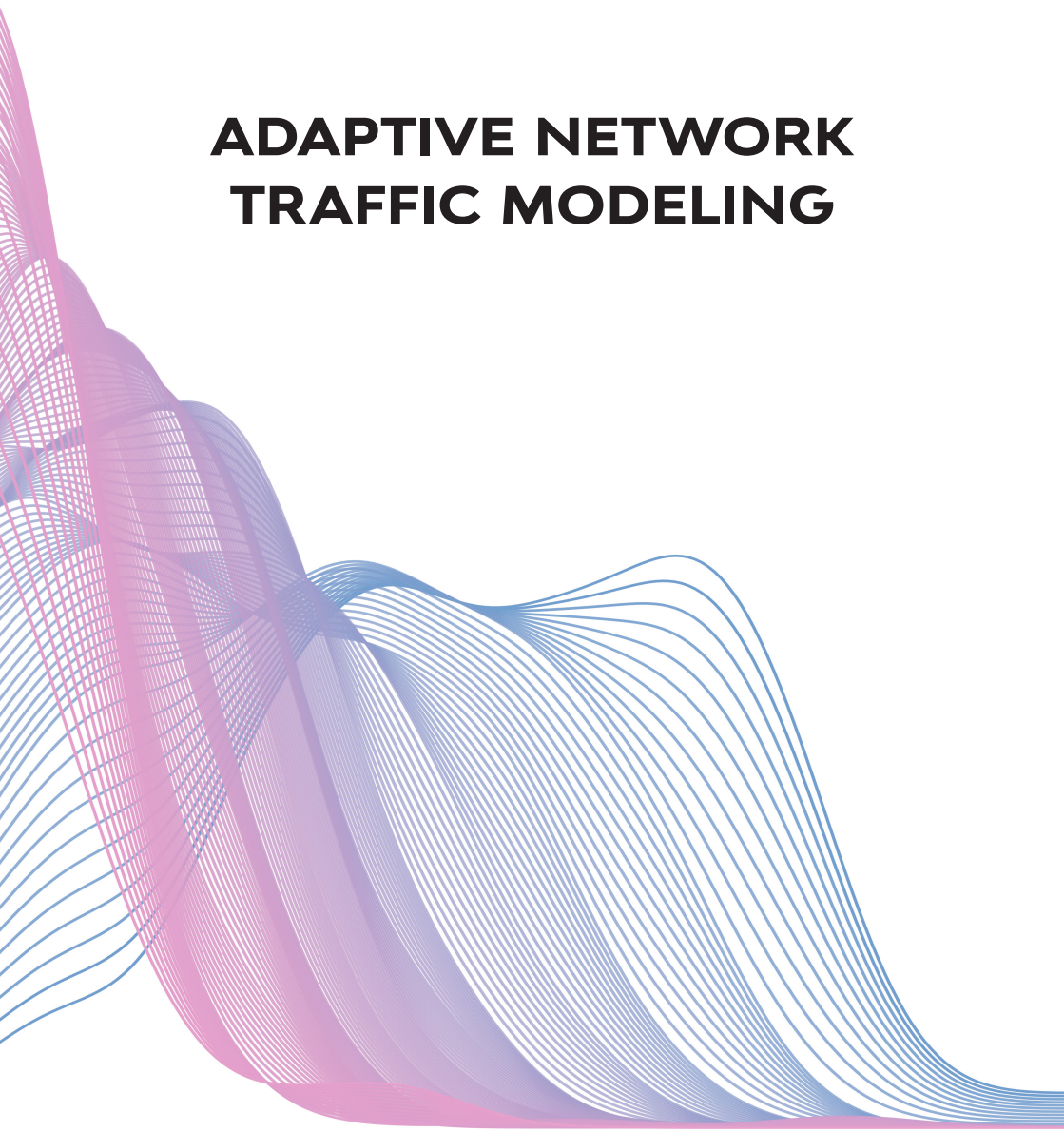
Alexander Dietmüller

ADAPTIVE NETWORK TRAFFIC MODELING



Diss. ETH No. 30455

DISS. ETH NO. 30455

ADAPTIVE NETWORK TRAFFIC MODELING

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

ALEXANDER DIETMÜLLER
MSc ETH EEIT
ETH Zurich

born on 27.07.1991

accepted on the recommendation of

Prof. Dr. Laurent Vanbever
Prof. Dr. Arpit Gupta
Dr. Francis Yan

2024

The cover shows the distribution of video chunk download times collected by the Puffer project on January 14th, 2024.

The download times are binned and shaded by the size of the downloaded video chunk, and printed on a logarithmic scale.

Even this small slice of data reveals the complexity and beauty of network traffic. A beauty this dissertation aims to explore.

Diss. ETH No. 30455
TIK-Schriftenreihe-Nr. 217

ACKNOWLEDGMENTS

It takes longer than you think.

From conducting experiments over writing papers to waiting for reviews; if I had to summarize research in a single sentence, it would be this one. But thanks to the people around me, I would do it all again without hesitation.

First and foremost, I owe my deepest thanks to Prof. Dr. Laurent Vanbever for his supervision. From my Master's thesis and through my PhD, he trusted my judgement and abilities, giving me the opportunity and freedom that made this dissertation possible. I am grateful for this trust and the invaluable advice Laurent has provided me over the past years.

I also thank my co-examiners, Prof. Dr. Aprit Gupta and Dr. Francis Yan, for their time and effort in thoroughly evaluating this dissertation.

I thank my family for their unwavering support and readiness to lend an ear to doubts and frustrations. In particular, I thank my partner, Tanja, my brother, Daniel, and my mother, Elfriede. You are the most important people in my life, and I would not have reached this point without you.

These thanks extend to all of my friends. To Hermann and Hannah, for the coffee breaks and games throughout the years. To Conrad and Hermann (once more) for every discussion we had since our first year at ETH. To my friends back home in Germany, who kept in touch despite the distance, and to all my friends in Zürich, who ensured that I never felt alone.

Last but not least, I thank the entire Networked Systems Group (NSG) at ETH Zürich. Dr. Albert Gran Alcoz, with whom I've been sharing an office since my PhD began and who took every step alongside me. Dr. Romain Jacob, who has been an integral part of much of the research presented in this dissertation. Dr. Tobias Bühler, who supervised my Master's thesis, the beginning of my research. Dr. Edgar Costa Molero, Dr. Thomas Holterbach, Dr. Rüdiger Birkner, and Dr. Maria Apostolaki, who, alongside Albert and Tobias, shaped my early years in the group. And indeed everyone in NSG for their discussions, feedback, and support. I also want to thank our administrator, Beat Futterknecht, for his assistance throughout the years.

I wish I could list every name here, for all of you have shaped my journey. It may have taken longer than expected, but you made it worth the time.

Alexander
Zürich, September 2024

ABSTRACT

The Internet is a decentralized and constantly growing control system that has become an integral part of the lives of over 5.3 billion people. With this scale comes a vast array of applications. Performant and robust control of applications like video streaming requires modeling network traffic, such as estimating whether the network is congested or how long it will take to transmit data. The complexity of this modeling problem has steadily increased over time: the model space is ever-growing with each new network algorithm and application, while observable signals have remained largely unchanged. It has become extremely difficult, and perhaps intractable, to model network traffic from first principles, and research has increasingly turned to machine learning (ML) to learn models from data.

This dissertation explores the opportunities and challenges of using ML for network traffic modeling and additionally investigates how advances in programmable networking may provide better signals.

First, we study *learning over time*. We present Memento, a sample selection system for updating ML models with a focus on tail performance while avoiding unnecessary retraining. The key insight behind Memento is that a smart data selection is crucial to maintain representative training data and to decide when retraining models with the selected data is beneficial.

Second, we investigate *learning over space*, the generalization of models to other network environments and tasks, and present a Network Traffic Transformer (NTT). NTT is a pre-trained Transformer-based model that can be efficiently fine-tuned to different networks and prediction tasks.

Third, we study the underlying problem of *learning latent network state* common to many prediction tasks. Through in-depth analysis and comparison of several ML-based models for video streaming, we gain important insights into modeling strategies and model generalizability.

Finally, we explore the potential of *programmable networks* to enhance observable signals by programmatically processing all packets in the network, albeit with limited computational resources. We present FitNets, which makes the most of constrained programmability with hardware-software co-design: FitNets learns accurate distributions of network traffic features in the control plane, enabled by efficient model scoring in the data plane.

ZUSAMMENFASSUNG

Das Internet ist ein dezentralisiertes und ständig wachsendes System, das zu einem integralen Bestandteil im Leben von über 5,3 Milliarden Menschen geworden ist. Mit diesem Umfang geht eine große Vielfalt an Anwendungen einher. Für eine effiziente und robuste Regelung von Anwendungen wie Video-Streaming muss der Netzwerkverkehr modelliert werden, etwa ob das Netzwerk überlastet ist oder wie schnell Daten übertragen werden können. Die Modellierungskomplexität nimmt stetig zu: Der Modellraum wächst mit jedem neuen Netzwerkalgorithmus und jeder neuen Anwendung, während die beobachtbaren Signale weitgehend unverändert bleiben. Es ist äußerst schwierig, vielleicht sogar unmöglich, den Netzwerkverkehr von Grund auf zu modellieren, und die Forschung wendet sich zunehmend dem maschinellen Lernen (ML) zu, um Modelle aus Daten zu lernen.

Diese Dissertation untersucht die Chancen und Herausforderungen der Modellierung von Netzwerkverkehr mit Hilfe von ML und zusätzlich, wie programmierbare Netzwerke bessere Signale liefern können.

Zunächst untersuchen wir das *Lernen über die Zeit*. Wir präsentieren Memento, ein System zur Aktualisierung von ML-Modellen mit Fokus auf Tail-Performance und Vermeidung unnötigen Trainings. Memento zeigt, dass eine intelligente Datenselektion entscheidend ist, um repräsentative Daten zu erhalten und zu entscheiden, wann erneutes Training hilfreich ist.

Zweitens untersuchen wir das *Lernen über den Raum*, die Generalisierung von ML-basierten Modellen, und präsentieren einen Network Traffic Transformer (NTT). NTT ist ein vortrainiertes Transformer-Modell, das effizient auf verschiedene Netzwerke und Aufgaben feinabgestimmt werden kann.

Drittens untersuchen wir das fundamentale Problem des *Lernens latenter Netzwerkzustände*, das vielen Vorhersageaufgaben zugrunde liegt. Durch den Vergleich mehrerer ML-basierter Modelle für Video-Streaming präsentieren wir wichtige Erkenntnisse für Modellierung und Generalisierbarkeit.

Schließlich erforschen wir das Potenzial *programmierbarer Netzwerke*, bessere Signale zu liefern. Dies wird durch die programmatische Verarbeitung aller Pakete im Netzwerk ermöglicht, wenngleich die Rechenressourcen begrenzt sind. Wir präsentieren FitNets, ein System, das dieses Potential nutzt, um akkurate Verteilungen von Netzwerkverkehrsmerkmalen zu lernen.

PUBLICATIONS

This dissertation is based on previously published conference and journal proceedings. The list of publications (and a preprint) is presented hereafter.

On Sample Selection for Continual Learning: a Video Streaming Case Study

Alexander Dietmüller, Romain Jacob, Laurent Vanbever.
In *ACM SIGCOMM CCR*, Volume 54, Issue 2, 2024.

A New Hope for Network Model Generalization

Alexander Dietmüller, Siddhant Ray, Romain Jacob, Laurent Vanbever.
In *ACM HOTNETS*, Austin, TX, USA, 2022.

Learning Distributions to Detect Anomalies using All the Network Traffic

Alexander Dietmüller, Georgia Fragkouli, Laurent Vanbever.
Poster in *ACM SIGCOMM*, New York, NY, USA, 2023.

FitNets: An Adaptive Framework to Learn Accurate Traffic Distributions

Alexander Dietmüller, Albert Gran Alcoz, Laurent Vanbever.
arXiv preprint, arXiv:2405.10931, 2024.

The following publications (and a preprint) were part of my PhD research, but they were led by other researchers and are not referenced in this thesis.

The Long Way Home: Iteratively Addressing Underspecification Issues to Develop Last Hop Medium Classifier for Speed Tests

Roman Beltiukov, Alexander Dietmüller, Phillipa Gill, Arpit Gupta.
Under submission

P2GO: P4 Profile-Guided Optimizations

Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, Laurent Vanbever.
In *ACM HOTNETS*, Chicago, IL, USA, 2020

SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues

Albert Gran Alcoz, Alexander Dietmüller, Laurent Vanbever.
In *USENIX NSDI*, Santa Clara, CA, USA, 2020

pForest: In-Network Inference with Random Forests

Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, Laurent Vanbever.

arXiv preprint, arXiv:1909.05680, 2019.

CONTENTS

1

INTRODUCTION

The Internet is a decentralized and constantly growing control system. Since its beginnings as ARPANET connecting just four computers in 1969 [8], it has become an integral part of the lives of over 5.3 billion people [9]. With this scale comes a vast array of applications. Performant and robust control of these applications requires modeling network traffic, such as estimating whether the network is congested or how long it will take to transmit data. One application has grown particularly dominant over time: *video streaming*, which accounted for two-thirds of all Internet traffic in 2024 [10].

Applications like video streaming demand high performance. Controllers require accurate models to meet such demands, but the complexity of modeling network traffic has steadily increased over time: the modeling space is ever-growing, while observable signals remain largely unchanged.

Good models must consider other applications. Application traffic is governed by transport protocols like TCP that decide when to send how many packets of data, based on observations from past traffic, like throughput, loss, and latency. Early transport protocols followed a simple control loop to ensure reliable delivery. Oblivious of other applications, they would retransmit lost packets without further adjustments. This resulted in one of the Internet's first major challenges in the 1980s when an increasing number of applications began to send more traffic than networks could deliver. Network devices had to drop packets, causing retransmissions and thus even more traffic, resulting in a series of congestion collapses [11]. Nowadays, transport protocols model the expected behavior of other applications to adjust their sending rate to a fair share of the network without overloading it. Since the time of congestion collapses, a plethora of congestion control algorithms have emerged and it has become progressively difficult to model every possible application behavior in order to select an optimal rate [12].

Good models must also consider the network itself. Early networks aimed to minimally interfere with applications. To prevent packet loss during burst of traffic, networks features increasingly large buffers to keep up with growing bandwidths. Unfortunately, the road to hell is paved with good intentions: to maximize average throughput, congestion control algorithms tend to fill all buffers to capacity. The resulting 'Bufferbloat', always-full large buffers [13], led to drastic increases in latency, e.g., delays of seconds

or more for packets that could have been delivered in milliseconds. In response, networks had to diversify. Some networks could revert to smaller buffers, while others began to employ active queue management algorithms that intervene before the buffer fills, e.g., by dropping packets early [14]. If not modeled correctly, such different network behaviors cause applications to misinterpret observed signals, resulting in suboptimal performance [15].

Finally, good models must consider the interplay of these factors and more: applications like video streaming further complicate modeling by using additional control loops on top of transport protocols. The space of possible interactions has grown exponentially, yet we largely observe the same signals as in the 1980s. It has become extremely difficult, and perhaps intractable, to model network traffic from first principles.

Opportunities The development of numerous new application- and network algorithms has made modeling network traffic increasingly difficult, but the advance of technology also provides an opportunity: over the last decades, *machine learning* (ML) has become an indispensable tool to learn sophisticated models from data; models that can predict patterns too complex to specify manually. Today, both client and server hardware has grown powerful enough to enable applications like ML-based video streaming. At the same time, advances in network hardware led to *programmable networking*, empowering networks to share a wider range of information, expanding the space of observable signals and facilitating more accurate modeling.

This dissertation explores these opportunities, investigating how machine learning and programmable networks can aid network traffic modeling.

Adaptive network traffic modeling *How* can ML help to model network traffic? A fundamental challenge arises from the stagnating signal- and expanding modeling space: there are many latent (hidden) variables. The interplay of algorithms impacts throughput, loss, and latency, but the true causes are almost never directly observable. For example, a video streaming application may observe a rise in latency, which may be caused by other applications, large buffers, or queue management algorithms. While difficult to interpret, networks offer an abundance of data, enabling ML-based models to learn these hidden interactions. However, ML models face their own challenges that thus far have prevented widespread adoption.

A key factor in ML performance is how well the training data represents the network in which the model is deployed. Even the brief examples above illustrate how the network is constantly evolving, and models that are effective today may not be effective tomorrow. There is a need to keep models up-to-date, i.e., *adapt over time*, also known as continual learning.

We address this first challenge with Memento, a sample selection system for updating the training data. We show that smart data selection is crucial to both maintaining representative training data and deciding when to retrain.

Furthermore, increasing use cases for ML-based network traffic models present the opportunity to transfer knowledge between them. Yet presently, ML-based models are mostly developed in isolation, resulting in models that are difficult to reuse for different networks or tasks. In other words, there is an opportunity to *adapt over space*. We demonstrate that pre-trained Transformer models can be effectively fine-tuned for various networks and prediction tasks. We introduce a proof-of-concept Network Traffic Transformer (NTT) that illustrates the potential of our approach.

Subsequently, we examine the crux of the matter: *learning latent network state*. Regardless of the final prediction task, the underlying challenge is to predict the latent network state from a history of observed traffic. Despite its importance, this problem has not yet been studied extensively. For example: which observations reveal the most about latent variables? At which point do past observations cease to have an impact on the present? Through an in-depth analysis and comparison of several ML-based models for video streaming, we answer key questions about learning latent network state.

Finally, we explore programmable networks to bridge the gap of limited observability. Programmable network devices can flexibly process all packets in the network, yet are constrained in computational resources, limiting the operations per packet. We present FitNets, which makes the most of constrained programmability with hardware-software co-design: FitNets learns accurate distributions of network traffic features in the control plane, enabled by feedback through efficient model scoring in the data plane.

Dissertation outline Chapter 2 discusses learning over time. We outline challenges of continual learning in network traffic modeling and present Memento, a sample selection system to effectively keep models up-to-date.
Chapter 3 discusses learning over space. We present a vision for shared pre-trained models that can be fine-tuned to different networks and prediction tasks at comparatively low cost, and present NTT as a proof-of-concept.
Chapter 4 zooms in on the fundamental task of predicting the latent network state from observed traffic. We contrast different models to formulate and answer key questions about latent network state estimation.
Chapter 5 explores how network programmability can support learning distributions of network traffic features. We propose a hardware-software codesign for improved tail performance and fast adaptation.
Finally, Chapter 6 summarizes this dissertation and outlines future research.

2

LEARNING OVER TIME: A VIDEO STREAMING CASE STUDY ON SAMPLE SELECTION FOR CONTINUAL LEARNING

In this chapter, we discuss adaptive traffic modeling from the perspective of continual learning, that is, adapting models over time. Network algorithms and thus patterns change over time, and thus the model training data may not be representative anymore, potentially leading to performance degradation. Yet, to date, there is no established methodology to answer the key questions: *With which samples to retrain? When should we retrain?*

We address these questions with the sample selection system Memento, which maintains a training set with the "most useful" samples to maximize sample space coverage. Memento particularly benefits rare patterns—the notoriously long "tail" in networking—and allows assessing rationally *when* retraining may help, i.e., when the coverage changes.

We deployed Memento on Puffer, the live-TV streaming project, and achieved a 14 % reduction of stall time, a 3.5× higher reduction compared to retraining with randomly selected samples. While we primarily evaluate Memento in the context of video streaming, its algorithm does not depend on a specific model architecture, and it likely may yield benefits in other ML-based networking applications as well.

This chapter is organized as follows:
Section 2.1 motivates why continual learning is particularly important for tail performance and why a smarter sample selection is more helpful than using more data or larger models.
Section 2.2 proposes a sample-space-aware continual learning algorithm based on estimating sample space density to identify rare samples.
Section 2.3 presents Memento, a prototype implementation fo this algorithm.
Section 2.4 validates the effectiveness of our algorithm by applying Memento on an extensive video streaming case study. We deployed Memento on Puffer and collected 10 stream-years of real-world data over 9 months to confirm that we improve tail performance and avoid unnecessary retraining.
Section 2.5 presents microbenchmarks using data center workloads to show that our algorithm is not limited to models for video streaming.
Section 2.6 closes this chapter by putting Memento into context with related work and discussing the limitations and future directions of our work.

2.1 WHY SAMPLE SELECTION MATTERS

Adaptive bitrate (ABR) algorithms increasingly leverage ML to optimize their performance. Current ML-based ABR algorithms already achieve good average performance; i.e., high image quality while largely avoiding stalls [16, 17]. But even if rare, stalls matter: they impact user experience far more than image quality [18, 19]. Moreover, as networks evolve, maintaining performance over time is a concern that hinders the deployment of ML-based solutions. This challenge is known as continual learning [20–23].

To meet this challenge, we must improve tail performance over time while maintaining the (already good) average. For ABR algorithms, this goal translates to reducing stalls while maintaining quality.

The most common approach to improve ML performance is **using more**: more data, more training, more complex models. However, this does not guarantee (tail) improvements. **More training** does not help if done using the wrong samples. Similarly, naively using **more data** does not improve tail performance if it does not address the imbalance between average and tail samples. Finally, **more models** can help to address this if they cover different parts of the data distribution. Otherwise, the individual models fail to complement each other and provide no benefit. Besides its uncertain effectiveness, **using more** increases training and inference time. For large networking applications, this can be significant. If YouTube were to use ML-based ABR, it would require inference for ≈30 billion video chunks per day [24]: slower inference means higher costs in delay, energy, and money.

Instead of **using more**, we propose to improve performance with a *smarter selection* of samples, motivated by the Puffer study.

Case study Puffer is perhaps the best study of continual learning in networking. It monitors ABR performance of users streaming live TV with randomly assigned algorithms. Puffer's authors proposed their own ML-based ABR, retrained daily using random samples from the past two weeks.

To the author's surprise [16], retraining every day brought essentially no benefits: Over almost 900 days, it improved image quality by only 0.17 % over a static—never retrained—version (Figure 2.1, top). On the tail, daily retraining reduced the fraction of stream time spent stalled by 4.17 % on average, with large fluctuations over time (Figure 2.1, bottom). This illustrates the **more training** approach falling short.

But *why?* Why is the daily-retrained model not consistently outperforming the static one? Why does it stall only half the time in some months but twice as much in others (Figure 2.1, bottom)?

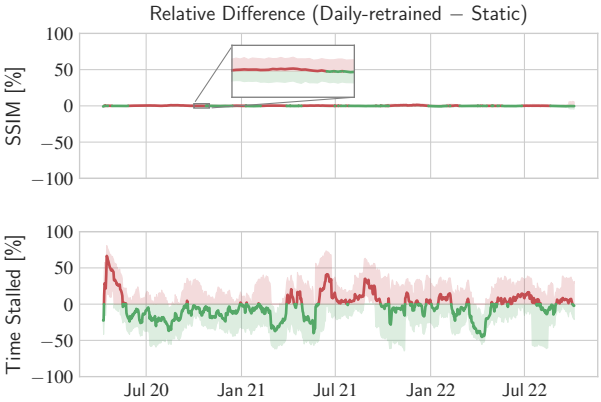


Figure 2.1: On Puffer [25], retraining daily with random samples did not outperform a never-retrained model consistently. On average, image quality and stream-time spent stalled differ by less than 0.2 % and 4.2 %.
Mean and 90% CI over a one-month sliding window. Data source: [25]

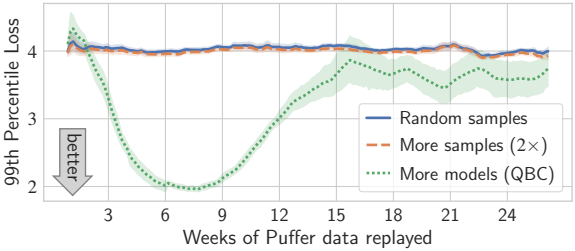


Figure 2.2: "Using more" does not guarantee better tail performance. More random samples have no effect. More models (QBC) initially help but degrade over time.
Mean and 90% CI over a two-week sliding window (see Section 2.4 for details).

We argue retraining did not help because training samples were selected randomly. Most streaming sessions perform similarly, leading to an imbalanced training set with many similar samples and few tail ones. Doubling the number of training samples—**more data**—does not help (Figure 2.2): we must address this imbalance to improve a model's tail performance.

Query-By-Committee (QBC) [26] is a classic approach to address such imbalance by selecting samples where a committee of models disagrees and prediction entropy is high. When applying QBC to the Puffer data, we observe initial improvements that vanish over time. The models fail to identify rare samples reliably and eventually overfit on noise (Figure 2.2). Using **more models** is not enough to improve tail performance.

Problem Figure 2.1 and Figure 2.2 suggest that retraining *can* improve tail performance, but we observe common approaches like random sampling or QBC to be ineffective or unreliable over time. If the selection got lucky, retraining improved the tail; other times, it was useless or even detrimental. We argue that we can do better with a smarter selection strategy that reliably picks up important tail samples. In summary, this leads to two questions:

1. From a stream of new samples, *with which samples should we retrain* to improve performance over time?
2. From an updated set of training samples in memory, *when should we retrain the model?* That is, can we avoid retraining "for nothing"?

Fundamentally, an algorithm addressing these questions requires: *(i)* a signal to *select important samples*, able to identify the tail; *(ii)* a signal to *quantify changes* to decide when to retrain; *(iii)* a mechanism to *forget* noise and outdated samples to avoid degradation over time (QBC in Figure 2.2). Finally, resource usage should be small compared to **using more**.

Solution We propose a sample-space-aware continual learning algorithm based on *coverage maximization*, i.e., prioritizing samples from low-density areas of the sample space, and present Memento, a prototype implementation of this idea. Figure 2.3 shows how Memento integrates with an ML-based ABR, where a *sample* encompasses telemetry data for a video chunk.

Memento estimates the sample-space density from both new and in-memory samples, prioritizing rare low-density samples to address dataset imbalance and improve tail performance. Memento uses the *difference in coverage* to assess whether there is something new to learn. If so, it retrains.

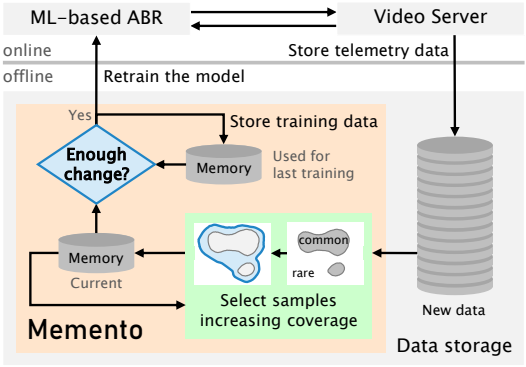


Figure 2.3: Memento selects samples to maximize sample space coverage, improving the tail performance while rationalizing when to retrain.

2.2 A CASE FOR DENSITY

We propose a continual learning algorithm based on estimating sample space density. Estimating density allows the algorithm to prioritize rare low-density samples, ultimately maximizing the sample space coverage. This section provides intuition as to why we propose this selection metric instead of per-sample metrics like sample loss or QBC's entropy.

Density for sample selection "The tail" is not a single pattern that rare traffic follows, but rather many patterns with only a few samples each. A random sample selection mirrors imbalances in the underlying distribution, e.g., over-represented common traffic patterns. This leads to *diminishing returns* as we sample more and more from common patterns and little from the tail. As illustrated by Figure 2.1, this yields good average performance but is unreliable at the tail. We need to improve the training data.

To correct dataset imbalance, we need a sample-space-aware selection. Traditional approaches use the model performance (e.g., entropy, loss, reward) to select samples [21, 22]. We found that this does not work well over the long timespan of the Puffer because it fails to *avoid catastrophic forgetting*. By considering each sample independently, they fail to preserve samples with good performance representing 'normal' traffic.

Instead, we propose to select samples based on the density of their neighborhood, which considers the whole sample space: a *sample-space-aware selection* that aims to *maximize coverage* of the sample space. The key insight to maximize coverage is to retain samples if only few similar samples exist. We achieve this by removing samples from high-density areas with the most similar samples. This decreases the density and we naturally stop removing further samples, leaving us with a diverse set of samples.

Figure 2.4 illustrates the benefits of samples-space-aware density versus sample-aware loss as a selection metric. In this experiment, we use the static model from Puffer's authors and 5 M samples collected by Puffer over a few days. We create batches of 256 samples and compute their loss and density (as detailed in Section 2.3). The top row of Figure 2.4 shows the mean loss and density per batch. Using each metric, we select 1 M samples and retrain the model. The bottom row compares the effect of each metric over the 5 M samples by showing the mean loss improvement per batch.

The left column shows that loss-based selection focuses *too much* on the tail, i.e., high-loss batches. It yields good improvements there but fails to preserve common samples, degrading performance on most other batches.

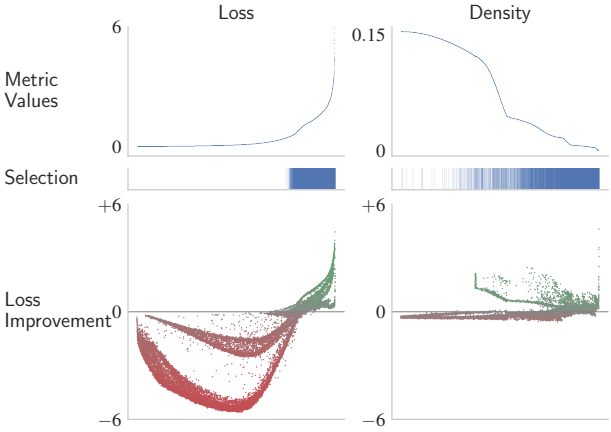


Figure 2.4: ***Loss improvement obtained by retaining with a selection of*** 1 M ***samples over a dataset of*** 5 M. ***The same batches of 256 samples are used for the loss- (left) and density-based selection (right).***
To improve tail performance, we need many low-density batches because they are all different. To maintain average performance, we need only a few high-density batches, as they are similar. Selecting based on density (right) achieves both. Conversely, loss-based selection (left) is too specific. It suffers from diminishing returns as it selects only high-loss batches and catastrophically forgets the average.

Conversely, the right column illustrates that density-based selection is more holistic, focusing on the tail while covering the entire sample space. Performance is improved at the tail, i.e., on low-density batches, without much degradation on the high-density ones. One may object that the loss selection could be tuned to maintain more "low-loss samples." We tried this and our evaluation shows that it does not perform as well as density and can easily get much worse (Section 2.4.4 and Figure 2.8).

These results also shed some light on why we observed QBC degrade over time (Section 2.1, Figure 2.2). Overall, QBC has some inertia, as models are gradually replaced (more details in Section 2.4.4), but we ultimately observe a similar effect as loss-based selection. Noisy samples are, by definition, random and have high entropy. Consequently, they are preferred by QBC. Over time, QBC accumulates noise and forgets the average and more common tail samples. As a result, it is unable to maintain the initial performance improvements on the tail.

Density for shift detection Continual learning aims to adapt when the data distribution changes. Data distribution changes can be broadly categorized into two categories: *(i) covariate shift* [27], i.e., previously unseen traffic patterns emerge or the prevalence of patterns changes; and *(ii) concept drift* [28], i.e., the underlying network dynamics change. Density-based sample selection is an effective tool to capture both kinds of changes.

When new traffic patterns appear, e.g., users starting to stream over satellite networks, they populate a previously-empty area of the sample space, resulting in a low density and, thus, a high probability of being selected for retraining. Similarly, patterns becoming less prevalent results in low density, and a high probability of remaining selected.

When the underlying network dynamics change, e.g., a new congestion control gets deployed, we should forget old samples that are no longer relevant. However, detecting those changes is difficult, and being wrong risks forgetting useful information such as hard-to-gather tail samples.

Using density for sample selection *correlates the probability of forgetting samples with how important it is to remember them.* Samples from dense regions are discarded readily, as we will likely get more of those samples. Conversely, low-density batches are less likely discarded as we only encounter these samples infrequently. This makes the selection more conservative at the tail, allowing to remember tail patterns; the model will perform well on similar traffic if it recurs. If it does not, i.e., it was essentially noise, then it will eventually be forgotten. We provide some empirical evidence of recurring patterns in the tail of the Puffer traffic in Appendix A.1.

Algorithm 2.1: Memento Coverage Maximization

Parameters: Capacity C, threshold τ,
batch size b, bandwidth h, temperature T
Input: In-memory S_{mem} and incoming S_{new} samples
Output: Selected samples S^*, decision *retrain*

begin
  $S^* \leftarrow S_{new} \cup S_{mem}$
  $B' \leftarrow \{\}$; // or last train batches
  retrain $\leftarrow$ *False*

  // Section 2.3.2: Distance measurement
  $B \leftarrow BatchSamples(S^*, b)$
  $B \leftarrow BBDR(B)$; // $(x, y) \rightarrow (\hat{y}, y)$
  $(D^{pred}, D^{out}) \leftarrow DistributionDistances(B)$

  // Section 2.3.3: Density estimation
  $\widehat{\rho}^k \leftarrow KDE(D^k, h)$ $\forall k \in \{pred, out\}$
  $\widehat{\rho} \leftarrow \min(\widehat{\rho}^{pred}, \widehat{\rho}^{out})$

  // Section 2.3.4: Sample selection
  while $|S^*| > C$ **do**
    $p^{discard} \leftarrow \text{soft}\max(\widehat{\rho} \,/\, T)$
    $i \leftarrow WeightedRandomChoice(p^{discard})$
    $S^*, B \leftarrow DiscardBatch(S^*, B, i)$
    $\widehat{\rho}, D \leftarrow UpdateDensities(\widehat{\rho}, D, i)$

  // Section 2.3.5: Retraining decision
  if $RCI(B, B') \geq \tau$ **then**
    retrain $\leftarrow$ *True*
    $B' \leftarrow B$; // remember train batches

2.3 COVERAGE MAXIMIZATION WITH MEMENTO

The core of Memento is Algorithm 2.1 which aims to maximizes sample-space coverage. Memento achieves this by estimating the sample space density: the higher the density of sample (i.e., the more other samples are close) the better the memory covers this part of space. Memento approximates optimal coverage by iteratively discarding samples, assigning a higher discard probability to high-density regions. It proceeds in four steps:

1. It computes pairwise distances between sample distributions, using batching and *black-box dimensionality reduction* to scale (Section 2.3.2);
2. It estimates density using *kernel density estimation* (Section 2.3.3).
3. It discards surplus batches *probabilistically*, mapping density to discard probability to balance tail-focus and noise rejection (Section 2.3.4).
4. It approximates how much the memory coverage has changed to decide whether retraining might be beneficial (Section 2.3.5).

Memento's sample selection relies on three internal parameters: the batch size b, the KDE bandwidth h, and the probability mapping temperature T. We provide default choices for these parameters in Section 2.4.2 and analyze the impact of each parameter on Memento's performance in Section 2.4.4.

2.3.1 *Definitions*

Process We consider a process $y = f(x)$ that maps inputs $x \in \mathcal{X} = \mathbb{R}^n$ to outputs $y \in \mathcal{Y}$, where $\mathcal{Y} = \mathbb{R}$ in regression or $\{1, 2, \ldots, k\}$ in classification problems. In the context of ABR, f models the network dynamics mapping traffic features x (e.g., video chunk size, TCP statistics, transmission times of past packets) to a prediction for the next chunk y (e.g., the current bandwidth or expected chunk transmission time).

