or cluster load, and its learned policy quickly becomes unstable as the load increases. Third, using unfixed job sequences across training episodes increases the variance in the reward signal (§4.5.3). As the load increases, job arrival sequences become more varied, which increases variance in the reward. At cluster load larger than 75%, reducing this variance via synchronized termination improves average JCT by $2\times$ when training Decima, illustrating that variance reduction is key to learning high-quality policies in long-horizon scheduling problems. Fourth, training only on batched job arrivals cannot generalize to continuous job arrivals. When trained on batched arrivals, Decima learns to systematically defer large jobs, as this results in the lowest sum of JCTs (lowest sum of penalties). With continuous job arrivals, this policy starves large jobs indefinitely as the cluster load increases and jobs arrive more frequently. Consequently, Decima underperforms the tuned weighted-fair heuristic at loads above 65% when trained on batched arrivals.

Generalizing to different workloads. We test Decima's ability to generalize by changing the training workload in the TPC-H experiment (§4.7.2). To simulate shifts in cluster workload, we train models for different job interarrival times between 42 and 75 seconds, and

Decima training scenario	average JCT (seconds)
Decima trained with test setting	3,290±680
Decima trained with 15× fewer jobs	3,540±450
Decima trained with test setting	610±90
Decima trained with 10× fewer executors	630±70

Table 4.3: Decima generalizes well to deployment scenarios in which the workload or cluster differ from the training setting. The test setting has 150 jobs and 10k executors.

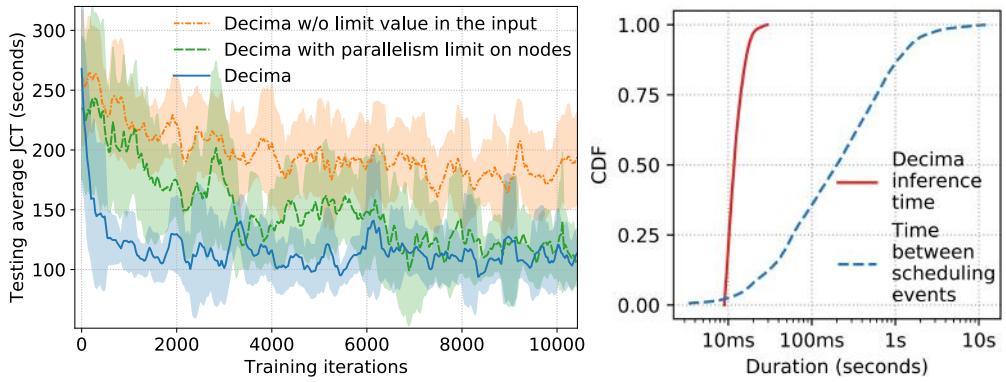
test them using a workload with a 45 second interarrival time. As Decima learns workload-specific policies, we expect its effectiveness to depend on whether broad test workload characteristics, such as interarrival time and job size distributions, match the training workload.

Table 4.2 shows the resulting average JCT. Decima performs well when trained on a workload similar to the test workload. Unsurprisingly, when Decima trains with an "anti-skewed" workload (75 seconds interarrival time), it generalizes poorly and underperforms the optimized weighted fair policy. This makes sense because Decima incorporates the learned interarrival time distribution in its policy.

When training with a mixed set of workloads that cover the whole interarrival time range, Decima can learn a more general policy. This policy fits less strongly to a specific interarrival time distribution and therefore becomes more robust to workload changes. If Decima can observe the interarrival time as a feature in its state (§4.6.1), it generalizes better still and learns an adaptive policy that achieves a 16% lower average JCT than the best heuristic. These results highlight that a diverse training workload set helps make Decima's learned policies robust to workload shifts; we discuss possible online learning in §4.8.

Generalizing to different environments. Real-world cluster workloads vary over time, and the available cluster machines can also change. Ideally, Decima would generalize from a model trained for a specific load and cluster size to similar workloads with different parameters. To test this, we train a Decima agent on a scaled-down version of the industrial workload, using 15× fewer concurrent jobs and 10× fewer executors than in the test setting.

Table 4.3 shows how the performance of this agent compares with that of one trained on the real workload and cluster size. Decima is robust to changing parameters: the agent trained with 15× fewer jobs generalizes to the test workload with a 7% reduced average JCT, and an agent trained on a 10× smaller cluster generalizes with a 3% reduction in average JCT. Generalization to a larger cluster is robust as the policy correctly limits jobs' parallelism even if vastly more resources are available. By contrast, generalizing to a workload with many more



(a) Learning curve. **(b)** Scheduling delay.

Figure 4-18: Different encodings of jobs parallelism (§4.5.2) affect Decima's training time. Decima makes low-latency scheduling decisions: on average, the latency is about 50× smaller than the interval between scheduling events.

jobs is harder, as the smaller-scale training lacks experiences with complex job combinations.

Training and inference performance. Figure 4-18a shows Decima's learning curve (in blue) on continuous TPC-H job arrivals (§4.7.2), testing snapshots of the model every 100 iterations on (unseen) job arrival sequences. Each training iteration takes about 5 seconds. Decima's design (§4.5.3) is crucial for training efficiency: omitting the parallelism limit values in the input (yellow curve) forces Decima to use separate score functions for different limits, significantly increasing the number of parameters to optimize over; putting fine-grained parallelism control on nodes (green curve) slows down training as it increases the space of algorithms Decima must explore.

Figure 4-18b shows cumulative distributions of the time Decima takes to decide on a scheduling action (in red) and the time interval between scheduling events (in blue) in our Spark testbed (§4.7.2). The average scheduling delay for Decima is less than 15ms, while the interval between scheduling events is typically in the scale of seconds. In less than 5% of the cases, the scheduling interval is shorter than the scheduling delay (e.g., when the cluster requests for multiple scheduling actions in a single scheduling event). Thus Decima's scheduling delay imposes no measurable overhead on task runtimes.

Optimality of Decima In §4.7, we show Decima is able to rival or outperform existing scheduling schemes in a wide range of complex cluster environments, including a real Spark testbed, real-world cluster trace simulations and a multi-resource packing environment. However, the optimality of Decima in those environments remains unknown due to the intractability of computing exact optimal scheduling solutions [211, 123], or tight lower

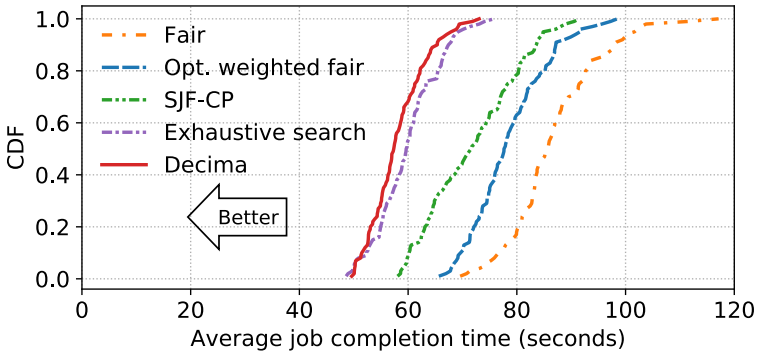


Figure 4-19: Comparing Decima with near optimal heuristics in a simplified scheduling environment.

bounds.[9] To nevertheless get an idea of how close Decima comes to an optimal scheduler, we test Decima in simplified settings where a brute-force search over different schedules is possible.

We consider the Spark scheduling framework simulated in §4.6.2 with an average JCT objective for a batch of jobs. To simplify the environment, we turn off the "wave" effect, executor startup delays and the artifact of task slowdowns at high degrees of parallelism. As a result, the duration of a stage has a strict inverse relation to the number of executors the stage runs on (i.e., it scales linearly with parallel resources), and the scheduler is free to move executors across jobs without any overhead. The dominating challenges in this environment are to pack jobs tightly and to favor short jobs as much as possible.

To find a good schedule for a batch of n jobs, we exhaustively search all $n!$ possible job orderings, and select the ordering with the lowest average JCT. To make the exhaustive search feasible, we consider a batch of ten jobs. For each job ordering, we select the unfinished job appearing earliest in the order at each scheduling event (§4.5.2), and use the DAG's critical path to choose the order in which to finish stages within each job. By considering all possible job orderings, the algorithm is guaranteed to consider, amongst other schedules, a strict shortest-job-first (SJF) schedule that yields a small average JCT. We believe this policy to be close to the optimal policy, as we have empirically observed that job orderings dominate the average JCT in TPC-H workloads (§4.7.4). However, the exhaustive search also explores variations of the SJF schedule, e.g., orders that prioritize jobs which can exploit parallelism to complete more quickly than less-parallelizable jobs that contain smaller total work.

Next, we train an unmodified Decima agent in this environment, similar to the setup

[9] In our setting (i.e., Spark's executor-based scheduling), we found lower bounds based on total work or the critical path to be too loose to provide meaningful information.

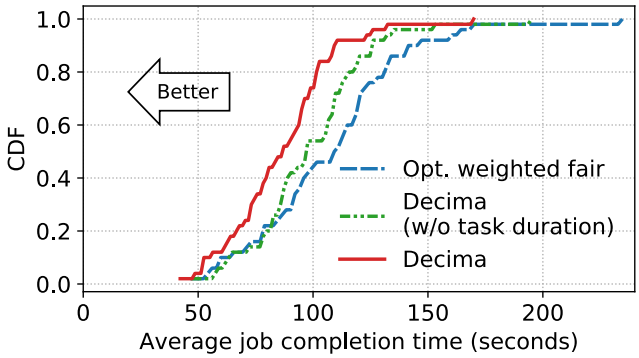


Figure 4-20: Decima performs worse on unseen jobs without task duration estimates, but still outperforms the best heuristic.

in §4.7.2. We compare this agent's performance with our exhaustive search baseline, a shortest-job-first critical-path heuristic, and the tuned weighted fair scheduler (described in §4.7.2).

Figure 4-19 shows the results. We make three key observations. First, unlike in the real Spark cluster (Figure 4-12), the SJF-CP scheme outperforms the tuned weighted fair scheduler. This meets our expectation because SJF-CP strictly favors small jobs to minimize the average JCT, which in the absence of the complexities of a real-world cluster is a good policy. Second, the exhaustive search heuristic performs better than SJF-CP. This is because SJF-CP strictly focuses on completing the job with the smallest total work first, ignoring the DAG structure and the potential parallelism it implies. The exhaustive search, by contrast, finds job orderings that prioritize jobs which can execute most quickly given the available executors on the cluster, their DAG structure, and their total work. While the search algorithm is not aware of these constraints, by trying out different job orderings, it finds the schedule that both orders jobs correctly *and* exploits cluster resources to complete the jobs as quickly as possible. Third, Decima matches the average JCT of the exhaustive search or even outperforms it slightly (by 9% on average). We found that Decima is better at dynamically prioritizing jobs based on their current structure at runtime (e.g., how much work remains on each dependency path), while the exhaustive search heuristic strictly follows the order determined in an offline static search and only controls when jobs start. This experiment shows that Decima is able to automatically learn a scheduling algorithm that performs as well as an offline-optimal job order.

Decima with missing information. In a real cluster, Decima will occasionally encounter unseen jobs without reliable task duration estimates. Unlike heuristics that fundamentally

rely on profiling information (e.g., weighted fair scheduling based on total work), Decima can still work with the remaining information and extract a reasonable scheduling policy.

Running the same setting as in §4.7.2, Figure 4-20 shows that training without task durations yields a policy that still outperforms the best heuristic, as Decima can still exploit the graph structure and other information such as the correlation between number of tasks and the efficient parallelism level.

4.8 Discussion

In this section, we discuss future research directions and other potential applications for Decima's techniques.

Robustness and generalization. Our experiments in §4.7.4 showed that Decima can learn generalizable scheduling policies that work well on an unseen workload. However, more drastic workload changes than interarrival time shifts could occur. To increase robustness of a scheduling policy against such changes, it may be helpful to train the agent on worst-case situations or adversarial workloads, drawing on the emerging literature on robust adversarial RL [247]. Another direction is to adjust the scheduling policy *online* as the workload changes. The key challenge with an online approach is to reduce the large sample complexity of model-free RL when the workload changes quickly. One viable approach might be to use meta learning [101, 89, 72], which allows training a "meta" scheduling agent that is designed to adapt to a specific workload with only a few observations.

Other learning objectives. In our experiments, we evaluated Decima on metrics related to job duration (e.g., average JCT, makespan). Shaping the reward signal differently can steer Decima to meet other objectives, too. For example, imposing a hard penalty whenever the deadline of a job is missed would guide Decima to a deadline-aware policy. Alternatively, basing the reward on e.g., the 90th percentile of empirical job duration samples, Decima can optimize for a tight tail of the JCT distribution. Addressing objectives formulated as constrained optimization (e.g., to minimize average JCT, but strictly guarantee fairness) using RL is an interesting further direction [3, 112].

Preemptive scheduling. Decima currently never preempts running tasks and can only remove executors from a job after a stage completes. This design choice keeps the MDP tractable for RL and results in effective learning and strong scheduling policies. However, future work might investigate more fine-grained and reactive preemption in an RL-driven

scheduler such as Decima. Directly introducing preemption would lead to a much larger action space (e.g., specifying arbitrary set of executors to preempt) and might require a much higher decision-making frequency. To make the RL problem tractable, one potential research direction is to leverage multi-agent RL [232, 128, 190]. For example, a Decima-like scheduling agent might controls which stage to run next and how many executors to assign, and, concurrently, another agent might decide where to preempt executors.

Potential networking and system applications. Some techniques we developed for Decima are broadly applicable to other networking and computer systems problems. For example, the scalable representation of input DAGs (§4.5.1) has applications in problems over graphs, such as database query optimization [210] and hardware device placement [5]. Our variance reduction technique (§4.5.3) generally applies to systems with stochastic, unpredictable inputs [207, 202].

4.9 Related Work

There is little prior work on applying machine learning techniques to cluster scheduling. DeepRM [201], which uses RL to train a neural network for multi-dimensional resource packing, is closest to Decima in aim and approach. However, DeepRM only deals with a basic setting in which each job is a single task and was evaluated in simple, simulated environments. DeepRM's learning model also lacks support for DAG-structured jobs, and its training procedure cannot handle realistic cluster workloads with continuous job arrivals. In other applications, Mirhoseini et al.'s work on learning device placement in TensorFlow (TF) computations [218] also uses RL, but relies on recurrent neural networks to scan through all nodes for state embedding, rather than a graph neural network. Their approach use recurrent neural networks to scan through all nodes for state embedding instead of using a scalable graph neural network. The objective there is to schedule a single TF job well, and the model cannot generalize to unseen job combinations [216].

Prior work in machine learning and algorithm design has combined RL and graph neural networks to optimize complex combinatorial problems, such as vertex set cover and the traveling salesman problem [73, 189]. The design of Decima's scalable state representation is inspired by this line of work, but we found that off-the-shelf graph neural networks perform poorly for our problem. To train strong scheduling agents, we had to change the graph neural network architecture to enable Decima to compute, amongst other metrics, the critical path

of a DAG (§4.5.1).

For resource management systems more broadly, Paragon [78] and Quasar [80] use collaborative filtering to match workloads to different machine types and avoid interference; their goal is complementary to Decima's. Tetrisched [297], like Decima, plans ahead in time, but uses a constraint solver to optimize job placement and requires the user to supply explicit constraints with their jobs. Firmament [119] also uses a constraint solver and achieves high-quality placements, but requires an administrator to configure an intricate scheduling policy. Graphene [123] uses heuristics to schedule job DAGs, but cannot set appropriate parallelism levels. Some systems "auto-scale" parallelism levels to meet job deadlines [100] or opportunistically accelerate jobs using spare resources [271, §5]. Carbyne [122] allows jobs to "altruistically" give up some of their short-term fair share of cluster resources in order to improve JCT across jobs while guarantee long-term fairness. Decima learns policies similar to Carbyne's, balancing resource shares and packing for low average JCT, but the current design of Decima does not have fairness an objective.

General-purpose cluster managers like Borg [305], Mesos [141], or YARN [302] support many different applications, making workload-specific scheduling policies are difficult to apply at this level. However, Decima could run as a framework atop Mesos or Omega [271].

4.10 Conclusion

Decima demonstrates that automatically learning complex cluster scheduling policies using reinforcement learning is feasible, and that the learned policies are flexible and efficient. Decima's learning innovations, such as its graph embedding technique and the training framework for streaming, may be applicable to other systems processing DAGs (e.g., query optimizers). We open source Decima, our models, and our experimental infrastructure at `https://web.mit.edu/decima`.

Chapter 5

Variance Reduction for RL in Input-Driven Environments

5.1 Introduction

Deep reinforcement learning (RL) has emerged as a powerful approach for sequential decision-making problems, achieving impressive results in domains such as game playing [220, 279] and robotics [186, 268, 191]. In this chapter, we consider RL in *input-driven environments*. Informally, input-driven environments have dynamics that are partially dictated by an exogenous, stochastic *input process*. Queuing systems [171, 166] are an example; their dynamics are governed by not only the decisions made within the system (e.g., scheduling, load balancing) but also the arrival process that brings work (e.g., jobs, customers, packets) into the system. Input-driven environments also arise naturally in many other domains: network control and optimization [319, 204], robotics control with stochastic disturbances [247], locomotion in environments with complex terrains and obstacles [138], vehicular traffic control [39, 321], tracking moving targets, and more (see Figure 5-1).

We focus on model-free policy gradient RL algorithms [317, 219, 268], which have been widely adopted and benchmarked for a variety of RL tasks [88, 323]. A key challenge for these methods is the high variance in the gradient estimates, as such variance increases sample complexity and can impede effective learning [269, 219]. A standard approach to reduce variance is to subtract a "baseline" from the total reward (or "return") to estimate the policy gradient [316]. The most common choice of a baseline is the *value function* — the expected return starting from the state.

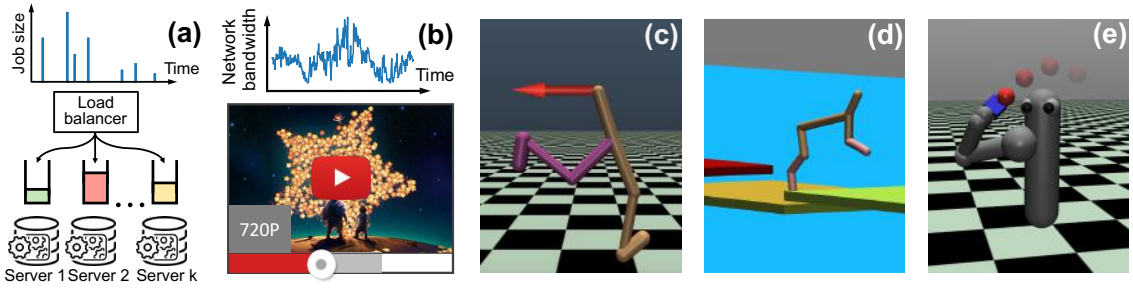


Figure 5-1: Input-driven environments: *(a)* load-balancing heterogeneous servers [133] with stochastic job arrival as the input process; *(b)* adaptive bitrate video streaming [204] with stochastic network bandwidth as the input process; *(c)* Walker2d in wind with a stochastic force (wind) applied to the walker as the input process; *(d)* HalfCheetah on floating tiles with the stochastic process that controls the buoyancy of the tiles as the input process; *(e)* 7-DoF arm tracking moving target with the stochastic target position as the input process. Environments *(c)–(e)* use the MuJoCo physics simulator [294].

Our main insight is that a state-dependent baseline — such as the value function — is a poor choice in input-driven environments, whose state dynamics and rewards are partially dictated by the input process. In such environments, comparing the return to the value function baseline may provide limited information about the quality of actions. The return obtained after taking a good action may be poor (lower than the baseline) if the input sequence following the action drives the system to unfavorable states; similarly, a bad action might end up with a high return with an advantageous input sequence. Intuitively, a good baseline for estimating the policy gradient should take the specific instance of the input process — the sequence of input values — into account. We call such a baseline an *input-dependent baseline*; it is a function of both the state and the entire future input sequence.

We formally define input-driven Markov decision processes, and we prove that an input-dependent baseline does not introduce bias in standard policy gradient algorithms such as Advantage Actor Critic (A2C) [219] and Trust Region Policy Optimization (TRPO) [268], provided that the input process is independent of the states and actions. We derive the optimal input-independent baseline and a simpler one to work with in practice; this takes the form of a *conditional value function* — the expected return given the state and the future input sequence.

Input-dependent baselines are harder to learn than their state-dependent counterparts; they are high-dimensional functions of the sequence of input values. To learn input-dependent baselines efficiently, we propose a simple approach based on meta-learning [101, 307]. The idea is to learn a "meta baseline" that can be specialized to a baseline for a specific input in-

stantiation using a small number of training episodes with that input. This approach applies to applications in which an input sequence can be repeated during training, e.g., applications that use simulations or experiments with previously-collected input traces for training [212].

We compare our input-dependent baseline to the standard value function baseline for the five tasks illustrated in Figure 5-1. These tasks are derived from queuing systems (load balancing heterogeneous servers [133]), computer networks (bitrate adaptation for video streaming [204]), and variants of standard continuous control RL benchmarks in the MuJoCo physics simulator [294]. We adapted three widely-used MuJoCo benchmarks [88, 71, 138] to add a stochastic input element that makes these tasks significantly more challenging. For example, we replaced the static target in a 7-DoF robotic arm target-reaching task with a randomly-moving target that the robot aims to track over time. Our results show that input-dependent baselines consistently provide improved training stability and better eventual policies. Input-dependent baselines are applicable to a variety of policy gradient methods, including A2C, TRPO, PPO, robust adversarial RL methods such as RARL [247], and meta-policy optimization such as MB-MPO [72]. Video demonstrations are available at https://sites.google.com/view/input-dependent-baseline/.

