PRACTICAL MACHINE LEARNING FOR SEQUENTIAL DECISION PROBLEMS ON THE INTERNET

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Francis Y. Yan
June 2020

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Keith Winstein, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Philip Levis, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Emma Brunskill

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

Abstract

Networking algorithms often perform sequential decision making under uncertainty: They observe a network path and decide, e.g., how many packets to send or what to put in them. The Internet presents a particularly challenging setting: performance varies across several orders of magnitude and changes with time, control is decentralized, each node observes only a noisy sliver of the overall system, and accurate simulators do not exist.

Despite the recent progress in applying machine learning (ML) to networking research, sequential decision problems on the Internet continue to rely on hand-designed algorithms. Slow adoption of ML in these scenarios can be attributed to the requirement that control algorithms be not just performant, but also *practical*: robust, generalizable, real-time, and resource-efficient. Lack of research platforms for studying ML approaches in the real world exacerbates the problem.

This dissertation presents the platforms and algorithms we developed to achieve practical ML in the context of video streaming and congestion control. We describe Puffer, a free, publicly accessible website that live-streams television channels and operates as a randomized experiment of adaptive bitrate (ABR) algorithms. As of June 2020, Puffer has attracted 120,000 real users and streamed 60 years of video across the Internet. Using Puffer, we developed an ML-based ABR algorithm, Fugu, that robustly outperformed existing schemes by learning *in situ*, on real data from its actual deployment environment.

Next, we describe Pantheon, a community "training ground" for Internet congestion-control research. It allows network researchers to benefit from and contribute to a common set of benchmark algorithms, a shared evaluation platform, and a public archive of results. Pantheon has assisted four algorithms from other research groups in publishing at NSDI

2018, ICML 2019, and SIGCOMM 2020. It also enabled our own ML-based congestion-control algorithm, Indigo, which was trained to imitate expert congestion-control algorithms we created in emulation and achieved good performance over the real Internet.

Previously Published Material

Chapter 2 revises a previous publication [126]: Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, February 2020.

Chapter 3 revises a previous publication [127]: Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (ATC 18)*, Boston, MA, July 2018.

Acknowledgments

I feel obliged to begin with the COVID-19 pandemic, a once-in-a-generation disaster that spread rapidly to the whole world. Thank you to the healthcare workers on the front line and everyone who is making sacrifices. This pandemic has reshaped the way we are living: With the shelter-in-place order remaining in effect, I have to defend my thesis remotely and attend a virtual commencement from home.

Although I would miss the privilege of being hooded by my primary advisor, Keith Winstein, at least I have had enough honor to be his student for the past five years. His research style of doing impactful work and philosophy of building systems have a profound influence on me. Keith is concerned not only with my accomplishments but also with my well-being and career development, and he always genuinely offers to help with my best interests at heart. He is a role model whom I wish to learn from for a lifetime.

My co-advisor, Philip Levis, has given me invaluable guidance and advice on research. His questions are often so inspirational and urge me to understand the problem at a deeper level. I have greatly benefited from his critical thinking and logical presentation of ideas. I put Phil's name in my essay when I applied to Stanford, and I'm lucky to have my wish fulfilled by spending unforgettable years working with him.

Thank you to my committee members, Keith Winstein, Philip Levis, Emma Brunskill, Percy Liang, and Bernd Girod, who have evaluated my work from networking and machine learning perspectives. I especially appreciate Emma for her feedback on reinforcement learning and suggestions on the dissertation title.

I enjoyed assisting John Ousterhout and David Mazières in teaching operating systems, and I'd like to thank Sachin Katti, Nick McKeown, and Dan Boneh for the knowledge they taught me.

During my Ph.D., I completed two fantastic internships and have had the fortune to be mentored by Yuchung Cheng at Google, and Ganesh Ananthanarayanan and Yuanchao Shu at Microsoft Research. They are essentially my additional Ph.D. advisors and have all given me tremendous help in different stages.

I'm grateful to my collaborators, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Emily Marx, Jestin Ma, Deepti Raghavan, and Riad Wahby, for the great help on Puffer and Pantheon. My thanks to Tim Shi and Danfei Xu, who kindly answered many of my questions on machine learning.

To my labmates in the Systems and Networking Research Group and Stanford Information Networks Group: Hudson Ayers, Holly Chiang, John Emmons, Sadjad Fouladi, Greg Hill, Luke Hsiao, Raejoon Jung, Kevin Kiningham, Dima Kogan, Omid Mashayekhi, Hang Qu, Deepti Raghavan, Laurynas Riliskis, Kexin Rong, Chinmayee Shah, Riad Wahby, and Judson Wilson, and to many others whom I have interacted with in the Gates building. Thank you for our time together.

Thank you to my friends in the Bay Area for the companionship, especially to Mingtian Yin, for our decade-long friendship and numerous wonderful weekends we spent together.

Special thanks to Andrew Chi-Chih Yao for founding the "Yao Class" at Tsinghua University, where I studied as an undergraduate. I wouldn't have had the opportunity to come to Stanford without his letter of recommendation. I will always appreciate Dongxiao Yu, who supervised my first publication and offered generous help during the graduate school application.

I will be eternally indebted to my parents, Xueqing Yan and Xiurong Wang, for their endless love and for giving me everything they have. There are no words to express my gratitude to them; in many senses, they made me who I am today. In Chinese culture, it's rare and awkward to say this to parents, but in a different language now: I love you, Mom and Dad.

I'm the luckiest to marry Andi Yang and have her in my life. We fell in love at Stanford, and the moment she said yes has been etched in my memory. Through my ups and downs with her around, the joy is doubled and the sorrow is halved. Thank you for always being there for me with your unconditional love and encouragement.

To my beloved wife and parents,
for their unwavering love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Sequential decision making under uncertainty has long been a fundamental problem in control theory, operations research, robotics, and artificial intelligence. It studies how an intelligent agent interacts with a stochastic environment and makes a sequence of decisions. The environment generally produces reward signals indicating the quality of decisions, and the agent's goal is to maximize the total reward received in the long run.

Networking algorithms running on the Internet often perform sequential decision making under uncertainty as well. They repeatedly observe a network path and decide, e.g., how many packets to send or what to put in them. Adaptive bitrate (ABR) algorithms, for example, observe the Internet and decide a video bitrate to transmit each time so as to maximize the total quality of experience (QoE). Congestion-control schemes observe congestion signals and decide the best congestion window size to use next.

A distinguishing feature and an inherent challenge of sequential decision problems is that each decision of the agent may influence its later observations of the environment. Therefore, maximizing the cumulative reward requires the agent to plan ahead and take into account the long-term consequences of its decisions in an uncertain environment.

Moreover, the Internet—as a unique environment—presents many challenges: common characteristics of the Internet (e.g., bandwidth and RTT) span several orders of magnitude and change with time, individual nodes observe only a sliver of the overall system, behavior is heavy-tailed and extremely noisy, and accurately simulating the Internet is beyond current capabilities because of its complexity and diversity.

Traditionally, control algorithms for real networks have been handcrafted, relying on human experts' knowledge and instincts during the design. This approach worked well when the Internet was simple in the past but started to fall apart as the Internet grew much larger and more complicated. For instance, these algorithms modeled the network in an unrealistic way or had too many parameters to tune.

In recent years, machine learning (ML) approaches, especially those based on reinforcement learning, have made rapid advances in solving sequential decision problems. While human experts are overwhelmed by the wild Internet and its massive volume of data, ML excels at automatically extracting algorithms by interacting with an environment and finding patterns in seas of data. It is a natural fit for this problem domain and has produced promising results.

Even with the progress in applying ML to create networking algorithms, hand-designed decision-makers still dominate the Internet. Taking video streaming and congestion control as examples again: simple heuristic algorithms remain prevalent in ABR, and TCP Cubic and BBR continue to prevail despite the proposal of many ML-based congestion-control schemes. The adoption of ML in these scenarios is slow because real networked systems require the control algorithms to be not only performant but also *practical*: performance needs to generalize from training to the real world and be robust over the diverse Internet; control algorithms must run in real time given a limited resource budget on the network device; preferably, the training is also stable and sample efficient, and algorithm behavior is interpretable.

Failing to satisfy practical requirements in the real world has become a significant hurdle for researchers hoping to deploy learning-based algorithms. Regardless of the performance in simulation or on small testbeds, a control algorithm needs to be validated on the real Internet to prove its practicality. Sometimes the training of control algorithms may also need a realistic environment, such as a real networked system, to learn from the experience of interaction. However, existing real-world systems are often designed for production purposes and are not intended for researchers to train or validate their algorithms. Lack of research platforms has exacerbated the slow adoption of ML-based control algorithms on the Internet.

1.1 Background on sequential decision problems

In this section, we provide background on sequential decision problems, a class of problems that commonly arise in networking research and many other research areas. We defer specific background and related work in networking research to their respective chapters (Chapter 2 for video streaming and Chapter 3 for congestion control).

1.1.1 Sequential decision making under uncertainty

Sequential decision making under uncertainty has a long and rich literature in control theory, operations research, robotics, and artificial intelligence. It studies how a decision-maker, or *agent*, achieves a goal by making a series of decisions in an uncertain world, or *environment*.

The agent, for example, can be an autonomous vehicle trying to reach a specific destination. Even if the vehicle is informed of a route, it will have to handle other real-world situations—traffic lights and road signs, other traffic and pedestrians, obstacles on the road, etc. The agent observes an environment *state* each time, such as a traffic light recognized by its sensors, and takes an *action*, such as going straight or making a turn, speeding up or slowing down. The next observation of the world may vary across different actions—the vehicle will encounter a different situation after going straight compared with making a turn. From the perspective of the agent, the environment changes in response to its decisions. The fact that each action may influence the agent's later observations of the environment is a key feature of sequential decision problems.

In contrast to making a single decision or isolated decisions as in classification or regression problems, sequential decision making is generally more challenging as it requires the agent to take into account the longer-term consequences of each decision and to plan ahead. An action that maximizes the cumulative return in the long run may not optimize the immediate return, e.g., if an autonomous vehicle tries to arrive at a destination as soon as possible, a small detour now might save more time by avoiding a later traffic jam.

A fundamental framework to formulate sequential decision making is the Markov decision process (MDP) [12]. MDP assumes that each environment state is fully observable, and environment models, namely state transitions, satisfy the Markov property and are known to the agent. Under these assumptions, sequential decision making is also known as *optimal control*. The goal of MDP is to find a decision-maker that maximizes the long-term expected return, or sum of *rewards*; each reward is received after transitioning from one state to another by taking an action. A well-known and efficient solution to solving MDP is dynamic programming [11].

By comparison, when the environment state is *partially* observable, meaning that the agent's observation of the world only partly captures the real underlying state, the MDP framework is extended to Partially Observable MDP (POMDP) [108], which is often computationally intractable to solve exactly. In other situations where the environment models are unknown, the agent is required to learn through interaction how the stochastic environment changes over time and responds to each action. These sequential decision problems with partially observable state or unknown models are the primary focus of reinforcement learning (RL) [112], the third machine learning paradigm in addition to supervised learning and unsupervised learning. Nevertheless, MDP remains an essential element of RL, and generally speaking, optimal control and dynamic programming are considered part of RL.

This dissertation draws a closer connection to RL when designing control algorithms to perform sequential decision making on the Internet. Next, we briefly introduce modern RL that has made rapid advances in recent years.

1.1.2 Reinforcement learning

Modern RL dramatically benefits from deep learning and the massive increase in computing power. An important early work of modern RL is learning to play Atari games at a superhuman level [73]. Although its influence in the ML community lies more on its demonstration of learning from a high-dimensional state space—directly from raw image pixels, we focus here on a high-level takeaway: RL algorithms are often driven by rewards and learned through trial-and-error. Given an image in an Atari game, the RL agent tries out an action and receives a corresponding score in the game. The score serves as a reward indicating the quality of the previous action(s). It then alters its neural network parameters that encode the control algorithm, or *policy*, to favor "high-score" actions.

Model-free RL. This trial-and-error paradigm forms the main body of modern RL, known as *model-free RL*. Unless expressly stated otherwise, "RL" refers to "model-free RL" in this dissertation. As its name suggests, model-free RL does not maintain a model of the environment; it is solely reward-driven and learns an end-to-end mapping from states to actions. Depending on whether it is representing a value function [11], an explicit policy, or both, model-free RL roughly consists of three classes, based on value functions [11, 98, 122], policy search [30, 113], and a hybrid—actor-critic—approach [59, 72], respectively. However, we are not concerned with the difference between various model-free RL approaches in this dissertation. For example, Pensieve [70] adopts an actor-critic approach to learning an ABR algorithm, and we simply refer to it as a typical, model-free RL algorithm.

Model-based RL. As opposed to model-free RL that learns from rewards and trial-and-error, *model-based RL* [78] learns an environment model first. In other words, given a state and an action, model-based RL learns to predict the next possible state and emitted reward based on past observations, often via supervised learning. Then by simulating the environment model, a model-based RL agent can plan with classical optimal control techniques, such as dynamic programming and model predictive control (MPC) [19]. Model-based RL is more sample efficient (i.e., fewer samples are required during training) than model-free methods [29] and plays an important role in today's RL research. However, a drawback of model-based RL is that modeling the environment introduces an extra layer of error, although MPC somewhat mitigates the model error. Our ABR algorithm, Fugu (§2.3), belongs exactly to the family of model-based RL.

Imitation learning. Another family of RL methods, *imitation learning*, studies the problem of extracting a policy from "expert demonstration," i.e., a sequence of states and the corresponding (optimal) actions output by an expert policy. The simplest form of imitation learning is *behavioral cloning* [7, 89], which learns to map each state to the expert action using supervised learning. It is proved sample efficient [49] but only feasible when the expert demonstration is not expensive to obtain. Although imitation learning also includes other methods such as inverse RL [83, 99], which infers an unknown reward function of an agent from its behavior, they are not the focus of this dissertation. Our congestion-control algorithm, Indigo (§3.7), uses a specific behavioral-cloning method called DAgger [97], and we simply refer to it as an imitation-learning algorithm.

Challenges of real-world RL. RL has reached a superhuman level in video games [73], chess [105], and many simulated environments [18, 66]. However, the employment of RL in the real world lags behind RL research due to a variety of practical challenges [36]:

1. Real systems are often too complex to simulate accurately and do not have a model to capture dynamics with accuracy. Therefore, previous success stories of RL, which mostly rely on environments we can properly simulate, may not generalize to real life if good simulators do not exist for these systems. This roadblock is known as *simulation-to-reality gap* [16], describing the discrepancy between the simulated and actual environments. Reducing the gap is a prerequisite for creating deployable RL algorithms.

2. Acquiring data on real-world systems is often expensive and intrusive to the production service, requiring RL to learn from limited training data. Access only to limited samples calls for sample-efficient RL algorithms [78, 97].

3. Real systems often impose safety constraints on RL agents. The consequences of a destructive action are limited in simulation but may destroy the systems in real life, e.g., violating safety constraints may crash a self-driving car. The safety of learning in an environment has drawn more attention recently [2, 27].

4. Real systems may exhibit noisier behavior and more severe partial observability. Although recent work has proposed approaches to addressing these issues, e.g., by incorporating state history [73] and devising policies robust to system noise [88, 95, 117], their effectiveness remains to be verified in the real world.

5. Real systems run in wall-clock time, rather than in simulation time, and often have delays in providing feedback while executing a policy; parallel training as in simulation may not be feasible anymore in the real world. Given these constraints, the training still has to finish within a reasonable time, and each inference must also be completed without missing the deadline for making the next control decision.

6. Real systems have a limited computational budget and prefer resource-efficient algorithms to reduce operational costs. E.g., the neural networks used in RL algorithms might significantly reduce the total number of concurrently running applications, so the performance gain must outweigh the computational overhead to provide an incentive for adopting RL.

1.2 Thesis

This dissertation proposes that ML may improve sequential decision making on the Internet, but should also be *practical*—robust, generalizable, real-time, and resource-efficient—to be adopted in the real world. In addition, open research platforms providing a realistic setting for training and validation may facilitate the development of practical ML algorithms.

Stated precisely, the thesis of this dissertation is: *Machine learning improves sequential decision making on the Internet and is practical.* We demonstrate the thesis using real-world platforms and algorithms we created for video streaming and congestion control.

The main body of this dissertation (Chapters 2 and 3) is organized as shown in Table 1.1, and we conclude with a discussion in Chapter 4.

	Problem domain	Research platform	ML algorithm
Chapter 2	video streaming (§2.1)	Puffer (§2.2)	Fugu (§2.3)
Chapter 3	congestion control (§3.2)	Pantheon (§3.3)	Indigo (§3.7)

Table 1.1: Outline of this dissertation's main body

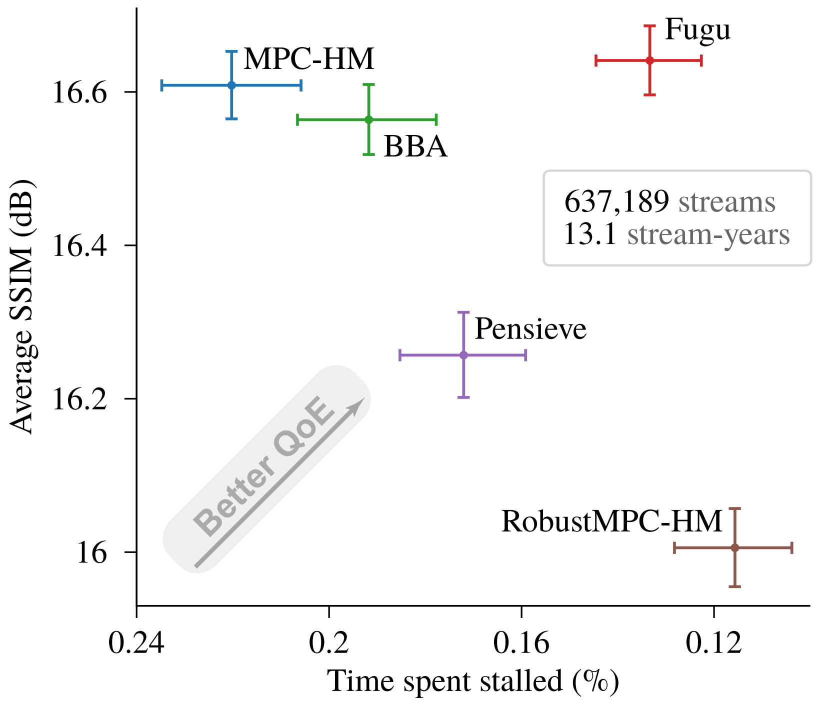
1.3 Summary of results

Puffer and Fugu (Chapter 2): To investigate video streaming and practical ML-based ABR schemes in the real world, we built Puffer (§2.2), a publicly accessible website that live-streams television channels for free. Puffer operates as a randomized controlled trial; sessions are randomly assigned to one of a set of ABR schemes, and users are blinded to

Results of primary Puffer experiment (Jan. 26–Aug. 7 & Aug. 30–Oct. 16, 2019)

Algorithm	Time stalled (lower is better)	Mean SSIM (higher is better)	SSIM variation (lower is better)	Mean duration (time on site)
Fugu	0.13%	16.64 dB	0.74 dB	33.6 min
MPC-HM [128]	0.22%	16.61 dB	0.79 dB	30.8 min
BBA [51]	0.19%	16.56 dB	1.11 dB	32.1 min
Pensieve [70]	0.17%	16.26 dB	1.05 dB	31.6 min
RobustMPC-HM	0.12%	16.01 dB	0.98 dB	31.0 min

(See the caption of Table 2.1)



(Adapted from the top panel of Figure 2.11)

algorithm assignment. As of June 2020, Puffer has attracted more than 120,000 real users and streamed 60 years of video across the Internet.

In an eight-month randomized experiment we conducted in 2019 (details in §2.4), we found that it is difficult for sophisticated or machine-learned control schemes to outperform a "simple" scheme (buffer-based control), notwithstanding good performance in network emulators or simulators. We performed a statistical analysis and found that reliably measuring ABR performance within 20% precision may require at least 2 years of video data per scheme. The heavy-tailed nature of network and user behavior, as well as the challenges of emulating diverse Internet paths during training, present obstacles for learned algorithms in this real-world setting.

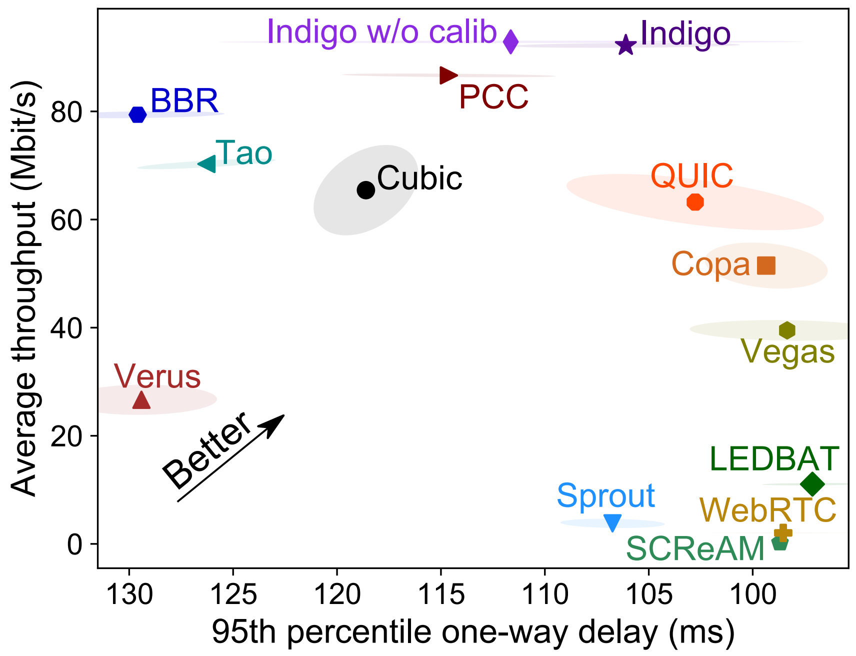
We then developed an ABR algorithm, Fugu (§2.3), that robustly outperformed other schemes, by leveraging data from its deployment and limiting the scope of machine learning only to making predictions that can be checked soon after. The system uses supervised learning *in situ*, with data from the real deployment environment, to train a probabilistic predictor of upcoming chunk transmission times. This module then informs a classical control policy (model predictive control). With this design, Fugu belongs to a family of approaches known as model-based reinforcement learning [78].

To support further investigation, we are publishing an archive of data and results every day and opening our ongoing study to the community. We welcome other researchers to use this platform to develop and validate new algorithms for bitrate selection, network prediction, and congestion control.

Pantheon and Indigo (Chapter 3): To study Internet transport protocols and practical ML-based congestion-control schemes, we built Pantheon (§3.3), a community "training ground" and benchmark platform for Internet congestion-control research. Pantheon consists of a software library containing a reference set of transport protocols and congestion-control algorithms, a diverse testbed of network nodes on wireless and wired networks around the world, and a continuous-testing system that regularly evaluates the Pantheon protocols over the real Internet.

We used Pantheon to generate calibrated network emulators (§3.5) that capture the diverse performance of real Internet paths. These enable reproducible and rapid experiments that closely approximate real-world results.

As a community resource, Pantheon has assisted four algorithms from independent research groups in publishing at NSDI 2018 [6, 33], ICML 2019 [53], and SIGCOMM 2020 [64] (§3.6). They deployed a series of prototypes on Pantheon and used Pantheon's measurements to evaluate their algorithms and inform each iteration in the design process.



(Adapted from Figure 3.4b)

Pantheon also enabled our own ML-based congestion-control scheme, Indigo (§3.7). At its core, Indigo learns to mimic the behavior of *congestion-control oracles*, idealized algorithms that perfectly output a correct adjustment to the congestion window. Congestion-control oracles only exist for emulated paths, so we trained Indigo in emulation using imitation learning [97]. The emulated paths in training included Pantheon's calibrated emulators, which boosted Indigo's performance by reducing the gap between simulation and reality.

Chapter 2

Puffer: Learning Adaptive Bitrate Streaming *In Situ*

Video streaming is the predominant Internet application, making up almost three quarters of all traffic [114]. One key algorithmic question in video streaming is *adaptive bitrate selection*, or ABR, which decides the compression level selected for each "chunk," or segment, of the video. ABR algorithms optimize the user's quality of experience (QoE): more-compressed chunks reduce quality, but larger chunks may stall playback if the client cannot download them in time.

In the academic literature, many recent ABR algorithms use statistical and machine-learning methods [4, 70, 109–111, 128], which allow algorithms to consider many input signals and try to perform well for a wide variety of clients. An ABR decision can depend on recent throughput, client-side buffer occupancy, delay, the experience of clients on similar ISPs or types of connectivity, etc. Machine learning can find patterns in seas of data and is a natural fit for this problem domain.

However, it is a perennial lesson that the performance of learned algorithms depends on the data or environments used to train them. ML approaches to video streaming and other wide-area networking challenges are often hampered in their access to good and representative training data. The Internet is complex and diverse, individual nodes only observe a noisy sliver of the system dynamics, and behavior is often heavy-tailed and changes with time. Even with representative throughput traces, accurately simulating or emulating the diversity of Internet paths requires more than replaying such traces and is beyond current capabilities [41, 42, 87, 127].

As a result, the performance of algorithms in emulated environments may not generalize to the Internet [10]. For example, CS2P's gains were more modest over real networks than in simulation [111]. Measurements of Pensieve [70] saw narrower benefits on similar paths [26] and a large-scale streaming service [69]. Other learned algorithms, such as the Remy congestion-control schemes, have also seen inconsistent results on real networks, despite good results in simulation [127].

