Data-Driven Datacenter Traffic Control

by

Fabian Ruffy Varga

B.Sc. Computer Science, Ludwig Maximilian University, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Data-Driven Datacenter Traffic Control

submitted by **Fabian Ruffy Varga** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Ivan Beschastnikh, Computer Science
Supervisor

Additional Supervisory Committee Members:

Margo Seltzer, Computer Science
Supervisory Committee Member

Abstract

Recent networking research has identified that data-driven congestion control (CC) can be more efficient than traditional CC in TCP. Deep reinforcement learning (RL), in particular, has the potential to learn optimal network policies. However, RL suffers from instability and over-fitting, deficiencies which so far render it unacceptable for use in data center networks. In this paper, we analyze the requirements for data-driven policies to succeed in the data center context. And, we present a new emulator, Iroko, which supports different network topologies, data center traffic engineering algorithms, and deployment scenarios. Iroko interfaces with the OpenAI gym toolkit, which enables fast and fair evaluation of RL as well as traditional algorithms under equal conditions. We present initial benchmarks of three deep RL algorithms against TCP New Vegas and DCTCP. The results show that these algorithms are able to learn a CC policy with comparative performance to TCP on a dumbbell and fat-tree topology. We make our emulator open-source and publicly available: https://github.com/dcgym/iroko.

Lay Summary

Data centers (DCs) are essential nowadays as they provide easy access to massive computing and storage. However, managing a DC is not an easy task for a company. Servers can send too much data at once, which leads to congestion in an overloaded network. Operators usually use manually configured network algorithms to avoid network congestion. However, manual tuning typically does not draw out the full potential out of these algorithms.

Fortunately, DCs offer plenty data that can be analyzed and used to optimize this process. For example, machine learning (ML), which draws insight and learns from collected data, can help improve algorithms by finding a better configuration than the operator.

We present a new testing platform intended to investigate the potential of ML in solving datacenter problems. Our preliminary results show that algorithms who know nothing about the network and learn from scratch already achieve comparative performance to traditional techniques.

Preface

At University of British Columbia (UBC), a preface may be required. Be sure to check the Graduate and Postdoctoral Studies (GPS) guidelines as they may have specific content to be included.

Table of Contents

List of Tables

List of Figures

Glossary

GPS	Graduate and Postdoctoral Studies
TE	traffic engineering
RTT	round-trip time
DC	datacenter
CC	congestion control
TCP	Transport Control Protocol
UDP	User Datagram Protocol
ML	Machine Learning
RL	Reinforcement Learning
DQN	Deep Q-Network
PG	Policy Gradient
DPG	Deterministic Policy Gradient
DDPG	Deep Deterministic Policy Gradient
TRPO	Trust Region Policy Optimization
PPO	Proximal Policy Optimization
KB	Kilobyte

Mb	Megabit
MB	Megabyte
GB	Gigabyte
GbE	Gigabit Ethernet
Mbps	Megabit per second
Gbps	Gigabit per second
DCTCP	Data Center TCP
TCP NV	TCP New Vegas
ToR	Top-of-Rack
OSPF	Open Shortest Path First
VLB	Valiant Load Balancing
ECMP	Equal Cost Multi Path
AIMD	Additive increase/multiplicative decrease
ECN	Explicit Congestion Notification
RED	Random Early Detection
NUM	Network Utility Maximization
PCC	Performance-oriented Congestion Control
MDP	Markov Decision Process
MC	Monte-Carlo
AC	Actor-Critic
SDN	Software-Defined Networking
NIC	Network Interface Card

API	application programming interface
FCT	flow completion time

Acknowledgments

This work is in part supported by the Engineering Research Council of Canada (NSERC) Discovery Grant and the Institute for Computing, Information, and Cognitive Systems (ICICS) at the University of British Columbia, Point Grey Campus.

Firstly, I would like to thank my advisor Ivan Beschastnikh for taking me on as a student and providing me with thorough guidance and ample opportunities in both career and research. My life's trajectory has definitely changed after the two years here at UBC. Secondly, I am also indebted to Mihir Nanavati for all his insight and help on low-level systems topics and general advice on graduate student life. Thirdly, I would also like to thank Margo Seltzer for being my second reader and letting me be a teaching assistant in her operating systems class.

A shout-out also goes to all the members of the Networks, Systems, and Security lab who made my working hour enjoyably unproductive. And to Michael Przystupa for all the help on this project while enduring my reinforcement learning jeremiads. I would also like to include my roommates Siddhesh Khandelwal, Alistair Charles Wick, and Clement Yat Fung, who weathered my peculiar music taste and Draconian cleaning plans. Finally, I want to thank my friends and family back home, who always kept me grounded and away from tech and research.

Chapter 1

Introduction

In the past decade, the optimization of utilization, cost, and latency in datacenters (DCs) has been a focal point of networking research. However, despite the innovation potential of datacenters, congestion control (CC) algorithms and central schedulers have remained fundamentally reactive. Most conventional algorithms are designed to respond to micro-bursts or flow collisions as quickly as possible, but cannot outright identify and preemptively avoid these events. Congestion control relies on human-tuned policies, which, albeit perceived as tried, tested, and reliable, are ultimately limited in their capabilities.

Automated traffic engineering (TE) systems can give administrators the advantage of tight control over the datacenter, mitigating the effects of such queue-buildup and bursts. Unfortunately, contemporary systems rely on human-tuned policies and are not forward-looking when performing operations. Controllers tackle congestion in the network by either requiring synchronous communication (e.g., receivers providing credits to senders), restricting flow admission, or by asynchronously optimizing TE rules *after* suboptimal behavior has been detected. These systems have been designed to operate on static and general policies and often do not capture the complexity and volatility of datacenters. An optimal congestion controller model should adjust its rate proactively on the premonition of flow collision. It should not adhere to signals of explicit congestion (such as increasing round-trip time (RTT) or loss) and instead identify traffic patterns that indicate inbound competing traffic. Ideally, a datacenter network's compute and

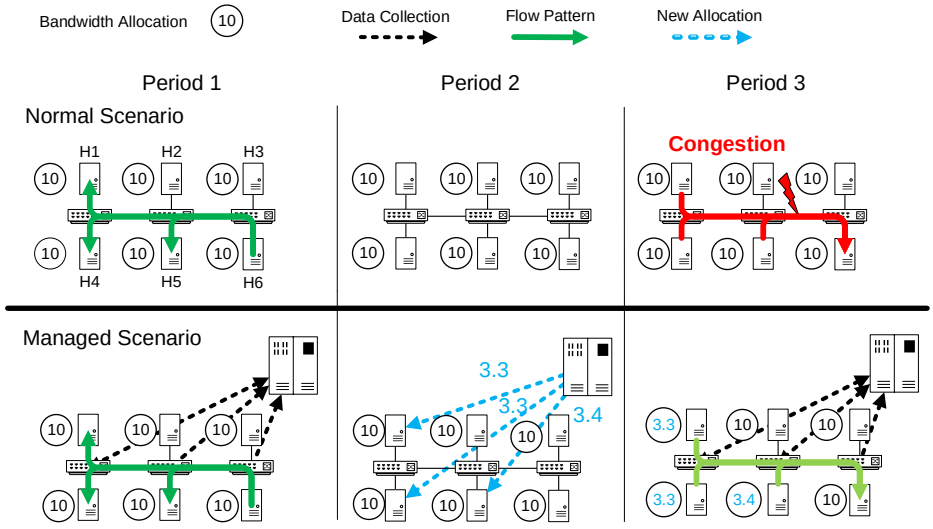

Figure 1.1: A parking lot topology experiencing congestion.

forwarding power should be treated as a single finite resource. A global, centralized arbiter could then act akin to an OS scheduler, which allocates CPU time slices to processes, and fairly distributes flows in the network. By allowing only a safe sending rate for each participant, such a system could minimize the number of bursts and loss.

As example of such a strategy, take the congestion scenario in Figure 1.1. In this scenario, each switch link has a capacity of 10 Gigabit per second (Gbps). `H6` sends data to `H1`, `H4`, and `H5` in Period 1, which will cause these hosts to reply with a large data stream in Period 3. In the normal case, every host is unrestricted in its sending rate, causing congestion at the bottleneck link of the switches. Packets are lost, latency surges and TCP's congestion control has to kick in. In the second scenario, an agent, which is aware of this pattern, observes the flows from `H6` and decides to act. In Period 2, it restricts the sending rate of the responsible hosts or provides a *hint* to the congestion control algorithm, and optimally and fairly allocates bandwidth. Hosts are able to send at a stable and reliable rate without experiencing packet loss and retransmission. This increases the overall goodput[1]

[1]Goodput refers application throughput without the overhead of Transport Control Protocol

of the network.

We claim that to realize a scenario as described in Figure 1.1 a scheduler needs to learn from past network information to continuously optimize its future strategy. Given the fact that recent advancements in network programmability [35, 72] and monitoring [32, 81] have made the collection of statistics on the order of microseconds feasible, devising a policy which is drawn from data and network characteristics instead of human intuition is possible.

Machine Learning (ML), in particular, is a potential strategy that can automate management by inferring optimal policies from past data. While ML is an empirically-driven, black-box approach, it has shown promising results in many network domains. Recent contributions include data-driven flow control for wide-area networks [25], job scheduling [61], and cellular congestion control [109]. Data centers are a promising domain as many datacenter networking challenges can be formulated as an optimization problem [94]. Adhering to the objective of maximizing future rewards [95], Reinforcement Learning (RL), in particular, has the potential to learn anticipatory policies. Researchers have used RL to address a range of datacenter tasks, such as routing [11, 102], power management [96], and traffic optimization [18].

RL is not without flaws. In fact, the field is plagued by shortcomings. RL suffers from a lack of generalizability [55, 63, 79, 105, 111] and reproducibility [37]. These limitations make it an unacceptable choice for a wide array of usage in datacenters, in particular traffic management, and congestion control. Datacenter operators expect stable, scalable, and accurate behavior, a property which RL can provide only after long periods of active training. Despite these limitations, RL is progressing quickly in fields such as locomotion [99], autonomous driving [86], and robotics [53]. These domains exhibit properties similar to datacenter control problems: both deal with a large input space and require continuous output actions. Decisions have to be made rapidly (on the order of microseconds) without compromising safety and reliability. Fix the citations here

What these fields have, and what current datacenter research is missing, is a platform to compare and evaluate techniques. RL benchmark toolkits such as the OpenAI gym [15] or RLgarage (formerly RLlab) [26] foster innovation and

(TCP) retransmissions, packet headers, or acknowledgment packets.

enforce a common standards framework. In the networking space, the Pantheon project [108] represents a step in this direction. It provides a system to compare congestion control solutions for wide area networks. No such framework currently exists for datacenters, partially because topology and traffic patterns are often considered private and proprietary [9]. For data-driven algorithms, such as RL, to be viewed as a viable, deployable tool for networking solutions, having a set of reproducible benchmarks to compare algorithm performance is critical.

Iroko We present *Iroko*, a datacenter emulator that enables understanding of the fundamental requirements and limitations of data-driven algorithms applied to datacenter networks. Iroko offers a way to fairly evaluate centralized and decentralized policies against conventional traffic control solutions by interfacing with the OpenAI Gym [15] and the Mininet [56] network emulation platform. In the Iroko emulator, all algorithms are evaluated on the basis of "reward", which describes how close an algorithm is to achieving a user-defined objective. As a concrete use-case, we apply reinforcement learning as an example of data-driven management. Using the emulator, we introduce reinforcement learning techniques into the networking domain. We show that, in a real-time networking environment, state-of-the-art RL algorithms are able to learn and effectively optimize a pre-defined policy.

In the following chapters, we first outline why a data-driven approach in datacenters will become a necessity in the near-future (Chapter 2). We then provide the background necessary to understand how datacenter traffic engineering and data-driven algorithms complement each other (Chapter 3), and subsequently we showcase the design of the Iroko emulator intended help other researchers realize this vision (Chapter 4). We then measure Iroko's limitations and compare the performance of three existing, advanced RL algorithms with state-of-the-art congestion control in different traffic scenarios (Chapter 5). We conclude with a discussion on the challenges of data-driven networking in datacenters (Chapter 6.1) and provide an outlook on the future of the field (Chapter 6.2).

Chapter 2

The Case for Data-Driven Datacenter Control

2.1 The Limits of Human-Defined Protocols.

The limits of TCP. TCP was designed to optimize traffic globally on two simple principles: do not push more packets than the network can handle and play fair with other participants. Correspondingly, every TCP node responds to packet loss and assumes that others will behave accordingly. Every node is assumed to be an independent actor, which learns of other participants and the network capacity via indirect congestion signals in the network (e.g., packet loss). TCP's algorithm is a best-effort solution to globally maximize utilization under the fairness principle.

This design has not fundamentally changed over the last thirty years. Recent research questions whether TCP can still be considered a viable option [17, 20, 25, 45]. TCP is a fundamentally reactive protocol [45]. It operates on local observations and attempts to converge globally. These two properties have fundamental implications on datacenters. Since TCP operates only on local knowledge and probes the network iteratively, small flows frequently fail to stabilize and never achieve optimal bandwidth [20, 45, 76]. DC network traffic measurements have shown that buffer overruns and small short-lived bursts are the primary factor of loss and retransmits [3, 20, 49, 83]. Suboptimal routing and congestion control

can quickly lead to bufferbloat and the eventual collapse of a high-load network, requiring sophisticated buffer tuning and queue management [70].

A large branch of research centers around optimizing TCP in the datacenter. Flow scheduling attempts to prioritize different application streams, dynamic routing entangles colliding flows and spreads bandwidth across networks. Latency-based congestion avoidance, such Data Center TCP (DCTCP) [3], tries to detect flow conflicts as early as possible to reduce their sending rate just-in-time. However, the substantial and noticeable increase of loss and latency in the network already portends a problem. Overflowing queues in forwarding elements or mismatched hardware capabilities imply that traffic has not been optimally distributed.