5.2 Preliminaries

Notation. We consider a discrete-time Markov decision process (MDP), defined by $(\mathcal{S},\mathcal{A},\mathcal{P},\rho_0,r,\gamma)$, where $\mathcal{S}\subseteq\mathbb{R}^n$ is a set of n-dimensional states, $\mathcal{A}\subseteq\mathbb{R}^m$ is a set of m-dimensional actions, $\mathcal{P}:\mathcal{S}\times\mathcal{A}\times\mathcal{S}\to[0,1]$ is the state transition probability distribution, $\rho_0:\mathcal{S}\to[0,1]$ is the distribution over initial states, $r:\mathcal{S}\times\mathcal{A}\to\mathbb{R}$ is the reward function, and $\gamma\in(0,1)$ is the discount factor. We denote a stochastic policy as $\pi:\mathcal{S}\times\mathcal{A}\to[0,1]$, which aims to optimize the expected return $\eta(\pi)=\mathbb{E}_\tau[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)]$, where $\tau=(s_0,a_0,...)$ is the trajectory following $s_0\sim\rho_0$, $a_t\sim\pi(a_t|s_t)$, $s_{t+1}\sim\mathcal{P}(s_{t+1}|s_t,a_t)$. We use $V_\pi(s_t)=\mathbb{E}_{a_t,s_{t+1},a_{t+1},...}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l},a_{r+l})|s_t\right]$ to define the value function, and $Q_\pi(s_t,a_t)=\mathbb{E}_{s_{t+1},a_{t+1},...}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l},a_{r+l})|s_t,a_t\right]$ to define the state-action value function. For any sequence $(x_0,x_1,...)$, we use $\boldsymbol{x}$ to denote the entire sequence and $x_{i:j}$ to denote $(x_i,x_{i+1},...,x_j)$.

Policy gradient methods. Policy gradient methods estimate the gradient of expected return with respect to the policy parameters [286, 161, 127]. To train a policy π_θ parameterized

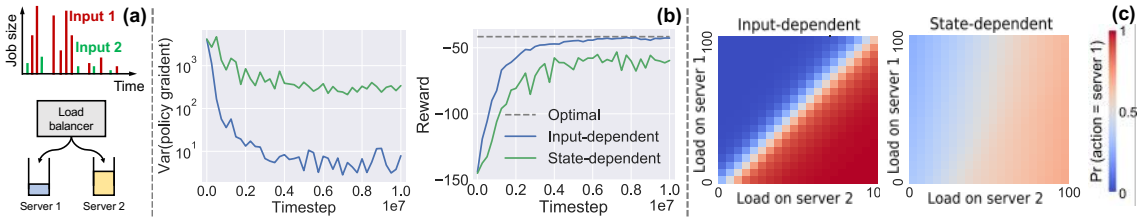


Figure 5-2: Load balancing over two servers. *(a)* Job sizes follow a Pareto distribution and jobs arrive as a Poisson process; the RL agent observes the queue lengths and picks a server for an incoming job. *(b)* The input-dependent baseline (blue) results in a 50× lower policy gradient variance (left) and a 33% higher test reward (right) than the standard, state-dependent baseline (green). *(c)* The probability heatmap of picking server 1 shows that using the input-dependent baseline (left) yields a more precise policy than using the state-dependent baseline (right).

by θ, the Policy Gradient Theorem [286] states that

$$\nabla_\theta \eta(\pi_\theta) = \mathbb{E}_{\substack{s\sim\rho_\pi \\ a\sim\pi_\theta}}[\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s,a)], \tag{5.1}$$

where $\rho_\pi(s) = \sum_{t=0}^{\infty}[\gamma^t \Pr(s_t = s)]$ denotes the discounted state visitation frequency. Practical algorithms often use the undiscounted state visitation frequency (i.e., $\gamma = 1$ in ρ_π), which can make the estimation slightly biased [292].

Estimating the policy gradient using Monte Carlo estimation for the Q function suffers from high variance [219]. To reduce variance, an appropriately chosen baseline $b(s_t)$ can be subtracted from the Q-estimate without introducing bias [125]. The policy gradient estimation with a baseline in Equation (5.1) becomes $\mathbb{E}_{\rho_\pi,\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)(Q_{\pi_\theta}(s,a) - b(s))]$. While an optimal baseline exists [125, 322], it is hard to estimate and often replaced by the value function $b(s_t) = V_\pi(s_t)$ [285, 219].

5.3 Motivating Example

We use a simple load balancing example to illustrate the variance introduced by an exogenous input process. As shown in Figure 5-2a, jobs arrive over time and a load balancing agent sends them to one of two servers. The jobs arrive according to a Poisson process, and the job sizes follow a Pareto distribution. The two servers process jobs from their queues at identical rates. On each job arrival, the load balancer observes state $s_t = (q_1, q_2)$, denoting the queue length at the two servers. It then takes an action $a_t \in \{1, 2\}$, sending the job to one of the servers. The goal of the load balancer is to minimize the average job completion time.

The reward corresponding to this goal is $r_t = -\tau \times j$, where τ is the time elapsed since the last action and j is total number of enqueued jobs.

In this example, the optimal policy is to send the job to the server with the shortest queue [74]. However, we find that a standard policy gradient algorithm, A2C [219], trained using a value function baseline struggles to learn this policy. The reason is that the stochastic sequence of job arrivals creates huge variance in the reward signal, making it difficult to distinguish between good and bad actions. Consider, for example, an action at the state shown in Figure 5-2a. If the arrival sequence following this action consists of a burst of large jobs (e.g., input sequence 1 in Figure 5-2a), the queues will build up, and the return will be poor compared to the value function baseline (average return from the state). On the other hand, a light stream of jobs (e.g., input sequence 2 in Figure 5-2a) will lead to short queues and a better-than-average return. Importantly, this difference in return has little to do with the action; it is a consequence of the random job arrival process.

We train two A2C agents [219], one with the standard value function baseline and the other with an input-dependent baseline tailored for each specific instantiation of the job arrival process (details of this baseline in §5.4). Since the the input-dependent baseline takes each input sequence into account explicitly, it reduces the variance of the policy gradient estimation much more effectively (Figure 5-2b, left). As a result, even in this simple example, only the policy learned with the input-dependent baseline comes close to the optimal (Figure 5-2b, right). Figure 5-2c visualizes the policies learned using the two baselines. The optimal policy (pick-shortest-queue) corresponds to a clear divide between the chosen servers at the diagonal.

In fact, the variance of the standard baseline can be arbitrarily worse than an input-dependent baseline: we refer the reader to Appendix B for an analytical example on a 1D grid world.

5.4 Reducing Variance for Input-Driven MDPs

We now formally define input-driven MDPs and derive variance-reducing baselines for policy gradient methods in environments with input processes.

Definition 1. *An input-driven MDP is defined by* $(\mathcal{S}, \mathcal{A}, \mathcal{Z}, \mathcal{P}_s, \mathcal{P}_z, \rho_0^s, \rho_0^z, r, \gamma)$*, where* $\mathcal{Z} \subseteq \mathbb{R}^k$ *is a set of* k*-dimensional input values,* $\mathcal{P}_s(s_{t+1}|s_t, a_t, z_t)$ *is the transition kernel of the states,* $\mathcal{P}_z(z_{t+1}|z_{0:t})$ *is the transition kernel of the input process,* $\rho_0^z(z_0)$ *is the distribution of the ini-*

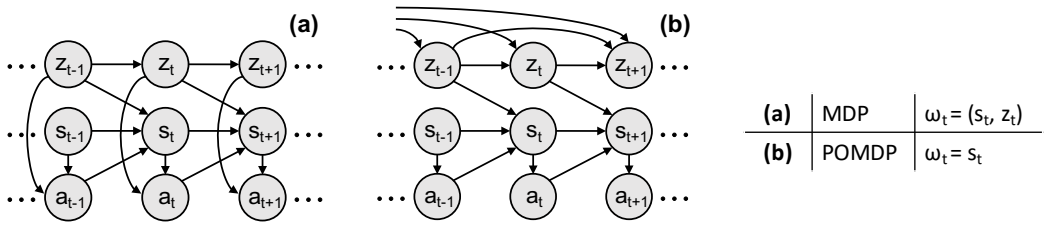


Figure 5-3: Graphical model of input-driven MDPs.

tial input, $r(s_t, a_t, z_t)$ is the reward function, and $\mathcal{S}$, $\mathcal{A}$, ρ_0^s, γ follow the standard definition in §5.2.

An input-driven MDP adds an *input process*, $\boldsymbol{z} = (z_0, z_1, \cdots)$, to a standard MDP. In this setting, the next state s_{t+1} depends on (s_t, a_t, z_t). We seek to learn policies that maximize cumulative expected rewards. We focus on two cases, corresponding to the graphical models shown in Figure 5-3:

Case 1: z_t is a Markov process, and $\omega_t = (s_t, z_t)$ is observed at time t. The action a_t can hence depend on both s_t and z_t.

Case 2: z_t is a general process (not necessarily Markov), and $\omega_t = s_t$ is observed at time t. The action a_t hence depends only on s_t.

We now prove that case 1 corresponds to a fully-observable MDP. This is evident from the graphical model in Figure 5-3a by considering $\omega_t = (s_t, z_t)$ to be the 'state' of the MDP at time t.

Proposition 1. *An input-driven decision process satisfying the conditions of case 1 in Figure 5-3 is a fully observable MDP, with state $\tilde{s}_t := (s_t, z_t)$, and action $\tilde{a}_t := a_t$.*

Proof.

$$\begin{aligned}
\Pr(\tilde{s}_{t+1} | \tilde{s}_{0:t}, \tilde{a}_{0:t}) &= \Pr(s_{t+1}, z_{t+1} | s_{0:t}, z_{0:t}, a_{0:t}) \\
&= \Pr(s_{t+1}, z_{t+1} | s_t, z_t, a_t) \quad \text{(by definition of case 1 in Figure 5-3a)} \\
&= \Pr(\tilde{s}_{t+1} | \tilde{s}_t, \tilde{a}_t).
\end{aligned}$$

□

Case 2, on the other hand, corresponds to a partially-observed MDP (POMDP) if we define the state to contain both s_t and $z_{0:t}$, but leave $z_{0:t}$ *unobserved* at time t.

Proposition 2. *An input-driven decision process satisfying the conditions of case 2 in Figure 5-3, with state $\tilde{s}_t := (s_t, z_{0:t})$ and action $\tilde{a}_t := a_t$ is a fully observable MDP. If only $\omega_t = s_t$ is observed at time t, it is a partially observable MDP (POMDP).*

Proof.

$$\begin{aligned}
\Pr(\tilde{s}_{t+1}|\tilde{s}_{0:t},\tilde{a}_{0:t}) &= \Pr(s_{t+1},z_{0:t+1}|s_{0:t},z_{0:t},a_{0:t}) \\
&= \Pr(s_{t+1}|s_{0:t},z_{0:t+1},a_{0:t})\Pr(z_{0:t+1}|s_{0:t},z_{0:t},a_{0:t}) \\
&= \Pr(s_{t+1}|s_t,z_{0:t+1},a_t)\Pr(z_{0:t+1}|s_t,z_{0:t},a_t) \text{ (by definition of case 2 in Figure 5-3b)} \\
&= \Pr(s_{t+1},z_{0:t+1}|s_t,z_{0:t},a_t) \\
&= \Pr(\tilde{s}_{t+1}|\tilde{s}_t,\tilde{a}_t).
\end{aligned}$$

Therefore, $(\tilde{s}_t,\tilde{a}_t)$ is a fully observable MDP. If only $\omega_t = s_t$ is observed, the decision process is a POMDP, since the $z_{0:t}$ component of the state is not observed. □

5.4.1 Variance Reduction

In input-driven MDPs, the standard input-agnostic baseline is ineffective at reducing variance, as shown by our motivating example (§5.3). We propose to use an *input-dependent* baseline of the form $b(\omega_t, z_{t:\infty})$ — a function of both the observation at time t and the input sequence from t onwards. An input-dependent baseline uses information that is not available to the policy. Specifically, the input sequence $z_{t:\infty}$ cannot be used when taking an action at time t, because $z_{t+1:\infty}$ has not yet occurred at time t. However, in many applications, the input sequence is known at training time. In some cases, we know the entire input sequence upfront, e.g., when training in a simulator. In other situations, we can record the input sequence on the fly during training. Then, after a training episode, we can use the recorded values, including those that occurred after time t, to compute the baseline for each step t.

We now analyze input-dependent baselines. Our main result is that input-dependent baselines are bias-free. We also derive the optimal input-dependent baseline for variance reduction. All the results hold for both cases in Figure 5-3. We first state two useful lemmas required for our analysis. The first lemma shows that under the input-driven MDP definition, the input sequence $z_{t:\infty}$ is conditionally independent of the action a_t given the observation ω_t, while the second lemma states the policy gradient theorem for input-driven MDPs.

Lemma 1. $\Pr(z_{t:\infty},a_t|\omega_t) = \Pr(z_{t:\infty}|\omega_t)\pi_\theta(a_t|\omega_t)$, *i.e.,* $z_{t:\infty} - \omega_t - a_t$ *forms a Markov chain.*

Proof. From the definition of an input-driven MDP (Definition 1), we have

$$\begin{aligned}\Pr(z_{0:\infty},\omega_t,a_t) &= \Pr(z_{0:t},\omega_t,a_t)\Pr(z_{t+1:\infty}|z_{0:t},\omega_t,a_t)\\ &= \Pr(z_{0:t},\omega_t)\Pr(a_t|z_{0:t},\omega_t)\Pr(z_{t+1:\infty}|z_{0:t})\\ &= \Pr(z_{0:t},\omega_t)\pi_\theta(a_t|\omega_t)\Pr(z_{t+1:\infty}|z_{0:t})\\ &= \Pr(z_{0:\infty},\omega_t)\pi_\theta(a_t|\omega_t). \end{aligned} \tag{5.2}$$

Notice that $\Pr(a_t|z_{0:t},\omega_t)=\pi_\theta(a_t|\omega_t)$ in both the MDP and POMDP cases in Figure 5-3. By marginalizing over $z_{0:t-1}$ on both sides, we obtain the result:

$$\Pr(z_{t:\infty},\omega_t,a_t)=\Pr(z_{t:\infty},\omega_t)\pi_\theta(a_t|\omega_t). \tag{5.3}$$

□

Lemma 2. *For an input-driven MDP, the policy gradient theorem can be rewritten as*

$$\nabla_\theta\eta(\pi_\theta)=\mathbb{E}_{\substack{(\omega,\boldsymbol{z})\sim\rho_\pi\\ a\sim\pi_\theta}}\Big[\nabla_\theta\log\pi_\theta(a|\omega)Q(\omega,a,\boldsymbol{z})\Big], \tag{5.4}$$

where $\rho_\pi(\omega,\boldsymbol{z})=\sum_{t=0}^{\infty}[\gamma^t\Pr(\omega_t=\omega,z_{t:\infty}=\boldsymbol{z})]$ *denotes the discounted visitation frequency of the observation* ω *and input sequence* $\boldsymbol{z}$*, and* $Q(\omega,a,\boldsymbol{z})=\mathbb{E}\big[\sum_{l=0}^{\infty}\gamma^l r_{t+l}\,\big|\,\omega_t=\omega,a_t=a,z_{t:\infty}=\boldsymbol{z}\big]$.

Proof. Expanding the Policy Gradient Theorem [285], we have

$$\begin{aligned}\nabla_\theta\eta(\pi_\theta) &= \mathbb{E}\left[\sum_{t=0}^{\infty}\nabla_\theta\log\pi_\theta(a_t|\omega_t)\sum_{t'\geq t}\gamma^{t'}r_{t'}\right]\\ &= \sum_{t=0}^{\infty}\mathbb{E}\left[\nabla_\theta\log\pi_\theta(a_t|\omega_t)\sum_{t'\geq t}\gamma^{t'}r_{t'}\right]\\ &= \sum_{t=0}^{\infty}\left[\sum_{\omega,a,\boldsymbol{z}}\Pr(\omega_t=\omega,a_t=a,z_{t:\infty}=\boldsymbol{z})\nabla_\theta\log\pi_\theta(a|\omega)\mathbb{E}\left[\sum_{t'\geq t}\gamma^{t'}r_{t'}|\omega_t=\omega,a_t=a,z_{t:\infty}=\boldsymbol{z}\right]\right]\\ &= \sum_{t=0}^{\infty}\left[\sum_{\omega,a,\boldsymbol{z}}\Pr(\omega_t=\omega,z_{t:\infty}=\boldsymbol{z})\pi_\theta(a|\omega)\nabla_\theta\log\pi_\theta(a|\omega)\mathbb{E}\left[\sum_{t'\geq t}\gamma^{t'}r_{t'}|\omega_t=\omega,a_t=a,z_{t:\infty}=\boldsymbol{z}\right]\right], \end{aligned} \tag{5.5}$$

where the last step uses Lemma 1. Using the definition of $Q(\omega,a,\boldsymbol{z})$, we obtain:

$$\begin{aligned}\nabla_\theta \eta(\pi_\theta) &= \sum_{t=0}^{\infty}\left[\sum_{\omega,a,\boldsymbol{z}} \Pr(\omega_t=\omega, z_{t:\infty}=\boldsymbol{z})\pi_\theta(a|\omega)\nabla_\theta \log\pi_\theta(a|\omega)\gamma^t Q(\omega,a,\boldsymbol{z})\right] \\ &= \sum_{\omega,a,\boldsymbol{z}}\left[\pi_\theta(a|\omega)\nabla_\theta\log\pi_\theta(a|\omega)Q(\omega,a,\boldsymbol{z})\left[\sum_{t=0}^{\infty}\gamma^t \Pr(\omega_t=\omega, z_{t:\infty}=\boldsymbol{z})\right]\right] \\ &= \sum_{\omega,a,\boldsymbol{z}} \pi_\theta(a|\omega)\nabla_\theta\log\pi_\theta(a|\omega)Q(\omega,a,\boldsymbol{z})\rho_\pi(\omega,\boldsymbol{z}) \\ &= \mathbb{E}_{\substack{(\omega,\boldsymbol{z})\sim\rho_\pi \\ a\sim\pi_\theta}}\Big[\nabla_\theta\log\pi_\theta(a|\omega)Q(\omega,a,\boldsymbol{z})\Big]. \end{aligned} \tag{5.6}$$

□

Equation (5.4) generalizes the standard Policy Gradient Theorem in Equation (5.1). $\rho_\pi(\omega,\boldsymbol{z})$ can be thought of as a joint distribution over observations and input sequences. $Q(\omega,a,\boldsymbol{z})$ is a "state-action-input" value function, i.e., the expected return when taking action a after observing ω, with input sequence $\boldsymbol{z}$ from that step onwards. The key ingredient in the proof of Lemma 2 is the conditional independence of the input process $z_{t:\infty}$ and the action a_t given the observation ω_t (Lemma 1).

Theorem 1. *An input-dependent baseline does not bias the policy gradient.*

Proof. Using Lemma 2, we need to show: $\mathbb{E}_{(\omega,\boldsymbol{z})\sim\rho_\pi, a\sim\pi_\theta}\left[\nabla_\theta\log\pi_\theta(a|\omega)b(\omega,\boldsymbol{z})\right] = 0$. We have:

$$\begin{aligned}\mathbb{E}_{\substack{(\omega,\boldsymbol{z})\sim\rho_\pi \\ a\sim\pi_\theta}}[\nabla_\theta\log\pi_\theta(a|\omega)b(\omega,\boldsymbol{z})] &= \sum_{\omega}\sum_{\boldsymbol{z}}\sum_{a}\rho_\pi(\omega,\boldsymbol{z})\pi_\theta(a|\omega)\nabla_\theta\log\pi_\theta(a|\omega)b(\omega,\boldsymbol{z}) \\ &= \sum_{\omega}\sum_{\boldsymbol{z}}\rho_\pi(\omega,\boldsymbol{z})b(\omega,\boldsymbol{z})\sum_{a}\pi_\theta(a|\omega)\nabla_\theta\log\pi_\theta(a|\omega). \end{aligned} \tag{5.7}$$

Since $\sum_a \pi_\theta(a|\omega)\nabla_\theta\log\pi_\theta(a|\omega) = \sum_a \nabla_\theta\pi_\theta(a|\omega) = \nabla_\theta\sum_a\pi_\theta(a|\omega) = 0$, the theorem follows. □

Input-dependent baselines are also bias-free for policy optimization methods such as TRPO [268], as we show in Appendix C. Next, we derive the optimal input-dependent baseline for variance reduction. As the gradient estimates are vectors, we use the trace of the covariance matrix as the minimization objective [125].

Theorem 2. *The input-dependent baseline that minimizes variance in policy gradient is given by*

$$b^*(\omega,\boldsymbol{z}) = \frac{\mathbb{E}_{a\sim\pi_\theta}\left[\nabla_\theta \log\pi_\theta(a|\omega)^T \nabla_\theta \log\pi_\theta(a|\omega) Q(\omega,a,\boldsymbol{z})\right]}{\mathbb{E}_{a\sim\pi_\theta}[\nabla_\theta \log\pi_\theta(a|\omega)^T \nabla_\theta \log\pi_\theta(a|\omega)]}. \tag{5.8}$$

Proof. Let $G(\omega,a)$ denote $\nabla_\theta \log\pi_\theta(a|\omega)^T \nabla_\theta \log\pi_\theta(a|\omega)$. For any input-dependent baseline $b(\omega,\boldsymbol{z})$, the variance of the policy gradient estimate is given by

$$\begin{aligned}
&\mathbb{E}_{\substack{(\omega,\boldsymbol{z})\sim\rho_\pi \\ a\sim\pi_\theta}}\left[\|\nabla_\theta \log\pi_\theta(a|\omega)[Q(\omega,a,\boldsymbol{z}) - b(\omega,\boldsymbol{z})] - \mathbb{E}_{\rho_\pi,\pi_\theta}[\nabla_\theta \log\pi_\theta(a|\omega)[Q(\omega,a,\boldsymbol{z}) - b(\omega,\boldsymbol{z})]]\|_2^2\right] \\
&= \mathbb{E}_{\rho_\pi,\pi_\theta}\left[G(\omega,a)\left[Q(\omega,a,\boldsymbol{z}) - b(\omega,\boldsymbol{z})\right]^2\right] - \left\|\mathbb{E}_{\rho_\pi,\pi_\theta}\left[\nabla_\theta \log\pi_\theta(a|\omega)\left[Q(\omega,a,\boldsymbol{z}) - b(\omega,\boldsymbol{z})\right]\right]\right\|_2^2 \\
&= \mathbb{E}_{\rho_\pi,\pi_\theta}\left[G(\omega,a)\left[Q(\omega,a,\boldsymbol{z}) - b(\omega,\boldsymbol{z})\right]^2\right] - \left\|\mathbb{E}_{\rho_\pi,\pi_\theta}\left[\nabla_\theta \log\pi_\theta(a|\omega) Q(\omega,a,\boldsymbol{z})\right]\right\|_2^2 \quad \text{(due to Theorem 1)} \\
&= \mathbb{E}_{\rho_\pi,\pi_\theta}\left[G(\omega,a) Q(\omega,a,\boldsymbol{z})^2\right] - \|\mathbb{E}_{\rho_\pi,\pi_\theta}[\nabla_\theta \log\pi_\theta(a|\omega) Q(\omega,a,\boldsymbol{z})]\|_2^2 \\
&\quad + \mathbb{E}_{\rho_\pi}\left[\mathbb{E}_{a\sim\pi_\theta}\left[G(\omega,a)\,\middle|\,\boldsymbol{z},\omega\right] b(\omega,\boldsymbol{z})^2 - 2\mathbb{E}_{a\sim\pi_\theta}\left[G(\omega,a) Q(\omega,a,\boldsymbol{z})\,\middle|\,\omega,\boldsymbol{z}\right] b(\omega,\boldsymbol{z})\right].
\end{aligned}$$

Notice that the baseline is only involved in the last term in a quadratic form, where the second order term is positive. To minimize the variance, we set baseline to the minimizer of the quadratic equation, i.e., $2\mathbb{E}_{a\sim\pi_\theta}\left[G(\omega,a)\,\middle|\,\omega,\boldsymbol{z}\right] b(\omega,\boldsymbol{z}) - 2\mathbb{E}_{a\sim\pi_\theta}\left[G(\omega,a) Q(\omega,a,\boldsymbol{z})\,\middle|\,\omega,\boldsymbol{z}\right] = 0$ and hence the result follows. □

Operationally, for observation ω_t at each step t, the input-dependent baseline takes the form $b(\omega_t, z_{t:\infty})$. In practice, we use a simpler alternative to Equation (5.8): $b(\omega_t, z_{t:\infty}) = \mathbb{E}_{a_t\sim\pi_\theta}[Q(\omega_t,a_t,z_{t:\infty})]$. This can be thought of as a value function $V(\omega_t, z_{t:\infty})$ that provides the expected return given observation ω_t and input sequence $z_{t:\infty}$ from that step onwards. We discuss how to estimate input-dependent baselines efficiently in §5.5.