This chapter seeks to answer: *what does it take to create a learned ABR algorithm that robustly performs well over the wild Internet?* We report the design and findings of Puffer[1], an ongoing research study that operates a video-streaming website open to the public. Over the past year, Puffer has streamed 38.6 years of video to 63,508 distinct users, while recording client telemetry for analysis (current load is about 60 stream-days of data per day). Puffer randomly assigns each session to one of a set of ABR algorithms; users are blinded to the assignment. We find:

In our real-world setting, sophisticated algorithms based on control theory [128] or reinforcement learning [70] did not outperform simple buffer-based control [51]. We found that more-sophisticated algorithms do not necessarily beat a simpler, older algorithm. The newer algorithms were developed and evaluated using throughput traces that may not have captured enough of the Internet's heavy tails and other dynamics when replayed in simulation or emulation. Training them on more-representative traces doesn't necessarily reverse this: we retrained one algorithm using throughput traces drawn from Puffer (instead of its original set of traces) and evaluated it also on Puffer, but the results were similar (§2.4.3).

Statistical margins of error in quantifying algorithm performance are considerable. Prior work on ABR algorithms has claimed benefits of 10–15% [128], 3.2–14% [111], or 12–25% [70], based on throughput traces or real-world experiments lasting hours or days. However, we found that the empirical variability and heavy tails of throughput evolution and rebuffering create statistical margins of uncertainty that make it challenging

[1] https://puffer.stanford.edu

to detect real effects of this magnitude. Even with a *year* of experience per scheme, a 20% improvement in rebuffering ratio would be statistically indistinguishable, i.e., below the threshold of detection with 95% confidence. These uncertainties affect the design space of machine-learning approaches that can practically be deployed [36, 71].

It is possible to robustly outperform existing schemes by combining classical control with an ML predictor trained *in situ* on real data. We describe Fugu, a data-driven ABR algorithm that combines several techniques. Fugu is based on MPC (model predictive control) [128], a classical control policy, but replaces its throughput predictor with a deep neural network trained using supervised learning on data recorded *in situ* (in place), meaning from Fugu's actual deployment environment, Puffer. The predictor has some uncommon features: it predicts *transmission time* given a chunk's file size (vs. estimating throughput), it outputs a probability distribution (vs. a point estimate), and it considers low-level congestion-control statistics among its input signals. Ablation studies (§2.3.6) find each of these features to be necessary to Fugu's performance.

In a controlled experiment during most of 2019, Fugu outperformed existing techniques—including the simple algorithm—in stall ratio (with one exception), video quality, and the variability of video quality (Table 2.1). The improvements were significant both statistically and, perhaps, practically: users who were randomly assigned to Fugu (in blinded fashion) chose to continue streaming for 5–9% longer, on average, than users assigned to the other ABR algorithms.[2]

Our results suggest that, as in other domains, good and representative training is the key challenge for robust performance of learned networking algorithms, a somewhat different point of view from the generalizability arguments in prior work [70, 106, 124]. One way to achieve representative training is to learn in place (*in situ*) on the actual deployment environment, assuming the scheme can be feasibly trained this way and the deployment is widely enough used to exercise a broad range of scenarios.[3] The approach we describe here is only a step in this direction, but we believe Puffer's results suggest that learned systems

[2]This effect was driven solely by users streaming more than 3 hours of video; we do not fully understand it.

[3]Even collecting traces from a deployment environment and replaying them in a simulator or emulator to train a control policy—as is typically necessary in reinforcement learning—is not what we mean by "*in situ*."

Results of primary experiment (Jan. 26–Aug. 7 & Aug. 30–Oct. 16, 2019)

Algorithm	Time stalled (lower is better)	Mean SSIM (higher is better)	SSIM variation (lower is better)	Mean duration (time on site)
Fugu	0.13%	16.64 dB	0.74 dB	33.6 min
MPC-HM [128]	0.22%	16.61 dB	0.79 dB	30.8 min
BBA [51]	0.19%	16.56 dB	1.11 dB	32.1 min
Pensieve [70]	0.17%	16.26 dB	1.05 dB	31.6 min
RobustMPC-HM	0.12%	16.01 dB	0.98 dB	31.0 min

Table 2.1: In an eight-month randomized controlled trial with blinded assignment, the Fugu scheme outperformed other ABR algorithms. The primary analysis includes 637,189 streams played by 54,612 client IP addresses (13.1 client-years in total). Uncertainties are shown in Figures 2.11 and 2.13.

will benefit by addressing the challenge of "*how will we get enough representative scenarios for training—what is enough, and how do we keep them representative over time?*" as a first-class consideration.

We intend to operate Puffer as an "open research" project as long as feasible. We invite the research community to train and test new algorithms on randomized subsets of its traffic, gaining feedback on real-world performance with quantified uncertainty. Along with this dissertation, we are publishing an archive of data and results back to the beginning of 2019 on the Puffer website, with new data and results posted weekly.

In the next few sections, we discuss the background and related work on this problem (§2.1), the design of our blinded randomized experiment (§2.2) and the Fugu algorithm (§2.3), with experimental results in Section 2.4, and a discussion of results and limitations in Section 2.5. In the appendices, we provide a standardized diagram of the experimental flow for the primary analysis (Figure A.1) and describe the data we are releasing (§B).

2.1 Background and related work

The basic problem of adaptive video streaming has been the subject of much academic work; for a good overview, we refer the reader to Yin et al. [128]. We briefly outline the problem here. A service wishes to serve a pre-recorded or live video stream to a broad array of clients over the Internet. Each client's connection has a different and unpredictable time-varying performance. Because there are many clients, it is not feasible for the service to adjust the encoder configuration in real time to accommodate any one client.

Instead, the service encodes the video into a handful of alternative compressed versions. Each represents the original video but at a different quality, target bitrate, or resolution. Client sessions choose from this limited menu. The service encodes the different versions in a way that allows clients to switch midstream as necessary: it divides the video into *chunks*, typically 2–6 seconds each, and encodes each version of each chunk independently, so it can be decoded without access to any other chunks. This gives clients the opportunity to switch between different versions at each chunk boundary; an illustration of this process is in Figure 2.1. The different alternatives are generally referred to as different "bitrates," although streaming services today generally use "variable bitrate" (VBR) encoding [90], where within each alternative stream, the chunks vary in compressed size [130].

Choosing which chunks to fetch. Algorithms that select which alternative version of each chunk to fetch and play, given uncertain future throughput, are known as *adaptive bitrate* (ABR) schemes. These schemes fetch chunks, accumulating them in a playback buffer, while playing the video at the same time. The playhead advances and drains the buffer at a steady rate, 1 s/s, but chunks arrive at irregular intervals dictated by the varying network throughput and the compressed size of each chunk. If the buffer underflows, playback

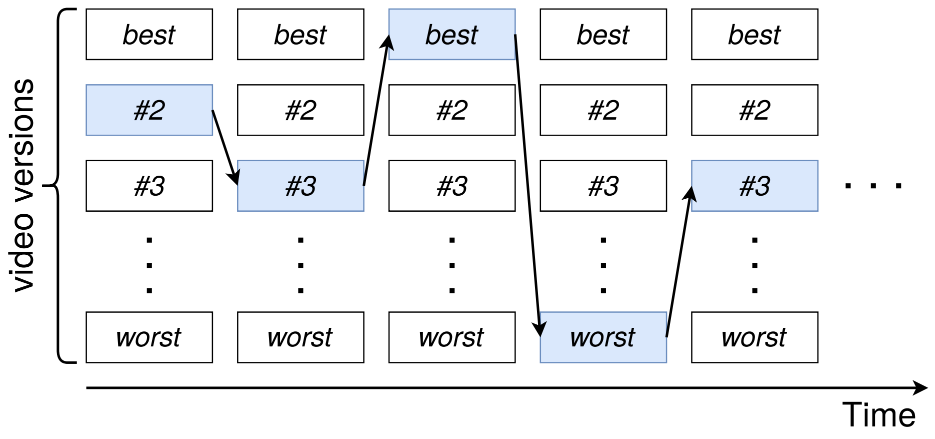


Figure 2.1: Illustration of switching between different video versions

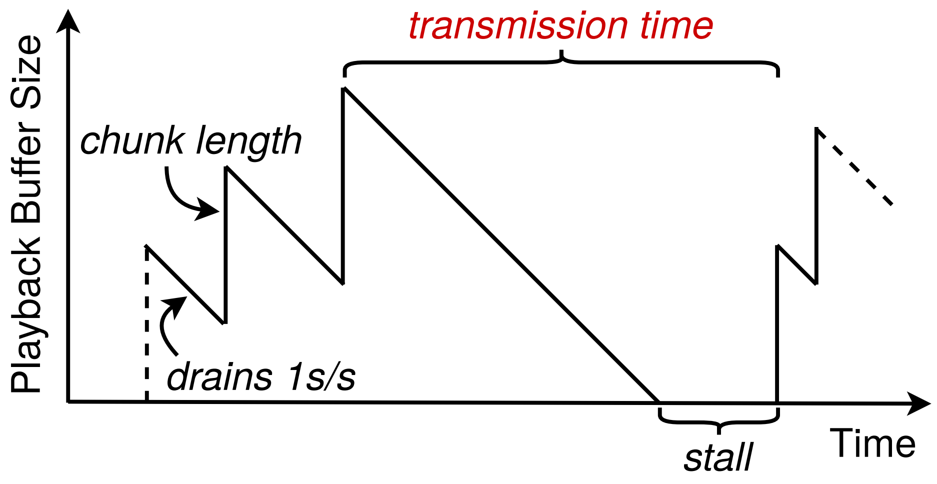
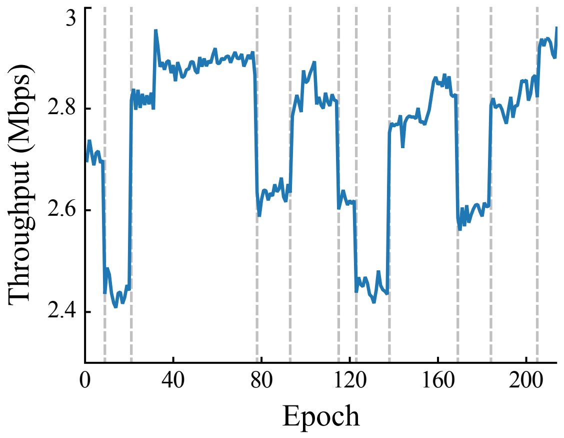


Figure 2.2: Example of an evolving video playback buffer

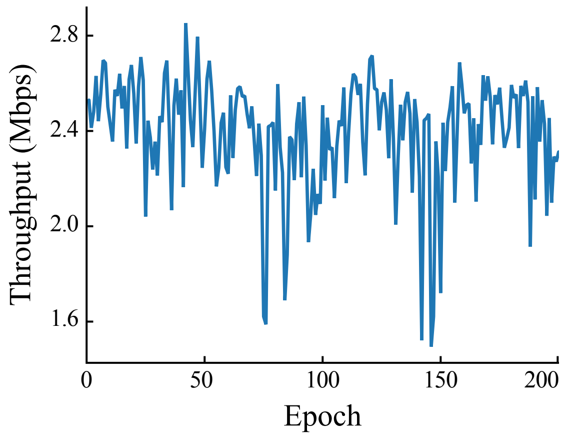
must stall while the client "rebuffers": fetching more chunks before resuming playback. Figure 2.2 gives an example of how the playback buffer changes over time; we refer to the time spent fetching a chunk as the *transmission time*. The goal of an ABR algorithm is typically framed as choosing the optimal sequence of chunks to fetch or replace [109], given recent experience and guesses about the future, to minimize startup time and presence of stalls, maximize the quality of chunks played back, and minimize variation in quality over time (especially abrupt changes in quality). The importance tradeoff for these factors is captured in a QoE metric; several studies have calibrated QoE metrics against human behavior or opinion [8, 35, 60].

Adaptive bitrate selection. Researchers have produced a literature of ABR schemes, including "rate-based" approaches that focus on matching the video bitrate to the network throughput [55, 65, 75], "buffer-based" algorithms that steer the duration of the playback buffer [51, 109, 110], and control-theoretic schemes that try to maximize expected QoE over a receding horizon, given the upcoming chunk sizes and a prediction of the future throughput.

Model Predictive Control (MPC), FastMPC, and RobustMPC [128] fall into the last category. They comprise two modules: a *throughput predictor* that informs a predictive *model* of what will happen to the buffer occupancy and QoE in the near future, depending



(a) CS2P example session (Figure 4a from [111])



(b) Typical Puffer session with similar mean throughput

Figure 2.3: Puffer has not observed CS2P's discrete throughput states. (Epochs are 6 seconds in both plots.)

on which chunks it fetches, with what quality and sizes. MPC uses the model to plan a sequence of chunks over a limited horizon (e.g., the next 5–8 chunks) to maximize the expected QoE. We implemented MPC and RobustMPC for Puffer, using the same predictor as the paper: the harmonic mean of the last five throughput samples.

CS2P [111] and Oboe-tuned RobustMPC [4] are related to MPC; they constitute better throughput predictors that inform the same control strategy (MPC). These throughput predictors were trained on real datasets that recorded the evolution of throughput over time within a session. CS2P clusters users by similarity and models their evolving throughput as a Markovian process with a small number of discrete states; Oboe uses a similar model to detect when the network path has changed state. In our dataset, we have not observed CS2P and Oboe's observation of discrete throughput states (Figure 2.3).

Fugu fits in this same category of algorithms. It also uses MPC as the control strategy, informed by a network predictor trained on real data. This component, which we call the Transmission Time Predictor (TTP), incorporates a number of uncommon features, none of which can claim novelty on its own. The TTP explicitly predicts the transmission time of a chunk with given size and isn't a "throughput" predictor *per se*. A throughput predictor models the transmission time of a chunk as scaling linearly with size, but it is well known that observed throughput varies with file size [10, 90, 130], in part because of the effects of congestion control and because chunks of different sizes experience different time intervals of the path's varying capacity. To our knowledge, Fugu is the first to use this fact operationally as part of a control policy.

Fugu's predictor is also *probabilistic*: it outputs not a single predicted transmission time, but a probability distribution on possible outcomes. The use of uncertainty in model predictive control has a long history [103], but to our knowledge Fugu is the first to use stochastic MPC in this context. Finally, Fugu's predictor is a neural network, which lets it consider an array of diverse signals that relate to transmission time, including raw congestion-control statistics from the sender-side TCP implementation [43,118]. We found that several of these signals (RTT, CWND, etc.) benefit ABR decisions (§2.4).

Pensieve [70] is an ABR scheme also based on a deep neural network. Unlike Fugu, Pensieve uses the neural network not simply to make predictions but to make *decisions* about which chunks to send. This affects the type of learning used to train the algorithm. While CS2P and Fugu's TTP can be trained with *supervised learning* (to predict chunk transmission times recorded from past data), it takes more than data to train a scheme that makes decisions; one needs training *environments* that respond to a series of decisions and judge their consequences. This is known as reinforcement learning (RL). Generally speaking, RL techniques expect a set of training environments that can exercise a control policy through a range of situations and actions [3], and need to be able to observe a detectable difference in performance by slightly varying a control action. Systems that are challenging to simulate or that have too much noise present difficulties [36,71].

2.2 Puffer: an ongoing live study of ABR

To understand the challenges of video streaming and measure the behavior of ABR schemes, we built Puffer, a free, publicly accessible website that live-streams six over-the-air commercial television channels (Figure 2.4). Puffer operates as a randomized controlled trial; sessions are randomly assigned to one of a set of ABR or congestion-control schemes. The study participants include any member of the public who wishes to participate. Users are blinded to algorithm assignment, and we record client telemetry on video quality and playback. A Stanford Institutional Review Board determined that Puffer does not constitute

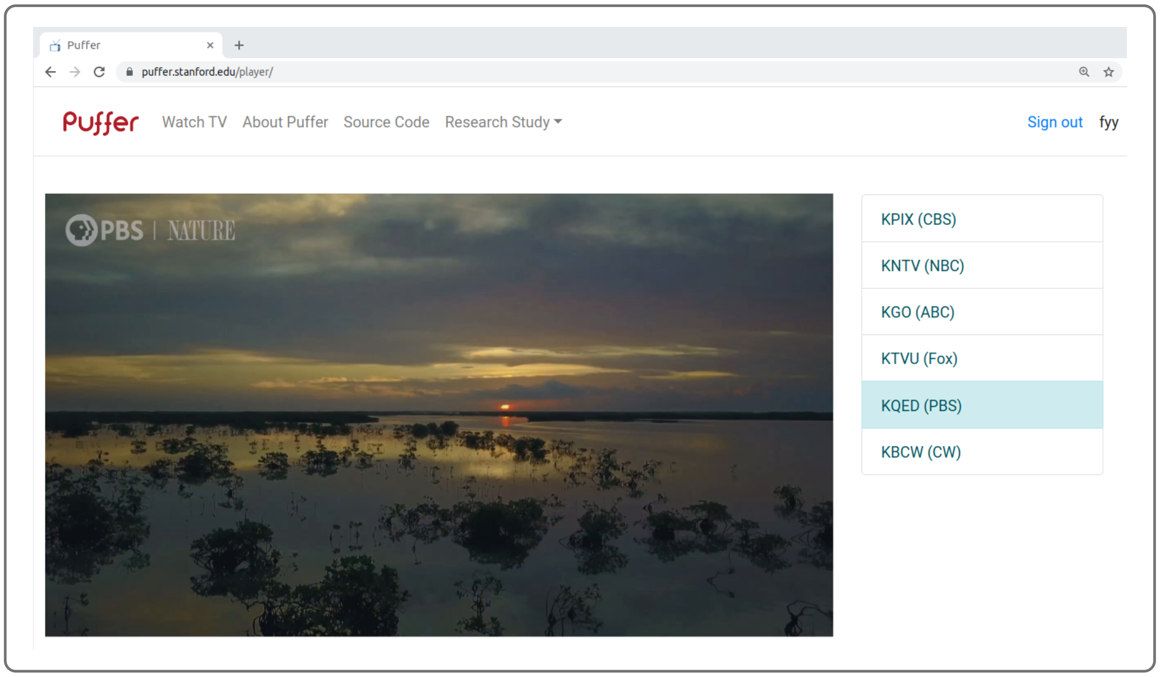


Figure 2.4: Puffer website

human subjects research.

Our reasoning for streaming live television was to collect data from enough participants and network paths to draw robust conclusions about the performance of algorithms for ABR control and network prediction. Live television is an evergreen source of popular content that had not been broadly available for free on the Internet. Our study benefits, in part, from a law that allows nonprofit organizations to retransmit over-the-air television signals without charge [1]. Here, we describe details of the system, experiment, and analysis.

2.2.1 Back-end: decoding, encoding, SSIM

Figure 2.5 shows the architecture of Puffer. Puffer receives six television channels using a VHF/UHF antenna and an ATSC demodulator, which outputs MPEG-2 transport streams in UDP. We wrote software to decode a stream to chunks of raw decoded video and audio, maintaining synchronization (by inserting black fields or silence) in the event of lost transport-stream packets on either substream. Video chunks are 2.002 seconds long,

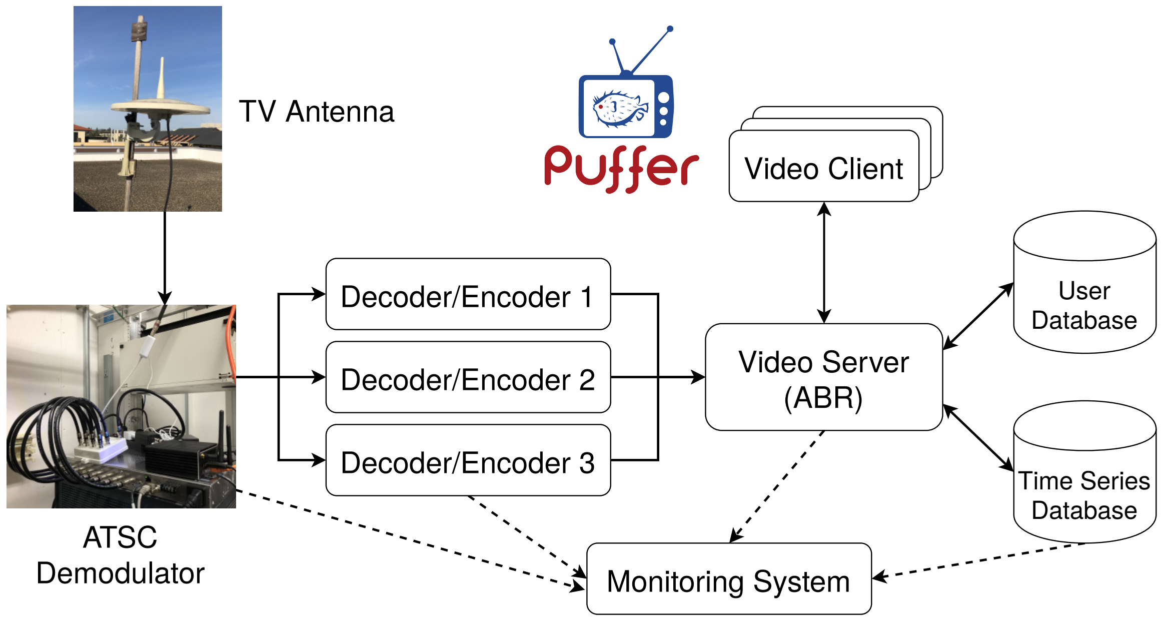


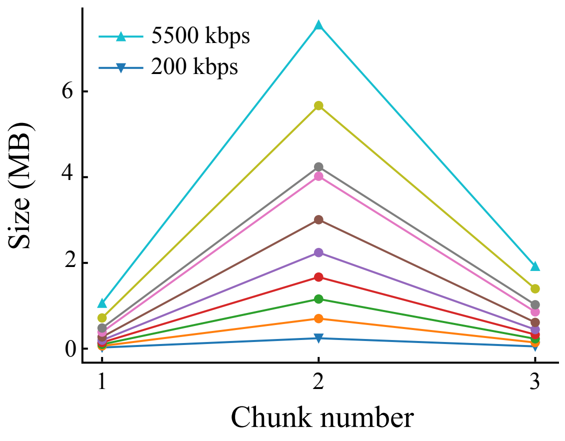
Figure 2.5: Architecture of Puffer

reflecting the 1/1001 factor for NTSC frame rates. Audio chunks are 4.8 seconds long. Video is de-interlaced with `ffmpeg` to produce a "canonical" 1080p60 or 720p60 source for compression.

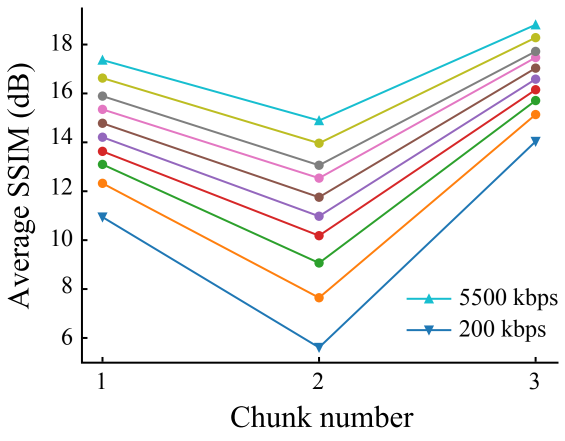
Puffer encodes each video chunk in ten different H.264 versions, using `libx264` in `veryfast` mode. The encodings range from 240p60 video with constant rate factor (CRF) of 26 (about 200 kbps) to 1080p60 video with CRF of 20 (about 5,500 kbps). Audio chunks are encoded in the Opus format.

Puffer then uses `ffmpeg` to calculate each encoded chunk's SSIM [121], a measure of video quality, relative to the canonical source. This information is used by the objective function of BBA, MPC, RobustMPC, and Fugu, and for our evaluation. In practice, the relationship between bitrate and quality varies chunk-by-chunk (Figure 2.6), and users cannot perceive compressed chunk sizes directly—only what is shown on the screen. ABR schemes that maximize bitrate do not necessarily see a commensurate benefit in picture quality (Figure 2.7).

Encoding six channels in ten versions each (60 streams total) with `libx264` consumes about 48 cores of an Intel x86-64 2.7 GHz CPU in steady state. Calculating the SSIM of



(a) VBR encoding lets chunk size vary within a stream [130].



(b) Picture quality also varies with VBR encoding [90].

Figure 2.6: Variations in picture quality and chunk size within each stream suggest a benefit from choosing chunks based on SSIM and size, rather than average bitrate (legend).