Predictions We consider a model $\hat{f}$ that is trained to predict $\hat{y} = \hat{f}(x)$ from a set of samples $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, i.e., a supervised setting.

Replay memory To account for concept drift or covariate shift, we retrain $\hat{f}$ with an updated set of samples S^*, stored in a *replay memory* with capacity C. A sample selection strategy decides how to update this memory.

Sample selection strategy Given a set S_{new} of new samples available and a set S_{mem} of samples currently stored in memory, with $|S_{new}| + |S_{mem}| > C$, a selection strategy must select $S^* \subset (S_{new} \cup S_{mem})$, such that $|S^*| \leq C$.

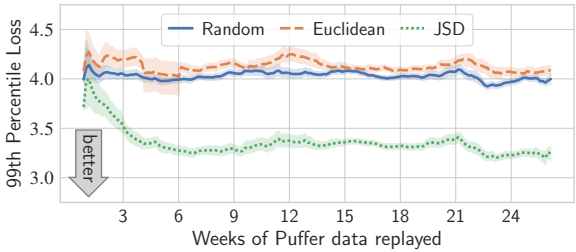


Figure 2.5: Selecting samples based on the Euclidean distance between batch averages does not improve tail performance. Computing the Jensen-Shanning distance between batch distributions is a better choice.
Mean and 90% CI over a two-week sliding window (see Section 2.4 for details).

2.3.2 *Distance measurement*

Batching Before discussing the distance computation, we need to consider scalability. Algorithm 2.1 has two main bottlenecks: *(i)* computing pairwise distances ($\mathcal{O}(n^2)$); and *(ii)* updating density estimates after discarding a sample ($\mathcal{O}(n)$). Puffer [25] currently collects over 1 M samples daily, and processing each sample individually would consume a prohibitive amount of resources for Memento to be practical. Memento scales by aggregating samples in batches, computing distances between aggregates, as well as discarding samples and updating densities for an entire batch at once.

Batching improves scalability but reduces the flexibility of sample selection. For example, if a common and a rare sample are aggregated in the same batch, they can only be kept or discarded together. To avoid suboptimal aggregation, samples are first grouped by outputs, then by predictions, and finally split into batches; this pools samples spatially to create homogeneous aggregates. We found this approach to work best, but Memento also supports batching based on sampling time or application-specific criteria.

Distribution distances A natural first choice for batch aggregation is to average sample vectors and compute the Euclidean distance between the averages. However, we find that this approach is flawed: averaging "dilutes" rare values in a batch and the resulting distance is not sensitive to tail differences. Figure 2.5 shows that selecting samples based on the Euclidean distance between batch averages does not improve tail performance compared to a random sample selection.

Instead, we aggregate batches by computing an empirical sample distribution.[1] This preserves rare values, but we still need to reduce the dimension to gain any computational benefit from batching. We address this problem using *black-box dimensionality reduction* (BBDR), an approach inspired by research on dataset shift detection [29] that has been found to outperform alternatives like PCA [30]. The idea is simple: Use the existing model—trained to identify important feature combinations—to predict $\hat{y}$ (the predicted transmission time), then compute distances in the low-dimensional prediction space. In essence, BBDR leverages prediction as dimensionality reduction tailored to the task at hand.

Finally, we compute the distance between batch distributions using the Jensen-Shannon Distance (JSD) [31] which is closely related to the Kullback-Leibler Divergence (KL) [32]. The KL is an attractive choice to improve tail performance as it puts a large weight on rare values, i.e., values with high probability in one distribution but not in the other. However, the KL has two drawbacks. It is: *(i)* asymmetric, only sensitive to rare values in one distribution; and *(ii)* only a divergence, while common density estimation methods require a distance (see below). Thus, we choose the closely related JSD, which is both symmetrical and a distance metric [31]:

Definition 2.1 (Jensen-Shannon Distance)**.** *Let P and Q be probability distributions, and let $M = \frac{1}{2}(P + Q)$. The Jensen-Shannon Distance between the distributions P and Q is defined by:*

$$JSD(P,Q) = \sqrt{\tfrac{1}{2}\left(KL(P,M) + KL(Q,M)\right)}$$

$$\textit{where} \qquad KL(P,M) = \sum_{x \in \mathcal{X}} P(x) \log_2 \left(\frac{P(x)}{M(x)}\right)$$

if P and Q are discrete distributions in the space $\mathcal{X}$; or

$$KL(P,M) = \int_{-\infty}^{\infty} p(x) \log_2 \left(\frac{p(x)}{m(x)}\right) dx$$

if P and Q are continuous probability distributions with probability density functions p and q, and $m = \frac{1}{2}(p + q)$.

1 This may also be interpreted as replacing distances between observations (samples) with distances between the underlying processes (distributions).

Inputs and Outputs It is not clear a priori whether distances in the prediction or output space are more important for tail performance. Thus, in supervised classification or regression problems—where outputs are available—Memento considers both equally and computes separate distances for both spaces, which are combined below (Section 2.3.3).

Probabilistic predictions During batching, Memento is able to leverage probabilistic predictions; e.g., the transition time predictor in Puffer [16] outputs a probability distribution over 21 transition time bins. This probability distribution captures whether predictions are somewhat uncertain—indicating a need for training using more such samples—or certain. However, we must handle probabilistic estimates differently than point estimates: *(i)* When batching, Memento groups probabilistic predictions first by the distribution mode, then by their probability; *(ii)* The batch distribution is computed as a mixture distribution: let P_j be the probabilistic prediction of sample j in batch i, and $|b_i|$ the number of samples in batch i. Then, the distribution for batch b_i is is $P_i(x) = (1/|b_i|) \cdot \sum P_j(x)$.

2.3.3 *Density estimation*

From the sample distances, we can compute the prediction- and output-space densities. Since we do not know the topology of the sample space, we cannot use simple approximations, such as the fraction of samples per cluster. A common general approach is kernel density estimation (KDE) [33].

Definition 2.2 (KDE)**.** *Let b be a sample batch and B a set of batches, and let $d^k(b, b') = JSD(P^k, P^{k'})$ with $k \in \{pred, out\}$ be the prediction or output distance between batches b and b' with distributions P^k, $P^{k'}$. Then, using a Gaussian kernel with* bandwidth *h, the kernel density estimate at the location of batch b is:*

$$\widehat{\rho}_B^k(b) = \frac{1}{\sqrt{2\pi}\,|B|} \sum_{b' \in B} \exp\left(-\frac{d^k(b, b')^2}{2h^2}\right) \tag{2.1}$$

Intuitively, the kernel density estimate of b is inversely proportional to the distances to other batches, with diminishing weights for larger distances. The bandwidth parameter h, also called "smoothing factor", determines how quickly this weight drop-off occurs.

Density aggregation Memento considers the prediction and output space as equally important. Thus, we aggregate densities using the minimum: we regard the batch as rare if its density is low in either space:

$$\widehat{\rho}_B(b) = \min\left(\widehat{\rho}_B^{pred}(b), \widehat{\rho}_B^{out}(b)\right) \tag{2.2}$$

If necessary, this approach can be generalized to fewer or more densities: for unsupervised learning without ground truth, Memento can only consider $\widehat{\rho}^{pred}$. For models with multiple outputs and thus multiple predictions, we can compute the minimum in Eq. (2.2) across all respective densities.

Alternative distances We leverage an information-based distance metric to focus on the tail. However, the density estimation is not tied to this method. Memento can work with any alternative provided distance metric.

2.3.4 *Sample selection*

Memento optimizes the sample-space coverage by iteratively discarding high-density batches. It computes the densities for all samples in $S^* = S_{new} \cup S_{mem}$, discards with a density-dependent probability (Algorithm 2.1, Line 10–14) until $|S^*| \leq C$, and updates densities after each discard.

Intuitively, Memento assigns a high discard probability to batches with high density—batches for which many similar samples exist—thereby protecting rare low-density samples. However, because noisy samples also seem "rare," we must retain some probability of discarding rare samples. Memento achieves this by mapping densities to probabilities using softmax with temperature scaling [34]:

$$p^{discard} = \text{soft}\max\left(\widehat{\rho}_B / T\right) \tag{2.3}$$

where $\widehat{\rho}_B$ is a vector of densities for all batches $b \in B$, and $p^{discard}$ is a corresponding vector of discard probabilities.

The temperature T allows balancing tail-focus with noise rejection. A low temperature assigns a higher discard probability to the highest-density batch(es). Conversely, a high temperature assigns more uniform discard probabilities, increasing the probability of discarding low-density batches. At the extreme, the discard probability with $T \to 0$ is a point mass; if Memento is configured with T=0, we thus deterministically discard the highest-density batch. With $T \to \infty$, the probability becomes uniform.

2.3.5 *Retraining decision*

Intuitively, retraining is beneficial if we collect *new* information, i.e., samples in areas of the sample space that were previously not covered. That is, we should retrain only when the coverage of sample space increases. Memento's density estimation allows estimating this increase in information to guide the retraining decision.

Definition 2.3 (Coverage increase)**.** *Let B be a set of batches with density estimates $\widehat{\rho}_B$. We can approximate the region of the sample space covered by B:*

$$Coverage(B) = \sum_{b \in B} \widehat{\rho}_B(b) \tag{2.4}$$

Let B' be a second set of sample batches. We can approximate the coverage increase (CI) of B with respect to B', i.e., the region of space covered by B but not by B':

$$CI(B, B') = \sum_{b \in B} \min\left(\widehat{\rho}_B(b) - \widehat{\rho}_{B'}(b), 0\right) \tag{2.5}$$

The relative *coverage increase RCI from B' to B is then*

$$RCI(B, B') = CI(B, B') / Coverage(B) \tag{2.6}$$

where $RCI(B, B') \in [0, 1]$; 0 means that the same area of sample space is covered, while 1 indicates that B covers an entirely different region of the space than B'.

Hence, with B the current memory batches and B' those used for the last model training, $RCI(B, B')$ estimates the coverage increase since the last training. If it exceeds the user-defined threshold τ, Memento triggers retraining. This approach presents a rational trade-off: the larger τ, the longer we wait for changes to accumulate before retraining. With a small τ, the model compensates for changes quicker at the cost of more retraining. Memento's training decision is sample-space-aware, it gives a rational argument that retraining is likely to be beneficial (even if there is no guarantee).

2.4 EVALUATION: REAL-WORLD BENEFITS

We continue to use Puffer to evaluate Memento's real-world benefits. This experiment aims to show that Memento improves the tail performance of *existing models* reliably without significantly impacting the average. Puffer provides both data collected daily over several years and a publicly available model that we can retrain with Memento and compare against the original. This makes Puffer a perfect case study to investigate the following questions:

Q1 *Does* Memento *improve the tail predictions?* ***Yes***
Over years of live and replay data, Memento significantly improves the 1st percentile prediction score.

Q2 *Does it improve the application performance?* ***Yes***
On live Puffer, over 10 stream-years of data, Memento achieves a 14 % smaller fraction of stream-time spent stalled with only 0.14 % degradation in image quality.

Q3 *Does* Memento *avoid unnecessary retraining?* ***Yes***
Memento retrains 4 times in the first 8 days, and only 3 times in the following 9 months (7 times in total).

Q4 *Are our improvements replicable?* ***Most likely***
Memento benefits appear replicable over different time periods of Puffer data. Its design parameters are intuitive and easy to tune.

Q5 *Can* Memento *benefit existing solutions?* ***Yes***
Memento further improves tail predictions achieved by more advanced training or prediction strategies.

2.4.1 *The Puffer project*

The Puffer project is an ongoing experiment comparing ABR algorithms for video streaming [25]. Puffer streams live TV with a random assignment of ABR algorithms and collects Quality-of-Experience (QoE) metrics: the mean image quality measured in SSIM [35] and the time spent with stalled video.

Fugu is the ABR algorithm proposed by Puffer's authors using a classical control loop built around a *Transmission Time Predictor* (TTP), a neural network predicting the probabilities for a set of discretized transmission times. The TTP was retrained daily with 1 M samples drawn from the past 2 weeks: ∼130 k samples from the latest day and ∼10 % fewer for day before. Fugu_{Feb} is a static variant, trained in February 2019 and never retrained.

Fugu was discontinued only 17 days after Memento's current deployment. Hence, we can only compare Memento's long-term performance to Fugu_{Feb}. As discussed in Section 2.1, Fugu and Fugu_{Feb} perform similarly: over almost three years, Fugu showed an SSIM improvement of 0.17 % and a reduction in the time spent stalled of 4.17 %. Thus, improvements over Fugu_{Feb} likely translate to similar—yet slightly lesser—improvements over Fugu.

2.4.2 *Retraining with Memento*

Every day, we use Memento to select training samples and decide whether to retrain Fugu's TTP. We assess Memento's benefits in two experiments:

deployment We deploy on Puffer two Memento variants, i.e., two variants of Fugu using Memento for retraining the TTP. One uses Memento's default parameters (see below), and the other deterministic sample selection (i.e., using temperature $T = 0$, Section 2.3.4). We collected data over 292 days (Oct. 2022 to Jul. 2023), about 10.8 stream-years of video data per variant. This allows answering **Q1**, **Q2**, and **Q3**.

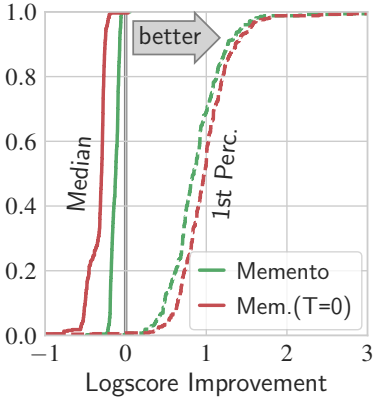
replay To confirm the deployment observations, evaluate design choices, and benchmark the impact of Memento's parameters, we replay Puffer data collected since 2021. To reduce the bias from a particular starting day, we replay 3 instances with 6 months of data each, a total of 90 stream-years of video data. This allows answering **Q4** and **Q5**.

Metrics We access Memento along three dimensions:

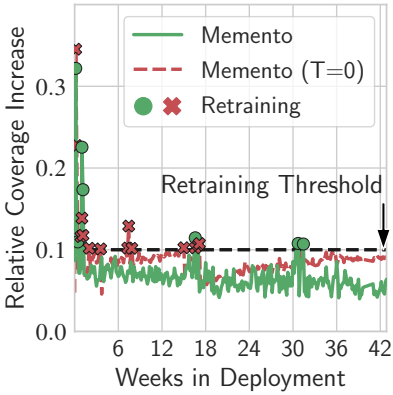
- *Prediction quality* is measured with $logscore(y) = \log p(y)$ [36]; y is the video chunk transmission time and $p(y)$ the TTP-predicted probability. Score improvement equals TTP loss decrease.[2]
- *Application performance* is measured with user QoE; Only available for the deployment experiment, where real user streams are impacted by the predictions and the resulting QoE can be measured.
- *Training resource usage* is measured by the number of retraining events.

Parameters We set Memento's memory capacity C to 1 M samples (same as the original Fugu model). Default parameters are a retraining threshold τ of 0.1, batching size b of 256, kernel bandwidth h of 0.1, and temperature T of 0.01. We evaluate the impact of different parameter values in Section 2.4.4.

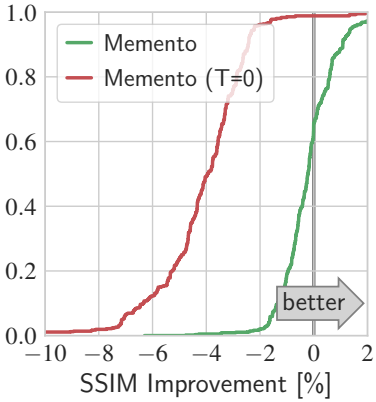
2 The logarithmic score is a commonly used metric for probabilistic predictions [36] like those produced by the Puffer TTP model. It is closely related to the *cross entropy loss* used to train the TTP: this is also known as the *logarithmic loss* and is the negative logarithmic score.



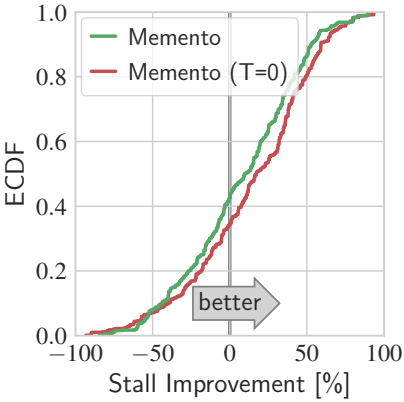
(a) Memento improves the prediction quality at the tail with only a small impact on the median prediction quality.



(b) Memento avoids unnecessary retraining. Its deterministic variant ($T = 0$) is less efficient and does not converge.



(c) Memento degrades the SSIM slightly (0.14 % worse). The det. variant degrades it notably (4.4 % worse).



(d) Memento reduces the fraction of stream-time spent stalled by 14 %, although results vary by day.

Figure 2.6: Memento achieves its goal: it improves the tail prediction quality with minimal impact on the average (Figure 2.6a). This requires little retraining (Figure 2.6b) and leads to notable QoE improvements (Figures 2.6c and 2.6d).

2.4.3 *Deployment results*

In this section, we show that Memento improves the tail prediction quality with little retraining and that this leads to QoE improvements over Fugu_{Feb}. In Appendix A.1, we provide additional plots with the evolution of QoE and predictions over time, as well as aggregate plots like those published on the Puffer paper and website [16, 25].[3]

Prediction quality Figure 2.6a shows the score difference ECFD between Memento and Fugu_{Feb} in median and 1st-percentile scores (higher is better). We observe that Memento improves tail predicition performance as intended, with a slight—yet expected—degradation on average: as memory is finite and Memento purposefully prioritizes "rare" samples, it must remove samples for the most common cases. The deterministic version of Memento prioritizes the tail more aggressively, which leads to slightly better tail improvements and worse median degradation.

Retraining count Figure 2.6b shows the relative coverage increase *RCI* between the current memory and the one last used to retrain the model, as well as the retraining threshold $\tau = 0.1$. We observe about eight "warm-up" days where Memento retrains four times. Afterward, RCI remains low and retraining due to changes in the data (RCI peaks) happens three times. This shows that there are fewer "new patterns" to learn from over time; retraining daily is unnecessary, but still beneficial at critical times.

Conversely, the deterministic variant of Memento keeps accumulating samples, exhibiting a different RCI pattern. After training five times during "warm-up," it trained 9 more times. Where RCI for default Memento stabilizes, the RCI of the deterministic variant slowly but steadily increases over time while it accumulates rarer and rarer samples, as it does not reject noise. Essentially, this variant "never forgets."

Application performance Figures 2.6c and 2.6d show the relative QoE improvements over Fugu_{Feb} for SSIM and time stalled (higher is better).[4]

First, we observe that Memento only marginally affects the SSIM; the average SSIM is 17.12 and 17.14 for Memento and Fugu_{Feb}, resp. (not shown). Memento's deterministic variant affects the SSIM more; its average is 16.39 (not shown) and the SSIM is consistently worse than Fugu_{Feb} (Figure 2.6c).

3 Our results per algorithm differ from official Puffer plots, as Puffer excludes some sessions in an attempt to exclude effects such as "client decoder too slow," while we consider all data points. As the filtering is ABR-independent, it does not impact relative results between ABRs.

4 To avoid bias towards either Memento or Fugu, we show the symmetric percent difference using the maximum: $100 \cdot (x - y)/\max(x,\ y)$.

Second, Figure 2.6d shows that stall improvements compared to Fugu_{Feb} are almost equal for both variants of Memento, even though there are large day-to-day variations: some days, Fugu_{Feb} stalls much less than Memento, and vice versa. Over the entire 292 days of deployment, Memento spent a fraction of 0.2 % of stream-time stalled, compared to 0.24 % for Fugu_{Feb}: Memento spent a 14 % smaller fraction of stream time stalled than Fugu_{Feb}.

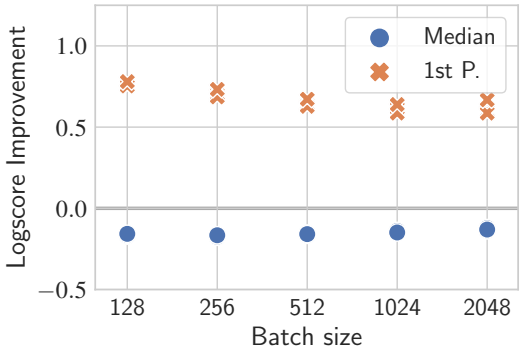
In the results above, we already see that Memento performs slightly worse without noise rejection (i.e., with $T = 0$). In a previous deployment, we observed that never forgetting ultimately prevented enough average samples from remaining in memory, which destroyed the average performance (Appendix A.1). The latest version of Memento made the deterministic variant more robust but we can still observe the signs of noise accumulation (worse predictive performance, more frequent retraining, steadily rising RCI). By contrast, the probabilistic default Memento naturally forgets noise and stabilizes, as can be seen in the RCI in Figure 2.6b.

Finally, we observe that Memento reduces stalls by 3.5 times as much as retraining daily with random samples. In Appendix A.1, Figure A.4 we show Figure 2.6a overlaid with the score improvements of Fugu in the past.[5] We observe that random retraining improved the tail prediction scores significantly less and even worsened them on 20 % of days.

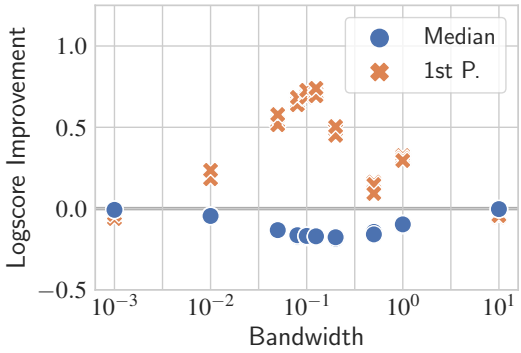
From predictions to QoE One may wonder why the average prediction degradation (Figure 2.6a) does not seem to strongly impact image quality (Figure 2.6c), and, conversely, why significant tail improvements yield only a modest reduction in stalls (Figure 2.6d). Our results illustrate the complex relationship between predictions and QoE, including the closed-loop control logic, which uses the predictions and aims to keep the video buffer at the receiver sufficiently full to avoid stalling.

Taking a closer look at the experiment results, we noticed that the transmission time of most chunks is very small, and most prediction errors are also small time-wise. Hence, slightly worse predictions have little effect on the buffer fill level; the controller has time to compensate and maintain image quality. Moreover, since most prediction errors overestimate the transmission time (not shown), it makes the closed-loop control more conservative. Thus, it manages to keep stalls low but struggles to maintain high image quality (compare Figures 2.6c and 2.6d).

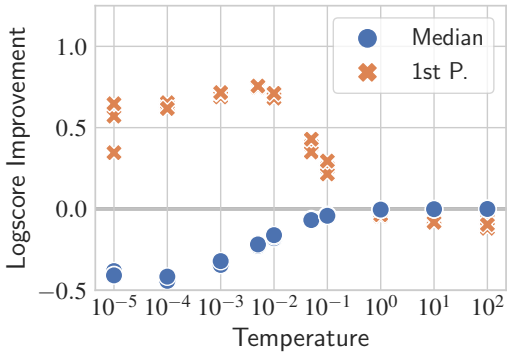
5 Appendix A.1, Figure A.4 must be considered with caution, as the underlying data comes from different time periods and may not be comparable.



(a) Larger batches slightly degrade the tail with no impact on the median.



(b) A kernel bandwidth around 0.1 is best. Extreme values nullify benefits.



(c) $T = 0.01$ is a good trade-off between tail prioritization and randomness.

Figure 2.7: Each marker shows the median prediction improvement of Memento over Fugu (random selection) over a 6-month replay, measured as median and 1st perc. improvement each day. Results are consistent across replays.

2.4.4 *Replay results*

In this section, we confirm that Memento's benefits are replicable and not just an artifact from deploying at an "easy time," its design decisions are justified, and it complements existing techniques. To do this, we replay 3 non-overlapping instances of 6 months with 25, 39 and 26 stream-years of video-data respectively and 3 M samples on average per day.

To monitor the memory quality over time, we disable threshold-based retraining and retrain every 7 days; one must retrain and test the model to assess whether the right samples were selected. We evaluate each day in terms of prediction improvement over retraining with random samples (Fugu) and report the mean over each 6-month period. We show the entire time series for each experiment in Appendix A.1.

Replicability All plots in Figure 2.7 show three data points per setting, which are average performance numbers over the entire 6 month period. We observe that all results are fairly stable, which gives reasonable confidence about the replicability of Memento's benefits on this use case.

Batch size Figure 2.7a shows Memento's prediction performance over the batch size; larger sizes improve scalability but make sample selection coarser, which can hurt performance (Section 2.3.2). We observe a slight tail performance drop for large batch sizes but little change on average.

Regarding scalability, differences are more pronounced: using a single CPU core to process 4 M samples takes on average 200 s with a batch size of 128, 48 s with 256 (the default), and 7 s with 1024. Benefits flatten out for larger batch sizes. Overall, computation is dominated by distance computation; batching the samples takes only about 2.5 s.

Bandwidth Memento estimates how close nearby batches are; the kernel bandwidth h determines what "nearby" means (Section 2.3.3). As the computed distances $JSD(P, Q) \in [0, 1]$, bandwidths > 1 over-smooth (all batches are always "nearby"), and bandwidths $\ll 1$ under-smooth (no other batches are ever "nearby"). Both cases nullify the idea of estimating density, effectively making the sample selection random. Figure 2.7b confirms this intuition: at the extremes, Memento performs like a random selection. We obtain the best tail improvement with a bandwidth around 1×10^{-1}.

Temperature Figure 2.7c shows Memento's prediction performance over the temperature T; a low temperature strongly prioritizes rare samples at the risk of accumulating noise, while a high temperature rejects noise by making the sample selection more random (Section 2.3.4). As expected, a

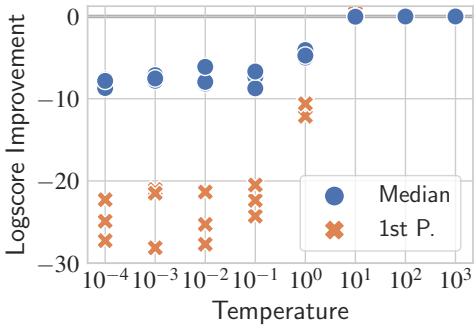


Figure 2.8: Loss-based selection is worse and less robust.

lower temperature yields better tail performance but degrades the average. The trade-off is not linear, though; we can select a temperature that provides tail benefits with minimal impact on the average. The best trade-off is a temperature around 1×10^{-2}.

Alternative selection metrics Figure 2.8 shows the performance of using loss as an alternative selection metric: we still use temperature-based probabilistic selection but discard samples with a low loss instead of high density. At best, it gives half the tail improvements of Memento, and it is much harder to tune: the benefits vanish for a slightly higher temperature. With a lower temperature, i.e., selecting more strongly based on loss, performance decreases drastically, which mirrors our observations in Section 2.2.

We evaluate additional metrics in Appendix A.1: prediction confidence, label counts, and whether a sample belongs to a stalled session or not. In summary, these perform worse or equal to loss-based selection in the best case, and most of them are as sensitive to tune. Probabilistic selection based on density performs better and is less sensitive (see Figure 2.7c). We also show detailed results for Euclidean distances (Section 2.3.2).

Alternative training decision We compare Memento's retraining decision based on the *relative coverage increase* RCI with a loss-based decision (not shown). We observe that for samples selected by Memento, either decision is effective. Overall, a coverage-based decision provides greater control over retraining frequency but struggles with low thresholds (e.g., 5 %). As Memento is probabilistic, the estimated RCI fluctuates at each iteration, which can be observed in Figure 2.6b, and the retraining threshold should be set above these fluctuations. It may be possible to further improve Memento by smoothing the RCI or by filtering the random fluctuations.

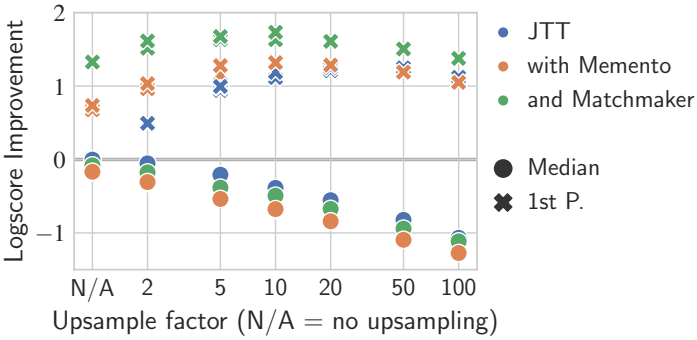


Figure 2.9: Memento complements training and prediction improvements from orthogonal algorithms such as JTT and Matchmaker.

Memento + Matchmaker + JTT MatchMaker [37] improves predictions by using an ensemble of models (the default ensemble size is 7) combined with an online algorithm to select the best model to make a prediction for each sample. However, this is limited by the performance of the individual models. Using an ensemble of models trained with a random selection, even with an oracle choosing the best model, we can only improve tail performance by half as much as a single model trained with Memento. However, we can get the best of both worlds by using MatchMaker with an ensemble of Memento-trained models, which yields double the tail performance with less decrease in median performance compared to a single Memento-trained model ('no upsampling' in Figure 2.9).

Just train twice (JTT) [38] improves performance by, as the name implies, training twice: after the first training, misclassified samples are upsampled in the second and final training. Figure 2.9 shows the same performance tradeoffs for JTT and Memento: both improve the tail and degrade the median: JTT with an upsampling factor of 3 is roughly equivalent to Memento's sample selection with 'normal' training. Yet this comes at different resource costs: JTT requires up to double the training time and resources, depending on how long the first training step is. Training models like Puffer takes time in the range of hours [16] and often requires expensive hardware (e.g., GPUs). Memento is more resource efficient: even on a single CPU core, it can process millions of samples in a few minutes (see above). However, JTT and Memento are not in competition, but complementary. We observe the best performance by combining Memento-selected samples with JTT's training and observe even further improvements when the resulting models are used in a MatchMaker ensemble for predictions (Figure 2.9).

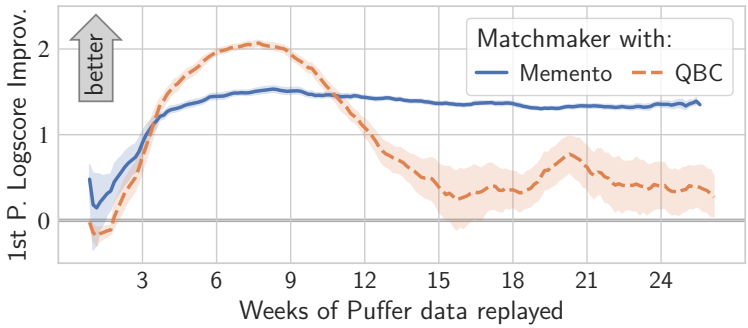


Figure 2.10: Over time, Memento outperforms QBC.

Query-by-Committee A Memento-trained ensemble not only outperforms an ensemble trained with random samples but also an ensemble trained with the Query-by-Committee (QBC) algorithm [26] (Figure 2.10). QBC selects samples with the highest prediction entropy between all members of the ensemble. In theory, these samples contain the most information for learning. We repeat the MatchMaker-Oracle experiment, again using an ensemble of 7 models, and compare an ensemble of Memento-trained models with an ensemble of QBC-trained models.

Figure 2.10 shows that the tail improvements of QBC initially exceed Memento, and are comparable to Memento with JTT (Figure 2.9). However, the QBC ensemble degrades over time and ultimately settles on less than a third of Memento's improvement. We suspect this comes from QBC accumulating noise. Both Memento and QBC initially pick up noise (low density; high entropy). Memento's probabilistic approach prevents noise from accumulating (Section 2.3.4) while QBC degrades: selecting noise biases the model toward random predictions, increasing the prediction entropy of any sample, thus decreasing QBC's ability to identify informative samples.

2.5 EVALUATION: SYNTHETIC SHIFTS

In the previous section, we showed that Memento provides significant benefits in a real-world use case. In this section, we use ns-3 [39] simulations of data center workloads to show that sample selection with Memento applies to other settings as well. Specifically, we show that Memento:

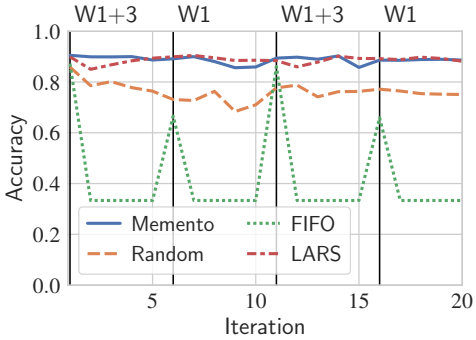
1. ensures good tail performance by reliably prioritizing samples from infrequent traffic patterns (Section 2.5.2);
2. picks up new patterns quickly (Section 2.5.3);
3. is applicable to classification and regression (Section 2.5.4).

2.5.1 *Experimental setup*

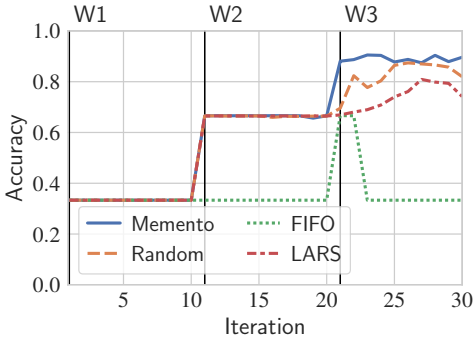
Sample selection strategies We compare Memento with two baselines: Random (random sampling) and FIFO (keep recent samples). In addition, we compare it to the state-of-the-art LARS (Loss-Aware Reservoir Sampling, [21]). LARS improves random sampling for classification, and has two stages: first, it randomly chooses to keep or discard a new sample, with probability exponentially decreasing over time; second, it considers both label counts and loss to decide which in-memory sample to replace.

Parameters For Memento, we use the same default parameters as before: a batching size of 256, kernel bandwidth h of 0.1, and temperature T of 0.01. We reduce the memory capacity C to 20 k samples (i.e., 1/50 compared to Section 2.4) for two reasons: *(i)* We aim to show the limitations of different sample selection strategies, which is easier with small memories; *(ii)* LARS scales poorly, making comparing performance on larger memory sizes impractical—we had to optimize it to scale to even 20 k samples.