Remark. Input-dependent baselines are generally applicable to reducing variance for policy gradient methods in input-driven environments. We apply input-dependent baselines to A2C (§5.6.1), TRPO (§5.6.2) and PPO [207, Appendix L]. Our technique is complementary and orthogonal to adversarial RL (e.g., RARL [247]) and meta-policy adaptation (e.g., MB-MPO [72]) for environments with external disturbances. Adversarial RL improves policy robustness by co-training an "adversary" to generate a worst-case disturbance process. Meta-policy optimization aims for fast policy adaptation to handle model discrepancy between training and testing. By contrast, input-dependent baselines improve policy optimization *itself* in the presence of stochastic input processes. Our work primarily focuses on learning a single policy in input-driven environments, without policy adaptation. However,

input-dependent baselines can be used as a general method to improve the policy optimization step in adversarial RL and meta-policy adaptation methods. For example, in §5.6.3, we empirically show that if an adversary generates high-variance noise, RARL with a standard state-based baseline cannot train good controllers, but the input-dependent baseline helps improve the policy's performance. Similarly, input-dependent baselines can improve meta-policy optimization in environments with stochastic disturbances, as we show in §5.6.4.

5.5 Learning Input-Dependent Baselines Efficiently

Input-dependent baselines are functions of the sequence of input values. A natural approach to train such baselines is to use models that operate on sequences (e.g., LSTMs [114]). However, learning a sequential mapping in a high-dimensional space can be expensive [30]. We considered an LSTM approach, but ruled it out when initial experiments showed that it fails to provide significant policy improvement over the standard baseline in our environments.

Fortunately, we can learn the baseline much more efficiently in applications where we can repeat the same input sequence multiple times during training. Input-repeatability is feasible in many applications: it is straightforward when using simulators for training, and also feasible when training a real system with previously-collected input traces outside simulation. For example, training a robot in the presence of exogenous forces might apply a set of time-series traces of these forces repeatedly to the physical robot. We now present two approaches that exploit input-repeatability to learn input-dependent baselines efficiently.

Multi-value-network approach. A straightforward way to learn $b(\omega_t, z_{t:\infty})$ for different input instantiations $\boldsymbol{z}$ is to train one value network to each particular instantiation of the input process. Specifically, in the training process, we first generate N input sequences $\{\boldsymbol{z}_1, \boldsymbol{z}_2, \cdots, \boldsymbol{z}_N\}$ and restrict training *only* to those N sequences. To learn a separate baseline function for each input sequence, we use N value networks with independent parameters $\theta_{V_1}, \theta_{V_2}, \cdots, \theta_{V_N}$, and single policy network with parameter θ. During training, we randomly sample an input sequence $\boldsymbol{z}_i$, execute a rollout based on $\boldsymbol{z}_i$ with the current policy π_θ, and use the (state, action, reward) data to train the value network parameter θ_{V_i} and the policy network parameter θ. Algorithm 1 depicts the details of this approach.

Meta-learning approach. The multi-value-network approach does not scale if the task requires training over a large number of input instantiations to generalize. The number of inputs needed is environment-specific, and can depend on a variety of factors, such as the

Algorithm 1 Training multi-value baselines for policy-based methods.

Require: pregenerated input seuqnces $\{\boldsymbol{z}_1,\boldsymbol{z}_2,\cdots,\boldsymbol{z}_N\}$, step sizes α,β

1: Initialize value network parameters $\theta_{V_1},\theta_{V_1},\cdots,\theta_{V_N}$ and policy parameters θ
2: **while** not done **do**
3: Sample a input sequence $\boldsymbol{z}_i$
4: Sample k rollouts $\mathcal{T}_1,\mathcal{T}_2,...,\mathcal{T}_k$ using policy π_θ and input sequence $\boldsymbol{z}_i$
5: Update policy with Equation (5.4) using baseline estimated with θ_{V_i}
6: Update i-th value network parameters: $\theta_{V_i} \leftarrow \theta_{V_i} - \beta \nabla_{\theta_{V_i}} \mathcal{L}_{1:k} \left[V_{\theta_{V_i}} \right]$
7: **end while**

time horizon of the problem, the distribution of the input process, the relative magnitude of the variance due to the input process compared to other sources of randomness (e.g., actions). Ideally, we would like an approach that enables learning across many different input sequences. We present a method based on *meta-learning* to train with an unbounded number of input sequences. The idea is to use *all* (potentially infinitely many) input sequences to learn a "meta value network" model. Then, for each specific input sequence, we first customize the meta value network using a few example rollouts with that input sequence. We then compute the actual baseline values for training the policy network parameters, using the customized value network for the specific input sequence. Our implementation uses Model-Agnostic Meta-Learning (MAML) [101].

Algorithm 2 Training a meta input-dependent baseline for policy-based methods.

Require: α, β: meta value network step size hyperparameters

1: Initialize policy network parameters θ and meta-value-network parameters θ_V
2: **while** not done **do**
3: Generate a new input sequence $\boldsymbol{z}$
4: Sample k rollouts $\mathcal{T}_1,\mathcal{T}_2,...,\mathcal{T}_k$ using policy π_θ and input sequence $\boldsymbol{z}$
5: Adapt θ_V with the first $k/2$ rollouts: $\theta_V^1 = \theta_V - \alpha \nabla_{\theta_V} \mathcal{L}_{\mathcal{T}_{1:k/2}}[V_{\theta_V}]$
6: Estimate baseline value $V_{\theta_V^1}(\omega_t)$ for $s_t \sim \mathcal{T}_{k/2:k}$ using adapted θ_V^1
7: Adapt θ_V with the second $k/2$ rollouts: $\theta_V^2 = \theta_V - \alpha \nabla_{\theta_V} \mathcal{L}_{\mathcal{T}_{k/2:k}}[V_{\theta_V}]$
8: Estimate baseline value $V_{\theta_V^2}(\omega_t)$ for $s_t \sim \mathcal{T}_{1:k/2}$ using adapted θ_V^2
9: Update policy with Equation (5.4) using the values from line (6) and (8) as baseline
10: Update meta value network: $\theta_V \leftarrow \theta_V - \beta \nabla_{\theta_V} \mathcal{L}_{k/2:k} \left[V_{\theta_V^1} \right] - \beta \nabla_{\theta_V} \mathcal{L}_{1:k/2} \left[V_{\theta_V^2} \right]$
11: **end while**

The pseudocode in Algorithm 2 depicts the training algorithm. We follow the notation of MAML, denoting the loss in the value function $V_{\theta_V}(\cdot)$ on a rollout $\mathcal{T}$ as $\mathcal{L}_{\mathcal{T}}[V_{\theta_V}] = \sum_{\omega_t, r_t \sim \mathcal{T}} \| V_{\theta_V}(\omega_t) - \sum_{t'=t}^{T} \gamma^{t'-t} r_t \|^2$. We perform rollouts k times with the same input sequence $\boldsymbol{z}$ (lines 3 and 4); we use the first $k/2$ rollouts to customize the meta value network

for this instantiation of z (line 5), and then apply the customized value network on the states of the other $k/2$ rollouts to compute the baseline for those rollouts (line 6); similarly, we swap the two groups of rollouts and repeat the same process (lines 7 and 8). We use different rollouts to adapt the meta value network and compute the baseline to avoid introducing extra bias to the baseline. Finally, we use the baseline values computed for each rollout to update the policy network parameters (line 9), and we apply the MAML [101] gradient step to update the meta value network model (line 10).

5.6 Experiments

Our experiments demonstrate that input-dependent baselines provide consistent performance gains across multiple continuous-action MuJoCo simulated robotic locomotions and discrete-action environments in queuing systems and network control. We conduct experiments for both policy gradient methods and policy optimization methods (see Appendix D for details). The videos for our experiments are available at https://sites.google.com/view/input-dependent-baseline/.

5.6.1 Discrete-Action Environments

Our discrete-action environments arise from widely-studied problems in computer systems research: load balancing and bitrate adaptation.[1] As these problems often lack closed-form optimal solutions [123, 329], hand-tuned heuristics abound. Recent work suggests that model-free reinforcement learning can achieve better performance than such human-engineered heuristics [201, 94, 204, 218]. We consider a load balancing environment (similar to the example in §5.3) and a bitrate adaptation environment in video streaming [329].

Load balancing across servers (Figure 5-1a). In this environment, an RL agent balances jobs over k servers to minimize the average job completion time. Similar to §5.3, the job sizes follow a Pareto distribution (scale $x_m = 100$, shape $\alpha = 1.5$), and jobs arrive in a Poisson process ($\lambda = 55$). We run over 10 simulated servers with different processing rates, ranging linearly from 0.15 to 1.05. In this setting, the load of the system is at 90% (i.e., on average, 90% of the queues are non-empty). In each episode, we generate 500 jobs as the ex-

[1] We considered Atari games often used as benchmark discrete-action RL environments [220]. However, Atari games lack an exogenous input process: a random seed perturbs the games' initial state, but it does not affect the environmental changes (e.g., in "Seaquest", the ships always come in a fixed pattern).

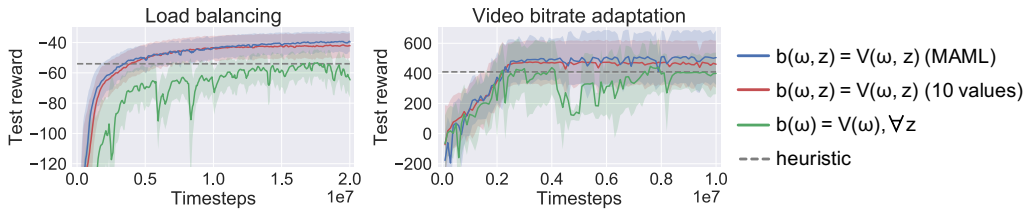


Figure 5-4: In environments with discrete action spaces, A2C [219] with input-dependent baselines outperforms the best heuristic and achieves 25–33% better testing reward than vanilla A2C [219]. Learning curves are on 100 test episodes with unseen input sequences; shaded area spans one standard deviation.

ogenous input process. The problem of minimizing average job completion time on servers with heterogeneous processing rates does not have a closed-form solution [133]; the most widely-used heuristic is to join the shortest queue [74]. However, understanding the workload pattern can give a better policy; for example, we can reserve some servers for small jobs. In this environment, the observed state is a vector of $(j,q_1,q_2,...,q_k)$, where j is the size of the incoming job, q_i is the amount of work currently in each queue. The action $a \in \{1,2,...,k\}$ schedules the incoming job to a specific queue. The reward is the number of active jobs times the negated time elapsed since the last action.

Bitrate adaptation for video streaming (Figure 5-1b). Streaming video over variable-bandwidth connections requires the client to adapt the video bitrates to optimize the user experience. This is challenging since the available network bandwidth (the exogenous input process) is hard to predict accurately. We simulate real-world video streaming using public cellular network data [259] and video with seven bitrate levels and 500 chunks [75]. The reward is a weighted combination of video resolution, time paused for rebuffering, and the number of bitrate changes [204]. The observed state contains bandwidth history, current video buffer size, and current bitrate. The action is the next video chunk's bitrate. State-of-the-art heuristics for this problem conservatively estimate the network bandwidth and use model predictive control to choose the optimal bitrate over the near-term horizon [329].

Results. We extend OpenAI's A2C implementation [83] for our baselines. The learning curves in Figure 5-4 illustrate that directly applying A2C with a standard value network as the baseline results in unstable test reward and underperforms the traditional heuristic in both environments. Our input-dependent baselines reduce the variance and improve test reward by 25–33%, outperforming the heuristic. The meta-baseline performs the best in all environments.

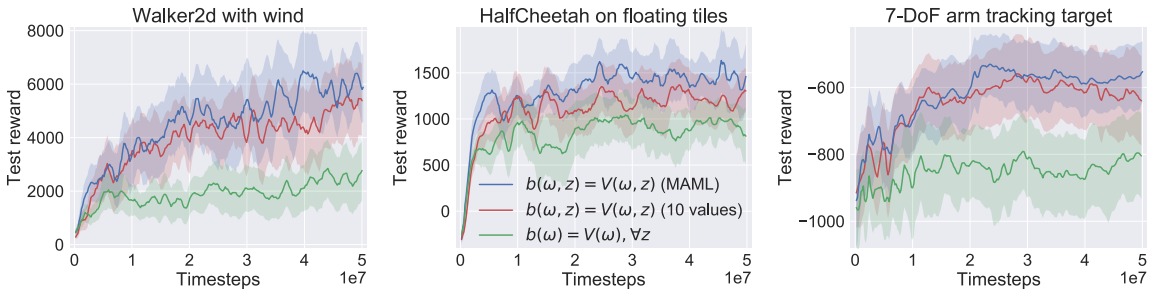


Figure 5-5: In continuous-action MuJoCo environments, TRPO [268] with input-dependent baselines achieve 25%–3× better testing reward than with a standard state-dependent baseline. Learning curves are on 100 testing episodes with unseen input sequences; shaded area spans one standard deviation.

5.6.2 Simulated Robotic Locomotion

We use the MuJoCo physics engine [294] in OpenAI Gym [53] to evaluate input-dependent baselines for robotic control tasks with external disturbance. We extend the standard Walker2d, HalfCheetah and 7-DoF robotic arm environments, adding a different external input to each (Figure 5-1).

Walker2d with random wind (Figure 5-1c). We train a 2D walker with varying wind, which randomly drags the walker backward or forward with different force at each step. The wind vector changes randomly, i.e., the wind forms a random input process. We add a force sensor to the state to enable the agent to quickly adapt. The goal is for the walker to walk forward while keeping balance.

HalfCheetah on floating tiles with random buoyancy (Figure 5-1d). A half-cheetah runs over a series of tiles floating on water [71]. Each tile has different damping and friction properties, which moves the half-cheetah up and down and changes its dynamics. This random buoyancy is the external input process; the cheetah needs to learn running forward over varying tiles.

7-DoF arm tracking moving target (Figure 5-1e). We train a simulated robot arm to track a randomly moving target (a red ball). The robotic arm has seven degrees of freedom and the target is doing a random walk, which forms the external input process. The reward is the negative squared distance between the robot hand (blue square) and the target.

The Walker2d and 7-DoF arm environments correspond to the fully observable MDP case in Figure 5-3, i.e. the agent observes the input z_t at time t. The HalfCheetah environment is a POMDP, as the agent does not observe the buoyancy of the tiles. In Appendix H of our pa-

per [207], we additionally show results for the POMDP version of the Walker2d environment.

Results. We build 10-value networks and a meta-baseline using MAML, both on top of the OpenAI's TRPO implementation [83]. Figure 5-5 shows the performance comparison among different baselines with 100 unseen testing input sequences at each training checkpoint. These learning curves show that TRPO with a state-dependent baseline performs worst in all environments. With the input-dependent baseline, by contrast, performance in unseen testing environments improves by up to $3\times$, as the agent learns a policy robust against disturbances. For example, it learns to lean into headwind and quickly place its leg forward to counter the headwind; it learns to apply different force on tiles with different buoyancy to avoid falling over; and it learns to co-adjust multiple joints to keep track of the moving object. The meta-baseline eventually outperforms 10-value networks as it effectively learns from a large number of input processes and hence generalizes better. In Appendix L of our paper [207], we also show a similar comparison with PPO [270].

5.6.3 Input-dependent baselines with RARL

Our work is orthogonal and complementary to adversarial and robust reinforcement learning (e.g., RARL [247]). These methods seek to improve policy robustness by co-training an adversary to generate a worst-case noise process, whereas our work improves policy optimization itself in the presence of inputs like noise. Note that if an adversary generates high-variance noise, similar to the inputs we consider in our experiments, techniques such RARL alone are not adequate to train good controllers.

To empirically demonstrate this effect, we repeat the Walker2d with wind experiment described in §5.6.2. In this environment, we add an additional noise (with the same scale as the original random walk) on the wind and co-train an adversary to control the strength and direction of this noise. We follow the training procedure described in RARL [247, §3.3].

Figure 5-6 depicts the results. With either the standard state-dependent baseline or our input-dependent baseline, RARL generally improves the robustness of the policy, as RARL achieves better testing rewards especially in the low reward region (i.e., compared the yellow curve to green curve, or red curve to blue curve in CDF of Figure 5-6). Moreover, input-dependent baseline significantly improves the policy optimization, which boosts the performance of both TRPO and RARL (i.e., comparing the blue curve to the green curve, and the red curve to the yellow curve). Therefore, in this environment, the input-dependent

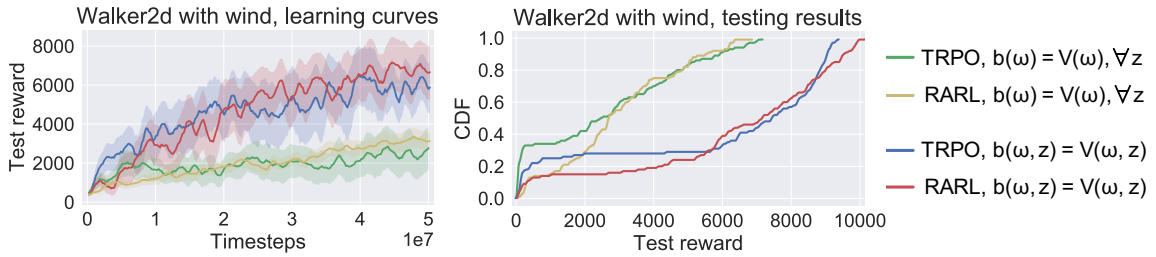


Figure 5-6: The input-dependent baseline technique is complementary and orthogonal to RARL [247]. The implementation of input-dependent baseline is MAML (§5.5). Left: learning curves of testing rewards; shaded area spans one standard deviation; the input-dependent baseline improves the policy optimization for both TRPO and RARL, while RARL improves TRPO in the Walker2d environment with wind disturbance. Right: CDF of testing performance; RARL improves the policy especially in the low reward region; applying the input-dependent baseline boosts the performance for both TRPO and RARL significantly (blue, red).

baseline helps improve the policy optimization methods and is complementary to adversarial RL methods such as RARL.

5.6.4 Input-dependent baselines with meta-policy adaptation

There has been a line of work focusing on fast policy adaptation [71, 72, 134]. For example, Clavera et al. [72] propose a model-based meta-policy optimization approach (MB-MPO). It quickly learns the system dynamics using supervised learning and uses the learned model to perform virtual rollouts for meta-policy adaptation. Conceptually, our work differs because the goal is fundamentally different: our goal is to learn a single policy that performs well in the presence of a stochastic input process, while MB-MPO aims to quickly adapt a policy to new environments.

It is worth noting that the policy adaptation approaches are well-suited to handling model discrepancy between training and testing. However, in our setting, there exists no model discrepancy. In particular, the distribution of the input process is the same during training and testing. For example, in our load balancing environment (Figure 5-1a, §5.6.1), the exogenous workload process is sampled from the same distribution during training and testing.

Therefore our work is conceptually complementary to policy adaptation approaches. Since some of these methods require a policy optimization step (e.g., [72, §4.2]), our input-dependent baseline can help these methods by reducing variance during training. We perform an experiment to investigate this. Specifically, we apply the meta-policy adaptation technique [72] in our Walker2d environment with wind disturbance (Figure 5-1c, §5.6.2).

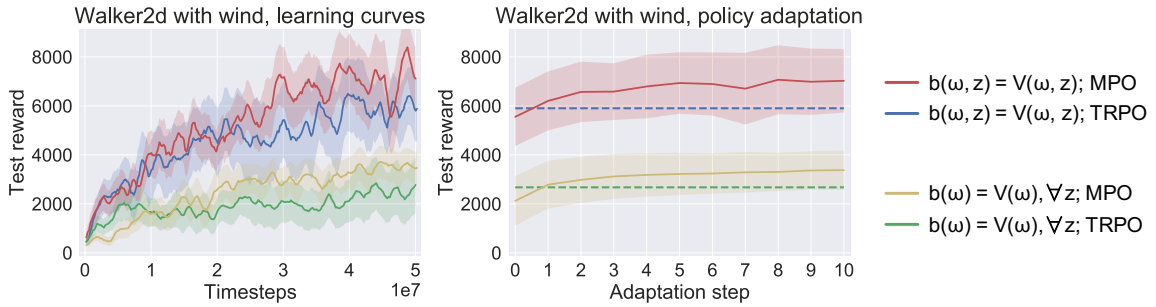


Figure 5-7: The input-dependent baseline technique is complementary to MPO [72]. The implementation of input-dependent baseline is MAML (§5.5). Left: learning curves in the testing Walker2d environment with wind disturbance; MPO is tested with adapted policy in each testing instance of the wind input; shaded area spans one standard deviation; the input-dependent baseline improves the policy optimization for both TRPO and MPO, while MPO improves TRPO. Right: meta policy adaptation at training timestep $5e7$; adapting the policy in specific input instances help boosting the performance (comparing yellow with green, and red with blue); applying input-dependent baseline generally improves the policy performance.