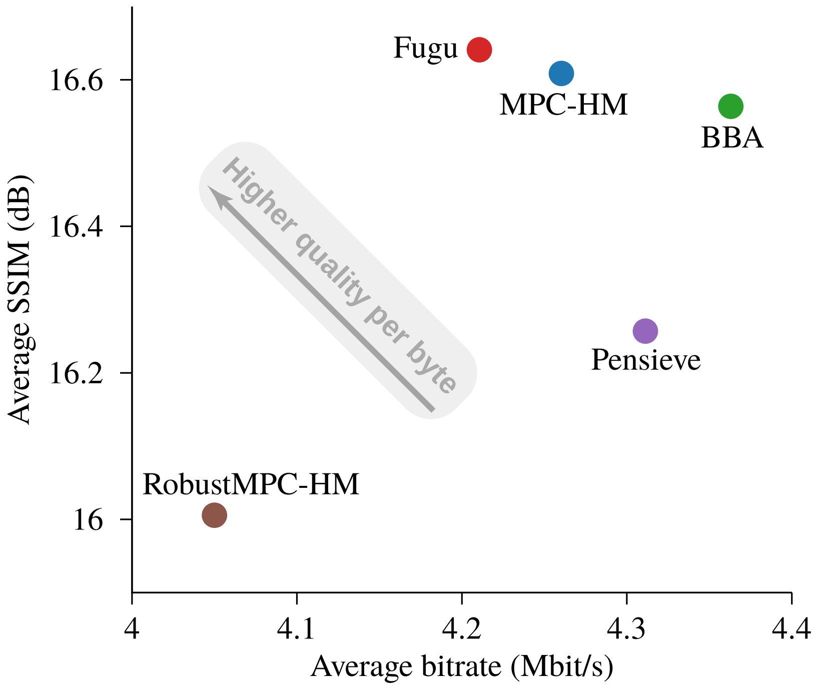


Figure 2.7: On Puffer, schemes that maximize average SSIM (MPC-HM, RobustMPC-HM, and Fugu) delivered higher quality video per byte sent, vs. those that maximize bitrate directly (Pensieve) or the SSIM of each chunk (BBA).

each encoded chunk consumes an additional 18 cores.

2.2.2 Serving chunks to the browser

To make it feasible to deploy and test arbitrary ABR schemes, Puffer uses a “dumb” player (using the HTML5 `<video>` tag and the JavaScript Media Source Extensions) on the client side, and places the ABR scheme at the server. We have a 48-core server with 10 Gbps Ethernet in a datacenter at Stanford. The browser opens a WebSocket (TLS/TCP) connection to a daemon on the server. Each daemon is configured with a different TCP congestion control (for the primary analysis, we used BBR [21]) and ABR scheme. Some schemes are more efficiently implemented than others; on average the CPU load from serving client traffic (including TLS, TCP, and ABR) is about 5% of an Intel x86-64 2.7 GHz core per stream. Sessions are randomly assigned to serving daemons. Users can switch channels without breaking their TCP connection and may have many “streams” within each session.

Puffer is not a client-side DASH [77] (Dynamic Adaptive Streaming over HTTP) system. Like DASH, though, Puffer is an ABR system streaming chunked video over a TCP connection, and runs the same ABR algorithms that DASH systems can run. We don’t expect this architecture to replace client-side ABR (which can be served by CDN edge nodes), but we expect its conclusions to translate to ABR schemes broadly. The Puffer website works in the Chrome, Firefox, Edge, and Opera browsers, including on Android phones, but does not play in the Safari browser or on iOS (which lack support for the Media Source Extensions or Opus audio).

2.2.3 Hosting arbitrary ABR schemes

We implemented buffer-based control (BBA), MPC, RobustMPC, and Fugu in back-end daemons that serve video chunks over the WebSocket. We use SSIM in the objective functions for each of these schemes. For BBA, we use the formula in the original paper [51] to decide the maximum chunk size, and subject to this constraint, the chunk with the highest SSIM is selected to stream. We also choose reservoir values consistent with our 15-second maximum buffer.

Deploying Pensieve for live streaming. We use the released Pensieve code (written in Python with TensorFlow) directly. When a client is assigned to Pensieve, Puffer spawns a Python subprocess running Pensieve's multi-video model.

We contacted the Pensieve authors to request advice on deploying the algorithm in a live, multi-video, real-world setting. The authors recommended that we use a longer-running training and that we tune the entropy parameter when training the multi-video neural network. We wrote an automated tool to train 6 different models with various entropy reduction schemes. We tested these manually over a few real networks, then selected the model with the best performance. We modified the Pensieve code (and confirmed with the authors) so that it does not expect the video to end before a user's session completes. We were not able to modify Pensieve to optimize SSIM; it considers the average bitrate of each Puffer stream. We adjusted the video chunk length to 2.002 seconds and the buffer threshold to 15 seconds to reflect our parameters. For training data, we used the authors' provided script to generate 1000 simulated videos as training videos, and a combination of the FCC and Norway traces linked to in the Pensieve codebase as training traces.

2.2.4 The Puffer experiment

To recruit participants, we purchased Google and Reddit ads for keywords such as "live tv" and "tv streaming" and paid people on Amazon Mechanical Turk to use Puffer. We were also featured in press articles. Popular programs (e.g. the 2019 and 2020 Super Bowls, the Oscars, World Cup, and "Bachelor in Paradise") brought large spikes ($> 20\times$) over baseline load. Our current average load is about 60 concurrent streams.

Between Jan. 26, 2019 and Feb. 2, 2020, we have streamed 38.6 years of video to 63,508 registered study participants using 111,231 unique IP addresses. About eight months of that period was spent on the "primary experiment," a randomized trial comparing Fugu with other algorithms: MPC, RobustMPC, Pensieve, and BBA (a summary of features is in Table 2.2). This period saw a total of 314,577 streaming sessions, and 1,904,316 individual streams. An experimental-flow diagram in the standardized CONSORT format [102] is in Appendix A.

We record client telemetry as time-series data, detailing the size and SSIM of every

Algorithm	Control	Predictor	Optimization goal	How trained
BBA	linear control	*n/a*	$+$SSIM *s.t.* bitrate $<$ limit	*n/a*
MPC-HM	MPC	HM	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	*n/a*
RobustMPC-HM	robust MPC	HM	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	*n/a*
Pensieve	DNN	*n/a*	$+$bitrate, $-$stalls, $-\Delta$bitrate	RL in simulation
Fugu	MPC	DNN	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	supervised learning *in situ*

Table 2.2: Distinguishing features of algorithms used in the primary experiment. HM = harmonic mean of last five throughput samples. MPC = model predictive control. DNN = deep neural network. RL = reinforcement learning.

video chunk, the time to deliver each chunk to the client, the buffer size and rebuffering events at the client, the TCP statistics on the server, and the identity of the ABR and congestion-control schemes. A full description of the data is in Appendix B.

Metrics and statistical uncertainty. We group the time series by user stream to calculate a set of summary figures: the total time between the first and last recorded events of the stream, the startup time, the total watch time between the first and last successfully played portion of the stream, the total time the video is stalled for rebuffering, the average SSIM, and the chunk-by-chunk variation in SSIM. The ratio between "total time stalled" and "total watch time" is known as the rebuffering ratio or stall ratio, and is widely used to summarize the performance of streaming video systems [63].

We observe considerable heavy-tailed behavior in most of these statistics. Watch times are skewed (Figure 2.13), and while the *risk* of rebuffering is important to any ABR algorithm, actual rebuffering is a rare phenomenon. Of the 637,189 eligible streams considered for the primary analysis across all five ABR schemes, only 24,328 (4%) of those streams had *any* stalls, mirroring commercial services [63].

These skewed distributions create more room for the play of chance to corrupt the bottom-line statistics summarizing a scheme's performance—even two identical schemes will see considerable variation in average performance until a substantial amount of data is assembled. In this study, we worked to quantify the statistical uncertainty that can be attributed to the play of chance in assigning sessions to ABR algorithms. We calculate confidence intervals on rebuffering ratio with the bootstrap method [37], simulating streams drawn empirically from each scheme's observed distribution of rebuffering ratio as a function of stream duration. We calculate confidence intervals on average SSIM using the formula for weighted standard error, weighting each stream by its duration.

These practices result in substantial confidence intervals: with at least 2.5 years of data for each scheme, the width of the 95% confidence interval on a scheme's stall ratio is between ±13% and ±21% of the mean value. This is comparable to the magnitude of the total benefit reported by some academic work that used much shorter real-world experiments (Appendix C shows the consequences of reporting performance on insufficient data). Even a recent study of a Pensieve-like scheme on Facebook [69], encompassing 30 million streams, did not detect a change in rebuffering ratio outside the level of statistical noise.

We conclude that considerations of uncertainty in real-world learning and experimentation, especially given uncontrolled data from the Internet with real users, deserve further study. Strategies to import real-world data into repeatable emulators [127] or reduce their variance [71] will likely be helpful in producing robust learned networking algorithms.

2.3 Fugu: design and implementation

Fugu is a control algorithm for bitrate selection, designed to be feasibly trained in place (*in situ*) on a real deployment environment. It consists of a classical controller (model predictive control, the same as in MPC-HM), informed by a nonlinear predictor that can be trained with supervised learning.

Figure 2.8 shows Fugu's high-level design. Fugu runs on the server, making it easy to update its model and aggregate performance data across clients over time. Clients send necessary telemetry, such as buffer levels, to the server.

The controller, described in Section 2.3.4, makes decisions by following a classical control algorithm to optimize an objective QoE function (§2.3.1) based on predictions for how long each chunk would take to transmit. These predictions are provided by the Transmission Time Predictor (TTP) (§2.3.2), a neural network that estimates a probability distribution for the transmission time of a proposed chunk with given size.

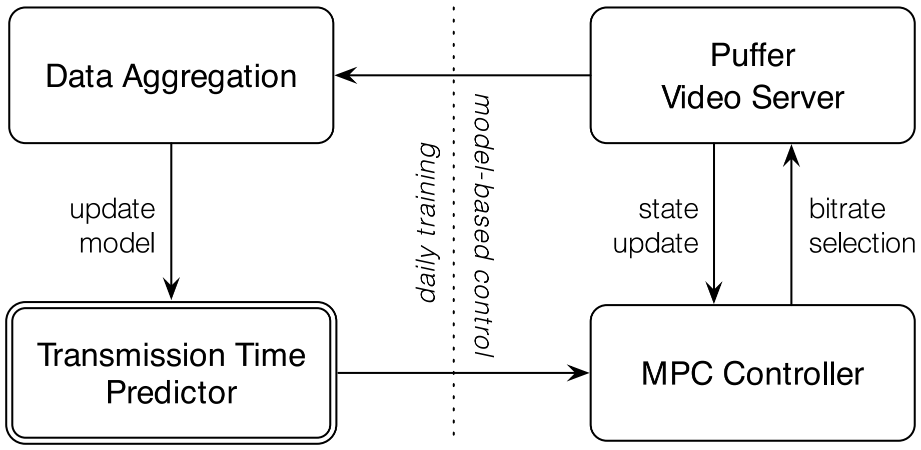


Figure 2.8: Overview of Fugu

2.3.1 Objective function

For each video chunk K_i, Fugu has a selection of versions of this chunk to choose from, K_i^s, each with a different size s. As with prior approaches, Fugu quantifies the QoE of each chunk as a linear combination of video quality, video quality variation, and stall time [128]. Unlike some prior approaches, which use the average compressed bitrate of each encoding setting as a proxy for image quality, Fugu optimizes a perceptual measure of picture quality—in our case, SSIM. This has been shown to correlate with human opinions of QoE [35]. We emphasize that we use the exact same objective function in our version of MPC and RobustMPC as well.

Let $Q(K)$ be the video quality of a chunk K, $T(K)$ be the uncertain transmission time of K, and B_i be the current playback buffer size. Following [128], Fugu defines the QoE obtained by sending K_i^s (given the previously sent chunk K_{i-1}) as

$$\begin{aligned} QoE(K_i^s, K_{i-1}) = Q(K_i^s) - \lambda |Q(K_i^s) - Q(K_{i-1})| \\ - \mu \cdot \max\{T(K_i^s) - B_i, 0\}, \end{aligned} \tag{2.1}$$

where $\max\{T(K_i^s) - B_i, 0\}$ describes the stall time experienced by sending K_i^s, and λ and μ are configuration constants for how much to weight video quality variation and rebuffering. Fugu plans a trajectory of sizes s of the future H chunks to maximize their expected total QoE.

2.3.2 Transmission Time Predictor (TTP)

Once Fugu decides which chunk from K_i^s to send, two portions of the QoE become known: the video quality and video quality variation. The remaining uncertainty is the stall time. The server knows the current playback buffer size, so what it needs to know is the transmission time: how long will it take for the client to receive the chunk? Given an oracle that reports the transmission time of any chunk, the MPC controller can compute the optimal plan to maximize QoE.

Fugu uses a trained neural-network transmission-time predictor to approximate the oracle. For each chunk in the fixed H-step horizon, we train a separate predictor. E.g., if optimizing for the total QoE of the next five chunks, five neural networks are trained. This lets us parallelize training.

Each TTP network for the future step $h \in \{0, \ldots, H-1\}$ takes as input a vector of:

1. sizes of past t chunks $K_{i-t}, \ldots, K_{i-1}$,
2. actual transmission times of past t chunks: $T_{i-t}, \ldots, T_{i-1}$,
3. internal TCP statistics (Linux `tcp_info` structure),
4. size s of a proposed chunk K_{i+h}^s.

The TCP statistics include the current congestion window size, the number of unacknowledged packets in flight, the smoothed RTT estimate, the minimum RTT, and the TCP estimated throughput (`tcpi_delivery_rate`).

Prior approaches have used Harmonic Mean (HM) [128] or a Hidden Markov Model (HMM) [111] to predict a single throughput for the entire lookahead horizon irrespective of the size of chunk to send. In contrast, the TTP acknowledges the fact that observed throughput varies with chunk size [10, 90, 130] by taking the size of proposed chunk K_{i+h}^s as an explicit input. In addition, it outputs a discretized probability distribution of predicted transmission time $\hat{T}(K_{i+h}^s)$.

2.3.3 Training the TTP

We sample from the real usage data collected by *any* scheme running on Fugu and feed individual user streams to the TTP as training input. For the TTP network in the future step h, each user stream contains a chunk-by-chunk series of (a) the input 4-vector with the last element to be size of the actually sent chunk K_{i+h}, and, (b) the actual transmission time T_{i+h} of chunk K_{i+h} as desired output; the sequence is shuffled to remove correlation. It is worth noting that unlike prior work [70, 111] that learned from throughput traces, TTP is trained directly on real chunk-by-chunk data.

We train the TTP with standard supervised learning: the training minimizes the cross-entropy loss between the output probability distribution and the discretized actual transmission time using stochastic gradient descent.

We retrain the TTP every day, using training data collected over the prior 14 days, to avoid the effects of dataset shift and catastrophic forgetting [94, 96]. Within the 14-day window, we weight more recent days more heavily. The weights from the previous day's model are loaded to warm-start the retraining.

2.3.4 Model-based controller

Our MPC controller (used for MPC-HM, RobustMPC-HM, and Fugu) is a stochastic optimal controller that maximizes the expected cumulative QoE in Equation 2.1 with replanning. It queries TTP for predictions of transmission time and outputs a plan $K_i^s, K_{i+1}^s, \ldots, K_{i+H-1}^s$ by value iteration [12]. After sending K_i^s, the controller observes and updates the input vector passed into TTP, and replans again for the next chunk.

Given the current playback buffer level B_i and the last sent chunk K_{i-1}, let $v_i^*(B_i, K_{i-1})$ denote the maximum expected sum of QoE that can be achieved in the H-step lookahead horizon. We have value iteration as follows:

$$\begin{aligned} v_i^*(B_i, K_{i-1}) = \max_{K_i^s} \Big\{ \sum_{t_i} & \Pr[\hat{T}(K_i^s) = t_i] \cdot \\ & (QoE(K_i^s, K_{i-1}) + v_{i+1}^*(B_{i+1}, K_i^s)) \Big\}, \end{aligned} \tag{2.2}$$

where $\Pr[\hat{T}(K_i^s) = t_i]$ is the probability predicted by TTP for the transmission time of K_i^s to be t_i, and B_{i+1} can be derived by system dynamics [128] given an enumerated (discretized) t_i. The controller computes the optimal trajectory by solving the above value iteration with dynamic programming (DP). To make the DP computational feasible, it also discretizes B_i into bins and uses forward recursion with memoization to only compute for relevant states.

2.3.5 Implementation

TTP takes as input the past $t = 8$ chunks, and outputs a probability distribution over 21 bins of transmission time: $[0, 0.25), [0.25, 0.75), [0.75, 1.25), \dots, [9.75, \infty)$, with 0.5 seconds as the bin size except for the first and the last bins. TTP is a fully connected neural network, with two hidden layers with 64 neurons each. We tested different TTPs with various numbers of hidden layers and neurons, and found similar training losses across a range of conditions for each. We implemented TTP and the training in PyTorch, but we load the trained model in C++ when running on the production server for performance. A forward pass of TTP's neural network in C++ imposes minimal overhead per chunk (less than 0.3 ms on average on a recent x86-64 core). The MPC controller optimizes over $H = 5$ future steps (about 10 seconds). We set $\lambda = 1$ and $\mu = 100$ to balance the conflicting goals in QoE. Each retraining takes about 6 hours on a 48-core server.

2.3.6 Ablation study of TTP features

We performed an ablation study to assess the impact of the TTP's features (Figure 2.9). The prediction accuracy is measured using mean squared error (MSE) between the predicted transmission time and the actual (absolute, unbinned) value. For the TTP that outputs a probability distribution, we compute the expected transmission time by weighting the median value of each bin with the corresponding probability. Here are the more notable results:

Use of low-level congestion-control statistics. The TTP's nature as a DNN lets it consider a variety of noisy inputs, including low-level congestion-control statistics. We feed the kernel's `tcp_info` structure to the TTP, and find that several of these fields contribute positively to the TTP's accuracy, especially the RTT, CWND, and number of packets

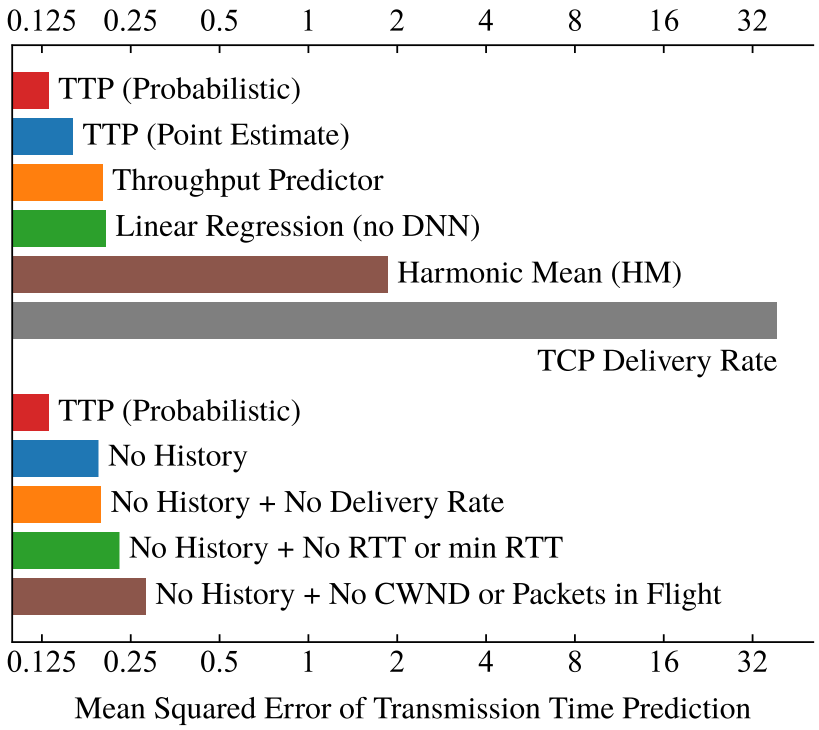


Figure 2.9: Ablation study of Fugu's Transmission Time Predictor (TTP). Removing any of the TTP's inputs reduced its ability to predict the transmission time of a video chunk. A non-probabilistic TTP ("Point Estimate") and one that predicts throughput without regard to chunk size ("Throughput Predictor") both performed markedly worse. TCP statistics (RTT, CWND, packets in flight) also proved helpful.

in flight (Figure 2.9). Although client-side ABR systems cannot typically access this structure directory because the statistics live on the sender, these results should motivate the communication of richer data to ABR algorithms wherever they live.

Transmission-time prediction. The TTP explicitly considers the size of a proposed chunk, rather than predicting throughput and then modeling transmission time as scaling linearly with chunk size [10,90,130]. We compared the TTP with an equivalent throughput predictor that is agnostic to the chunk's size (keeping everything else unchanged). The TTP's predictions were much more accurate (Figure 2.9).

Prediction with uncertainty. The TTP outputs a *probability distribution* of transmission times. This allows for better decision making compared with a single point estimate without uncertainty. We evaluated the expected accuracy of a probabilistic TTP vs. a point-estimate version that outputs the median value of the most-probable bin, and found an improvement in prediction accuracy with the former (Figure 2.9). To measure the end-to-end benefits of a probabilistic TTP, we deployed both versions on Fugu in August 2019 and collected 39 stream-days of data. It performed much worse than normal Fugu: the rebuffering ratio was 5× worse, without significant improvement in SSIM (data not shown).

Use of neural network. We found a significant benefit from using a deep neural network in this application, compared with a linear-regression model that was trained the same way. The latter model performed much worse on prediction accuracy (Figure 2.9). We also deployed it on Fugu and collected 448 stream-days of data in Aug.–Oct. 2019; its

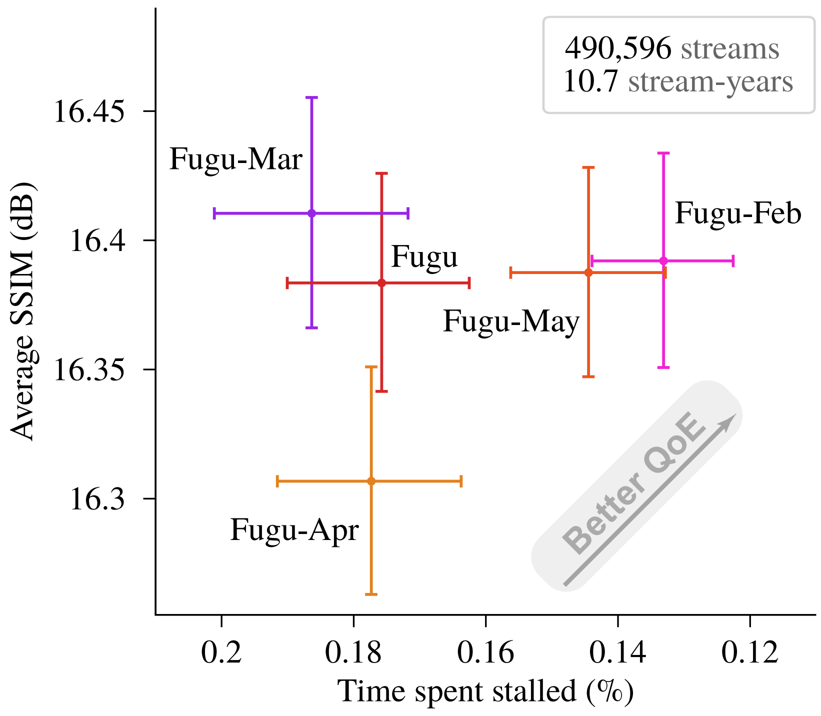


Figure 2.10: Fugu, which is retrained every day, did not outperform older versions of itself that were trained up to 11 months earlier. Our practice of daily retraining appears to be overkill.

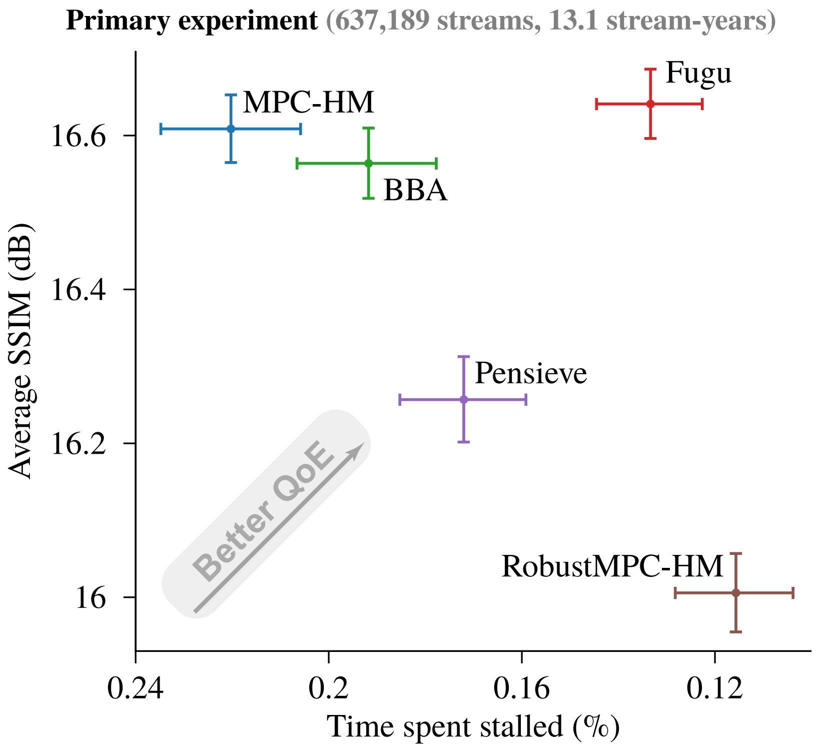
rebuffering ratio was 2.5× worse (data not shown).

Daily retraining. To evaluate our practice of retraining the TTP each day, we conducted a randomized comparison of several "out-of-date" versions of the TTP on Fugu between Aug. 7 and Aug. 30, 2019, and between Oct. 16, 2019 and Jan. 2, 2020. We compared vintages of the TTP that had been trained in February, March, April, and May 2019, alongside the TTP that is retrained each day. (We emphasize that the older TTP vintages were also learned *in situ* on two weeks of data from the actual deployment environment—they are simply earlier versions of the same predictor.) Somewhat to our surprise and disappointment, we were not able to document a benefit from daily retraining (Figure 2.10). This may reflect a lack of dynamism in the Fugu userbase, or the fact that once "enough" data is available to put the predictor through its paces, more-recent data is not necessarily beneficial, or some other reason. We suspect the older predictors might become stale at some point in the future, but for the moment, our practice of daily retraining appears to be overkill.