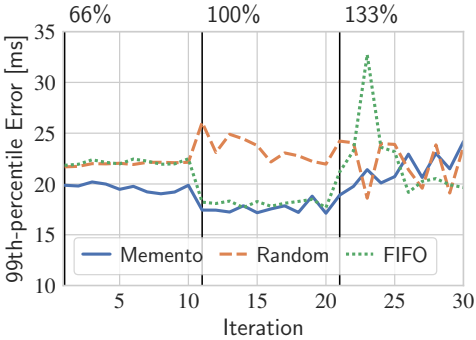
Workloads The simulation setup (Appendix A.2, Figure A.14) consists of two nodes and applications sending messages with sizes drawn from three empirical traffic distributions from the Homa project (Figure A.15, [40]): Facebook web server (**W1**), DC-TCP (**W2**), and Facebook Hadoop (**W3**). W1 and W3 are similar, while W2 messages are about an order of magnitude smaller. For each workload, we generate 20 traffic traces of 1 min each. During this time several senders transmit a combined 20 Mbps, resulting in an average network utilization of 66 %. We repeat this process by injecting additional cross traffic to reach an average utilization of 100 % and 133 %, respectively. We use different random initializations to generate a total of 180 distinct runs that we use in the following experiments.



(a) Rare and infrequent traffic patterns: Only LARS and Memento avoid forgetting. Traffic for W2 is always present; for W1 and W3 only where marked.



(b) Incremental learning: Memento efficiently integrates workload traffic patterns as they appear sequentially where marked, replacing the previous ones.



(c) Increasing congestion: Memento has comparable or lower 99th percentile error for the marked level of network utilization,i.e., increasing congestion.

Figure 2.11: Memento can handle various traffic patterns and prediction types.

Models We compare the selection strategies for two neural networks; one for classification and one for regression. The classification model predicts the application workload: Each input is a trace of the past 128 application packet sizes, and the model predicts the probabilities for each workload. The regression model predicts the next transmission time from past packets: Each input contains a trace of the past 127 packet sizes and transmission times, the current packet size, and the model predicts the transmission time. See Appendix A.2 for architecture and hyperparameter details.

Metrics For classification tasks, we measure the balanced accuracy, i.e., the accuracy obtained over an equal number of evaluation samples per workload, ensuring equal importance of each workload. In other words, performance is evaluated over an equal distribution of overall workloads, regardless of whether they are present at the current iteration. Good performance requires both picking up new patterns quickly and avoiding catastrophic forgetting by retaining previous patterns.

For regression, we investigate changes in traffic distribution and measure the 99th percentile absolute prediction error over the data in the latest iteration. Good performance requires picking up new patterns quickly.

2.5.2 *Classification: Rare patterns*

In the first experiment, we show that Memento successfully picks up samples from infrequent traffic patterns. To do so, we use highly imbalanced traffic: W1 and W3 only constitute $\leq$2 % of overall traffic. Good tail performance implies high accuracy not only for W2 but also for W1 & W3.

Setup We use the classification model and iterate over samples from 20 runs. We use W2 at every iteration, representing a large part of traffic that remains relatively unchanged. On top of that, we include W1 once every five iterations, and W3 once every ten iterations; they represent sporadic traffic patterns that make up for 1.3 and 0.5 % of traffic respectively.

Results Memento and LARS retain sufficient samples from each workload and show the best accuracy over all iterations (Figure 2.11a). On the other hand, FIFO shows good accuracy only while all workloads are present, as the large number of samples of W2 quickly overwrites W1 & W3 otherwise. While Random achieves better results than FIFO, it ultimately retains too few samples of W1 & W3, as they make of less than 2 % of samples in memory.

2.5.3 *Classification: Incremental learning*

Next, we show that Memento picks up new patterns and avoids catastrophic forgetting in 'Incremental Learning': a setting known to be challenging [41].

Setup As in Section 2.5.2, but iterate over samples from each workload sequentially; first W1, then W2, and finally W3, for 10 iterations each.

Results We find that overall, Memento exhibits the best performance. Both Random and LARS struggle because of their sample selection rate (Figure 2.11b); these two flavors of random memory avoid forgetting by decreasing the probability of selecting new samples over time. When W3 is introduced, they are slow to incorporate new samples (Appendix A.2, Table A.2). While both LARS and Random are slow to react, the fact that the balanced accuracy of LARS is worse than Random's is mostly an artifact of the similarity of W1 and W3: LARS has (desirably) retained more samples of W1, yet this causes its model to mistake W3 for W1 more often than Random, which has forgotten most of W1 and is consequently less biased. By manually tuning the sampling rate of LARS to be much more aggressive, we were able to achieve the same performance as Memento (not shown). This highlights the benefit of the self-adapting nature of Memento's sample-space-aware approach: If a new label appears, Memento discovers that this part of the sample space is not well covered yet. It quickly prioritizes discarding common in-memory samples over rare new ones.

2.5.4 *Regression*

In this experiment, we show that Memento is applicable to regression and handles complex traffic changes. We iterate from 66 to 133 % network utilization, which presents more complex gradual changes in traffic patterns than the abrupt changes in workload distributions (Sections 2.5.2 and 2.5.3).

Setup We iterate over traffic from all workloads using runs with increasing congestion. For the first 10 iterations, we use runs with 66 % network utilization, followed by 10 iterations of 100 %, and finally 10 iterations of 133 %. We report the 99th percentile error for the current utilization.

Results Memento generally shows the lowest 99th percentile prediction error (Figure 2.11c). Random is slow to react to the new patterns and requires several iterations to adjust to the new traffic conditions. Perhaps surprisingly, FIFO performs well up to 100 % utilization but shows very unstable performance for 133 %. LARS is not applicable to regression.

2.6 RELATED WORK & DISCUSSION

Limitations Let us address the elephant in the room: Memento cannot do anything with a bad model or dataset. It helps identify the most useful samples for training, but those samples must be present in the dataset in the first place, and the model must be capable of learning from them.

We find that density-based selection performs well but is probably not optimal. It addresses the problem of dataset imbalance well, but it is less effective to differentiate "hard-to-learn" from "easy-to-learn" samples, which is better captured by the model loss. Finding a way to effectively combining both metrics would likely be beneficial. Finally, density computations limit Memento's scalability. Batching helps (Figure 2.5), but it would not be enough to process data streams with billions of samples per day.

Continual learning Continual learning and dataset imbalance correction are well-studied problems. Fundamentally, continual learning suffers from the *stability-plasticity* dilemma [42]: a stable memory consolidates existing information yet fails to adapt to changes, while a plastic memory readily integrates new information at the cost of forgetting old information. Forgetting old-yet-still-useful information is known as *catastrophic forgetting* [43]. Continual learning approaches aim to be as plastic as possible while minimizing catastrophic forgetting. They can be broadly categorized as either prior-based or rehearsal-based [21]. Prior-based methods aim to prevent catastrophic forgetting by protecting model parameters from later updates [20, 44, 45]. Rehearsal-based methods collect samples over time in a replay memory and aim to prevent forgetting by learning from both new and replayed old data [21, 22, 46]. Hybrid methods combine both, e.g., training with a replay memory and a loss term penalizing performance degradation on old samples to protect model parameters [23].

Memento builds on previous rehearsal-based approaches, incorporating ideas such as coverage maximization [47]. It extends existing ideas by considering both prediction and output spaces, leveraging temperature scaling to control the tail-focus and introducing a novel *coverage increase* criterion to reason about when to retrain. Similar to Memento, Arzani et al. [48] also suggest using the "right samples" to retrain AutoML systems [49]: They use the disagreement among a set of models to identify "the tail" and guide the user to collect more tail samples and add them to the training set. However, this does not apply to all networking applications; e.g., we cannot "force" streaming sessions to come from rare network paths or experience particular congestion patterns; we must do with the available samples.

Distribution shift detection Continual learning closely relates to a branch of research aiming to keep models up-to-date upon changes in the data-generating process—known as distribution shifts. State-of-the-art methods rely on statistical hypothesis testing [50] or changes in empirical loss [51], plus a time-based window (or multiple parallel windows) [42]. When change is detected, these algorithms advance the window(s), discard outdated samples whose timestamp falls outside of the window(s), and retrain.

Shift detection algorithms make *sample-space-aware* retraining decisions but lack a comparable selection strategy. Once a change is detected, they discard all old samples. This approach is too coarse-grained for networking: Network traffic is composed of many patterns (Appendix A.1, Figure A.1) and not all patterns get outdated at once and do not all need to be discarded.

Matchmaker [37] proposes a more incremental approach to shifts in networking data. It uses an ensemble of models and "matches" each sample to the model trained with the most similar data. However, it does not address the sample selection problem: If no model in the ensemble is good at the tail, matching will not help. By contrast, Memento builds upon ideas originating from shift detection (BBDR [29]) and extends the sample-space-aware strategy to the sample selection to train better models. Our evaluation shows that a single Memento-trained can outperform Matchmaker at the tail, even if using an oracle to match the optimal model (Section 2.4.4).

Experience replay for reinforcement learning While we evaluate Memento in the context of supervised learning, it may also be used for *reinforcement learning* (RL), a popular approach to ML-based ABR [52, 53]. In fact, RL commonly uses a replay memory [54, 55], for better performance [23, 56].

Transformers and other large models In recent years, large models with billions of parameters have become popular in natural language processing [57–59] and computer vision [60, 61], usually based on Transformer architectures [62]; there are also first trials in networking [63–65].

We have evaluated Memento on small models to investigate a *smarter* sample selection instead of using more resources such as more complex models and cannot confidently claim that we would see comparable improvements. Yet, there is mounting evidence that even large transformer models suffer from data bias [66–68], which Memento may help to address.

Furthermore, an important part of these models is an encoder that translates text, images, or other data into a latent space. Latent space representations have seen use in clustering [69], and Memento may benefit from computing sample distances in latent space, similar to how it uses BBDR.

Data processing Data validation and augmentation are important steps of any ML pipeline. Especially in ML systems that are evolving over time, new bugs may be introduced any time the model or data collecting system are updated. These bugs may lead to erroneous data, and data validation is necessary to prevent such data from becoming part of the training data [70]. Furthermore, collected data is often augmented to address dataset imbalance by generating additional synthetic samples or upscaling existing ones, which can improve performance and reduce overfitting [71]. One example is Just Train Twice (JTT) [38], which trains a model, uses it to identify misclassified samples, upsamples those, and trains again.

As a replay memory, Memento operates between the validation and augmentation steps, and complements them. For example, we showed in Section 2.4.4 that JTT's upsampling is more effective with tail samples identified by Memento. That being said, when combining methods, only validated data should be considered by the sample selection, and selected samples may be augmented. In particular, the memory should only store non-augmented samples, as augmenting the data before passing it to the memory can result in the sample selection to overfit to the augmentation [21].

Generalization While this dissertation focuses on video streaming, Memento could be applied to other ML-based networking applications, including congestion control [72–75], traffic optimization [76], routing [77], flow size prediction [78, 79], MAC protocol optimization [80, 81], traffic classification [7, 82], network simulation [83], or DDoS detection [82]. Networking has proven to be a challenging environment for ML, and many proposed systems have only delivered modest or inconsistent improvements in real networks [16, 84–86]. In response, research has focused on providing better model architectures [16, 72, 73] and training algorithms [52], model ensembles for predictions and active learning [37, 49], real-world evaluation platforms [16, 86], uncertainty estimation [87] and model verification [88].

A better sample selection is beneficial to all these advances. Memento is orthogonal to and complements these works, opening an exciting potential. We show in Section 2.4.4 that combining Memento with JTT or Matchmaker improves performance further: once Memento decides to retrain, we train better with JTT, and MatchMaker benefits from an ensemble of Memento-trained models. However, optimizing training or model architectures is beyond the scope of this work, which focuses on *identifying the most valuable samples* for retraining and deciding *when to retrain.*

3

LEARNING OVER SPACE: A VISION FOR NETWORK MODEL GENERALIZATION

In Chapter 2, we evaluated how ML-based ABR algorithms can *adapt over time* by using a smarter sample selection. We constantly updated the training data, prioritizing rare patterns for tail performance. Fundamentally, this approach assumes that even rare patterns are worth remembering.

But what if this assumption does not hold, as in the following two examples: *(i)* what if we face a completely new environment, e.g., a different network topology with an entirely different tail? or *(ii)* what if we face a new prediction task that makes existing samples obsolete, e.g., predicting network path delay instead of video chunk download times?

This chapter explores these questions to promote a new vision for *learning over space*, i.e., generalizing models to new environments and prediction tasks. Until now, generalizing ML models for network traffic dynamics tended to be considered a lost cause. For every new task, we design new models and train them from scratch on specifically collected datasets.

At the same time, an ML architecture called *Transformer* has enabled previously unimaginable generalization in natural language processing and computer vision. Nowadays, models pre-trained on massive datasets are fine-tuned to other tasks and environments with comparatively little time and data, delivering state-of-the-art performance for many benchmarks.

We believe this progress could translate to networking and propose a Network Traffic Transformer (NTT), a transformer adapted to learn network dynamics from packet traces. Our initial results are promising: NTT seems able to generalize to new prediction tasks and environments.

This chapter is organized as follows: Section 3.1 motivates the problems and opportunities of model generalization in networking.
Section 3.2 provides a brief background on transformers.
Section 3.3 presents NTT: our proof-of-concept *Network Traffic Transformer*.
Section 3.4 provides first evidence from simulations that NTT can learn network traffic dynamics and generalize to new tasks and environments.
Finally, Section 3.5 outlines open questions and future research directions to realize the vision of network model generalization.

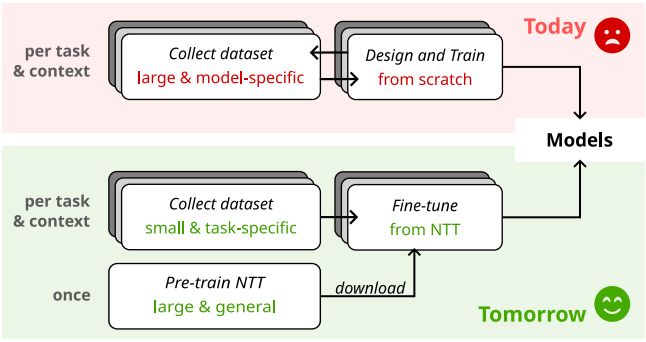


Figure 3.1: Could we collectively learn general network traffic dynamics *once* and focus on task-specific data collecting and learning for *all future models?*

3.1 TOWARDS MODEL GENERALIZATION

Problem Today's network traffic models do not generalize well; i.e., they often fail to deliver outside of their original training environments [16, 84–86, 89]; generalizing to different tasks is not even considered. The Puffer paper argues that, rather than hoping for generalization, one obtains better results by training in-situ, i.e., using data collected in the deployment environment [16]. Thus, today we tend to design and train models from scratch using model-specific datasets (Figure 3.1, top). This process is repetitive, expensive, and time-consuming and hinders collective progress.

Vision We argue there is still hope for generalization in networking. Even if the networking contexts (topology, network configuration, traffic, etc.) are very diverse, the underlying dynamics remain similar; e.g., when buffers fill up, queuing disciplines delay or drop packets. These dynamics can be learned with ML, and there is no need to relearn everything every time. We envision a *generic network model* trained to capture the shared dynamics underpinning any network. Such a model could be fine-tuned for many different networking tasks and contexts with only a small task-specific dataset for fine-tuning (Figure 3.1, bottom).

Existing approaches—while not performing *optimally* outside of their training environments—provide evidence for generalization. The congestion control algorithm Aurora [73] performs adequately in environments with bandwidth more than an order of magnitude higher than during training. Models trained with Genet [90] on simulation data perform well in several real-world settings. But truly "generic" models—able to perform well on a wide range of tasks and networks—remain unavailable, as mixing different

contexts is unpredictable. In some cases, more diverse training data has been shown to provide benefits without consequences, e.g., training over a range of propagation delays in [91]. Yet in other cases, mixing contexts can decrease performance,e.g., varying numbers of senders in [91] or wired and wireless traces in [85], if the model is not able to tell these contexts apart.

Game-changer A few years ago, a new architecture for sequence modeling was proposed: the *Transformer* [62]. This architecture is designed to train efficiently,[1] enabling learning from massive datasets and unprecedented generalization across *multiple* contexts. In a pre-training phase, the transformer learns contextual sequential "structures," e.g., the structure of a language from a large corpus of texts. Then, in a much quicker fine-tuning phase, the final stages of the model are adapted to a specific prediction task. Today, transformers are among the state-of-the-art in natural language processing (NLP [92]) and computer vision (CV [60, 61]).

Transformers generalize well because they can learn to distinguish different contexts during pre-training; they learn rich contextual representations [57] where the representation of the same element, e.g., a word, depends on its context, inferred from the sequence. Consider two input sequences: *Stick to it!* and *Can you hand me this stick?* The transformer output for each *stick* is different as it encodes the word's context. This contextual output is an efficient starting point for fine-tuning the model to diverse downstream tasks, e.g., question answering, text comprehension, or sentence completion [92]. We can draw parallels between networking and NLP: individual packet metadata (headers, delay, etc.) provides limited insights; we also need the sequence context, i.e., the history of past packets. For example, increasing latency over multiple packets indicates congestion.

Challenges Naively transposing NLP or CV transformers to networking fails, unsurprisingly. We must adapt them to the peculiarity of networks. In particular, "sequences" must be carefully defined: While text snippets and images are relatively self-contained, any packet trace only gives a partial view of the network. Moreover, generalizing the diversity of protocol interactions is not trivial. Ultimately, we identify three main open questions.

1. How to adapt transformers for networking?
2. Which pre-training task would allow the model to generalize, and how far can we push generalization?
3. How to assemble a dataset diverse enough to allow generalization?

1 Transformers scale better than recurrent neural networks, another popular architecture for sequence modeling that Transformers effectively succeeded.

3.2 BACKGROUND ON TRANSFORMERS

In this section, we introduce *attention*, the mechanism behind Transformers; detail the idea of pre-training and fine-tuning; and present insights from adapting Transformers from NLP to CV.

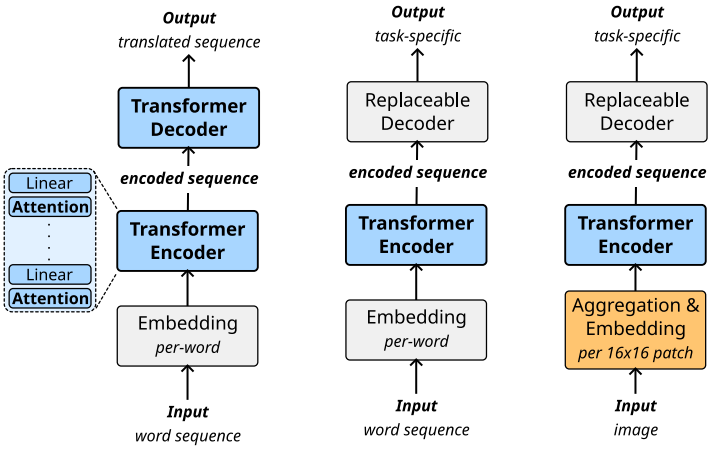
Sequence modeling with attention Transformers are built around the *attention* mechanism, which maps an input sequence to an output sequence of the same length. Every output encodes its own information *and its context*, i.e., information inferred from related elements in the sequence. For a detailed explanation, we refer to [62] and online guides [93, 94].

While attention originated as an improvement to recurrent neural networks (RNNs), Vaswani et al. [62] realized that it could replace them entirely. They propose the following architecture for translation tasks: an *embedding* layer maps words to vectors; a *transformer encoder* encodes the input sequence; and a *transformer decoder* generates an output sequence based on the encoded input (Figure 3.2a). Transformer blocks alternate between attention and linear layers, i.e., between encoding context and refining features.

Pre-training and fine-tuning Transformers are used for a wide range of NLP tasks, and the prevailing strategy is to use pre-training and fine-tuning. We explain this approach on the example of BERT [57], one of the most widely used transformer models. BERT uses only the transformer encoder, followed by a small and replaceable decoder.[2] BERT is *pre-trained* with a task that requires learning language structure. Concretely, a fraction of words in the input sequence is masked out, and the decoder is tasked to predict the original words from the encoded input sequence (Figure 3.2b). Conceptually, this is only possible if the encoding includes sufficient *context* to infer the missing word. Afterward, the unique pre-trained model can be fine-tuned to many different tasks by replacing the small decoder with task-specific ones, e.g., language understanding, question answering, or text generation [57, 95, 96]. The model has already learned to encode language context and only needs to learn to extract the task-relevant information from this context. This requires far less data compared to starting from scratch: BERT is pre-trained from text corpora with several billion words and fine-tuned with $\sim$100 thousand examples per task. Furthermore, BERTs pre-training task is unsupervised, i.e., it requires only "cheap" unlabeled data for masking and reconstruction. "Expensive" labeled data, e.g., for text classification, is only needed for fine-tuning.

2 Usually, a multilayer perceptron (MLP) with a few linear layers; this decoder is often called the 'MLP head'.

Vision transformers Following their success in NLP, Transformers gained traction in CV as well, with two notable distinctions: *(i)* input aggregation; and *(ii)* a domain-specific pre-training task. While attention is efficient to parallelize, it needs to compare each element in the sequence with each other element to encode context. Consequently, the required computation scales quadratically with the input sequence length, and using sequences of individual pixels does not scale to images of high resolution. As a solution, the Vision Transformer (ViT, [97, 98]) aggregates pixels into 16×16 patches and applies the embedding and transformer layers to the resulting sequence of patches, using an architecture similar to BERT (Figure 3.2c). However, using a classification task to pre-train ViT delivered better results than a reconstruction task. This shows the importance of domain-appropriate pre-training: it may be possible to reconstruct a patch by only considering neighboring ones, but classification requires understanding the whole image, i.e., the context of the entire sequence.



(a) Transformer **(b) BERT** **(c) ViT**

Figure 3.2: Transformer variants.

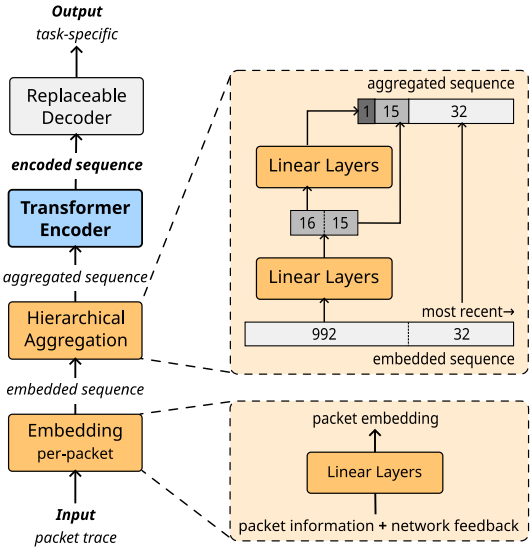


Figure 3.3: NTT combines embedding, aggregation, and a transformer encoder. It outputs a context-rich encoded sequence that is fed into a task-specific decoder.

3.3 A NETWORK TRAFFIC TRANSFORMER

Given the success of Transformers in NLP and CV and the similarities between the underlying sequence modeling problems, we argue that transformers could also generalize network traffic dynamics. This section presents our proof-of-concept: the Network Traffic Transformer (NTT, Figure 3.3).

1. Packets are more complex than words or pixels. Which packet features are helpful and which are necessary to learn network dynamics?
2. The fate of a packet may depend on much older ones. As the sequence length is practically limited (Section 3.2), how can we capture both short- and long-term network dynamics in the input sequence?
3. Which pre-training task enables the model to learn general network patterns effectively?

Learning feature extraction Packets carry a lot of information that could be used as model features, e.g., header fields. Today, we typically use domain knowledge to manually extract and aggregate features and feed these into off-the-shelf ML architectures. We argue this is sub-optimal for two reasons: *(i)* we select features for a specific task, which limits generalization; *(ii)* the features may not even be the 'right ones' for the task.

Instead, we propose to let the model learn useful features from raw data. To learn traffic dynamics from a sequence of packets, we must provide the model with information about the packets as well as their fate in the network. Since we do not want to define a priori how important the individual pieces of information are, we feed them all into a first embedding layer (Figure 3.3). It is applied to every packet separately.

In our proof-of-concept, we use minimal information: *timestamp*, *packet size*, *receiver ID*,[3] and *end-to-end delay*. These enable learning temporal (delays over time) and spatial (impact of packet size on delay) patterns. We discuss the challenge of embedding more information in Section 3.5.

Learning packet aggregation Packet sequences must be long enough to capture more than short-term dynamics. But Transformer training time scales quadratically with the sequence length, posing practical limitations.

We address this with a hierarchical aggregation layer (Figure 3.3). We aggregate a long packet sequence into a shorter one while letting the model learn how to aggregate the relevant historical information, similar to the pixel patch aggregation in ViT [97]. However, we aim to both aggregate *and* retain recent packet-level details. To achieve this, we keep the most recent packets without aggregation and the longer traffic is in the past, the more we aggregate, as details become less relevant to predict current dynamics.

In our proof-of-concept, we set the input sequence length to 1024 packets, enough to cover the number of in-flight packets in our experiments. We aggregate this sequence into 48 elements in two stages (Figure 3.3). We show in Section 3.4 that is beneficial, but it is unclear which sequence length and levels of aggregation generalize best; we discuss this further in Section 3.5.

Learning network patterns Finally, we need a training task that allows NTT to learn network dynamics: in our proof-of-concept, we use end-to-end delay prediction. We aim to pre-train NTT to generalize to a large set of fine-tuning tasks. Consequently, we need a pre-training task that is generic enough to be affected by many network effects. As almost everything in a network affects packet delays (e.g., path length, buffer sizes), a delay prediction task seems a rational choice to achieve this goal.

To pre-train NTT, we mask the delay of the most recent packet in the sequence and use a decoder with linear layers to predict the actual delay. During training, the NTT must learn which features are useful (embedding layer), how to aggregate them over time (aggregation layer), and to infer context from the whole sequence (transformer encoder layers).

3 An IP addresses proxy, as we do not want to learn IP address parsing (yet).

During fine-tuning, one can update or replace the decoder (Figure 3.3) to adapt NTT to a new environment (e.g., same decoder in a different network) or to new tasks (e.g., predicting message completion times). This is efficient as the knowledge accumulated by NTT during pre-training generalizes well to the new task, as we demonstrate in the next section.

3.4 PRELIMINARY EVALUATION

Our preliminary evaluation of NTT in simulation shows that:

1. NTT is able to learn some network dynamics;
2. Pre-training helps to generalize;
3. Networking-specific design helps generalization.

Importantly, we do *not* aim to show that NTT outperforms existing specialized models (yet[4]). We focus on assessing the potential of our approach.

Datasets We use ns-3 [39] to generate datasets: one for pre-training, and several for fine-tuning (Figure 3.4). We reserve a fraction of each for testing.

In the pre-training dataset, 60 senders generate 1Mbps of messages each, following real-world traffic distributions [40]. They send messages over a bottleneck link with 30Mbps bandwidth and a queue size of 1000 packets. We run 10 simulations for 1 minute each with randomized application start times. This dataset contains about 1.2 million packets. For the fine-tuning datasets, we add cross-traffic (case 1) and additionally extend the network topology (case 2). Cross-traffic is modeled as 20Mbps of TCP flows. Note that the datasets do *not* contain the cross-traffic packets. For each case, we generate a dataset containing roughly as many packets as the pre-training dataset and a "smaller" dataset containing about 10% of the packets.

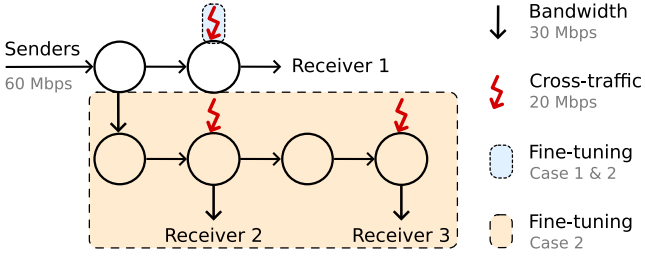


Figure 3.4: Dataset generation setup.

4 The research on Transformers in CV showed that large datasets are required for transformers to outperform the state-of-the-art.

Models We compare several versions of NTT. The *pre-trained* models first learn from the pre-training and then one fine-tuning dataset, while the *from scratch* versions only learn from one fine-tuning dataset. We also pre-train ablated versions of NTT: we compare our multi-timescale aggregation of 1024 packets into 48 aggregates (see Section 3.3) with *no aggregation* (using only 48 individual packets) and *fixed aggregation* (using 48 aggregates of 21 packets each, i.e., 1008-packet sequences). In addition, we pre-train one model *without delay* and one *without packet size* information in the input sequences. Finally, we consider two naive baselines: one always returns the *last observed* output value; another returns an *EWMA*.[5]

Tasks We evaluate our models on two prediction tasks. The first is to predict the delay of the most recent packet; this task is also used for pre-training. The second task is to predict the message completion times (MCTs), i.e., the time until the final packet of a message is delivered. This flow-level task uses a decoder with two inputs: the NTT-encoded sequence and the message size. We report the mean-squared error (MSE) for both tasks and process MCTs on a logarithmic scale to limit the impact of outliers.[6]

Case #1 – Generalization on the same topology We first consider the fine-tuning case 1, where we add unseen cross-traffic on the same topology (Figure 3.4, Tabs. 3.1 and 3.2). The *pre-trained* NTT beats all basic baselines (Table 3.1). While this is no breakthrough, it suggests that NTT learns sensible values. Next, we observe that pre-training is beneficial: the *pre-trained* NTT outperforms the *from scratch* version (Table 3.1); it generalizes to a new context (unseen cross-traffic) and a new task (MCT prediction).

5 Exponentially Weighted Moving Average; we used $\alpha = 0.01$.
6 MCT mean: 0.2s ; 99.9th percentile: 23s

all values $\times 10^{-3}$	Pre-training	Fine-tuning (10%)	
	Delay	Delay	log MCT
NTT			
Pre-trained	0.072	0.097	65
From scratch	-	0.313	117
Baselines			
Last observed	0.142	0.121	2189
EWMA	0.259	0.211	1147
NTT (Ablated)			
No aggregation	0.258	0.430	61
Fixed aggregation	0.055	0.134	115
Without packet size	0.001	8.688	94
Without delay	15.797	10.898	802

Table 3.1: Mean Squared Error for all models and tasks. The *pre-trained* NTT outperforms the *from-scratch version* and thanks to our design choices (Section 3.3).

	Layers trained	MSE(Delay) $\times 10^{-3}$	Training time
Pre-trained			
Fine-tuning (full)	Decoder only	0.033	8h45
Fine-tuning (10%)	Decoder only	0.037	3h45
From scratch			
Fine-tuning (full)	Full NTT	0.036	26h
Fine-tuning (10%)	Full NTT	0.118	8h40

Table 3.2: Pre-training saves resources: we achieve better performance with less fine-tuning data and computing power.

	MSE(Delay) $\times 10^{-3}$	Training time
Pre-trained		
Fine-tuning (full)	0.004	10h
Fine-tuning (10%)	0.035	8h
From scratch		
Fine-tuning (full)	5.2	20h
Fine-tuning (10%)	8.2	11h

Table 3.3: Pretraining can be essential: on a larger topology, training *from scratch* fails even with a large dataset.

We also observe the benefits of hierarchical aggregation and the mix of network and traffic information in the raw data (Table 3.1). With *no aggregation*, the model has little history available; we observe that, perhaps surprisingly, this affects the delay predictions but not the MCT ones. Conversely, with a *fixed aggregation*, the model loses packet-level details but has access to a longer history; this seems sufficient to predict delays but not MCT. More generally, this initial result suggests that both recent packet-level information and an aggregated history are useful to generalize to a large set of tasks. Considering the NTT versions *without packet size* and *without delay* information, we observe that neither generalize. Without packet size, the model overfits the pre-training dataset and performs poorly on predicting delay for fine-tuning. Without delay information, the model can logically not produce any sensible prediction for packet delays or MCTs.

Finally, one can argue that the *pre-trained* NTT has an unfair advantage as it trained on about ten times more data than the *from scratch* version. To put things into perspective, Table 3.2 compares the delay MSE and training time for NTT versions fine-tuned on different datasets. We observe that fine-tuning on a *full* dataset from scratch yields about the same performance as the on the *10%* dataset after pre-training.[7] However, fine-tuning on the *full* dataset also requires almost seven times as much training time (26h vs. 3h45). In practice, collecting fine-tuning data is often expensive; it is thus beneficial to require less. Finally, fine-tuning from scratch may just not work in more complex settings, as shown next.

Case #2 – Generalization on a larger topology We now consider the fine-tuning case 2, with several cross-traffic sources on a larger topology (Figure 3.4). In this setting, packets toward different receivers experience different path delays and different levels of congestion from cross-traffic.

As evident from Table 3.3, pre-training is essential for NTT to learn basic congestion dynamics first, then generalize later on to the topology's specifics during fine-tuning.[8] When fine-tuning *from scratch*, even the *full* dataset is not enough to learn; performance is worse than the baselines (MSE of 11.2 and 4.0–not shown). Without addressing information, NTT cannot differentiate between the receivers and thus cannot predict the packet delay accurately (MSE of 2.8–not shown).

7 Tabs. 3.1 and 3.2 were obtained with different "*10%* fine-tuning datasets". The data allows comparison within each table but not across the two tables.

8 The importance of learning increasingly complex tasks is a problem known as curriculum learning [99] and was recently considered in networking [90].

3.5 DISCUSSION & FUTURE RESEARCH

Our initial results are promising: they show that NTT effectively learns, that the pre-training knowledge generalizes to new tasks and contexts and that its specific design benefits overall performance. Nevertheless, it merely validates that NTT *may work* in practice. We discuss key questions below.

Does the premise hold? We showed some potential for pre-training and fine-tuning with small-scale simulations. However, real networks are undeniably more complex than this environment. Real topologies include many paths where many different applications, transport protocols, queuing disciplines, etc.coexist. There are also many more fine-tuning tasks to consider, e.g., flow classification for security or anomaly detection. Testing our NTT prototype in real, diverse environments and with multiple fine-tuning tasks would provide invaluable insights into the strengths and weaknesses of our architecture and the 'learnability' of network dynamics in general. In Chapter 4, we begin to investigate this question by analyzing an NTT-like model on preliminary real-world video streaming data.

Advancing NTT Our prototype architecture [100] needs enhancements to be helpful in more diverse environments. We see three directions for improvement: *(i)* packet headers; *(ii)* network telemetry; and *(iii)* sequence aggregation. Considering packet headers may be essential to learning the behavioral differences of transport protocols or network prioritization of different traffic classes. However, raw headers are challenging inputs for an ML model, as they may appear in many combinations and contain values that are difficult to learn, like IP addresses [83]. Research from the network verification community on header space analysis [101] may provide valuable insights on header representations and potential first steps in this direction. In addition, we may collect telemetry data like packet drops or buffer occupancy. This may help to learn, but not every trace will contain all telemetry, and future research will need to address this potential mismatch. Finally, we base our prototype aggregation levels on the number of in-flight packets, i.e., whether packets in the sequence may share some fate, usually determined by buffer sizes. The further packets are apart, the less likely they do, and the more we aggregate. We believe matching individual aggregation levels to typical buffer sizes (e.g., flow and switch buffers) may be beneficial. Still, future research needs to put this hypothesis to the test and determine the best sequence sizes and aggregation levels across multiple networks.