For this environment, although the wind pattern is drawn from the same stochastic process (random walk), we aim to adapt the policy to each particular instantiation of the wind.

Operationally, to reduce complexity, we bypass the supervised learning step for the system dynamics and use the simulator to generate rollouts directly, since the interaction with the simulator is not costly for our purpose and the state transition in our environment is not deterministic. Following the meta-policy adaptation approach, the policy optimization algorithm is TRPO [268]. The meta-policy adaptation algorithm is MAML [101]. In particular, we performed ten gradient steps to specialize the meta-policy for each instantiation of the input process. For input-dependent baseline, we inherit our meta-baseline approach from §5.5. Similar to policy adaptation, we adapt our meta-baseline alongside with the policy adaptation in the ten gradient steps for each input instance.

The results of our experiment is shown in Figure 5-7. The learning curve (left figure) shows the policy performance for 100 unseen test input sequences at each training checkpoint. We measure the performance of MPO after ten steps of policy adaptation for each of the 100 input sequences. As expected, policy adaptation specializes to the particular instance of the input process and improves policy performance in the learning curve (e.g., MPO improves over TRPO, as shown by the green and yellow learning curve). However, policy adaptation does not solve the problem of variance caused by the input process, since the policy optimization step within policy adaptation suffers from large variance. Using an input-dependent baseline improves performance both for TRPO and MPO. Indeed, MPO

trained with the input-dependent baseline (and adapted for each input sequence) outperforms the single TRPO policy, as shown by the red learning curve.

This effect is more evident in the policy adaptation curve (right figure). The policy adaptation curve shows the testing performance of the adapted policy at each adaptation step (the meta-policy is taken from the $5e7$ training timestep). With an input-dependent baseline, the meta policy already performs quite well at the 0^{th} step of policy adaptation (without any adaptation). This is perhaps unsurprising, since a single policy (e.g., the TRPO policy trained with input-dependent baseline) can achieve good performance in this environment. However, specializing the meta-policy for each particular input instance further improves performance, which shows these approaches are complement to each other.

5.7 Related Work

Policy gradient methods compute unbiased gradient estimates, but can experience a large variance [285, 316]. Reducing variance for policy-based methods using a baseline has been shown to be effective [317, 285, 316, 125, 219]. Much of this work focuses on variance reduction in a general MDP setting, rather than variance reduction for MDPs with specific stochastic structures. Wu et al. [322]'s techniques for MDPs with multi-variate independent actions are closest to our work. Their state-action-dependent baseline improves training efficiency and model performance on high-dimensional control tasks by explicitly factoring out, for each action, the effect due to other actions. By contrast, our work exploits the structure of state transitions instead of stochastic policy.

Recent work has also investigated the bias-variance tradeoff in policy gradient methods. Schulman et al. [269] replace the Monte Carlo return with a λ-weighted return estimation (similar to TD(λ) with value function bootstrap [289]), improving performance in high-dimensional control tasks. Other recent approaches use more general control variates to construct variants of policy gradient algorithms. Tucker et al. [296] compare the recent work, both analytically on a linear-quadratic-Gaussian task and empirically on complex robotic control tasks. Analysis of control variates for policy gradient methods is a well-studied topic, and extending such analyses (e.g., [125]) to the input-driven MDP setting could be interesting future work.

In other contexts, prior work has proposed new RL training methodologies for environments with disturbances. Clavera et al. [72] adapts the policy to different pattern of distur-

bance by training the RL agent using meta-learning. RARL [247] improves policy robustness by co-training an adversary to generate a worst-case noise process. Our work is orthogonal and complementary to these work, as we seek to improve policy optimization itself in the presence of inputs like disturbances.

5.8 Conclusion

We introduced input-driven Markov Decision Processes in which stochastic input processes influence state dynamics and rewards. In this setting, we demonstrated that an input-dependent baseline can significantly reduce variance for policy gradient methods, improving training stability and the quality of learned policies. Our work provides an important ingredient for using RL successfully in a variety of domains, including queuing networks and computer systems, where an input workload is a fundamental aspect of the system, as well as domains where the input process is more implicit, like robotics control with disturbances or random obstacles.

We showed that meta-learning provides an efficient way to learn input-dependent baselines for applications where input sequences can be repeated during training. Investigating efficient architectures for input-dependent baselines for cases where the input process cannot be repeated in training is an interesting direction for future work.

Chapter 6

An Open Platform for Learning-Augmented Computer Systems

6.1 Introduction

Deep reinforcement learning (RL) has emerged as a general and powerful approach to sequential decision making problems in recent years. However, real-world applications of deep RL have thus far been limited. The successes, while impressive, have largely been confined to controlled environments, such as complex games [220, 278, 293, 234, 308] or simulated robotics tasks [253, 235, 148]. This thesis concerns applications of RL in computer systems, a relatively unexplored domain where RL could provide significant real-world benefits.

Computer systems are full of sequential decision-making tasks that can naturally be expressed as Markov decision processes (MDP). Examples include caching (operating systems), congestion control (networking), query optimization (databases), scheduling (distributed systems), and more (§6.2). Since real-world systems are difficult to model accurately, state-of-the-art systems often rely on human-engineered heuristic algorithms that can leave significant room for improvement [218]. Further, these algorithms can be complex (e.g., a commercial database query optimizer involves hundreds of rules [38]), and are often difficult to adapt across different systems and operating environments [206, 210] (e.g., different workloads, different distribution of data in a database, etc.). Furthermore, unlike control applications in physical systems, most computer systems run in software on readily-available commodity machines. Hence the cost of experimentation is much lower than physical environments such as robotics, making it relatively easy to generate abun-

dant data to explore and train RL models. This mitigates (but does not eliminate) one of the drawbacks of RL approaches in practice — their high sample complexity [22]. The easy access to training data and the large potential benefits have attracted a surge of recent interest in the systems community to develop and apply RL tools to various problems [201, 206, 155, 175, 210, 173, 209, 228, 49, 204, 68, 218, 217, 109, 5, 57, 314, 187, 111].

From a machine learning perspective, computer systems present many challenging problems for RL. The landscape of decision-making problems in systems is vast, ranging from centralized control problems (e.g., a scheduling agent responsible for an entire computer cluster) to distributed multi-agent problems where multiple entities with partial information collaborate to optimize system performance (e.g., network congestion control with multiple connections sharing bottleneck links). Further, the control tasks manifest at a variety of timescales, from fast, reactive control systems with sub-second response-time requirements (e.g., admission/eviction algorithms for caching objects in memory) to longer term planning problems that consider a wide range of signals to make decisions (e.g., VM allocation/-placement in cloud computing). Importantly, computer systems give rise to new challenges for learning algorithms that are not common in other domains (§6.3). Examples of these challenges include time-varying state or action spaces (e.g., dynamically varying number of jobs and machines in a computer cluster), structured data sources (e.g., graphs to represent data flow of jobs or a network's topology), and highly stochastic environments (e.g., random time-varying workloads). These challenges present new opportunities for designing RL algorithms. For example, motivated by applications in networking and queuing systems, recent work [207] developed new general-purpose control variates for reducing variance of policy gradient algorithms in "input-driven" environments, in which the system dynamics are affected by an exogenous, stochastic process.

Despite these opportunities, there is relatively little work in the machine learning community on algorithms and applications of RL in computer systems. We believe a primary reason is the lack of good benchmarks for evaluating solutions, and the absence of an easy-to-use platform for experimenting with RL algorithms in systems. Conducing research on learning-based systems currently requires significant expertise to implement solutions in real systems, collect suitable real-world traces, and evaluate solutions rigorously. The primary goal of this chapter is to lower the barrier of entry for machine learning researchers to innovate in computer systems.

We present Park, an open, extensible platform that presents a common RL interface to

connect to a suite of 12 computer system environments (§6.4). These representative environments span a wide variety of problems across networking, databases, and distributed systems, and range from centralized planning problems to distributed fast reactive control tasks. In the backend, the environments are powered by both real systems (in 7 environments) and high fidelity simulators (in 5 environments). For each environment, Park defines the MDP formulation, e.g., events that triggers an MDP step, the state and action spaces and the reward function. This allows researchers to focus on the core algorithmic and learning challenges, without having to deal with low-level system implementation issues. At the same time, Park makes it easy to compare different proposed learning agents on a common benchmark, similar to how OpenAI Gym [53] has standardized RL benchmarks for robotics control tasks. Finally, Park defines a RPC interface [283] between the RL agent and the backend system, making it easy to extend to more environments in the future.

We benchmark the 12 systems in Park with both RL methods and existing heuristic baselines (§6.5). The experiments benchmark the training efficiency and the eventual performance of RL approaches on each task. The empirical results are mixed: RL is able to outperform state-of-the-art baselines in several environments where researchers have developed problem-specific learning methods; for many other systems, RL has yet to consistently achieve robust performance. We open-source Park as well as the RL agents and baselines in https://github.com/park-project/park.

6.2 Sequential Decision Making Problems in Computer Systems

Sequential decision making problems manifest in a variety of ways across computer systems disciplines. These problems span a multi-dimensional space from centralized vs. multi-agent control to reactive, fast control loops vs. long-term planning. In this section, we overview a sample of problems from each discipline and how to formulate them as MDPs. Appendix E provides further examples and a more formal description of the MDPs that we have implemented in Park.

Networking. Computer network problems are fundamentally distributed, since they interconnect independent users. One example is congestion control, where hosts in the network must each determine the rate to send traffic, accounting for both the capacity of the under-

lying network infrastructure and the demands of other users of the network. Each network connection has an agent (typically at the sender side) setting the sending rate based on how previous packets were acknowledged. This component is crucial for maintaining a large throughput and low delay.

Another example at the application layer is bitrate adaptation in video streaming. When streaming videos from content provider, each video is divided into multiple chunks. At watch time, an agent decides the bitrate (affecting resolution) of each chunk of the video based on the network (e.g., bandwidth and latency measurements) and video characteristics (e.g., type of video, encoding scheme, etc.). The goal is to learn a policy that maximizes the resolution while minimizing chance of stalls (when slow network cannot download a chunk fast enough).

Databases. Databases seek to efficiently organize and retrieve data in response to user requests. To efficiently organize data, it is important to index, or arrange, the data to suit the retrieval patterns. An indexing agent could observe query patterns and accordingly decide how to best structure, store, and over time, re-organize the data.

Another example is query optimization. Modern query optimizers are complex heuristics which use a combination of rules, handcrafted cost models, data statistics, and dynamic programming, with the goal to re-order the query operators (e.g., joins, predicates) to ultimately lower the execution time. Unfortunately, existing query optimizers do not improve over time and do not learn from mistakes. Thus, they are an obvious candidate to be optimized through RL [210]. Here, the goal is to learn a query optimization policy based on the feedback from optimizing and running a query plan.

Distributed systems. Distributed systems handle computations that are too large to fit on one computer; for example, the Spark framework for big-data processing computes results across data stored on multiple computers [333]. To efficiently perform such computations, a job scheduler decides how the system should assign compute and memory resources to jobs to achieve fast completion times. Data processing jobs often have complex structure (e.g., Spark jobs are structured as dataflow graphs, Tensorflow models are computation graphs). The agent in this case observes a set of jobs and the status of the compute resources (e.g., how each job is currently assigned). The action decides how to place jobs onto compute resources. The goal is to complete the jobs as soon as possible.

Operating systems. Operating systems seek to efficiently multiplex hardware resources (compute, memory, storage) amongst various application processes. One example is pro-

viding a memory hierarchy: computer systems have a limited amount of fast memory and relatively large amounts of slow storage. Operating systems provide caching mechanisms which multiplex limited memory amongst applications which achieve performance benefits from residency in faster portions of the cache hierarchy. In this setting, an RL agent can observe the information of both the existing objects in the cache and the incoming object; it then decides whether to admit the incoming object and which stale objects to evict from the cache. The goal is to maximize the cache hit rate (so that more application reads occur from fast memory) based on the access pattern of the objects.

Another example is CPU power state management. Operating systems control whether the CPU should run at an increased clock speed and boost application performance, or save energy with at a lower clock speed. An RL agent can dynamically control the clock speed based on the observation of how each application is running (e.g., is an application CPU bound or network bound, is the application performing IO tasks). The goal is to maintain high application performance while reducing the power consumption.

6.3 RL for Systems Characteristics and Challenges

In this section, we explain the unique characteristics and challenges that often prevent off-the-shelf RL methods from achieving strong performance in different computer system problems. Admittedly, each system has its own complexity and contains special challenges. Here, we primarily focus on the common challenges that arise across many systems in different stages of the RL design pipeline.

6.3.1 State-action Space

The needle-in-the-haystack problem. In some computer systems, the majority of the state-action space presents little difference in reward feedback for exploration. This provides no meaningful gradient during RL training, especially in the beginning, when policies are randomly initialized. Network congestion control is a classic example: even in the simple case of a fixed-rate link, setting the sending rate above the available network bandwidth saturates the link and the network queue. Then, changes in the sending rate above this threshold result in an equivalently bad throughput and delay, leading to constant, low rewards. To exit this bad state, the agent must set a low sending rate for multiple *consecutive* steps to drain the

	GCN direct	GCN transfer	LSTM direct	LSTM transfer	Random
CIFAR-10 [176]	**1.73**±0.41	**1.81**±0.39	**1.78**±0.38	1.97±0.37	2.15±0.39
Penn Tree Bank [208]	**4.84**±0.64	**4.96**±0.63	5.09±0.63	5.28±0.6	5.42±0.57
NMT [30]	**1.98**±0.55	**2.07**±0.51	2.16±0.56	2.88±0.66	2.47±0.48

Table 6.1: Generalizability of GCN and LSTM state representation in the Tensorflow device placement environment. The numbers are average runtime in seconds. ± spans one standard deviation. Bold font indicate the runtime is within 5% of the best runtime. "Transfer" means testing on unseen models in the dataset.

queue before receiving any positive reward. Random exploration is not effective at learning this behavior because any random action can easily overshadow several good actions, making it difficult to distinguish good action sequences from bad ones. Circuit design is another example: when *any* of the circuit components falls outside the operating region (the exact boundary is unknown before invoking the circuit simulator), the circuit cannot function properly and the environment returns a constant bad reward. As a result, exploring these areas provides little gradient for policy training.

In these environments, using domain-knowledge to confine the search space helps to train a strong policy. For example, we observed significant performance improvements for network congestion control problems when restricting the policy (see also Figure 6-3d). Also, environment-specific reward shaping [231] or bootstrapping from existing policies [277, 140] can improve policy search efficiency.

Representation of state-action space. When designing RL methods for problems with complex structure, properly encoding the state-action space is the key challenge. In some systems, the action space grows exponentially large as the problem size increases. For example, in switch scheduling, the action is a bijection mapping (a matching) between input and output ports — a standard 32-port would have 32! possible matching. Encoding such a large action space is challenging and makes it hard to use off-the-shelf RL agents. In other cases, the size of the action space is constantly changing over time. For example, a typical problem is to map jobs to machines. In this case, the number of possible mappings and thus, actions increases with the number of new jobs in the system.

Unsurprisingly, domain specific representations that capture inherent structure in the state space can significantly improve training efficiency and generalization. For example, Spark jobs, Tensorflow components, and circuit design are to some degree dataflow graphs. For these environments, leveraging Graph Convolutional Neural Networks (GCNs) [170] rather than LSTMs can significantly improves generalization (see Table 6.1). However, find-

ing the right representation for each problem is a central challenge, and for some domains, e.g., query optimization, remains largely unsolved.

6.3.2 Decision Process

Stochasticity in MDP causing huge variance. Queuing systems environments (e.g., job scheduling, load balancing, cache admission) have dynamics partially dictated by an exogenous, stochastic *input process*. Specifically, their dynamics are governed not only by the decisions made within the system, but also the arrival process that brings work (e.g., jobs, packets) into the system. In these environments, the stochasticity in the input process causes huge variance in the reward.

For illustration, consider the load balancing example in Figure 4-9. If the arrival sequence after time t consists of a burst of large jobs (e.g., job sequence 1), the job queue will grow and the agent will receive low rewards. In contrast, a stream of lightweight jobs (e.g., job sequence 2) will lead to short queues and large rewards. The problem is that this difference in reward is independent of the action at time t; rather, it is caused purely by the randomness in the job arrival process. In these environments, the agents cannot tell whether two reward feedbacks differ due to disparate input processes, or due to the quality of the actions. As a result, standard methods for estimating the value of an action suffer from high variance.

Prior work proposed an input-dependent baseline (§5) that effectively reduces the variance from the input process [207]. Figure 5-4 shows the policy improvement when using input-dependent baselines in the load-balancing and adaptive video streaming environments. However, the proposed training implementations ("multi-value network" and "meta baseline") are tailored for policy gradient methods and require the environments to have a repeatable input process (e.g., in simulation, or real systems with controllable input sequence). Thus, coping with input-driven variance remains an open problem for value-based RL methods and for environments with uncontrollable input processes.

Infinite horizon problems. In practice, production computer systems (e.g., Spark schedulers, load balancers, cache controllers, etc.) are long running and host services indefinitely. This creates an infinite horizon MDP [37] that prevents the RL agents from performing episodic training. In particular, this creates difficulties for bootstrapping a value estimation since there is no terminal state to easily assign a known target value. Moreover, the discounted total reward formulation in the episodic case might not be suitable — an action

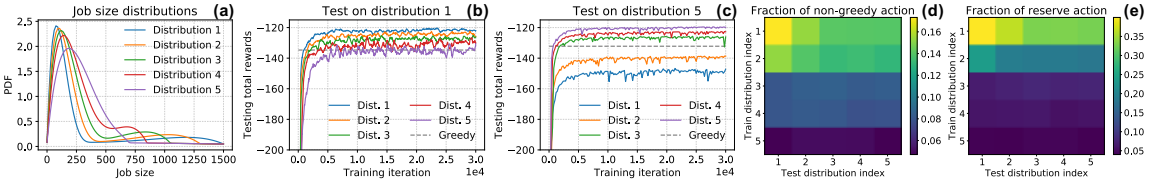


Figure 6-1: Demonstration of the gap between simulation and reality in the load balancing environment. (a) Distribution of job sizes in the training workload. (b, c) Testing agents on a particular distribution. An agent trained with distribution 5 is more robust than one trained with distribution 1. (d, e) A "reservation" policy that keeps a server empty for small jobs. Such a policy overfits distribution 1 and is not robust to workload changes.

in a long running system can have impact beyond a fixed discounting window. For example, scheduling a large job on a slow server blocks future small jobs (affecting job runtime in the rewards), no matter whether the small jobs arrive immediately after the large job or much farther in the future over the course of the lifetime of the large job. Average reward RL formulations can be a viable alternative in this setting (see §10.3 in [285] for an example).

6.3.3 Simulation-Reality Gap

Unlike training RL in simulation, robustly deploying a trained RL agent or directly training RL on an actual running computer systems has several difficulties. First, discrepancies between simulation and reality prevent direct generalization. For example, in database query optimization, existing simulators or query planners use offline cost models to predict query execution time (as a proxy for the reward). However, the accuracy of the cost model quickly degrades as the query gets more complex due to both variance in the underlying data distribution and system-specific artifacts [183].

Second, interactions with some real systems can be slow. In adaptive video streaming, for example, the agent controls the bitrate for each chunk of a video. Thus, the system returns a reward to the agent only after a video chunk is downloaded, which typically takes a few seconds. Naively using the same training method from simulation (as in Figure 6-3a) would take a single-threaded agent more than 10 years to complete training in reality.

Finally, live training or directly deploying an agent from simulation can degrade the system performance. Figure 6-1 describes a concrete example for load balancing. The reason is that based on the bimodal distribution in the beginning, it learns to reserve a certain server for small jobs. However, when the distribution changes, blindly reserving a server wastes compute resource and reduces system throughput. Therefore, to deploy training algorithms

online, these problems require RL to train robust policies that ensure safety [110, 3, 162].

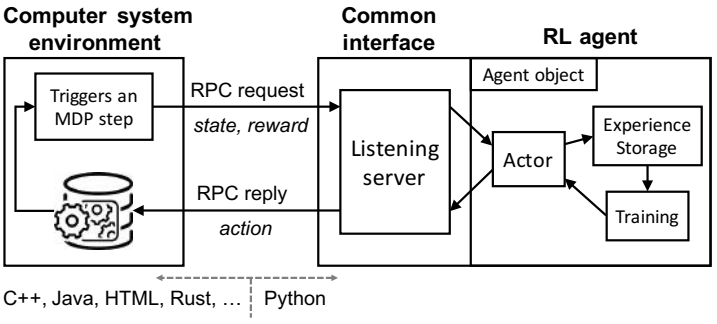
6.3.4 Understandability over Existing Heuristics

As in other areas of ML, interpretability plays an important role in making learning techniques practical. However, in contrast to perception-based problems or games, for system problems, many reasonable good heuristics exist. For example, every introductory course to computer science features a basic scheduling algorithm such as FIFO. These heuristics are often easy to understand and to debug, whereas a learned approach is often not. Hence, making learning algorithms in systems as debuggable and interpretable as existing heuristics is a key challenge. Here, a unique opportunity is to build hybrid solutions, which combine learning-based techniques with traditional heuristics. Existing heuristics can not only help to bootstrap certain problems, but also help with safety and generalizability. For example, a learned scheduling algorithm could fall back to a simple heuristic if it detects that the input distribution significantly drifted.

6.4 The Park Platform

Park follows a standard request-response design pattern. The backend system runs continuously and periodically send requests to the learning agent to take control actions. To connect the systems to the RL agents, Park defines a common interface and hosts a server that listens for requests from the backend system. The backend system and the agent run on different processes (which can also run on different machines) and they communicate using remote procedure calls (RPCs). This design essentially structures RL as a service. Figure 6-2 provides an overview of Park.