To better manage traffic, there exists a class of centralized schedulers which can achieve close-to-optimal bandwidth utilization [2, 10, 42, 44, 76]. However, these schedulers are still *reactive* in nature. The central controller responds to signals in the network or requests by applications, which may cost valuable round-trip latency. Often, short-term flows or bursts are unaccounted for, which, again, causes undesirable packet loss and back-propagating congestion [45]. Ideally, a network should always be "zero-queue", i.e., latency will merely be induced by propagation, not queuing delay.

While many techniques are designed to respond to micro-bursts or flow collisions as quickly as possible, they are not capable of preemptively identifying and avoiding these events [20, 45]. Any time that flows collide, packets are lost and application goodput decreases. This problem is exacerbated by the advent of 200 and 400Gigabit Ethernet (GbE) Network Interface Card (NIC)s. Packets and congestion signals propagate at a much faster rate than the end host networking stack can respond [98]. In future datacenters, where the slowing of Moore's Law [100] makes CPUs a precious commodity, wasting unnecessary cycles on complicated TCP stacks will become unacceptable.

The limits of admission control. To achieve "zero-queue", one recent thread of research advocates for admission-control as an alternative to the conventional burst-and-backoff mechanism of TCP [20, 45, 76]. In a DC, nodes can be restricted in behavior and arbitrarily modified, which permits substantial simplification of enforcement and prioritization policies. Limited admission can provide

operators with the benefit of tight control and predictability, and has been shown to dramatically reduce volatile flow behavior [43, 76].

However, instead of tolerating queuing in the network, admission systems push it onto servers. For example, FastPass [76], which rate limits end hosts and tightly manages the entire datacenter network, operates synchronously in response to flow requests. In ExpressPass [20], servers are only permitted to send after receiving credit packets. Small packet sizes (64 bytes) and non-handshake-based transmissions incur unnecessary credit waste and round-trip latency.

This unnecessary latency is partially incurred because these systems operate conservatively and use manually designed heuristics to assign bandwidth or relax rate-limiting. Neither traditional congestion control nor admission control algorithm generalize particularly well. As discussed in Section 3, datacenters are diverse and exhibit domain-specific idiosyncrasies. A human-designed, general purpose algorithm might perform adequate in many scenarios but is unlikely to ever achieve optimality. This trade-off has generally been accepted, because manually tuning policies and algorithms for a specific workload is tenuous and often plain impossible. We require a mechanism that automates this particular challenge, for example a policy improvement technique such as RL.

2.2 Opportunities in the Datacenter

Although datacenters present several challenges in terms of congestion control design and deployment, they are also boons for innovation. An operator has full control over all components and can freely develop a clean-slate design and collect data.

Policies instead of protocols. Software-Defined Networking (SDN) [30] has led to a paradigm shift in datacenter networking research. Instead of operating on the notion of *distributed* protocols SDN advocates separating the network into a control and data plane. The data plane is a parameterizable set of primitives that operates at high performance. The control plane "manages" these primitives by adjusting the data-plane control interface.

SDN gives operators the ability to freely control and adapt their network ar-

chitecture, leading to highly customized systems and fine-grained optimization. One such example is centralized and automated network management. A single controller, with global knowledge, is able to automatically modify and adapt the forwarding tables of all switches in the network. This full architectural control led to new opportunities in the space of TCP congestion research. Datacenters can now be managed in a centralized fashion based on global knowledge of the entire topology, job schedules, and traffic patterns. All the complexity of the network is opaque to the controller. It operates only on the policy defined by the datacenter operator and the data provided from the network.

A ancillary effect of this separation is that the datacenter environment can be treated as an optimization problem. Given the global state of the network the algorithm returns an optimal solution, which is then used to correct suboptimal behavior. An embodiment of such a philosophy is the Hedera [2] flow scheduler, which treats flow collision as a multi-commodity flow problem [28] and reroutes flows to distribute traffic more evenly.

Data in datacenters. Datacenters are a convenient venue for data collection. All network devices, servers, and applications are under a single administrative domain and the delay to retrieve statistics is low. While very precise measurements are still challenging, recent work [81] has demonstrated that it is possible to observe network statistics on a milli- to even a microsecond scale, approaching the update delay of the TCP ACK clock [71].By precisely inferring and measuring data it may be possible to devise a proactive mechanism that forecasts the traffic pattern of next iteration, and adjusts admission rate for hosts accordingly.

It is unclear how predictable DC traffic truly is. Several projects have shown that repeated patterns exist [10, 49, 65, 83, 93]. With monitoring becoming increasingly precise, it may be possible to infer more accurate insights about the nature of datacenter traffic. If, and only if, DC traffic is sufficiently predictable designing a proactive algorithm is feasible. Machine learning, which has the capability of extracting patterns from seemingly random data can aid in this process. While the use of online-learning algorithms for the Internet is controversial [87], Performance-oriented Congestion Control (PCC) [25] and Remy [106] have demonstrated that congestion control algorithms that evolve on trained data can compete with or even

exceed conventional, manually tuned algorithms. We agree that a local, greedily optimizing algorithm may not be able to achieve this goal [87]. Instead, utility needs to be maximized by leveraging *global* knowledge, either in a centralized solution such as FastPass [76] or distributed in the form of an asynchronous message passing solution such as PERC [45].

2.3 Data-Driven Management

We believe the next potential frontier of networking research is the design of data-driven, holistic algorithms, which actively improve their policy via the sampling of live data in the network. Ideally, a datacenter operator will deploy a general purpose system, which then actively adapts to the particular domain-specific workloads, topologies, and idiosyncrasies. The idea of drawing from past data to prepare for the future is attractive. It is particularly viable in datacenters, which are (in theory) fully observable, exhibit specific application patterns, and operate on recurrent tasks. We do not advocate to completely supersede tried and tested approaches. Instead we posit that by designing classical traffic engineering techniques to be parameterizable and explicitly configurable it is possible to achieve effective cross-layer optimization.

Projects such as the congestion control platform (CCP) [71], which redesigns TCP to expose a control application programming interface (API), are a step into the right direction. CCP allows for the implementation of any default TCP congestion control algorithm, which can then be further tuned by adjusting exposed parameters.

2.3.1 The Iroko Emulator

We investigate the potential of data-driven policies by designing an emulator, Iroko, which expedites the prototyping and evaluation of such algorithms.

Iroko represents networking problems as an OpenAI gym [15], which acts as an interface between the concrete reinforcement learning agents and the datacenter. All system-specific information is abstracted away from the agent, providing flexibility in data acquisition and modeling. OpenAI gyms are a common RL benchmarking platform intended to encourage reproducibility and to allow performance

benchmarking of an algorithm across many different domains. Our intention is similar. By explicitly designing the emulator as a gym we hope to lower the barrier of entry for both data scientists and networking researchers and facilitate progress in the design of data-driven networking algorithms. RL researchers are able to challenge their agents in a new setting without requiring extensive domain knowledge. Networking researchers, on the other hand, are able explore data-driven networking solutions with out-of-the box algorithms and solvers.

Chapter 3

Background and Assumptions

This chapter provides a summary on essential background in datacenter networking and RL. It also establishes our baseline assumptions for the development of the emulation platform. A comprehensive summary on the systems used in datacenters can be found in *Datacenter traffic control: Understanding techniques and tradeoffs* by Noormohammadpour and Raghavendra, 2018 [73]. A reader already familiar with the fields of datacenter networking and deep RL may skip to Chapter 4.

3.1 Datacenter Networks

The most important criterion of datacenter design is cost-minimization. Operators prefer to buy switches and standard Linux servers as off-the-shelf hardware and maximize the utilization of the datacenter network[1]. A typical datacenter consists of some number of racks that communicate over an Ethernet network. A rack is defined as an array of several dozen hosts, all of which are attached to a Top-of-Rack (ToR) switch. To provide connectivity between racks, ToRs are typically linked to high-capacity spine switches. To save costs, operators frequently *oversubscribe* their network. Oversubscription implies that the combined bandwidth of hosts in a rack exceeds the total available bandwidth between racks. To compensate for the lack of inter-cluster bandwidth, and to make their network scalable, companies such as Facebook deploy compartmentalized, autonomous pockets of servers,

[1] Although in practice, large companies tend to overprovision their resources

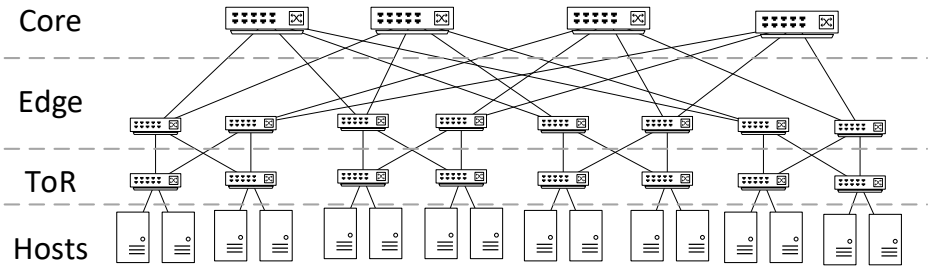

Figure 3.1: A simplified fat-tree network.

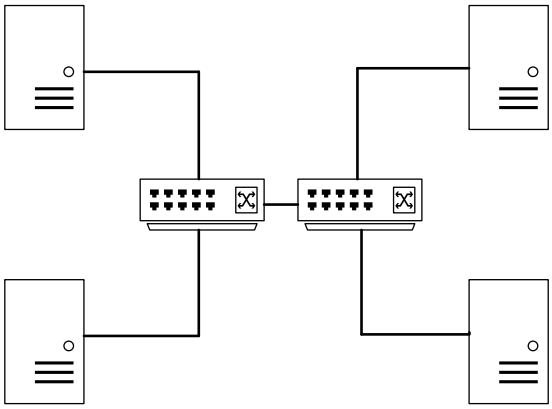

Figure 3.2: The dumbbell scenario.

also referred to as pods [64]. Pods are managed independently and keep traffic largely within the management domain. Datacenter sizes are largely bimodal, large cloud providers run hundreds of millions machines across the globe [42, 83], whereas many smaller businesses and conventional companies set up fewer than a hundred [1] to at most a few thousand machines [47].

We base our design assumptions on smaller datacenters and the pod design, in part because data-driven, centralized solutions are intractable at the scale of modern cloud providers.

3.1.1 Workloads

Datacenter workloads primarily depend on the datacenter type and the applications that are deployed. Datacenters can be private- or public-facing, host customer applications, or provide general services to consumers. Common applications that run in a datacenter include MapReduce-style [24] batch-jobs, web servers or databases serving requests, storage systems, and hosted virtual machines.

Diversity makes it difficult to generalize traffic behavior or patterns. Nonetheless, research literature [9, 49, 83] shows that datacenter workloads exhibit several basic characteristics and challenges.

Flow Types

A large number of varying flow[2] sizes, lengths, and patterns coexist in datacenters [73]. Different applications generate different flow patterns [21]. For example, flows created by batch-jobs jobs are typically long-running and bandwidth heavy, whereas a key-value store serves a flurry of short-lived, small requests. Large, long-running streams of packets are referred to as "elephant flows" whereas the short variant are called "mice flows". Most measurements indicate that the majority of flows are mice flows that transport less than 10 Kilobyte (KB) and only last a few seconds. While there many more mice flows in a datacenter, elephant flows consume the bulk of link bandwidth. Flow arrival patterns largely depend on the size of the datacenter but are commonly recorded to be within a few microseconds to 10 milliseconds.

Flow Requirements

In addition, applications and their flows may have different requirements that need to be satisfied by traffic engineering. For example, a batch-job is typically throughput sensitive and requires high bandwidth to operate, whereas database queries depend on low-latency message delivery. Another class of flows may have to meet strict timing deadlines. If a deadline is missed the result is rendered invalid and is discarded. Such applications require a guarantee that, if they are cleared to send, their data delivery will be successful. An example may be a search query, which

[2] A flow is identified by the "five-tuple": source/destination IP and port plus transport type.

needs to complete before a given deadline or the user will be dissatisfied.

Merely measuring throughput and latency metrics often does not capture these flow requirements and the end-to-end performance of a system. Correspondingly, datacenter systems are often measured on the basis of the 99th percentile of flow completion time (FCT). Averages mask the long-tail latencies that can occur due to unstable traffic and resource usage. FCT can accurately measure the goodput and the latency of a system.

3.1.2 Traffic Engineering

Datacenter operators rely on sophisticated traffic engineering to accommodate their workloads and applications. Measures range from topology design, to traffic control, or flow steering and prioritization.

Topologies

Trees are the most typical topology structure in datacenters, the Fat-Tree [58] being a particularly popular variation. Fat-trees are a three-layer Clos [22]-like topology with multiple tree roots. A nice property of fat-trees is that they are considered to be near-*universal*, meaning that they can express and support any arbitrary topology or communication pattern with only minimal performance loss.

In a fat-tree ToR switches are connected to middle-layer aggregation switches, which are in turn linked together via core switches. In the three-layer network, any host will have to traverse at most five switches to reach another host (e.g., Fig. 3.1). Despite the high-path diversity, measurements [9] indicate that fat-trees lose packets at every layer, with the core layer experiencing the most persistent hot-spots.

We also model the dumbbell topology (Fig. 3.2), a simplified version of a fat-tree network. The dumbbell topology represents an oversubscribed slice of a fat-tree network, for example an inter-rack path. Dumbbell topologies are a popular format to evaluate new traffic control techniques as they isolate flow collision and fairness problems and are easy to comprehend and debug.