Message-based NTT Our current NTT prototype is focused exclusively on packet sequences. However, for applications, it may be easier to collect telemetry on sequences of messages like HTTP requests or video chunks instead of the underlying packets. Messages typically contain data that is sent over many packets and may be leveraged as a "natural" way of aggregating sequences. For example, we observe video chunks for a Puffer video stream to average about 200 packets, up to about 1000 packets.[9] Instead of purely focusing on advancing the packet-based NTT, it may also be beneficial to investigate a message-based NTT. However, we must keep in mind that be focusing *only* on messages, we may lose valuable fine-grained information that we may only infer from packet-level timescales. Chapter 4 investigates both a packet- and message-only NTT variants as well as combining packet and message sequences into a hybrid model.

Continual learning A cat remains a cat, but the Internet is an evolving environment. Protocols, applications, etc., change over time. We conjecture that underlying network dynamics change less frequently than specific environments; thus, the same NTT may be used for several updates of the same fine-tuned model. Nevertheless, even a pre-trained model may become outdated. It is already difficult to determine when to re-train a specific model [16]; it might be even more difficult for a model supposed to capture a large range of environments. While we have treated this chapter independently from our results on *learning over time* in Chapter 2, we believe that it would be an interesting future research direction to combine both.

9 The Puffer data itself does not contain packet data, but several sample videos are available in their public repository, which we used to determine this number.

4

LEARNING LATENT NETWORK STATE: A LOOK BEHIND THE CURTAIN OF TRAFFIC MODELING

In the previous chapters, we have investigated learning over time and space. In both cases, we considered specific models: the Fugu transmit time predictor, a standard neural network (Chapter 2), and NTT, a Transformer model for packet traces (Chapter 3). Just as we leveraged a Transformer architecture and pretraining to improve generalization, a plethora of modeling strategies have been proposed for different traffic modeling tasks.

Consider the following video-streaming tasks: *(i)* predicting the future download time (like Fugu); and *(ii) counterfactual reasoning*, which aims to answer questions like *How would past download times have changed with a different ABR?* Solutions for counterfactual reasoning, such as CausalSim [102] and Veritas [103] use different models and training methods than Fugu, and it is impossible to directly apply and compare them in each other's domain. This lack of comparability limits knowledge transfer and generalization.

Here, we see an opportunity: at their core, network traffic models share an underlying problem: from past observations, estimate the latent network state to predict future or counterfactual network responses. By evaluating the different strategies through a common lens of latent state estimation, they become comparable, allowing us to transfer successful ideas, identify strategies that generalize well, and remove unnecessary constraints. For example, CausalSim can only be trained from data with multiple distinct ABRs and cannot be used otherwise. Is this constraint necessary?

We realize this opportunity by formulating the general problem of learning latent network state and identifying key steps needed to solve it. This allows us to analyze how Fugu, NTT, CausalSim and Veritas solve these individual steps, making them comparable even if their final tasks are not.

This chapter is structured as follows. Section 4.1 summarizes the learning strategies of Fugu, NTT, CausalSim, and Veritas. Section 4.2 presents the general problem of learning latent network state, its key components, and how the different strategies address them. Section 4.3 evaluates the different approaches on both the Puffer video streaming dataset and a preliminary packet dataset. Finally, Section 4.4 summarizes our findings, discusses limitations, and presents open research questions.

4.1 OVERVIEW

In this section, we provide an overview of the different prediction tasks and systems we analyze. In particular, we explain how they share a fundamental problem: estimating the latent network state. For the first task, *download time prediction*, we summarize Fugu and NTT (see Chapters 2 and 3 for more details). For the second task, *counterfactual reasoning*, we introduce CausalSim [102] and Veritas [103].

4.1.1 *Download time prediction*

ABR algorithms select the quality of the next video chunk to optimize the user Quality of Experience (QoE), i.e., they select the chunk size maximizing quality while minimizing stalls (see also Chapter 2 for details) This is a control loop that must model the current network traffic to make optimal decisions. Placing more load on the network, e.g., by downloading a larger high-quality video chunk, can improve the user experience—but it may also overload the network, leading to interrupted streams and a worse QoE. ABR algorithms must predict the outcome of their actions, typically the time to download a chunk of a given size. In the following, we discuss ABR algorithms that combine a classical model-driven control loop with an ML model. Alternatively, *reinforcement learning*-based ABR algorithms like Pensieve [53] remove the control loop and predict the next action directly. We focus on the former, as it allows us to analyze the quality of modeling the network independently of the quality of the planning algorithm.

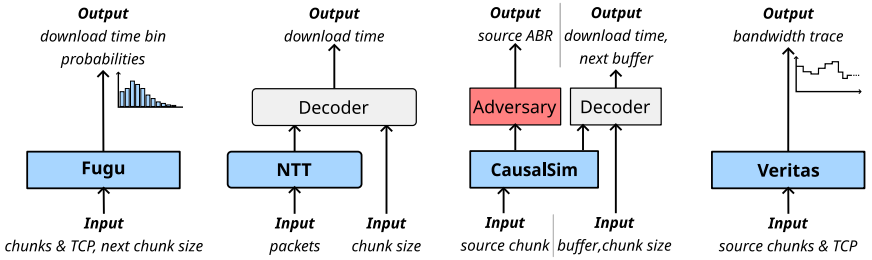


Figure 4.1: Models for download time prediction and counterfactual reasoning.

Buffer-based ABR Not all ABR algorithms are based on ML, nor do all of them explicitly estimate the latent network state. Linear BBA [104] and other *buffer-based* algorithms select the next video chunk based on the current buffer level, selecting higher quality chunks if the buffer is full and vice versa. Still, this *implicitly* models the latent network state, although coarsely: among other options, a low buffer may indicate a network with low inherent capacity, or a network that is under a heavy load. However, this model only describes the current state and has no predictive power. Predictions are made with a heuristic, assigning each chunk size a minimum buffer level that is assumed to be sufficient to download the chunk in time.

Model-predictive ABR More advanced ABR algorithms use model predictive control (MPC) to plan ahead and optimize expected QoE over a future time horizon [105]. By employing a model to predict download times,[1] they can improve over heuristics like BBA if the model is good enough. Yet, if the models do not capture the network state accurately, these algorithms can fail to outperform simpler alternatives like BBA [16].

Fugu improves model-predictive ABR with an ML-based model, predicting a distribution over download times based on the network state estimated from the past 8 video chunks (Figure 4.1a). Fugu uses application and transport protocol inputs: it observes video chunk sizes and download times as well as TCPInfo metrics like the congestion window size.

Instead of Fugu, we could also use NTT to predict the download time. In Chapter 3, we have tested NTT with multiple prediction tasks, including predicting the message completion time. In the context of video streaming where messages are video chunks, this is equivalent to predicting the download time of the next video chunk. However, where Fugu considers application- (video chunks) and transport protocol (TCPInfo) observations, NTT steps closer to the network and considers packet traces (Figure 4.1b).

4.1.2 *Counterfactual Reasoning*

Another important use case for predicting the latent network state is *counterfactual reasoning*. In video streaming, this typically means: given actions and observations from a *source ABR*, how would both have changed if a different *target ABR* had been used instead? The latent network state impacts both actions and observations; in counterfactual reasoning, latent variables that introduce such causal dependencies are called *confounders*.

1 Predicting download times is typically more accurate than predicting throughput, as the relationship between them is nonlinear [16, 103].

Accurate counterfactual reasoning requires accounting for confounders. In fields like medicine, randomized controlled trials (RCTs) are considered the "gold standard" to achieve this. In an RCT, actions (called interventions) are randomly assigned to make them independent of the confounders. This has also been applied in networking. The Puffer project is a large-scale RCT, randomizing the choice of ABR to make the aggregated long-term performance independent of user network conditions. However, this is no silver bullet for counterfactual queries in video streaming: RCTs can remove the impact of confounders on the aggregated performance, but not on individual actions. ABRs make decisions based on the current state and may influence the network state, and thus the future observations, e.g., by causing congestion. Thus, we cannot remove the impact of confounders on individual ABR actions through our experiment setup and must estimate and compensate for them explicitly. In other words, counterfactual reasoning for video streaming requires estimating the latent network state.

Estimating confounders in video streaming The need to model confounders for counterfactual reasoning in video streaming was first highlighted in a case study by Sruthi et al. [106]. Faced with RCT limitations, the study argues for using domain knowledge to *(i)* identify confounders, and *(ii)* build a causal graph relating them to actions and observations.

The study models a single confounder, the available network bandwidth, and models it as a square wave (high and low bandwidth alternating in fixed intervals), and its impact on actions and download times. They generate synthetic data from this model and show that modeling dependencies improves results over ignoring confounders or relying on RCTs.

However, even for this simple case with minimal confounders, the derived causal graph is complex and difficult to scale up to real-world networks. As a result, the case study became the foundation of CausalSim and Veritas, which address the study's limitations in two opposing ways.

CausalSim: removing RCT bias with adversarial learning Instead of hand-crafting a complex causal dependency graph to avoid RCTs, CausalSim [102] learns how to remove biases from RCT data.

The key idea behind CausalSim is to leverage adversarial learning [107]. Adversarial learning uses an encoder-decoder structure with an additional adversarial decoder (Figure 4.1c). The encoder and first decoder solve the prediction task: From past actions and observations, the encoder extracts the latent state, and given the latent state and action, the decoder predicts the original observations. The trained decoder can later use the estimated latent state and an *alternative* action for counterfactual reasoning.

However, training with RCT data may result in a biased latent state. The goal of the second, adversarial decoder is to reveal these biases. It learns to classify the ABR from the latent state, exploiting any ABR-dependent biases. This can be leveraged to improve the encoder by training it to both minimize the prediction loss (the actual task) and to *maximize* classification loss. As predicting the correct ABR requires latent state biases, maximizing the loss of the adversary promotes removing these biases during training.

Veritas: learning a causal dependencies Where CausalSim aims to remove RCT biases, Veritas [103] follows the case study's approach and proposes a causal model suitable to represent real-world video streaming dynamics.

Veritas achieves this with a hidden Markov model (HMM). An HMM combines a latent state with learnable transition- and emission probabilities. The transition probabilities describe how the latent state evolves, and the emission probabilities describe the likelihood of observations conditioned on the latent state. Veritas modifies this HMM to make learning tractable, and we discuss the most important modifications below.

First of all, Veritas uses a partially observable state. It uses only a single truly latent variable, the available network bandwidth, and several observable variables: the chunk size, the buffer level, and TCPInfo metrics.[2] As a result, it can express complex state transitions, yet only needs to learn the transition probabilities for the single latent variable given the observable ones, instead of all combinations. To further simplify learning these probabilities, Veritas discretizes the bandwidth and assumes that it only changes in fixed intervals (by default 0.5 Mbps bins and 5 seconds intervals).

Furthermore, Veritas replaces learnable emission probabilities with a hand-crafted emission function that translates the state into the observed throughput. This further reduces the complexity of fitting the HMM to the data, completely removing the need to learn emission probabilities.

Veritas uses this modified HMM for counterfactual reasoning as follows: Given a video streaming trace, it samples one (or several) traces of latent states, i.e., available bandwidths during the stream (Figure 4.1d). These sequences are then used as input to a network simulator, which can now be used for counterfactual reasoning. Compared to using the potentially biased observed bandwidth, Veritas argues that the latent available bandwidth allows for more accurate counterfactual reasoning. Additionally, sampling multiple traces can provide a distribution over counterfactual outcomes, allowing for both more conservative and more optimistic estimates.

2 Veritas is trained on Puffer and uses the same statistics as Fugu.

4.2 LEARNING NETWORK DYNAMICS

In this section, we formalize the general network modeling problem as a nonlinear dynamical system. First, we demonstrate how download time prediction and counterfactual reasoning can be represented in this framework. Next, we then discuss four key components for learning this system:

1. the choice of model architecture,
2. the representation of latent state space,
3. the history, i.e., observations used to estimate the latent state,
4. the training objectives and data.

Our goal is to understand how to best implement these components to learn the latent network state. Thus, we analyze how the overall strategies of Fugu, NTT, CausalSim, and Veritas approach these individual components and summarize their differences as open questions to guide our evaluation in Section 4.3. Table 4.1 provides a summary at the end of this section.

4.2.1 *Dynamic equations*

We model the network traffic dynamics as a nonlinear dynamical system:

$$\frac{dx}{dt}(t) = f(t,\ x(t),\ u(t)) \tag{4.1}$$

$$y(t) = g(t,\ x(t),\ u(t)) \tag{4.2}$$

where $x(t)$ and $\frac{dx}{dt}(t)$ are the *system state* and its evolution; $u(t)$ are inputs; and $y(t)$ are *observable variables*. The function f determines the evolution of the system state, while the function g captures the observability of the system. Typically, g does not allow observing the full state $x(t)$, and the unobserved part is called the *latent state*. This latent state captures anything that impacts observations, from the network topology to congestion control algorithms used by other applications. We concretize this in Section 4.2.3.

Prediction and counterfactual reasoning We can represent both prediction and counterfactual reasoning in the framework of Eqs. (4.1) and (4.2).

For download time prediction, we are given a chunk size $u(t^*)$ at time t^* and a trace of past observations $\mathcal{T}$, i.e., a set of $(t',\ u(t'),\ y(t'))$ where $t' < t^*$. We aim to predict the download time $y(t^*)$. This implies estimating the latent state $\tilde{x}(t)$ such that Eqs. (4.1) and (4.2) hold for all observations in $\mathcal{T}$ with a minimal error. Note that we can typically not perfectly estimate $x(t)$ and $y(t)$, as the system is noisy and not fully observable.

For counterfactual reasoning, we are again given a trace $\mathcal{T}$ of past observations and inputs $(t_{src},\ u_{src}(t_{src}),\ y(t_{src}))$ collected from a source ABR, and we are asked to predict the outcome of a different input sequence $u_{tgt}(t)$ produced by a target ABR. As before, this implies estimating the latent state $\tilde{x}(t)$, such that we can predict the alternative state evolution given $u_{tgt}(t)$ and predict the counterfactual observations $y_{tgt}(t)$.

Learning strategies It is not necessary for a system to explicitly go through the steps described above. For example, we may train a model to predict download times without explicitly returning the estimated latent state $\tilde{x}(t)$. Implicitly, the model must still estimate the latent state to correctly estimate the network behavior and predict accurately. In the following, we discuss the key steps to learning a system for download time prediction and/or counterfactual reasoning under the framework of Eqs. (4.1) and (4.2).

4.2.2 *Model architecture*

The first and most fundamental decision is the choice of model architecture, that is, how we choose to represent the abstract approach presented in Section 4.2.1 as an actual model that we can train and evaluate.

Like Veritas, we can attempt to design such a model by hand. Alternatively, we can use a black-box model, as done by Fugu, NTT, and CausalSim. A hand-crafted model can be easier to interpret and analyze but may suffer from underspecification, not capturing all important network dynamics. Black-box models can learn more, and often more complex dynamics, but may be harder to interpret and analyze. Additionally, they require the training data to be rich enough to contain all important dynamics; if this is not the case, a black-box model may provide no advantage.

However, even a black-box model is not free of design choices. One important difference is whether we predict the latent state $\tilde{x}(t)$ explicitly or implicitly. Fugu uses a single model, mapping all available inputs to the required predictions directly. Concretely, this means that the inputs include both past observations $\mathcal{T}$ as well as the next input $u(t^*)$, and the model returns the prediction $y(t^*)$. The model does not expose the latent state, it is *implicit* in the model. This results in a simple but task-specific model architecture: while the model has learned to estimate the latent state, we cannot easily transfer this knowledge to other tasks.

Alternatively, NTT and CausalSim use an encoder-decoder architecture:

$$\tilde{x}(t) = \text{Encoder}(\mathcal{T}) \tag{4.3}$$

$$y(t^*) = \text{Decoder}_{predict}(t^*,\ \tilde{x}(t^*),\ u(t^*)) \tag{4.4}$$

$$y_{tgt}(t) = \text{Decoder}_{counterfactual}(t,\ \tilde{x}(t),\ u_{tgt}(t)) \tag{4.5}$$

In this architecture, a general encoder is trained to estimate the latent state which is passed to task-specific decoders. The latent state can be reused for different tasks by exchanging the (typically small) decoder, as we've already shown with NTT in Chapter 3. This architecture also enables designs like CausalSim, where a single encoder is combined with multiple decoders.

We can summarize this design space with two key questions:

> In which contexts are hand-crafted models worth the effort?
> What are the trade-offs between explicit and implicit latent state?

4.2.3 *Latent space representation*

For hand-crafted models or models with an explicit latent state, we must decide whether to represent this state with a low or high-dimensional space.

For NTT, we have argued for learning the "best features" from traffic, i.e., *learning* a high-dimensional latent representation (see Chapter 3).

Both CausalSim and Veritas use a one-dimensional latent space. Veritas models the latent state as the available capacity of the network. The authors of CausalSim similarly interpret their latent state as the available capacity and implicitly assume that a higher-dimensional latent space is not necessary to capture the network dynamics, not discussing it further.

We argue that a single latent variable is insufficient. For example, what happens if we exceed the available capacity? The network may carry predominantly elastic traffic like TCP; upon exceeding the capacity, this traffic backs off, making additional capacity available. Alternatively, inelastic traffic like UDP continues to send at the same rate. An ABR algorithm may deliberately choose to exceed the capacity if the buffer level is high and additional download time is acceptable; with a single latent variable, we cannot fully capture the network response. It is easy to imagine such scenarios, yet difficult to determine whether they are *relevant*. In other words:

> Is there a benefit from a high-dimensional latent space?

4.2.4 *History*

We must estimate the latent state from past observations $\mathcal{T}$. A richer history can help to estimate the latent state in several ways. First of all, observations are noisy, and a longer trace can help to find a more robust estimate. Furthermore, the history may also reveal the trajectory of the latent state, which may be impossible to estimate from a single observation. The different systems we analyze treat that history very differently.

NTT considers a trace of individual packets, i.e., the finest granularity of observations. For each packet, it includes minimal information (timestamp, size, and RTT). Fugu, Veritas, and CausalSim are all designed for the Puffer data, which does not include packet-level information. Thus, they include metrics at a coarser granularity. All of them include the size and download time of video chunks. Fugu und Veritas include additional transport protocol statistics by considering TCPInfo metrics, which include the current transport protocol state (congestion window size) and traffic aggregates (RTT and delivery rate estimates). CausalSim and Veritas also include an application metric: the current buffer level. To be precise, CausalSim does not provide the buffer level as an input to the encoder, but as an input to the decoder. At the same time, the current playback buffer is a result of past decisions and network conditions, and thus implicitly provides additional history to the decoder, albeit in an aggregated form.

The systems also differ in the amount of history they consider: NTT and Fugu include a fixed window of 1024 packets and 8 video chunks, respectively. In theory, Veritas considers the *whole trace* of past observations. In practice, however, its performance depends on the sequence length, and the published Veritas model is limited to traces of at most 200 video chunks. CausalSim considers only the current video chunk and buffer size.

We have seen metrics from fine granularity (packets) to coarse granularity (video chunks, which can encompass hundreds of packets) as well as the use of additional metrics provided by underlying protocols (TCPInfo) or the application (buffer). In general, the more metrics we include, the more information we have to estimate the latent state. Yet having a richer history also comes with the overhead of collecting, storing, and processing the additional information. This trade-off leads to the following questions:

Which metrics, in particuar at which granularity, are most helpful?
Is a longer history always better, or are there diminishing returns?

4.2.5 *Training*

Hand-crafted models Hand-crafted models often require specialized training algorithms. Veritas, for example, implements a modified HMM training algorithm to make training their model tractable. In this case, the choice of model architecture forces our hand in the choice of training algorithm.

Black-box models Black-box models approximate an arbitrary function with a parametrized model, most commonly a neural network. They are typically trained with supervised learning algorithms: an off-the-shelf optimizer searches for the model parameters that minimize a loss $l(y, \tilde{y})$ between the ground truth y and prediction $\tilde{y}$. A key design decision is the choice of output, e.g., whether to predict a distribution or a point value.

Fugu, NTT, and CausalSim use two common approaches for their primary predictions: Fugu is a probabilistic model and predicts a distribution of download times. NTT predicts the download time as a point value. CausalSim also predicts the download time as a point value and additionally predicts the buffer size to predict rebuffering more accurately. Alternatively, we could also simulate the buffer size from the download time:

$$\text{buffer}_{t+1} = \max\left(0,\ \min\left(\text{buffer}_{max},\ \text{buffer}_t - dt + d\right)\right) \tag{4.6}$$

where dt is the predicted download time and d the video chunk duration. Any time above buffer_{max} is spent *waiting*, as we can only download the next chunk once there is free space. Time below 0 is time spent *rebuffering*.

Biased training data CausalSim argues that the training data is biased, and thus requires data from an RCT with multiple ABRs and aims to remove this bias with the help of adversarial learning (see Section 4.1).

More generally, such data biases imply that the training data does not sufficiently cover the space of possible system evolutions (called *trajectories*) to learn the system dynamics. In data-driven control, training data that fully explores the trajectory space is called *persistently exciting*. The more persistently exciting the data, the better we can learn the system dynamics [108]. Do we need an RCT with multiple ABRs, or can a single ABR provide persistently exciting data to learn unbiased network dynamics?

In summary, we investigate the following questions:

How much does the choice of output matter?
What determines "good" training data?

Application	System	Model	Latent Space	History Metrics	History Length	Output
Prediction	Puffer	Single Model	Implicit	Chunk size Download time TCPInfo	8 Chunks	Download time (distribution)
	NTT	Encoder/Decoder	Explicit high-dim. (black box)	Packet timestamp Packet size Packet delay	1024 Packets	Download time (point value)
Counterfactual	CausalSim	Encoder/Decoder with Adversary	Explicit 1 dim. (black box)	Chunk size Download time Buffer level	1 Chunk	Download time, next buffer level (point value) ABR (distribution)
	Veritas	Modified HMM	Explicit 1 dim. (Available bandwidth)	Chunk size Download time TCPInfo Buffer level	Full trace ($\leq$ 200 chunks)	Bandwidth trace (distribution)

Table 4.1: Overview of key differences between Fugu, NTT, CausalSim, and Veritas.

4.3 EVALUATION

In this section, we put the questions posed in Section 4.2 to the test. We train and evaluate 49 different model variants with 5 different datasets representing different times and network environments, considering both download time prediction and counterfactual reasoning. We evaluate each model with over 12 stream-years of video data.

Limitations We discuss the limitations of our evaluation in Section 4.4, but want to highlight a key point here: The majority of our results are based on the Puffer dataset, which is limited to North American users and network environments and has been relatively stable over time.[3] In other parts of the world, network dynamics may be different and lead to different conclusions. We are working to collect a geographically diverse dataset and present preliminary results at the end of this section.

4.3.1 *Models, datasets, and metrics*

Models We train and evaluate three different model architectures:

LINEAR A linear, Fugu-like model. It combines history (past observations) and action into a single input.

ENCODER A linear encoder/decoder model. The encoder processes the history and predicts an explicit latent state. This latent state is then combined with the action and passed to the decoder.

TRANSFORMER Encoder/decoder model with Transformer encoder.

By default, we train these models to predict download times; to use them for counterfactual simulations, we use the hand-crafted buffer model described in Section 4.2.5. We train these models with different latent space dimensions, history metrics and lengths, and outputs (including directly predicting buffer levels). We trim the total space of combinations to 44 distinct models. We provide the model hyperparameters in Appendix A.3.

We also train and evaluate 5 different variants of CausalSim, varying the importance of the adversarial decoder as well as removing RCT data. Finally, we evaluate the published Veritas model. We are not able to retrain this model, as the training code is non-functional. Veritas' custom training is complex and we were unable to repair it. We have reached out to the authors, but at the time of writing, the issue remains unresolved.

3 When retraining with Memento in Chapter 2, after retraining a few initial times, we did not need further retraining for the rest of our deployment (over half a year).

Datasets CausalSim and Veritas are trained and evaluated on Puffer data, collected between summer 2020 and 2021, including only "slow" streams with an average delivery rate below 6 Mbps. For a fair comparison, we also use this subset. Additionally, we consider "fast" streams with an average delivery rate above 6 Mbps and data from 2023. This gives us four datasets with different environments and times. We use the following shorthands:

SLOW/FAST STREAMS refers to the environment for training or testing.

SLOW/FAST MODEL refers to a model trained with slow/fast streams.

2021/2023 refers to the time ranges 2020–2021, and 2023, respectively.

For evaluation, we use a month of data each.[4] For training, we use the whole time range, excluding evaluation data. Unless otherwise stated, we show results for the 2021 slow model evaluated on 2021 slow streams.

The Puffer dataset lacks packet-level data. This prevents us from evaluating NTT. To address this, we have extended the Puffer media server to collect packet traces, and are preparing a global experiment with real users to create a geographically diverse dataset. This is a lengthy process that is not yet completed. We tested our modified media server in cloud environments, streaming between Europe, South America, and Asia. We use these results as a preliminary packet dataset, even though it is comparatively small, containing 245 stream hours, 200 for training and 45 for evaluation.

Metrics For download time prediction, we compute the mean squared error (MSE).[5] We also compute the 99th percentile error per stream, averaged over all streams (99th percentile MSE), as a measure of tail performance.

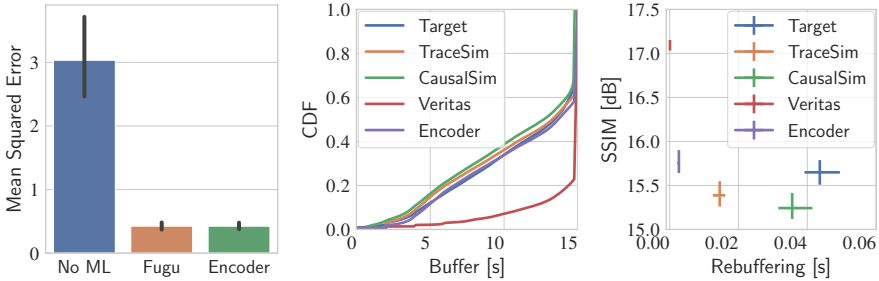
For counterfactual reasoning, we use Linear BBA as the target ABR and all other ABRs as sources. We report the Earth Mover's Distance (EMD) between the counterfactual buffer distribution and the target ABR buffer distribution in the same environment and time. A smaller distance indicates a more closely matching ABR state. In addition, we also evaluate the Quality of Experience (QoE), measured by image quality and rebuffering time.

We aggregate each metric per day and report the mean and 90th percentile bootstrap confidence interval over all days. This allows us to put differences in the context of random daily fluctuations to judge their significance.

By default, we report MSE and EMD for the 2021 slow model on 2021 slow evaluation data. We show other metrics, environments, and times when they show significantly different patterns.

4 CausalSim is evaluated on a year of data instead of one month. We considered this and found no significant differences. Thus, we opted for a shorter period to allow for more experiments.

5 Our model variants do not use probabilistic predictions, thus the logscore as used in Chapter 2 is not applicable. For Fugu, we report the expected MSE over all predictions.



(a) Download time: Fugu and a linear Encoder outperform a baseline that uses the harmonic mean of the throughput.

(b) Counterfactual, buffer CDF: all models except Veritas estimate the target distribution well, the Encoder performs best.

(c) Counterfactual, QoE: the Encoder predicts image quality best, but underestimates rebuffering, vice versa for CausalSim.

Figure 4.2: Comparing an Encoder, Fugu, CausalSim, and Veritas with non-ML baselines. Not shown: a linear model performed identically to the Encoder; a Transformer performed worse but still outperformed the baselines.

4.3.2 *Model architectures*

Figure 4.2 shows the comparison of different model architectures for download time prediction and counterfactual reasoning on the 2021 slow streams for a fair comparison with CausalSim and Veritas. In comparison to non-ML baselines, Fugu (we use the Fugu_{Feb} variant, see Chapter 2), CausalSim, and Veritas, it shows a linear Encoder architecture. We show a 'baseline' Encoder with 128 latent space dimensions and 8 video chunks of history, including minimal chunk information (timestamp, size and download time).

Implicit vs explicit latent state In all experiments, the linear architecture performs equally well as the Encoder, i.e., we observe no performance difference between implicit or explicit latent state estimation (not shown).

To our surprise, we found a Transformer encoder extracts a *worse* latent state. Its prediction error is 2× higher than the Encoder (not shown). We observe better results with packet data (see Section 4.3.6 below).

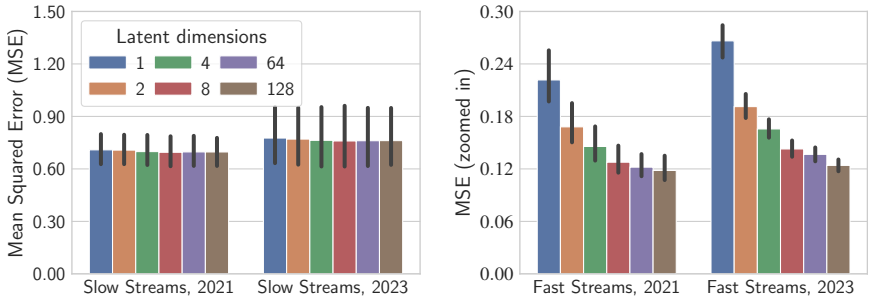
Download times Figure 4.2a shows the download time prediction error. Both Fugu and the Encoder perform similarly; there is no significant difference between the probabilistic and point value prediction. Both outperform an ML-free baseline, which uses the harmonic mean (HM) of the throughput over the last 8 chunks to predict the next download time. This is in line with results presented by the authors of Fugu that similarly show an HM error an order of magnitude higher than Fugu [16, Figure 7].

Counterfactual buffer distribution Figure 4.2b shows the counterfactual buffer distribution. As a non-ML baseline, we use a trace simulator that replays the observed throughput trace. The simulator, CausalSim, and Encoder perform similarly, with the Encoder matching the overall distribution most closely, yet underestimating low buffer levels (show in more detailed in Section 4.3.5 below). CausalSim tends to overestimate low buffer levels, in line with observations made by the authors of Veritas. However, we observe discrepancies with the results reported by CausalSim and Veritas.

The authors of CausalSim find it to outperform a trace simulator [102, Figure 7a], while both perform similarly in our evaluation. We find two probable causes for this: first, their reported EMD may not be statistically significant, it falls within the 90% confidence interval over daily fluctuations; Nevertheless, we compare their trace simulator to ours to find alternative explanations, finding only one notable difference: our trace simulator excludes sessions if there is insufficient observed data to allow any choice of initial chunk size for the simulated ABR algorithm. These are typically traces with only one or two video chunks that are recorded when a user is flipping through channels. We also exclude these traces from all systems for a fair comparison; it seems that some of the observed differences may be due to these ultra-short traces; not due to improvements on longer traces.

The discrepancy with Veritas is more striking. Veritas claims to perform similarly to CausalSim while offering a more interpretable and generalizable causal model. Our results show a significantly different buffer distribution. However, Veritas is trained and evaluated on a only single day of Puffer (August 24, 2020). On this day, Veritas does perform similarly to CausalSim. Yet, as our results show, it does not generalize well. Unable to retrain Veritas, we could not verify whether it would perform better with a larger dataset. Notably, Veritas is also much slower than CausalSim or the Encoder, taking approximately 13× as long to evaluate on the same hardware.

Counterfactual QoE Figure 4.2c shows the counterfactual QoE. CausalSim is closest to the target in terms of rebuffering but underestimates the image quality slightly. This matches the buffer distribution: CausalSim overestimates low buffer levels and thus more likely predicts rebuffering. The Encoder matches the overall distribution more closely, which is reflected in the most accurate image quality prediction. However, our hand-crafted buffer model fails to predict low buffer levels well and predicts a significantly lower rebuffering rate. We investigate this further in Section 4.3.5 below, and show that learning to directly predict the buffer level can improve the prediction of rebuffering compared to a hand-crafted approach.



(a) Slow streams. **(b) Fast streams.**

Figure 4.3: A high-dimensional latent space seems useless for a slow model on slow streams, but significantly improves generalization, in particular over time.

4.3.3 *Latent Space*

Figure 4.3 show the impact of a higher-dimensional latent space on the download time prediction performance of the Encoder model. Figure 4.3a shows the 2021 slow Encoder evaluated on slow streams, where we observe no changes in the average MSE. This reinforces the design choices by CausalSim and Veritas: both were trained and tested in the same environment and performed well with a just one-dimensional latent space.

Perhaps surprisingly, evaluating the same model on fast streams shows a significant improvement with a higher-dimensional latent space. While throughputs are higher and thus the MSE is lower overall, a second dimension already reduces the MSE by 24 %, and this benefit increases to a reduction of 46 % with 128 dimensions (Figure 4.3b). The model has learned to extract a complex latent representation, even if it may not be immediately apparent or seemingly not useful in the current environment.

We find that these results are conditional: while generalization for the fast model to slow streams is also improved, the benefit is smaller (6.6 %, Table 4.2). We suspect that the slow stream dataset is more persistently exciting, which allows the model to extract more diverse patterns—if the latent space is large enough. We present additional evidence for this in our evaluation of adversarial training (see Section 4.3.5 below).

Finally, Figure 4.4 shows the results for a slow model in counterfactual simulation. Simulating the buffer distribution appears to be much easier for fast streams, as the buffer is almost always full. Thus, we cannot observe notable benefits from a higher-dimensional latent space for the slow model. We still observe the benefits of generalizing the fast model to slow streams.

		Mean Squared Error		
Train	Test	1 Dim.	128 Dim.	Δ [%]
slow	slow	0.71	0.70	-1.70
	fast	0.22	0.12	-46.67
fast	slow	1.08	1.00	-6.60
	fast	0.07	0.07	-2.03

Table 4.2: Download time: A higher latent dimension aids generalization.

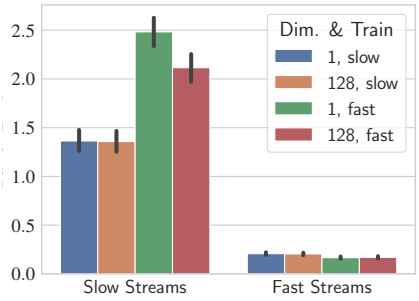


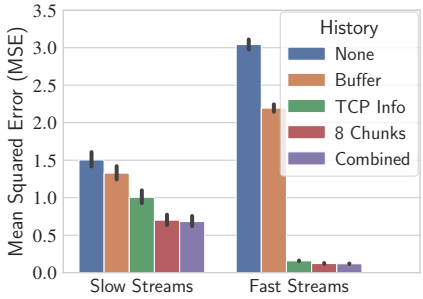
Figure 4.4: Counterfactual: a higher latent dimension has a lesser impact.

4.3.4 *History*

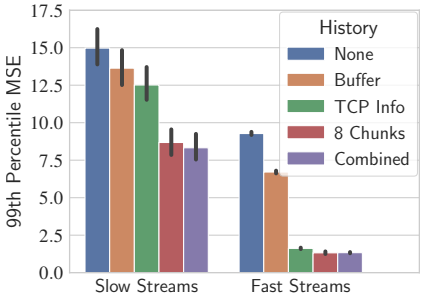
Figure 4.5 compares different metrics to represent the recent history, from coarse-grained buffer level to aggregated TCPInfo statistics, to using several past video chunks, the finest granularity available in the Puffer dataset. The buffer level alone does allow the model to extract a useful latent state, and performance is not significantly better than providing no history at all. It is marginally more useful for generalization, but still far behind more fine-grained metrics. TCPInfo as well as past chunks significantly improve the prediction performance both in the same environment as well as for generalization, with chunks slightly outperforming TCPInfo. In particular, we notice a difference between average (Figure 4.6a) and tail ((Figure 4.6b)) performance: TCPInfo provides a noticeably lesser benefit for the tail, where the more fine-grained chunk history appears to be crucial.