Real system interaction loop. Each system defines its own events to trigger an MDP step. At each step, the system sends an RPC request that contains the current state and a reward corresponding to the *last* action. Upon receiving the request, the Park server invokes the RL agent. The implementation of the agent is up to the users (e.g., feature extraction, training process, inference methods). In Figure 6-2, Algorithm 1 depicts this interaction process. Notice that invoking the agent incurs a physical delay for the RPC response from the server. Depending on the underlying implementation, the system may or may not wait synchronously during this delay. For non-blocking RPCs, the state observed by the agent can be stale (which



Algorithm 1 Interface for real system interaction.

```
1: def env.run(agent):
2:     while not done:
3:         state, reward, done = server.listen()
4:         # reward for the previous action
5:         action = agent.act(state, reward, done)
6:         server.reply(action)
```

Algorithm 2 Interface for simulated interaction.

```
1: def env.step(action):
2:     # OpenAI Gym style of interaction
3:     server.reply(action)
4:     state, reward, done = server.listen()
5:     return state, reward, done
```

Figure 6-2: Park architects an RL-as-a-service design paradigm. The computer system connects to an RL agent through a canonical request/response interface, which hides the system complexity from the RL agent. Algorithm 1 describes a cycle of the system interaction with the RL agent. By wrapping with an agent-centric environment in Algorithm 2, Park's interface also supports OpenAI Gym [53] like interaction for simulated environments.

typically would not occur in simulation). On the other hand, if the system makes blocking RPC requests, then taking a long time to compute an action (e.g., while performing MCTS search [278]) can degrade the system performance. Designing high-performance RL training or inference agents in a real computer system should explicitly take this delay factor into account.

Wrapper for simulated interaction. By wrapping the request-response interface with a shim layer, Park also supports an "agent-centric" style of interaction advocated by OpenAI Gym [53]. In Figure 6-2, Algorithm 2 outlines this option in simulated system environments. The agent explicitly steps the environment forward by sending the action to the underlying system through the RPC response. The interface then waits on the RPC server for the next action request. With this interface, we can directly reuse existing off-the-shelf RL training implementations benchmarked on Gym [83].

Scalability. The common interface allows multiple instances of a system environment to run concurrently. These systems can generate the experience in parallel to speed up RL training. As a concrete example, to implement IMPALA [93] style of distributed RL training, the interface takes multiple actor instance at initialization. Each actor corresponds to an environment instance. When receiving an RPC request, the interface then uses the RPC request ID to route the request to the corresponding actor. The actor reports the experience to the learner (globally maintained for all agents) when the experience buffer reaches the batch size for training and parameter updating.

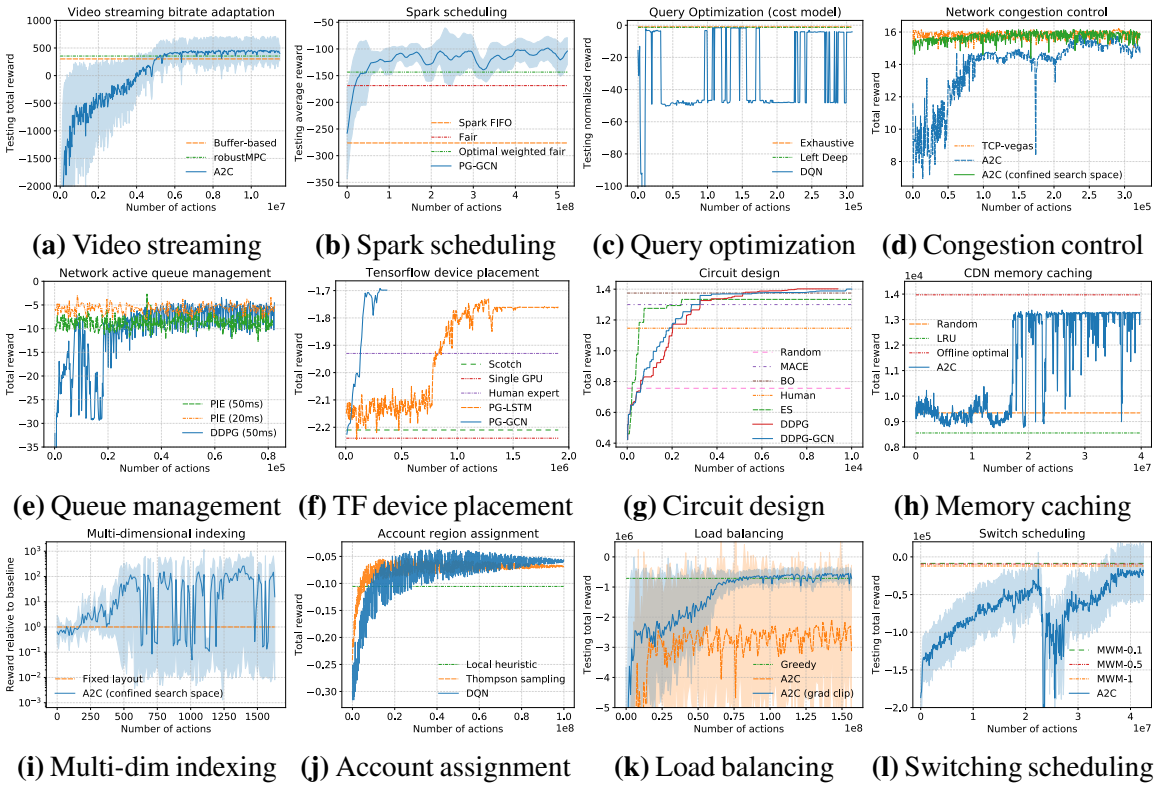
Environments. Table 6.2 provides an overview of 12 environments that we have imple-

Environment	Type	State space	Action space	Reward	Step time	Challenges (§6.3)
Adaptive video streaming	Real/sim	Past network throughput measurements, playback buffer size, portion of unwatched video	Bitrate of the next video chunk	Combination of resolution and stall time	Real: ∼3s Sim: ∼1ms	Input-driven variance, slow interaction time
Spark cluster job scheduling	Real/sim	Cluster and job information as features attached to each node of the job DAGs	Node to schedule next	Runtime penalty of each job	Real: ∼5s Sim: ∼5ms	Input-driven variance, state representation, infinite horizon, reality gap
SQL database query optimization	Real	Query graph with predicate and table features on nodes, join attributes on edges	Edge to join next	Cost model or actual query time	∼5s	State representation, reality gap
Network congestion control	Real	Throughput, delay and packet loss	Congestion window and pacing rate	Combination of throughput and delay	∼10ms	Sparse space for exploration, safe exploration, infinite horizon
Network active queue management	Real	Past queuing delay, enqueue/dequeue rate	Drop rate	Combination of throughput and delay	∼50ms	Infinite horizon, reality gap
Tensorflow device placement	Real/sim	Current device placement and runtime costs as features attached to each node of the job DAGs	Updated placement of the current node	Penalty of runtime and invalid placement	Real: ∼2s Sim: ∼10ms	State representation, reality gap
Circuit design	Sim	Circuit graph with component ID, type and static parameters as features on the node	Transistor sizes, capacitance and resistance of each node	Combination of bandwidth, power and gain	∼2s	State representation, sparse space for exploration
CDN memory caching	Sim	Object size, time since last hit, cache occupancy	Admit/drop	Byte hits	∼2ms	Input-driven variance, infinite horizon, safe exploration
Multi-dim database indexing	Real	Query workload, stored data points	Layout for data organization	Query throughput	∼30s	State/action representation, infinite horizon
Account region assignment	Sim	Account language, region of request, set of linked websites	Account region assignment	Serving cost in the future	∼1ms	State/action representation
Server load balancing	Sim	Current load of the servers and the size of incoming job	Server ID to assign the job	Runtime penalty of each job	∼1ms	Input-driven variance, infinite horizon, safe exploration
Switch scheduling	Sim	Queue occupancy for input-output port pairs	Bijection mapping from input ports to output ports	Penalty of remaining packets in the queue	∼1ms	Action representation

Table 6.2: Overview of the computer system environments supported by Park platform.

mented in Park. Appendix E contains the detailed descriptions of each problem, its MDP definition, and explanations of why RL could provide benefits in each environment. Seven of the environments use real systems in the backend (see Table 6.2). For the remaining five environments, which have well-understood dynamics, we provide a simulator to facilitate easier setup and faster RL training. For these simulated environments, Park uses real-world traces to ensure that they mimic their respective real-world environments faithfully. For example, for the CDN memory caching environment, we use an open dataset containing 500 million requests, collected from a public CDN serving top-ten US websites [42]. Given the request pattern, precisely simulating the dynamics of the cache (hits and evictions) is straightforward. Moreover, for each system environment, we also summarize the potential challenges from §6.3.

Extensibility. Adding a new system environment in Park is straightforward. For a new system, it only needs to specify (1) the state-action space definition (e.g., tensor, graph, pow-



(a) Video streaming **(b)** Spark scheduling **(c)** Query optimization **(d)** Congestion control

(e) Queue management **(f)** TF device placement **(g)** Circuit design **(h)** Memory caching

(i) Multi-dim indexing **(j)** Account assignment **(k)** Load balancing **(l)** Switching scheduling

Figure 6-3: Benchmarks of the existing standard RL algorithms on Park environments. In y-axes, "testing" means the agents are tested with unseen settings in the environment (e.g., newly sampled workload unseen during training, unseen job patterns to schedule, etc.). The heuristic or optimal policies are provided as comparison.

erset, etc.), (2) the event to trigger an MDP step, at which it sends an RPC request and (3) the function to calculate the reward feedback. From the agent's perspective, as long as the state-action space remains similar, it can use the same RL algorithm for the new environment. The common interface decouples the development of an RL agent from the complexity of the underlying system implementations.

6.5 Benchmark Experiments

We train the agents on the system environments in Park with several existing RL algorithms, including DQN [220], A2C [219], Policy Gradient [286] and DDPG [191]. When available, we also provide the existing heuristics and the optimal policy (specifically designed for each environment) for comparison. The details of hyperparameter tunings, agent architecture and system configurations are in Appendix G. Figure 6-3 shows the experiment results. As a sanity check, the performance of the RL policy improves over time from random initialization

in all environments.

Room for improvement. We highlight system environments that exhibit unstable learning behaviors and potentially have large room for performance improvement. We believe that the instability observed in some of the environments are due to fundamental challenges that require new training procedure. For example, the policy in Figure 6-3h is unable to smoothly converge partially because of the variance caused by the cache arrival input sequence (§6.3.2). For database optimization in Figure 6-3c, RL methods that make one-shot decisions, such as DQN, do not converge to a stable policy; combining with explicit search [210] may improve the RL performance. In network congestion control, random exploration is inefficient to search the large state space that provides little reward gradient. This is because unstable control policies (which widely spans the policy space) cannot drain the network queue fast enough and results in indistinguishable (e.g., delay matches max queuing delay) poor rewards (as discussed in §6.3.1). Confining the search space with domain knowledge significantly improves learning efficiency in Figure 6-3d (implementation details in Appendix G). For Tensorflow device placement in Figure 6-3f, using graph convolutional neural networks (GCNs) [170] for state encoding is natural to the problem setting and allows the RL agent to learn more than 5 times faster than using LSTM encodings [217]. Using more efficient encoding may improve the performance and generalizability further.

For some of the environments, we were forced to simplify the task to make it feasible to apply standard RL algorithms. Specifically, in CDN memory caching (Figure 6-3h), we only use a small 1MB cache (typical CDN caches are over a few GB); a large cache causes the reward (i.e., cache hit/miss) for an action to be significantly delayed (until the object is evicted from the cache, which can take hundreds of thousands of steps in large caches) [42]. For account region assignment in Figure 6-3j, we only allocate an account at initialization (without further migration). Active migration at runtime requires a novel action encoding (how to map any account to any region) that is scalable to arbitrary size of the action space (since the number of accounts keep growing). In Figure 6-3l, we only test with a small switch with 3×3 ports, because standard policy network cannot encode or efficiently search the exponentially large action space when the number of ports grow beyond 10×10 (as described in §6.3.1). These tasks are examples where applying RL in realistic settings may require inventing new learning techniques (§6.3).

6.6 Conclusion

Park provides a common interface to a wide spectrum of real-world systems problems, and is designed to be easily-extensible to new systems. Through Park, we identify several unique challenges that may fundamentally require new algorithmic development in RL. The platform makes systems problems easily-accessible to researchers from the machine learning community so that they can focus on the algorithmic aspect of these challenges. We have open-sourced Park along with the benchmark RL agents and the existing baselines in https://github.com/park-project.

Chapter 7

Conclusion

This thesis takes a first step towards building computer systems that can learn to efficiently optimize performance on their own through modern reinforcement learning. We demonstrate that learning-based systems are able to achieve superior performance than human-engineered heuristics in a wide range of environments, from bitrate adaptation in video streaming (§3) to cluster scheduling for complex data processing jobs (§4). The key advantage of these data-driven approaches is their ability to tailor for the specific deployment settings (e.g., network types, workload patterns, etc.) and to automatically adapt to challenging environments, especially those for which the fixed heuristics were not custom-designed in advance.

While building these learning-based systems, we have also identified several unique problem structures that fundamentally require new machine learning algorithms. §5 describes the details of an input-driven problem abstraction that commonly occurs in many systems. The problem structure and our new technique — the input-dependent baseline methods for problems with external input processes — are applicable beyond networking and systems to other domains such as robust robotic control with external disturbance.

To facilitate future research endeavor, We have developed an open, extensible platform that uses a common RL interface to connect to 12 system environments, ranging across networking, databases, and distributed systems (§6). This platform lowers the barrier to entry for machine learning researchers by bypassing the low-level system implementation details. Yet, it exposes many unique machine learning challenges in those system environments. With this platform and the easy-to-compare benchmarks, we hope to help facilitate more interaction across research communities and enable researchers to invent and evaluate different AI approaches on real-world networking and systems problems.

With the rising interest of applying machine learning and data-driven methods to networking systems, many pristine and interesting research directions and problems lie ahead. To conclude this thesis, we outline some important research directions for the next future steps.

7.1 Looking Forward

Safe exploration and deployment. RL fundamentally requires the agent to explore different actions in order to compare the empirical returns and learn. Context-free random explorations, such as ϵ-greedy or entropy-based policy randomization, can bring the system into an unsafe region and lead to catastrophic outcomes. For example, when load balancing among several heterogeneous servers (i.e., servers with different processing rate), random workload assignment can *systematically* overwhelm the slow server while leaving fast servers idle, which effectively reduces the service capacity. Moreover, data-driven systems should ideally learn *in situ* — train in a live system directly online — in order to avoid unforeseen scenarios in an offline simulator [325]. Random exploration thus becomes a critical danger to a running system and may even inhibit further learning (e.g., by creating an insurmountable backlog of work that halts the generation of new learning experience).

Safe AI in general, and in particular safe exploration in RL, is an active research area [244, 110, 44]. In the context of building software systems, however, we emphasize learning methods that can provide provable guarantees on system behavior and runtime performance. Along this direction, existing work has proposed to use formal verification [76] to bound the system behavior when the system dynamics and the control policy can be accurately modeled beforehand [21]. The fundamental challenge of extending this approach to deep RL is that the agent's superior performance often comes at the cost of using complex and not (yet) interpretable neural network; it is still an open research area for how to properly integrate the formal verification framework. There has also been some ongoing work concerning a relaxed setting, where a system contains some simple, well-defined policies that can bring the system back to normal from any dangerous state [205]. In the load balancing example, any work conservative policy (e.g., a join-shortest-queue heuristic) is guaranteed to drain the server queues (over time) as long as the system load is below its capacity. In many systems, such a safeguarding policy is not necessarily performant, but is often easy to find. We leveraged this safeguard as "training wheels" during the RL random exploration process: it takes over the

control when some safety condition is violated (e.g., some server queue is dangerously large). The training wheels prevent the novice RL agent from ever transitioning into the unsafe state space. We resume the exploration and training after the fallback policy brings the system back to a normal or blank state (e.g., all servers are empty). Following this line of early work, many research questions are still open. For example, fallback policy can often dominate the system and prevent RL exploration (e.g., when initial RL policy is weak and easily triggers safety violation), how can we leverage the fallback policy experience for RL training as well? Can we robustly learn the system dynamics and find a safeguarding policy when the dynamics of the system is not unknown a priori? What is the system architecture and software design principles with the safeguarding policy in place and constantly monitoring?

Policy interpretation. Somewhat coupled with random exploration is the unpredictable nature of deep neural networks. Skepticism is ever present for applying modern RL in mission critical systems, as even the well-trained neural networks can sometimes lead to catastrophic outcomes in corner cases [137], or can be exploited by adversarial attacks [27]. Many ongoing efforts are trying to translate neural network policies into deterministic, rule-based models, such as decision trees [34, 215]. In a short term, these techniques can serve as a reliable middle ground for deploying neural network policies in production (e.g., §3.6.5 provides a simple translation example for production systems). For these techniques, the key challenges is to come up with efficient pruning methods for rule generation (e.g., minimum branches in decision trees) and to find *human intuitive* models after translation. Another line of work, recently represented by neural network surgery [251], tries to directly probe into neural networks to understand their response to certain type of inputs (e.g., control action under certain state space). We can potentially leverage these techniques to understand how RL trained policies outperform existing method. With the model debugging and translation techniques, we can envision building a two-stage production system, where we perform policy exploration, training, debugging and rule translation in the first small scale environment, and then deploy the verified rule-based model on the second fleet of services with larger scale.

Model-based efficient learning. In this thesis, we mostly focus on model-free RL where we assume the complex dynamics (i.e., state transition map) is unknown or hard to model. However, system operators often *do* understand most parts of the system dynamics very well; it is usually only a small component (e.g., network throughput fluctuation, server capacity variation due to interfering workloads, etc.) that creates all the uncertainty and complexity. For example, in video bitrate adaptation (§3), the dynamics of the buffer is perfectly determined

once the bandwidth is given — i.e., the resulting video quality and stall time are exactly known for each bitrate decision. Given the network bandwidth distribution, finding the optimal bitrate decisions is a dynamic programming problem (e.g., the online optimal calculation in §3.5.4). Intuitively, only learning the dynamics or response of the unknown component, rather than learning the control policy for the entire system end-to-end, should have lower sample complexity (i.e., need a smaller amount of data to train). Therefore, efficient model-based RL is viable to train in slow-to-interact environments (e.g., waiting to download a real video chunk without simulation, executing the actual job binary in a cluster). In general, however, it is yet unclear how to optimally integrate a learned prediction model — which may be inherently inaccurate — with the end-to-end system control. In the bitrate adaptation case, a good controller should incorporate the network bandwidth prediction error. Large error may be an indicator for network uncertainties and we should favor towards smaller bitrate to build up the buffer. But most popular neural network models only output the predicted values rather than its prediction errors or uncertainties under different inputs; building a general framework for neural network uncertainty estimation largely remains an active research area [179].

Adapting to changing workloads. The high level promise of RL is to automatically adapt and optimize for different kinds of system dynamics. In practice, however, the RL-trained agent can only generalize to a narrow set of environments that are share similar characteristics as the training environment [325]. This restriction is common to most current machine learning methods. Moreover, current RL suffers from high sample complexity during training, which prevents agents from agilely adapt to fast changing environment online. Facing these constraints and challenges, there are several research directions worth investigating:

1. Meta learning concerns learning a model that can adapt to a *new* learning task with as little data as possible, as opposed to "overfitting" any particular learning task [101, 71, 307]. Recent prominent methods in this domain include gradient-based approaches such as MAML [101], approaches based on recurrent models [264] and more. In the RL setting, researchers have trained agents that can adapt to changing objectives (e.g., different headings or a navigation robot) [101, §5.3]. However, for changing workloads, the learning signal may be much more subtle — it may be hidden in the change of state transition or reward function and the exact change is unknown for the meta learner. It is worth investigating the principles of modeling environment changes and integrate them with meta learning.
2. Mixture of experts considers training and properly switching among a group of mod-

els — each of which is specialized to solve a particular learning task — in order to master an ensemble of tasks [275]. This method provides a promising framework for learning multiple control agents to handle different kinds of changing workload scenarios. Many fundamental questions are still open. For example, what are the criteria for spawning a new expert; are the criteria learned or provisioned? If the model capacity is limited (e.g., constrained by GPU memory limit), how can we deprecate old experts? What are the principles for avoiding training duplicated experts for similar workloads?

3. Hierarchical RL seeks to co-train a high-level planning agent with a low-level action-taking agent [306, 89, 33]. Traditionally researchers apply this approach to tasks that require sequential planning in a long time horizon (e.g., complex games with multiple sub-tasks to solve). Based on different stage of a task, the planner adaptively decides how to activate or tune the action-maker at the low level. This separation of controller may be applicable to dealing with the change in workloads. In principle, one can imagine a high-level agent that builds a model for different families of workload patterns and adaptively tunes the low level agent to adjust its policy for different actions.

Model control and federated learning. Data-driven distributed control systems (e.g., Pensieve in §3) involve a large number of edge devices (e.g., mobile phones) making decisions across heterogeneous environments. This distributed setting raises interesting questions about how best to collect observations at the edge, what data to process locally and what to send to the cloud for federated learning, and how to train and manage control models across heterogeneous environments. From a statistical perspective, more diverse experience data increases the quality of learned models. However, shipping all data to a centralized datacenter for training can be expensive in terms of bandwidth, data processing, and storage. Only useful data should need be analyzed and processed when updating control models. Hence, balancing the abundance of new data with the costs of data transfer, storage, and computation creates an interesting opportunity and challenge for future control data management systems.

Appendix A

Decima Implementation Details

Algorithm 3 presents the pseudocode for Decima's training procedure as described in §4.5.3. In particular, line 3 samples the episode length τ from an exponential distribution, with a small initial mean τ_{mean}. This step terminates the initial episodes early to avoid wasting training time (see challenge #1 in §4.5.3). Then, we sample a job sequence (line 4) and use it to collect N episodes of experience (line 5). Importantly, the baseline b_k in line 8 is computed with the *same* job sequence to reduce the variance caused by the randomness in the job arrival process (see challenge #2 in §4.5.3). Line 10 is the policy gradient REINFORCE algorithm described in Eq. (4.3). Line 13 increases the average episode length (i.e., the curriculum learning procedure for challenge #1 in §4.5.3). Finally, we update Decima's policy parameter θ on line 14.