Routing

Tree topologies provide path redundancy and multiple end-to-end paths for packets in the network. However, to prevent loops, outages and to balance traffic fairly, these topologies require meticulous routing strategies. Straightforward routing in a datacenter network is either performed either on layer three using conventional protocols such as Open Shortest Path First (OSPF) [69], or on layer two by forwarding based on MAC addresses only. These simple routing schemes guarantee connectivity, but to balance traffic across links, static forwarding strategies such as Valiant Load Balancing (VLB) [114] or Equal Cost Multi Path (ECMP) [97] have to be deployed. Modern implementations of VLB spray flows randomly across all possible links whereas ECMP determines per-flow forwarding decisions via consistent hashing. Unfortunately, because these techniques operate per-flow, randomness often triggers collisions of elephant flows. This causes the affected output port to be choked, impacting all active flows on the path. Many solutions to evenly distribute packets across the data links exist [73]. For simplicity sake, we assume a statically configured ECMP network. Our data-driven traffic engineering platform is able to support these routing strategies, but we have decided to focus on congestion control for this particular project.

Congestion Control Challenges

Careful routing mitigates loss in networks, but cannot prevent hosts from oversaturating popular links. To ensure that no flow is able to crowd out other participants[3], congestion control limits the rate of data entering the network. Every congestion control algorithm aims to satisfy two fundamental requirements: Reach a stable state for a given network matrix and allocate a fair share for each participant.

A common technique to achieve this state is to use the Additive increase/multiplicative decrease (AIMD) algorithm in TCP, which is proven to eventually converge to a fair rate, given that all participants use the same technique [50]. Note the "eventually" in this statement. In datacenters, propagation delay is extremely low and the available bandwidth high. Every host is sending at maximum rate and small

[3] We consider any flow that has entered the network a participant.

short-lived flows are competing for bandwidth with long-running elephant flows. Datacenters face several congestion challenges, because of the highly specialized topologies and routing techniques paired with application-driven traffic patterns.

Microbursts Most flavors of TCP have relied on slow-start and the loss-based congestion avoidance techniques originally developed by Van Jacobson [41]. TCP effectively operates as a burst-and-backoff protocol which quickly reaches high-bandwidth while continuously probing the network. Unfortunately, slow start's bursty behavior frequently leads to a destructive pattern at switch ports, known as "microbursts". Microbursts cause high loss at switch links when multiple quick burst of packets, which exceed the switches' forwarding capacity, arrive. The sudden wave of flows fills up the switch buffer and causes packet loss for all flows on the same link, detrimentally affecting stable flows by forcing them to unnecessarily reduce their sending rate. While remediation tactics, such as traffic shaping, exist, microbursts remain a persistent problem in today's datacenters. In fact, measurements by Zhang et al., 2017 [112] indicate that microbursts today last only several hundred microseconds. These bursts are often not captured by standard network telemetry but still have significant implications on the performance of congestion control algorithms.

Bufferbloat To compensate, switch vendors have introduced big buffer switches [8] and continuously increased the size of switch port buffers. A consequence of the increased buffer size of switches is "bufferbloat" [33]. Bufferbloat describes a phenomenon that occurs on switch queues that are perpetually filled with packets, which adds queuing delay to packet transmission in the network. Because TCP uses only packet loss as congestion indicator, it does not detect this buffering and thus does not decrease its sending rate. In wide-area networks, where large propagation latency masked the queuing delay, this was considered acceptable. In datacenters, however, many applications are latency-critical and require reliable performance. Bufferbloat has led to the development of delay-based TCP algorithms, which, instead of using loss, try to infer congestion signals by measuring latency variation. The intention is to identify queuing as early as possible and

reduce the sending rate in a timely manner.

Incast Incast refers to a bottleneck phenomenon originating from the use of aggregation patterns such as MapReduce [24]. In the aggregate phase, many servers reply to the aggregator in parallel, cause bottlenecks at the inbound link, and drown out other flows. Because all flows flood in at once, stabilization takes a substantial amount of time. Incast problems are highly correlated with microbursts and are usually handled by limiting the data transfer rate at end hosts.

Datacenter Congestion Control

Delay-based algorithms are the current control mechanism of choice in datacenters. Popular variants include TCP New Vegas (TCP NV), TIMELY [66], and the DCTCP [3] algorithm. Because a public implementation of TIMELY was not available during the development of this project, we focus on the use of TCP NV and DCTCP.

TCP NV TCP Vegas is an end-to-end delay algorithm originally developed by Brakmo and Peterson [14]. Vegas gauges network capacity by probing the RTT of TCP packets instead of reacting to explicit loss of packet segments. TCP Vegas increases its sending rate more conservatively than standard TCP and recedes once measured latency exceeds a specific threshold from the base experienced minimum latency. By doing so, TCP Vegas ensures that switch queues are emptied periodically and kept low. TCP NV is based on the original TCP Vegas and has been optimized by Facebook to operate in datacenters with more than 10Gbps link bandwidth. TCP NV has been contributed to the Linux kernel and is available since version 4.8 [13].

DCTCP While delay-based congestion control performs better than standard TCP in datacenters, it is dependent on potentially noisy RTT measurements. This flaw becomes palpable in low-latency topologies, which have limited tolerance for variance. The minimum and current experienced RTT inferred by TCP NV is distorted by unstable server performance due to interrupts, coalescing, slow TCP stacks, or

high CPU load. At the speed modern NIC hardware is operating, RTT becomes an unreliable congestion signal .

A popular alternative to end-to-end TCP is DCTCP, which requires explicit feedback from the network. DCTCP depends on Active Queue Management (AQM) [7] on switches, which marks excess packets that arrive over a configured queue threshold with an Explicit Congestion Notification (ECN) tag before delivering them. DCTCP uses the ECN tags on packets to preemptively adjust the sending rate. This simple technique has seen success in datacenters and established DCTCP as a popular datacenter protocol. In our experiments we treat DCTCP and TCP NV as baselines.

Datacenter Scheduling

TCP algorithms are typically designed to be agnostic of flow priorities. To adequately support variable sized flows and deadlines, many datacenter operators make use of scheduling strategies. Dynamic load-balancing strategies, such as Hedera [2], approach this problem by proposing a centralized network controller, which identifies colliding flows and relocates them to underutilized links. Systems such as FastPass [76] go a step further and explicitly allow network admission only to flows that are prioritized and do not exceed the capacity threshold of the network.

A fundamental strategy necessitated by congestion is the use of queuing disciplines at switch buffers. Queuing disciplines prioritize packets in the queue based on specific characteristics. Flow and queue scheduling can highly influence the performance of any transmission control protocol. For example, DCTCP's performance depends on the marking threshold configured on the queue management algorithm at the switch buffer [107]. If the switch tags packets too early, DCTCP can't achieve optimal throughput, if packets are marked too late then DCTCP will perform even worse than conventional TCP. Many different queuing strategies exist. We follow the advice outlined by Judd, 2015 [47] and use Random Early Detection (RED) [31] with ECN marking at the switch when deploying DCTCP. Otherwise we rely on simple drop-tail queuing (packets at the end of the queue are dropped).

Utility Maximization Networks integrate various techniques, such as routing, scheduling, or rate-limiting to optimize the flow of traffic. Each technique can be interpreted as a different layer which controls a subset of the network, has a limited view of state, and optimizes a local objective. This design has contributed to the success of the Internet, but may be ill-suited for datacenters. Even subtle parameter changes in one domain can dramatically affect others and causes sub-optimal global behavior. For example, increasing the packet marking threshold of RED can dramatically lower DCTCP's congestion performance. Or a tweak in a routing scheme, such as ECMP, may affect the total amount of flow collisions in the network, which then has to be compensated by congestion protocols or load balancers.

Most datacenter engineering solutions are deployed *ad hoc* [19]. Tools and algorithms are tuned based on tribal knowledge and good practice instead of rigorous formalization. Ideally, a network should be designed and parametrized with all layers and constraints in mind to optimize a global objective. The field of Network Utility Maximization (NUM) (also referred to as "Layering as Optimization Decomposition") [19] has been developed to solve precisely this disconnect of individual network mechanisms and the global utility goal.

NUM *views the network as the optimizer itself, puts the end-user application needs as the optimization objective, establishes the globally optimal performance benchmark, and offers a common set of methodologies to design modularized and distributed solutions that may attain the benchmark* [19].

In recent years, NUM has found popularity in congestion control theory. Rate control schemes such as Remy [106], PCC Vivace [25], or Copa [6] express the congestion control problem as an objective function. Instead of using an explicit control window these schemes continuously infer utility from observed network metrics and gradually adjust sending rate. Eventually, the algorithms converge to a strategy that performs best in the given network environment.

Our network emulator is inspired by this very idea of defining and solving an objective function. In particular, we believe that, because of its approximation capabilities, RL can serve as a viable solution to design traffic engineering applications that span across multiple layers and efficiently solve the NUM problem.

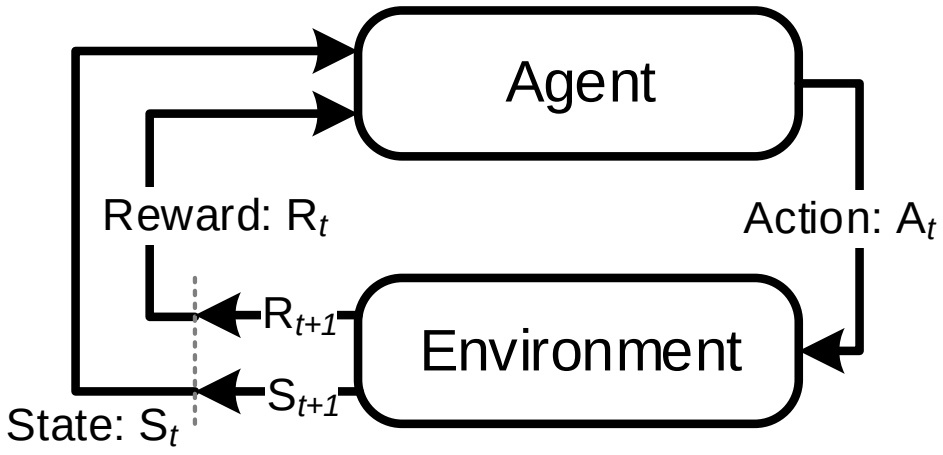

Figure 3.3: The typical reinforcement learning process.

3.2 Reinforcement Learning

We specifically investigate the potential of reinforcement learning heuristics in the datacenter setting. The fundamentals in this section are primarily drawn from the introductory book by Sutton and Barto [95].

Machine Learning Reinforcement Learning is a subclass of ML. Broadly speaking, ML refers to a class of algorithms capable of inferring a mathematical model by ingesting large amounts of data. Most use-cases are formulated as an optimization problem, in which the algorithm minimizes a *loss* function that describes the difference between the predicted and actual result. In the most common form of ML, supervised learning, the algorithm ingests a *training set* (a subset of the full available data) and performs gradient descent [51] until the loss has decreased to an acceptable value or all data has been sampled. The goal of an ML algorithm is to learn a distribution that most accurately classifies given input data while avoiding ”overfitting”, i.e., learning a mathematical model that merely describes the set of data the algorithm has processed. The ideal ML algorithm generalizes to data it has not been trained on from merely observing a small sample set.

3.2.1 The Reinforcement Learning Problem

RL can be motivated by the conditional sub-domain of behavioral psychology. RL mimics an *agent* that is conditioned to adapt to an unknown environment. Instead of minimizing the loss, the agent improves its behavior by receiving a *reward* from its outputs. It does so by "stepping" through the environment. In each step, the agent observes the current state, performs an action, and receives a reward. The function output of an RL algorithm is considered an *action* whereas the reward function is provided by the environment. Fig. 3.3 shows the typical cycle. Actions, which lead to a desired outcome are rewarded positively whereas wrong behavior is penalized by the reward function. The goal is to learn a *policy* that maps optimal actions for a given input state.

Although RL is often compared to mammalian perception and behavior, it is purely data-driven and operates best in data-rich environments. A datacenter can be viewed as such an environment and can be designed to provide RL algorithms with the information they need to maximize varying objectives. A particular challenge of RL is that the algorithm output may influence the subsequent sampling of data. This property is modeled as a form of a Markov Decision Process (MDP). A RL MDP contains the following components:

1. The initial state: s_0.
2. A set of possible states the environment or agent can be in: $S = s_0, s_1, ..., s_m$.
3. A of possible actions an agent can execute: $A = a_0, a_1, ..., a_n$.
4. A transition model which returns the next likely state under a specific action: $T(s, a, s')$.
5. A reward function which describes the benefit gained from reaching a certain state : $R(s)$.

Actions are selected on the basis of the policy $\pi(s)$, which maximizes the reward for the agent for a given state. The state-value function $V_\pi(s)$ describes the expected reward for the current state under a specific policy. This function makes use of the Bellman Equation, which introduces the discount factor $\gamma \in [0, 1]$. γ denotes the preference of the agent to prioritize future rewards over immediate reward. The overall value function can be denoted as $V_\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$, which estimates the total reward until a terminal or cut-off state is reached. Complementary

to the state-value function, the Q-function $Q(s,a)$ defines the expected future reward given a state s and an action a. The optimal value function $V^*$is often defined as the aggregate of the optimal Q-function Q^* for all states: $V^* = \max_a Q^*(s,a)$. The Q-function defines the utility of a state-action pair, which then allows the construction of a policy based on the highest Q returns. In essence, the Q-function defines the value of an action in a state, whereas the policy $\pi(s)$ defines which actions to pick. Since both functions are complementary, two fundamental techniques to improve the behavior of a RL agent exist: Value Iteration and Policy Iteration.

Value Iteration Value Iteration, also known as Q-learning, is an off-policy algorithm, meaning it does not depend on a pre-defined policy in order to choose a transition function. In Q-learning, the agent begins with an arbitrary, fixed policy π and will generate an optimal policy π^* in parallel. It does this by selecting the predetermined Q-function value as the next action, which may be unreliable or useless. Over time, Q-learning improves the agents second Q-function Q^* by observing the effects of the first policy. It uses the following update function:

$$Q(s_t,a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimated optimal value}} \right)}^{\text{learned value}}$$

$Q(s_t,a_t)$ is updated based on the newly experienced reward and the former estimated return, eventually approximating an ideal Q-function Q^* and overall policy π. The original Q-function Q and policy π do not change.