We can confirm the observation made by the Puffer authors that TCPInfo reduces the error for the first chunk in a stream where no chunk history is available (Figure 4.7). This benefit vanishes with the second chunk, and we conjecture that it is not any particular information in the TCPInfo that is helpful but rather the general fact of providing any history for the first chunk. However, a video stream does not start with the first chunk, as we usually download the website or a manifest file first. It may be possible to track this download as initial history instead of TCPInfo. The Puffer data does not include this information, and we leave this idea for future work.

For overall performance, there is no difference between using a chunk history alone or in combination with TCPInfo, and we thus omit TCPInfo in the following. Figure 4.6b shows that a longer history is beneficial, but there are diminishing returns, in particular in the slow environment, where improvements for longer histories are barely out of the confidence interval.

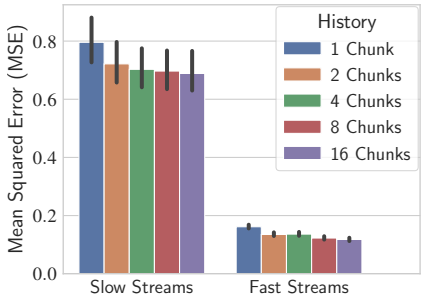


(a) Mean Squared Error (MSE).

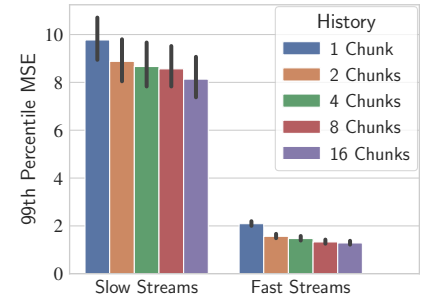


(b) 99th percentile MSE.

Figure 4.5: Using a fine-grained chunk history outperforms coarse metrics like buffer level or TCPInfo, in particular at the tail and for generalization.



(a) Mean Squared Error (MSE).



(b) 99th percentile MSE.

Figure 4.6: The slow model benefits from a longer history. The relative improvements are more significant when generalizing to fast streams.

			Δ Mean Squared Error [%]			
			1 → 8 Chunks		1 → 128 Dim.	
MSE	Train	Test	1 Dim.	128 Dim.	1 Chunk	8 Chunks
Mean	slow	slow	-12.6	-13.5	-0.7	-1.7
		fast	-15.3	-26.8	-38.3	-46.7
	fast	slow	-26.4	-32.3	1.6	-6.6
		fast	-7.1	-8.9	-0.2	-2.0
99th Perc.	slow	slow	-13.2	-13.1	0.7	0.9
		fast	-23.6	-40.5	-20.6	-38.1
	fast	slow	-19.5	-22.9	0.6	-3.6
		fast	-9.3	-9.2	-0.9	-0.8

Table 4.3: Relative benefits for increasing history *or* latent space.

We also investigate the relationship between the history and the latent space dimension in Table 4.3, which shows the relative difference between increasing the history length (for a fixed latent space dimension of 1 or 128) as well as increasing the latent space dimension (for a fixed history length of 1 or 8 chunks). For an increasing history length, performance universally improves. On the other hand, a higher-dimensional latent space is more beneficial for generalization, but without a sufficiently long history, we cannot realize these gains, in particular at the tail: increasing the latent space dimension from 1 to 128 reduces the 99th percentile MSE of the slow model tested on fast streams by 20.6 % when using single chunk as history, but by 38.1 % when using 8 chunks. This indicates that we require a sufficiently long history to distinguish some patterns in the higher-dimensional latent space. With a single data point, we cannot fully leverage the latent space.

For counterfactual buffer distribution prediction, we expect the buffer level to be a more relevant aggregator of history. Figure 4.8 shows that this is indeed the case. We test four settings: no history, i.e., using only the current chunk from the source ABR, using the source buffer level as history, and using the last 8 chunks. Compared to using no history, the buffer level reduces the EMD by 24 %. When testing on fast streams, no history fails to generalize, even though the environment is easy (see Figure 4.4). Using the buffer level is a significant improvement over no history, but using a chunk further reduces the EMD by 40 % compared to the buffer level.

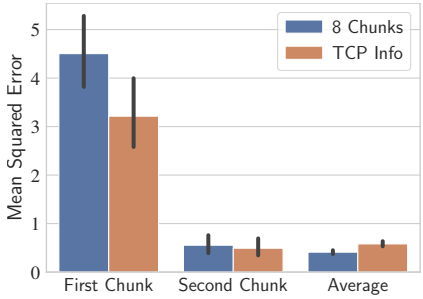


Figure 4.7: TCPInfo statistic improves the download time prediction of the first chunk in the video stream, as there are no past chunks to use as history.

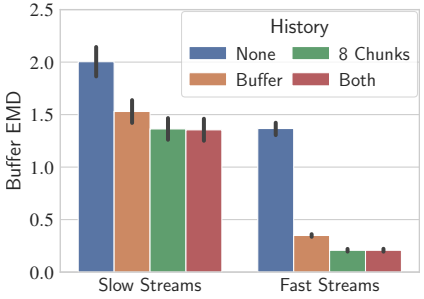


Figure 4.8: For predicting the counterfactual buffer distribution, using the source buffer level is effective, but using a trace of chunks is still better.

4.3.5 *Training*

Buffer prediction For the next experiments, we focus on counterfactual reasoning. In general, a download time prediction task allows an Encoder to learn a good representation of the latent network state. We used a hand-crafted buffer model to use this Encoder for counterfactual

This knowledge transfers to counterfactual reasoning, where the Encoder predicts the overall buffer distribution and image quality best. However, it significantly underestimates rebuffering, as it tends to underestimate the fraction of low buffer levels (Figure 4.9b). These buffer levels are crucial for predicting rebuffering, as a full buffer rarely leads to rebuffering.

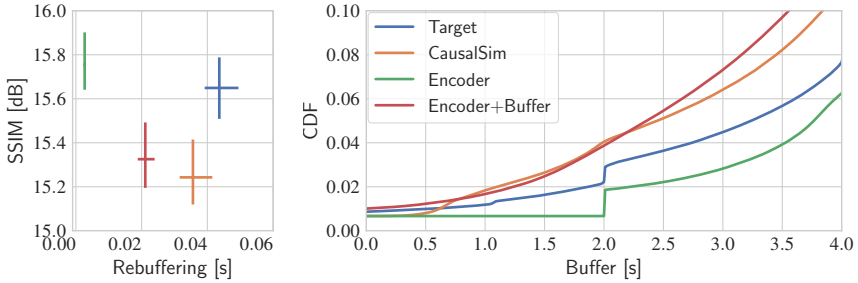
One benefit of using an Encoder model with explicit latents is the flexibility to add additional decoders for different prediction tasks. When adding a decoder predicting the buffer level, the Encoder can learn to predict the rebuffering rate more accurately (Figure 4.9a) and the predicted distribution for low buffer levels matches CausalSim more closely (Figure 4.9b).

Adversarial learning Yet, the performance does not fully match, which raises the question: is this caused by subtle differences in the model hyperparameters, or is the adversarial training of CausalSim the key to its success? We investigate this by training five different variants of CausalSim. For the first two variants, we remove the adversarial encoder. For the first variant, we also use training data from only a single ABR. For the remaining three variants, we gradually increase the weight of the adversarial loss.

To our surprise, Figure 4.10a shows that the adversarial decoder does not improve CausalSim. A strong adversarial loss can even prevent generalization: CausalSim trained on fast streams performs well on slow streams with no or low adversarial loss, it breaks down with higher adversarial loss.[6] This implies that CausalSim does not learn an ABR-biased latent state.

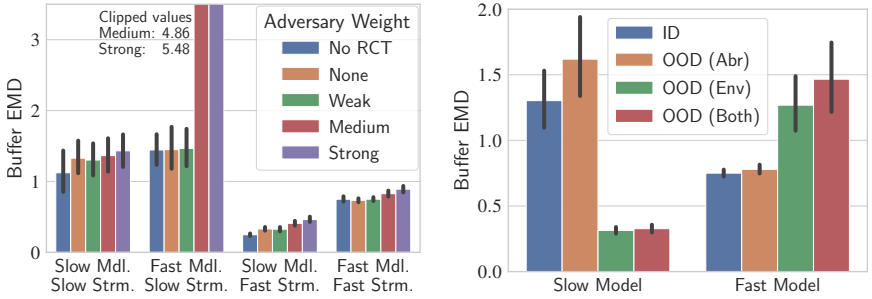
Figure 4.9b investigates this further by comparing the performance of CausalSim models in-distribution (i.e., a slow model evaluated on slow streams, with a source ABR seen during training) with two different out-of-distribution settings: *(i)* OOD(ABR): a source ABR not seen during training; and *(ii)* OOD(Env) a different environment, i.e., testing the slow

6 This result, once again, seems to contradict the results presented by the Veritas paper, which finds that training CausalSim without RCT data significantly degrades its performance. We hypothesize that this may be a result of removing RCT data without removing the adversarial decoder. With only a single ABR, the adversarial decoder can always predict perfectly. Thus, the encoder cannot learn how to minimize the adversarial loss, which may degrade performance similar to a strong weight in Figure 4.10a. We have reached out to the Veritas authors to clarify how they retrained CausalSim, but have not received a response at the time of writing.



(a) Counterfactual QoE **(b) Counterfactual: buffer CDF (zoomed in).**

Figure 4.9: An additional buffer prediction for the Encoder improves the prediction of the rebuffering rate but degrades image quality prediction.



(a) CausalSim performs better without the adversarial encoder, and degrades with increasing adversarial weight. It learns well without RCT data.

(b) Out-of-distribution (OOD): The environment (slow/fast) has a far greater impact than ABR: CausalSim does not seem to learn a biased latent state.

Figure 4.10: Adversarial training is not the key to CausalSims performance.

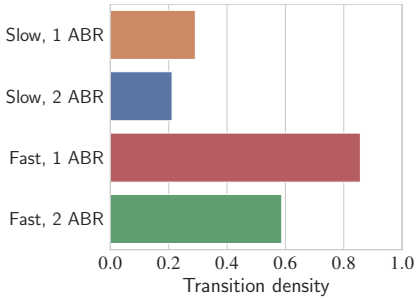


Figure 4.11: The slow dataset is more persistently exciting (lower density) than the fast dataset, and additional ABRs only marginally improve this.

model on fast streams and vice versa. If CausalSim learns a latent state strongly biased towards some ABRs, the performance of OOD(Abr) should degrade significantly, but the evidence for this is weak. While there is some degradation, it is barely outside the confidence intervals, and far less significant than changes in the environment.

As we argue in Section 4.2, it may be more important for the observed trace to be *persistently exciting* in order to learn a rich latent state. We test this hypothesis by extracting simplified trajectories from the datasets, containing state transitions in the form of $(\text{buffer}_k, \text{action}_k, \text{buffer}_{k+1})$, that is, the buffer level before and after a chunk download and the chosen chunk size. A set of trajectories is the more persistently exciting, the more it allows to explore the state space. While there is no closed form to quantify this in a nonlinear, noisy setting, we use Memento (see Chapter 2) to estimate the density of the trajectory sample space. If the density is low, individual trajectories are not similar and thus explore the space better than if the density is high, which implies many similar trajectories.

Figure 4.11 show that the degree of persistent excitation as measured by the density matches our previous observations: the difference between environments is much larger than the difference between including one or more ABRs; the slow environment is generally more persistently exciting. This also matches the fact that we learn a more generalizable latent state only from the slow dataset, and not as much from the fast dataset (Section 4.3.3); the fast dataset is less diverse and there may be fewer patterns to learn.

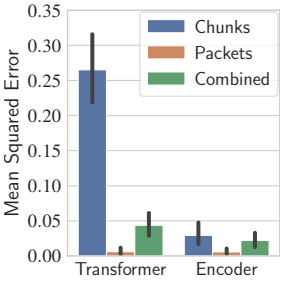
4.3.6 *Packet traces*

In the following, we present preliminary results on the packet-level dataset. There are two important limitations to this dataset: *(i)* it is much smaller than the Puffer dataset, containing only 45 hours instead of several years of streaming data; and *(ii)* it was collected between cloud servers in well-connected data centers. As a result, all streams are fast and the environment is stable. It is "too easy" for counterfactual reasoning: the buffers are always full, and all algorithms perform equally well. Nevertheless, for download time prediction, our results show the potential of packet-level data.

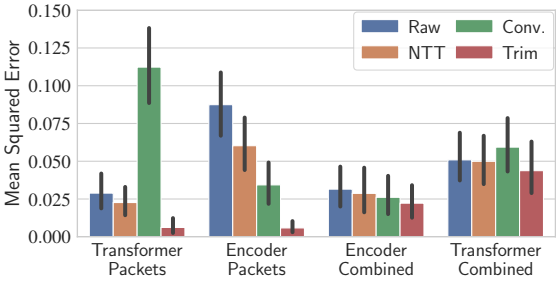
Figure 4.12 compares a history of 8 chunks with a history of 1024 packets. First of all, a packet Transformer performs much better than its chunk counterpart, on par with the linear Encoder (Figure 4.12a). Interestingly, a packet-only history significantly outperforms both a chunk-only history and a combination of chunks and packets.

Additionally, we can evaluate NTT's hierarchical aggregation. Compared to no aggregation or an off-the-shelf convolutional layer, the hierarchical aggregation works best for the Transformer (Figure 4.12b); in particular, using a convolutional layer does not seem to work with the Transformer. This is in contrast to the linear Encoder, which benefits more from the convolutional layer compared to the hierarchical aggregation. For both models, however, the most effective strategy is to use only a trimmed packet sequence of only 32 packets instead of aggregation.

However, we suspect that these results may be biased by the cloud environment, where long-term patterns may not play a significant role. We are currently working on collecting a global dataset with real users, and plan to release it publicly to enable future research in this direction.



(a) For packet-level data, a Transformer architecture shows its potential.



(b) Packet aggregation: no aggregation, NTT-like hierarchical aggregation, convolutional layers, and trimming the sequence to the most recent packets.

Figure 4.12: In our preliminary dataset, a packet history allows for much better download time predictions than chunk history.

4.4 DISCUSSION & FUTURE RESEARCH

In this chapter, we have discussed predicting the latent network state. We explained how this problem is a fundamental part of downstream tasks like download time prediction or counterfactual reasoning. Furthermore, we explained several components necessary to learn the latent network state. Based on this common problem, we analyzed four different systems (Fugu, NTT, CausalSim, and Veritas) and identified how they implement each component of learning latent state. From their different implementations, we formulated key design questions, which we investigated with over 12 years of video stream data. We made several notable discoveries:

- **ML models outperform hand-crafted solutions.** Veritas implements a complex causal graph with a modified Hidden Markov Model but ultimately underperforms in our evaluation. It is both slower to run and does not generalize to data outside of its training environment. Similarly, a hand-crafted buffer model predicts rebuffering significantly less accurately than learning the buffer evolution from data.
- **A high-dimensional latent space helps generalization.** Both CausalSim and Veritas use a one-dimensional latent space, which seems optimal for their original environment. However, a larger latent space helps the model generalize better to new environments. Models with a large latent space performed significantly better with data from different environments as well as data from different time periods.
- **A fine-grained history is important for tail performance.** A history of video chunks allowed the model to learn a more informative latent state, enabling both the benefits of high-dimensional latent space above, as well as improving tail performance. Coarser metrics like the application buffer level or TCPInfo statistic lead the model to estimate a worse latent state, reducing downstream task performance. In a small preliminary dataset, we observe that an even finer packet-level history can further improve performance.
- **Persistent excitation is more important than RCTs.** We find that data from a single ABR can be highly persistently exciting, covering a wide range of network behaviors. Additional ABRs as part of a Randomized Controlled Trial (RCT) do not necessarily explore the space further. As a result, we do not find significant evidence that the estimated latent space is biased towards specific ABRs. Strategies like CausalSim's adversarial training to remove ABR biases seem unnecessary, and we were able to estimate the latent state equally well without it.

Limitations An empirical study as presented in this chapter is fundamentally tied to the data it is based on. The majority of our results are based on the Puffer dataset. This is a rich long-term dataset, yet it is limited to North American users and network environments. In other parts of the world, network dynamics may be different, as other technologies are prevalent, networks are differently managed, or other applications are more popular. Additionally, it does not contain packet-level data, which prevents us from evaluating the benefits of NTT-like solutions.

We are in the process of collecting a geographically diverse dataset that includes packet-level data to complement the Puffer dataset in future research. While this process is ongoing, we presented preliminary results highlighting the potential benefits of packet-level data, and thus such a dataset. However, this preliminary dataset was only collected in a cloud environment and does not include geographically diverse data from real users, which we aim to address with our ongoing data collection efforts.

Additionally, while our analysis is in-depth, it is not exhaustive. We closely evaluated Fugu, NTT, CausalSim, and Veritas, but many other ML-based systems in networking also learn network traffic models, from congestion control [72–75] over RL-based video streaming [53] to emulating traffic between datacenter pods [83]. Future research may extend our analysis to include these systems to understand how they learn latent network state.

Finally, while our model for learning latent network state is general, it is also explicitly focused on predicting the *network* behavior. Notably, we assume that application decisions such as the ABR chunk size are provided as input to the model, and do not learn them. In other domains such as security, we often aim to not understand not the network, but the applications using it. While our work is not directly applicable to this, our methods may be adapted to a more application-focused approach.

A place for domain knowledge In our results, we find that hand-crafted models underperform compared to black-box models learned from data. This does not imply that domain knowledge is useless, but rather that it perhaps should be applied differently. Overly tailoring models or the latent space to any particular problem limits generalizability as well as the possibility of reusing models for different yet similar tasks. Instead, we argue for building models as general as possible and focus domain knowledge on ensuring that we provide a diverse dataset that captures the full range of network behaviors, as well as a training output and loss function that promotes learning important network dynamics.

In this section, we have only taken a small step in these directions, by analyzing the impact of predicting buffer levels in addition to download times on the one hand, and by evaluating the degree of persistent excitation of the training datasets on the other. Nevertheless, we believe that this is an important avenue for future research.

A case for model explainability A key disadvantage of black-box models like CausalSim is their lack of inherent interpretability compared to hand-crafted models like Veritas. This applies to the latent space representation as well: we observe that a larger latent space dimension helps to generalize, but we do not know what each dimension represents. This lack of understanding can lead to problems, as the model may overfit irrelevant features, learn spurious correlations, or exploit shortcuts in the data, i.e., patterns that have high predictive power in the training data but do not generalize.

Model explainability tools such as Trustee [109] aim to fill this gap. Trustee extracts an interpretable decision tree from a black-box model, allowing to reason about the model's decisions and identify potential shortcuts or spurious correlations. These techniques can be used to actively ensure that the chosen inputs and training tasks are relevant and allow for learning meaningful latent network dynamics. They can also help to better understand some of the patterns we observed in this chapter, such as understanding how additional latent dimensions impact model decisions.

A question of scale Finally, when designing models to learn latent network dynamics, we must consider scalability. To deploy these models in real-world scenarios, we need to consider the computational cost of running them for potentially millions of users. In the future, we must not only answer *how can we learn the latent state?* but also *how can we do so efficiently?*.

Consider Transformer models like NTT (Chapter 3). They have the potential to improve generalizability, but at the same time, they are computationally much more expensive than the simple linear Encoder we used in this Chapter. Where the Encoder can run efficiently on a CPU, the Transformer requires a GPU for the same performance. Similarly, increasing the latent space or using a larger history also increases the computational cost. Investigating the trade-off between model complexity and computational cost is an important direction for future research, in order to offer the most useful state estimation for a given computational budget.

5

LEARNING IN THE DATA PLANE: HARDWARE-SOFTWARE CO-DESIGN LEVERAGING PROGRAMMABLE NETWORKS

In the previous chapters, we focused on improving ML-based applications. A key reason for using ML is limited observability: applications must estimate the latent state from past observations, as discussed in Chapter 4.

Networks can improve observability by sampling traffic or collecting aggregate statistics. However, this is limited by the sheer amount of traffic in today's networks: we cannot sample all of it. To limit the overhead in communication and processing, networks must either sample at low rates or aggregate aggressively. In the end, this turns *not observable* into just about *barely observable*, as the data plane can only share a fraction of its insights.

This chapter explores how we can utilize programmable network devices to extract information from the network more efficiently, opening two avenues for improvement. We can use the improved network feedback:
(i) to guide sampling towards more informative subsets of traffic;
or *(ii)* to side-step sampling and directly optimize model parameters.

Our key insight: while learning predictive models is too computationally expensive for the data plane, *evaluating* their accuracy can be made feasible. This enables hardware-software co-designs that leverage the data plane's observability while offloading heavy computations to the control plane.

We present FitNets, a system for monitoring and anomaly detection implementing this insight with a closed loop between the data- and control plane. FitNets learns traffic distributions, as this is a simple yet useful learning task that allows adapting to changing traffic patterns quickly. We discuss extending FitNets to different models at the end of the chapter.

This chapter is organized as follows: Section 5.1 introduces programmable networking fundamentals. Section 5.2 provides a brief background and overview of FitNets. Section 5.3 explains the FitNets data plane which extracts feature values and scores the current distributions. Section 5.4 demonstrates how FitNets can improve sampling, while Section 5.5 shows how it can alternatively search for optimal model parameters without sampling. Finally, Section 5.6 presents preliminary evaluation results and Section 5.7 outlines future directions of data-plane assisted learning.

5.1 PROGRAMMABLE NETWORKS

This section provides an overview of classical and programmable networking, focusing on concepts that are relevant to the remainder of this chapter. We begin by summarizing the data- and control split in classical networks, followed by a simplified overview of programmable networks.

5.1.1 *Classical networking*

Classical networks feature a separation between the data plane and the control plane. The data plane is responsible for forwarding packets, while the control plane is responsible for managing the data plane configuration.

Data plane Network devices require a variety of components to parse and buffer packets, but the core of the data plane processing are *match-action tables*. As the name suggests, the data plane uses these tables to match packets against configured rules and execute corresponding actions. The most common example of such a table is the *forwarding table*, which matches packet destination address and sets the output port accordingly. These tables are implemented in hardware to process packets at line rate.

Classical network devices are often called 'fixed function' devices. They may support a large variety of features with different match-action tables, but these tables are pre-determined and cannot be exchanged. For example, we cannot modify the forwarding table to match the source address.

In addition to match-action tables, network devices also support data structures like *counters* and *meters*. Counters are used to count packets or bytes, while meters are used to measure rates or enforce rate limits. Like match-action tables, these data structures also fixed functions. Similarly, network devices support *sampling* packets with a configurable probability.

Control plane The control plane, on the other hand, is implemented in software and is responsible for configuring the match-action tables in the data plane. For example, the control plane may execute a complex routing algorithm to determine the optimal path through the network for each destination address. Once computed, it installs the resulting forwarding rules (matching destinations to ports) in the data plane.

The control plane also provides an interface to access other data structures, offering functions such as reading and resetting counters, configuring meter rates, or setting sampling probabilities.

5.1.2 *Programmable networking*

Programmable networks extend classical networks by allowing operators to modify the data plane's behavior, replacing fixed-function tables and data structures with reprogrammable alternatives.

Match-action pipelines Programmable switches process packets in a multi-stage pipeline. In each pipeline stage, one or more match-action tables can be applied, which can in turn read and modify both packet headers and metadata. The metadata contains variables such as output port, allowing them to be matched and updated like header values.

Programmable match-action tables can be freely configured to match and modify any header fields and metadata. Multiple tables can be applied in sequence, allowing for complex packet processing pipelines. Whether a table is applied or not is determined by the *control flow* of the pipeline, which defines the sequence and conditions under which tables are applied.

Match-action pipelines also support programmable counters, meters, and sampling. Additionally, they offer programmable *registers*. Registers are array-like data structures that can persist state between packets. Registers can be read and written by both data- and control plane.

Control plane The match-action pipeline's control flow, tables, and data structures are programmed using a domain-specific language like P4 [110].

The control plane compiles the P4 program and reconfigures the data plane accordingly. This compilation step also generates the control-plane interface to access the individual match-action tables, counters, meters, and registers, allowing them to be re-configured at runtime.

Limitations While programmable networks offer more flexibility than classical networks, they also have limitations. To ensure that packets can be processed at line rate, the control flow does not support loops. Thus, the number of stages in the pipeline determines the maximum depth of processing. Depending on the devices, 10 to 20 stages are common.

Additionally, the persistent state in registers is restricted in several ways. Total register memory is limited to a few Megabytes, depending on the device. Registers memory can also not be (re)allocated at runtime; all space must be assigned when the P4 program is installed.

Consequently, it is not possible to run complex algorithms in the data plane, as the number of operations and available memory is strictly limited.

5.2 TRAFFIC MONITORING WITH FITNETS

This section provides a brief background on network monitoring from common tools to in-network monitoring with programmable data planes. We highlight challenges, in particular the difficulty of monitoring rare events, and motivate the learning oftraffic distributions. Finally, we introduce FitNets and provide an overview of its inputs, outputs, and variants.

Network monitoring with sampled traffic Collecting reliable traffic statistics is a fundamental problem of network monitoring and is crucial for successfully operating a network. Traffic statistics are used in a wide range of network management tasks including capacity planning, traffic engineering, and billing. An especially important task is anomaly detection, which is essential to DDoS detection, root cause analysis, and many more applications. Despite this importance, detecting anomalies in network traffic is notoriously hard to get right. Anomaly detection requires establishing a sense of normality [111]. This includes monitoring *rare* events (e.g., tails of traffic distributions) to avoid errors on uncommon yet benign traffic.

Because of the scale of the Internet, collecting accurate statistics in general and rare events in particular remains challenging with current monitoring tools. The majority of common tools, e.g., NetFlow [112] and sFlow [113] are based on random packet sampling. Studies have shown that sampling with fixed low rates fails to estimate many important statistics [114, 115].

In-network monitoring With the advent of programmable network devices, a range of solutions have been proposed to leverage the data plane for monitoring. Two prominent directions are: (i) processing *queries* such as `filter` and `map` directly in the data plane [116, 117]; and (ii) aggregating statistics using probabilistic data structures like *sketches* (e.g., [118, 119]).

However, the limited data plane processing capabilities present a tradeoff: while the increased visibility can improve some aspects of monitoring, the limited capabilities can restrict others. Often, these limitations are especially affecting the detection of rare events. Sketches, for example, must allow for errors in their estimated aggregates, and their memory footprint can be determined as a function of the desired error bounds. These bounds are usually derived with respect to all traffic and are thus much tighter for heavy hitters than for rare events.[1]

1 Consider a 1% error bound and a total of 1000 packets to be counted in the sketch, which is split between a common flow of 900 packets and a rare flow of 100 packets. The error bound is 10 packets, which equals about 1.1% of the common flow, but 10% of the rare flow.

Despite these limitations, in-network monitoring has shown promising results, e.g., for DDoS detection: ACC-Turbo [120], a state-of-the-art detection system in the data plane, uses an online learning approach that adaptively clusters traffic. As it runs in the data plane, it can adapt quickly to all incoming traffic. However, it can only track the minimum and maximum values of traffic features,[2] and cannot track the distribution within a cluster. It identifies anomalies as high-volume clusters with high similarity, i.e., similar min and max values. As such, it is vulnerable to rare outliers that stretch the min/max value of a cluster, making an anomalous cluster seem dissimilar and harmless, even if the distribution remains narrow.

Opportunity We argue that applications such as anomaly detection can be improved by learning feature *(i) distributions* based on the *(ii) entire* traffic. Distributions allow deeper insights, in particular into rare low-probability events, which are often missed by sampling or "swallowed" by aggregation. But how can we learn accurate distributions? We cannot sample enough traffic to learn them accurately in the control plane: we are already missing rare events and cannot magically learn a distribution that is more accurate than the data we have. But we also do not have the resources to learn the distributions in the data plane. We must leverage the strengths of both data- and control plane to overcome their respective limitations.

FitNets Our key insight is that while learning distributions is only feasible in the control plane, verifying already learned distributions is possible in the data plane. This allows us to *close the loop* and learn distributions in rounds: after learning one or more candidate distributions in the control plane, we can evaluate them on all traffic, leveraging data plane visibility. With the evaluation results, we can refine the distributions in the control plane, and repeat the process in the next round.

We call this closed-loop system FitNets, and in the following chapters, we evaluate two different variants: *(i)* a sampling-based variant, which learns non-parametric distributions from samples, and uses the evaluation results to guide the sampling process; *(ii)* a search-based variant, which learns parametrized distributions by navigating their parameter space with Bayesian optimization, and uses the evaluation results to guide the search process to optimize distribution parameters. Both variants share the same data-plane pipeline for extracting features and scoring the predicted distributions. They only differ in the control-plane learning process, which can be selected based on requirements on the learned distributions.

2 Measurable properties like ports, size, inter-packet time, flow duration, etc.

Task	Location	Constraints	Feature
1	Switch 1	src(42.0.0.0/8)	burst_size
2			burst_duration
3		src(43.0.0.0/8)	queue_time
4	Switch 2	src(13.37.0.0/16) & proto(TCP)	packet_size

Figure 5.1: FitNets is configured with monitoring tasks: *where* to monitor *which features* under *which constraints* on the traffic, i.e., which subsets of traffic.

Inputs and outputs Operators configure FitNets with a list of monitoring tasks and in return, FitNets provides a stream of probability distributions per task along with their data-plane scores, updated after each round.

Monitoring tasks are specified by *(i)* the *location* where traffic is monitored, i.e., the programmable switch; *(ii) constraints*, i.e., which subset of traffic is monitored; and *(iii)* which *features* are monitored (Figure 5.1). For each location, multiple constraints can be specified, and for each constraint, multiple features can be monitored. We have implemented common constraints and features, but in general, FitNets can be extended to any constraint that can be matched on programmable network devices, as well as any feature that can be extracted. For *constraints*, we implement arbitrary combinations of source address, destination address, protocol, source port and destination port. For *features*, we implement both features that can be directly extracted from a single packet, e.g., packet size, as well as more complex stateful features, e.g., inter-arrival time and flowlet size. We focus on flowlet instead of flow state to limit memory usage. Flowlets are short bursts of packets, separated by an inter-packet gap of inactivity, and the number of concurrently active flowlets for gaps such as $500ms$ is typically well below $100k$, feasible for programmable network devices [121]. Flows can be active for minutes or longer, and thus millions of them can be concurrently active at the same time. Solving this challenge is out of scope for FitNets.

The sampling variant of FitNets requires an adaptation objective; it can either maximize the accuracy of learned distributions given a fixed sampling budget or minimize the sampling rate while meeting a target accuracy. The search variant requires a distribution parameter space to search. Finally, we must specify the round duration, i.e., the update frequency.

5.3 DATA-PLANE SCORING

In the following sections, we will discuss two different ways of learning distributions in the control plane, leveraging data-plane scores to guide the learning process. In this section, we focus on one round of FitNets. Given a probability density estimate f_n by the control plane, we discuss how we can use programmable network devices to score this estimate using *all traffic*.

Figure 5.2 shows the data-plane processing pipeline of FitNets, from matching packets and extracting feature values over sampling to scoring.

Matching and feature extraction We extract features from traffic matching the constraints specified in the monitoring tasks. For the constraints, FitNets employs a table matching the flow id (5-tuple) of a packet. The table uses ternary matches, allowing flexible queries such as ranges and longest-prefix-match, covering even complex constraints such as 'UDP packets from the subnet `42.0.0.0/8` with a destination port below 1024'.

If required by any monitoring task, FitNets checks the *flowlet state* using a hash table. This table stores the flow id and the timestamp to check for collisions and timeouts. A packet is considered to start a new flowlet if there is no entry in the hash table, or if the current entry is older than a configured *inter-packet gap*. The flowlet state can be either (i) the packet starts a new flowlet; (ii) the packet belongs to an active flowlet; (iii) the packet explicitly ends a flowlet (e.g., after a TCP FIN); or (iv) the state could not be determined because of a hash collision.FitNets keeps a collision counter such that operators can resize the flowlet state table manually if there are too many collisions, as memory cannot be changed at runtime.

Finally, FitNets extracts the feature values. If the same feature is used for multiple monitoring tasks, it is only extracted once and re-used. Flowlet features might keep additional state, e.g., a byte counter for flowlet sizes. Flowlet features are only extracted if there has *not* been a collision. FitNets cannot extract every feature after every packet, e.g., the flowlet duration cannot be extracted while the flowlet is still active. Only successfully extracted feature values are sampled or used for scoring.

Sampling For each task, FitNets individually decides whether the feature value is sampled via rate-limiting meters configured from the control plane. If any feature value is sampled, FitNets creates a new packet containing all sampled features and forwards it to the control plane. This step is skipped for the search variant of FitNets, as it does not require sampling.

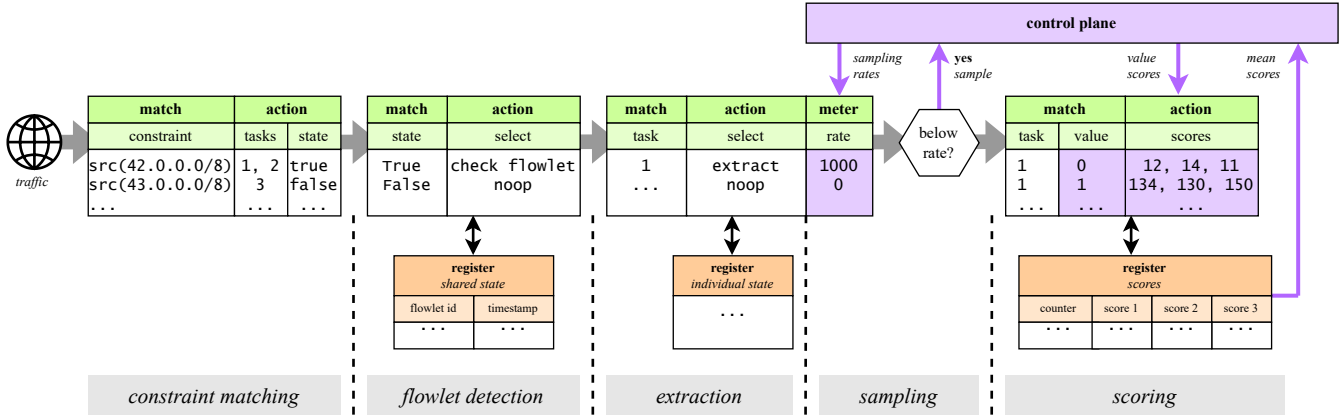


Figure 5.2: The FitNets data plane first matches traffic and extracts features, then samples and scores predictions. Multiple features can be processed in parallel; we show the per-feature pipeline here. Flowlet state is split in a shared part (flowlet id, timestamp) and feature-specific state (e.g., flowlet size byte counter). Sampling is skipped for the search variant of FitNets.