Our neural network architecture is described in §4.6.1, and we set the hyperparameters in Decima's training as follows. The number of incoming jobs is capped at 2000, and the episode termination probability decays linearly from 5×10^{-7} to 5×10^{-8} throughout training. The learning rate α is 1×10^{-3} and we use Adam optimizer [169] for gradient descent. Finally, we train Decima for at least 50,000 iterations for all experiments.

For continuous job arrivals, an average reward formulation, which maximizes $\lim_{T\to\infty} \mathbb{E}[1/T \sum_{k=0}^{T} r_k]$, is a better objective than the total reward formulation. To convert the objective from the sum of rewards to the average reward, we replace the reward r_k with a *differential reward*. Operationally, at every step k, the environment modifies the reward to the agent as $r_k \leftarrow r_k - \hat{r}$, where $\hat{r}$ is a moving average of the rewards across a large number of previous steps (across many training episodes). In our implementation, the moving window for estimating $\hat{r}$ spans 10^5 time steps. With this modification, we can reuse the same

Algorithm 3 Policy gradient method used to train Decima.

for each iteration **do**
  $\Delta\theta \leftarrow 0$
  Sample episode length $\tau \sim \text{exponential}(\tau_{\text{mean}})$
  Sample a job arrival sequence
  Run episodes $i = 1,...,N$:
    $\{s_1^i, a_1^i, r_1^i, ..., s_\tau^i, a_\tau^i, r_\tau^i\} \sim \pi_\theta$
  Compute total reward: $R_k^i = \sum_{k'=k}^{\tau} r_{k'}^i$
  for k = 1 to τ **do**
    compute baseline: $b_k = \frac{1}{N} \sum_{i=1}^{N} R_k^i$
    for i = 1 to N **do**
      $\Delta\theta \leftarrow \Delta\theta + \nabla_\theta \log \pi_\theta(s_k^i, a_k^i)(R_k^i - b_k)$
    end for
  end for
  $\tau_{\text{mean}} \leftarrow \tau_{\text{mean}} + \epsilon$
  $\theta \leftarrow \theta + \alpha \Delta\theta$
end for

policy gradient method as in Equation (2.1) and (2.2) to find the optimal policy. Sutton and Barto [285, §10.3, §13.6] describe the mathematical details on how this approach optimizes the average reward objective.

We implemented Decima's training framework using TensorFlow [1], and we use 16 workers to compute episodes with the same job sequence in parallel during training. Each training iteration, including interaction with the simulator, model inference and model update from all training workers, takes roughly 1.5 seconds on a machine with Intel Xeon E5-2640 CPU and Nvidia Tesla P100 GPU.

All experiments in §4.7 are performed on test job sequences unseen during training (e.g., unseen TPC-H job combinations, unseen part of the Alibaba production trace, etc.).

Appendix B

Illustration of Variance Reduction in 1D Grid World

Consider a walker in a 1D grid world, where the state $s_t \in \mathbb{Z}$ at time t denotes the position of the walker, and action $a_t \in \{-1,+1\}$ denotes the intent to either move forward or backward. Additionally let $z_t \in \{-1,+1\}$ be a uniform i.i.d. "exogenous input" that perturbs the position of the walker. For an action a_t and input z_t, the state of the walker in the next step is given by $s_{t+1} = s_t + a_t + z_t$. The objective of the game is to move the walker forward; hence, the reward is $r_t = a_t + z_t$ at each time step. $\gamma \in [0,1]$ is a discount factor.

While the optimal policy for this game is clear ($a_t = +1$ for all t), consider learning such a policy using policy gradient. For simplicity, let the policy be parametrized as $\pi_\theta(a_t = +1|s_t) = e^\theta/(1+e^\theta)$, with θ initialized to 0 at the start of training. In the following, we evaluate the variance of the policy gradient estimate at the start of training under (i) the standard value function baseline, and (ii) a baseline that is the expected cumulative reward conditioned on all future z_t inputs.

Variance under standard baseline. The value function in this case is identically 0 at all states. This is because $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t] = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t (a_t + z_t)] = 0$ since both actions a_t and inputs z_t are i.i.d. with mean 0. Also note that $\nabla_\theta \log \pi_\theta(a_t = +1) = 1/2$ and $\nabla_\theta \log \pi_\theta(a_t = -1) = -1/2$; hence $\nabla_\theta \log \pi_\theta(a_t) = a_t/2$. Therefore the variance of the policy gradient estimate can be written as

$$V_1 = \mathrm{Var}\left[\sum_{t=0}^{\infty} \frac{a_t}{2} \sum_{t'=t}^{\infty} \gamma^{t'} r_{t'}\right] = \mathrm{Var}\left[\sum_{t=0}^{\infty} \frac{a_t}{2} \sum_{t'=t}^{\infty} \gamma^{t'} (a_{t'} + z_{t'})\right]. \tag{B.1}$$

Variance under input-dependent baseline. Now, consider an alternative "input-dependent" baseline $V(s_t|\boldsymbol{z})$ defined as $\mathbb{E}[\sum_{t=0}^{\infty}\gamma^t r_t|\boldsymbol{z}]$. Intuitively this baseline captures the average reward incurred when experiencing a particular fixed $\boldsymbol{z}$ sequence. We refer the reader to §5.4 for a formal discussion and analysis of input-dependent baselines. Evaluating the baseline we get $V(s_t|\boldsymbol{z}) = \mathbb{E}[\sum_{t=0}^{\infty}\gamma^t r_t|\boldsymbol{z}] = \sum_{t=0}^{\infty}\gamma^t z_t$. Therefore the variance of the policy gradient estimate in this case is

$$V_2 = \mathrm{Var}\left[\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}r_{t'} - \sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right] = \mathrm{Var}\left[\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}a_{t'}\right)\right]. \tag{B.2}$$

Reduction in variance. To analyze the variance reduction between the two cases (Equations (B.1) and (B.2)), we note that

$$V_1 = V_2 + \mathrm{Var}\left[\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right] + 2\mathrm{Cov}\left(\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}a_{t'}\right), \sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right) \tag{B.3}$$

$$= V_2 + \mathrm{Var}\left[\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right]. \tag{B.4}$$

This follows because

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right] = \sum_{t=0}^{\infty}\sum_{t'=t}^{\infty}\frac{\gamma^{t'}}{2}\mathbb{E}[a_t z_{t'}] = 0, \quad \text{and}$$

$$\mathbb{E}\left[\left(\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}a_{t'}\right)\right)\left(\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right)\right] = \sum_{t_1=0}^{\infty}\sum_{t_1'=t_1}^{\infty}\sum_{t_2=0}^{\infty}\sum_{t_2'=t_2}^{\infty}\mathbb{E}\left[\frac{a_{t_1}a_{t_1'}a_{t_2}z_{t_2'}}{4}\gamma^{t_1'+t_2'}\right] = 0.$$

Therefore the covariance term in Equation (B.3) is 0. Hence the variance reduction from Equation (B.4) can be written as

$$V_1 - V_2 = \mathrm{Var}\left[\sum_{t=0}^{\infty}\frac{a_t}{2}\left(\sum_{t'=t}^{\infty}\gamma^{t'}z_{t'}\right)\right] = \sum_{t_1=0}^{\infty}\sum_{t_1'=t_1}^{\infty}\sum_{t_2=0}^{\infty}\sum_{t_2'=t_2}^{\infty}\mathbb{E}\left[\frac{a_{t_1}a_{t_2}z_{t_1'}z_{t_2'}}{4}\gamma^{t_1'+t_2'}\right]$$

$$= \sum_{t_1=0}^{\infty}\sum_{t_1'=t_1}^{\infty}\mathbb{E}\left[\frac{a_{t_1}^2 z_{t_1'}^2}{4}\gamma^{2t_1'}\right] = \frac{\mathrm{Var}(a_0)\mathrm{Var}(z_0)}{4(1-\gamma^2)^2}.$$

Thus the input-dependent baseline reduces variance of the policy gradient estimate by an amount proportional to the variance of the external input. In this toy example, we have chosen z_t to be binary-valued, but more generally the variance of z_t could be arbitrarily large and might be a dominating factor of the overall variance in the policy gradient estimation.

Appendix C

Input-Dependent Baseline for TRPO

We show that the input-dependent baselines are bias-free for Trust Region Policy Optimization (TRPO) [268].

Preliminaries. Stochastic gradient descent using Equation (5.1) does not guarantee consistent policy improvement in complex control problems. TRPO is an alternative approach that offers monotonic policy improvements, and derives a practical algorithm with better sample efficiency and performance. TRPO maximizes a surrogate objective, subject to a KL divergence constraint:

$$\underset{\theta}{\text{maximize}} \quad \mathbb{E}_{\substack{s\sim\rho_{\pi_{\text{old}}} \\ a\sim\pi_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)} Q_{\pi_{\text{old}}}(s,a)\right] \tag{C.1}$$

$$\text{subject to} \quad \mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}}[D_{\text{KL}}(\pi_{\text{old}}(\cdot|s)||\pi_\theta(\cdot|s))] \leq \delta, \tag{C.2}$$

in which δ serves as a step size for policy update. Using a baseline in the TRPO objective, i.e. replacing $Q_{\pi_{\text{old}}}(s,a)$ with $Q_{\pi_{\text{old}}}(s,a)-b(s)$, empirically improves policy performance [269].

Similar to Theorem 2, we generalize TRPO to input-driven environments, with $\rho_\pi(\omega,\boldsymbol{z}) = \sum_{t=0}^{\infty}[\gamma^t \Pr(\omega_t=\omega, z_{t:\infty}=\boldsymbol{z})]$ denoting the discounted visitation frequency of the observation ω and input sequence $\boldsymbol{z}$, and $Q(\omega,a,\boldsymbol{z}) = \mathbb{E}\left[\sum_{l=0}^{\infty}\gamma^l r_{t+l} \,\middle|\, \omega_t=\omega, a_t=a, z_{t:\infty}=\boldsymbol{z}\right]$. The TRPO objective becomes $\mathbb{E}_{(\omega,\boldsymbol{z})\sim\rho_{\text{old}},a\sim\pi_{\text{old}}}[Q_{\pi_{\text{old}}}(\omega,a,\boldsymbol{z})\pi_\theta(a|\omega)/\pi_{\text{old}}(a|\omega)]$, and the constraint is $\mathbb{E}_{(\omega,\boldsymbol{z})\sim\rho_{\pi_{\text{old}}}}[D_{\text{KL}}(\pi_{\text{old}}(\cdot|s)||\pi_\theta(\cdot|s))] \leq \delta$.

Theorem 3. *An input-dependent baseline does not change the optimal solution of the opti-*

mization problem in TRPO, that is

$$argmax_\theta \mathbb{E}_{(\omega,\boldsymbol{z})\sim\rho_{old},a\sim\pi_{old}}\left[\frac{\pi_\theta(a|\omega)}{\pi_{old}(a|\omega)}Q_{\pi_{old}}(\omega,a,\boldsymbol{z})\right] = argmax_\theta \mathbb{E}_{(\omega,\boldsymbol{z})\sim\rho_{old},a\sim\pi_{old}}\left[\frac{\pi_\theta(a|\omega)}{\pi_{old}(a|\omega)}(Q_{\pi_{old}}(\omega,a,\boldsymbol{z})-b(\omega,\boldsymbol{z}))\right]. \quad \text{(C.3)}$$

Proof.

$$\begin{aligned}\mathbb{E}_{(\omega,\boldsymbol{z})\sim\rho_{\text{old}},a\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|\omega)}{\pi_{\text{old}}(a|\omega)}b(\omega,\boldsymbol{z})\right] &= \sum_\omega\sum_{\boldsymbol{z}}\rho_{\text{old}}(\omega,\boldsymbol{z})\sum_a \pi_{\text{old}}(a|\omega)\left[\frac{\pi_\theta(a|\omega)}{\pi_{\text{old}}(a|\omega)}b(\omega,\boldsymbol{z})\right] \\ &= \sum_\omega\sum_{\boldsymbol{z}}\rho_{\text{old}}(\omega,\boldsymbol{z})\sum_a \pi_\theta(a|\omega)b(\omega,\boldsymbol{z}) \\ &= \sum_\omega\sum_{\boldsymbol{z}}\rho_{\text{old}}(\omega,\boldsymbol{z})b(\omega,\boldsymbol{z}),\end{aligned}$$

which is independent of θ. Therefore, $b(\omega,\boldsymbol{z})$ does not change the optimal solution to the optimization problem. □

Appendix D

Input-Dependent Baseline Implementation Details

In our discrete-action environments (§5.6.1), we build 10-value networks and a meta-baseline using MAML [101], both on top of the OpenAI A2C implementation [83]. We use $\gamma = 0.995$ for both environments. The actor and the critic networks have 2 hidden layers, with 64 and 32 hidden neurons on each. The activation function is ReLU [226] and the optimizer is Adam [64]. We train the policy with 16 (synchronous) parallel agents. The learning rate is 1^{-3}. The entropy factor [219] is decayed linearly from 1 to 0.001 over 10,000 training iterations. For the meta-baseline, the meta learning rate is 1^{-3} and the model specification has five step updates, each with learning rate 1^{-4}. The model specification step in MAML is performed with vanilla stochastic gradient descent.

We introduce disturbance into our continuous-action robot control environments (§5.6.2). For the walker with wind (Figure 5-1c), we randomly sample a wind force in $[-1,1]$ initially and add a Gaussian noise sampled from $\mathcal{N}(0,1)$ at each step. The wind is bounded between $[-10,10]$. The episode terminates when the walker falls. For the half-cheetah with floating tiles, we extend the number of piers from 10 in the original environment [71] to 50, so that the agent remains on the pathway for longer. We initialize the tiles with damping sampled uniformly in $[0,10]$. For the 7-DoF robot arm environments, we initialize the target to randomly appear within $(-0.1,-0.2,0.5),(0.4,0.2,-0.5)$ in 3D. The position of the target is perturbed with a Gaussian noise sampled from $\mathcal{N}(0,0.1)$ in each coordinate at each step. We bound the position of the target so that it is confined within the arm's reach. The episode length of all these environments are capped at 1,000.

We build the multi-value networks and meta-baseline on top of the TRPO implementation by OpenAI [83]. We turned off the GAE enhancement by using $\lambda = 1$ for fair comparison. We found that it makes only a small performance difference (within $\pm 5\%$ using $\lambda = \{0.95, 0.96, 0.97, 0.98, 0.99, 1\}$) in our environments. We use $\gamma = 0.99$ for all three environments. The policy network has two hidden layers, with 128 and 64 hidden neurons on each. The activation function is ReLU [226]. The KL divergence constraint δ is 0.01. The learning rate for value functions is 1^{-3}. The hyperparameter of training the meta-baseline is the same as the discrete-action case.

Appendix E

Detailed descriptions of Park environments

We describe the details of each system environment in Park. Formulating the MDP is an important, problem-specific step for applying RL to systems. Our guiding principle is to provide the RL agent with all the information and actions available to existing baselines schemes in that environment, such that the agent could at least express existing human-engineered policies. In most cases, the MDP formulations are straightforward and self-explanatory. However, some are more subtle (e.g., the Spark scheduling and TF device placement), and in these cases we adopt the formulations from prior work. In the following, each description is structured to follow the problem background, MDP abstraction of the system interaction, the existing system-specific baseline heuristic approach, and how RL is suitable for the system problem.

Adaptive video streaming. The volume of video streaming has reached almost 60% of all the Internet traffic [263]. Streaming video over variable-bandwidth networks (e.g., cellular network) requires the client to adapt the video bitrate to optimize the user experience. In industrial DASH standard [9], videos are divided into multiple chunks, each of which represents a few seconds of the overall video playback. Each chunk is encoded at several discrete bitrates, where a higher bitrate implies a higher resolution and thus a larger chunk size. For this problem, each MDP episode is a video playback with a particular network trace (i.e., a time series of network throughput). At each step, the agent observes the past network throughput measurement, the current video buffer size, and the remaining portion of the video. The action is the bitrate for the next video chunk. The objective is to maximize

the video resolution and minimize the stall (which occurs when download time of a chunk is larger than the current buffer size) and the reward is structured to be a linear combination of selected bitrate and the stall when downloading the corresponding chunk. Prior adaptive bitrate approaches construct heuristic based on the buffer and network observations. For example, a control theoretic based approach [329] conservatively estimates the network bandwidth and use model predictive control to choose the optimal bitrate over the near-term horizon. In practice, the network condition is hard to model and estimate, making a fixed, hard-coded model-based approach insufficient to adapt to changing network conditions [204, 11, 68].

Spark cluster job scheduling. Efficient utilization of expensive compute clusters matters for enterprises: even small improvements in utilization can save millions of dollars at scale [32]. Cluster schedulers are key to realizing these savings. A good scheduling policy packs work tightly to reduce fragmentation [304], prioritizes jobs according to high-level metrics such as user-perceived latency [305], and avoids inefficient configurations [100]. Since hand-tuning scheduling policies is uneconomic for many organizations, there has been a surge of interest in using RL to generate highly-efficient scheduling policies automatically [201, 61, 206].

We build our scheduling system on top of the Spark cluster manager [333]. Each Spark job is represented as a DAG of computation stages, which contains identical tasks that can run in parallel. The scheduler maps executors (atomic computation units) to the stages of each job. We modify Spark's scheduler to consult an external agent at each scheduling event (i.e., each MDP step). A scheduling event occurs when (1) a stage runs out of tasks (i.e., needs no more executors), (2) a stage completes, unlocking the tasks of one or more of its children, or (3) a new job arrives in the system. At each step, the cluster has some available executors and some runnable stages from pending jobs. Thus, the scheduling agent observes (1) the number of tasks remaining in the stage, (2) the average task duration, (3) the number of executors currently working on the stage, (4) the number of available executors, and (5) whether available executors are local to the job. This set of information is embedded as features on each node of the job DAGs. The scheduling action is two-dimensional—(1) which node to work on next and (2) how many executors to assign to the node. We structure the reward at step k as $r_k = -(t_k - t_{k-1}) J_k$, where J_k is the number of jobs in the system during the physical time interval $[t_{k-1}, t_k)$. Sum of such rewards penalize the agent in order to minimize the average job completion time. Park platform supports replaying an one-month industrial workload trace from Alibaba.

SQL Database query optimization. Queries in relational databases often involve retrieving data from multiple tables. The standard abstraction for combining data is through a sequential process that joins entries from two tables based on the provided filters (e.g., actor `JOIN` country `ON` actor.country_id = country.id) at each step. The most important factor that affects the query execution time is the order of joining the tables [175]. While any ordering leads to the same final result, an efficient ordering keeps the intermediate results small, which minimizes the number of entries to read and process. Finding the optimal ordering remains an active research area, because (1) the total number of orderings is exponential in the number of filters and (2) the size of intermediate results depends on hard-to-model relationship among the filters. There have been a few attempts to learn a query optimizer using RL [175, 237, 210].

Building the sequence of joins naturally fits in the MDP formulation. At each step, the agent observes the remaining tables to join as a query graph, where each node represents a table and the edges represent the join filters. The agent then decides which edge to pick (corresponds to a particular join) as an action. Park supports rewards from a cost model (a join cost estimate provided by commercial engines) and the final physical duration. In our implementation, we use Calcite [38] as the query optimization framework, which can serve as a connector to any database management system (e.g., Postgres [248]).

Network congestion control. Congestion control has been a perennial problem in networking for three decades [152], and governs when hosts should transmit packets. Transmitting packets too frequently leads to congestion collapse (affecting all users) [225] while over-conservative transmission schemes under-utilize the available network bandwidth. Good congestion control algorithms achieve high throughput and low delay while competing fairly for network bandwidth with other flows in the network. Various congestion control algorithms, including learning-based approaches [87, 326, 155], optimize for different objectives in this design space. It remains an open research question to design an end-to-end congestion control scheme that can automatically adapt to high-level objectives under different network condition [274].

We implement this enviroment using CCP [227], a platform for expressing congestion control algorithms in user-space. At each step, the agent observes the network state, including the throughput and delay.[1] The action is a tuple of pacing rate and congestion window. The pacing rate controls the inter-packet send time, while the congestion window limits the

[1] See Table 2 in [227] for full list.

total number of packets in-flight (sent but not acknowledged). We set our (configurable) action interval at 10 ms (suitable for typical Internet delays). Our reward function is adopted from the Copa [25] algorithm: `log(throughput) - log(delay)/2 - log(lost packets)`. This environment supports different network traces, from cellular networks to fixed-bandwidth links (emulated by Mahimahi [230]).

Network active queue management. In network routers and switches, active queue management (AQM) is a fundamental component that controls the queue size [28]. It monitors the queuing dynamics and decides to drop packets when the queue gets close to full [104]. The goal for AQM is to achieve high throughput and low delay for the packets passing through the queue. Designing a strong AQM policy that achieves this high-level objective for a wide range of network condition can be complex. Standard methods — such as PIE [142], based on PID control [26] — construct a policy for a low-level goal that maintains the queue size at a certain level. In our setting, the agent observes the queue size and network throughput measurement; it then sets the packet drop probability. The action interval is configurable (default interval 10 ms; can also go down to per packet level control). The reward can be configured as a penalty for the difference between observed and target queue size, or a weighted combination of network throughput and delay. Similar to the congestion control environment, we emulate the network dynamics using Mahimahi with a wide range of real-world network traces.

Tensorflow device placement. Large scale machine learning applications use distributed training environments, where neural networks are split across multiple GPUs and CPUs [218]. A key challenge for distributed training is how to split a large model across heterogeneous devices to speed up training. Determining an optimal device placement is challenging and involves intricate planning, particularly as neural networks grow in complexity and approach device memory limits [217]. Motivated by these challenges, several learning based approaches have been proposed [218, 217, 109, 5].

We build our placement system on top of Tensorflow [1]. Each model is represented as a computational graph of neural network operations. A placement scheme maps nodes to the available devices. We formulate the MDP as an iterative process of placement improvement steps [5]. At each step, the agent observes an existing placement graph and tries to improve its runtime by updating the placement at a particular node. The state observation is the computation graph of a Tensorflow model, with features attached to each node which include (1) estimated node run time (2) output tensor size (3) current device placement (4) flag of

the "current" node (5) flag if previously placed. The action places the current node on a device. Since the goal is to learn a policy that can iteratively improve placements, the reward $r_i = -(t_i - t_{i-1})$, where t_i is the runtime of the placement at step i. Park supports optimizing placements for graphs with hundreds of nodes across a configurable number of devices. To speedup training, Park also provides a simulator for the runtime of a device placement (based on measurements from prior executions, see Appendix A4 in [5] for details).