Deep Q-Networks If the state-space is too large, function approximators such as neural networks are used. However, neural nets are known to behave erratically in reinforcement learning due to the assumption of independent and identically distributed (i.i.d) samples. RL problems are often highly variable which leads to instability between training episodes. As a mitigation tactic, Deep Q-Networks (DQNs) [67] are introduced. In addition to using a convolutional neural net to model the state-action space, DQNs leverage *experience replay* [95] as a stabilization mechanism. Experience replay collects a buffer of historical transition models and randomly inserts the transition values during the Q-update function.

Any historical transition may influence the future action selection, thus averaging the behavior distribution overall and mitigating the occurrence of local minima or divergence. The sample set more closely resembles an i.i.d. distribution.

Secondly, in every state-action update the algorithm trains a target network independently of the actual update function for several update epochs. This guarantees that an update to $Q(s_t, a_i)$ does not immediately impact updates to $Q(s_{t+1}, a)$. If the model is at risk of diverging or oscillation, this method establishes a potential buffer to stabilize. It can be considered a hybrid off-policy approach as the policy is only improved every N-steps.

Policy Iteration Instead of improving the Q-function, Policy Gradient (PG) methods optimize the policy function directly and learn a stochastic policy function $\pi_\theta(a|s)$ with respect to the weights θ. Actions are sampled from a distribution instead of being picked on the basis of their utility. A policy gradient algorithm generally uses gradient ascent to infer the best values of θ, i.e., the policy, that maximize the expected reward return. PGs are commonly used in environments in which the transition model is unavailable and the true value of actions is unknown (also known as *model-free* RL). The value for a given action has to be inferred by continuously probing and re-evaluating the policy. To do so, PG agents often use Monte-Carlo (MC) methods to learn a policy. MC randomly samples data points to infer a probability distribution and observe properties about a given, initially unknown scenario. The stochastic approximation returns an action, which is likely to be most rewarding and is able to handle even highly continuous and large state-spaces. This makes policy gradients particularly attractive for use in datacenters, where the policy and state-transition matrix are largely undecipherable by humans and must be inferred. PG methods can provide an approximation which accurately describes traffic behavior and may help in finding optimal actions in a highly stochastic environment.

3.2.2 REINFORCE

REINFORCE [95] is a basic instantiation of a policy gradient. REINFORCE uses MC sampling to generate random state transitions and is thus also known as MC

Policy Differentiation. The pseudocode of Algorithm 1 details the exact steps of one episode. In line six, the values of θ are updated proportional to the probability α of the action and the reward r_t. The higher the reward, and the lower the probability of the action, the more the policy is skewed towards the taken action in the next episode. Note how the policy gradient REINFORCE optimizes towards the likelihood of taking an action, whereas Q-learning improves the utility of taking an action. Eventually, good actions will have an increased likelihood to get sampled in future iterations.

We use REINFORCE as a baseline in our experiments. Its sampling policy essentially approaches random choice, which causes high variance in policy updates and ultimately leads to instability. If REINFORCE is able to perform well, the implication is that the environment does not represent a challenging enough problem to warrant more sophisticated approaches.

Algorithm 1 REINFORCE

1: Input: A differentiable policy parametrization $\pi(a|s,\theta)$
2: Initialize policy parameter θ randomly
3: **for** $t = 1,2,\ldots,T$ **do**
4: Generate a state-action trajectory: π_θ:$S_0,A_0,R_1,S_1,A_1,\ldots,S_T$
5: $r_t \leftarrow$ reward return from step t
6: $\theta \leftarrow \theta + \alpha\gamma^t r_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$
7: **end for**

3.2.3 Deep Deterministic Policy Gradient (DDPG)

Continuous Control While DQNs were able to model settings with high-dimensional *state* spaces, they were unable to pick sensible actions when facing continuous, high-dimensional *action* spaces. For example, agents in physical environments have to sample from an unbounded range of fine-grained output values to operate in a precise fashion. This limitation also applies to rate-control in datacenters, which may range from mere bits-per-second to GB of throughput. While it is theoretically possible to discretize the actions, the degrees of freedom quickly leads to a curse of dimensionality and an action-space explosion [59].

Algorithm 2 The DDPG algorithm.

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

To handle such a class of problems, Deterministic Policy Gradients (DPGs) [91] can be used. DPGs are capable of representing continuous action spaces as one cohesive policy and iterate over the state space, thus reducing the amount of required computational power dramatically. DPGs traditionally rely on an Actor-Critic (AC) model.

AC is considered a decoupling technique that splits a reinforcement system into an action-selection and action-evaluation component. Actors are any type of RL techniques, which actively improve the policy based on new action-value informa-

tion. In general, the actor computes a full policy based on a policy gradient method whereas the critic uses Q-learning to guarantee an off-policy Q-value improvement.

DPG is limited in its ability to represent complex or perception-based state spaces. This causes it to become unstable and slow in high-dimensional state spaces. To be able simulate movement and physical environments, Lillicrap, Hunt, et al., 2017 [59] developed the Deep Deterministic Policy Gradient (DDPG). DDPG uses two neural nets as the actor and critic and combines the model with experience replay and as well as the periodical buffering of Q-functions to guarantee stability.

Algorithm 2 describes the implementation of DDPG. DDPG is a combination of DQN and DPG. Drawing from DQN, it uses experience replay and updates target values based on sampled transitions. The algorithm buffers target values in the replay buffer R to prevent quick divergence caused by the use of neural networks. The actor-critic model introduced by DPG is intended to handle the large state space and prevent divergence. The actor decides on an action deterministically on the basis of an approximation function $\mu(s_t|\theta^{\mu})$. This also affects the computation of the target y_i which is now updated on the basis of $\mu(s_{i+1}|\theta^{\mu'})$, i.e., the action chosen by the buffered target actor for state s_{i+i}. The critic itself attempts to minimize the sum of least-squares and updates its target weights θ^{Q} correspondingly. Afterwards, the actor modifies its policy following the critic's updated action-value function $Q(s,a|\theta^{Q})$. As a last step, the buffered weights for the actor and critic function are "softly updated" using a regularization parameter τ. This smoothing ensures that the algorithm remains stable and does not drastically change actions after an update.

Due to its alleged properties of being able to model highly complex, continuous environments and act deterministically, we chose DDPG as a representative state-of-the-art algorithm for value-iterators.

3.2.4 Proximal Policy Optimization (PPO)

Trust Region Policy Optimization A major challenge for PGs is the estimation of the gradient step size. A step size that is too small will lead to slow improvement. On the other hand, if the step size is too large, the agent is likely to "fall of a cliff", i.e., it will get stuck in a local minimum. For example, in a datacenter it may be

hard to recover from an erroneous traffic update that causes congestion.

Trust Region Policy Optimization (TRPO) [88] is a technique that ensures that every gradient update is within a reasonable range while still ensuring continuous improvement. The core of TRPO is the use of Kullback-Leibler divergence [54], which computes the distance between two probability distributions. If the distance between the old and new policies is too large before an update, TRPO will smoothen the gradient update and ensure that the new θ parameters do not cause divergence. An important property of TRPO is the *guarantee of monotonic improvement*. Any update to the policy is proven to improve over the old parameter space. This makes TRPO particularly attractive for environments in which even minor changes can lead to high penalties (suboptimal system states).

A major constraint of TRPO is its computational cost and complexity, limiting its usability in more complex scenarios. PPO [90] has been designed as a practical alternative and is typically viewed as a more scalable version of TRPO that retains its monotonic behavior. In essence, PPO clips the update function if the step size violates a predefined boundary.

PPO vs DDPG We use both PPO and DDPG as sample RL algorithms in our emulator evaluation. Our intuition is that the algorithms will learn to adapt in the datacenter environment, as both are designed for tasks that exhibit complex continuous action and state spaces. A major difference between PPO and DDPG is that PPO operates on-policy whereas DDPG uses Q-learning to improve off-policy. An on-policy algorithm typically exhibits higher variance because the policy is updated every step and may require more samples to improve. In addition, because PPO does not use experience replay like DDPG, spare rewards may cause it to stall. On the other hand PPO is more robust than DDPG due to its monotonic improvement guarantee in stochastic environments. Outliers will not skew the policy.

Chapter 4

The Iroko Emulator

4.1 Emulator Design

A key feature of Iroko is its extensible and modular design. The *Gym* is assembled by combining a set of core tools, including a network topology, a traffic generator, datacenter monitors, and the OpenAI Gym interface (see Figure 4.1). The emulator's flexibility allows it to support centralized arbiters as well as decentralized, host-level approaches. For example, in a congestion control scenario, decentralized agents could be traditional TCP algorithms, such **TCP New Vegas!** (**TCP New Vegas!**), whereas centralized agents might be RL algorithms. The emulator can also implement hybrid deployments, which constitute multi-agent systems [94]; one agent may be a rate-limiting arbiter, the other could configure routing. Together, these agents optimize towards a common goal. In designing Iroko we are particularly concerned about supporting the following four require-

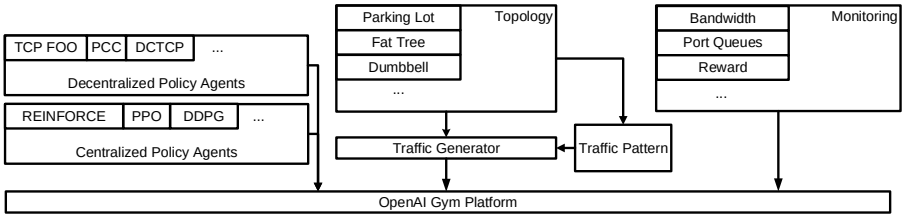

Figure 4.1: Architecture of the Iroko Gym.

ments:

(1) Arbitrary topologies and traffic patterns: As outlined in Section 3, datacenters are highly diverse. To be effective, Iroko needs to be able to support most common types of topologies and traffic patterns.
(2) Low CPU Utilization: To approximate mid-tier datacenters, Iroko has to scale to several hundreds of hosts on a modern commodity server. This mandates that all the core components of the emulator are lightweight and CPU-efficient. We can not waste disk space, memory, or CPU cycles.
(3) Flexible monitoring: Data-driven networks rely on periodic sample collection. The platform has to provide a set of tools and APIs that allow a user to dynamically assemble a state matrix and infer system properties. In addition, the monitoring needs to be light-weight and fine-grained to capture the majority of events during emulation.
(4) Algorithm Support: The emulator interface should be generic enough that reinforcement learning, optimization solvers, or decentralized algorithms can be fairly compared on the same platform. This requires an API that is common-purpose enough to interface with any of these approaches. Fortunately, the openAI Gym interface satisfies this requirement.

Listing 1 A policy interacting with the Iroko Gym.

```
from dc_gym.factories import env_factory
def test_run(config, policy, n_steps):
    # Create the testing environment
    dc_env = env_factory.create(config)
    # Initialize the environment
    obs, reward = dc_env.reset()
    # Step through the environment for n steps
    for epoch in range(n_steps):
        # Sample a random available action
        action = policy.compute(obs, reward)
        # Perform the action
        # Retrieve reward and observation
        obs, reward = dc_env.step(action)
    # All done, tear down the environment
    dc_env.close()
```

4.1.1 The Gym Interface

Our emulator primarily draws from the OpenAI Gym API to ensure compatibility with modern RL algorithms. Figure 1 shows a simple policy interacting with an Iroko Gym. The `env_factory` module uses the configuration file to assemble a Gym consisting of a topology, traffic pattern, state matrix, reward function, and action enforcement. The environment is then initialized via the `reset()` call, which creates the virtual environment and starts traffic generation. The call also returns an initial state matrix and reward. After initialization, the policy proceeds to `step` through the environment, picks an action every iteration, and observes the result and reward. The policy is free to ignore any variable provided by the environment. For example, decentralized TCP does not use observation, action, or reward, but the system still collects metrics on the theoretical reward the policy

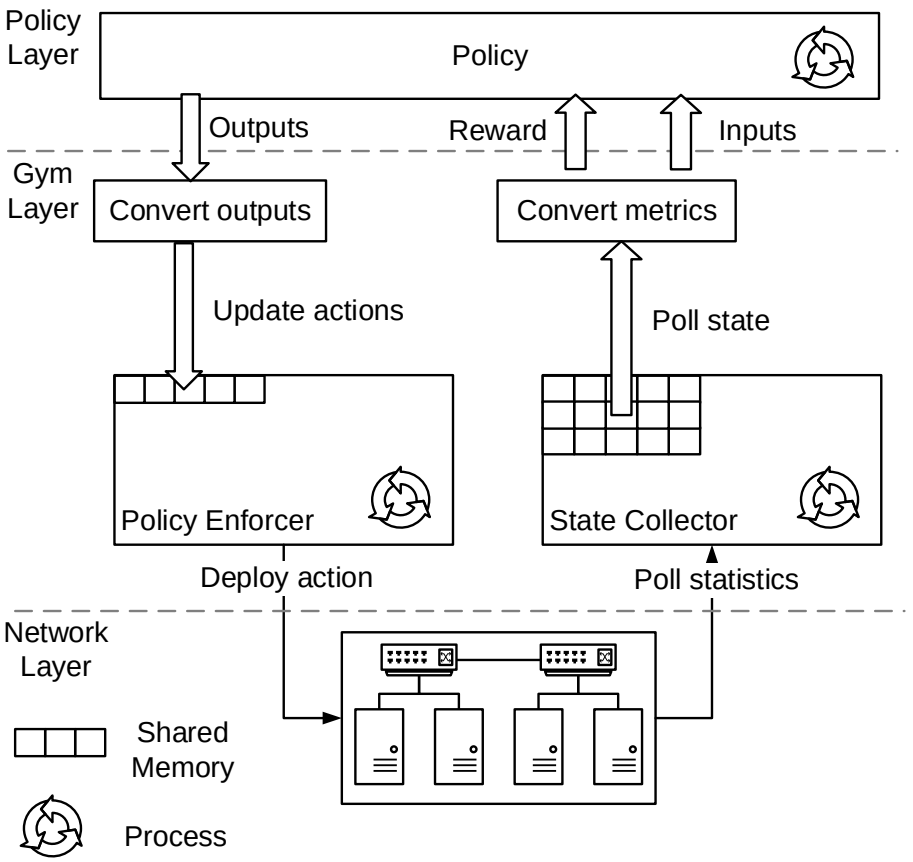

Figure 4.2: The Iroko process flow.

achieves.