2.4 Experimental results

We now present findings from our experiments with the Puffer study, including the evaluation of Fugu. Our main results are shown in Figure 2.11. In summary, we conducted a parallel-group, blinded-assignment, randomized controlled trial of five ABR schemes between Jan. 26 and Aug. 7, and between Aug. 30 and Oct. 16, 2019. The data include 13.1 stream-years of data split across five algorithms, counting all streams that played at least 4 seconds of video. A standardized diagram of the experimental flow is available in the appendix (Figure A.1).

We found that simple "buffer-based" control (BBA) performs surprisingly well, despite its status as a frequently outperformed research baseline. The only scheme to consistently outperform BBA in both stalls and quality was Fugu, but only when *all* features of the TTP were used. If we remove the probabilistic "fuzzy" nature of Fugu's predictions, *or* the "depth" of the neural network, *or* the prediction of transmission time as a function of chunk size (and not simply throughput), Fugu forfeits its advantage (§2.3.6). Fugu also



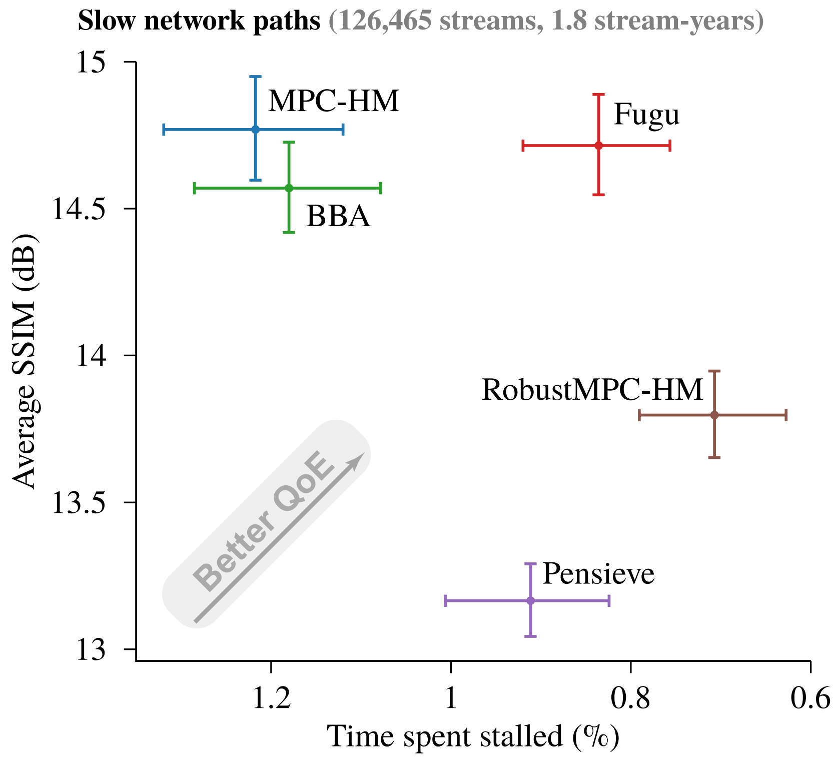


Figure 2.11: **Main results.** (caption continued on next page)

Figure 2.11: (caption continued from previous page) In a blinded randomized controlled trial that included 13.1 years of video streamed to 54,612 client IP addresses over an eight-month period, Fugu reduced the fraction of time spent stalled (except with respect to RobustMPC-HM), increased SSIM, and reduced SSIM variation within each stream (tabular data in Table 2.1). "Slow" network paths have average throughput less than 6 Mbit/s; following prior work [70, 128], these paths are more likely to require nontrivial bitrate-adaptation logic. Such streams accounted for 14% of overall viewing time and 83% of stalls. Error bars show 95% confidence intervals.

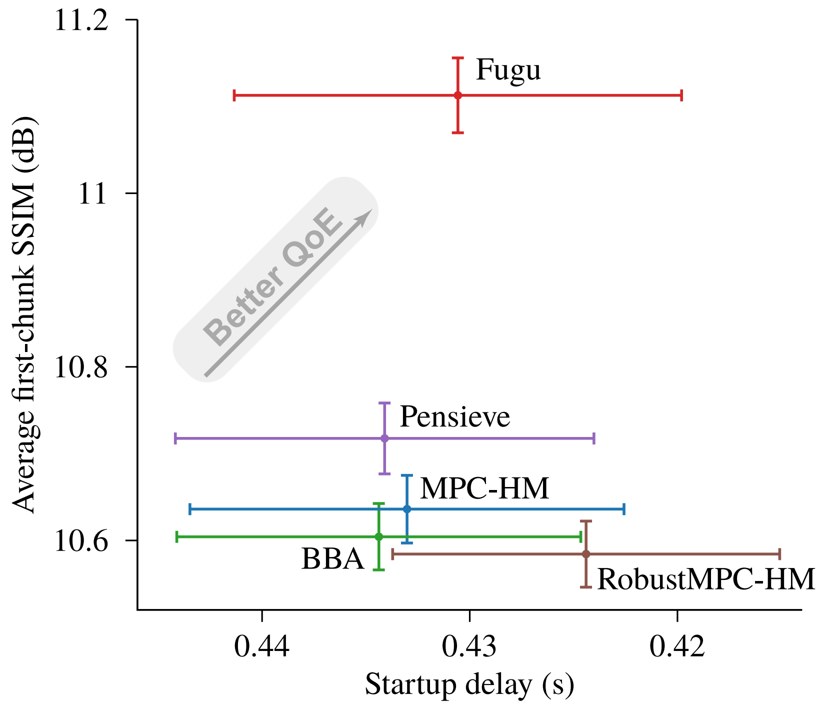


Figure 2.12: On a cold start, Fugu's ability to bootstrap ABR decisions from TCP statistics (e.g., RTT) boosts initial quality.

outperformed other schemes in terms of SSIM variability (Table 2.1). On a cold start to a new session, prior work [54, 111] suggested a need for session clustering to determine the quality of the first chunk. TTP provides an alternative approach: low-level TCP statistics are available as soon as the (HTTP/WebSocket, TLS, TCP) connection is established and allow Fugu to begin safely at a higher quality (Figure 2.12).

We conclude that robustly beating "simple" algorithms with machine learning may be surprisingly difficult, notwithstanding promising results in contained environments such as simulators and emulators. The gains that learned algorithms have in optimization or smarter decision making may come at a tradeoff in brittleness or sensitivity to heavy-tailed behavior.

2.4.1 Fugu users streamed for longer

We observed significant differences in the session durations of users across algorithms (Figure 2.13). Users whose sessions were assigned to Fugu chose to remain on the Puffer video player about 5–9% longer, on average, than those assigned to other schemes. Users were blinded to the assignment, and we believe the experiment was carefully executed not to “leak” details of the underlying scheme (MPC and Fugu even share most of their codebase). The average difference was driven solely by the upper 4% tail of viewership duration (sessions lasting more than 3 hours)—viewers assigned to Fugu are much more likely to keep streaming beyond this point, even as the distributions are nearly identical until then.

Time-on-site is a figure of merit in the video-streaming industry and might be increased by delivering better-quality video with fewer stalls, but we simply do not know enough about what is driving this phenomenon. In fact, although Fugu achieves the highest SSIM on average in the primary experiment (Figure 2.11), the difference in video quality between Fugu and RobustMPC (lowest average SSIM) is unlikely to be perceptible from any single video chunk; RobustMPC even has a lower stall ratio than Fugu. Therefore, time-on-site might also be correlated with variation in quality over time and other metrics affecting QoE.

2.4.2 The benefits of learning *in situ*

Each of the ABR algorithms we deployed has been evaluated in emulation in prior work [70, 128]. Notably, the results in those works are qualitatively different from some of the real world results we have seen here—for example, buffer-based control matching or outperforming MPC-HM and Pensieve.

To investigate this further, we constructed an emulation environment similar to that used in [70]. This involved running the Puffer media server locally, and launching headless

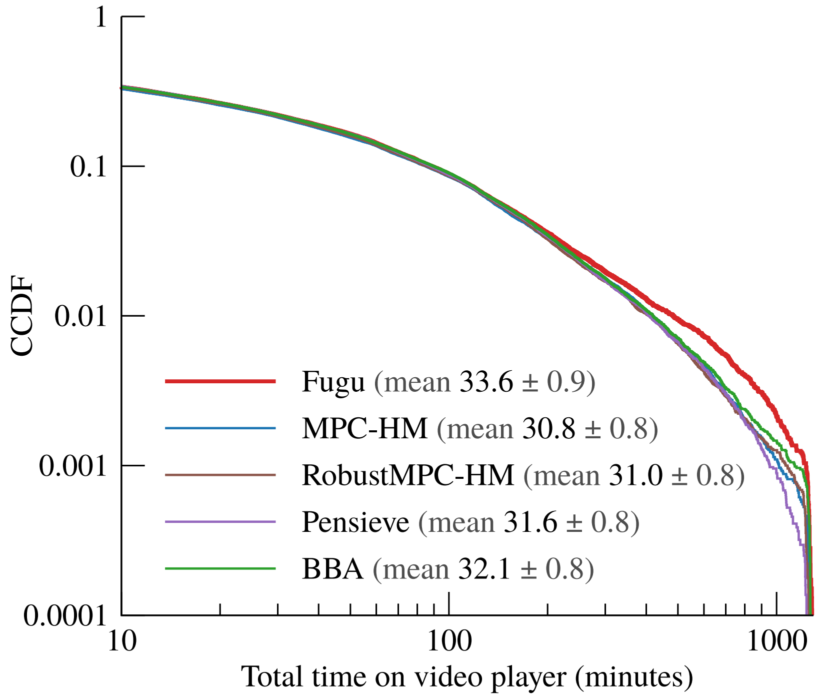
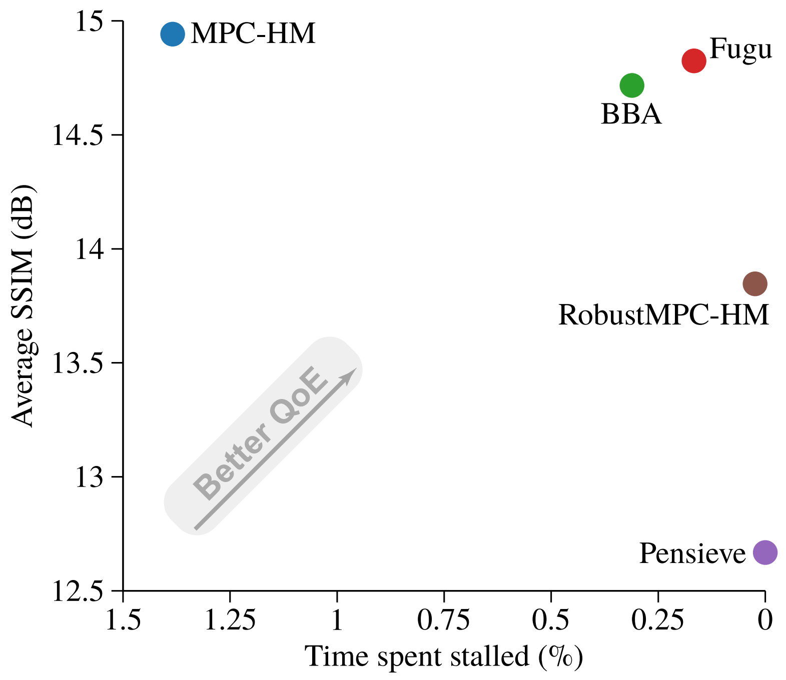


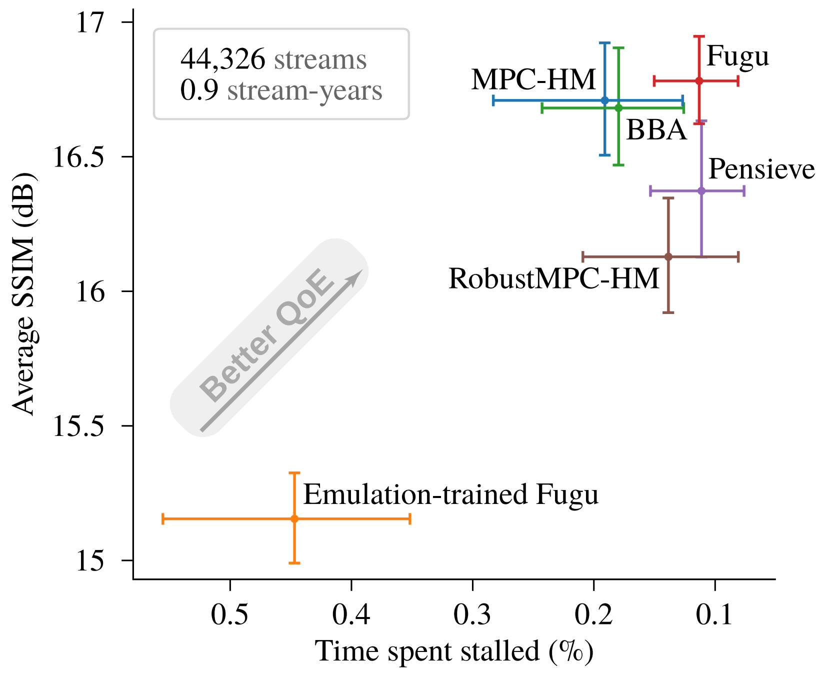
Figure 2.13: Users randomly assigned to Fugu chose to remain on the Puffer video player about 5%–9% longer, on average, than those assigned to other schemes. Users were blinded to the assignment. Legend shows 95% confidence intervals on the average time-on-site in minutes.

Chrome clients inside mahimahi [82] shells to connect to the server. Each mahimahi shell imposed a 40 ms end-to-end delay on traffic originating inside it and limited the downlink capacity over time to match the capacity recorded in a set of FCC broadband network traces [25]. As in the Pensieve evaluation, uplink speeds in all shells were capped at 12 Mbps. Within this test setup, we automated 12 clients to repeatedly connect to the media server, which would play a 10 minute clip recorded on NBC over each network trace in the dataset. Each client was assigned to a different ABR algorithm, and played the 10 minute video repeatedly over more than 15 hours of FCC traces. Results are shown in Figure 2.14.

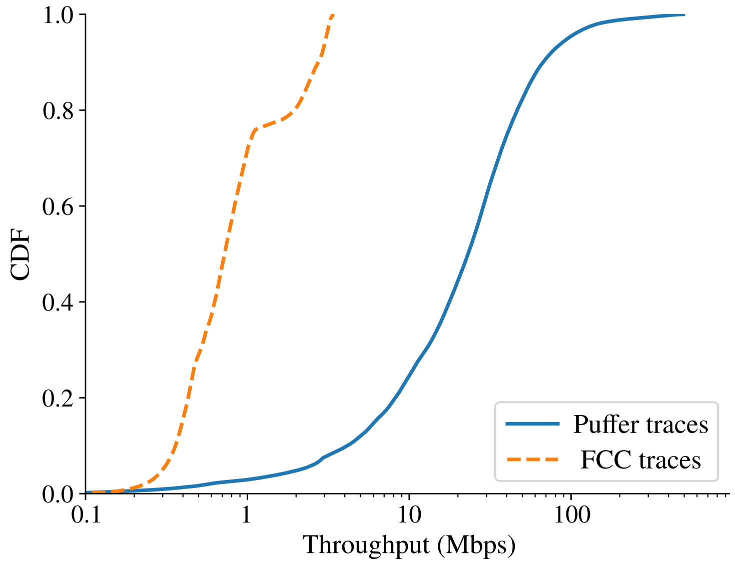
We trained a version of Fugu in this emulation environment to evaluate its performance. Compared with the *in situ* Fugu—or with every other ABR scheme—the real-world performance of emulation-trained Fugu was horrible (Figure 2.14b). Looking at the other ABR schemes, almost each of them lies somewhere along the SSIM/stall frontier in emulation (Figure 2.14a), with Pensieve rebuffering the least and MPC delivering the highest quality



(a) Performance in emulation, run in mahimahi [82] using the FCC traces [25], following the method of Pensieve [70].



(b) During Jan. 26–Apr. 2, 2019, we randomized sessions to a set of algorithms including “emulation-trained Fugu.” For Fugu, training in emulation did not generalize to the deployment environment. In addition, emulation results (left) are not indicative of real-world performance.



(c) Comparison of throughput distribution of FCC traces and of real Puffer sessions.

Figure 2.14: **Gap between simulation and reality (Puffer).**

video. In the real experiment (Figure 2.14b), we see a more muddled picture, with a different qualitative arrangement of schemes.

2.4.3 Remarks on Pensieve and RL for ABR

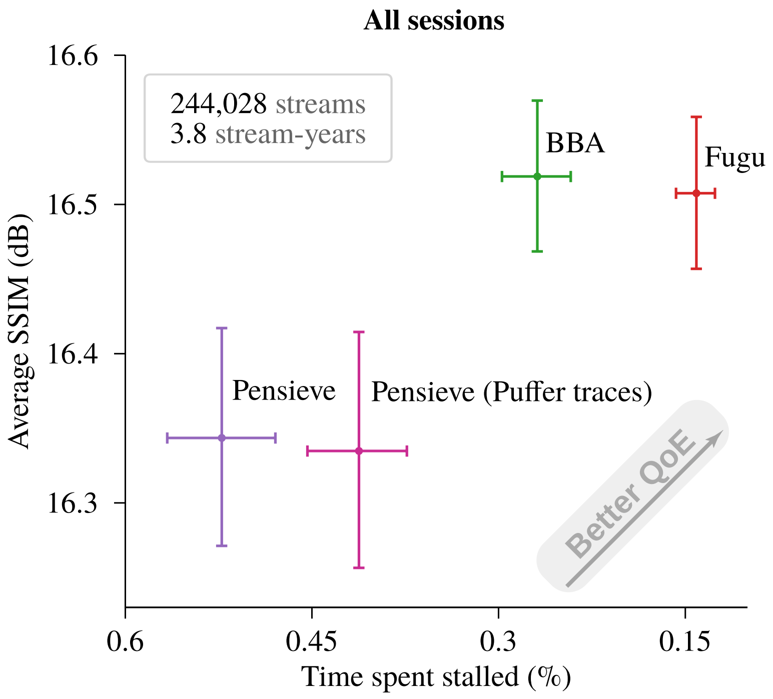
The original Pensieve paper [70] demonstrated that Pensieve outperformed MPC-HM, RobustMPC-HM, and BBA in both emulation-based tests and in video streaming tests on low and high-speed real-world networks. Our results differ; we believe the mismatch may have occurred for several reasons.

First, we have found that simulation-based training and testing do not capture the vagaries of the real-world paths seen in the Puffer study. Unlike real-world randomized trials, trace-based emulators and simulators allow experimenters to limit statistical uncertainty by running different algorithms on the same conditions, eliminating the effect of the play of chance in giving different algorithms a different distribution of watch times, network behaviors, etc. However, it is difficult to characterize the *systematic* uncertainty that comes from selecting a set of traces that may omit the variability or heavy-tailed nature of a real deployment experience (both network behaviors as well as user behaviors, such as watch duration).

Reinforcement learning (RL) schemes such as Pensieve may be at a particular disadvantage from this phenomenon. Unlike supervised learning schemes that can learn from training "data," RL typically requires a training *environment* to respond to a sequence of control decisions and decide on the appropriate consequences and reward. That environment could be real life instead of a simulator, but the level of statistical noise we observe would make this type of learning extremely slow or require an extremely broad deployment of algorithms in training. RL relies on being able to slightly vary a control action and detect a change in the resulting reward. By our calculations, the variability of inputs is such that it takes about 2 stream-years of data to reliably distinguish two ABR schemes whose innate "true" performance differs by 15%. To make RL practical, future work may need to explore techniques to reduce this variability [71] or construct more faithful simulators and emulators that model tail behaviors and capture additional dynamics of the real Internet that are not represented in throughput traces (e.g. varying RTT, cross traffic, interaction between throughput and chunk size [10]).

Second, most of the evaluation of Pensieve in the original paper focused on training and evaluating Pensieve using a single test video. As a result, the state space that model had to explore was inherently more limited. Evaluation of the Pensieve "multi-video model"—which we have to use for our experimental setting—was more limited. Our results are more consistent with a recent large-scale study of a Pensieve-multi-video-like scheme on 30 million streams at Facebook [69].

Third, Figure 2.14c shows that the distribution of throughputs in the FCC traces differs markedly from those on Puffer. This dataset shift could have harmed the performance of Pensieve, which was trained on the FCC traces. In response to reviewer feedback, we trained a version of Pensieve on throughput traces randomly sampled from real Puffer video sessions. This is essentially as close to a "learned *in situ*" version of Pensieve as we think we can achieve, but is not quite the same (§2.4.3). We compared "Pensieve on Puffer traces" with the original Pensieve, BBA, and Fugu between Jan. 2 and Feb. 2, 2020 (Figure 2.15). The results were broadly similar; the new Pensieve achieved better performance, but was still significantly worse than BBA and Fugu. The results deserve further study; they suggest that the representativeness of training data is not the end of the story when it comes to the



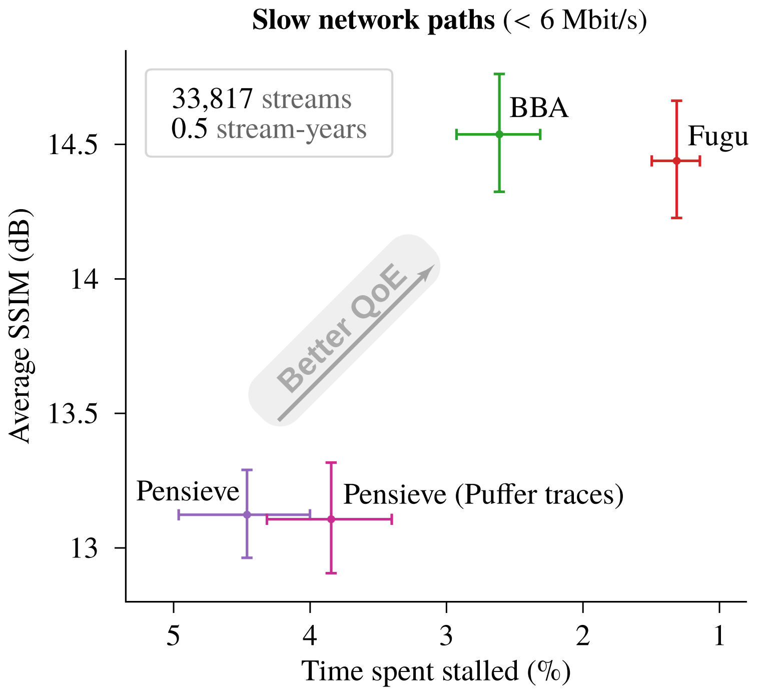


Figure 2.15: During Jan. 2–Feb. 2, 2020, we evaluated a version of Pensieve that was trained on a collection of network traces drawn randomly from actual Puffer sessions. This improved its performance compared with the original Pensieve, but the overall results were broadly similar.

real-world performance of RL schemes trained in simulation.

Finally, Pensieve optimizes a QoE metric centered around bitrate as a proxy for video quality. We did not alter this and leave the discussion to Section 2.5. Figure 2.7 shows that Pensieve was the #2 scheme in terms of bitrate (below BBA) in the primary analysis. We emphasize that our findings do not indicate that Pensieve cannot be a useful ABR algorithm, especially in a scenario where similar, pre-recorded video is played over a familiar set of known networks.

2.5 Limitations

The design of the Puffer experiment and the Fugu system are subject to important limitations that may affect their performance and generalizability.

2.5.1 Limitations of the experiments

Our randomized controlled trial represents a rigorous, but necessarily "black box," study of ABR algorithms for video streaming. We don't know the true distribution of network paths and throughput-generating processes; we don't know the participants or why the distribution in watch times differs by assigned algorithm; we don't know how to emulate these behaviors accurately in a controlled environment.

We have supplemented this black-box work with ablation analyses to relate the real-world performance of Fugu to the l^2 accuracy of its predictor, and have studied various ablated versions of Fugu in deployment. However, ultimately part of the reason for this dissertation is that we *cannot* replicate the experimental findings outside the real world—a real world whose behavior is noisy and takes lots of time to measure precisely. That may be an unsatisfying conclusion, and we doubt it will be the final word on this topic. Perhaps it will become possible to model enough of the vagaries of the real Internet "in silico" to enable the development of robust control strategies without extensive real-world experiments.

It is also unknown to what degree Puffer's results—which are about a single server in a university datacenter, sending to clients across our entire country over the wide-area Internet—generalize to a different server at a different institution, much less the more typical paths between a user on an access network and their nearest CDN edge node. We don't know for sure if the pre-trained Fugu model would work in a different location, or whether training a new Fugu based on data from that location would yield comparable results. Our results show that learning *in situ* works, but we don't know how specific the *situs* needs to be. And while we expect that Fugu could be implemented in the context of client-side ABR (especially if the server is willing to share its `tcp_info` statistics with the client), we haven't demonstrated this.