Proper scoring rules A proper scoring rule [36] assigns a score $S(f_n, x)$ to probability estimate $f_n(x)$ for a value x. The sample score for a sample $\mathcal{X}$ is the average score over all sample values:

$$S(f_n,\ \mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} S(f_n,\ x) \tag{5.1}$$

A proper score can only be maximized by the true distribution. That is, if $\mathcal{X} \sim f$, then the maximum score can only be achieved by $f_n = f$.

As we can only compute the score for a finite sample, it is noisy and may be unreliable for small sample sizes. Here, programmable data planes shine as we can compute the sample score for *all observed traffic*, i.e., the largest possible sample size and thus most reliable score possible.

Data planes typically cannot handle floating-point operations, so we cannot compute the average score directly. Instead, we keep a running sum of scores and a value counter in the data plane, which enables us to compute the average score in the control plane.

Lookup tables and binning Depending on the complexity of f_n and S, it may be impossible to compute $S(f_n, x)$ in the data plane, e.g., we cannot compute the logarithmic score as the data plane cannot compute logarithms. Thus, we translate $S(f_n, x)$ into a lookup table storing pre-computed scores over the range of possible feature values and corresponding predictions.

However, this range can be enormous, resulting in unsatisfiable memory requirements for the lookup table. To address this issue, we implement the following binning scheme to handle large feature ranges with efficient data-plane lookup methods: we use bins with a width of a power of 2, which essentially enables us to use longest-prefix matching for the lookup table as well as other TCAM-based range matches. While this is our default choice, FitNets can work with any other implemented lookup method.

Distance measures Proper scoring rules have an associated distance measure, which reflects the difference between the true and estimated distribution. For example, the logarithmic score $\log(f_n(x))$ is associated with the Kullback-Leibler divergence [32]. Maximizing the expected logarithmic score minimizes the Kullback-Leibler divergence between f and f_n.

5.4 FITNETS FOR SAMPLING

In this section, we introduce the first variant of FitNets, which focuses on improving sampling. We start by introducing the problem of sampling in network monitoring and then explain how FitNets can improve the sampling process by leveraging the data-plane scoring introduced above.

In a nutshell, this variant of FitNets (in the following simply FitNets) learns distributions in the control plane by computing kernel density estimates (KDE, see Section 5.4.2 below) from samples. As a result, FitNets can optimize for one of two objectives: *(i)*, minimize the required sampling rate for a given accuracy requirement, or *(ii)*, maximize the accuracy across multiple monitoring tasks for a given sampling rate.

5.4.1 *Overview*

Sampling on a budget We cannot sample all network traffic. Ultimately, this presents a trade-off: a high sampling rate is more accurate, but also more expensive in terms of communication and processing overhead.

The following example illustrates how this trade-off may look in practice. Consider the internet service providers A and B. Both have a set of *monitoring tasks* (see Section 5.2), and they have different sampling goals:

- A wants to minimize overhead costs. Monitoring should reach a certain level of accuracy with as little sampling as possible.
- B wants to maximize accuracy. Using a given bandwidth available for sampling, we should extract the most information.

With *accuracy*, we refer to the distance between the learned and true distribution, i.e., maximizing the accuracy means minimizing this distance. Given a set of monitoring tasks, we consider the *max-min* accuracy, i.e., we aim to maximize the lowest accuracy, minimizing the largest distance.

Despite their different goals, both A and B ultimately try to determine the optimal sampling rate across their sets of monitoring tasks. This rate is not necessarily equal for all tasks, i.e., not all subsets of traffic and features benefit equally from high sampling rates. In general, the more complex a distribution is, the higher the sampling rate needs to be. In particular, this applies to distributions with many distinct but rare values, like heavy-tailed distributions: a low sampling rare is bound to miss most of these rare values, making an accurate estimation of the tail difficult. Furthermore, increasing the sampling rate leads to diminishing returns.

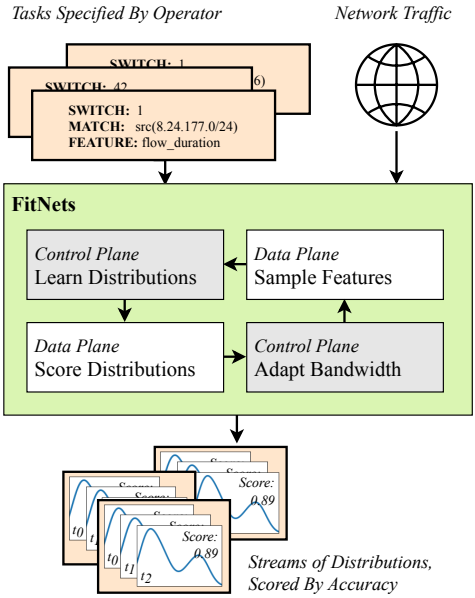


Figure 5.3: FitNets learns from traffic and adapts sampling.

Problem A priori, it is not possible to estimate how complex a particular traffic distribution is or to which degree we experience diminishing returns. To make matters worse, using samples to estimate the complexity of a distribution is also prone to error. Typically, only a few percent of network traffic can be realistically monitored, and attempting to estimate the accuracy based on only this fraction of traffic potentially misses the majority of information, in particular at the tail.

Both *A* and *B* end up either sampling too much or too little traffic: *A* resorts to over-provisioning to ensure that the minimum accuracy requirements are met, resulting in unnecessary cost, while *B* tries to manually adjust the sampling for each task, prone to miss new trends or react slowly.

Solution To optimally adapt the sampling rate for learning distributions, we build on two theoretical results: one the one hand, the asymptotic convergence of the mean-integrated-squared error (*MISE*) of the kernel-density estimate as a function of sample size has been extensively studied; and on the other hand, the *MISE* is also the distance metric associated with the *Quadratic Score*. Combined, this allows us to compute the required sampling rate from quadratic scores collected in the data plane. In the following, we explain these components in detail.

5.4.2 *Kernel density estimation, scores, and sampling rate*

FitNets estimates distributions using Kernel Density Estimation (KDE), a non-parametric method that can approximate any distribution with sufficient samples. FitNets needs to pick a sampling rate of n packets per round for each task. To adapt n, we must estimate the accuracy of a KDE fit to n samples. In the following, we explain how we can achieve this by leveraging asymptotic properties of KDE and the quadratic score.

Kernel Density Estimation Let $x_1, \ldots, x_n$ be independent and identically distributed samples from an unknown probability density f. The Kernel Density Estimate (KDE) f_n is defined as [122, p. 11]:

$$f_n(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{5.2}$$

The function K is the *kernel* and h is the *bandwidth* of the estimator. The density estimate at point x is inversely proportional to the distances to the samples x_i, with diminishing weights for larger distances. The bandwidth or "smoothing factor" determines how quickly this weight drop-off occurs. Heuristics such as Silverman's Rule or algorithms such as the Improved Sheather-Jones algorithm can estimate the optimal bandwidth from samples.

Asymptotic Convergence of KDE KDEs are asymptotically unbiased, i.e., given sufficient samples, they can estimate any distribution in the sense that the *Mean Integrated Square Error* (MISE), which is defined as

$$\mathrm{MISE}(f_n) = \mathrm{E} \int (f_n(x) - f(x))^2 \, dx \tag{5.3}$$

goes to zero for $n \to \infty$ [122, p. 23]. In particular, for an optimally chosen bandwidth, the MISE can be asymptotically approximated by

$$\mathrm{MISE}(f_n) \approx c n^{-4/5} \tag{5.4}$$

where the constant factor $c > 0$ depends mainly on the complexity of the unknown distribution in terms of its curvature $|f''|$ [122, p. 22]. It also depends on the shape of the kernel function, yet the difference in c between commonly used kernels is small [122, p. 31] and can be neglected.

The factor c is different for each measurement task and may change over time. To optimally adapt the sampling rate for each task, we need to not only estimate it but also continually adapt this estimate.

Quadratic Score The quadratic score for a sample x is defined as [36]:

$$\mathrm{QS}(f_n, x) = \underbrace{2f_n(x)}_{\textit{reward}} - \underbrace{\int f_n(\omega)^2 \, d\omega}_{\textit{regularization}} \tag{5.5}$$

This can be interpreted as a combination of a reward for predicting the correct probability of a test sample and a regularization term, penalizing overly complex estimates (independent of the test sample).

The associated distance measure for the quadratic score is the integrated square error (see Section 5.3), which is defined as:

$$d(f_n, f) = \mathrm{QS}(f, f) - \mathrm{QS}(f_n, f) = \int (f_n(x) - f(x))^2 \, dx \tag{5.6}$$

In expectation w.r.t to f, we can relate this to the convergence results above:

$$\begin{aligned} \mathrm{MISE}(f_n) &= \mathrm{E} \int (f_n(x) - f(x))^2 \, dx && (5.7)\\ &= \mathrm{E}\left[\mathrm{QS}(f, f) - \mathrm{QS}(f_n, f)\right] && (5.8)\\ &= \mathrm{QS}(f, f) - \mathrm{E}\left[\mathrm{QS}(f_n, f)\right] && (5.9)\\ &\approx c\, n^{-4/5} && (5.10) \end{aligned}$$

This equation also highlights the benefit of data plane visibility. We can only estimate $\mathrm{E}\left[\mathrm{QS}(f_n, f)\right]$ empirically. With a small sample in the control plane, this estimate is much noisier than using all traffic in the data plane, i.e., a sample that is several orders of magnitude larger.

Constrained Linear Optimization We can obtain $\mathrm{QS}(f_n, f)$ from the data plane. However, to estimate the $MISE$ (the distance between f and f_n) as a function of n, we also need to determine c and $\mathrm{QS}(f, f)$, which both rely on the unknown f. By scoring distributions for different sample sizes, we can estimate these parameters through solving a linear optimization problem.

To define this optimization problem, we first rewrite Eq. (5.10). Let $\vec{N}$ be a vector of sample sizes, and $\vec{S}$ be a vector of quadratic scores, s.t. $\vec{S}_i$ is the score of the distribution learned from a sample of size $\vec{N}_i$. Then:

$$\begin{aligned} c\vec{N}^{-4/5} &\approx \vec{1} \cdot \mathrm{QS}(f, f) - \vec{S} && (5.11)\\ \Leftrightarrow \quad 0 &\approx \vec{1} \cdot \mathrm{QS}(f, f) - \vec{S} - c\vec{N}^{-4/5} && (5.12) \end{aligned}$$

Next, we can define constraints to improve the quality of the solution. The constant c is by definition positive. Furthermore, as the quadratic score is proper, the optimal (highest) score can only be achieved for the true distribution (see Section 5.3), and thus must be at least as large as any observed score, i.e., $\mathrm{QS}(f, f) \geq \max \vec{S}$. Finally, c and $\mathrm{QS}(f, f)$ are solutions to the following constrained linear optimization problem:

$$\min_{c,\ \mathrm{QS}(f,f)} \quad \| \vec{1} \cdot \mathrm{QS}(f, f) - \vec{S} - c\vec{N}^{-4/5} \| \tag{5.13}$$

$$\text{subject to} \quad \begin{bmatrix} \mathrm{QS}(f, f) \\ c \end{bmatrix} \geq \begin{bmatrix} \max \vec{S} \\ 0 \end{bmatrix} \tag{5.14}$$

Prediction Given c and $\mathrm{QS}(f, f)$, we can predict both the estimated score for a given sample size, as well as the expected $MISE$:

$$\mathrm{QS}(n) = \mathrm{QS}(f, f) - c\, n^{-4/5} \tag{5.15}$$

$$\mathrm{MISE}(n) = c\, n^{-4/5} \tag{5.16}$$

Multiple sample sizes We can estimate densities for multiple sample sizes from a single sample by using sub-sampling, which we will explain the the following subsection. Additionally, as explained in Section 5.3, we can easily score multiple densities in parallel in the data plane.

Generalization With mild assumptions, we can generalize Eq. (5.13) to a nonlinear optimization problem to more complex models and scoring rules.

Let $S(\cdot)$ be a proper scoring rule with associated distance metric $d(f_n, f)$. Assume that d asymptotically decreases with increasing sample size, albeit with an unknown rate r. Then, we can write:

$$d(f_n, f) = \mathrm{S}(f, f) - \mathrm{E}\left[\mathrm{S}(f_n, f)\right] \approx c\, n^{-r} \tag{5.17}$$

where $c, r > 0$. As above, we can formulate this as an optimization problem:

$$\min_{c,\ r, S(ff)} \quad \|\mathrm{S}(f, f) - c\vec{N}^{-r} - \vec{S}\| \tag{5.18}$$

$$\text{subject to} \quad c, r > 0 \tag{5.19}$$

$$\mathrm{S}(f, f) > \max \vec{S} \tag{5.20}$$

This optimization problem is non-linear and is more complex to solve than Eq. (5.13), but demonstrates how our approach can be generalized to other models (i.e., FitNets is not limited to KDE) and other proper scoring rules.

5.4.3 *Control Plane*

Figure 5.4 shows the control plane of FitNets, which consists of: *(i)* estimating densities from samples; *(ii)* processing the scores received from the data plane; and *(iii)* adapting the sampling rate based on the scores. Most of the computations can be parallelized, allowing FitNets to scale to a large number of tasks and samples. Density computation and score processing can even be distributed across multiple machines, as tasks are independent of each other. Only adaptation requires the processed scores and is thus not fully independent, but it is also the computationally cheapest step.

Density Estimation To estimate the parameters c and $\mathrm{QS}(f, f)$ that are required to predict the expected $MISE$, we need to estimate densities for multiple sample sizes. Repeatedly drawing samples, estimating distributions, and scoring them over multiple rounds is impractical and delays FitNets' ability to return updated distributions quickly. Fortunately, we can score *multiple* distributions in the data plane in parallel with relatively low overhead (see Section 5.3). Thus, if we can estimate multiple distributions, we can score them in a single round and avoid unnecessary delays.

To estimate multiple distributions with minimal overhead, we transform a single set of samples per measurement task into multiple sets of samples by sub-sampling. As there are diminishing returns for increasing sample sizes, we choose a geometric progression of sample sizes by default: n, $n/2$, $n/3$, $n/4$, etc. Different sub-sample sizes can be configured by operators if desired. In general, the more unique sample sizes we use, the more accurate the estimation of the optimal sampling rate will be, but the more overhead we incur for estimation and data plane lookups.

For each sample size, we estimate a density using KDE and pre-compute the data-plane scores (see Section 5.3). We record the selected sample sizes, as they are required to solve the optimization problem in the next round.

Score Processing In parallel to estimating new densities, we process the data-plane results for the previous round. We only compute the reward term of the quadratic score in the data plane, which we combine with the pre-computed regularization term to compute the final score. Together with the recorded sample sizes, we can solve the linear optimization problem defined in Eq. (5.13), giving us the parameters c and $\mathrm{QS}(f, f)$ for each monitoring task, which are required for the adaptation step.

Adaptation With c and $\mathrm{QS}(f, f)$ parameters, we can predict the expected score and *MISE* for a given sample size (Eq. (5.16)). How we adapt the sampling rate depends on the operator's objective.

For *Resource Minimization*, we can directly solve either Eq. (5.15) or Eq. (5.16) for n, the required sample size to reach a specific accuracy, which can be given in terms of a target score or *MISE*.

For *Accuracy Maximization*, we need to optimally distribute the available sampling budget across all tasks to maximize the lowest expected *MISE*. To avoid having to search all possible sample size combinations across all tasks, we invert the problem: for any given accuracy, we predict the sample size according to Eq. (5.16) and return the sum of sample sizes across tasks. Subsequently, finding the accuracy that uses all available resources, but not more, is a *scalar* root-finding problem that can be solved quickly.

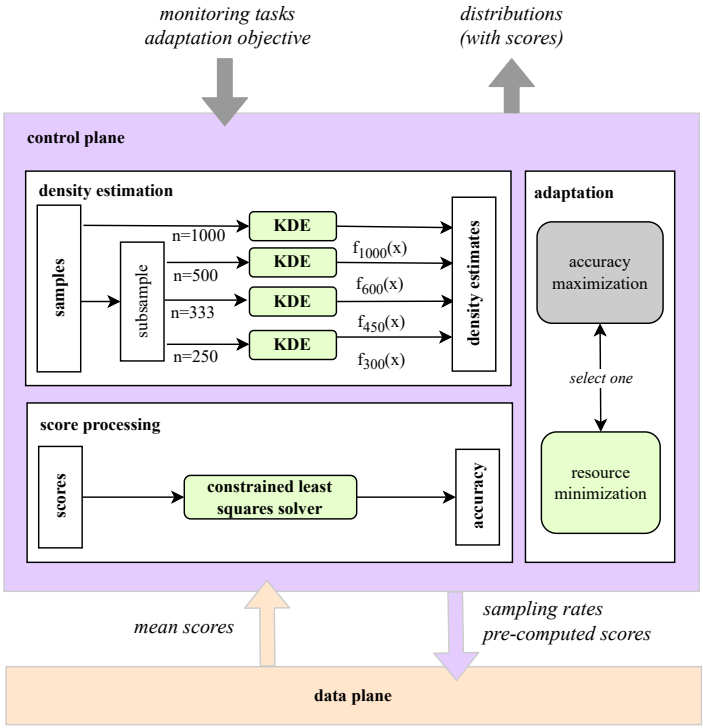


Figure 5.4: In the control plane we estimate densities and solve a constrained linear optimization problem to optimize the sampling rate from scores.

5.5 FITNETS FOR SEARCHING

In this section, we introduce the second variant of FitNets, which searches for the optimal distribution parameters while side-stepping sampling entirely. We first motivate sampling-free learning and then discuss the challenges of using this approach for anomaly detection as well as how to solve them.

In a nutshell, this variant of FitNets (in the following again simply FitNets) learns by searching a parameter space for the best-fitting distribution. We use Bayesian optimization to guide this search with data-plane scores.

Sampling-free learning Even if optimized by FitNets (see Section 5.4), sampling is inherently limited: even if we optimize the sampling budget allocation, the overall rate is finite. We are bound to miss some traffic, in particular rare events. Thus, as the control plane can only learn from samples, learning the tails of distributions is challenging. On the other hand, data plane scores can be computed for all traffic. Also, by using scores like the logarithmic score, which is particularly sensitive to rare events, we can ensure that the score sufficiently represents the distribution tail. This raises the question: *can we use data plane scores for learning without sampling?*

The answer is yes, but it is not straightforward: without sampling, we need to search a space of parameters to find the distribution that best fits the data, i.e., achieves the highest scores. Naively, we could try to evaluate all possible distributions, but this can quickly become intractable with a large parameter space. Similarly, randomly testing distribution candidates may require a prohibitively large number of trials to find a good fit.

We can do better by *learning* from data-plane scores to guide our search. The key tool enabling this is Bayesian optimization, which we explain below.

Bayesian optimization Bayesian Optimization (BO) maximizes a black-box objective function from potentially noisy evaluations. To learn distributions, maximizing a proper scoring rule is a natural choice, as this minimizes the distance to the true distribution (see Section 5.3). Thus, learning distributions via BO can be stated as $\arg\max_{p \in \mathcal{P}} S(f_p)$, where f_p is a distribution parametrized by a set of parameters p from the space $\mathcal{P}$, and S is the data-plane score of f_p, which is indeed a noisy objective.[3]

The parameter space $\mathcal{P}$ is defined by the operator and may range from a single distribution to a mixture of different distribution families. Theoretically, BO can handle any parameter space, but in practice, the complexity of the space may limit the effectiveness of the search (see Section 5.7).

3 The data-plane score is less noisy than a control-plane score as the effective sample size is significantly larger (see Section 5.4), but it is noisy nonetheless.

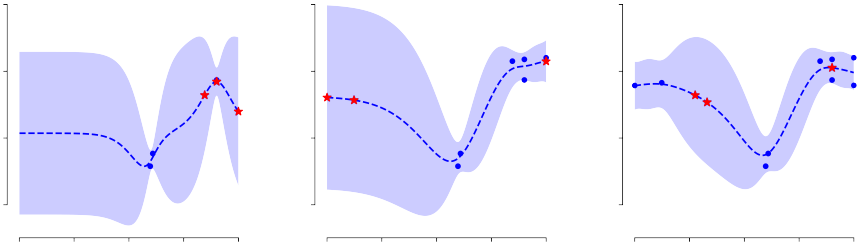


Figure 5.5: Abstract representation of Bayesian optimization: BO uses a surrogate model to estimate a black-box function, here a Gaussian process. The x-axis represents the parameter space, and the y-axis the function value. The dashed line shows the best current estimate, and the shaded area shows the uncertainty of the Gaussian process. Blue dots are past function evaluations, and red stars are proposed candidates to evaluate. We can observe how Bayesian optimization explores the parameter space over three rounds. Using multiple candidates allows for exploiting the current best parameters while still exploring alternatives.

Bayesian optimization (BO) explores the parameter space with the help of a tractable surrogate model, which is used to approximate the objective function. Evaluating the surrogate model is much faster than evaluating the objective function, allowing BO to quickly search the parameter space of the surrogate model. After generating a set of candidate parameters, the objective function is evaluated, the result is used to update the surrogate model, and the process is repeated. The most common surrogate model is a Gaussian process, i.e., a distribution over functions that naturally captures uncertainty and/or noise in the objective function.

To fully leverage data-plane scoring and make the most of observed traffic, we should evaluate multiple candidate parameters in parallel (see Section 5.3). However, BO only returns a single candidate in each round. This forces a decision between exploration and exploitation: do we stick to the current best distribution or explore other promising parameters? A common approach to generate multiple candidates is to *lie* to BO: after generating the first candidate, we update the surrogate model with a fake low function value for this candidate, motivating BO to explore. Then, we generate another candidate, and so on. Once all candidates are generated, they are evaluated and the fake values are replaced with real results.

Control plane BO integrates into FitNets easily. In each round, we generate multiple distribution parameter candidates. For each candidate distribution, we precompute scores (see Section 5.3) and send them to the data plane. In the next round, we update BO with the data-plane scores and repeat.

Anomaly detection We can use data-plane scoring to detect anomalies in the network. As anomalies are by definition rare, this particularly benefits from being able to learn distributions that cover rare events: these distributions allow us to distinguish rare but expected events from truly anomalous traffic, such as attacks. However, this comes at a cost: if we naively learn from the data plane, we risk poisoning the learning process with attack traffic, counteracting our ability to detect it. In the following, we first describe how we can detect anomalies, and then how this approach can also be used to protect the learning process from attack traffic.

The key insight is that anomalies follow a different distribution than benign traffic: anomalous traffic scores differently than benign traffic. Thus, we calculate a confidence interval on the score of expected traffic for the current best distribution estimate and classify any traffic scoring outside of this interval as anomalous. By deferring this until n packets arrive, we can reduce false positives on single packets and only alert on anomalous sequences of size n. The exact number of packets can be chosen by the operator to balance false positives with detection delay.

We can use this result as an additional step in the data-plane pipeline (Figure 5.2) to avoid poisoning the learning process with attack traffic by scoring in two stages. The first stage keeps scores until n packets arrive. If the aggregated score of the first stage is within the confidence interval, it is added to the second stage, keeping the final scores. As a result, we can detect anomalies while keeping them separate from the learning process.

Fundamentally, this approach is built on the assumption that benign shifts in traffic are gradual, while anomalies like attacks are sudden. Gradual shifts are persistent but small, and thus more likely to fall into the confidence interval. As a result, they are not classified as anomalies but rather contribute to the learning process. The width of the confidence interval represents the cutoff between benign shifts and anomalies.

5.6 EVALUATION

In this section, we evaluate data-plane scoring and FitNets. We evaluate the sampling variant of FitNets with a focus on resource usage and accuracy, and the search variant of FitNets with a focus on anomaly detection.

5.6.1 *Evaluation data*

To evaluate data-plane scoring and sampling adaptation, we extracted representative distributions for five packet and flowlet features from real-world network traces. We analyzed one hour of CAIDA backbone traces [123] and collected data for packet size, inter-arrival time, packets per flowlet, flowlet size, and flowlet duration. We use an inter-packet gap of 500*ms* for flowlets, i.e., after 500*ms* of inactivity, a new flowlet starts. Figure 5.6 shows the distributions per feature, which we use as ground truth. The distributions exhibit a broad range of shapes, e.g., the size distribution is bimodal, showing a large peak for very small packets, and a smaller peak for packets at maximum frame size, with other sizes mixed in between. While the used distributions are *representative*, we want to stress that the distributions represent just a moment in time and space. In different networks and at different times, the same features are likely distributed differently.

However, we cannot evaluate anomaly detection with this data, as we do not know if any subset of traffic is benign or malicious. Thus, we evaluate our anomaly detection approach with synthetic data. We simulate a traffic feature that follows a Zipf distribution with $a = 1.3$ and attack traffic that follows another Zipf distribution with $a = 1.1$. This represents a feature like destination ports per source: For benign sources, the distribution is narrow, as most send packets to widely used ports. For attack sources, e.g., a port-scanning attacker, the distribution is naturally wider.

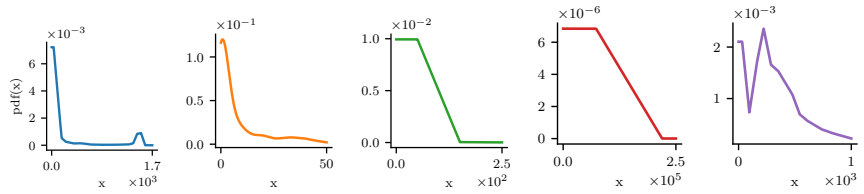


Figure 5.6: Ground truth distributions. Left to right: packet size [bytes], inter-arrival time [ms], flowlet size [packets], flowlet size [bytes], flowlet duration [ms].

5.6.2 *Data-plane scoring*

Figure 5.7 shows the relative error score between the empirical and true score along with different ratios of test sample size to training sample size to evaluate the benefit of data-plane scoring. We repeatedly train KDEs with sample sizes from 100 to 10000 and score with sample sizes from 10 to 1000000. We show the average error and standard deviation over 100 repetitions for each feature, train-, and test sample size.

Test samples at about as large as the training sample lead to reliable ($< 10\%$ error) scores. For smaller fractions, the error varies strongly (up to 40%). For increasingly large test samples, the error approaches 0% for all features. This highlights the benefits of data-plane scoring: FitNets does not need to reserve a small fraction of the already limited sampled data and can instead use all traffic as one large test set.

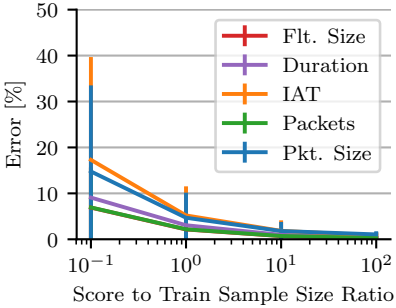


Figure 5.7: Without a large test sample, scores are inaccurate.

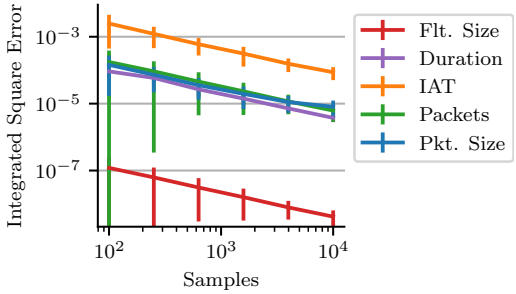


Figure 5.8: We need to adapt sampling rates as different features are not equally complex.

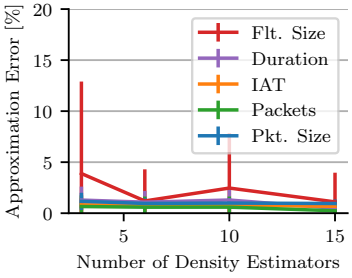


Figure 5.9: A few KDEs suffice to approximate c and $\mathrm{QS}(f, f)$.

5.6.3 *FitNets for sampling*

Density Estimation Figure 5.8 shows the $MISE$ between the estimated and true distribution for increasing training sample size (on a log scale). We again show the mean and standard deviation over 100 repetitions each.

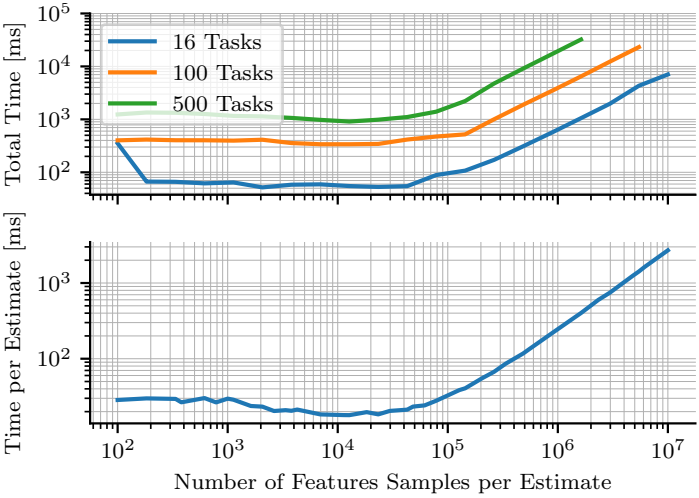
We can see that the error decreases with increasing sample size, and this experiment highlights the need for adapting sampling rates to the complexity of monitoring tasks. For example, reaching an error below 10^{-4} requires 100 k samples for the inter-arrival time and less than 1000 samples for the flowlet duration. Assigning a fixed equal sampling rate to both features would either waste resources for the flowlet duration or not provide enough samples for the inter-arrival time.

Score Processing Figure 5.9 show the approximation error of the constrained linear optimization used by FitNets to approximate the unknown parameters c and $\mathrm{QS}(f, f)$; mean and std. over 100 repetitions each.

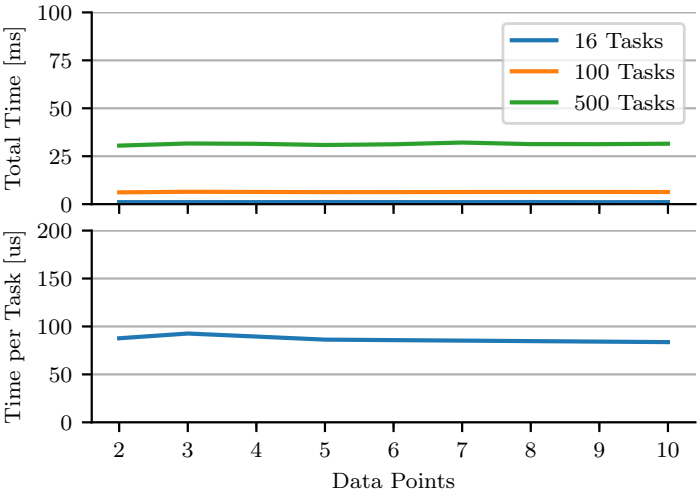
FitNets approximates the unknown parameters c and $\mathrm{QS}(f, f)$ well, even if only a small number of distinct sample sizes are evaluated. The approximation error is below 15% for 3 distinct sample sizes and below 5% for 6 sample sizes. Our experiments indicate that using a larger number of distinct sample sizes with our subsampling strategy does not substantially improve the performance further. We suspect that we observe diminishing returns as we need to split the training sample into subsamples of decreasing size, which decreases the quality of the additional estimators.

Benchmarks Figure 5.10a shows that computing probability densities is fast, requiring roughly 40 milliseconds to estimate a distribution for up to $100k$ samples. For larger sample sizes, the time increases linearly and requires about 1 second for $3.9M$ samples. Extending estimation to multiple densities can be efficiently parallelized, as all estimations are independent. On our (16 core) test machine, estimating distributions for 16 densities with $100k$ samples each requires about 100 milliseconds (there is some overhead from initializing worker processes and distributing samples). In one second, FitNets can process up to $1.5M$ samples each for 16 densities in parallel, and a bit short of $100k$ samples each for 500 densities, which corresponds to $24M$ and $50M$ samples in total, highlighting that FitNets can process millions of samples for up to hundreds of densities in parallel.

Figure 5.10b shows that solving the optimization problem is even faster and *requires less than* 10 *milliseconds per optimization*. On our test machine, we can solve the problem for 500 tasks in parallel in less than 50 ms.



(a) Density estimation time is almost constant up to 10 k samples, as other operations dominate the runtime. For larger sample sizes, the time increases linearly.



(b) Score processing is fast, even with several data points (distinct sample sizes) per monitoring task.

Figure 5.10: FitNets is efficient and parallelizable.

Adaptation Finally, we evaluate the two adaptation objectives of FitNets, resource minimization and accuracy maximization in two small case studies. For easier interpretation, we show accuracy with a normalized score: an accuracy of 0.9 means the observed score was 10% below the optimal score.

Figure 5.11 showsresource minimization with an accuracy objective of 0.98. FitNets sets the sampling rates appropriately, matching the target accuracy within ±2% in general, and ±4% in the worst case.

Figure 5.12 show the results for accuracy maximization with fixed bandwidth, comparing the performance of FitNets with a static allocation. On average, FitNets achieves higher accuracy with the same total resources.

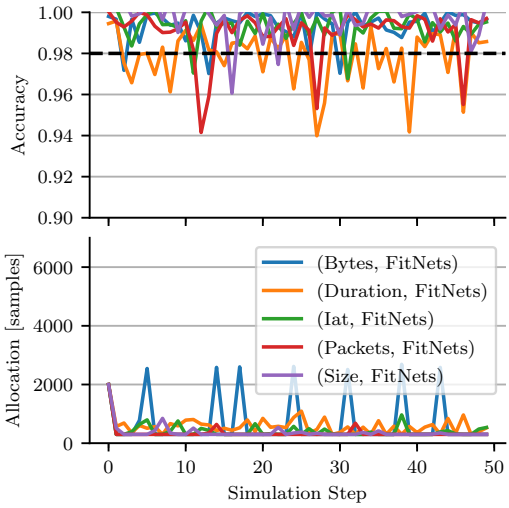


Figure 5.11: FitNets minimizes the sampling rate to meet a target accuracy.

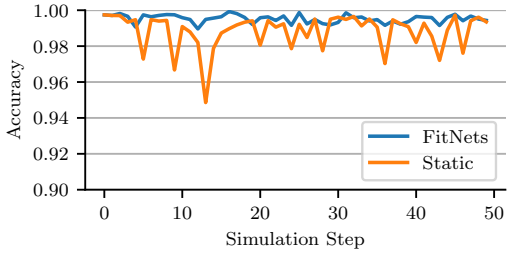


Figure 5.12: FitNets maximizes accuracy for a given sampling rate.

.

5.6.4 *FitNets for search*

We evaluate search-based FitNets with the synthetic data described above. For all tests, we generate three distribution candidates with BO each round: the current best estimate, which is also used for anomaly detection, and two others for exploring the parameter space. For anomaly detection, we update our classification every 10th packet (see Section 5.5 for details).

Anomaly detection The ROC curve in Figure 5.13 shows the true and false positive rate for confidence interval (CI) thresholds from 0–100%.