4.1.2 Network Emulation

We use the Mininet [56] real-time network emulator to construct and deploy network topologies. Mininet generates a virtual network topology using the network emulation facilities provided by the Linux kernel [36]. In Mininet, each "host" runs in its own isolated network namespace but otherwise uses the default Linux application binary interface (ABI). The hosts are connected via Open vSwitch [78], a common high-performance software switch. The emulator comes with a set of

Table 4.1: A sample flow pattern.

Source	Destination	Port	Start (s)	Length (s)	Flow Size (Byte)	Reps	Flow Gap (s)
H1	H3	80	0.0	0.0	1000000	inf	0.0
H1	H4	80	15.0	15.0	inf	10	1.0

pre-defined topologies that can be imported by the Gym. New topologies can easily be added. Each topology defines a set of available traffic patterns that can be used by the Gym to generate testing traffic.

Why Mininet? We chose Mininet over discrete event simulators such as ns-3 [82] or Omnet++ [103] because of its ability to approximate real network behavior. Discrete simulators do not capture the burstiness, interrupts, and idiosyncrasies of the kernel and hardware stack. Simulated protocols often have to undergo substantial revision before they can be deployed [74]. In contrast, systems running on top of Mininet use real OS components and applications. An application that successfully runs on the emulator can be ported to actual hardware more easily than an abstract protocol optimized for a simulator. While Mininet is entirely virtual and is limited in its ability to generalize datacenter behavior, it provides an environment most similar to real datacenters. Mininet already has been effectively used as platform for larger emulator frameworks [23, 77] and can also be scaled across machines using systems such as Maxinet [104]. Our other intention is to craft a challenging environment for algorithms. Any policy that performs well in our emulator has the potential to succeed in real scenarios.

4.1.3 Traffic Patterns

Because the emulation platform does not include any type of datacenter application by default, we synthesize datacenter traffic. However, generating real network traffic is a challenging task. A traffic generator in the system has to be lightweight as well as capable of supporting diverse traffic matrices and virtual interfaces. Unfortunately, many traffic generators do not meet all three of these requirements. For example, iPerf [27] and Netperf [40] are CPU-efficient and support virtual interfaces but are capable of generating only static flows. TRex [39] and WARP17 [48],

on the other hand, are expressive, but have limited support for virtual interfaces and consume a lot of CPU cycles.

We use a modified traffic generator (Goben [101]). At the beginning of a training episode, the generator is launched on all datacenter hosts. Each host is fed with a flow matrix inspired by the Hedera [2] benchmark and the default Mininet traffic generator. Table 4.1 contains an example. `Host 1` constantly sends 1 Megabyte of traffic to `Host 3`. After 15 seconds, `Host 1` starts a second flow to `Host 4` which lasts about 15 seconds and is repeated 10 times with a gap of one second between each flow. Not shown: Both inter-flow and inter-packet gaps can also be randomized to introduce jitter.

Note that the flow generators are a complementary feature of the emulator, intended for microbenchmarks. They are unnecessary if real application macrobenchmarks are performed.

4.1.4 Network Monitoring

One of the core design principles is to run the Gym asynchronously of the hosts, policy enforcers and data collectors (see Figure 4.2).

Monitoring is handled by the *State Collector*, which assembles information in a state matrix and records data for evaluation. The State Collector launches a process per monitor and initializes a memory map shared between all child processes. The state processes independently collect metrics from the network and atomically update their shared memory region per user-configured collection interval. When the policy agent requests a state update, the Gym selects features from the memory region based on a pre-configured index and converts the values into a compatible input matrix. This asynchronous design guarantees that both state and policy can be computed independently and minimizes the propagation latency of network updates.

The current Gym configuration defines which monitoring tools are launched. The Iroko Gym can support any software native to the Linux kernel. However, to keep CPU utilization as low as possible, and to provide very fine-grained measurements, we developed custom tools that provide a dedicated query API for the agent.

We currently focus on the monitoring of interfaces. We have implemented monitors that record interfaces statistics, sending and receiving bandwidth, and the active flows in the network.

4.2 Use Case: A Congestion Control Gym

We developed a proof-of-concept OpenAI Gym, which models the bufferbloat problem in datacenters. The Gym is kept simple and explicitly focuses on the congestion control layer. The goal is to reduce switch interface queues, as they indicate congestion and inefficient flow distribution. The only possible action per step is to adjust the allowed output bandwidth of each host interface. In essence, any RL agent interacting with this Gym has the objective function of centrally approximating a congestion control scheme that outperforms traditional competitors.

4.2.1 Centralized Control

We opted for centralized management of the datacenter, which requires a global view of the network state. A central autonomous arbiter can manage nodes based on its current optimal network model. It periodically queries switches or hosts to update its global view and executes actions based on this information. It learns by observing the network state feedback and continuously updates its action policy. The central, managed approach has several advantages. Firstly, the controller acts as a propagator of global optimal information, which it can disseminate across the network in the form of actions and sending rate policies. Secondly, the agent acts as an interface between the RL algorithm and the data center. Concrete datacenter information is abstracted away from the algorithm, providing flexibility in data acquisition and modeling. Thirdly, the major concern of signaling latency delay of central scheduling systems is mitigated by operating asynchronously and predictively. This is especially true for reinforcement learning algorithms, which select output values based on their *anticipated* utility trajectory, a scheme that follows the proactive congestion control school of thought [45].

4.2.2 Defining the Environment State

A fundamental challenge is to formalize what the algorithm ”observes”, i.e., how its environment represented. The available options of data acquisition in a DC are copious, ranging from simple switch statistics, to application flows, job deployment monitoring, or even explicit application notifications. The agent can predict localized surges of traffic from an incoming job request or flow pattern and execute anticipatory actions to soften the impact. Ideally, the central controller should have knowledge about all activities in the data center to infer traffic correlations. This may be infeasible due to constraints in security as well as hardware monitoring.

We focus on the transport layer. The agent remains independent of higher layer applications and operates only based on its view of the global datacenter. Iroko deploys monitors that collect statistics from switches in the network and store them as an $d \times n$ traffic matrix. The matrix models the datacenter as a list of n ports with d network features. Theoretically, it is possible to query for switch buffer occupancy, packet drops, port utilization, active flows, RTT. End-hosts can provide metrics in goodput, latency, jitter, and individual loss. In this scenario, the agent uses only switch buffer occupancy per interface[1] and the measured delta per iteration. Queue length is a discrete value and can be inferred as quickly as RTT, making it an equivalent congestion signal metric. We use the Linux Netlink [85] API to update the state matrix on the scale of microseconds, which is sufficient to sample the majority of queues and DC flows [18, 83, 112].

4.2.3 The Agent Actions

As discussed in Section 3 common options to mitigate congestion include admission control [20, 76], load-balancing of network traffic [2, 10], queue management [7, 34], and explicit hardware modification [3, 5]. As TCP is inherently a self-regulating, rate-limiting protocol, we focus on admission control as a tool to moderate excessive traffic in the simulator.

Admission control can be expressed in varying granularity, from the flow to the switch-level. Ideally, the agent would model every single flow in the network to guarantee an optimum. This, however, is infeasible, as accounting for short bursty

[1] We assume that switches in a real datacenter are capable of reliably providing these statistics.

flows and isolated transmissions (a substantial port of flows in datacenter [49, 83]) would increase the management complexity of the controller dramatically. Instead, this system rate-limits on a per-interface basis, and provides each host with a minimum guarantee of safe transmit. Instead of operating on the full bandwidth provided by the interface, a host is just able to send on a subset. The controller guarantees that the host will not experience latency increases or loss on this subset. The host itself still retains the capability to optimize flows locally, and spreads bandwidth allocation per flow based on demand or priority.

In an ideal instantiation of a DC under this system, packet-loss will only rarely, if ever, occur, with minimal impact on network utilization.

Implementation

The control scheme specifies the percentage of maximum allowed bandwidth each host can send. We represent this action set as a vector $\vec{a} \in \mathbb{R}^n$ of dimensions equal to the number of host interfaces. The Gym converts the agent output to the appropriate action range. We have found hyperbolic tangent normalization to be the most effective normalization technique:

$$\mathrm{a}_{norm} \leftarrow (\tanh(a_i) + 1.0)/2.0 * (a_{max} - a_{min}) + a_{min} \quad \forall\, \mathrm{i} \in \mathrm{hosts} \tag{4.1}$$

The converted actions represent a percentage allocation and are multiplied by the interface bitrate bw_{max}:

$$\mathrm{bw}_i \leftarrow a_{norm} * bw_{max} \quad \forall\, \mathrm{i} \in \mathrm{hosts} \tag{4.2}$$

The policy enforcer receives this input and sends control packets to each host in the network. Each host runs a token bucket traffic shaper accessible via netlink which controls the interface bitrate.

4.2.4 Congestion Feedback

The Iroko emulator allows the definition of arbitrary reward functions based on the provided input state. Choosing an appropriate reward function is crucial for the agent to learn an optimal policy as mismatched reward can lead to unexpected

behavior [57]. A naive model would simply involve giving a positive reward when a reduction in queue length is measured, but this can lead to a sub-optimal policy; no queues can occur if no data is sent. A related concern is the credit assignment problem: It is unclear if actions and not random chance (e.g., a host being shut off, changes in sending patterns, etc.) caused the buffer reduction.

We follow a common trade-off model, inspired by recent work on TCP CC optimization [93]:

$$R \leftarrow \underbrace{\bar{a}/a_{max}}_{\text{action reward}} - \underbrace{2}_{\text{weight}} * \underbrace{\bar{a}/a_{max} * (\max_i q_i/q_{max})}_{\text{queue penalty}} \tag{4.3}$$

This equation prevents switch queues while trying to find the optimal bandwidth allocation for each host. In the equation, *action reward* is the average bandwidth allocation for each host on a percentage range from 0 to 1. It is the only positive feedback, incentivizing to raise its action output to maximize throughput. We selected a reward parameter that is independent of the network state to guarantee that the agent's objective function does not depend on current network utilization.

Queue penalty is a weighted action-dependent penalty calculated based on the largest queue in the network. The maximum observed queue utilization on a range from 0 to 1 is subtracted from the action reward. The higher the average action output the higher the penalty. We use a fixed weight factor of 2, which ensures that the penalty dominates the reward. This model encourages a lower action output when queues in the network are high. Once queuing has decreased to near zero, high action output becomes rewarding again. Ideally, agents eventually converge to a rate which maximizes throughput without incurring high latency.

Chapter 5

Preliminary Experiments

The emulator has to be reliable, fair, and must generate results that are at least approximate to real-world behavior. We run several tests to assess the current benchmarking properties of the platform. More specifically, we had the following questions in mind while designing the experiments:

1. Does the emulator satisfy the scalability requirement?
2. Does the Gym accommodate state-of-the-art RL algorithms?
3. How well does a purely data-driven solution perform?
4. How do different bandwidths affect Iroko's performance?

5.1 Scalability Tests

Since Mininet operates in real-time and uses general purpose Linux tools, it does have scalability limitations. We design a simple load-generation scenario to identify constraints. We start out with a dumbbell topology that has four hosts connected using two switches and a single 10 Megabit (Mb) link[1] (Figure 3.2). Hosts on one side are sending constant 10 Megabit per second (Mbps) traffic to the hosts on the opposite side, causing congestion on the central link. With each iteration we increase the stress on the emulator server by doubling the number of Mininet hosts. We run two variants of this experiment. In the first version, all interfaces are

[1] We use 10 Mb as the default size because it is low enough to alleviate CPU burden while still allowing comparable networking benchmarks.

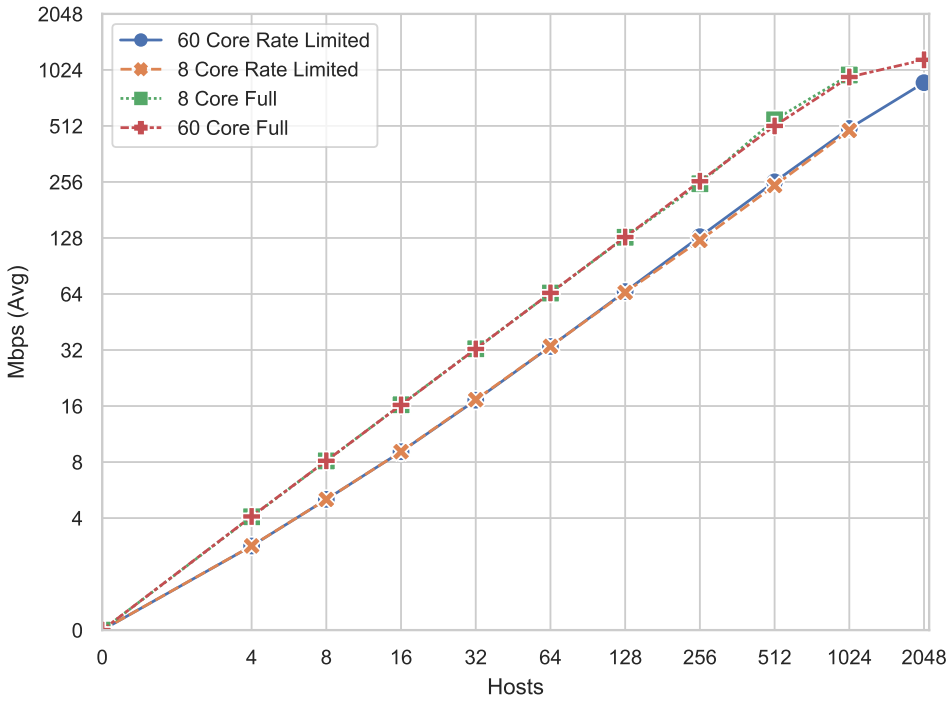

Figure 5.1: Emulator Scalability Test.

limited to 10 Mbps, in the second we remove the limiting constraint but continue to push 10 Mbps from the hosts. We run the experiments on two different machines: A conventional Linux server with 8 cores (2x Intel Xeon E5-2407) and 32Gigabyte (GB) of RAM (`M1`) and a computationally powerful 60-core (4x Intel Xeon E7-4870) machine with 512GB of RAM (`M2`). Both machines run Ubuntu 18.04 with a Linux 4.15.0-34 kernel. We modify the default security configurations (see Appendix 2) to bypass hard limits on the number of file descriptors and process forks.