Although we believe that past research papers may have underestimated the uncertainties in real-world measurements with realistic Internet paths and users, we also may be guilty of underestimating our own uncertainties or emphasizing uncertainties that are only relevant to small or medium-sized academic studies, such as ours, and irrelevant to the industry. The current load on Puffer is about 60 concurrent streams on average, meaning we collect about 60 stream-days of data per day. Our primary analysis covers about 2.6 stream-years of data per scheme collected over an eight-month period, and was sufficient to measure its performance metrics to within about ±15% (95% CI). By contrast, we understand YouTube has an average load of more than 60 *million* concurrent streams at any given time. We imagine the considerations of conducting data-driven experiments at this level may be completely different—perhaps less about statistical uncertainty, and more about systematic uncertainties and the difficulties of running experiments and accumulating so much data.

Some of Fugu's performance (and that of MPC, RobustMPC, and BBA) relative to Pensieve may be due to the fact that these four schemes received more information as they ran—namely, the SSIM of each possible version of each future chunk—than did Pensieve. It is possible that an "SSIM-aware" Pensieve might perform better. The load of calculating SSIM for each encoded chunk is not insignificant—about an extra 40% on top of encoding the video.

2.5.2 Limitations of Fugu

There is a sense that data-driven algorithms that more "heavily" squeeze out performance gains may also put themselves at risk to brittleness when a deployment environment drifts from one where the algorithm was trained. In that sense, it is hard to say whether Fugu's performance might decay catastrophically some day. We tried and failed to demonstrate a quantitative benefit from daily retraining over "out-of-date" vintages, but at the same time, we cannot be sure that some surprising detail tomorrow—e.g., a new user from an unfamiliar network—won't send Fugu into a tailspin before it can be retrained. A year of data on a growing userbase suggests, but doesn't guarantee, robustness to a changing environment.

Fugu does not consider several issues that other research has concerned itself with—e.g., being able to "replace" already-downloaded chunks in the buffer with higher quality versions [109], or optimizing the joint QoE of multiple clients who share a congestion bottleneck [80].

Fugu is not tied as tightly to the TCP or congestion control as it might be—for example, Fugu could wait to send a chunk until the TCP sender tells it that there is a sufficient congestion window for most of the chunk (or the whole chunk) to be sent immediately. Otherwise, it *might* choose to wait and make a better-informed decision later. Fugu does not schedule the transmission of chunks—it will always send the next chunk as long as the client has room in its playback buffer.

Chapter 3

Pantheon: Training Ground for Congestion Control

3.1 Introduction

Despite thirty years of research, Internet congestion control and the development of transport-layer protocols remain cornerstone problems in computer networking. Congestion control was originally motivated by the desire to avoid catastrophic network collapses [52], but today it is responsible for much more: allocating capacity among contending applications, minimizing delay and variability, and optimizing high-level metrics such as video re-buffering, Web page load time, the completion of batch jobs in a datacenter, or users' decisions to engage with a website.

In the past, the prevailing transport protocols and congestion-control schemes were developed by researchers [44, 52] and tested in academic networks or other small testbeds before broader deployment across the Internet. Today, however, the Internet is more diverse, and studies on academic networks are less likely to generalize to, e.g., CDN nodes streaming video at 80 Gbps [81], smartphones on overloaded mobile networks [22], or security cameras connected to home Wi-Fi networks.

As a result, operators of large-scale systems have begun to develop new transport algorithms in-house. Operators can deploy experimental algorithms on a small subset of their live traffic (still serving millions of users), incrementally improving performance and broadening deployment as it surpasses existing protocols on their live traffic [5, 21, 63]. These results, however, are rarely reproducible outside the operators of large services.

Outside of such operators, research is usually conducted on a much smaller scale, still may not be reproducible, and faces its own challenges. Researchers often create a new testbed each time—interesting or representative network paths to be experimented over—and must fight "bit rot" to acquire, compile, and execute prior algorithms in the literature so they can be fairly compared against. Even so, results may not generalize to the wider Internet. Examples of this pattern in the academic literature include Sprout [125], Verus [129], and PCC [32].

This dissertation describes the **Pantheon**: a distributed, collaborative system for researching and evaluating end-to-end networked systems, especially congestion-control schemes, transport protocols, and network emulators. The Pantheon has four parts:

1. a software library containing a growing collection of transport protocols and congestion-control algorithms, each verified to compile and run by a continuous-integration system, and each exposing the same interface to start or stop a full-throttle flow,

2. a diverse testbed of network nodes on wireless and wired networks around the world, including cellular networks in Stanford (U.S.), Guadalajara (Mexico), São Paulo (Brazil), Bogotá (Colombia), New Delhi (India), and Beijing (China), and wired networks in all of the above locations as well as London (U.K.), Iowa (U.S.), Tokyo (Japan), and Sydney (Australia),

3. a collection of network emulators, each calibrated to match the performance of a real network path between two nodes, or to capture some form of pathological network behavior, and

4. a continuous-testing system that regularly evaluates the Pantheon protocols over the real Internet between pairs of testbed nodes, across partly-wireless and all-wired network paths, and over each of the network emulators, in single- and multi-flow scenarios, and publicly archives the resulting packet traces and analyses at https://pantheon.stanford.edu.

The Pantheon's calibrated network emulators address a tension that protocol designers face between experimental realism and reproducibility. Simulators and emulators are reproducible and allow rapid experimentation, but may fail to capture important dynamics of real networks [41, 42, 87]. To resolve this tension, the Pantheon generates network emulators calibrated to match real Internet paths, graded by a novel figure of merit: their accuracy in matching the performance of a set of transport algorithms. Rather than focus on the presence or absence of modeled phenomena (jitter, packet loss, reordering), this metric describes how well the end-to-end performance (e.g., throughput, delay, and loss rate) of a set of algorithms, run over the emulated network, matches the corresponding performance statistics of the same algorithms run over a real network path.

Motivated by the success of ImageNet [31] and OpenAI Gym [18] in the machine learning community, we believe a common reference set of runnable benchmarks, continuous experimentation and improvement, and a public archive of results will enable faster innovation and more effective, reproducible research. Early adoption by independent research groups provides encouraging evidence that this is succeeding.

Summary of results:

- Examining more than a year of measurements from the Pantheon, we find that transport performance is highly variable across the type of network path, bottleneck network, and time. There is no single existing protocol that performs well in all settings. Furthermore, many protocols perform differently from how their creators intended and documented (§3.4).

- We find that a small number of network-emulator parameters (bottleneck link rate, isochronous or memoryless packet inter-arrival timing, bottleneck buffer size, stochastic per-packet loss rate, and propagation delay) is sufficient to replicate the performance of a diverse library of transport protocols (with each protocol matching its real-world throughput and delay to within 17% on average), in the presence of both natural and synthetic cross traffic. These results go against some strains of thought in computer networking, which have focused on building detailed network emulators (with mechanisms to model jitter, reordering, the arrival and departure of cross traffic, MAC dynamics, etc.), while leaving the questions open of how to configure an emulator to accurately model real networks and how to quantify the emulator's overall fidelity to a target (§3.5).

- We discuss new approaches to congestion control that are using the Pantheon as a shared evaluation testbed, giving us encouragement that it will prove useful as a community resource. Four are from research groups distinct from the present authors: Copa [6] and Vivace [33] published at NSDI 2018, Aurora [53] published at ICML 2019, and FillP [64] to appear in SIGCOMM 2020 (§3.6).
- We also describe our own learning-based congestion control, Indigo, based on neural networks that can be trained on a collection of the Pantheon's emulators and in turn achieve good performance over real Internet paths (§3.7).

3.2 Related work

Pantheon benefits from a decades-long body of related work in Internet measurement, network emulation, transport protocols, and congestion-control schemes.

Tools for Internet measurement. Systems like PlanetLab [24], Emulab [123], and ORBIT [86] provide measurement nodes for researchers to test transport protocols and other end-to-end applications. PlanetLab, which was in wide use from 2004–2012, at its peak included hundreds of nodes, largely on well-provisioned (wired) academic networks around the world. Emulab allows researchers to run experiments over configurable network emulators and on Wi-Fi links within an office building.

While these systems are focused on allowing researchers to borrow nodes and run their own tests, the Pantheon operates at a higher level of abstraction. Pantheon includes a *single* community software package that researchers can contribute algorithms to. Anybody can run any of the algorithms in this package, including over Emulab or any network path, but Pantheon also hosts a common repository of test results (including raw packet traces) of scripted comparative tests.

Network emulation. Early congestion-control research relied heavily on simulation, e.g., through the ns-2 simulator [84]. When the Internet was simpler, largely wired, and devoid of middleboxes, the community adopted recommendations for simulation parameters to accurately capture the network's behavior [39]. Shared experimental methods and metrics helped congestion-control research flourish, giving a reproducibility that contributed to TCP's robustness and prevalence. However, as the Internet's complexity grew, the limitations of ns-2 became apparent, especially as application protocols became more complex and simulating a TCP flow became insufficient. Numerous network emulators allow a real application or protocol implementation to interact with a software emulator as if it is a real network [20, 28, 38, 47, 48, 82, 92, 93, 120, 123].

These emulators provide increasing numbers of parameters and mechanisms to recreate different network behaviors, such as traffic shapers, policers, queue disciplines, stochastic i.i.d. or autocorrelated loss, reordering, bit errors, and MAC dynamics. However, properly setting these parameters to emulate a particular target network remains an open problem.

One line of work has focused on improving emulator precision in terms of the level of detail and fidelity at modeling small-scale effects (e.g., "Two aspects influence the accuracy of an emulator: how detailed is the model of the system, and how closely the hardware and software can reproduce the timing computed by the model" [20]). Pantheon takes a different approach, instead focusing on accuracy in terms of how well an emulator recreates the performance of a set of transport algorithms.

Congestion control. Internet congestion control has a deep literature. The original DECBit [91] and Tahoe [52] algorithms responded to one-bit feedback from the network. As wired network speeds increased, operating systems transitioned to hybrid schemes that could also handle high bandwidth-delay product networks (very large window sizes), such as Cubic [44] and Compound TCP [116]. All of these schemes rely on the concept of "TCP friendliness" [40]. In this model, packet loss is predominantly due to queue drops and so should be taken as a congestion signal. While early wireless LANs violated this assumption, leading to wireless-specific congestion control schemes [9], wireless LANs today have high enough link reliability that these schemes are not used today.

Since this seminal work, two classes of applications and networks have emerged that lead protocols to rely on a different congestion signal: delay. First, applications such as the macOS update downloader and BitTorrent wish to use as much spare capacity as possible without increasing the latency of standard TCP flows. LEDBAT, used by both systems, uses delay as a measure of congestion, under the assumption that as long as there is a steady send rate, filling queues does not improve throughput [104]. Second, mobile networks provide more variable links than wired networks, along with deeper queues. This led to the development of congestion control schemes, such as Sprout [125] and Verus [129], that use delay as a signal to fully utilize a link without unnecessarily increasing delay. SCReAM adds TCP's self-clocking to LEDBAT to support conversational video over LTE [56].

Remy [124] and PCC [32] are different kinds of "learned" schemes. Remy uses an offline optimizer that generates a decision tree to optimize a user-defined utility score, calculated at the end of a network simulation. We evaluated a Remy-designed congestion control scheme (the "100x" Tao algorithm [106] that was trained in simulation, labeled "Tao" on our plots) as part of our experiments. While it performed well in emulation (Appendix D), it did not yield consistently good performance across real-world paths.

PCC is based on an online hill-climbing algorithm that uses randomized trials to optimize a utility function balancing throughput and packet loss [32]. Since PCC learns its behavior online, its optimization procedure accounts only for locally-observable phenomena. Moreover, its hill-climbing algorithm can become trapped in local maxima far from optimal (a phenomenon we observe on real-world links; §3.4).

In our current work (§3.7), we ask whether it is possible to quickly train an algorithm from first principles to produce good *global* performance on real Internet paths.

3.3 Pantheon: design and implementation

This section describes the design and implementation of the Pantheon, a system that automatically measures the performance of many transport protocols and congestion-control schemes across a diverse set of network paths. By allowing the community to repeatably evaluate transport algorithms in scripted comparative tests across real-world network paths, posted to a public archive of results, the Pantheon aims to help researchers develop and test algorithms more rapidly and reproducibly.

Below, we demonstrate several uses for Pantheon: comparing existing congestion-control schemes on real-world networks (§3.4); calibrating network emulators that accurately reproduce real-world performance (§3.5); and designing and testing new congestion-control schemes (§3.6 and §3.7).

3.3.1 Design overview

Pantheon has three components: (1) a software repository containing pointers to transport-protocol implementations, each wrapped to expose a common testing interface based on the abstraction of a full-throttle flow; (2) testing infrastructure that runs transport protocols in scripted scenarios, instruments the network to log when each packet was sent and received, and allows flows to be initiated by nodes behind a network address translator (NAT); and (3) a global observatory of network nodes, enabling measurements across a wide variety of paths. We describe each in turn.

(1) A collection of transport algorithms, each exposing the same interface

To test each transport protocol or congestion-control scheme on equal footing, Pantheon requires it to expose a common abstraction for testing: a full-throttle flow that runs until a sender process is killed. The simplicity of this interface has allowed us (and a few external contributors so far) to write simple wrappers for a variety of schemes and contribute them to the Pantheon, but limits the kinds of evaluations the system can do.[1]

Figure 3.1 lists the currently supported schemes, plus the size (in lines of code) of a wrapper script to expose the required abstraction. For all but three schemes, no modification was required to the existing implementation. The remaining three had a hard-coded MTU size and required a small patch to adjust it for compatibility with our network instrumentation; please see (2) below.

As an example, we describe the Pantheon's wrapper to make WebRTC expose the interface of a full-throttle flow. The Pantheon tests the Chromium implementation of

[1]For example, the interface allows measurements of combinations of long-running flows (with timed events to start and stop a flow), but does not allow the caller to run a scheme until it has transferred exactly x bytes. This means that the Pantheon cannot reliably measure the flow-completion time of a mix of small file transfers.

Label	Scheme	LoC
Copa	Copa [6]	46
LEDBAT	LEDBAT/μTP [104] (`libutp`)	48
PCC	PCC† [32]	46
QUIC	QUIC Cubic [63] (`proto-quic`)	119
SCReAM	SCReAM [56]	541
Sprout	Sprout† [125]	46
Tao	RemyCC “100x” (2014) [106]	43
BBR	TCP BBR [21]	52
Cubic	TCP Cubic [44] (Linux default)	30
Vegas	TCP Vegas [17]	50
Verus	Verus† [129]	43
WebRTC	WebRTC media [14] in Chromium	283
—	Vivace [33]	37
—	Aurora [53]	39
—	FillP [64]	41
Indigo	LSTM neural network	35

Table 3.1: The Pantheon’s transport schemes and the labels used for them in figures in this dissertation. Shown are the number of lines of Python, C++, or JavaScript code in each wrapper that implements the common abstraction. Schemes marked † are modified to reduce MTU.

WebRTC media transfer [14] to retrieve and play a video file. The wrapper starts a Chromium process for the sender and receiver, inside a virtual X frame buffer, and provides a signaling server to mediate the initial connection. This comprises about 200 lines of JavaScript.

Pantheon is designed to be easily extended; researchers can add a new scheme by submitting a pull request that adds a submodule reference to their implementation and the necessary wrapper script. Pantheon uses a continuous-integration system to verify that each proposed scheme builds and runs in emulation.

(2) Instrumenting network paths

For each IP datagram sent by the scheme, Pantheon’s instrumentation tracks the size, time sent, and (if applicable) time received. Pantheon allows either side (sender or receiver) to initiate the connection, even if one of them is behind a NAT, and prevents schemes from communicating with nodes other than the sender and receiver. To achieve this, Pantheon creates a virtual private network (VPN) between the endpoints, called a *Pantheon-tunnel*, and runs all traffic over this VPN.

Pantheon-tunnel comprises software controlling a virtual network device (TUN) [119] at each endpoint. The software captures all IP datagrams sent to the local TUN, assigns each a unique identifier (UID), and logs the UID and a timestamp. It then encapsulates the packet and its UID in a UDP datagram, which it transmits to the other endpoint via the path under test. The receiving endpoint decapsulates, records the UID and arrival time, and delivers the packet to its own Pantheon-tunnel TUN device.

This arrangement has two main advantages. First, UIDs make it possible to unambiguously log information about every packet (e.g., even if packets are retransmitted or contain identical payloads). Second, either network endpoint can be the sender or receiver of an instrumented network flow over an established Pantheon-tunnel, even if it is behind a NAT (as long as one endpoint has a routable IP address to establish the tunnel).

Pantheon-tunnel also has disadvantages. First, encapsulation costs 36 bytes (for the UID and headers), reducing the MTU of the virtual interface compared to the path under test; for schemes that assume a fixed MTU, Pantheon patches the scheme accordingly. Second, because each endpoint records a timestamp to measure the send and receive time of each datagram, accurate timing requires the endpoints' clocks to be synchronized; endpoints use NTP [85] for this purpose. Finally, Pantheon-tunnel makes all traffic appear to the network as UDP, meaning it cannot measure the effect of a network's discrimination based on the IP protocol type.[2]

Evaluation of Pantheon-tunnel. To verify that Pantheon-tunnel does not substantially alter the performance of transport protocols, we ran a calibration experiment to measure the tunnel's effect on the performance of three TCP schemes (Cubic, Vegas, and BBR). We ran each scheme 50 times inside and outside the tunnel for 30 seconds each time, between a colocation facility in India and the EC2 India datacenter, measuring the mean

[2]Large-scale measurements by Google [63] have found such discrimination, after deployment of the QUIC UDP protocol, to be rare.

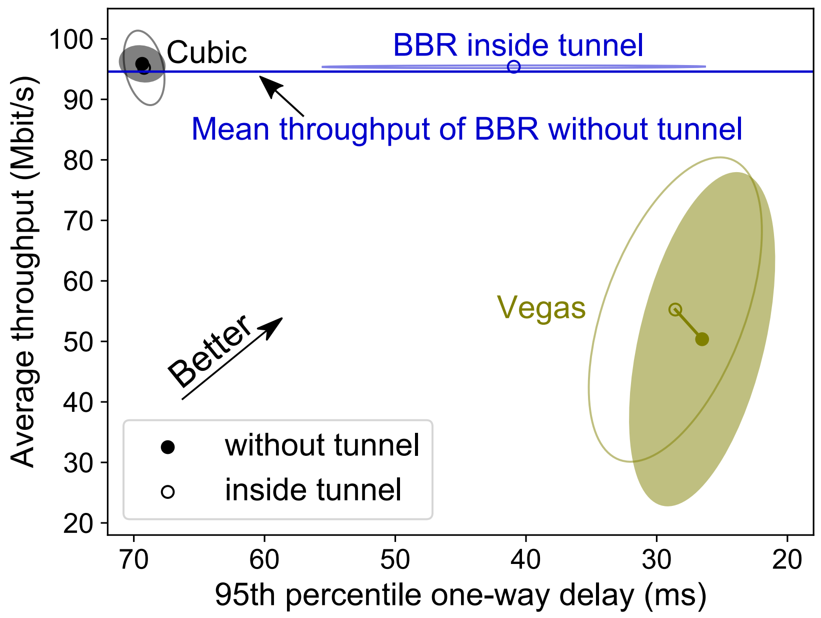


Figure 3.1: Pantheon-tunnel does not substantially impact the performance of congestion-control algorithms.

throughput and 95th-percentile per-packet one-way delay of each run.[3] We ran a two-sample Kolmogorov-Smirnov test for each pair of statistics (the 50 runs inside vs. outside the tunnel for each scheme's throughput and delay). No test found a statistically significant difference below $p < 0.2$ (Figure 3.1).

(3) A testbed of nodes on interesting networks

We deployed observation nodes in countries around the world, including cellular (LTE/UMTS) networks in Stanford (USA), Guadalajara (Mexico), São Paulo (Brazil), Bogotá (Colombia), New Delhi (India), and Beijing (China), wired networks in all of the above locations as well as London (U.K.), Iowa (U.S.), Tokyo (Japan), and Sydney (Australia), and a Wi-Fi mesh network in Nepal. These nodes were provided by a commercial colocation facility (Mexico, Brazil, Colombia, India), by volunteers (China and Nepal), or by Google Compute Engine (U.K., U.S., Tokyo, Sydney).

[3]For BBR running outside the tunnel, we were only able to measure the average throughput (not delay). Run natively, BBR's performance relies on TCP segmentation offloading [23], which prevents a precise measurement of per-packet delay without the tunnel's encapsulation.

We found that hiring a commercial colocation operator to maintain LTE service in far-flung locations has been an economical and practical approach; the company maintains, debugs, and “tops up” local cellular service in each location in a way that would otherwise be impractical for a university research group. However, this approach limits us to available colocation sites and ones where we receive volunteered nodes. We are currently bringing up a volunteered node with cellular connectivity in Saudi Arabia and welcome further contributions.

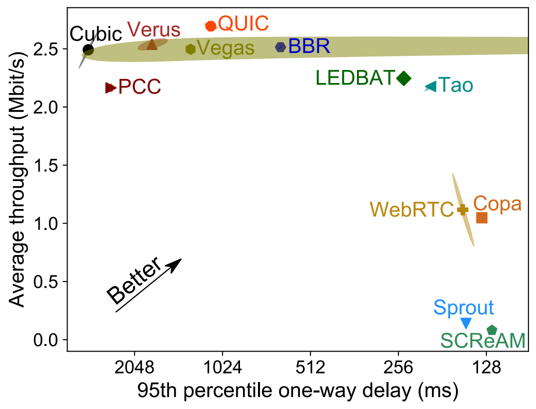
3.3.2 Operation and testing methods

The Pantheon frequently benchmarks its stable of congestion-control schemes over each path to create an archive of real-world network observations. On each path, Pantheon runs multiple benchmarks per week. Each benchmark follows a software-defined scripted workload (e.g., a single flow for 30 seconds; or multiple flows of cross traffic, arriving and departing at staggered times), and for each benchmark, Pantheon chooses a random ordering of congestion-control schemes, then tests each scheme in round-robin fashion, repeating until every scheme has been tested 10 times (or 3 for partly-cellular paths). This approach mirrors the evaluation methods of prior academic work ([32,125,129]).

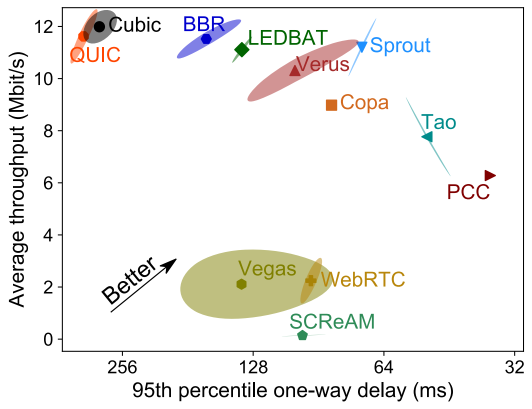
During an experiment, both sides of a path repeatedly measure their clock offset to a common NTP server and use these to calculate a corrected one-way delay of each packet. After running an experiment, a node calculates summary statistics (e.g., mean throughput, loss rate, and 95th-percentile one-way delay for each scheme) and uploads its logs (packet traces, analyses, and plots) to AWS S3 and the Pantheon website (`https://pantheon.stanford.edu`).

3.4 Findings

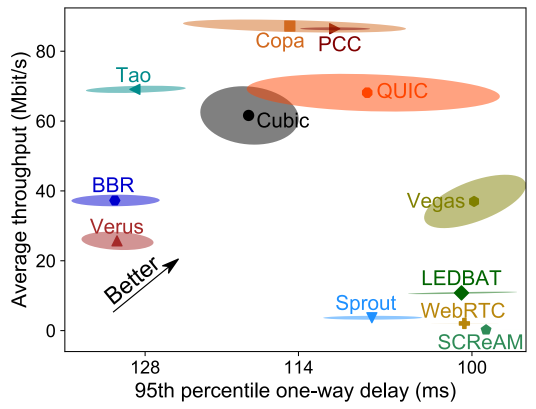
The Pantheon has collected and published measurements of a dozen protocols taken over the course of more than a year. In this section, we give a high-level overview of some key findings in this data, focusing on the implications for research and experimental methodology. We examine comparative performance between protocols rather than the detailed behavior of



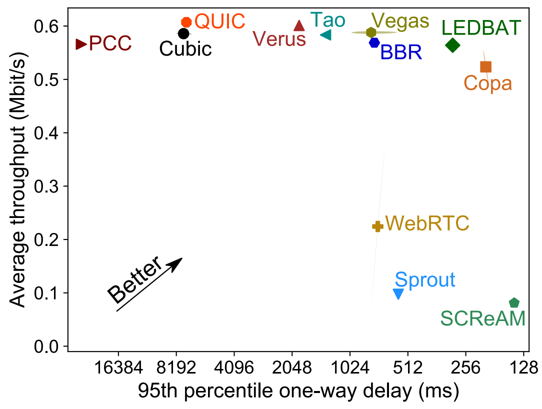
(a) AWS Brazil to Colombia (cellular), 1 flow, 3 trials. P1392.



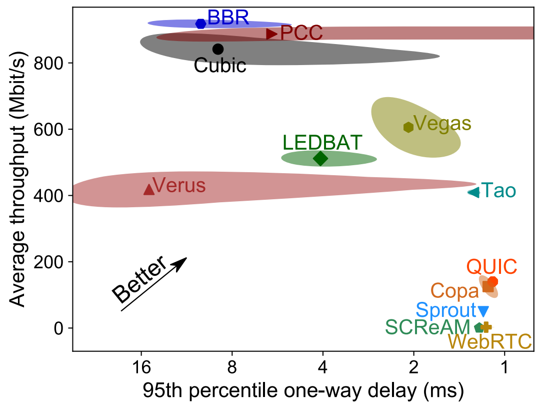
(b) Stanford to AWS California (cellular), 1 flow, 3 trials. P950.