Our approach works well if we can learn an accurate distribution of benign traffic. The ROC curve in Figure 5.13 shows the true and false positive rate for confidence interval (CI) thresholds from 0–100%. With a 95% CI, we detect 94% of anomalies with 5% false positives (green line, top left). We also show how performance degrades if we learn from attack traffic, highlighting the importance of detecting and excluding this traffic when learned based on data-plane scores.

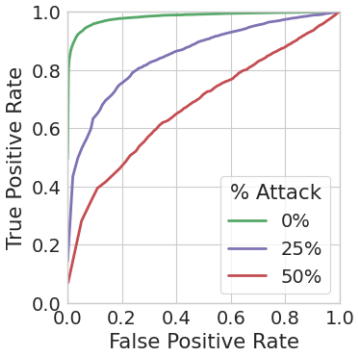


Figure 5.13: Anomaly detection degrades significantly if the distribution is learned from data containing a high fraction of attack packets.

Case study Figure 5.14 shows a small case study comparing sampling-based learning with search-based learning for anomaly detection. For search-based learning, we include a both naive version that learns from all data plane traffic as well as a FitNets, which excludes anomalies from the learning process. We bootstrap learning with 20 iterations of 1000 benign packets each, followed by 10 iterations of 1000 benign and attack packets each. For sampling-based learning, we sample a rather high rate of 10% of traffic for learning. We show the mean and standard deviation over three repetitions.

Overall, we confirm that sampling-based learning struggles to detect anomalies. As sampling misses many rare events, we struggle to learn the tail-heavy Zip distribution. However, there is a silver lining: we also sample only a fraction of attack traffic, limiting the attack's impact on the model. We do not observe a particular degradation after the attack begins.

By combining scoring with anomaly detection, we avoid poisoning the model with attack traffic. The naive search-based learning experiences significant drops in the true positive rate and increases in the false positive rate after the attack begins. On the other hand, FitNets remains stable and is not affected. Note that BO re-estimates the parameters based on the current and past measurements at each iteration. Thus, the naive search does not immediately degrade once attack traffic is introduced but rather becomes unreliable over time as the attack slowly poisons the estimates and makes the surrogate model less useful.

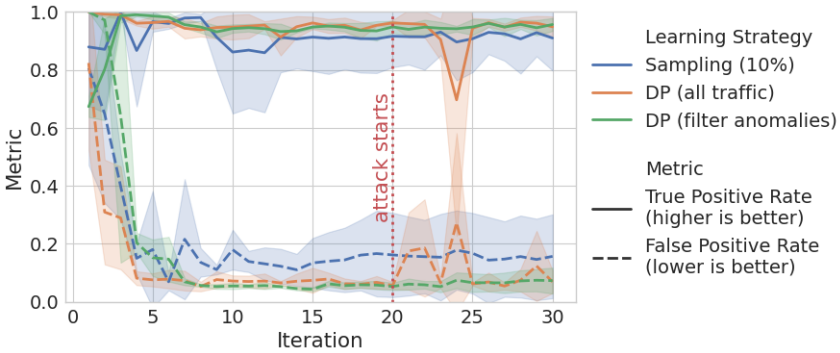


Figure 5.14: The search variant of FitNets can efficiently learn a distribution from data-plane scores (DP). Combined with anomaly detection, we avoid poisoning with attack traffic. Sampling detects anomalies worse as it misses the distribution tail. Lines show mean over three runs, shaded areas show standard deviation.

5.7 DISCUSSION & FUTURE RESEARCH

We have shown how to leverage programmable network devices to improve the observability of network traffic. To achieve this, we had to overcome one critical limitation: we cannot simply sample all traffic. Without a strategy to extract information from the network efficiently, the network may be able to observe all traffic, but cannot share its insights.

We introduced FitNets, a system for monitoring and anomaly detection that implements a closed loop between the data- and control plane. FitNets learns in the control plane, leveraging its computational resources, while scoring the learned distributions in the data plane, leveraging its observability. However, FitNets is only a prototype towards a more general approach to leverage programmable network devices for learning tasks, and we outline open questions and future directions below.

Sampling and searching We introduced two distinct variants of FitNets. Both build on data-plane scoring, but implement different learning strategies in the control plane. The first variant learns distributions from sampled traffic, and the second variant learns by using Bayesian optimization to search for the best model parameters. We have shown the limitations of sampling-based approaches and highlighted how the search-based FitNets can perform better for heavy-tailed distributions, delivering better results in an anomaly detection case study. But search with Bayesian optimization has limitations as well. In particular, it loses effectiveness with an increasing number of parameters and learning complex models such as deep neural networks is simply infeasible with this approach.

Future work might explore hybrid approaches that combine the best of both worlds. Leveraging an available sampling budget to allow training more complex models in the control plane, while still using data plane search, e.g., to tune hyperparameters for better tail performance.

Large models We have shown that FitNets can learn distributions from traffic traces and detect anomalies with short delays. We focused on learning distributions to make the most out of the loop between the data- and control plane: they can be learned efficiently, so we can adapt to changing traffic patterns quickly. However, these models are limited in their complexity and predictive power. More complex models, such as deep neural networks, may make better predictions, but also come with much larger computational requirements for the data- and control plane. In the control plane, they take much larger to train, and in the data plane, they are much more difficult to evaluate. While one-dimensional distributions are easy to translate to a

lookup table, deep neural networks are not. Can we still benefit from the tight control loop while using larger and more complex models?

In the data plane, there have already been steps to implement more complex models like random forests on programmable network devices [7]. In the control plane, incremental updates to the model might be a promising direction to explore. Similar to the concept of cheap finetuning of large pre-trained models discussed in Chapter 3, we might be able to update a large model incrementally, guided by the data plane feedback, keeping it up to date while still benefiting from fast data-plane feedback.

Applications Aside from considering different models, future work could explore how different learning tasks could benefit from FitNets. In this chapter, we have considered general monitoring and anomaly detection, but many other applications are limited by a lack of network observability.

How can applications like video streaming best leverage a system like FitNets? Are distributions the best way to provide additional information, or would a more specialized interface be more beneficial? Also, considering the diversity of applications, how can we use the limited data-plane resources most effectively to support a wide range of applications?

Adaptation Finally, we have not thoroughly discussed how to adapt to changing traffic patterns over time. For the sampling variant of FitNets, we relied on the latest set of samples to learn updated distributions. For the search variant of FitNets, we assumed that benign traffic undergoes gradual changes, while anomalies are sharp deviations, allowing us to safely update the surrogate model over time. However, as we have discussed in Chapter 2, changes in traffic patterns over time can be much more complex.

Can we make the detection of changes over time a first-class citizen of FitNets? There are many interesting directions to explore. For example, we might combine FitNets with Memento: instead of computing the density of in-memory samples based on collected samples, we might be able to leverage data-plane scoring to evaluate how common or rare parts of the sample space really are. This might allow us to update Memento's memory more efficiently. We might also leverage this information as an alternative way to adapt sampling rates, e.g., by increasing the sampling rate in regions of the sample space that are currently underrepresented.

6

CONCLUSION AND OUTLOOK

6.1 SUMMARY

In this dissertation, we have explored how ML and programmable networks can aid network traffic modeling. We have analyzed and addressed challenges of ML-based network traffic models and presented strategies for adaptive network traffic modeling, i.e., strategies to *(i) adapt over time*, keeping up with the network's evolution; and *(ii) adapt over space*, transferring knowledge between different networks and tasks. Moreover, we investigated how to improve network traffic modeling in general by studying the fundamental challenge of *learning latent network state*. Finally, we explored the potential of programmable networks for *learning in the data plane*.

Learning over time In Chapter 2, we have addressed the challenge of continual learning or adapting models over time. We have shown that common approaches of using more—more data, more complex models, more training—are insufficient to ensure good performance, in particular at the long tail of network traffic. We presented Memento, which meets this challenge through a *smarter* sample selection. Memento estimates the sample space density, allowing it to retain samples in low-density regions while readily replacing samples in high-density regions. This allows Memento to adapt to changes in common traffic patterns quickly while remembering rare ones that are crucial for tail performance. Finally, by monitoring sample space changes, Memento allows us to assess rationally when to retrain.

Learning over space In Chapter 3, we have presented a vision for generalizing network traffic models to new environments and tasks. We explored Transformer architectures, which have enabled previously unimaginable generalization in domains such as natural language processing, and proposed a Network Traffic Transformer (NTT) that learns network dynamics from packet traces. In particular, we advocate for *pre-training*: we can pre-train models on massive datasets and fine-tune them to new tasks and environments with comparatively little time and data. We showed promising initial results on simulation data, demonstrating that NTT effectively learns traffic patterns and that the pre-training knowledge generalizes.

Learning latent variables In Chapter 4, we have investigated the fundamental problem of network traffic modeling: predicting the latent network state from observed traffic metrics. We first formalized this problem as a nonlinear dynamical system and identified important components necessary to learn its behavior. Then, we analyzed four different systems (Fugu, NTT, CausalSim, and Veritas) in-depth to identify how they implement each component. Based on this comparison, we evaluated over 12 years of video streaming data, considering both download time prediction and counterfactual reasoning. We made the following key observations: *(i)* ML models outperform hand-crafted models; *(ii)* a high-dimensional latent space helps generalization; *(iii)* a fine-grained history is important for both tail performance and generalization; and *(iv)* a high degree of persistent excitation, i.e., data diversity, is important. However, we note that this is an empirical study, and results may vary in other network environments.

Learning in the data plane Finally, in Chapter 5, we explored how we can leverage programmable networks to learn distributions of traffic features such as packet sizes or interarrival times. One of the key benefits of programmable networks is the ability to programmatically process *all* traffic in the network, providing unmatched observability of traffic. We introduced FitNets, a system for monitoring and anomaly detection that implements a closed loop between the data- and control plane. It learns distributions in the control plane, leveraging computational resources, and evaluates them with easy-to-compute scoring rules in the data plane, leveraging the data-plane observability. We presented two variants of FitNets: *(i)* a *sampling-based* variant that uses data-plane scores to optimize the sampling rate for different subsets and features, matching their complexity. And *(ii)*, a *search-based* variant that uses data-plane scores to guide Bayesian optimization in the control plane. While the latter approach is restricted to parametrized distributions, it can effectively learn heavy-tailed distributions that are challenging to fit from samples.

6.2 FUTURE RESEARCH DIRECTIONS

The data In Chapters 2 and 4, we have seen the importance of good training data for ML-based network traffic models. We believe that this is a critical direction for future research in ML-based traffic modeling. There are several dimensions to this problem:

First of all, we must address *data diversity*. Chapter 4 has shown that more diverse data is particularly important for generalization and tail performance. In different parts of the world, different applications are popular, different network topologies are prevalent, and so on. With more diverse data, we can make models useful to a broader range of users.

However, Chapter 2 showed that data is often biased towards common traffic patterns, making it challenging to learn important rare patterns. Simply collecting more leads to collecting mostly irrelevant data. Future research could investigate methods to collect more *useful* data. This is a key principle of *active learning*, but given the complexity of network traffic, it can be challenging to decide *what* data to collect. Methods similar to Memento could help to discover important traffic patterns.

Finally, Chapters 3 and 4 have shown the benefits of combining different types of data, such as using packet traces to predict application-level download times. To build on these ideas, we need more datasets that include *multi-modal* data. Currently, most available datasets are either one or the other: packet-traces without application metrics or application traces without packet data. There is an opportunity for future data collection to be more holistic, enabling models to learn the connection between low-level network traffic and high-level application performance.

The models In no other domain have multi-modal models advanced as quickly as in natural language processing and computer vision. Where initial Transformer models were "only" state-of-the-art translators, modern models can summarize videos, generate images from text, or even interact with other applications. In Chapter 3, we have discussed generalization for network traffic models with Transformers, yet still limited ourselves to *networking tasks*. There has been some research on using language models for network traffic by converting traffic into text-like tokens, but this is a crude approximation. Future research could investigate integrating network traffic into large multi-modal models as a first-class citizen, which could unlock applications such as explaining packet traces to human operators or generating customized traffic patterns based on text descriptions.

While impressive, modern language models are also incredibly large. But even with more modest models such as the ones discussed throughout this dissertation, we are facing a challenge of scale. These models can be applied to applications such as video streaming, where a new chunk is requested every few seconds. However, applying them to individual packets would necessitate significantly more computational power. Past research has proposed ML models for congestion control, routing, and other packet-level tasks, but these models are not used online. Instead, they are used offline to learn non-ML algorithms which are then deployed to make packet-level decisions. Instead of focussing on how to train *better* models, future research could explore how to train *more efficient* ones that could be applied to individual packets. This presents an interesting trade-off between model complexity, prediction accuracy, and resource requirements.

The network Finally, we close this dissertation where it began: by looking at the network. We have mostly discussed how we can use ML models to *interpret* network traffic, but networks can also play a more active role.

Future research may consider additional signals the network could provide to ML models. In Chapter 5, we have shown how data-plane scores can improve *learning* in the data plane, and future research may investigate what signals the network could provide to allow applications to estimate a more accurate latent network state. This is an active area of research for non-ML algorithms, e.g., HPCC [124] provides additional signals for hand-crafted network models. Future research could analyze whether ML-based applications also benefit from these signals, or whether there are even more complex or noisy signals thay may need ML models to fully leverage them.

Finally, we have already discussed research that uses ML to optimize network algorithms and the potential to apply future ML models to individual packets. Yet, we may go further. With the advance of programmable networking, it is not implausible that future network devices may support ML-capable hardware, and we may become able to deploy ML models directly in the data plane. Fundamentally, these models are not different from the ones we discussed in Chapter 4: they still predict the latent network state from observed traffic—they just have access to far more traffic. Thus, continued research in this direction may ultimately benefit in-network ML, reaching an even greater scale than we have discussed in this dissertation.

A

APPENDIX

A.1 MEMENTO: SUPPLEMENTAL REAL-WORLD RESULTS

Recurring patterns Figure A.1 illustrates that the Puffer traffic does contain patterns that recur at the tail. Each line in this plot corresponds to one batch in the memory assembled by Memento after three weeks of sample selection. After these three weeks, we do not change the memory anymore and only observe whether the patterns are observed again in the following weeks. The lines show the density of batches with respect to the daily samples over the next 200 days; this captures "how similar the batches in memory are to the traffic of the current day." It shows that some tail batches (the lines with the lowest densities) are sometimes more represented in the daily traffic for a couple of days, then fade away again.

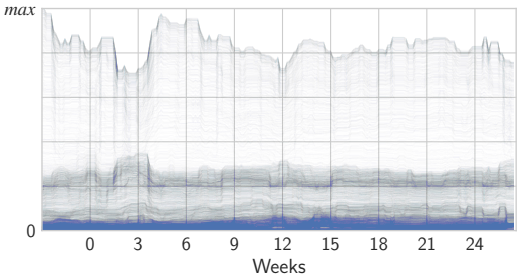
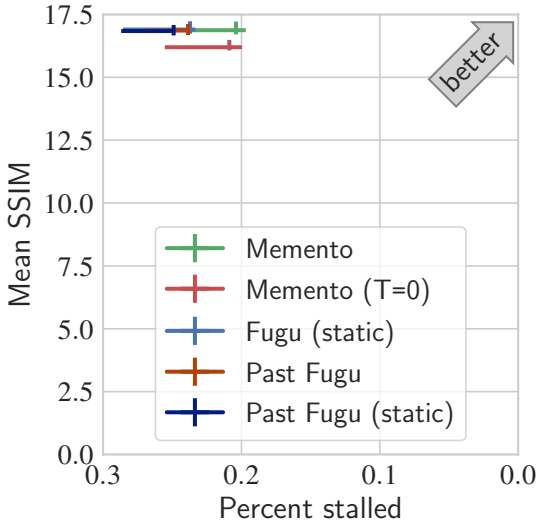
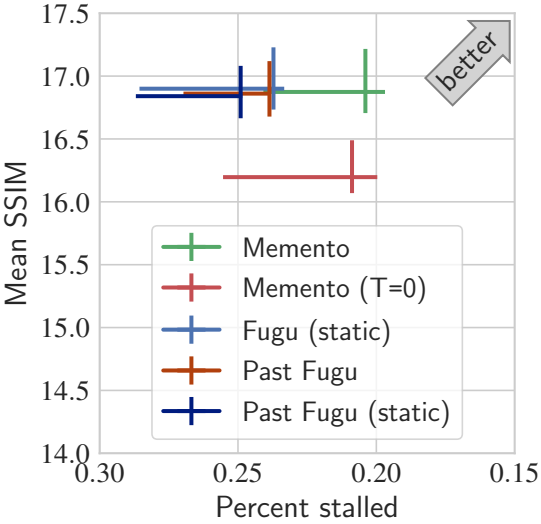


Figure A.1: Coverage of batches in the daily samples

Aggregate performance Figure A.2 shows the aggregated SSIM and percent of stream-time spent stalled on a 2D grid in the style of the Puffer [16] publication and website [25]. Concretely, we show two sets of ABRs. First Memento (default and $T = 0$) and Fugu_{Feb}, aggregated since the latest version of Memento was deployed on September 19th, 2022 until February 13th, the cutoff for our current evaluation. We cannot aggregate over the deployment duration Fugu as it was discontinued on October 5th. Thus, we additionally include a second set consisting of Fugu and Fugu_{Feb}, aggregated from 2020 until Fugu was discontinued.



(a) Image quality and fraction of stream-time spent stalled.



(b) Zoomed in.

Figure A.2: Aggregate performance. Mean and bootstrapped 90% confidence intervals from the deployment of the current version of Memento on September 19th, 2022 until February 13th, 2023. ABRs annotated with 'past' indicate past data from April 9th, 2020 until October 5th, 2022, when Fugu was discontinued.

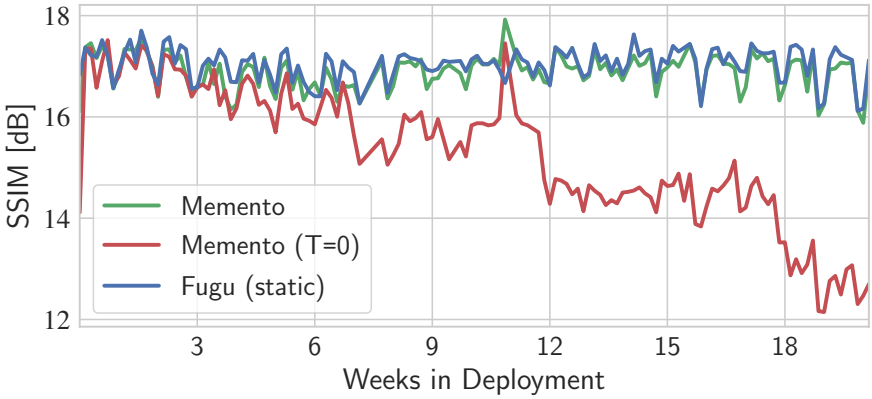


Figure A.3: Long-term degradation of Memento ($T = 0$) in an earlier deployment.

Past degradation of Memento ($T = 0$) over time In a previous deployment, we observed Memento ($T = 0$) to degrade over time, as shown in Figure A.3. Without forgetting, it kept accumulating noise and retraining. At first, the Puffer control loop around the TTP was able to compensate for this degradation, but as the model became too bad, it failed. Over time, the Image quality degraded by over 30 %.

However, we later discovered an issue in this deployment that prevented Memento from using the deployed models' prediction, effectively disabling BBDR. After fixing this issue, we observed that the performance of Memento ($T = 0$) recovered, highlighting the benefits of considering both prediction and output space. Nevertheless, we are still observing signs of noise accumulation like stronger coverage increase and frequent retraining.

Prediction score improvements compared to the past Figure A.4 shows Figure 2.6a overlayed with the prediction score improvements of Fugu compared to Fugu_{Feb} in the past. These curves are not directly comparable, as they come from different periods of time and the underlying data may have shifted. However, we can see that daily retraining with random samples did not consistently improve the TTP tail score; it even worsened the tail score for 20 % of days. This may explain why daily retraining yields significantly smaller tail QoE improvements than retraining with samples selected by Memento, which consistently improves the tail predictions.

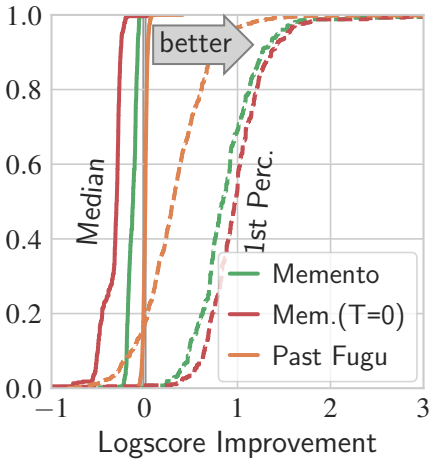


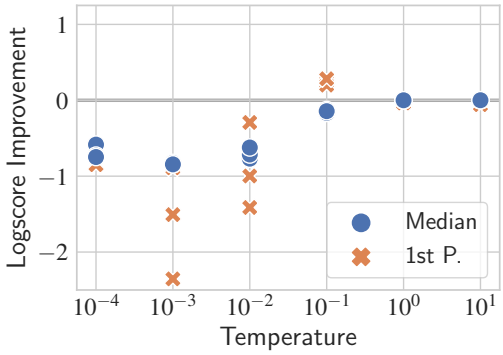
Figure A.4: Retraining daily did not consistently improve the tail prediction.

Alternative selection metrics Figure A.5 shows further alternative sample selection metrics in addition to loss (Section 2.4.4).

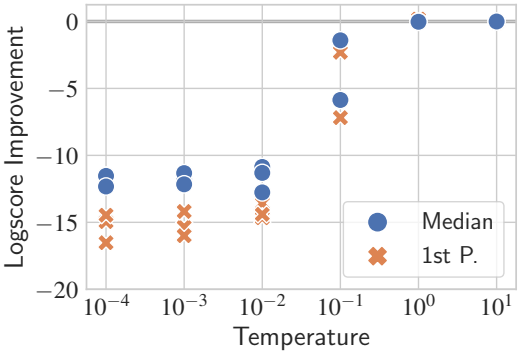
Figure A.5a shows the results for model confidence, i.e., the probability of the predicted transmission time bin (Puffer separates the transmit time into 21 bins ranging from 0.125 s to 10 s). With this metric, Memento prefers to discard samples with high confidence to keep 'difficult' samples with low confidence. For high temperatures, performance improvements are small. For low temperatures, we observe strong variation between runs, with more performance degradation than improvement. In summary, this selection metric is unreliable and fails to consistently improve performance.

Figure A.5b shows the results for label counts, a simplified version of density that is often used for classification data. We use the transmission time bin of each sample as the label. With this metric, Memento prefers to discard samples with high label counts to keep 'rare' samples with low label counts. We observe this approach to drastically reduce performance. In the Puffer environment, the majority of samples are assigned to the lowest transmit time bins. Using label counts alone removes too many of these samples and the model forgets common patterns, similar to loss.

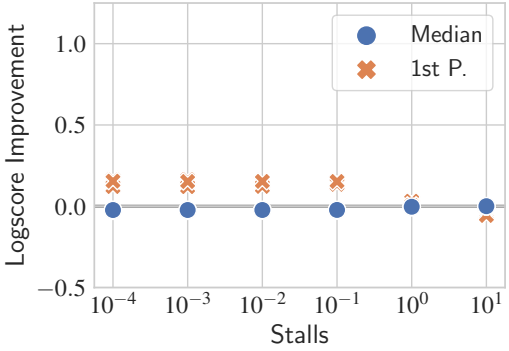
Finally, Figure A.5c shows the results for stalled sessions. With this metric, Memento prefers to discard samples if they belong to a session that did not stall to keep samples from sessions that did stall. We observe consistent improvements, but they are small. It does not provide a fine-grained enough selection to significantly improve tail performance.



(a) Confidence



(b) Label counts



(c) Stalled sessions

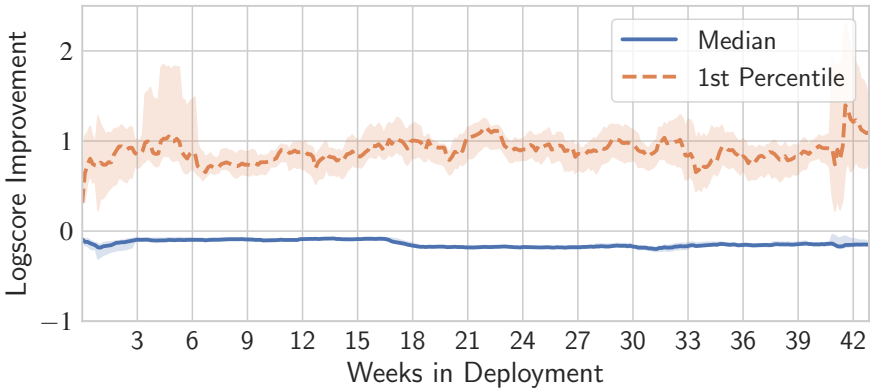
Figure A.5: Additional alternative selection metrics.

Timeseries (real-world) Figure A.6 shows the prediction improvements over Fugu_{Feb} for Memento and Memento ($T = 0$). Figure A.7 shows QoE results per day over time for Memento, Memento ($T = 0$) and for Fugu_{Feb} since September 19th. Figure A.8 shows the same results, but the mean and bootstrapped 90% CI over a two-week sliding window.

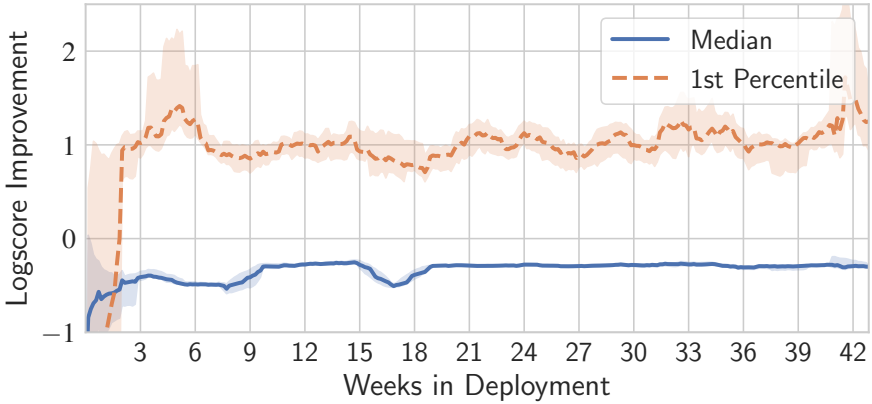
Increased memory capacity Figure A.9 shows the results for Memento compared to a random memory with the same capacity (both 1 M) and to a random memory with a double capacity (2 M). The random memory is set up like Fugu, selecting samples randomly over the last two weeks. We can see that simply increasing the capacity fails to address dataset imbalance, and performance is virtually identical at double the training effort.

Timeseries (replay) Figure A.10 shows the evolution of each benchmark experiment over the whole 6-month duration. Figure A.11 and Figure A.12 show the same time series for the experiments with alternative selection metrics, and for combining Memento with Matchmaker and JTT, respectively.

For temperature-related benchmarks, we observe that a high temperature (uniformly random selection) performs slightly worse at the tail than Fugu. This may be because Fugu keeps samples from the past 14 days, while a uniformly random selection phases samples out more quickly.

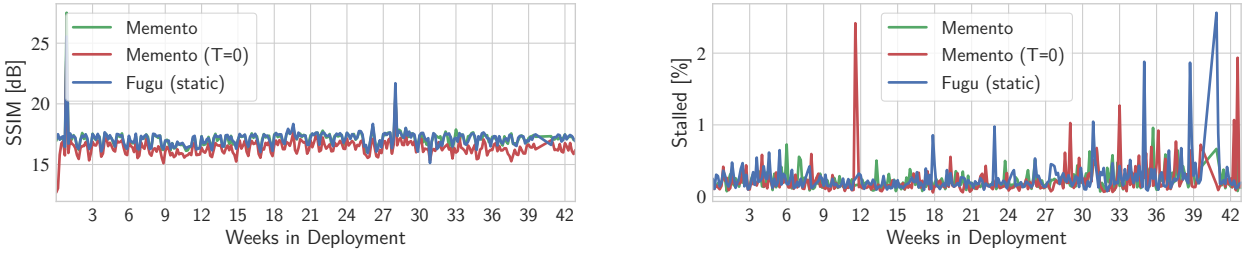


(a) Memento.



(b) Memento ($T = 0$).

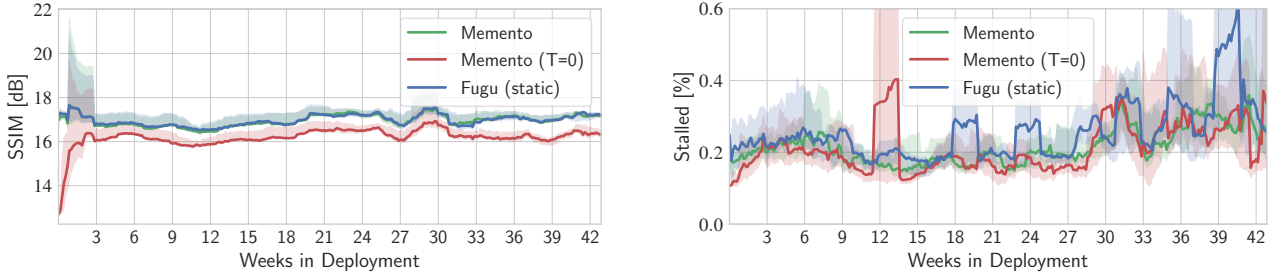
Figure A.6: Logscore improvements compared to Fugu_{Feb}.



(a) Image quality, measured in SSIM.

(b) Percent of stream-time spent stalled.

Figure A.7: Evolution of QoE metrics over time. Absolute Values for each ABR.



(a) Image quality, measured in SSIM.

(b) Percent of stream-time spent stalled.

Figure A.8: Evolution of QoE metrics over time. Absolute Values for each ABR. Mean and bootstrapped 90% confidence interval over two-week sliding windows.

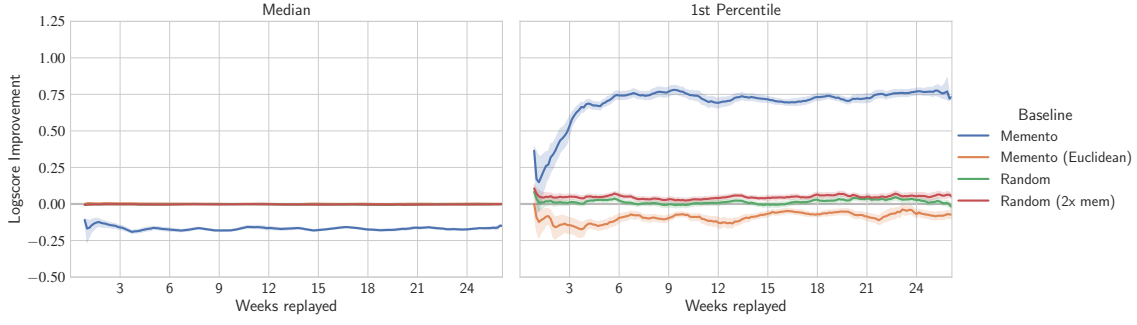
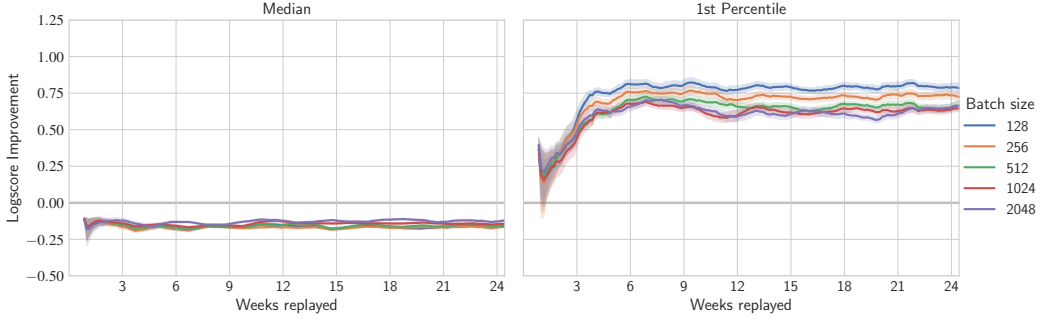
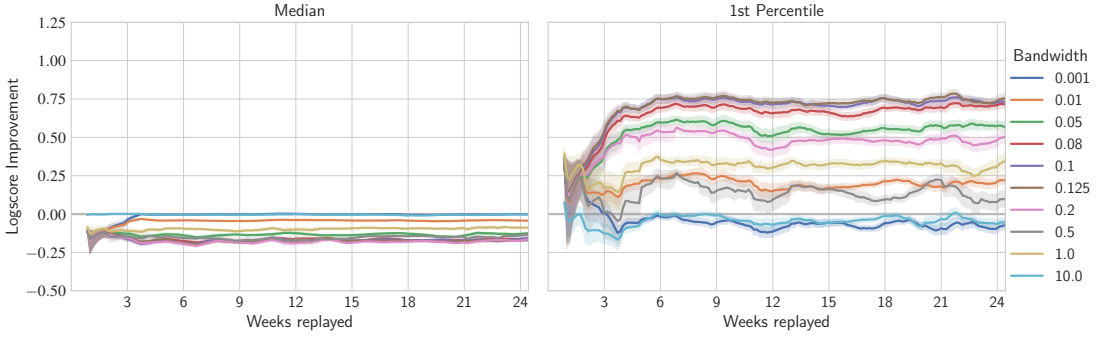


Figure A.9: Logscore improvements compared to Fugu, median and tail (1st percentile) improvements per day. We compute the mean over all three 6-month replays, and also show bootstrapped 90% confidence intervals over two-week windows. Here: Memento compare to random sampling and to using Euclidean distances instead of JSD.

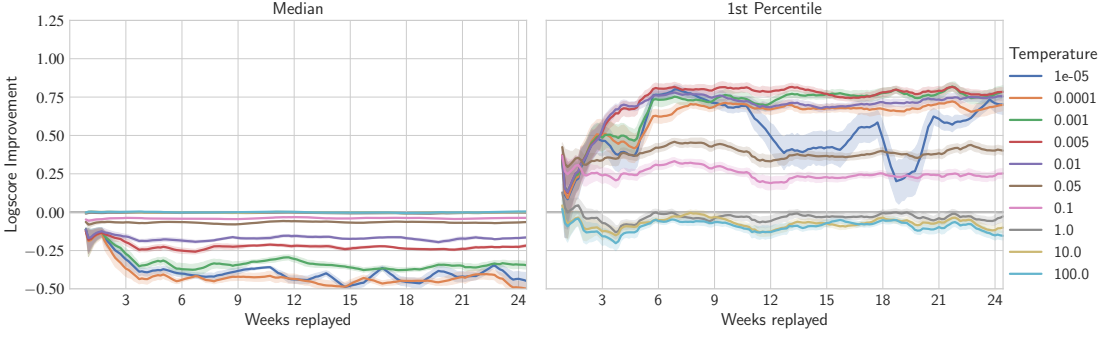


(a) Batch size.