Figure 5.1 show the results of the scalability benchmark. Aggregate throughput increases exponentially up until 2048 hosts, at which point both machines hit their limit. `M1` ran out of OS resources and terminated prematurely. `M2` was able to complete the experiment, but the throughput diminished. The results indicate that even on machine `M1`, the emulator can scale to around a 1024 hosts, meeting the initial scalability target. A limitation of the experiments is that we scaled up only

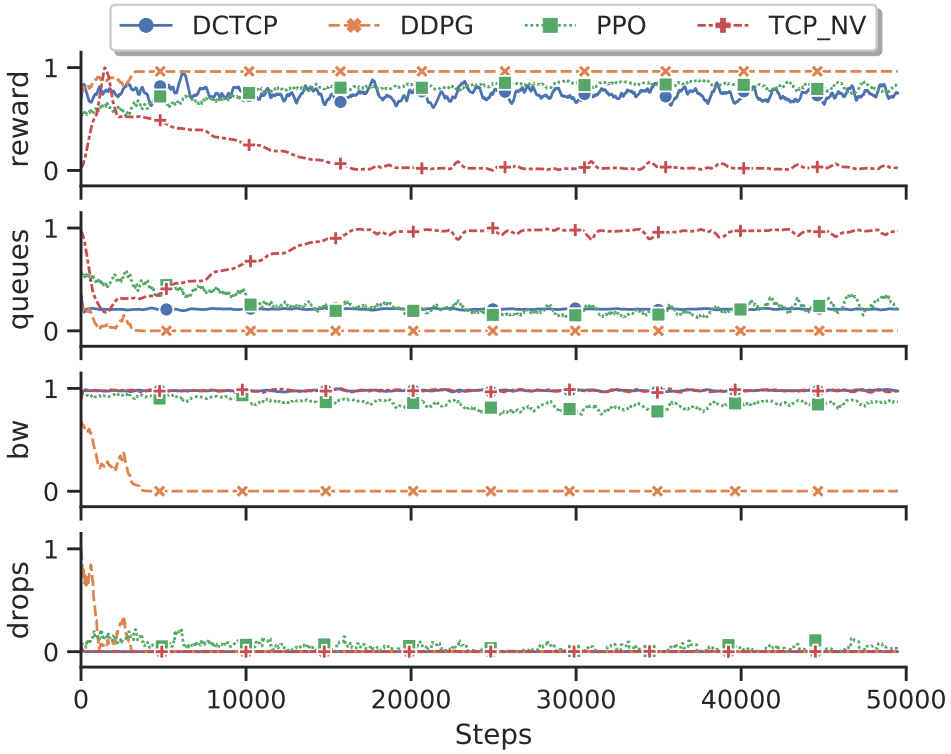

Figure 5.2: Algorithm performance in the 10Mbps TCP dumbbell scenario.

the number of hosts and not switches. The sending rate is also limited to 10 Mbps, an extremely low output rate for modern networks. We intend to investigate the scalability potential of more complex topologies in further experiments.

5.2 Gym Tests

The second set of experiments evaluates the properties of Gyms developed on top of the platform. We run the bufferbloat Gym described in Section 4.2 and benchmark several TCP and RL algorithms with it.

Ray We use the Ray project [68] to deploy RL algorithms. Ray is a distributed, high-performance computing framework developed by the Berkeley RISELab. Ray is specifically intended for production-ready RL by facilitating the end-to-end de-

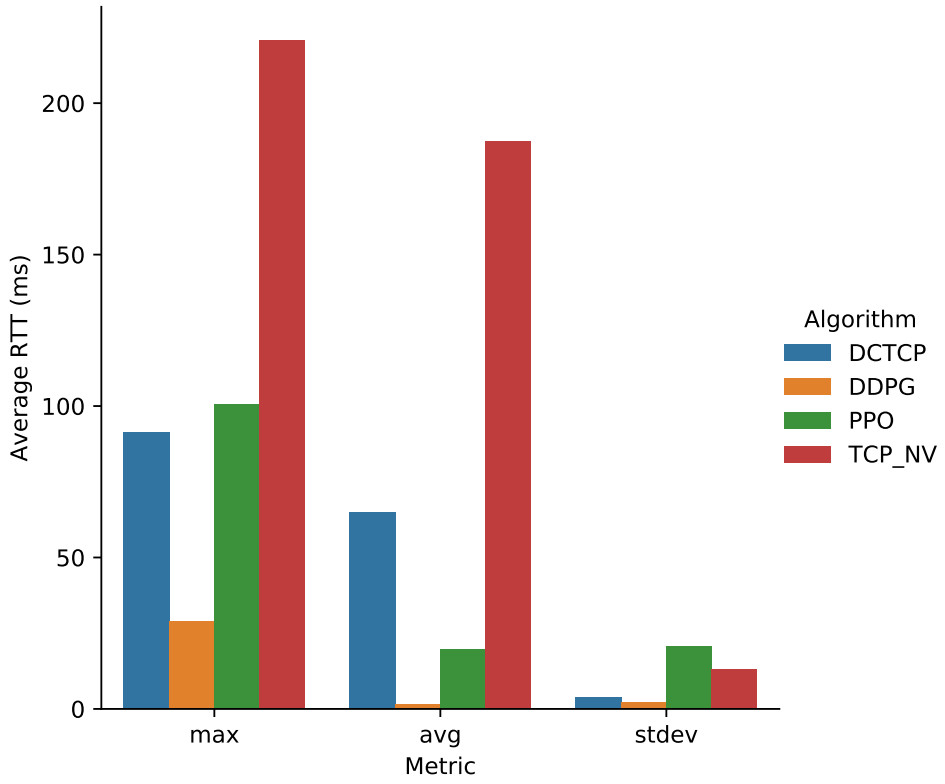

Figure 5.3: Last 10% RTT average in the 10Mbps TCP dumbbell scenario.

sign of RL applications. Using Ray, RL algorithms can first be trained and tuned in large-scale, distributed OpenAI Gym simulations and then, once they are stable and accurate enough, deployed in serving mode onto many machines. This property is attractive for datacenters where a central RL policy may have to be scalable and operate in a fault-tolerant distributed fashion. Another attractive feature of Ray is its extensive and mature RL library. By adopting Ray we can examine state-of-the-art RL algorithms with little to no overhead in the Gym environment.

5.2.1 Setup

We compare the performance of the three previously established deep RL algorithms: REINFORCE, PPO, and DDPG. We run the asynchronous, high-throughput versions of PPO (APPO) and DDPG to generate actions as quickly as possible. As

competitors, we also run scenarios which use DCTCP and TCP NV only and measure the amount of reward these schemes achieve. We use the standardized Ray implementations of the three deep RL algorithms and flatten the collected state into a fully connected neural network architecture (similarly to [18] and [61]).

Although choosing appropriate hyper-parameters can drastically affect algorithm performance [37], we leave tuning to future work. For the TCP algorithms we use the kernel modules provided by the Linux kernel version 4.15.0-34[2]. We configure AQM ECN marking to be very aggressive to make DCTCP perform in an optimal fashion. The full configuration details of all algorithms and hardware specifications of the setup are available in the appendix.

5.2.2 Experiments

We reuse the same dumbbell topology described in Section 5.1. A trivial and fair solution to this scenario is an allocation of 5 Mbps for each host pair. This serves as a baseline comparison and gives empirical evidence on the behavior of a classical CC scheme viewed through the lens of a reinforcement policy.

Dumbbell Tests We run each policy three times for 1,000,000 timesteps using TCP and UDP as the underlying transport protocols. TCP's flow control already acts as a decentralized CC agent, which is a potential factor in confounding the contribution of the policy learned by the RL algorithm. User Datagram Protocol (UDP) does not have flow control and is not reliable, which isolates all congestion management to the RL algorithm. We measure the change in average reward, the bandwidth each host receives, and the queue buildup and packet drops on the network links. We also record the total RTT experienced by each host and truncate the average to the last 10% of measured data to get a snapshot of the "trained" system. Each timestep takes approximately 100 microseconds to several milliseconds, subject to the algorithm in use.

High-Throughput Tests We were interested in the effects of using different bandwidth on the algorithm performance. We reran the dumbbell scenario pushing 1

[2]DCTCP was considered buggy before the 4.9 kernel [12].

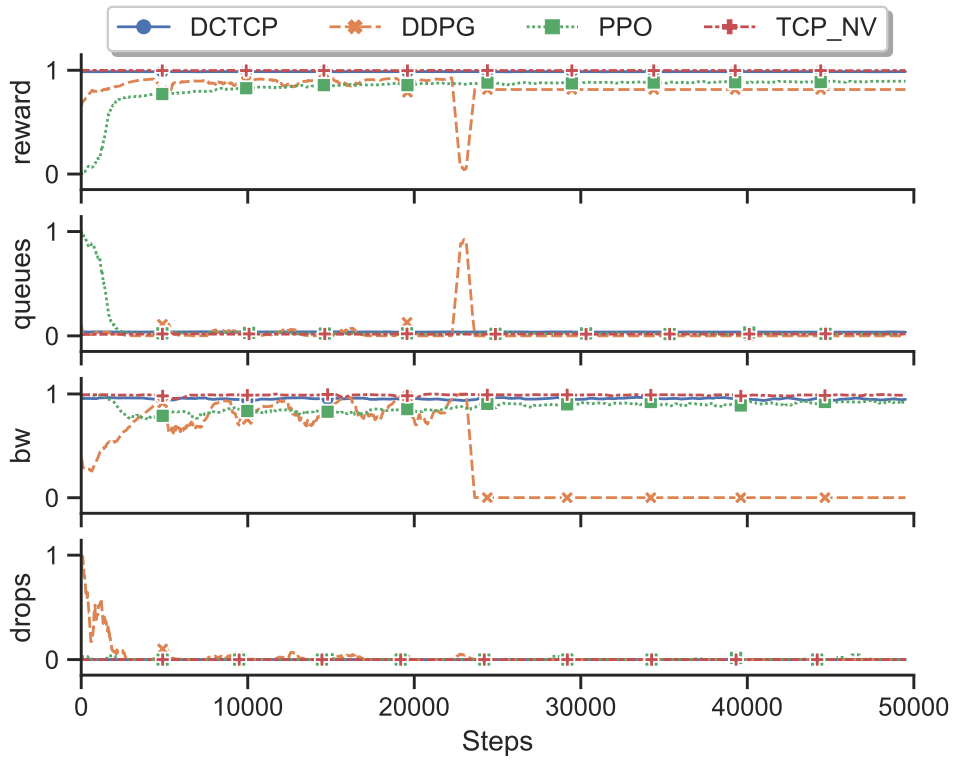

Figure 5.4: Algorithm performance in the TCP 1-GbE dumbbell scenario.

Gbps instead of 10 Mbps. Another major difference is the queue size of each interface. Instead of a 400 KB queue size we use the 4 Megabyte (MB) queue size as specified by Wu et al., 2012 [107]. We ran the test once for 1,000,000 iterations.

Fat-Tree Tests We also investigate APPO's performance in a more complex scenario. We run the algorithms for 5,000,000 timesteps on a 1-GbE fat-tree topology with 16 hosts and 20 switches in total. We generate a Hedera-like [2] hotspot pattern, which causes bottlenecks on several links in the network.

Results

Dumbbell Tests Figures 5.2 and 5.3 plot the results of the TCP settings. We min-max normalize the measurements and apply running mean smoothing to the timestep curve. For clarity we also omit REINFORCE from the graphs. It failed to perform better than PPO, which suggests that the scenario is challenging enough to warrant more complex RL techniques. The algorithms perform equivalent or even better in the UDP scenario. We focus on the optimization over default TCP.

Overall, the graphs demonstrate that the agents are able *learn* with this Gym. Both DDPG and APPO rapidly minimize the queuing in the network. This is also reflected in the RTT measurements, which demonstrate that both DDPG and APPO achieve lower latency than DCTCP. TCP NV severely underperformed. This is not surprising considering it has been optimized for networks with bandwidth higher than 1Gbps. We are assuming that, because TCP NV calculates its expected rate based on the packets in-flight, it overestimates bandwidth in long-running, low-throughput streams and eventually the congestion window overflows.

Unfortunately, the minimization of queuing comes at a high cost of bandwidth for the RL policies. Although the algorithms are rewarded for high action outputs, they remain conservative in order to minimize the penalty of incurring congestion. This implies that the algorithms quickly learned a rewarding, but suboptimal allocation. DDPG in particular is disappointing, it converges to a minimal setting, which gives adequate reward but essentially chokes all hosts in the network. This indicates that tuning of the reward function or DDPG itself is needed.