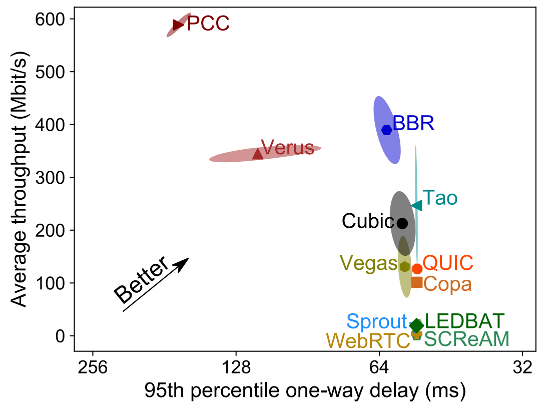
(c) AWS Brazil to Colombia (wired), 1 flow, 10 trials. P1271.



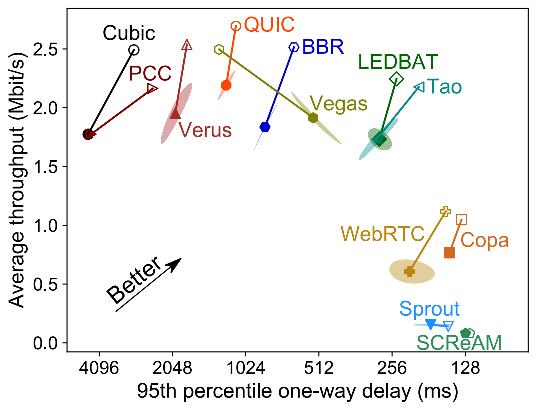
(d) Colombia to AWS Brazil (cellular), 1 flow, 3 trials. P1391.



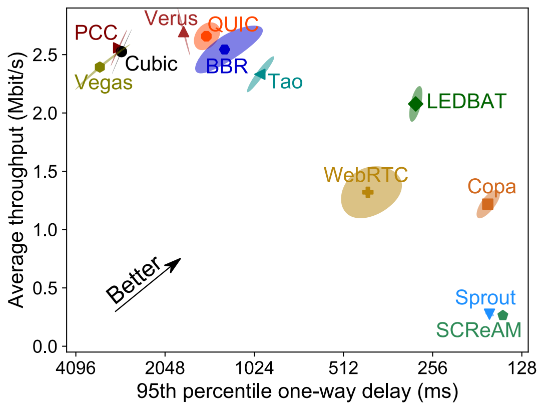
(e) Stanford to AWS California (wired), 3 flows, 10 trials. P1238.



(f) GCE Tokyo to GCE Sydney (wired), 3 flows, 10 trials. P1442.



(g) AWS Brazil to Colombia (cellular), 1 flow, 3 trials. 2 days after Figure 3.2a (shown in open dots). P1473.



(h) AWS Brazil to Colombia (cellular), 3 flows, 3 trials. P1405.

Figure 3.2: Compared with Figure 3.2a, scheme performance varies across the type of network path (Figure 3.2c), number of flows (Figure 3.2h), time (Figure 3.2g), data flow direction (Figure 3.2d), and location (Figure 3.2b). Figure 3.2e and 3.2f show that the variation is not limited to just cellular paths. The shaded ellipse around a scheme's dot represents the 1-σ variation across runs. Given a measurement ID, e.g. P123, the full result can be found at `https://pantheon.stanford.edu/result/123/`.

particular protocols, because comparative analyses provide insight into which protocol end hosts should run in a particular setting.

To ground our findings in examples from concrete data, we select one particular path: AWS Brazil to Colombia. This path represents the performance a device in Colombia would see downloading data from properly geo-replicated applications running in AWS (Brazil is the closest site).

Finding 1: Which protocol performs best varies by path. Figure 3.2a shows the throughput and delay of 12 transport protocols from AWS Brazil to a server in Colombia, with an LTE modem from a local carrier (Claro).[4] Figure 3.2b shows the throughput and delay for the same protocols from a node at Stanford University with a T-Mobile LTE modem, to a node in AWS California. The observed performance varies significantly. In Brazil-Colombia, PCC is within 80% of the best observed throughput (QUIC) but with

[4]All results in this dissertation and supporting raw data can be found in the Pantheon archive; e.g. the experiment indicated as P123 can be found at `https://pantheon.stanford.edu/result/123/`.

delay 20 times higher than the lowest (SCReAM). In contrast, for Stanford-California, PCC has only 52% of the best observed throughput (Cubic) and the lowest delay. The Sprout scheme, developed by one of the present authors, was designed for cellular networks in the U.S. and performs well in that setting (Figure 3.2b), but poorly on other paths.

These differences are not only due to long haul paths or geographic distance. Figure 3.2c shows the performance of the transport protocols from AWS Brazil to a wired device in Colombia. Performance is completely different. Delays, rather than varying by orders of magnitude, differ by at most 32%. At the same time, some protocols are strictly better: QUIC (Cubic) and (TCP) Cubic have both higher throughput and lower delay than BBR and Verus.

Differences are not limited to paths with cellular links. Figure 3.2e shows performance between Stanford and AWS California using high-bandwidth wired links and Figure 3.2f shows performance between the Google Tokyo and Sydney datacenters. While in both cases PCC shows high throughput and delay, in the AWS case BBR is better in throughput while between Google data centers it provides 34% less throughput. Furthermore, LEDBAT performs reasonably well on AWS, but has extremely low throughput between Google datacenters.

This suggests that evaluating performance on a small selection (or, in the worst case, just one) of paths can lead to misleadingly positive results, because they are not generalizable to a wide range of paths.

Finding 2: Which protocol performs best varies by path direction. Figure 3.2d shows the performance of the opposite direction of the path, from the same device with cellular connection in Colombia to AWS Brazil. This configuration captures the observed performance of uploading a photo or streaming video through a relay.

In the Brazil to Colombia direction, QUIC strictly dominates Vegas, providing both higher throughput and lower delay. In the opposite direction, however, the tradeoff is less clear: Vegas provides slightly lower throughput with a significant (factor of 9) decrease in delay. Similarly, in the Brazil to Colombia direction, WebRTC provides about half the throughput of LEDBAT while also halving delay; in the Colombia to Brazil direction, WebRTC is strictly worse, providing one third the throughput while quadrupling delay.

This indicates that evaluations of network transport protocols need to explicitly measure both directions of a path. On the plus side, a single path can provide two different sets of conditions when considering whether results generalize.

Finding 3: Protocol performance varies in time and only slightly based on competing flows. Figure 3.2g shows the Brazil-Colombia path measured twice, separated by two days (the first measurement shown in open dots is the same as in Figure 3.2a). Most protocols see a strict degradation of performance in the second measurement, exhibiting lower throughput and higher delay. Cubic and PCC, once clearly distinguishable, merge to have equivalent performance. More interestingly, the performance of Vegas has 23% lower throughput, but cuts delay by more than a factor of 2.

Finally, Figure 3.2h shows performance on the Brazil-Colombia path when 3 flows compete. Unlike in Figure 3.2a, PCC and Cubic dominate Vegas, and many protocols see similar throughput but at greatly increased latency (perhaps due to larger queue occupancy along the path).

This indicates that evaluations of network transport protocols need to not only measure a variety of paths, but also spread those measurements out in time. Furthermore, if one protocol is measured again, all of them need to be measured again for a fair comparison, as conditions may have changed. Cross traffic (competing flows) is an important consideration, but empirically has only a modest effect on relative performance. We do find that schemes that diverge significantly from traditional congestion control (e.g., PCC) exhibit poor fairness in some settings; in a set of experiments between Tokyo and Sydney (P1442), we observed the throughput ratios of three PCC flows to be 32:4:1. This seems to contradict fairness findings in the PCC paper and emphasizes the need for a shared evaluation platform across diverse paths.

3.5 Calibrated emulators

The results in Section 3.4 show that transport performance varies significantly over many characteristics, including time. This produces a challenge for protocol development and the ability of researchers to reproduce each others' results. One time-honored way to achieve controlled, reproducible results, at the cost of some realism, is to measure protocols in simulation or emulation [39] instead of the wild Internet, using tools like Dummynet [20,93], NetEm [48], Mininet [47], or Mahimahi [82].

These tools each provide a number of parameters and mechanisms to recreate different network behaviors, and there is a traditional view in computer networking that the more fine-grained and detailed an emulator, the better. The choice of parameter values to faithfully emulate a particular target network remains an open problem.

In this dissertation, we propose a new figure of merit for network emulators: the degree to which an emulator can be substituted for the real network path in a full system, including the endpoint algorithm, without altering the system's overall performance. In particular, we define the emulator's accuracy as the average difference of the throughput and of the delay of a set of transport algorithms run over the emulator, compared with the same statistics from the real network path that is the emulator's target. The broader and more diverse the set of transport algorithms, the better characterized the emulator's accuracy will be: each new algorithm serves as a novel probe that could put the network into an edge case or unusual state that exercises the emulator and finds a mismatch.

In contrast to some conventional wisdom, we do not think that more-detailed network models are necessarily preferable. Our view is that this is an empirical question, and that more highly-parameterized network models create a risk of overfitting—but may be justified if lower-parameter models cannot achieve sufficient accuracy.

3.5.1 Emulator characteristics

We found that a five-parameter network model is sufficient to produce emulators that approximate a diverse variety of real paths, matching the throughput and delay of a range of algorithms to within 17% on average. The resulting calibrated emulators allow researchers to test experimental new schemes—thousands of parallel variants if necessary—in emulated environments that stand a good chance of predicting future real-world behavior.[5]

The five parameters are:

[5] In a leave-one-out cross-validation experiment, we confirmed that emulators trained to match the performance of $n - 1$ transport algorithms accurately predicted the unseen scheme's performance within about 20% (results not shown).

1. a bottleneck link rate,
2. a constant propagation delay,
3. a DropTail threshold for the sender's queue,
4. a stochastic loss rate (per-packet, i.i.d.), and
5. a bit that selects whether the link runs isochronously (all interarrival times equal), or with packet deliveries governed by a memoryless Poisson point process, characteristic of the observed behavior of some LTE networks [125].

To build emulators using these parameters, the Pantheon uses Mahimahi container-based network emulators [82]. In brief: Mahimahi gives the sender and receiver each its own isolated Linux network namespace, or container, on one host. An emulator is defined by a chain of nested elements, each one modeling a specific network effect: e.g., an `mm-loss` container randomly drops packets in the outgoing or incoming direction at a specified rate.

3.5.2 Automatically calibrating emulators to match a network path

Given a set of results over a particular network path, Pantheon can generate an emulator that replicates the same results in about two hours, using an automated parameter-search process that we now describe.

To find an appropriate combination of emulator parameters, Pantheon searches the space using a non-linear optimization process that aims to find the optimal value for a vector x, which represents the <*rate*, *propagation delay*, *queue size*, *loss rate*> for the emulator.[6]

The optimization derives a replication error for each set of emulator parameters, $f(x)$, which is defined as the average of the percentage changes between the real and emulated mean throughput, and the real and emulated mean 95th-percentile delay, across each of the set of reference transport algorithms. To minimize $f(x)$, nonlinear optimization is necessary because neither the mathematical expression nor the derivative of $f(x)$ is known. In addition, for both emulated and real world network paths, $f(x)$ is non-deterministic and noisy.

[6]The optimization is run twice, to choose between a constant rate or a Poisson delivery process.

The Pantheon uses Bayesian optimization [74], a standard method designed for optimizing the output of a noisy function when observations are expensive to obtain and derivatives are not available.[7] The method starts with the assumption that the objective function, $f(x)$, is drawn from a broad prior (Gaussian is a standard choice and the one we use). Each sample (i.e., calculation of the emulator replication error for a given set of emulator parameters x) updates the posterior distribution for $f(x)$. Bayesian optimization uses an *acquisition function* to guide the algorithm's search of the input space to the next value x. We use the Spearmint [107] Bayesian-optimization toolkit, which uses "expected improvement" as its acquisition function. This function aims to maximize the expected improvement over the current best value [74].

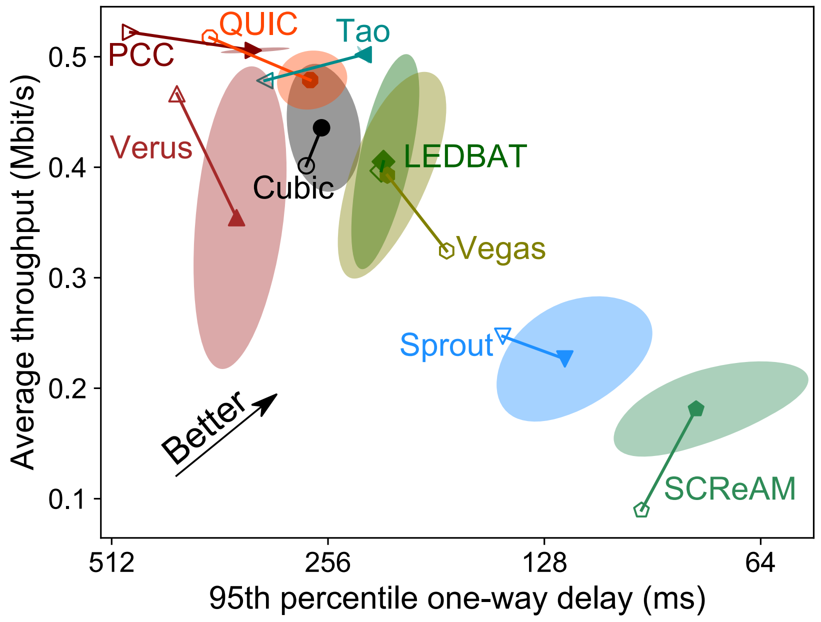
3.5.3 Emulation results

We trained emulators that model six of Pantheon's paths, each for about 2 hours on 30 EC2 machines with 4 vCPUs each. Figure 3.3 shows per-scheme calibration results for two representative network paths, a wireless device in Nepal and a wired device in Mexico. Filled dots represent the measured *mean* performance of the scheme on the real network path, while the open dot represents the performance on the corresponding calibrated emulator. A closer dot means the emulator is better at replicating that scheme's performance.

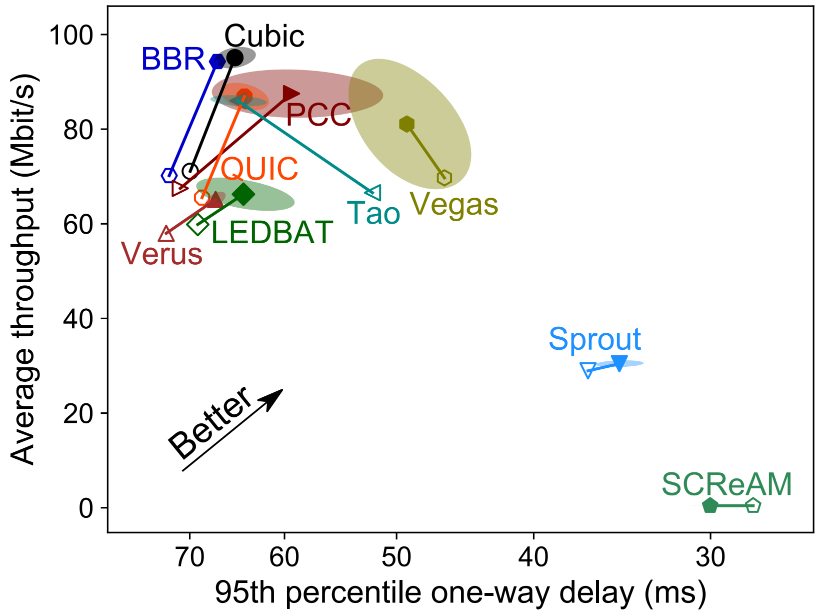
We observe that the emulators roughly preserve the relative order of the mean performance of the schemes on each path. Table 3.2 shows mean error in replicating the throughput and delay performance of all of Pantheon's congestion-control schemes by a series of emulators. To ensure each parameter is necessary, we measured the benefits of adding delay, queue size, and loss information to a base emulator that uses a constant rate, in replicating the China wired device path. For the cellular device path we measured the benefit of using a Poisson based link rate rather than a constant rate. As shown in Table 3.3, each added parameter improves the emulator's fidelity.

Pantheon includes several calibrated emulators, and regularly runs the transport algorithms in single- and multi-flow scenarios over each of the emulators and publishes the

[7]Each evaluation of $f(x)$ involves running all of Pantheon's congestion-control schemes in a scripted 30-second scenario, three times, across the emulated path. This is done in parallel, so each evaluation of $f(x)$ takes about 30 seconds of wall-clock time.



(a) Nepal to AWS India (wireless), 1 flow, 10 trials. Mean replication error: 19.1%. P188.



(b) AWS California to Mexico (wired), 3 flows, 10 trials. Mean replication error: 14.4%. P1237.

Figure 3.3: Examples of per-scheme calibrated emulator errors. The filled dots represent real results over each network path; the open dots represent the corresponding result over the emulator that best replicates all of the results. Emulators for all-wired paths give better fidelity than emulators for partly-wireless paths (§3.5.3).

Path	Error (%)
Nepal to AWS India (Wi-Fi, 1 flow, P188)	19.1
AWS Brazil to Colombia (cellular, 1 flow, P339)	13.0
Mexico to AWS California (cellular, 1 flow, P196)	25.1
AWS Korea to China (wired, 1 flow, P361)	17.7
India to AWS India (wired, 1 flow, P251)	15.6
AWS California to Mexico (wired, 1 flow, P353)	12.7
AWS California to Mexico (wired, 3 flows, P1237)	14.4

Table 3.2: Replication error of calibrated emulators on six paths with a single flow, and one path with three flows of staggered cross traffic.

Path	Feature change	Error (%)
China wired	link rate only	211.8
	add delay	211.8 → 189.7
	add buffer size	189.7 → 32.3
	add stochastic loss	32.3 → 17.7
Colombia cellular	constant → Poisson	23.7 → 13.0

Table 3.3: Each of the emulator's five parameters is helpful in reducing replication error. For the China wired path, we started with a single parameter and added the other three features one by one, in the order of their contribution. The Colombia cellular path required jitter (Poisson deliveries) to achieve good accuracy.

results in its public archive. Researchers are also able to run the calibrated emulators locally.

In addition, Pantheon includes, and regularly evaluates schemes over, a set of "pathological" emulators suggested by colleagues at Google. These model extreme network behaviors seen in the deployment of the BBR scheme: very small buffer sizes, severe ACK aggregation, and token-bucket policers.

Overall, our intention is that Pantheon will contain a sufficient library of well-understood network emulators so that researchers can make appreciable progress evaluating schemes (perhaps thousands of variants at once) in emulation—with some hope that there will be fewer surprises when a scheme is evaluated over the real Internet.

3.6 Pantheon use cases

We envision Pantheon as a common evaluation platform and an aid to the development of new transport protocols and congestion-control schemes. In this section, we describe the experiences that other research groups have had using Pantheon to assist their efforts.

Case 1. Vivace: validating a new scheme in the real world. Dong et al. [33] describe a new congestion-control scheme called Vivace, the successor to PCC [32]. They contributed three variants of the scheme to Pantheon in order to evaluate and tune Vivace's performance, by examining Pantheon's packet traces and analyses of Vivace in comparison with other schemes across an array of real-world paths. This is consistent with Pantheon's goal of being a resource for the research community (§1).

Case 2. Copa and FillP: iterative design with measurements. Arun and Balakrishnan [6] describe another new scheme, Copa, which optimizes an objective function via congestion window and sending rate adjustments. In contrast to Vivace, which was deployed on Pantheon largely as a completed design, Copa used Pantheon as an integral part of the design process: the authors deployed a series of six prototypes, using Pantheon's measurements to inform each iteration. Li et al. [64] improved wireless transport performance with FillP and used Pantheon, especially the wireless paths, in the same way as Copa did. They demonstrate another use of Pantheon, automatically deploying and testing prototypes on the real Internet, and gathering *in vivo* performance data.

Case 3. Aurora: evaluating a reinforcement learning scheme. Jay et al. [53] apply reinforcement learning to generate a congestion-control scheme called Aurora. They added the scheme to Pantheon's reference set and performed an extensive evaluation using Pantheon's network emulators.

3.7 Indigo: extracting an algorithm from data

As an extreme example of data-driven design, we present Indigo, a machine-learned congestion-control scheme whose design we extract from data gathered by Pantheon.

Using machine learning to train a congestion-control scheme for the real world is challenging. The main reason is that it is impractical to learn directly from the Internet: machine-learning algorithms often require thousands of iterations and hours to weeks of training time, meaning that paths evolve in time (§3.4) more quickly than the learning algorithm can converge. Pantheon's calibrated emulators (§3.5) provide an alternative: they are reproducible, can be instantiated many times in parallel, and are designed to replicate the behavior of congestion-control schemes. Thus, our high-level strategy is to train Indigo using emulators, then evaluate it in the real world using Pantheon.

Indigo is one example of what we believe to be a novel family of data-driven algorithms enabled by Pantheon. Specifically, Pantheon facilitates realistic offline training and testing by providing a communal benchmark, evolving dataset, and calibrated emulators to allow approximately realistic offline training and testing.

3.7.1 Overview of Indigo

At its core, Indigo does two things: it observes the network state, and it takes an *action* to adjust its *congestion window*, i.e., the allowable number of in-flight packets. Observations occur each time an ACK is received, and their effect is to update Indigo's internal *state*, consisting of:

1. An exponentially-weighted moving average (EWMA) of the queuing delay, measured by the difference between the current RTT and the minimum RTT observed during the current connection.

2. An EWMA of the sending rate, which we define as the number of bytes sent since the last ACK'ed packet was sent, divided by the RTT.

3. An EWMA of the receiving rate, which we define as the number of bytes received since the ACK preceding the transmission of the most recently ACK'ed packet, divided by the corresponding duration (similar to and inspired by TCP BBR's delivery rate [21]).

4. The current congestion window size.

5. The previous action taken.

Indigo adjusts its congestion window no more frequently than once every 10 ms, with an action from $\div 2$, -10, $+0$, $+10$, $\times 2$ relative to the current congestion window (so $\times 2$ doubles the window).

Indigo stores the mapping from states to actions in a Long Short-Term Memory (LSTM) recurrent neural network [50] with 1 layer of 32 hidden units. It uses a recurrent neural network because this architecture is useful in solving POMDPs [57] (a kind of randomized decision problems under partial information); congestion control can be modeled as such a problem [124]. Indigo learns the mapping through a *training phase* (described below); once trained and deployed, this mapping is fixed.

We note that there may be better parameter choices: number of hidden units, action space, state contents, etc. We have found that the above choices already achieve good performance; further improvements are future work. As one step toward validating our choices, we trained and tested several versions of Indigo with a range of hidden units, from 1 to 256, on an emulated network; choices between 16 and 128 yielded good performance.

3.7.2 Indigo's training phase

Indigo uses imitation learning [13, 97] to train its neural network. At a high level, this happens in two steps: first, we generate one or more *congestion-control oracles*, idealized algorithms that perfectly map states to correct actions, corresponding to links on which Indigo is to be trained. Then we apply a standard imitation learning algorithm that uses these oracles to generate training data.

Of course, no oracle exists for real-world paths. Instead, we generate oracles corresponding to *emulated* paths; this is possible because Pantheon's emulators (§3.5) have few parameters. By the definition of an oracle, if we know the ideal congestion window for a given link, we have the oracle for the link: for any state, output whichever action results in a congestion window closest to the ideal value.

A key insight is that for emulated links, we can very closely approximate the ideal congestion window. For simple links with a fixed bandwidth and minimum one-way delay, the ideal window is given by the link's bandwidth-delay product (BDP) per flow. For calibrated emulators, the ideal window size is further from the BDP. This is because our calibrated emulators also specify a DropTail queue buffer size, per-packet loss rate, etc. Since these variables affect the number of packets in flight at any given time, they also affect the ideal window size. Therefore, we compute the BDP and then search near this value in emulation to find the best fixed congestion window size.

After generating congestion-control oracles corresponding to each training link, we use a distributed, synchronous version of a state-of-the-art imitation learning algorithm called DAgger to train the neural network [97]. The system consists of a server and many workers. At a high level, the workers collect (network link state, ideal window size) pairs, which they send to the server; the server uses these pairs to train a neural net. To gather the pairs, each worker contains:

1. a local copy of the neural network, updated periodically;
2. a sender and receiver in an emulated network link created using Mahimahi [82];
3. an oracle for the above network link; and
4. a buffer for storing (network link state, ideal window size) pairs.

The worker simulates a flow between the sender and the receiver. After each ACK, the sender modifies the congestion window by sampling an action from the local neural network.

After each such action, the worker gathers one data point by querying the oracle on the new state of the network link; it records this state and oracle's answer in its buffer. Once the worker has gathered 1000 pairs, it sends the server the contents of its buffer and receives an updated copy of the neural network. The server uses pairs gathered from the workers to run a typical supervised learning routine [50], which updates the neural network's weights. The server and workers repeat their interaction, iteratively improving the neural network; training is complete when the neural network's outputs converge to the oracle's outputs.

Put another way, the workers observe an active link in order to build a set of (state, action) pairs, and they use a snapshot of the Indigo's neural network in training as the link's congestion-control scheme. The server aggregates all of the workers' pairs and uses them to train new versions of the neural network.