Circuits Design. Analog integrated circuits often involve complex non-linear models relating the transistor sizes and the performance metrics. Common practice for optimizing analog circuits relies on expensive simulations and tedious manual tuning from human experts [257]. Prior work has applied Bayesian optimization [197] and evolution strategy [192] as general black-box parameter tuning tools to optimize the analog circuit design pipeline. [312, 313] recently proposed to use RL to end-to-end optimize the circuit performance.

Park supports transistor-level analog circuit design [257], where the circuit schematic is fixed and the agent decides the component parameters. For each schematic, the agent observes a circuit graph where each node contains the component ID, type (e.g., NMOS or PMOS) and static parameters (e.g., Vth_0). The corresponding action is also a graph in which each node must specify the transistor size, capacitance and resistance. Then, the underlying HSPICE circuit simulator [287] returns a configurable combination of bandwidth, power and gain as a reward. We refer the readers to [312] for more details.

CDN memory caching. In today's Internet, the majority of content is served by Content Delivery Networks (CDNs) [233]. CDNs enable fast content delivery by caching content in servers near the users. To reduce the content retrieval cost from a data center, CDNs aim to maximize the fraction of bytes served locally from the cache, known as the byte hit ratio (BHR) [135]. The admission control problem of CDN caching fits naturally to the MDP setting. At each step when an uncached object arrives in the CDN, the agent observes the object size, the time since the previous visit (if available) and the remaining CDN cache size. The agent then takes an action to admit or drop the uncached object. To maximize BHR, the reward at each step is the total byte hits since the last action (i.e., counting the size of cached objects served). Coupled with the admission policy is an eviction policy that decides which cached object to remove in order to make room for a newly admitted object. By default, our environment uses a fixed least-recently-used policy for object eviction. The environment also supports training an eviction agent together with the admission agent (e.g., via multi-agent RL). Our setup includes a real world trace with 500 million requests collected from a

public CDN serving top-ten US websites [42].

Multi-dim database indexing. Many analytic queries to a database involve filter predicates (e.g., for query "`SELECT COUNT(*) FROM` TransactionTable `WHERE` state = CA `AND` day1 $\leq$ time $\leq$ day2", the filters are over state and time). Key to efficiently answering such range queries is the database index — the layout in which the underlying data is organized (e.g., sorted by a particular dimension). Many databases choose to index over multiple dimensions because analytics queries typically involve filters over multiple attributes [149, 332]. A good index is able to quickly return the query result by minimizing the number of points it scans. We found empirically that a well-chosen index can achieve query performance three orders of magnitude faster than one that is randomly selected. In practice, choosing a good index depends on the underlying data distribution and query workload at runtime; therefore, many current approaches rely on routine manual tuning by database administrators.

We consider the problem of selecting a multi-dimensional index from an RL perspective. We target grid-based indexes, where the agent is responsible for determining the size of the cells in the grid. We found that this type of index is competitive with traditional data structures, while offering more learnable parameters. At each step of our MDP formulation, the database receives a new set queries to run, and the agent has the opportunity to modify the grid layout. The observation consists of both the dataset (i.e., list of records in the database) and queries (i.e., a list of range boundaries for each attribute) that have arrived since the previous action. The environment then (1) samples a workload from a distribution that changes (slowly) over time, (2) uses it to evaluate the agent-generated index on a real column-oriented datastore, and (3) reports the query throughput (i.e., queries per second) as the agent's reward. Our environment uses a real dataset collected from Open Street Maps [236] with 105 million records, along with queries sampled from a set of relevant business analytic questions. In this setup, there are more than 7 trillion possible grid layouts that the agent must encode in its action space.

Account region assignment. Social network websites reduce access latency by storing data on servers near their users. For each user-uploaded piece of content, the service providers must decide which region to serve the content from. These decisions have a multitude of tradeoffs: storing a piece of content in many regions incurs increased storage cost (e.g., from a cloud service provider), and storing a piece of content in the "wrong" region can substantially increase access latency, diminishing the end user's experience [13].

To faithfully simulate this effect, our environment includes a real trace of one million

posts created on a medium-sized social network over eight months from eight globally distributed regions. Park supports two variants of the assignment task. First, the agent chooses a region assignment when a new piece of content is initially created. At each content creation step, the observation includes the language, outgoing links, and posting user (anonymized) ID. The action is one of the eight regions to store the content. The reward is based on the fraction of accesses from within the assigned region. This variant can be viewed as a contextual multi-armed bandit problem [194]. The second variant is similar to the first one, except that the agent has the opportunity to migrate any content to any region at the end of each 24 hour time period. The action space spans all possible mappings between the users and the regions. In this case, the agent must balance the cost of a migration against the potential decrease in access latency.

Server load balancing. In this simulated environment, an RL agent balances jobs over multiple heterogeneous servers to minimize the average job completion time. Jobs have a varying size that we pick from a Pareto distribution [123] with shape 1.5 and scale 100. The job arrival process is Poisson with an inter-arrival rate of 55. The number of servers and their service rates are configurable, resulting in different amounts of system load. For example, the default setting has 10 servers with processing rates ranging linearly from 0.15 to 1.05. In this setting, the load is 90%. The problem of minimizing average job completion time on servers with heterogeneous processing rates does not have a closed-form solution [133]; a widely-used heuristic is to join the shortest queue [74]. However, understanding the workload pattern can give a better policy; for example, one strategy is to dedicate some servers for small jobs to allow them finish quickly even if many large jobs arrive [98]. In this environment, upon each job arrival, the observed state is a vector $(j, s_1, s_2, ..., s_k)$, where j is the incoming job size and s_k is the size of queue k. The action $a \in \{1,2,...,k\}$ schedules the incoming job to a specific queue. The reward $r_i = \sum_n [\min(t_i, c_n) - t_{i-1}]$, where t_i is the time at step i and c_n is the completion time of active job n.

Switch scheduling. Switch scheduling poses a matching problem that transfers packets from the incoming ports to the outgoing ports [213, 272, 200]. This abstracted model is ubiquitous in many real world systems, such as datacenter routers [117] and traffic junctions [147]. At each step, the scheduling agent observes a matrix of queue lengths, with element (i, j) indicating the packet queue from input port i to output port j. The matching action is bijective — no two incoming packets shall pass through the same output ports. Notice that in

a switch with n input/output ports, the action space is the $n!$ possible bijection matchings.[2] After each scheduling round, one packet is transferred per each input/output port pair. The goal is to maximize switch throughput while minimizing packet delay. The optimal scheduling policy for this problem is unknown and is conjectured to depend on the underlying traffic pattern [272]. For example, the max weight matching policy empirically performs well only under high load [200]. Adapting the scheduling policy under dynamics load to optimize an arbitrary combination of throughput and delay is challenging.

[2]Typical routers can have 144 ports [124].

Appendix F

Benchmarking RL algorithms in Park

We follow the standard implementations of existing RL algorithms in OpenAI baselines [83]. We performed a coarse grid search for finding a good set of hyperparameters. Specifically, A2C [219] uses separated policy and value network and it has training batch of size 64. For discrete-action environments, A2C explores using an entropy term in policy loss [219, 323], with the entropy factor linearly decay from 1 to 0.01 in 10,000 iterations. For continuous-action environments, the policy network outputs the mean of a Gaussian distribution. The variance is controlled by an external factor that decays according to the same schedule as the discrete case. In Policy Gradient (PG) [286], we rollout 16 parallel trajectory and we use a simple time-based baseline averaging the return across the trajectories. DQN [220] employs a replay memory with size 50,000 and updates the target Q network every 100 steps. DDPG [191] uses a small replay memory with 2048 objects and updates the target networks every 1000 steps.

For feed forward networks, we use simple fully connected architecture with two hidden layers of 16 and 32 neurons. For recurrent neural networks, we use LSTM with 4 hidden layers. We use graph convolution neural networks (GCNs) [170] to encode the states that involve a graph structure. In particular, we modify the message passing kernel in Spark scheduling and Tensorflow device placement problems. The kernel is $\mathbf{e}_v \leftarrow g\left[\sum_{u\in\xi(v)} f(\mathbf{e}_u)\right] + \mathbf{e}_v$, where $\mathbf{e}$ is the feature vector on each node, f and g are non-linear transformatio implemented by feed forward networks, $\xi(\cdot)$ denotes the child nodes. When updating the neural network parameters, we use Adam [64] as the optimizer. The non-linear activation function is Leaky-ReLU [226]. We do not observe significant performance change when changing the hyperparameter settings.

Appendix G

Park Environment configuration and comparing baselines

This section details the experiment setup for benchmarking existing RL algorithms in Park. We show the result of the benchmarks in Figure 6-3.

Adaptive video streaming. We train and test the A2C agent on the simulated version of the video streaming environment since the interaction with real environment is slow. However, the learned policy can generalize to a real video environment if the underlying network conditions are similar [204]. We compare the learned A2C policy against two standard schemes. The "buffer-based" heuristic switches the bitrates purely based on the current playback buffer size [146]. "robustMPC" uses a model predictive control framework to decide the bitrate based on a combination of the current buffer size and a conservative estimate of the future network throughput [329]. We use the default parameters in the baseline algorithm from their original paper [329].

Spark cluster job scheduling. The benchmark experiment is on a cluster of 50 executors with a batch of 20 Spark jobs from the TPC-H dataset [295]. During training in simulation, we sample 20 jobs uniformly at random from all available jobs. We test on a real cluster with the same setup and unseen job combinations. The "fair" scheduler gives each job an equal fair share of the executors and round-robins over tasks from runnable stages to drain all branches concurrently. The "optimal weighted fair" scheduler is carefully-tuned to give each job $T_i^\alpha / \sum_i T_i^\alpha$ of the total executors, where T_i is the total work of each job i and α is a tuning factor. Notice that $\alpha = 0$ reduces to a simple fair scheme and $\alpha = 1$ reduces to a weighted fair

scheme based on job size. We sweep through $\alpha \in \{-2, -1.9, ..., 2\}$ for the optimal factor.

SQL Database query optimization. We train and test a DQN agent on a cost model implemented in the open source query optimization framework, Calcite. This provides an estimate of the number of records that would have to be processed when we choose an edge in the query graph (apply a Join), and how long it would take to process them based on the hardware characteristics of the system. The cost model is based on the non-linear cost model ('CM2') described by [175], where the non-linearity models the random access memory constraints of a physical system. The training set, and test set, are generated from 113 queries in the Join Order Benchmark [183], with a 50% train-test split. We use the following baselines from traditional database research to compare against the RL approach. *(1) Exhaustive Search:* For a given cost model, we can find the optimal policy using a dynamic programming algorithm (Exhaustive Search) and all our results are presented relative to this (-1.00 means the plan was as good as Exhaustive Search plan). *(2) Left Deep Search:* Is a popular baseline in practice since it finds the the optimal plan in a smaller search space (only considering join plans that form a left deep tree [175]) making it computationally much faster than Exhaustive Search.

Network congestion control. We train and test the A2C agent in the centralized control setting (a single TCP connection) on a simple single-hop topology. We used a 48Mbps fixed-bandwidth bottleneck link with 50ms round-trip latency and a drop-tail buffer of 400 packets (2 bandwidth-delay products of maximum size packets) in each direction. For comparison, we run TCP Vegas [52]. Vegas attempts to maintain a small number of packets (by default, around 3) in the bottleneck queue, which results in an optimal outcome (minimal delay and packet loss, maximal throughput) for a single-hop topology without any competing traffic. "Confined search space" means we confine the action space of A2C agent to be only within 0.2 and $2\times$ of the average action output from Vegas.

Network active queue management. We train and test the agent on a 10Mbps fixed-bandwidth bottleneck link with 100ms round-trip latency where there are 5 competing TCP flows. The agent examines the state and takes an action every 50ms. We configure the reward to be the current distance from the target queuing delay (20ms). As a comparison, we run "PIE" [142], a classic PID control scheme, with the same target queuing delay.

Tensorflow device placement. We consider device placement optimization for a neural machine translation (NMT) model [30] over two devices (GPUs). This is a popular language

translation model that has an LSTM-based encoder-decoder and attention architecture to translate a source sequence to a target sequence. The training is done over a reliable simulator [5] to quickly obtain run-time estimates given a placement configuration. In the "Single GPU" heuristic, all ops are co-located on the same device, which is optimal for models that can fit in a single device and which do not have significant parallelism in their structure. Scotch [245] is a graph partitioning based heuristic that takes as input both the computational cost of each node and the communication cost along each edge. It then outputs a placement that minimizes total communication cost, while load balancing computation across the devices to within a specified tolerance. The human expert places each LSTM layer on a different device as recommended by Wu et al. [30]. PG-LSTM [218] embeds the graph model as a sequence of node features, and uses an LSTM to output the corresponding placement for each node in the sequence. The PG-GCN [5] on the other hand, uses a graph neural network [54, 131] for embedding the model, and represents the policy as performing iterative placement improvements rather than outputting a placement for all the nodes in one shot.

Circuits Design. The benchmark trains and tests on a fixed three-stage transimpedance amplifier analog circuit. "BO" is a simple Bayesian optimization approach to tune the model parameter. "MACE" is a prior work based on acquisition function ensemble [196]. "ES" stands for evolutional strategy approach [261]. "NG-RL" is the short of non-grach Reinforcement Learning in which we do not involve graph informantion in the optimzation loop. "GCN-RL" is the Reinforcement Learning with graph convolutional neural networks. From the results, we can observe that "GCN-RL" could consistently achieve higher Figure of Merits (FoM) value than other methods. Comparing to "NG-RL", "GCN-RL" has higher FoM value and also faster convergence speed, which indicates the critical role of the graph information.

CDN memory caching. We train and test A2C on several synthetic traces (10000 requests long) produced by an open-source trace generator [43]. We consider a small cache size of 1024KB for the experiment. The LRU heuristic always admits requests, with stale objects evicted based on the last recently used (LRU) policy. Offline optimal uses dynamic programming to compute the best sequence of actions, with the knowledge of future object arrivals.

Multi-dim database indexing. We train and test on a real in-memory column-store, using a dataset from Open Street Maps [236], comprised of 105 million points, each with 6 attributes. The dataset is unchanged across all steps. The query workload shifts continuously between different query distributions, completing a full shift to a new distribution every 20 steps. At each step, the agent observes the previous workload and produces a parametrization of the

grid index that is tested on the next workload. We use a batch size of 1, and the environment is terminal at every state (i.e., the discount factor γ is 0).

We heavily restrict the state and action space to make this environment tractable. The agent does *not* observe the underlying data, since the dataset does not change; it observes only the query workload. Each workload consists of 10 queries, each with two 6-dimensional points to specify the query rectangle, producing a 120-dimensional observation space. Each query coordinate is scaled to $[0,1]$, relative to the range of the corresponding attribute in the OSM dataset. If an attribute is not present in the range filter, the query coordinates for that dimension are 0 and 1. For the agent's action, we fix an ordering of dimensions that we have found to work well empirically; the agent is responsible solely for determining the number of columns along each dimension in the grid, which is a 4-dimensional action space. The baseline is a fixed layout that is run on the same workloads as the agent, tuned roughly by hand to produce low running times on the *entire* sequence of workloads. The baseline layout uses the same dimension ordering that was fixed for the agent and is not re-optimized for each new workload.

Account region assignment. The setup for this experiment follows the first variant of the assignment task outlined in Appendix E, in which the agent has to assign newly created accounts to one of eight regions. Local heuristic is a simple baseline that assigns an account directly to the region it was created in. The Thompson sampling [59] approach uses a random forest model comprising of 100 trees. We train and test DQN over the real trace of one million posts included with Park.

Server load balancing. In this experiment we consider the setup as described in Appendix E, with 10 heterogenous servers. The A2C [219] learning approach is elaborated in Appendix F; 'grad clip' refers to gradient clipping, in which we normalize the policy gradient by its l2 norm when the l2 norm is over 10. The greedy heuristic assigns each incoming job to that queue having the lowest queue size to processing rate ratio.

Switch scheduling. We consider scheduling in a crossbar switch (Appendix E) with 3 input ports and 3 output ports. Time is discretized for simplicity. Traffic between each port pair (i,j) is generated according to a Bernoulli process, with rate given by the (i,j)-th entry of a random bistochastic traffic matrix. The load of the system (i.e., the row and column sums of the traffic matrix) is set to 90%. MWM, or Max-Weight-Matching [272], is a well-known scheduling policy that forwards packets at each time-step according to the maximum weighted matching on the bipartite graph between the set of input and output ports. The

weight of each edge (i,j) on the bipartite graph is set equal to the size of the virtual-output queue (VOQ) j at input port i [272]. For a parameter $\alpha > 0$, MWM-α refers to an analogous policy where the weight of edge (i,j) on the bipartite graph is set equal to the size of VOQ j at input port i raised to the power α.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 265–283, November 2016.

[2] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems*, page 1, 2007.

[3] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 22–31, 2017.

[4] Joshua Achiam, Ethan Knight, and Pieter Abbeel. Towards characterizing divergence in deep q-learning. *arXiv preprint arXiv:1903.08894*, 2019.

[5] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Learning generalizable device placement algorithms for distributed machine learning. In *Advances in Neural Information Processing Systems*, pages 3981–3991, 2019.

[6] Vijay Kumar Adhikari, Sourabh Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting youtube: An active measurement study. In *2012 Proceedings IEEE INFOCOM*, pages 2521–2525. IEEE, 2012.

[7] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 281–294, 2012.

[8] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallel dag jobs online to minimize average flow time. In *Proceedings of the 27th annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 176–189. Society for Industrial and Applied Mathematics, 2016.

[9] Akamai. dash.js. https://github.com/Dash-Industry-Forum/dash.js/, 2016.

[10] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys. ACM, 2011.

[11] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58. ACM, 2018.

[12] Alibaba. Cluster data collected from production clusters in alibaba for cluster management research. https://github.com/alibaba/clusterdata, 2017.

[13] Mansoor Alicherry and TV Lakshman. Network aware resource allocation in distributed clouds. In *2012 Proceedings IEEE INFOCOM*, pages 963–971. IEEE, 2012.

[14] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[15] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[16] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[17] 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 *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, April 2012.

[18] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[19] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. *RFC 5681*, 2009.

[20] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[21] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, New Orleans, Louisiana, USA, February 2018.

[22] Dario Amodei and Danny Hernandez. Ai and compute. https://openai.com/blog/ai-and-compute/, 2018.

[23] Apache Hadoop. Hadoop fair scheduler. Accessed 13/03/2014.

[24] Apache Spark. Spark: Dynamic resource allocation. Spark v2.2.1 Documentation.

[25] Venkat Arun and Hari Balakrishnan. Copa: Congestion Control Combining Objective Optimization with Window Adjustments. In *NSDI*, 2018.

[26] Karl Johan Astrom, Tore Hägglund, and Karl J Astrom. *Advanced PID control*, volume 461. ISA-The Instrumentation, Systems, and Automation Society Research Triangle âĂę, 2006.

[27] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. In *International conference on machine learning*, pages 284–293, 2018.

[28] Sanjeewa Athuraliya, Victor H Li, Steven H Low, and Qinghe Yin. Rem: Active queue management. In *Teletraffic Science and Engineering*, volume 4, pages 817–828. Elsevier, 2001.

[29] Sanjeewa Athuraliya, Steven H Low, Victor H Li, and Qinghe Yin. Rem: Active queue management. *IEEE network*, 15(3):48–53, 2001.

[30] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[31] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking*, 2017.

[32] Luiz André Barroso, Jimmy Clidaras, and Urs HÃűlzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[33] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

[34] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in neural information processing systems*, pages 2494–2504, 2018.

[35] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[36] Lujo Bauer, Scott Garriss, and Michael K Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):2, 2011.

[37] J. Baxter and P. L. Bartlett. Infinite-Horizon Policy-Gradient Estimation. *Journal of Artificial Intelligence Research*, 15:319–350, November 2001.

[38] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230. ACM, 2018.

[39] Francois Belletti, Daniel Haziza, Gabriel Gomes, and Alexandre M Bayen. Expert level control of ramp metering based on multi-task deep reinforcement learning. *IEEE Transactions on Intelligent Transportation Systems*, 19(4):1198–1207, 2018.

[40] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[41] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual International Conference on Machine Learning (ICML)*, pages 41–48, 2009.

[42] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140. ACM, 2018.

[43] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 483–498, 2017.

[44] Felix Berkenkamp, Matteo Turchetta, Angela P. Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *NIPS*, 2017.

[45] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[46] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In

Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC), pages 4:1–4:15, October 2013.

[47] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[48] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[49] Justin A Boyan and Michael L Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, 1994.

[50] Mehran Bozorgi, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–114. ACM, 2010.

[51] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 24–35, New York, NY, USA, 1994. ACM.

[52] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[53] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. https://gym.openai.com/docs/, 2016.

[54] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[55] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[56] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[57] Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. Learned garbage collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 38–44, 2020.

[58] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, June 2010.

[59] Olivier Chapelle and Lihong Li. An empirical evaluation of Thompson sampling. In *Advances in Neural Information Processing Systems*, NIPS'11, 2011.

[60] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with ε resource augmentation. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 363–372, 2004.

[61] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 191–205. ACM, 2018.

[62] Dilip Chhajed and Timothy J Lowe. *Building intuition: insights from basic operations management models and principles*, volume 115. Springer Science & Business Media, 2008.

[63] Federico Chiariotti, Stefano D'Aronco, Laura Toni, and Pascal Frossard. Online learning adaptation strategy for dash clients. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 8. ACM, 2016.

[64] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[65] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 393–406. ACM, 2015.

[66] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.

[67] Cisco. Cisco visual networking index: Forecast and methodology, 2015-2020. 2016.

[68] Maxim Claeys, Steven Latre, Jeroen Famaey, and Filip De Turck. Design and evaluation of a self-learning http adaptive video streaming client. *IEEE communications letters*, 18(4):716–719, 2014.

[69] Maxim Claeys, Steven Latré, Jeroen Famaey, Tingyao Wu, Werner Van Leekwijck, and Filip De Turck. Design and optimisation of a (fa) q-learning-based http adaptive streaming client. *Connection Science*, 2014.