High-Throughput Tests The result show that the difference in queue size and link bandwidth has a major effect on all the algorithms. TCP NV performs performs much better at the rates it has been originally optimized for and even detects subtle changes in queue size. APPO also achieves great results and even approaches the throughput of TCP NV and DCTCP. This time, DDPG initially performs well but then suddenly collapses in throughput.

Fat-Tree Tests The fat-tree UDP results (Fig. 5.5) show interesting algorithm behavior. APPO optimizes its policy and outperforms both TCP NV and DCTCP in throughput while minimizing queues. The experiment stopped before the algo-

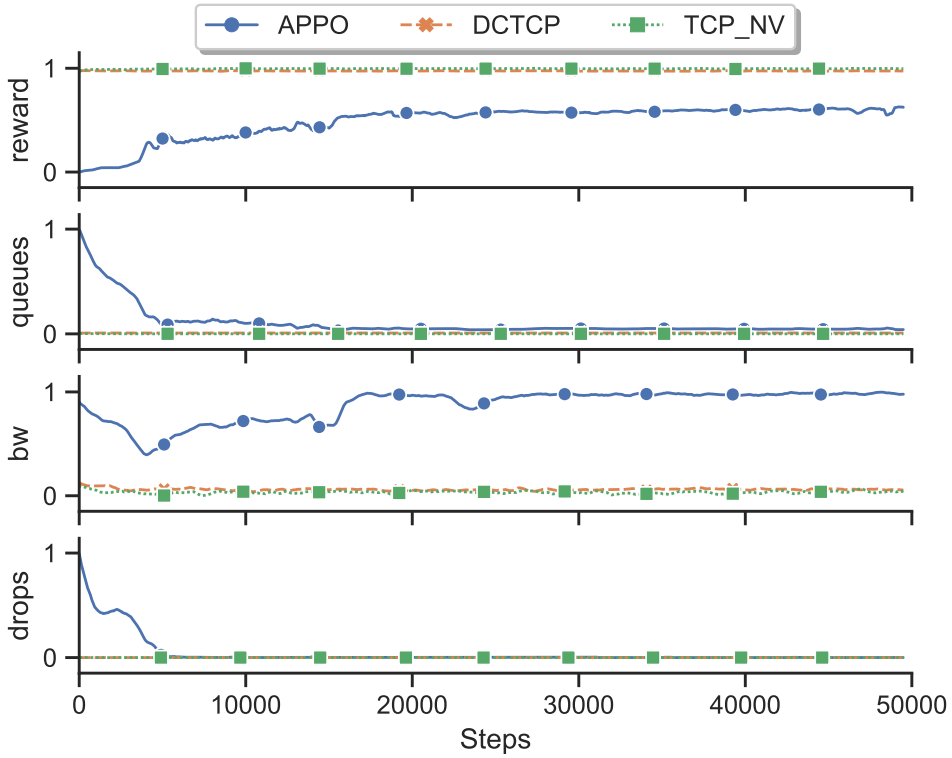

Figure 5.5: Algorithm performance in the UDP fat-tree incast scenario.

rithm was able to fully converge. We are investigating the reason for this behavior, however, overall, these results indicate that there is significant potential for data-driven rate-limiting algorithms in contentious, congestion environments.

Discussion

There is indeed a major difference in performance for different types of bandwidths, likely caused by subtle differences in buffer sizes and marking thresholds. We also believe that at 10 Mbps the token bucket rate-limiting mechanism we use is ill-suited for the type of control we require. A more fine-grained window adjustment or pacing technique such as Carousel [84] could give us better stability and more deterministic algorithm performance.

Interestingly, it seems that using the stochastic PPO in conjunction with TCP

performs the best in both minimizing queue length and maximizing reward. Policies such as PPO are estimated to work better in stochastic environments [37]. DC environments and TCP in particular exhibit stochastic characteristics (e.g., wildly varying throughput or unstable flow behavior), which may explain the good performance at even low step counts. We hypothesize that because of DDPG's deterministic nature and use of the greedy Q-function, it prefers to pick actions which guarantee reward. In the case of a stochastic networking environment where queues are penalized, this mean that setting bandwidth low yields higher reward overall.

For a complete picture of the actual performance, it is likely necessary to measure the average FCT and the 99% latency spikes which measure the stability of the control scheme. We leave these measurements as future work.

Chapter 6

Concluding Remarks

6.1 The Limitations of Data-Driven Networking

Moving towards an autonomously managed DC, raises concerns about the feasibility of such a system. In this section, we discuss some of the challenges.

6.1.1 Telemetry

Effective policy improvement depends on the quality and freshness of the feedback it receives. This requires efficient telemetry which streams in traffic statistics to the central monitor from all the participating systems. Recent advancements in SDN and programmable switches aim to provide flow monitoring statistics without introducing too much overhead [29].

Even though the technology has grown enough to make data collection easier, we do not yet have a good estimate on how much information needs to be collected. In our current model, we only sample few interface statistics. Incorporating more information can improve the policy to make much better predictions. For example, job scheduling or concrete end-host statistics such as FCT or RTT.

The list of scalability concerns is numerous and present when using a central agent. Since we are collecting data at per-interface granularity, the total amount of data collected could potentially blow up. The central controller could become a bottleneck in terms of network throughput as well as processing data on time. Since we are continuously training the model, the amount and frequency of data collec-

tion is also limited by how fast the controller can process the data. The approach we took is to look only at snapshots of the state space in the network topology, which reduces the frequency of data collection at the cost of losing some information. Deployment in a large scale will require multiple controllers co-operating each other. Distributed machine learning is a viable strategy to reduce the cost of model computation. FastPass [76] faces a similar problem as we do. They propose multiple approaches like hierarchical controller architecture. Recent developments on using TPUs[46] for training machine learning models looks promising. Systems such as AuTO [18] attempt to mitigate this limitation by introducing a fast and slow decision path. We consider these limitations to be a fundamental trade-off in the development of a network agent.

6.1.2 Robustness

An ideal data-driven scheme needs to handle all the traffic changes in the network. A wrong prediction can have huge repercussions and could potentially cost millions of dollars. One caveat of RL is that the agent is learning by doing. It is impossible to build an infallible agent that also improves continuously. However, it is possible to design a more robust system and make mispredictions rare or less harmful. For example, failure scenarios such as partitions and outages are a persistent aspect of datacenter operation which can substantially impact the agent. Machine, routers, and link failures in data centers are frequent, but not common enough for the model to learn how to react to those situations.

One approach would be to use the multi-agent architecture proposed in [94]. Instead of training one single agent that can handle all situations, the responsibility is divided among multiple agents each trained for handling separate situations, a form of ensemble learning. Unfortunately, training multiple models is a tedious undertaking and success depends on many factors. Frameworks such as Ray, with ample support for tuning, parallel training, and end-to-end production pipelines, can ease this process.

Before deployment, the agent is likely pre-trained using off-policy learning and observing network traffic. However, even when training for several months, it is not guaranteed that the algorithms will have observed and processed every possible

networking event. Adversial examples in autonomous driving are a good example how to break a model with only small changes to the input environment [16]. This again raises the question what type of input state space matrix is reliable enough to ensure robustness and predictable behavior. The , unlike a routing reinforcement agent, a congestion control agent has only minimal abilities to impact the network (if the minimum guarantees are honored). At worst, it may cause nodes to straggle by limiting their sending rate too heavily.

A second approach to circumvent this problem is to to pre-train models by using imitation learning [115]. Imitation learning algorithms first observe *expert* (e.g., TCP) behavior and infer a baseline policy, which can then be optimized.

6.1.3 Model Definition

One of the major challenges in modeling a machine learning problems is to identify the right abstraction. Finding the best RL model for traffic management is a challenging research problem. We explored several techniques, but the potential number of combinations is nearly infinite. Optimizing for queue length is merely a potential target for a predictive agent. Defining which features, actions, and objective function work in the networking space requires extreme scrutiny. Both the opaqueness of neural networks and the volatility of network communication prohibit full understanding what parameters to tweak and which model to adjust.

Deep learning has shown success, but at a high cost. Until the agent has converged to a successful model, it may take millions of episodes and dozens of hours of computing. AlphaGo Zero [92] played 4.9 million games over three days on specialized hardware (TPUs). Rainbow [38], a more efficient framework, requires approximately 18-million game frames to learn how to play simple games. The time needed to bootstrap such a framework, and its lack of ability to change in volatile environments, can potentially be a impeding factor on its practicability.

Efforts to reduce the amount of required iterations and samples exist, but they may be limited by the sample-inefficient nature of neural networks [60]. We have not tried other, simpler RL approaches such as linear policy gradients or radial basis functions [75]. As baseline, we intend to also implement simpler, easier understandable models and compare their performance to the complex neural net

architectures. One potential, more advanced, method we are considering is a linear policy gradient developed by Rajeswaran, Aravind, et al., [80].

As we noted in the beginning, deep RL has significant problems in reproducibility and success rate [62]. We have yet to observe deterministic success in our experiments. This could be due to many reasons, such as bugs in the implementation or a confusing state and reward model. It is unclear, which change could let to a functional model. For example, the replay buffer can significantly affect the performance of a deep RL model [113].

Another constraint is the domain we are working in. Networks alone are already complex systems, with rippling effects and slow momentum. Changes do not apply immediately and positive effects are often only seen many iterations later. This makes is hard to assign reward properly and may prevent the agent from understanding the environment.

A fruitful thread of research are NUM and network calculus approaches [110]. Both domains attempt to concretely define and standardize utility and properties of congestion control, which can be used by machine learning algorithms as precise objective function.

6.2 Outlook

Deploying a data-driven technique such as RL in the DC remains challenging. The tolerance for error is low and decisions have to be made on less than a millisecond scale. Compared to a TCP algorithm on a local host, the agent has to cope with significant delay in its actions. The chaotic, opaque nature of networks makes appropriately assigning credit for actions almost impossible.

Rewards, actions, and state can be mix-and-matched arbitrarily. There is no indication or theoretical insight if a particular combination will be successful. The fact that traffic has to be evaluated in real-time leads to slow prototyping and agent learning curve. Optimizing a network of a mere 16 hosts already is an arduous task. Each node is an independent actor with unpredictable behavior.

Nonetheless, our initial results are encouraging. In the dumbbell tests, the agents can quickly learn an optimal distribution policy, despite the volatility of the network traffic. PPO even performs comparable to the highly engineered TCP

baselines. We intend to continue the work on our benchmarking tool and focus on improving the emulator performance and stability. We are looking into using meta-information such as job deployments, bandwidth requests by nodes, or traffic traces as additional state information. We also plan to extend the range of reward models, topologies, traffic patterns, and algorithms to truly evaluate the performance of RL policies. Iroko is an open-source project available at https://github.com/dcgym/iroko.

Bibliography

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018. → page 12

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010. → pages 6, 8, 18, 33, 35, 43

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM computer communication review*, volume 40, pages 63–74. ACM, 2010. → pages 5, 6, 17, 35

[4] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of dctcp: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 73–84. ACM, 2011. → page 65

[5] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014. → page 35

[6] V. Arun and H. Balakrishnan. Copa: Congestion control combining objective optimization with window adjustments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association. URL https://www.usenix.org/node/210915. → page 19

[7] S. Athuraliya, S. H. Low, V. H. Li, and Q. Yin. Rem: Active queue management. *IEEE network*, 15(3):48–53, 2001. → pages 18, 35

[8] A. Bechtolsheim, L. Dale, H. Holbrook, and A. Li. Why big data needs big buffer switches. *Arista White Paper*, 2016. → page 16

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0483-2. doi:10.1145/1879141.1879175. URL http://doi.acm.org/10.1145/1879141.1879175. → pages 4, 13, 14

[10] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011. → pages 6, 8, 35

[11] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in Neural Information Processing Systems 6*, pages 671–678. Morgan Kaufmann, 1994. → page 3

[12] L. Brakmo, B. Burkov, G. Leclercq, and M. Mugan. Experiences evaluating dctcp. 2018. → page 42

[13] L. S. Brakmo. tcp: add nv congestion control, July 2015. URL https://lwn.net/Articles/650325/. [Last Accessed: March 26th, 2019]. → page 17

[14] L. S. Brakmo and L. 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. → page 17

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540. → pages 3, 4, 9

[16] T. B. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017. → page 49

[17] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50:20–50:53, Oct. 2016. ISSN 1542-7730. doi:10.1145/3012426.3022184. URL http://doi.acm.org/10.1145/3012426.3022184. → page 5

[18] L. Chen, J. Lingys, K. Chen, and F. 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*, SIGCOMM '18, pages 191–205, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5567-4. doi:10.1145/3230543.3230551. URL http://doi.acm.org/10.1145/3230543.3230551. → pages 3, 35, 42, 48

[19] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, 2007. → page 19

[20] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252. ACM, 2017. → pages 5, 6, 7, 35

[21] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *HotNets*, pages 31–36, 2012. → page 13

[22] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953. → page 14

[23] A. Csoma, B. Sonkoly, L. Csikor, F. Németh, A. Gulyás, W. Tavernier, and S. S. Sahhaf. Escape: extensible service chain prototyping environment using mininet, click, netconf and pox. In *ACM SIGCOMM COMPUTER COMMUNICATION REVIEW*, volume 44, pages 125–126, 2014. ISBN 978-1-4503-2836-4. URL http://dx.doi.org/10.1145/2619239.2631448. → page 32

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. → pages 13, 17

[25] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. Pcc vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018. → pages 3, 5, 8, 19

[26] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016. → page 3

[27] esNet. iperf3: A tcp, udp, and sctp network bandwidth measurement tool. https://github.com/esnet/iperf, Apr. 2019. → page 32

[28] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 184–193. IEEE, 1975. → page 8

[29] N. Feamster and J. Rexford. Why (and how) networks should run themselves. *CoRR*, abs/1710.11583, 2017. URL http://arxiv.org/abs/1710.11583. → page 47