3.7.3 Indigo's performance

In this section, we compare Indigo's performance with that of other congestion-control schemes, and we evaluate the effect of Pantheon's calibrated emulators on performance, versus only training on fixed-bandwidth, fixed-delay emulators.

Indigo in simulation. We evaluate Indigo on 210 synthetic networks, consisting of all combinations of (5, 10, 20, 30, . . . , 200 Mbps) link rate and (10, 20, . . . , 100 ms) min one-way delay. All synthetic networks have infinite queue depth and no loss. For each, we ran Indigo and the Pantheon's schemes (§3.3) for 30 seconds.

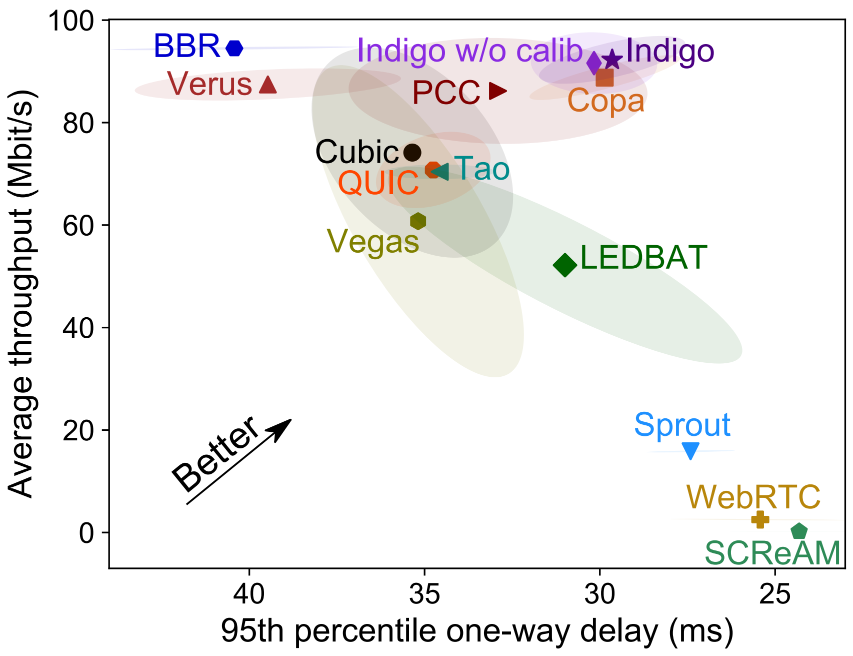
Out of the 210 tests, Indigo ranks first 175 times (83%) and in the top two 206 times (98%). Indigo's performance is near best in both throughput and delay, and is the best in Kleinrock's power metric [58]. In Appendix D we give more detail.

We trained Indigo on 24 synthetic emulators uncorrelated to Pantheon's real network paths, and on the calibrated emulators (§3.5). The synthetic emulators comprise all combinations of (5, 10, 20, 50, 100, and 200 Mbps) link rate and (10, 20, 40, 80 ms) minimum one-way delay, with infinite buffers and no loss.

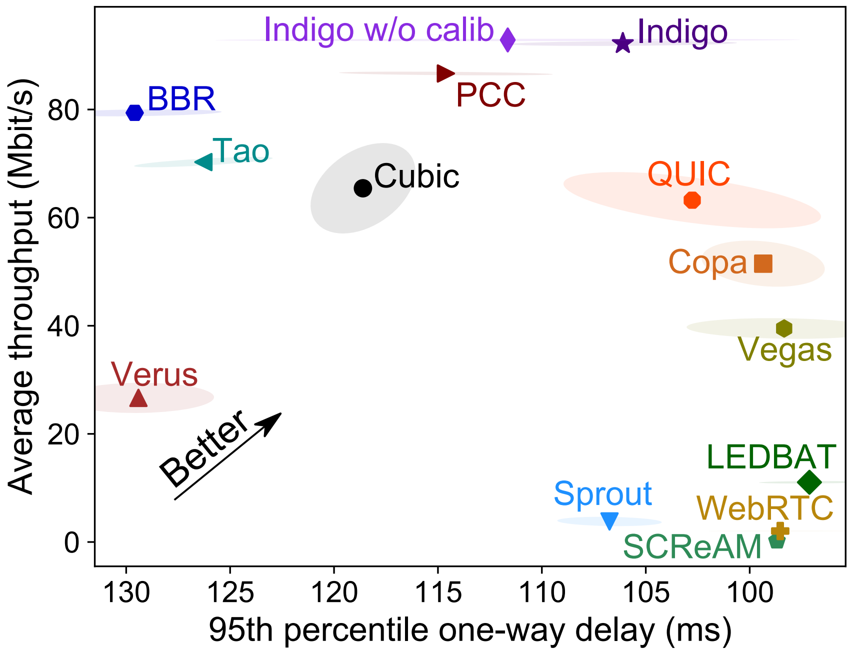
Indigo on Pantheon. We find that Indigo consistently achieves good performance. Figure 3.4 compares Indigo to other schemes in single flow on two wired paths. In both cases, Indigo is at the throughput/delay tradeoff frontier.

Figure 3.5 shows Indigo's performance in the multi-flow case. Figure 3.5a shows the performance of all of Pantheon's congestion-control schemes on a wired path from India to AWS India; Indigo is once again on the throughput/delay tradeoff frontier. Figure 3.5b is a time-domain plot of one trial from Figure 3.5a, suggesting that Indigo shares fairly, at least in some cases.

Benefit of calibrated emulators. Figures 3.4 and 3.5 also depict a variant of Indigo, "Indigo w/o calib," that is only trained on the synthetic emulators, but not the calibrated emulators. The version trained on calibrated emulators is always as least as good or better.

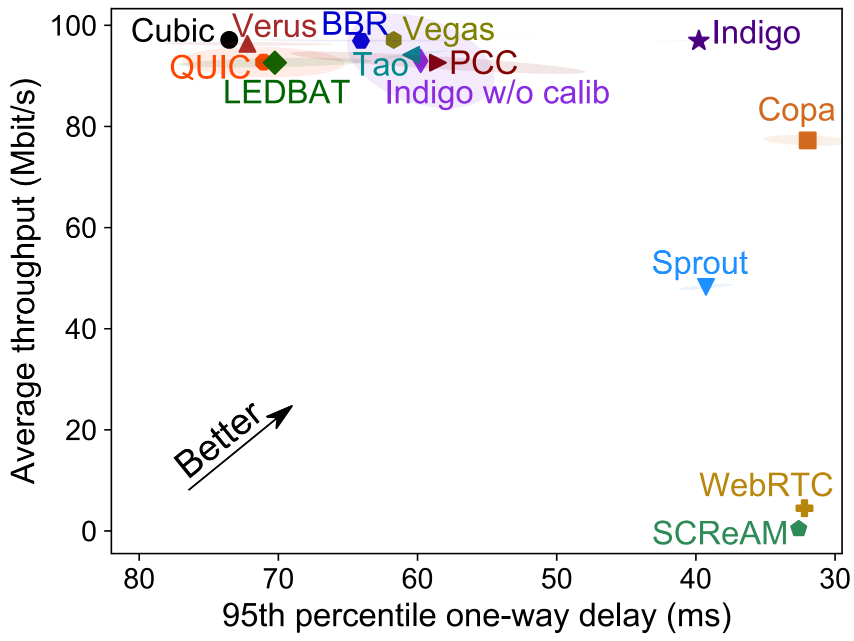


(a) Mexico to AWS California, 10 trials. P1272.

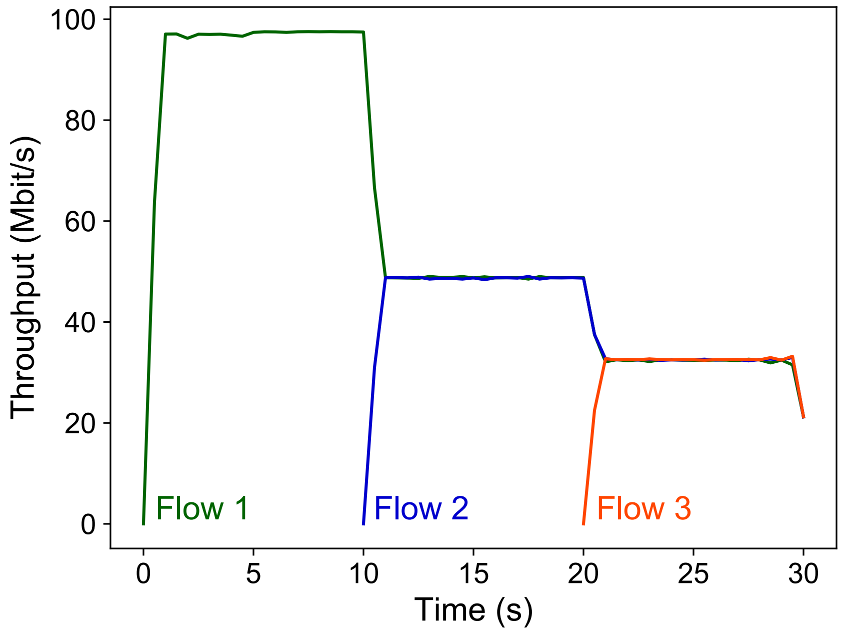


(b) AWS Brazil to Colombia, 10 trials. P1439.

Figure 3.4: Real wired paths, single flow. Indigo's performance is at the throughput/delay tradeoff frontier. Indigo without calibrated emulators ("Indigo w/o calib") gives worse performance.



(a) India to AWS India, 10 trials. P1476.



(b) Time-domain three-flow test. One trial in Figure 3.5a.

Figure 3.5: Real wired paths, multiple flows. Figure 3.5a shows the performance of all congestion-control schemes on multi-flow case. Figure 3.5b shows throughput vs. time for a three-flow run in Figure 3.5a starting 10 seconds apart. Indigo shares the bandwidth fairly.

3.8 Discussion, limitations, and future work

Improving Pantheon. Pantheon would be more useful if it collected more data about congestion-control schemes. For instance, Pantheon currently gathers data only from a handful of nodes—vastly smaller than the evaluations large-scale operators can perform on even a small fraction of a billion-user population.

Moreover, geographic locality does not guarantee network path similarity: two nodes in the same city can have dramatically different network connections. Pantheon also only tests congestion-control schemes at full throttle; other traffic patterns (e.g., Web-like workloads) may provide researchers with valuable information (e.g., how their scheme affects page-load times).

Finally, Pantheon currently measures the interaction between multiple flows of cross-traffic governed by the same scheme, but we are working to make it measure interactions between different schemes. These measurements will help evaluate fairness in the real world.

Improving the calibrated emulators. Our current emulators replicate throughput and delay metrics only within 17% accuracy on average. An open question is whether we can improve emulator fidelity—especially on cellular paths—without risk of overfitting. Considering metrics other than 95th-percentile delay and mean throughput may be one path forward. Adding more schemes to Pantheon could also help—or it might reveal that the current set of emulator parameters, which we have empirically determined, is insufficient for some schemes.

Indigo. We have presented a case study on Indigo, a data-driven approach to congestion-control design that crucially relies on Pantheon's family of emulators. Indigo's trained model is complex and may have unknown failure modes, but the results to date demonstrate how Pantheon can enable new approaches to protocol design.

Indigo produces encouraging results, but there is more evaluation to explore. For example: other training algorithms (e.g., reinforcement learning) may allow for online or hybrid online-offline learning; different neural network designs may reduce costs; and additional inputs (e.g., current delivery rate, as in BBR [21]) or control mechanisms (e.g., packet pacing) may give better performance.

Addressing some shortcomings of Indigo's design should improve its performance. For example, with Indigo's current design, oracles do not adequately capture links whose behaviors change quickly. Finally, training Indigo on more than 30 links should also improve performance.

Chapter 4

Discussion

As much as we welcome researchers to use Puffer and Pantheon for training and validating novel ML algorithms, we also urge caution in applying ML to networking research. Deploying ML algorithms on real systems can be much more challenging and time-consuming than presenting "proof of concept" as related work commonly does. In this chapter, we describe the pitfalls and lessons from our own experience and discuss promising directions forward. We argue that the practicality of ML algorithms for networking research deserves more research study.

4.1 Sequential decisions vs. isolated decisions

Although this dissertation has focused on sequential decision problems on the Internet, many other algorithms running on real networks only make isolated decisions. For instance, network operators deploy traffic classifiers to guarantee Quality of Service (QoS) requirements, detect anomaly and intrusion, and allocate or provision resources more efficiently. The classification decisions made by these algorithms are isolated—a decision classifying a group of network packets into one of the traffic classes of interest does not affect later traffic or decisions of its own[1].

[1] A traffic classifier may decide to drop all the subsequent suspicious packets once it detects an anomalous event, so strictly speaking, its decisions might not be isolated in rare circumstances.

Network researchers have proposed ML approaches to traffic classification, based on supervised learning [45,100] and unsupervised learning [15,67]. Other problems of "isolated decision making," such as network fault management [46,62], have also seen an increase in ML solutions.

We believe algorithms making isolated decisions on networks are more hospitable to ML because the network only serves as another data source for ML and is not "controlled" by ML. Taking traffic classification as an example: traffic classifiers sniff packets on the network and do not usually inject new packets or affect the network behavior. Other real-world requirements are also more flexible, e.g., ML-based traffic classifiers do not necessarily need to achieve robust performance since their results, such as a suspected intrusion, can be confirmed by existing hand-crafted classifiers or even human engineers. Traffic classification can tolerate a higher latency as well, giving more time to ML algorithms to generate results without delaying production services. Besides, the relevant ML techniques (supervised learning and unsupervised learning) have been extensively studied and well understood by the ML community, reducing surprises arising from deployment.

Nevertheless, we still urge a thorough evaluation of these ML algorithms in the real world. Evaluation using recorded network traffic or network simulators cannot capture the complexities of real systems (§2.4 and §3.4).

4.2 RL for networking research

We discussed the challenges of real-world RL in Section 1.1.2, and unfortunately, we observed every single of them on Puffer and Pantheon. For example, our best effort of faithfully simulating a given network path still yields a considerable error (17% on average; §3.5). A congestion-control algorithm equipped with neural networks must still run at line speed and avoid packet flooding in the worst case, which is arguably more critical than good performance on average. Reliable measurement of video-streaming algorithms requires years of data due to the heavy-tailed behavior of the Internet (§2.2.4), whereas collecting data from real users is very costly.

Our experience finds that building real-world systems helps understand and address the practical challenges better than inventing algorithms "in the lab." However, existing work often heavily relies on network simulators for evaluation and conducts small-scale field experiments, if any, in the last stage of algorithm design.

It is a common pitfall because, first, we have shown that simulation results are not indicative of real-world performance (§2.4 and §3.4). In addition, we underscore that replaying pre-recorded throughput traces in a network simulator does *not* make simulation results indicative either. Faithfully replicating real networks is more than replaying throughput traces and still beyond our current capabilities [41, 42, 87]. Future work needs to improve calibrated emulators (§3.5) or invent other techniques to better model real networks.

Second, practical requirements will affect and often constrain the design space of algorithms. Therefore, we suggest including real-world evaluations in the loop to drive each iteration. Last-minute field experiments or no experiments on real systems at all might render previous efforts in vain or the resulting algorithm impractical. For example, we suspect that directly applying model-free RL to designing ABR algorithms for the real Internet may not be feasible yet (§2.4.3). An early experiment on any real video-streaming platform, such as Puffer, would have given a second thought to this learning method. By contrast, we restricted our use of RL in both problems to model-based RL and imitation learning, which use supervised learning as part of their approaches and have desirable properties, such as robust performance, stable training, and sample efficiency [29, 49]. They are grounded on control theory that has a long history for addressing practical problems and remain active areas of research today.

Finally, "promising" simulation results can not only mislead the algorithm designer but also give false hope to other researchers, encouraging the eager adoption of RL and often deep RL. However, deep RL is infamous for overfitting, so readers should be more cautious when interpreting the results if the algorithm is trained and tested in the same network simulator. We should assume that simulation results do *not* represent real-world performance unless explicit evidence is provided.

4.2.1 Generalizing RL algorithms to real networks

Besides the testing performance, we also want to generalize the training performance of RL—either in simulation or real life—to the actual deployment environment. Depending on the location of training, we discuss three approaches below.

1. **Training in network simulators.** Many RL algorithms require training in network simulators for various reasons: 1) collecting training data on real networks can be expensive or intrusive to existing services; 2) training in real life can be too slow and impractical compared with instantiating as many simulators as needed; 3) the exploratory learning phase might violate safety requirements of real systems and prevent useful communication of other network applications; 4) the learning itself might require information available only within network simulators (e.g., Indigo).

   For these RL algorithms training in simulated environments, it is well-known that the discrepancy between simulation and reality creates hurdles for generalizability [16]. Therefore, we proposed calibrated emulators (§3.5) to bridge the gap and demonstrated their usefulness for generalizing Indigo's performance (§3.7.3). We discussed the future work of improving calibrated emulators in Section 3.8, and believe that further reducing the simulation-to-reality gap deserves more effort from the research community.

2. **Training on real networks.** Besides the discussed challenges of training RL in the real world, real networks—at least the Internet—also exhibit extremely noisy behavior, as we observed on Puffer (§2.2.4). The substantial system noise limits or even prevents the trial-and-error paradigm of (model-free) RL since the algorithm can no longer vary an action slightly and detect a reliable change in the resulting reward. Therefore, we suspect training RL algorithms directly on the Internet is not feasible yet (§2.4.3).

   Recently, researchers have proposed RL algorithms that are robust to system noise, including noisy rewards [88, 95, 117], but these algorithms remain to be applied and verified on the Internet. Alternatively, future work can investigate and better understand the heavy-tailed Internet behavior, so we can adjust RL algorithms to this particular environment without the need of achieving general robustness.

3. **Training on log data from real systems.** When good simulators are not available, and training on real networks is not feasible, RL algorithms can be trained offline on batches of log data from real systems without interaction with the environment. This is known as *offline RL* or *batch RL* [61, 101]. Fugu adopts offline (model-based) RL, but we further

emphasize that the log data should come from the actual deployment environment so as to ensure better generalization.

Training on log data is suitable for the systems that cannot afford service interruption or performance degradation in the algorithm's learning phase, but it also excludes many powerful trial-and-error RL techniques that require interaction with the environment. Future work needs to pay more attention to RL algorithms that learn from log data, such as model-based RL and imitation learning, to supplement model-free RL. We can also study how to adapt real systems for training these RL algorithms offline.

4.2.2 Other promising directions

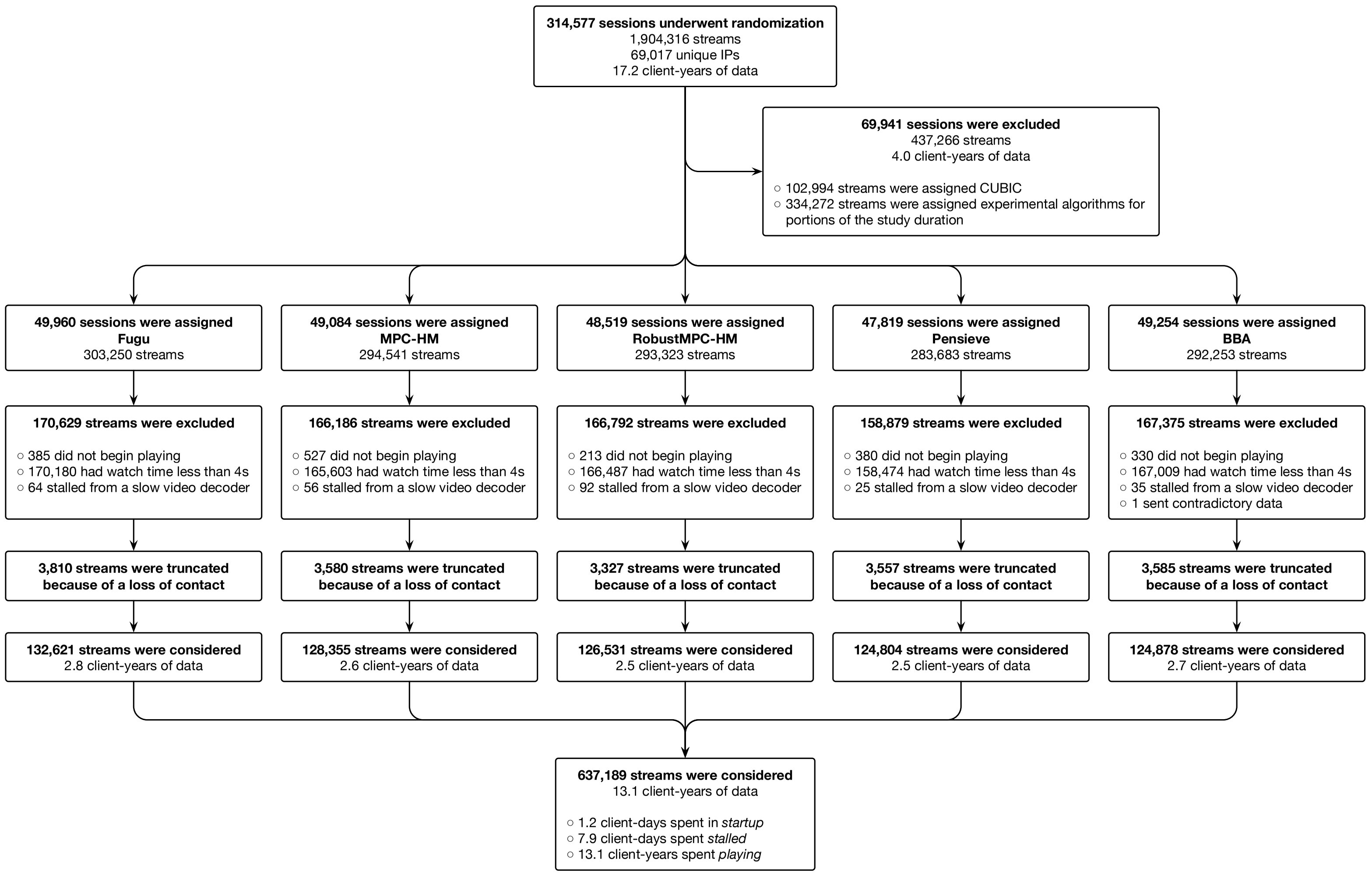
In addition to the future work mentioned above, we believe the guarantee of worst-case performance and interpretability of algorithms are also promising directions forward.

Production systems are often interested in more than the average performance of an algorithm; they are also concerned with the worst-case performance. This criterion is different from the optimization goal of long-term average return in most RL research. For instance, if a congestion-control algorithm mostly works well but can overwhelm the network with packets even occasionally, it may cause congestion collapse [79] and bring down other services running on the network. Recent research in RL has proposed algorithms to optimize worst-case value functions as the objective [68, 115], and we are eager to know if these algorithms are effective on real networks and if they may accelerate the adoption of RL.

When the algorithm causes service degradation and even outages (e.g., due to the absence of worst-case performance guarantee), we want to debug its behavior and avoid similar issues in the future. However, neural-network-based algorithms are often not interpretable, which is an underestimated reason that production systems are reluctant to deploy them. The interpretability of RL algorithms might also partly explain why simple algorithms, such as BBA [51] and TCP Cubic [44], are still widely used on the Internet. Interpretable machine learning has seen an increasing interest in recent years [34, 76], and we believe real networks would welcome more explainable ML algorithms in the future.

Appendices

A Randomized trial flow diagram



314,577 sessions underwent randomization
1,904,316 streams
69,017 unique IPs
17.2 client-years of data
69,941 sessions were excluded
437,266 streams
4.0 client-years of data
○ 102,994 streams were assigned CUBIC
○ 334,272 streams were assigned experimental algorithms for portions of the study duration
49,960 sessions were assigned Fugu
303,250 streams
49,084 sessions were assigned MPC-HM
294,541 streams
48,519 sessions were assigned RobustMPC-HM
293,323 streams
47,819 sessions were assigned Pensieve
283,683 streams
49,254 sessions were assigned BBA
292,253 streams
170,629 streams were excluded
○ 385 did not begin playing
○ 170,180 had watch time less than 4s
○ 64 stalled from a slow video decoder
166,186 streams were excluded
○ 527 did not begin playing
○ 165,603 had watch time less than 4s
○ 56 stalled from a slow video decoder
166,792 streams were excluded
○ 213 did not begin playing
○ 166,487 had watch time less than 4s
○ 92 stalled from a slow video decoder
158,879 streams were excluded
○ 380 did not begin playing
○ 158,474 had watch time less than 4s
○ 25 stalled from a slow video decoder
167,375 streams were excluded
○ 330 did not begin playing
○ 167,009 had watch time less than 4s
○ 35 stalled from a slow video decoder
○ 1 sent contradictory data
3,810 streams were truncated because of a loss of contact
3,580 streams were truncated because of a loss of contact
3,327 streams were truncated because of a loss of contact
3,557 streams were truncated because of a loss of contact
3,585 streams were truncated because of a loss of contact
132,621 streams were considered
2.8 client-years of data
128,355 streams were considered
2.6 client-years of data
126,531 streams were considered
2.5 client-years of data
124,804 streams were considered
2.5 client-years of data
124,878 streams were considered
2.7 client-years of data
637,189 streams were considered
13.1 client-years of data
○ 1.2 client-days spent in *startup*
○ 7.9 client-days spent *stalled*
○ 13.1 client-years spent *playing*

Figure A.1: CONSORT-style diagram [102] of experimental flow for the primary results (Table 2.1 and Figure 2.11), obtained during the period Jan. 26–Aug. 7, 2019, and Aug. 30–Oct. 16, 2019. A "session" represents one visit to the Puffer video player and may contain many "streams." Reloading starts a new session, but changing channels only starts a new stream and does not change TCP connections or ABR algorithms.