[70] Maxim Claeys, Steven Latré, Jeroen Famaey, Tingyao Wu, Werner Van Leekwijck, and Filip De Turck. Design of a q-learning-based client quality selection algorithm for http adaptive video streaming. In *Adaptive and Learning Agents Workshop*, 2013.

[71] Ignasi Clavera, Anusha Nagabandi, Ronald S Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Learning to adapt: Meta-learning for model-based control. *arXiv preprint arXiv:1803.11347*, 2018.

[72] Ignasi Clavera, Jonas Rothfuss, John Schulman, Yasuhiro Fujita, Tamim Asfour, and Pieter Abbeel. Model-based reinforcement learning via meta-policy optimization. *arXiv preprint arXiv:1809.05214*, 2018.

[73] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *NIPS*, 2017.

[74] DJ Daley. Certain optimality properties of the first-come first-served discipline for g/g/s queues. *Stochastic Processes and their Applications*, 25:301–308, 1987.

[75] DASH Industry Form. Reference Client 2.4.0. http://mediapm.edgesuite.net/dash/public/nightly/samples/dash-if-reference-player/index.html, 2016.

[76] Luca De Alfaro. *Formal verification of probabilistic systems*. Number 1601. Citeseer, 1997.

[77] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *CoRR*, 2016.

[78] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[79] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[80] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[81] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[82] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.

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

[84] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the Impact of Video Quality on User Engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM. ACM, 2011.

[85] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 431–442. ACM, 2014.

[86] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI*, pages 395–408, Oakland, CA, May 2015. USENIX Association.

[87] Mo Dong, Tong Meng, D Zarchy, E Arslan, Y Gilad, B Godfrey, and M Schapira. PCC Vivace: Online-Learning Congestion Control. In *NSDI*, 2018.

[88] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[89] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[90] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[91] Loren C Eiseley. *The firmament of time*. U of Nebraska Press, 1960.

[92] Eleazar Eskin, Wenke Lee, and Salvatore J Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 1, pages 165–175. IEEE, 2001.

[93] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1406–1415, 2018.

[94] Richard Evans and Jim Gao. DeepMind AI Reduces Google Data Centre Cooling Bill by 40%. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/, 2016.

[95] Michael Fairbank and Eduardo Alonso. The divergence of reinforcement learning algorithms with value-iteration and function approximation. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2012.

[96] G Fairhurst, A Sathiaseelan, and R Secchi. Updating tcp to support rate-limited traffic. *RFC 7661*, 2015.

[97] Federal Communications Commission. Raw Data - Measuring Broadband America 2016. https://www.fcc.gov/reports-research/reports/measuring-broadband-america, 2016.

[98] Hanhua Feng, Vishal Misra, and Dan Rubenstein. Optimal state-free, size-aware dispatching for heterogeneous m/g/-type systems. *Performance evaluation*, 62(1-4):475–492, 2005.

[99] W. Feng, K. Shin, D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. on Networking*, August 2002.

[100] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012.

[101] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135, 2017.

[102] Sally Floyd. Highspeed tcp for large congestion windows. *RFC 3649*, 2003.

[103] Sally Floyd, Ramakrishna Gummadi, and Scott Shenker. Adaptive red: An algorithm for increasing the robustness of red's active queue management, 2001.

[104] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)*, 1(4):397–413, 1993.

[105] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. Tcp extensions for multipath operation with multiple addresses. Technical report, 2013.

[106] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

[107] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[108] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1582–1591, 2018.

[109] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1662–1670, 2018.

[110] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[111] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*, 2018.

[112] Peter Geibel. Reinforcement learning for mdps with constraints. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, pages 646–653, 2006.

[113] Xinyang Geng, Marvin Zhang, Jonathan Bruce, Ken Caluwaerts, Massimo Vespignani, Vytas SunSpiral, Pieter Abbeel, and Sergey Levine. Deep reinforcement learning for tensegrity robot locomotion. *arXiv preprint arXiv:1609.09049*, 2016.

[114] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[115] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[116] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 365–378, April 2013.

[117] Paolo Giaccone, Balaji Prabhakar, and Devavrat Shah. Towards simple, high-performance schedulers for high-aggregate bandwidth switches. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1160–1169. IEEE, 2002.

[118] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[119] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–115, November 2016.

[120] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, Santa Clara, CA, February 2020. USENIX Association.

[121] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource Packing for Cluster Schedulers. In *Proceedings of SIGCOMM*, 2014.

[122] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 65–80, 2016.

[123] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of OSDI*, pages 81–97. USENIX Association, 2016.

[124] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[125] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004.

[126] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI*, pages 1–14, 2015.

[127] Shixiang Gu, Tim Lillicrap, Richard E Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3849–3858, 2017.

[128] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *Proceedings of the 2017 International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 66–83, 2017.

[129] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[130] Martin T Hagan, Howard B Demuth, Mark H Beale, and Orlando De Jesús. *Neural network design*. PWS publishing company Boston, 1996.

[131] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[132] M Handley, J Padhye, and S Floyd. Tcp congestion window validation. *RFC 2861*, 2000.

[133] Mor Harchol-Balter and Rein Vesilo. To balance or unbalance load in size-interval task allocation. *Probability in the Engineering and Informational Sciences*, 24(2):219–244, April 2010.

[134] James Harrison, Animesh Garg, Boris Ivanovic, Yuke Zhu, Silvio Savarese, Li Fei-Fei, and Marco Pavone. Adapt: zero-shot adaptive policy transfer for stochastic dynamical systems. *arXiv preprint arXiv:1707.04674*, 2017.

[135] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM 2014-IEEE conference on computer communications*, pages 460–468. IEEE, 2014.

[136] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, (1), 1970.

[137] Douglas Heaven. Why deep-learning ais are so easy to fool. *Nature*, 574(7777):163–166, 2019.

[138] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[139] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. NSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[140] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep Q-learning from Demonstrations. In *Thirty-Second AAAI Conference on Artifical Intelligence*, AAAI '18, New Orleans, April 2017. IEEE.

[141] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[142] Chris V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. On designing improved controllers for aqm routers supporting tcp flows. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1726–1734. IEEE, 2001.

[143] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[144] Wenjie Hu, Yihua Liao, and V Rao Vemuri. Robust anomaly detection using support vector machines. In *Proceedings of the international conference on machine learning*, pages 282–289, 2003.

[145] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC. ACM, 2012.

[146] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM. ACM, 2014.

[147] David K Hunter, David Cotter, R Badlishah Ahmad, W David Cornwell, Tim H Gilfedder, Peter J Legg, and Ivan Andonovic. 2/spl times/2 buffered switch fabrics for traffic routing, merging, and shaping in photonic cell networks. *Journal of lightwave technology*, 15(1):86–101, 1997.

[148] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[149] IBM. The Spatial Index. https://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.spatial.doc/ids_spat_024.htm.

[150] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[151] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *ACM SIGOPS*, 2009.

[152] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[153] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[154] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[155] Nathan Jay, Noga H Rotman, P Godfrey, Michael Schapira, and Aviv Tamar. Internet congestion control via deep reinforcement learning. *arXiv preprint arXiv:1810.03259*, 2018.

[156] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. CFA: A Practical Prediction System for Video QoE Optimization. In *NSDI*, NSDI. USENIX Association, 2016.

[157] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2012.

[158] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 393–406, Boston, MA, 2017. USENIX Association.

[159] Gong Jie, Kuang Xiao-Hui, and Liu Qiang. Survey on software vulnerability analysis method based on machine learning. In *Data Science in Cyberspace (DSC), IEEE International Conference on*, pages 642–647. IEEE, 2016.

[160] Cheng Jin, D.X. Wei, and S.H. Low. Fast tcp: motivation, architecture, algorithms, performance. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2490 – 2501 vol.4, march 2004.

[161] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[162] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. Model assertions for debugging machine learning. In *NeurIPS MLSys Workshop*, 2018.

[163] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference*, pages 485–497, 2015.

[164] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[165] James E. Kelley Jr and Morgan R. Walker. Critical-path planning and scheduling. In *Proceedings of the Eastern Joint IRE-AIEE-ACM Computer Conference (EJCC)*, pages 160–173, 1959.

[166] Frank P Kelly. *Reversibility and stochastic networks*. Cambridge University Press, 2011.

[167] Tom Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.

[168] István Ketykó, Katrien De Moor, Toon De Pessemier, Adrián Juan Verdejo, Kris Vanhecke, Wout Joseph, Luc Martens, and Lieven De Marez. QoE Measurement of Mobile YouTube Video Streaming. In *Proceedings of the 3rd Workshop on Mobile Video Delivery*, MoViD. ACM, 2010.

[169] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2015.

[170] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[171] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[172] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[173] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. 2019.

[174] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC. ACM, 2012.

[175] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[176] Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. 2010.

[177] Prasad A Kulkarni. Jit compilation policy for modern machines. In *ACM SIGPLAN Notices*, volume 46, pages 773–788. ACM, 2011.

[178] S. Kunniyur and R. Srikanth. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In *SIGCOMM*, 2001.

[179] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in neural information processing systems*, pages 6402–6413, 2017.

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

[181] Stefan Lederer, Christopher Müller, and Christian Timmerer. Dynamic adaptive streaming over http dataset. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 89–94. ACM, 2012.

[182] Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *Proceedings of the 29th annual Symposium on Foundations of Computer Science (FOCS)*, pages 256–269, 1988.

[183] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[184] Benjamin Letham and Eytan Bakshy. Bayesian optimization for policy search via online-offline experimentation. *arXiv preprint arXiv:1904.01049*, 2019.

[185] Benjamin Letham, Brian Karrer, Guilherme Ottoni, Eytan Bakshy, et al. Constrained bayesian optimization with noisy experiments. *Bayesian Analysis*, 2018.

[186] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[187] Yuxi Li. Reinforcement learning applications. https://medium.com/@yuxili/rl-applications-73ef685c07eb, 2019.

[188] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, April 2014.

[189] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*, pages 539–548, 2018.

[190] Eric Liang and Richard Liaw. Scaling multi-agent reinforcement learning. https://bair.berkeley.edu/blog/2018/12/12/rllib/, 2018.

[191] Timothy Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

[192] Bo Liu, Francisco V Fernández, Qingfu Zhang, Murat Pak, Suha Sipahi, and Georges Gielen. An enhanced moea/d-de and its application to multiobjective analog cell sizing. In *CEC*, 2010.

[193] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *Proceedings of the 2017 IEEE International Conference on Big Data (BigData)*, pages 2884–2892. IEEE, 2017.

[194] Tyler Lu, Dávid Pál, and Martin Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492, 2010.

[195] Tyler Lu, Dale Schuurmans, and Craig Boutilier. Non-delusional q-learning and value-iteration. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9949–9959. 2018.

[196] Wenlong Lyu, Fan Yang, Changhao Yan, Dian Zhou, and Xuan Zeng. Batch bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design. In *International Conference on Machine Learning*, pages 3312–3320, 2018.

[197] Wenlong Lyu, Fan Yang, Changhao Yan, Dian Zhou, and Xuan Zeng. Multi-objective bayesian optimization for analog/rf circuit synthesis. In *DAC*, 2018.

[198] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI*, pages 589–603, 2015.

[199] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 144–159, New York, NY, USA, 2016. ACM.

[200] Siva Theja Maguluri and R Srikant. Heavy traffic queue length behavior in a switch under the maxweight algorithm. *Stochastic Systems*, 6(1):211–250, 2016.

[201] 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)*, Atlanta, GA, November 2016.

[202] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. In *Proceedings of the 2019 Reinforcement Learning for Real Life Workshop*, 2019.

[203] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. In *Advances in Neural Information Processing Systems*, pages 2494–2506, 2019.

[204] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the ACM SIGCOMM 2017 Conference*. ACM, 2017.

[205] Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. Towards safe online reinforcement learning in computer systems. In *NeurIPS Machine Learning for Systems Workshop*, 2019.

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

[207] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

[208] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[209] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814*, 2020.

[210] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, 2019.

[211] Monaldo Mastrolilli and Ola Svensson. (acyclic) job shops are hard to approximate. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 583–592. IEEE, 2008.

[212] A Stephen McGough, Noura Al Moubayed, and Matthew Forshaw. Using machine learning in trace-driven energy-aware simulations of high-throughput computing systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 55–60. ACM, 2017.

[213] Nick McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking*, (2):188–201, 1999.

[214] Ishai Menache, Shie Mannor, and Nahum Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, (1), 2005.

[215] Zili Meng, Minhu Wang, Mingwei Xu, Hongzi Mao, Jiasong Bai, and Hongxin Hu. Explaining deep learning-based networked systems. *arXiv preprint arXiv:1910.03835*, 2019.

[216] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[217] 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.

[218] 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 *Proceedings of The 33rd International Conference on Machine Learning*, 2017.

[219] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pages 1928–1937, 2016.

[220] 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, Demis Hassabis Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[221] Ricky KP Mok, Edmond WW Chan, and Rocky KC Chang. Measuring the quality of experience of http video streaming. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 485–492. IEEE, 2011.

[222] Ricky K.P. Mok, Edmond W.W. Chan, Xiapu Luo, and Rocky K.C. Chang. Inferring the QoE of HTTP Video Streaming from User-viewing Activities. In *Proceedings of the First ACM SIGCOMM Workshop on Measurements Up the Stack*, W-MUST. ACM, 2011.

[223] Srinivas Mukkamala, Guadalupe Janoski, and Andrew Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707. IEEE, 2002.

[224] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 188–201, 2016.

[225] John Nagle. Rfc-896: Congestion control in ip/tcp internetworks. *Request For Comments*, 1984.

[226] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[227] Akshay Narayan, Frank J. Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *SIGCOMM*, 2018.

[228] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. Cost-guided cardinality estimation: Focus where it matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 154–157. IEEE, 2020.

[229] Limelight Netowrks. The state of online video. https://www.limelight.com/resources/white-paper/state-of-online-video-2019/, 2019.

[230] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*, 2015.

[231] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[232] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multi-agent systems: A review of challenges, solutions and applications. *arXiv preprint arXiv:1812.11794*, 2018.

[233] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[234] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[235] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, RafaÅĆ JÃşzefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, 2018.

[236] OpenStreetMap contributors. US Northeast dump obtained from https://download.geofabrik.de/. https://www.openstreetmap.org, 2019.

[237] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint arXiv:1803.08604*, 2018.

[238] Teunis J Ott, TV Lakshman, and Larry H Wong. Sred: stabilized red. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1346–1355. IEEE, 1999.

[239] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–307, Oakland, California, USA, May 2015.

[240] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[241] Noam Palatin, Arie Leizarowitz, Assaf Schuster, and Ran Wolff. Mining for misconfigured machines in grid systems. *Data Mining Techniques in Grid Computing Environments*, page 71, 2008.

[242] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. A stateless active queue management scheme for approximating fair bandwidth allocation. In *Proceedings of Infocom 200*. Citeseer, 2000.

[243] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51(12):3448–3470, 2007.

[244] Martin Pecka and Tomas Svoboda. Safe exploration techniques for reinforcement learning–an overview. In *International Workshop on Modelling and Simulation for Autonomous Systems*, pages 357–375. Springer, 2014.

[245] François Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *European Conference on Parallel Processing*, pages 195–204. Springer, 2007.

[246] Kandaraj Piamrat, Cesar Viho, Jean-Marie Bonnin, and Adlen Ksentini. Quality of Experience Measurements for Video Streaming over Wireless Networks. In *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, ITNG. IEEE Computer Society, 2009.

[247] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *International Conference on Machine Learning*, pages 2817–2826, 2017.

[248] PostgreSQL. postgres. https://www.postgresql.org/, 2019.

[249] W. B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2007.

[250] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 2015.

[251] Jonathan Raiman, Susan Zhang, and Christy Dennison. Neural network surgery with sets. *arXiv preprint arXiv:1912.06719*, 2019.

[252] Chandrasekharan Rajendran. A no-wait flowshop scheduling heuristic to minimize makespan. *Journal of the Operational Research Society*, 45(4):472–478, 1994.

[253] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017.

[254] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 303–313, New York, NY, USA, 1988. ACM.

[255] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 36:1–36:15, 2016.

[256] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

[257] Behzad Razavi. *Design of Analog CMOS Integrated Circuits*. Tata McGraw-Hill Education, 2002.

[258] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[259] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute Path Bandwidth Traces from 3G Networks: Analysis and Applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, MMSys. ACM, 2013.

[260] Joanne K Rowling. Harry potter and the goblet of fire. *London: Bloomsbury*, 2000.

[261] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[262] Sandvine. Global internet phenomena-latin american & north america. 2015.

[263] I Sandvine. Global internet phenomena report. *North America and Latin America*, 2018.

[264] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850, 2016.

[265] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[266] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

[267] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

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

[269] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

[271] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[272] Devavrat Shah and Damon Wischik. Optimal scheduling algorithms for input-queued switches. In *Proceedings of the 25th IEEE International Conference on Computer Communications: INFOCOM 2006*, pages 1–11. IEEE Computer Society, 2006.

[273] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[274] Michael Shapira and Keith Winstein. Congestion-Control Throwdown. In *HotNets*, 2017.

[275] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[276] David B Shmoys, Clifford Stein, and Joel Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, 1994.

[277] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershevlvam, 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*, 2016.

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

[279] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[280] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 479–490. ACM, 2014.

[281] Iraj Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[282] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. BOLA: near-optimal bitrate adaptation for online videos. In *Infocom*, 2016.

[283] Raj Srinivasan. Rpc: Remote procedure call protocol specification version 2. Technical report, 1995.

[284] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77–90. ACM, 2003.

[285] Richard. S. Sutton and Andrew. G. Barto. *Reinforcement Learning: An Introduction, Second Edition*. MIT Press, 2017.

[286] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[287] Synopsys. Hspice. https://www.synopsys.com/verification/ams-verification/hspice.html, 2019.

[288] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *INFOCOM*, 2006.

[289] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[290] Apache Tez. https://tez.apache.org/.

[291] TFLearn. TFLearn: Deep learning library featuring a higher-level API for TensorFlow. http://tflearn.org/, 2017.

[292] Philip Thomas. Bias in natural actor-critic algorithms. In *International Conference on Machine Learning*, pages 441–448, 2014.

[293] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems*, pages 2659–2669, 2017.

[294] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[295] The TPC-H Benchmarks. www.tpc.org/tpch/.

[296] George Tucker, Surya Bhupatiraju, Shixiang Gu, Richard E Turner, Zoubin Ghahramani, and Sergey Levine. The mirage of action-dependent baselines in reinforcement learning. *arXiv preprint arXiv:1802.10031*, 2018.

[297] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 35. ACM, 2016.

[298] Alfonso Valdes and Keith Skinner. Adaptive, model-based monitoring for cyber attack detection. In *Recent Advances in Intrusion Detection*, pages 80–93. Springer, 2000.

[299] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[300] Jeroen van der Hooft, Stefano Petrangeli, Maxim Claeys, Jeroen Famaey, and Filip De Turck. A learning-based algorithm for improved bandwidth-awareness of adaptive streaming clients. In *2015 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE.

[301] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[302] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[303] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[304] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 48–56, September 2014.

[305] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[306] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

[307] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

[308] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II, 2019.

[309] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[310] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: Wan-aware optimization for analytics queries. In *OSDI*, 2016.

[311] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, pages 323–336, 2015.

[312] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. Transferable automatic transistor sizing with graph neural networks and reinforcement learning. 2019.

[313] Hanrui Wang, Jiacheng Yang, Hae-Seung Lee, and Song Han. Learning to design circuits. *arXiv preprint arXiv:1812.02734*, 2019.

[314] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning caching policies with subsampling. In *NeurIPS Machine Learning for Systems Workshop*, 2019.

[315] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R Ganger. Storage device performance prediction with cart models. In *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.*, pages 588–595. IEEE, 2004.

[316] Lex Weaver and Nigel Tao. The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 538–545. Morgan Kaufmann Publishers Inc., 2001.

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

[318] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[319] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[320] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, pages 459–471, Lombard, IL, 2013. USENIX.

[321] Cathy Wu, Aboudy Kreidieh, Kanaad Parvate, Eugene Vinitsky, and Alexandre M Bayen. Flow: Architecture and benchmarking for reinforcement learning in traffic control. *arXiv preprint arXiv:1710.05465*, 2017.

[322] Cathy Wu, Aravind Rajeswaran, Yan Duan, Vikash Kumar, Alexandre M Bayen, Sham Kakade, Igor Mordatch, and Pieter Abbeel. Variance reduction for policy gradient with action-dependent factorized baselines. In *International Conference on Learning Representations*, 2018.

[323] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *Submitted to International Conference on Learning Representations*, 2017.

[324] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. *IEEE/ACM Transactions on Networking*, 16(6):1281–1294, 2008.

[325] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 495–511, 2020.

[326] Francis Y Yan, Jestin Ma, Greg Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: The Training Ground for Internet Congestion-Control research. In *USENIX ATC*, 2018.

[327] Dit-Yan Yeung and Yuxin Ding. Host-based intrusion detection using dynamic and static behavioral models. *Pattern recognition*, 36(1):229–243, 2003.

[328] Sun Yi, Yin Xiaoqi, Jiang Junchen, Sekar Vyas, Lin Fuyuan, Wang Nanshu, Liu Tao, and Bruno Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. SIGCOMM, New York, NY, USA, 2016. ACM.

[329] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM. ACM, 2015.

[330] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. Via: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM*, SIGCOMM '16, 2016.

[331] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 28–28. USENIX Association, 2011.

[332] Zack Slayton. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB, 2017.

[333] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[334] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 509–522. ACM, 2015.

[335] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *SIGCOMM*, 2015.

[336] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pages 377–392, 2017.

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

[338] Zheng Zhang, Jun Li, CN Manikopoulos, Jay Jorgenson, and Jose Ucles. Hide: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification. In *Proc. IEEE Workshop on Information Assurance and Security*, pages 85–90, 2001.

[339] Xuan Kelvin Zou, Jeffrey Erman, Vijay Gopalakrishnan, Emir Halepovic, Rittwik Jana, Xin Jin, Jennifer Rexford, and Rakesh K. Sinha. Can accurate predictions improve video streaming in cellular networks? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile. ACM, 2015.

[340] George Zyskind and Frank B Martin. On best linear estimation and general gauss-markov theorem in linear models with arbitrary nonnegative covariance structure. *SIAM Journal on Applied Mathematics*, 17(6):1190–1202, 1969.