[30] N. Feamster, J. Rexford, and E. Zegura. The road to sdn. *Queue*, 11(12): 20, 2013. → page 7

[31] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, (4):397–413, 1993. → page 18

[32] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019. → page 3

[33] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40, 2011. → page 16

[34] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *NSDI*, pages 1–14, 2015. → page 35

[35] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5567-4. doi:10.1145/3230543.3230555. URL http://doi.acm.org/10.1145/3230543.3230555. → page 3

[36] S. Hemminger. Network emulation with netem. In *linux.conf.au - Australia's National Linux Conference*, pages 18–23, 2005. → page 31

[37] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2018. → pages 3, 42, 46

[38] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. → page 49

[39] C. Inc. Trex low-cost, high-speed stateful traffic generator. https://github.com/cisco-system-traffic-generator/trex-core, Apr. 2019. → page 32

[40] H. Inc. Netperf. https://github.com/HewlettPackard/netperf, Apr. 2019. → page 32

[41] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988. → page 16

[42] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013. → pages 6, 12

[43] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. Eyeq: Practical network performance isolation at the edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, 2013. → page 7

[44] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 539–550. ACM, 2014. → page 6

[45] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High speed networks need proactive congestion control. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 14:1–14:7, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4047-2. doi:10.1145/2834050.2834096. URL http://doi.acm.org/10.1145/2834050.2834096. → pages 5, 6, 9, 34

[46] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017. → page 48

[47] G. Judd. Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 145–157, 2015. → pages 12, 18

[48] I. Juniper Networks. The stateful traffic generator for layer 1 to layer 7. https://github.com/Juniper/warp17, Apr. 2019. → page 32

[49] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-771-4. doi:10.1145/1644893.1644918. URL http://doi.acm.org/10.1145/1644893.1644918. → pages 5, 8, 13, 36

[50] F. P. Kelly, A. K. Maulloo, and D. K. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998. → page 15

[51] J. Kiefer, J. Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952. → page 20

[52] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. → page 67

[53] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11): 1238–1274, 2013. doi:10.1177/0278364913495721. URL https://doi.org/10.1177/0278364913495721. → page 3

[54] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951. → page 27

[55] M. Lanctot, V. F. Zambaldi, A. Gruslys, A. Lazaridou, K. Tuyls, J. Pérolat, D. Silver, and T. Graepel. A unified game-theoretic approach to multiagent reinforcement learning. *CoRR*, abs/1711.00832, 2017. URL http://arxiv.org/abs/1711.00832. → page 3

[56] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010. → pages 4, 31

[57] J. Leike, M. Martic, V. Krakovna, P. A. Ortega, T. Everitt, A. Lefrancq, L. Orseau, and S. Legg. AI safety gridworlds. *CoRR*, abs/1711.09883, 2017. URL http://arxiv.org/abs/1711.09883. → page 37

[58] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, Oct. 1985. ISSN 0018-9340. URL http://dl.acm.org/citation.cfm?id=4492.4495. → page 14

[59] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971. → pages 24, 26

[60] R. Livni, S. Shalev-Shwartz, and O. Shamir. On the computational efficiency of training neural networks. In *Advances in neural information processing systems*, pages 855–863, 2014. → page 49

[61] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 50–56, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4661-0. doi:10.1145/3005745.3005750. URL http://doi.acm.org/10.1145/3005745.3005750. → pages 3, 42

[62] G. Marcus. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018. → page 50

[63] V. N. Marivate. *Improved empirical methods in reinforcement-learning evaluation*. PhD thesis, Rutgers University-Graduate School-New Brunswick, 2015. → page 3

[64] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407. ACM, 2018. → page 12

[65] M. Mirza, J. Sommers, P. Barford, and X. Zhu. A machine learning approach to tcp throughput prediction. *IEEE/ACM Transactions on Networking*, 18(4):1026–1039, 2010. → page 8

[66] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015. → page 17

[67] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602. → page 22

[68] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018. → page 40

[69] J. Moy. Ospf version 2. RFC 2178, RFC Editor, July 1997. URL http://www.rfc-editor.org/rfc/rfc2178.txt. http://www.rfc-editor.org/rfc/rfc2178.txt. → page 15

[70] A. Mushtaq, R. Mittal, J. McCauley, M. Alizadeh, S. Ratnasamy, and S. Shenker. Datacenter congestion control: Identifying what is essential and making it practical. → page 6

[71] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 30–43. ACM, 2018. → pages 8, 9

[72] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4653-5. doi:10.1145/3098822.3098829. URL http://doi.acm.org/10.1145/3098822.3098829. → page 3

[73] M. Noormohammadpour and C. S. Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525. → pages 11, 13, 15

[74] A. Ousterhout, J. Perry, H. Balakrishnan, and P. Lapukhov. Flexplane: an experimentation platform for resource management in datacenters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 438–451, 2017. → page 32

[75] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural computation*, 3(2):246–257, 1991. → page 49

[76] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015. → pages 5, 6, 7, 9, 18, 35, 48

[77] M. Peuster, H. Karl, and S. Van Rossem. Medicine: Rapid prototyping of production-ready network services in multi-pop environments. *arXiv preprint arXiv:1606.05995*, 2016. → page 32

[78] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. URL http://dl.acm.org/citation.cfm?id=2789770.2789779. → page 31

[79] M. Raghu, A. Irpan, J. Andreas, R. Kleinberg, Q. V. Le, and J. M. Kleinberg. Can deep reinforcement learning solve erdos-selfridge-spencer games? *CoRR*, abs/1711.02301, 2017. URL http://arxiv.org/abs/1711.02301. → page 3

[80] A. Rajeswaran, K. Lowrey, E. V. Todorov, and S. M. Kakade. Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, pages 6553–6564, 2017. → page 50

[81] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 407–418. ACM, 2014. → pages 3, 8

[82] G. F. Riley and T. R. Henderson. The ns-3 network simulator. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010. → page 32

[83] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM. ISBN

978-1-4503-3542-3. doi:10.1145/2785956.2787472. URL http://doi.acm.org/10.1145/2785956.2787472. → pages 5, 8, 12, 13, 35, 36

[84] A. Saeed, N. Dukkipati, V. Valancius, C. Contavalli, A. Vahdat, et al. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 404–417. ACM, 2017. → page 45

[85] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an ip services protocol. RFC 3549, RFC Editor, July 2003. → page 35

[86] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *CoRR*, abs/1704.02532, 2017. → page 3

[87] M. Schapira and K. Winstein. Congestion-control throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 122–128, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5569-8. doi:10.1145/3152434.3152446. URL http://doi.acm.org/10.1145/3152434.3152446. → pages 8, 9

[88] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel. Trust region policy optimization. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1889–1897. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045319. → page 27

[89] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015. → page 67

[90] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. → page 27

[91] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR. → page 25

[92] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017. → page 49

[93] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 479–490, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi:10.1145/2619239.2626324. URL http://doi.acm.org/10.1145/2619239.2626324. → pages 8, 37

[94] C. Streiffer, H. Chen, T. Benson, and A. Kadav. Deepconfig: Automating data center network topologies management with machine learning. *CoRR*, abs/1712.03890, 2017. URL http://arxiv.org/abs/1712.03890. → pages 3, 28, 48

[95] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 262193981. → pages 3, 20, 22, 23

[96] G. Tesauro, R. Das, H. Chan, J. Kephart, D. Levine, F. Rawson, and C. Lefurgy. Managing power consumption and performance of computing systems using reinforcement learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1497–1504. Curran Associates, Inc., 2008. → page 3

[97] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, RFC Editor, Nov. 2000. URL http://www.rfc-editor.org/rfc/rfc2991.txt. http://www.rfc-editor.org/rfc/rfc2991.txt. → page 15

[98] S. Thomas, R. McGuinness, G. M. Voelker, and G. Porter. Dark packets and the end of network scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 1–14. ACM, 2018. → page 6

[99] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. → page 3

[100] E. Track, N. Forbes, and G. Strawn. The end of moore's law. *Computing in Science & Engineering*, 19(2):4, 2017. → page 6

[101] udhos. goben. https://github.com/udhos/goben, Apr. 2019. → page 33

[102] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 185–191. ACM, 2017. → page 3

[103] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and . . . , 2008. → page 32

[104] P. Wette, M. Draxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014. → page 32

[105] S. Whiteson, B. Tanner, M. E. Taylor, and P. Stone. Protecting against evaluation overfitting in empirical reinforcement learning. *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 120–127, 2011. → page 3

[106] K. Winstein and H. Balakrishnan. Tcp ex machina: Computer-generated congestion control. 43(4):123–134, 2013. → pages 8, 19

[107] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang. Tuning ecn for data center networks. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 25–36. ACM, 2012. → pages 18, 43

[108] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. Pantheon: the training ground for internet congestion-control research. In *USENIX Annual Technical Conference*, pages 731–743. USENIX Association, 2018. → page 4

[109] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 509–522. ACM, 2015. → page 3

[110] D. Zarchy, R. Mittal, M. Schapira, and S. Shenker. An axiomatic approach to congestion control. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 115–121. ACM, 2017. → page 50

[111] C. Zhang, O. Vinyals, R. Munos, and S. Bengio. A study on overfitting in deep reinforcement learning. *CoRR*, abs/1804.06893, 2018. → page 3

[112] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85. ACM, 2017. → pages 16, 35

[113] S. Zhang and R. S. Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017. → page 50

[114] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone with valiant load-balancing. In *International Workshop on Quality of Service*, pages 178–192. Springer, 2005. → page 15

[115] Y. Zhu, Z. Wang, J. Merel, A. Rusu, T. Erez, S. Cabi, S. Tunyasuvunakool, J. Kramár, R. Hadsell, N. de Freitas, et al. Reinforcement and imitation learning for diverse visuomotor skills. *arXiv preprint arXiv:1802.09564*, 2018. → page 49

.1 Appendix

Calculating the ECN marking threshold

We calculate the DCTCP marking threshold based on the Bandwidth Delay Product (BDP). We follow the specifications outlined by Alizadeh, et al., 2017 [4]. Our measured system RTT is approximately a 100 microseconds, which translates to 0.0001 seconds. The marking threshold is set to 17% of the BDP:

$$ecn_{dctcp} = \underbrace{bw_{max} * 0.0001}_{\text{bandwidth delay product}} * 0.17 \tag{1}$$

Parameter	Dumbbell 10Mbit	Dumbbell 1Gbit	Fat-Tree 1Gbit
Network Configuration			
Queue Size	400 KB	4MB	4MB
Hosts	4	4	16
Switches	2	2	20
Switch Interfaces	6	6	80
State Matrix	6×2	6×2	80×2
Action Matrix	1×4	1×4	1×16
RED ECN Thresholds			
Limit	400 KB	4 MB	4 MB
Bandwidth	10Mbit	10Gbit	10Gbit
Avg Pkt Size	1500	1500	1500
Min Marking	3000	17000	17000
Max Marking	100000	1000000	1000000
Burst	23	229	229
Marking Prob.	0.1	0.1	0.1
Environment			
Operating System	Ubuntu 18.04.2 LTS		
Kernel	4.15.0-46		
Ray Version	0.7.0-dev2		
Mininet Version	2.3.0d5		
Python Version	2.7.15rc1		

Table 1: Experiment-specific configurations.

Listing 2 Security Configuration for scalability tests.

```
echo "* soft nofile 1048576" >> /etc/security/limits.conf
echo "* hard nofile 1048576" >> /etc/security/limits.conf
echo "* soft nproc unlimited" >> /etc/security/limits.conf
echo "* hard nproc unlimited" >> /etc/security/limits.conf
echo "* soft stack unlimited" >> /etc/security/limits.conf
echo "* hard stack unlimited" >> /etc/security/limits.conf
echo "kernel.threads-max = 2091845" >> /etc/sysctl.conf
echo "kernel.pty.max = 210000" >> /etc/sysctl.conf
echo "DefaultTasksMax=infinity" >> /etc/systemd/system.conf
echo "UserTasksMax=infinity" >> /etc/systemd/logind.conf
sysctl -p
systemctl daemon-reload
systemctl daemon-reexec
```

Listing 3 The rate control function instrumented at each host.

```
void ctrl_set_bw(void *data) {
    uint64_t tx_rate, factor;
    ctrl_pckt *pkt;

    /* Calculate the burst factor based
     on the Linux timer HZ (100) */
    factor = 10e6 / (100 * 8);
    /* Convert payload to control packet structure */
    pkt = (ctrl_pckt *) data;
    tx_rate = pkt->tx_rate / 8;
    /* Set tbf qdisc burst and rate */
    rtnl_qdisc_tbf_set_limit(fq_qdisc, tx_rate);
    rtnl_qdisc_tbf_set_rate(fq_qdisc, tx_rate, factor, 0);
    /* Update the qdisc on the interface */
    rtnl_qdisc_add(qdisc_sock, fq_qdisc, NLM_F_REPLACE);
}
```

Hyperparameter	Value
DDPG	
θ	0.15
σ	0.2
Target network update frequency	Every update
τ	10^{-3}
Use Prioritized Replay Buffer	True
α	0.6
β	0.4
temporal difference ε	10^{-6}
Optimizer	Adam [52]
Actor Learning rate	10^{-4}
Critic Learning rate	10^{-3}
Weight decay coefficient	10^{-6}
Critic Loss function	Square loss
PPO	
Use GAE [89]	True
GAE Lamda	1.0
KL coefficient	0.2
Train batch size	1000 (4000 for fat tree)
Mini batch size	128
Optimizer	Adam [52]
Learning rate	10^{-5}
Value function coefficient	1.0
Entropy Coefficient	0.0
Clip parameter	0.3
Target Value for KL	0.01
REINFORCE	
Learning rate	10^{-4}
Optimizer	Adam [52]

Table 2: Configurations for all algorithms.