B Description of open data

The open data we are releasing comprise different "measurements"—each measurement contains a different set of time-series data collected on Puffer servers. Below we highlight the format of interesting fields in three measurements that are essential for analysis: `video_sent`, `video_acked`, and `client_buffer`.

`video_sent` collects a data point every time a Puffer server sends a video chunk to a client. Each data point contains:

- `time`: timestamp when the chunk is sent
- `session_id`: unique ID for the video session
- `expt_id`: unique ID to identify the experimental group; `expt_id` can be used as a key to retrieve the experimental setting (e.g., ABR, congestion control) when sending the chunk, in another file we are providing.
- `channel`: TV channel name
- `video_ts`: unique presentation timestamp of the chunk
- `format`: encoding settings of the chunk, including resolution and constant rate factor (CRF)
- `size`: size of the chunk
- `ssim_index`: SSIM of the chunk
- `cwnd`: congestion window size (`tcpi_snd_cwnd`)
- `in_flight`: number of unacknowledged packets in flight (`tcpi_unacked` - `tcpi_sacked` - `tcpi_lost` + `tcpi_retrans`)
- `min_rtt`: minimum RTT (`tcpi_min_rtt`)
- `rtt`: smoothed RTT estimate (`tcpi_rtt`)
- `delivery_rate`: estimate of TCP throughput (`tcpi_delivery_rate`)

`video_acked` collects a data point every time a Puffer server receives a video chunk acknowledgement from a client. Each data point can be matched to a data point in `video_sent` using `video_ts` (if the chunk is ever acknowledged) and used to calculate the transmission time of the chunk—difference between the timestamps in the two data points. Specifically, each data point in `video_acked` contains:

- `time`: timestamp when the chunk is acknowledged

- `session_id`
- `expt_id`
- `channel`
- `video_ts`

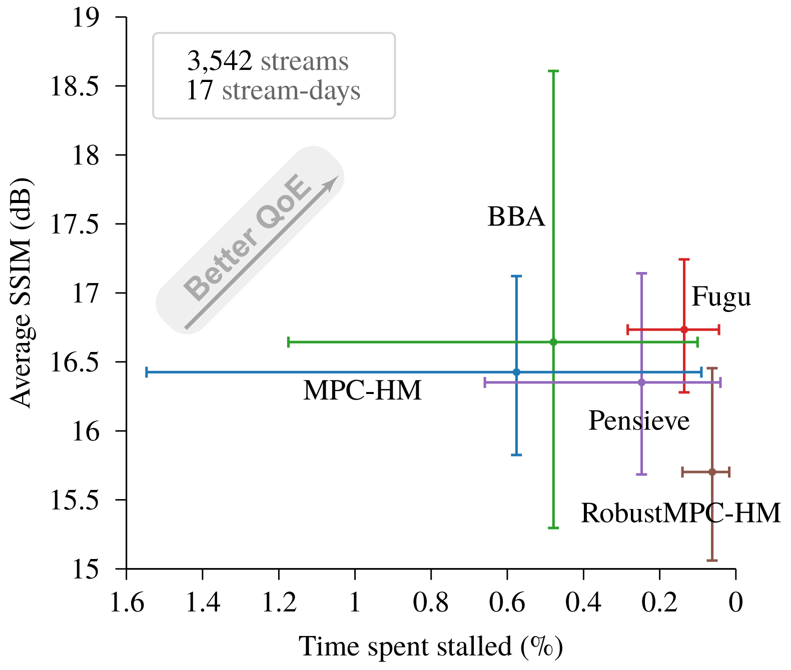
`client_buffer` collects client-side information reported to Puffer servers on a regular interval and when certain events occur. Each data point contains:

- `time`: timestamp when the client message is received
- `session_id`
- `expt_id`
- `channel`
- `event`: event type, e.g., was this triggered by a regular report every quarter second, or because the client stalled or began playing.
- `buffer`: playback buffer size
- `cum_rebuf`: cumulative rebuffer time in the current stream

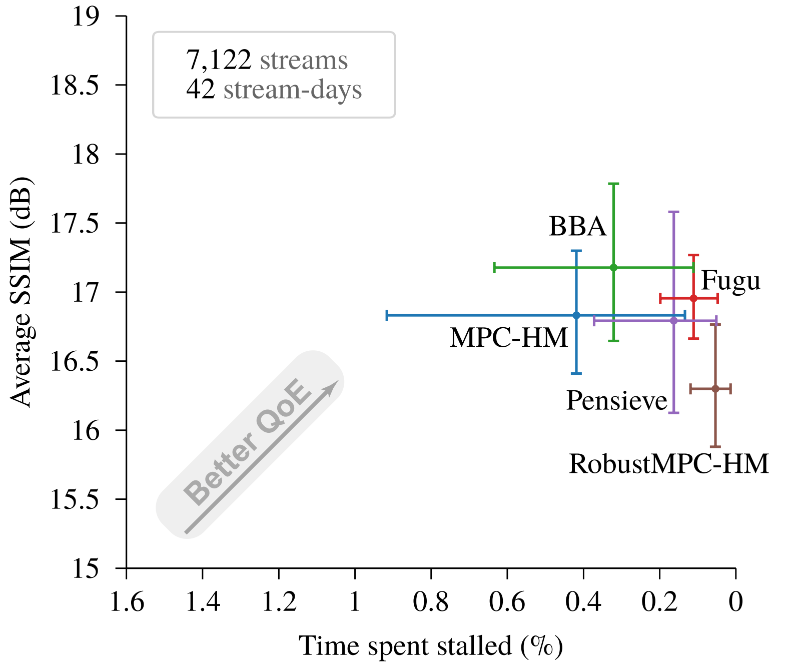
Between Jan. 26, 2019 and Feb. 2, 2020, we collected 675,839,652 data points in `video_sent`, 677,956,279 data points in `video_acked`, and 4,622,575,336 data points in `client_buffer`.

C Statistical uncertainty in experiments of various lengths

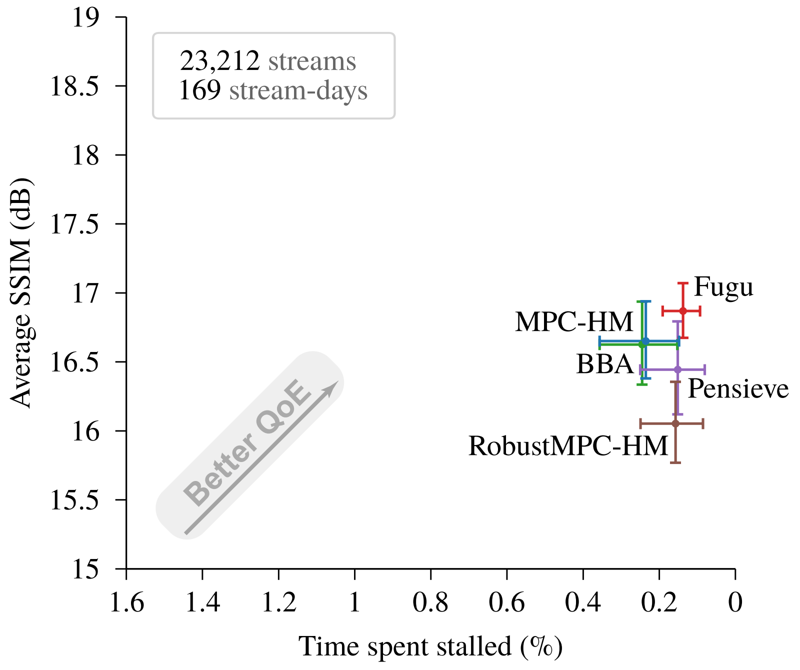
Figure C.1 shows that the confidence intervals of five ABR schemes reduce as the experiment length increases from a day to eight months, within the period of the primary experiment on all paths (Figure 2.11). Any benefits reported on an insufficient amount of data can be purely statistical noise, but results are reliable after collecting about 2.5 years of video data per scheme.



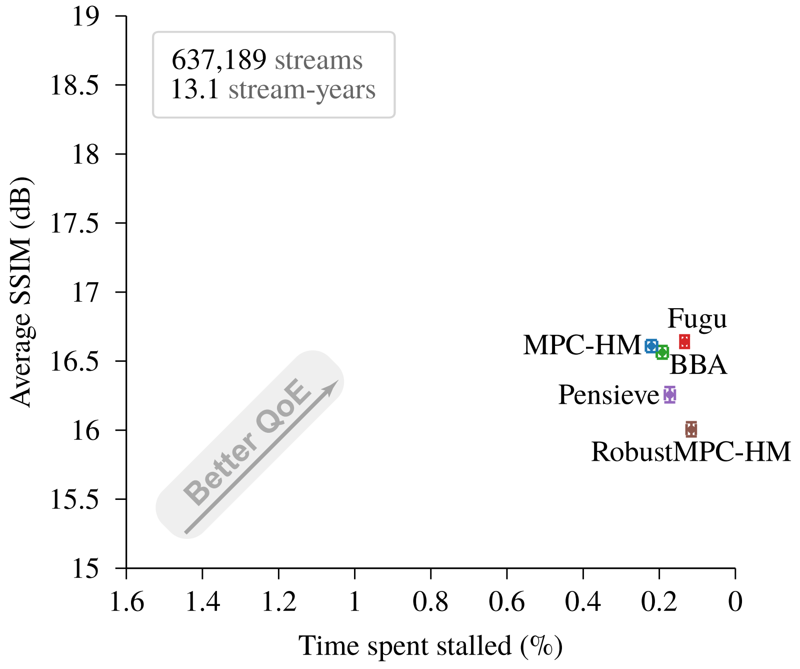
(a) Experiment lasting a single day on Jan. 26, 2019



(b) Experiment lasting a week since Jan. 26, 2019



(c) Experiment lasting a month since Jan. 26, 2019



(d) Experiment lasting eight months (same as Figure 2.11 after zooming in)

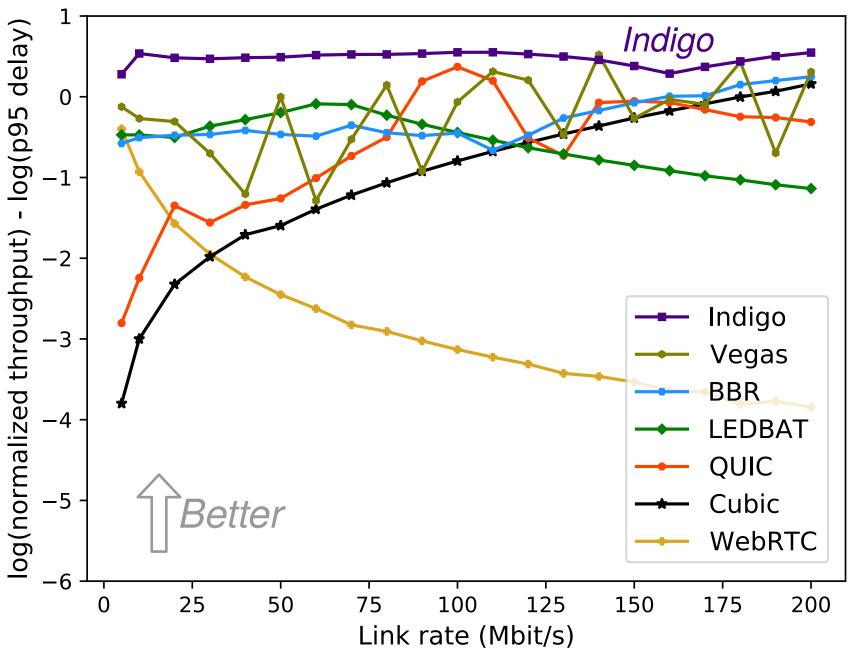
Figure C.1: Confidence intervals narrowed with more data collected for each scheme.

D Indigo on simulated networks

Table D.1 gives average Kleinrock's power of each scheme over the 210 synthetic networks in Section 3.7.3. We also remove each of the input features of Indigo one at a time, which results in significantly worse performance than Indigo with all input features. Figure D.1 shows a representative result at 50 ms min one-way delay and across all link rates, where Indigo is as good as or better than the other schemes.

Scheme	**Average Kleinrock's power**
Indigo	0.530
Indigo w/o delivery rate	0.480
Indigo w/o CWND	0.479
Indigo w/o sending rate	0.366
Tao	0.104
Indigo w/o queuing delay	-0.112
BBR	-0.231
Vegas	-0.261
QUIC	-0.660
PCC	-0.792
LEDBAT	-0.817
Cubic	-1.100
Verus	-1.723
Sprout	-1.986
WebRTC	-2.781
SCReAM	-5.084

Table D.1: Average Kleinrock's power of each scheme over all 210 synthetic networks.



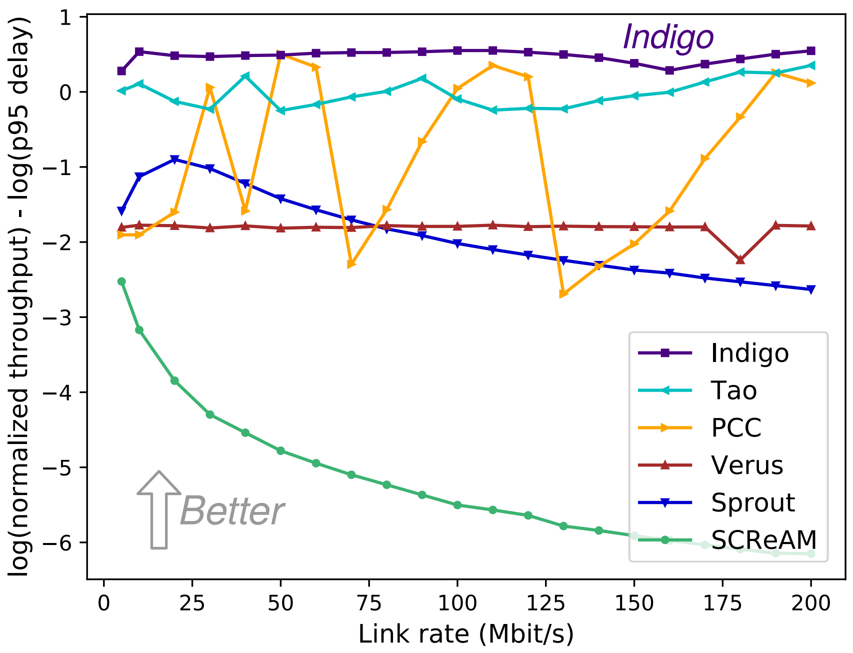


Figure D.1: Power of schemes over synthetic networks with varying link rates and 50 ms min one-way delay (§3.7.3). The schemes are split into two graphs for clarity.

Bibliography

[1] Locast: Non-profit retransmission of broadcast television, June 2018. https://news.locast.org/app/uploads/2018/11/Locast-White-Paper.pdf.

[2] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[3] Alekh Agarwal, Nan Jiang, and Sham M. Kakade. Lecture notes on the theory of reinforcement learning. 2019.

[4] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: Auto-tuning video ABR algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM SIGCOMM*, pages 44–58, 2018.

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

[6] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018.

[7] Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129, 1995.

[8] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for Internet video. *ACM SIGCOMM Computer Communication Review*, 43(4):339–350, 2013.

[9] Hari Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, EECS Department, University of California, Berkeley, July 1998.

[10] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198, 2017.

[11] Richard Bellman. Dynamic programming. *Princeton University Press, Princeton*, 1957.

[12] Richard Bellman. A Markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[13] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[14] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. WebRTC 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[15] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, 2006.

[16] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4243–4250. IEEE, 2018.

[17] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, 1994.

[18] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym, 2016.

[19] Eduardo F. Camacho and Carlos Bordons Alba. *Model predictive control*. Springer Science & Business Media, 2013.

[20] Marta Carbone and Luigi Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[21] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016.

[22] Jay Chen, Lakshmi Subramanian, Janardhan Iyengar, and Bryan Ford. TAQ: Enhancing fairness and performance predictability in small packet regimes. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 7:1–7:14, New York, NY, USA, 2014. ACM.

[23] Yuchung Cheng and Neal Cardwell. Making Linux TCP fast. In *Netdev Conference*, 2016.

[24] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[25] Federal Communications Commission. Measuring Broadband America. https://www.fcc.gov/general/measuring-broadband-america.

[26] Paul Crews and Hudson Ayers. CS 244 '18: Recreating and extending Pensieve, 2018. https://reproducingnetworkresearch.wordpress.com/2018/07/16/cs-244-18-recreating-and-extending-pensieve/.

[27] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*, 2018.

[28] Nigel Davies, Gordon S. Blair, Keith Cheverst, and Adrian Friday. A network emulator to support the development of adaptive applications. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, MLICS '95, pages 47–56, Berkeley, CA, USA, 1995. USENIX Association.

[29] Marc Deisenroth and Carl E. Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML)*, pages 465–472, 2011.

[30] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. *A survey on policy search for robotics*. Now Publishers, 2013.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition (CVPR)*, pages 248–255. IEEE, 2009.

[32] Mo Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. PCC: Re-architecting congestion control for consistent high performance. In *Presented as part of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[33] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

[34] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[35] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016.

[36] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. In *ICML 2019 Workshop RL4RealLife*, 2019.

[37] Bradley Efron and Robert Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75, 1986.

[38] Kevin Fall. Network emulation in the Vint/NS simulator. In *Proceedings IEEE International Symposium on Computers and Communications (Cat. No. PR00250)*, pages 244–250. IEEE, 1999.

[39] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Comput. Commun. Rev.*, 26(3):5–21, July 1996.

[40] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. TCP friendly rate control (TFRC): Protocol specification. Technical Report RFC 5348, Internet Engineering Task Force, 2009.

[41] Sally Floyd and Eddie Kohler. Internet research needs better models. *ACM SIGCOMM Computer Communication Review*, 33(1):29–34, 2003.

[42] Sally Floyd and Vern Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, 2001.

[43] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, 2018.

[44] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[45] Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. ACAS: Automated construction of application signatures. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 197–202, 2005.

[46] Hassan Hajji. Statistical analysis of network traffic for adaptive faults detection. *IEEE Transactions on Neural Networks*, 16(5):1053–1063, 2005.

[47] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[48] Stephen Hemminger. Network emulation with NetEm. In *linux.conf.au*, pages 18–23, 2005.

[49] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pages 4565–4573, 2016.

[50] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[51] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 Conference of the ACM SIGCOMM*, pages 187–198, 2014.

[52] Van Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

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

[54] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. CFA: A practical prediction system for video QoE optimization. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 137–150, 2016.

[55] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies*, pages 97–108, 2012.

[56] Ingemar Johansson and Zaheduzzaman Sarker. Self-clocked rate adaptation for multimedia. Technical Report RFC 8298, Internet Engineering Task Force, 2017.

[57] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

[58] Leonard Kleinrock. On flow control in computer networks. In *Proceedings of the International Conference on Communications*, volume 2, pages 27–2, 1978.

[59] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, pages 1008–1014, 2000.

[60] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[61] Aviral Kumar, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. Stabilizing off-policy Q-learning via bootstrapping error reduction. *Advances in Neural Information Processing Systems*, 32, 2019.

[62] Yash Kumar, Hasan Farooq, and Ali Imran. Fault prediction and reliability analysis in a real cellular network. In *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1090–1095. IEEE, 2017.

[63] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the 2017 Conference of the ACM SIGCOMM*, pages 183–196, 2017.

[64] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. TACK: Improving wireless transport performance by taming acknowledgments. To appear in *SIGCOMM 2020*.

[65] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C. Begen, and David Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, 2014.

[66] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[67] Justin Ma, Kirill Levchenko, Christian Kreibich, Stefan Savage, and Geoffrey M. Voelker. Unexpected means of protocol inference. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 313–326, 2006.

[68] Daniel J. Mankowitz, Timothy A. Mann, Pierre-Luc Bacon, Doina Precup, and Shie Mannor. Learning robust options. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[69] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. In *ICML 2019 Workshop RL4RealLife*, 2019.

[70] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the 2017 Conference of the ACM SIGCOMM*, pages 197–210. ACM, 2017.

[71] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *International Conference on Learning Representations*, 2019.

[72] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods

for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

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

[74] Jonas Močkus. On Bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pages 400–404. Springer, 1975.

[75] Ricky K.P. Mok, Xiapu Luo, Edmond W.W. Chan, and Rocky K.C. Chang. QDASH: a QoE-aware DASH system. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 11–22, 2012.

[76] Christoph Molnar. *Interpretable Machine Learning*. 2019. https://christophm.github.io/interpretable-ml-book/.

[77] *Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*, April 2012. ISO/IEC 23009-1 (http://standards.iso.org/ittf/PubliclyAvailableStandards).

[78] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[79] John Nagle. Congestion control in IP/TCP internetworks. RFC 896, RFC Editor, January 1984.

[80] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video QoE fairness. In *SIGCOMM*, pages 408–423. 2019.

[81] Netflix Open Connect. https://openconnect.netflix.com/.

[82] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 417–429, 2015.

[83] Andrew Y. Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *ICML*, volume 1, page 2, 2000.

[84] The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[85] NTP: The network time protocol. `http://www.ntp.org/`.

[86] Maximilian Ott, Ivan Seskar, Robert Siraccusa, and Manpreet Singh. ORBIT testbed software architecture: Supporting experiments as a service. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 136–145. IEEE, 2005.

[87] Vern Paxson and Sally Floyd. Why we don't know how to simulate the Internet. In *Proceedings of the 29th conference on Winter simulation*, pages 1037–1044, 1997.

[88] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 2817–2826, 2017.

[89] Dean A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, pages 305–313, 1989.

[90] Yanyuan Qin, Shuai Hao, Krishna R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 366–378. ACM, 2018.

[91] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Trans. on Comp. Sys.*, 8(2):158–181, May 1990.

[92] George F. Riley. Simulation of large scale networks II: Large-scale network simulations with GTNetS. In *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*, WSC '03, pages 676–684. Winter Simulation Conference, 2003.

[93] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, January 1997.

[94] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[95] Joshua Romoff, Peter Henderson, Alexandre Piché, Vincent Francois-Lavet, and Joelle Pineau. Reward estimation for variance reduction in deep reinforcement learning. *arXiv preprint arXiv:1805.03359*, 2018.

[96] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 627–635, 2011.

[97] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010.

[98] Gavin A. Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[99] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, 1998.

[100] Dominik Schatzmann, Wolfgang Mühlbauer, Thrasyvoulos Spyropoulos, and Xenofontas Dimitropoulos. Digging into HTTPS: flow-based classification of webmail traffic. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 322–327, 2010.

[101] Bruno Scherrer, Victor Gabillon, Mohammad Ghavamzadeh, and Matthieu Geist. Approximate modified policy iteration. *arXiv preprint arXiv:1205.3054*, 2012.

[102] Kenneth F. Schulz, Douglas G. Altman, and David Moher. CONSORT 2010 statement: updated guidelines for reporting parallel group randomised trials. *BMC medicine*, 8(1):18, 2010.

[103] Alexander T. Schwarm and Michael Nikolaou. Chance-constrained model predictive control. *AIChE Journal*, 45(8):1743–1752, 1999.

[104] Sea Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. Low extra delay background transport (LEDBAT). Technical Report RFC 6817, Internet Engineering Task Force, 2012.

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

[106] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 Conference of the ACM SIGCOMM*, pages 479–490, 2014.

[107] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[108] Edward Jay Sondik. The optimal control of partially observable Markov processes. Technical report, Doctoral dissertation, Stanford University, 1971.

[109] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the DASH reference player. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, pages 123–137, New York, NY, USA, 2018. ACM.

[110] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. BOLA: Near-optimal bitrate adaptation for online videos. In *INFOCOM 2016-The 35th Annual IEEE*

International Conference on Computer Communications, IEEE, pages 1–9. IEEE, 2016.

[111] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 Conference of the ACM SIGCOMM*, pages 272–285, 2016.

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

[113] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[114] Cisco Systems. Cisco Visual Networking Index: Forecast and trends, 2017–2022, November 2018. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf.

[115] Aviv Tamar, Shie Mannor, and Huan Xu. Scaling up robust MDPs using function approximation. In *International Conference on Machine Learning*, pages 181–189, 2014.

[116] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound TCP approach for high-speed and long distance networks. Technical Report MSR-TR-2005-86, July 2005.

[117] Yee Teh, Victor Bapst, Wojciech M. Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506, 2017.

[118] Guibin Tian and Yong Liu. Towards agile and smooth video adaptation in dynamic HTTP streaming. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies*, pages 109–120, 2012.

[119] Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[120] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, December 2002.

[121] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

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

[123] Brian White, Jay Lepreau, and Shashi Guruprasad. Lowering the barrier to wireless and mobile experimentation. *SIGCOMM Comput. Commun. Rev.*, 33(1):47–52, January 2003.

[124] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated congestion control. *Proceedings of the 2013 Conference of the ACM SIGCOMM*, 43(4):123–134, 2013.

[125] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 459–471. USENIX, 2013.

[126] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[127] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, July 2018. USENIX Association.

[128] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 Conference of the ACM SIGCOMM*, pages 325–338, 2015.

[129] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 509–522, New York, NY, USA, 2015. ACM.

[130] Tong Zhang, Fengyuan Ren, Wenxue Cheng, Xiaohui Luo, Ran Shu, and Xiaolan Liu. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in DASH. In *IEEE INFOCOM 2017*, pages 1–9. IEEE, 2017.