Figure A.10: Logscore improvements compared to Fugu (see Figure A.9) for all selection metric experiments (1/2).

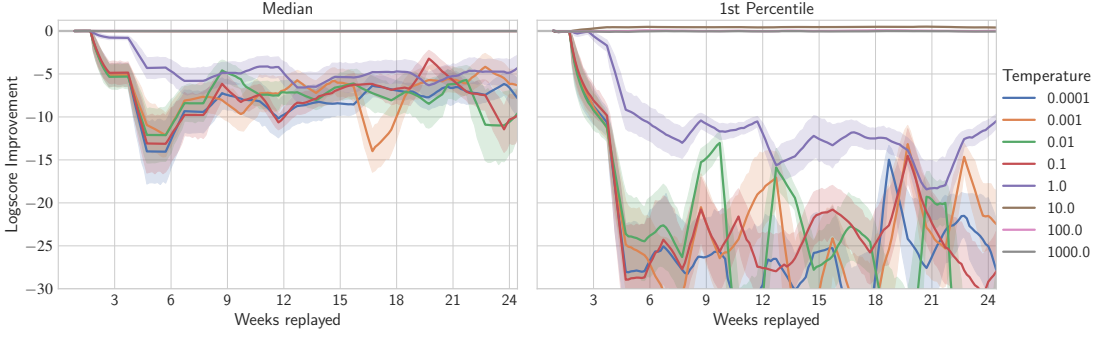


(b) Bandwidth

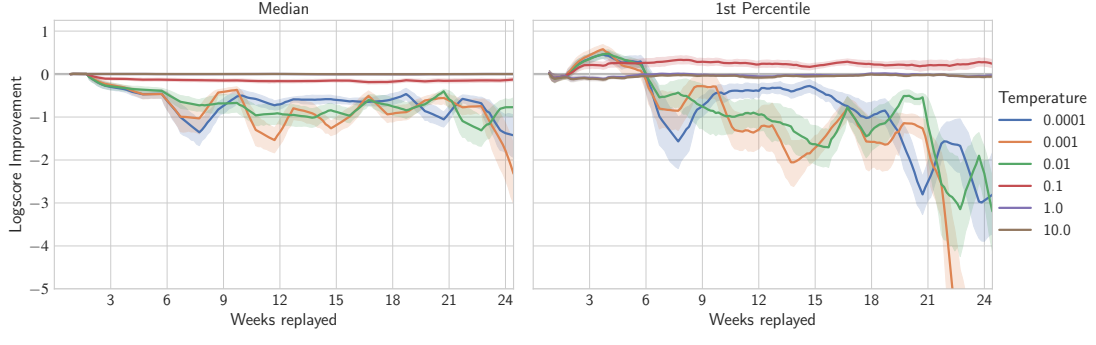


(c) Temperature

Figure A.10: Logscore improvements compared to Fugu (see Figure A.9) for all selection metric experiments (2/2).

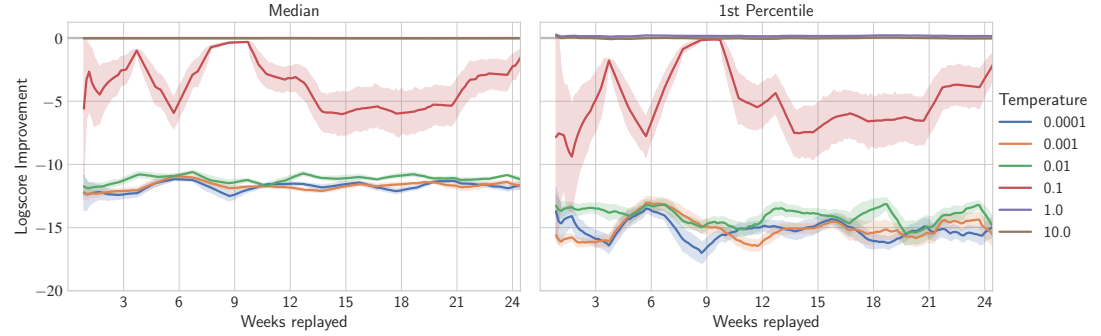


(a) Loss.

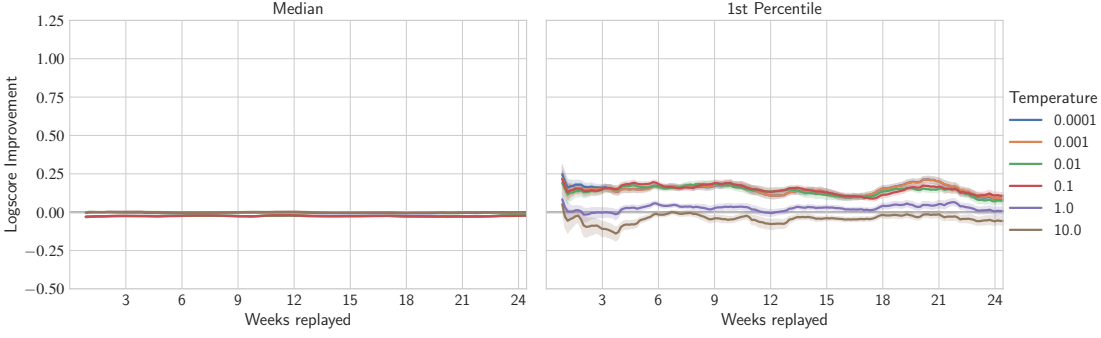


(b) Confidence.

Figure A.11: Logscore improvements compared to Fugu (see Figure A.9) for all selection metric experiments (1/2).

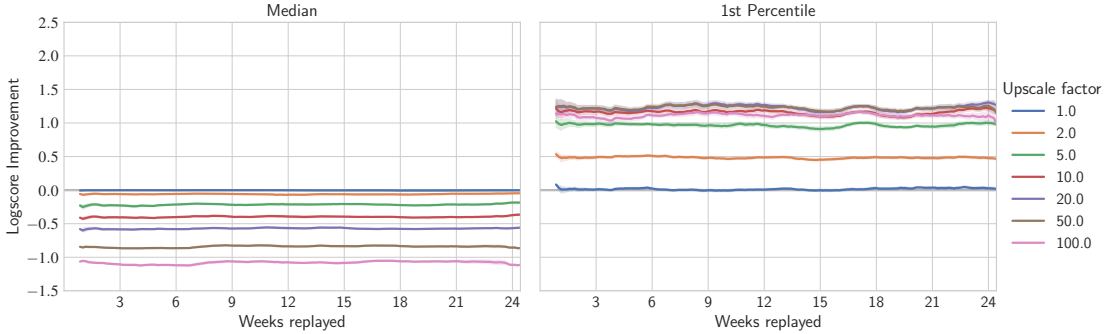


(c) Label counts

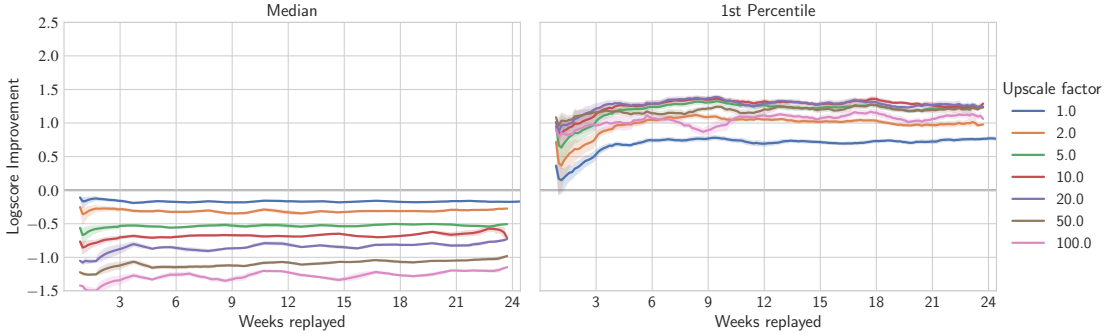


(d) Stalled sessions

Figure A.11: Logscore improvements compared to Fugu (see Figure A.9) for all selection metric experiments (2/2).

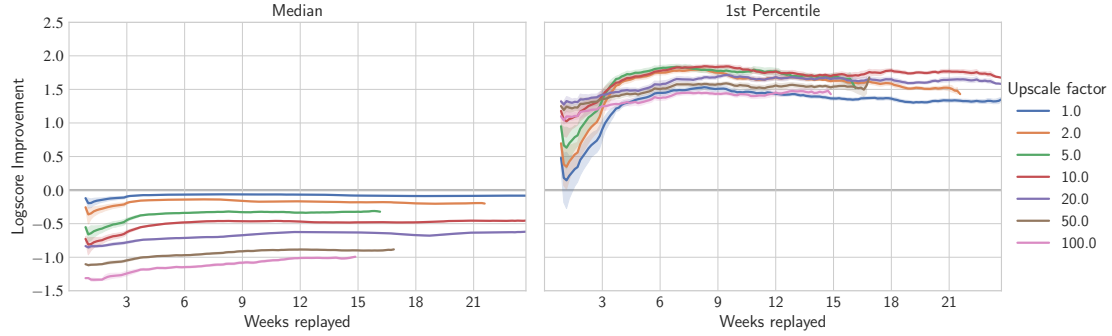


(a) JTT with random sample selection.

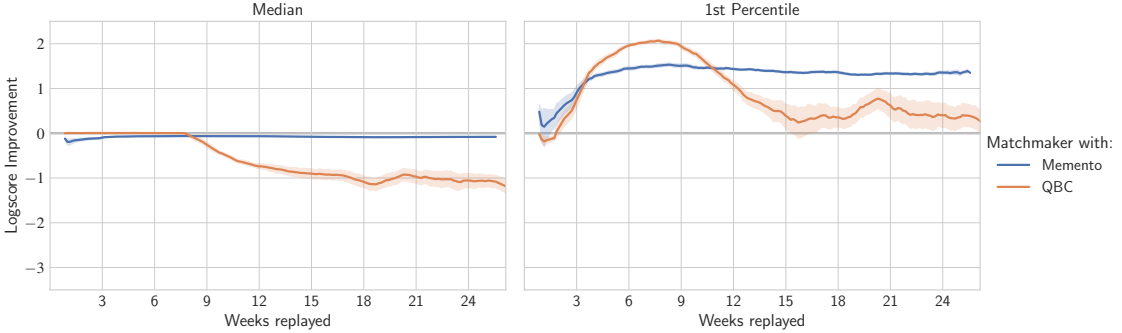


(b) JTT with Memento's sample selection.

Figure A.12: Logscore improvements compared to Fugu (see Figure A.9) for all combination experiments (1/2).



(a) JTT with Memento's sample selection and optimal Matchmaker predictions.



(b) Memento compared to Query-By-Committee (QBC) sample selection, both with optimal Matchmaker predictions.

Figure A.13: Logscore improvements compared to Fugu (see Figure A.9) for all combination experiments (2/2).

A.2 MEMENTO: SUPPLEMENTAL SYNTHETIC RESULTS

Simulation setup Figure A.14 illustrates the setup we used for the evaluation of Memento in simulation.

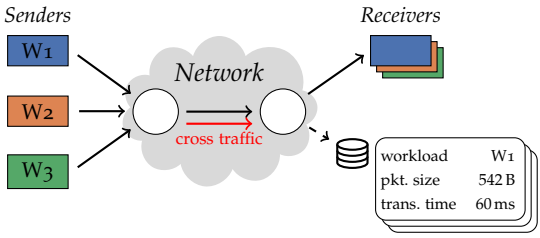


Figure A.14: Simulation setup.

Model Architecture We use the following parameters for the classification and regression models used in our simulation experiments. We use supervised training and select the number of layers, neurons, and training parameters via hyperparameter optimization [125].

We have implemented both models using Keras [126]. For all layers, we use batch normalization [127] and ReLU activation [128]. We train both networks with the Adam optimizer [129] using the default decay parameters $\beta_1 = 0.9,\ \beta_2 = 0.999$. We train for up 200 epochs with early stopping.

	Model	
Parameter	Classification	Regression
Hidden Layers	3	4
Hidden Units	512	362
Learning Batchsize	512	512
Learning Rate	5.91×10^{-5}	0.382

Table A.1: Model parameters.

Workload distributions Figure A.15 shows the message size distribution for the workloads we use, published by the HOMA project [40]): Facebook web server (**W1**), DC-TCP (**W2**), and Facebook Hadoop (**W3**). Messages are generated with Poisson-distributed inter-arrival times.

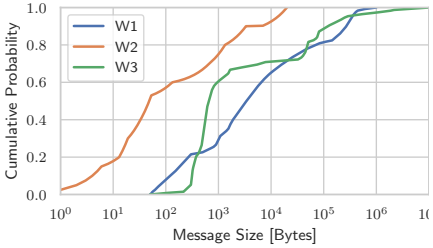


Figure A.15: Message size distributions [40].

Samples in memory Table A.2 shows the number of samples in memory after the last iteration of Section 2.5.2 and Section 2.5.3. Memento retains all workloads in memory, whereas other approaches either forget existing or fail to pick up new samples.

	W1	W2	W3
Rare Patterns (Section 2.5.2)			
Memento	5696	9053	5219
Random	252	19634	114
LARS	3159	14231	2610
FIFO	0	20000	0
Incremental Learning (Section 2.5.3)			
Memento	5376	5888	8704
Random	1266	17836	898
LARS	9540	9557	903
FIFO	0	0	20000

Table A.2: In-memory samples after each experiment.

A.3 LATENT LEARNING: MODEL HYPERPARAMETERS

In this section, we provide the model hyperparameters for Section 4.2.

Model We use linear layers with GeLU activation, layer normalization, and dropout (0.1) in between layers. All layers use 128 hidden units. The linear model uses four layers, while the Encoder model uses two layers to extract the latent state from history, and two additional layers for each prediction head. For variants restricting the dimension of latent space, we add a learnable linear projection layer after the latent extraction layers.

For the Transformer model, we use a default Transformer encoder with 8 heads and 128 hidden units. We trained both a shallow transformer with only a single transformer layer (containing attention and two linear layers) as well as a deeper transformer with 4 transformer layers. Additionally, we add a single embedding layer to upscale inputs to 128 dimensions.

For packet aggregation layers, we either use the hierarchical aggregation as described in Chapter 3 or a convolutional layer with kernel size 32, stride 16, and a padding size of 8.

Training Both both download time and buffer level prediction we use the Huber loss. We use an Adam optimizer with a learning rate of 4×10^{-3}, $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 1 \times 10^{-5}$, and a weight decay of 0.1. We use a batch size of 1024, except for the Transformer models, where we use a batch size of 64 due to memory constraints.

With the Puffer dataset, we train for 3 epochs with 10 M samples per epoch. With our preliminary packet dataset, we train for 15 epochs using 365 k samples per epoch.

CausalSim We retrain CausalSim with its default hyperparameters and $\kappa \in \{0, 0.1, 1.0, 10.0\}$ as the weight of the adversarial loss, corresponding to the labels none, weak, medium, and strong respectively. For the case of $\kappa = 0$, we train CausalSim without the adversarial loss, i.e., we completely remove the adversarial decoder.

REFERENCES

[1] Alexander Dietmüller, Romain Jacob, and Laurent Vanbever. "On Sample Selection for Continual Learning: a Video Streaming Case Study". In: *Computer Communication Review* 54.2 (2024).

[2] Alexander Dietmüller, Siddhant Ray, Romain Jacob, and Laurent Vanbever. "A New Hope for Network Model Generalization". In: *HotNets '22: Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. New York, NY: Association for Computing Machinery, 2022, 152.

[3] Alexander Dietmüller, Georgia Fragkouli, and Laurent Vanbever. "Poster: Learning distributions to detect anomalies using all the network traffic". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. , New York, NY, USA, Association for Computing Machinery, 2023, 1108.

[4] Alexander Dietmüller and Laurent Vanbever. "FitNets: An Adaptive Framework to Learn Accurate Traffic Distributions". In: *arXiv preprint arXiv:2405.10931* (2024).

[5] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. "P2GO: P4 Profile-Guided Optimizations". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. New York , NY: Association for Computing Machinery, 2020, 146.

[6] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. "SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues". In: *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. Ed. by Ranjita Bhagwan and George Porter. Berkeley, CA: USENIX Association, 2020, 59.

[7] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. *pForest: In-Network Inference with Random Forests*. `http://arxiv.org/abs/1909.05680`. 2019.

[8] Michael Hauben and R Hauben. "Behind the Net: The Untold History of the ARPANET and Computer Science". In: *Netizens: on the history and impact of Usenet and the internet* (2006).

[9] *Internet Use in 2024*. `https://datareportal.com/reports/digital-2024-deep-dive-the-state-of-internet-adoption`. 2024.

[10] Sandvine. *Global Internet Phenomena*. https://www.sandvine.com/phenomena.

[11] V. Jacobson. "Congestion Avoidance and Control". In: *ACM SIGCOMM Computer Communication Review* 18.4 (1988). https://dl.acm.org/doi/10.1145/52325.52356, 314.

[12] Kanon Sasaki, Masato Hanai, Kouto Miyazawa, Aki Kobayashi, Naoki Oda, and Saneyasu Yamaguchi. "TCP Fairness Among Modern TCP Congestion Control Algorithms Including TCP BBR". In: *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. https://ieeexplore.ieee.org/abstract/document/8549505. 2018, 1.

[13] *Bufferbloat – Communications of the ACM*. https://cacm.acm.org/practice/bufferbloat/. 2012.

[14] Richelle Adams. "Active Queue Management: A Survey". In: *IEEE Communications Surveys & Tutorials* 15.3 (2013). https://ieeexplore.ieee.org/abstract/document/6329367, 1425.

[15] Ferenc Fejes, Gergő Gombos, Sándor Laki, and Szilveszter Nádas. "Who Will Save the Internet from the Congestion Control Revolution?" In: *Proceedings of the 2019 Workshop on Buffer Sizing*. BS '19. https://doi.org/10.1145/3375235.3375242. Palo Alto, CA, USA: Association for Computing Machinery, 2019, 1.

[16] 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)*. https://www.usenix.org/conference/nsdi20/presentation/yan. 2020, 495.

[17] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. https://dl.acm.org/doi/10.1145/3098822.3098842. Los Angeles CA USA: ACM, 2017, 183.

[18] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. "A Quality-of-Experience Index for Streaming Video". In: *IEEE Journal of Selected Topics in Signal Processing* 11.1 (2017), 154.

[19] S. Shunmuga Krishnan and Ramesh K. Sitaraman. "Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs". In: *IEEE/ACM Transactions on Networking* 21.6 (2013), 2001.

[20] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. "Continual Lifelong Learning with Neural Networks: A Review". In: *Neural Networks* 113 (2019). http://www.sciencedirect.com/science/article/pii/S0893608019300231, 54.

[21] Pietro Buzzega, Matteo Boschini, Angelo Porrello, and Simone Calderara. "Rethinking Experience Replay: A Bag of Tricks for Continual Learning". In: *arXiv:2010.05595 [cs, stat]* (2020). http://arxiv.org/abs/2010.05595.

[22] David Isele and Akansel Cosgun. "Selective Experience Replay for Lifelong Learning". In: *arXiv:1802.10269 [cs]* (2018). http://arxiv.org/abs/1802.10269.

[23] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. "Experience Replay for Continual Learning". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett. http://papers.nips.cc/paper/8327-experience-replay-for-continual-learning.pdf. Curran Associates, Inc., 2019, 350.

[24] *YouTube Statistics 2024 [Users by Country + Demographics]*. https://www.globalmediainsight.com/blog/youtube-users-statistics/. 2024.

[25] *Puffer*. https://puffer.stanford.edu/.

[26] H. S. Seung, M. Opper, and H. Sompolinsky. "Query by Committee". In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. https://dl.acm.org/doi/10.1145/130385.130417. New York, NY, USA: Association for Computing Machinery, 1992, 287.

[27] Hidetoshi Shimodaira. "Improving Predictive Inference under Covariate Shift by Weighting the Log-Likelihood Function". In: *Journal of Statistical Planning and Inference* 90.2 (2000). https://www.sciencedirect.com/science/article/pii/S0378375800001154, 227.

[28] Gerhard Widmer and Miroslav Kubat. "Learning in the Presence of Concept Drift and Hidden Contexts". In: *Machine Learning* 23.1 (1996). https://doi.org/10.1023/A:1018046501280, 69.

[29] Zachary C. Lipton, Yu-Xiang Wang, and Alex Smola. "Detecting and Correcting for Label Shift with Black Box Predictors". In: *arXiv:1802.03916 [cs, stat]* (2018). http://arxiv.org/abs/1802.03916.

[30] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. "Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett. http://papers.nips.cc/paper/8420-failing-loudly-an-empirical-study-of-methods-for-detecting-dataset-shift.pdf. Curran Associates, Inc., 2019, 1396.

[31] D. M. Endres and J. E. Schindelin. "A New Metric for Probability Distributions". In: *IEEE Transactions on Information Theory* 49.7 (2003), 1858.

[32] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951). https://www.jstor.org/stable/2236703, 79.

[33] Emanuel Parzen. "On Estimation of a Probability Density Function and Mode". In: *The Annals of Mathematical Statistics* 33.3 (1962). https://www.jstor.org/stable/2237880, 1065.

[34] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. *On Calibration of Modern Neural Networks*. http://arxiv.org/abs/1706.04599. 2017.

[35] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), 600.

[36] Tilmann Gneiting and Adrian E. Raftery. "Strictly Proper Scoring Rules, Prediction, and Estimation". In: *Journal of the American Statistical Association* 102.477 (2007). https://doi.org/10.1198/016214506000001437, 359.

[37] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. "Matchmaker: Data Drift Mitigation in Machine Learning for Large-Scale Systems". In: *Proceedings of Machine Learning and Systems* 4 (2022). https://proceedings.mlsys.org/paper/2022/hash/1c383cd30b7c298ab50293adfecb7b18-Abstract.html, 77.

[38] Evan Z. Liu, Behzad Haghgoo, Annie S. Chen, Aditi Raghunathan, Pang Wei Koh, Shiori Sagawa, Percy Liang, and Chelsea Finn. "Just Train Twice: Improving Group Robustness without Training Group Information". In: *Proceedings of the 38th International Conference on Machine Learning*. https://proceedings.mlr.press/v139/liu21f.html. PMLR, 2021, 6781.

[39] George F. Riley and Thomas R. Henderson. "The Ns-3 Network Simulator". In: *Modeling and Tools for Network Simulation*. Ed. by Klaus Wehrle, Mesut Güneş, and James Gross. https://doi.org/10.1007/978-3-642-12331-3_2. Berlin, Heidelberg: Springer, 2010, 15.

[40] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. https://doi.org/10.1145/3230543.3230564. Budapest, Hungary: Association for Computing Machinery, 2018, 221.

[41] Sebastian Farquhar and Yarin Gal. "Towards Robust Evaluations of Continual Learning". In: *arXiv:1805.09733 [cs, stat]* (2019). http://arxiv.org/abs/1805.09733.

[42] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. "Learning in Nonstationary Environments: A Survey". In: *IEEE Computational Intelligence Magazine* 10.4 (2015), 12.

[43] James L. McClelland, Bruce L. McNaughton, and Randall C. O'Reilly. "Why There Are Complementary Learning Systems in the Hippocampus and Neocortex: Insights from the Successes and Failures of Connectionist Models of Learning and Memory". In: *Psychological Review* 102.3 (1995), 419.

[44] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. "Overcoming Catastrophic Forgetting in Neural Networks". In: *Proceedings of the National Academy of Sciences* 114.13 (2017). https://www.pnas.org/content/114/13/3521, 3521.

[45] Friedemann Zenke, Ben Poole, and Surya Ganguli. "Continual Learning Through Synaptic Intelligence". In: *International Conference on Machine Learning*. http://proceedings.mlr.press/v70/zenke17a.html. PMLR, 2017, 3987.

[46] Roger Ratcliff. "Connectionist Models of Recognition Memory: Constraints Imposed by Learning and Forgetting Functions". In: *Psychological Review* 97.2 (1990), 285.

[47] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška. "Improved Deep Reinforcement Learning for Robotics through Distribution-Based Experience Retention". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, 3947.

[48] Behnaz Arzani, Kevin Hsieh, and Haoxian Chen. "Interpretable Feedback for AutoML and a Proposal for Domain-customized AutoML for Networking". In: *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. HotNets '21. https://doi.org/10.1145/3484266.3487373. New York, NY, USA: Association for Computing Machinery, 2021, 53.

[49] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A Survey of the State-of-the-Art". In: *Knowledge-Based Systems* 212 (2021). https://www.sciencedirect.com/science/article/pii/S0950705120307516, 106622.

[50] Fabian Hinder, André Artelt, and Barbara Hammer. "Towards Non-Parametric Drift Detection via Dynamic Adapting Window Independence Drift Detection (DAWIDD)". In: *Proceedings of the 37th International Conference on Machine Learning*. https://proceedings.mlr.press/v119/hinder20a.html. PMLR, 2020, 4249.

[51] Ashraf Tahmasbi, Ellango Jothimurugesan, Srikanta Tirthapura, and Phillip B. Gibbons. "DriftSurf: Stable-State / Reactive-State Learning under Concept Drift". In: *Proceedings of the 38th International Conference on Machine Learning*. https://proceedings.mlr.press/v139/tahmasbi21a.html. PMLR, 2021, 10054.

[52] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. "Genet: Automatic Curriculum Generation for Learning Adaptation in Networking". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22. https://doi.org/10.1145/3544216.3544243. New York, NY, USA: Association for Computing Machinery, 2022, 397.

[53] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. "Neural Adaptive Video Streaming with Pensieve". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. https://doi.org/10.1145/3098822.3098843. Los Angeles, CA, USA: Association for Computing Machinery, 2017, 197.

[54] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (2016). https://ojs.aaai.org/index.php/AAAI/article/view/10295.

[55] Martin Riedmiller. "Neural Fitted Q Iteration–First Experiences with a Data Efficient Neural Reinforcement Learning Method". In: *European Conference on Machine Learning*. 2005, 317.

[56] Rahaf Aljundi, Eugene Belilovsky, Tinne Tuytelaars, Laurent Charlin, Massimo Caccia, Min Lin, and Lucas Page-Caccia. "Online Continual Learning with Maximal Interfered Retrieval". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett. http://papers.nips.cc/paper/9357-online-continual-learning-with-maximal-interfered-retrieval.pdf. Curran Associates, Inc., 2019, 11849.

[57] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv:1810.04805 [cs]* (2019). http://arxiv.org/abs/1810.04805.

[58] OpenAI et al. *GPT-4 Technical Report*. http://arxiv.org/abs/2303.08774. 2023.

[59] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao,

Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. http://arxiv.org/abs/2307.09288. 2023.

[60] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. "A Survey on Vision Transformer". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), 1.

[61] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. "Transformers in Vision: A Survey". In: *ACM Computing Surveys* (2021). https://doi.org/10.1145/3505244.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. Curran Associates, Inc., 2017.

[63] Alexander Dietmüller, Siddhant Ray, Romain Jacob, and Laurent Vanbever. "A New Hope for Network Model Generalization". In: *HotNets '22: Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. New York, NY: Association for Computing Machinery / ETHZ and ETHZ, 2022, 152.

[64] Franck Le, Mudhakar Srivatsa, Raghu Ganti, and Vyas Sekar. "Rethinking Data-Driven Networking with Foundation Models: Challenges and Opportunities". In: *Proceedings of the 21st ACM Workshop*

on Hot Topics in Networks. HotNets '22. https://dl.acm.org/doi/10.1145/3563766.3564109. New York, NY, USA: Association for Computing Machinery, 2022, 188.

[65] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. "What Do LLMs Need to Synthesize Correct Router Configurations?" In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. HotNets '23. https://dl.acm.org/doi/10.1145/3626111.3628194. New York, NY, USA: Association for Computing Machinery, 2023, 189.

[66] Abeba Birhane, Vinay Prabhu, Sang Han, Vishnu Naresh Boddeti, and Alexandra Sasha Luccioni. *Into the LAIONs Den: Investigating Hate in Multimodal Datasets*. http://arxiv.org/abs/2311.03449. 2023.

[67] Shayne Longpre, Gregory Yauney, Emily Reif, Katherine Lee, Adam Roberts, Barret Zoph, Denny Zhou, Jason Wei, Kevin Robinson, David Mimno, and Daphne Ippolito. *A Pretrainer's Guide to Training Data: Measuring the Effects of Data Age, Domain Coverage, Quality, & Toxicity*. http://arxiv.org/abs/2305.13169. 2023.

[68] Dora Zhao, Angelina Wang, and Olga Russakovsky. "Understanding and Evaluating Racial Biases in Image Captioning". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. https://openaccess.thecvf.com/content/ICCV2021/html/Zhao_Understanding_and_Evaluating_Racial_Biases_in_Image_Captioning_ICCV_2021_paper.html. 2021, 14830.

[69] Sudipto Mukherjee, Himanshu Asnani, Eugene Lin, and Sreeram Kannan. "ClusterGAN: Latent Space Clustering in Generative Adversarial Networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (2019). https://ojs.aaai.org/index.php/AAAI/article/view/4385, 4610.

[70] Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. "Data Validation for Machine Learning". In: *Proceedings of Machine Learning and Systems*. Ed. by A. Talwalkar, V. Smith, and M. Zaharia. Vol. 1. https://proceedings.mlsys.org/paper/2019/file/5878a7ab84fb43402106c575658472fa-Paper.pdf. 2019, 334.

[71] Sebastien C. Wong, Adam Gatt, Victor Stamatescu, and Mark D. McDonnell. "Understanding Data Augmentation for Classification: When to Warp?" In: *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. 2016, 1.

[72] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. "Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. https://dl.acm.org/doi/10.1145/3387514.3405892. Virtual Event USA: ACM, 2020, 632.

[73] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. "A Deep Reinforcement Learning Perspective on Internet Congestion Control". In: *Proceedings of the 36th International Conference on Machine Learning*. https://proceedings.mlr.press/v97/jay19a.html. PMLR, 2019, 3050.

[74] X. Nie, Y. Zhao, Z. Li, G. Chen, K. Sui, J. Zhang, Z. Ye, and D. Pei. "Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning". In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), 1231.

[75] Keith Winstein and Hari Balakrishnan. "TCP Ex Machina: Computer-generated Congestion Control". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. http://doi.acm.org/10.1145/2486001.2486020. New York, NY, USA: ACM, 2013, 123.

[76] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. "AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. https://doi.org/10.1145/3230543.3230551. New York, NY, USA: Association for Computing Machinery, 2018, 191.

[77] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. "Learning to Route". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. HotNets-XVI. http://doi.acm.org/10.1145/3152434.3152441. New York, NY, USA: ACM, 2017, 185.

[78] Vojislav Dukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. "Is Advance Knowledge of Flow Sizes a Plausible Assumption?" In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. https://www.usenix.org/conference/nsdi19/presentation/dukic. Boston, MA: USENIX Association, 2019, 565.

[79] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Yanhui Geng, Li Chen, K. Chen, and Hao Jin. "Online Flow Size Prediction for Improved Network Routing". In: *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. 2016, 1.

[80] Suraj Jog, Zikun Liu, Antonio Franques, Vimuth Fernando, Sergi Abadal, Josep Torrellas, and Haitham Hassanieh. "One Protocol to Rule Them All: Wireless {Network-on-Chip} Using Deep Reinforcement Learning". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. https://www.usenix.org/conference/nsdi21/presentation/jog. 2021, 973.

[81] Yiding Yu, Taotao Wang, and Soung Chang Liew. "Deep-Reinforcement Learning Multiple Access for Heterogeneous Wireless Networks". In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), 1277.

[82] Matthias Wichtlhuber, Eric Strehle, Daniel Kopp, Lars Prepens, Stefan Stegmueller, Alina Rubina, Christoph Dietzel, and Oliver Hohlfeld. "IXP Scrubber: Learning from Blackholing Traffic for ML-driven DDoS Detection at Scale". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22. https://doi.org/10.1145/3544216.3544268. New York, NY, USA: Association for Computing Machinery, 2022, 707.

[83] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. "MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. https://doi.org/10.1145/3452296.3472926. New York, NY, USA: Association for Computing Machinery, 2021, 287.

[84] Eytan Bakshy. "Real-World Video Adaptation with Reinforcement Learning". In: (2019). https://openreview.net/forum?id=SJlCkwN8iV.

[85] 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*. HotNets-XVI. https://doi.org/10.1145/3152434.3152448. New York, NY, USA: Association for Computing Machinery, 2017, 192.

[86] 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)*. https://www.usenix.org/conference/atc18/presentation/yan-francis. 2018, 731.

[87] Noga H. Rotman, Michael Schapira, and Aviv Tamar. "Online Safety Assurance for Learning-Augmented Systems". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. HotNets '20. https://doi.org/10.1145/3422604.3425940. New York, NY, USA: Association for Computing Machinery, 2020, 88.

[88] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. "Verifying Learning-Augmented Systems". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. https://doi.org/10.1145/3452296.3472936. New York, NY, USA: Association for Computing Machinery, 2021, 305.

[89] Silvery Fu, Saurabh Gupta, Radhika Mittal, and Sylvia Ratnasamy. "On the Use of ML for Blackbox System Performance Prediction". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. https://www.usenix.org/conference/nsdi21/presentation/fu. 2021, 763.

[90] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. *Automatic Curriculum Generation for Learning Adaptation in Networking*. http://arxiv.org/abs/2202.05940. 2022.

[91] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. "An Experimental Study of the Learnability of Congestion Control". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. http://doi.acm.org/10.1145/2619239.2626324. New York, NY, USA: ACM, 2014, 479.

[92] Shane Storks, Qiaozi Gao, and Joyce Y. Chai. *Recent Advances in Natural Language Inference: A Survey of Benchmarks, Resources, and Approaches*. http://arxiv.org/abs/1904.01172. 2020.

[93] Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, Stella Biderman, and Sasha Rush. *The Annotated Transformer*. http://nlp.seas.harvard.edu/annotated-transformer/.

[94] Jay Alammar. *The Illustrated Transformer*. https://jalammar.github.io/illustrated-transformer/.

[95] Munazza Zaib, Dai Hoang Tran, Subhash Sagar, Adnan Mahmood, Wei E. Zhang, and Quan Z. Sheng. *BERT-CoQAC: BERT-based Conversational Question Answering in Context*. http://arxiv.org/abs/2104.11394. 2021.

[96] Yen-Chun Chen, Zhe Gan, Yu Cheng, Jingzhou Liu, and Jingjing Liu. *Distilling Knowledge Learned in BERT for Text Generation*. http://arxiv.org/abs/1911.03829. 2020.

[97] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *arXiv:2010.11929 [cs]* (2021). http://arxiv.org/abs/2010.11929.

[98] Lucas Beyer, Xiaohua Zhai, and Alexander Kolesnikov. *Better Plain ViT Baselines for ImageNet-1k*. http://arxiv.org/abs/2205.01580. 2022.

[99] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. *Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey*. http://arxiv.org/abs/2003.04960. 2020.

[100] Siddhant Ray and Alexander Dietmüller. *Network Traffic Transformer*. Zenodo. https://doi.org/10.5281/zenodo.7186893. 2022.

[101] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian. 2012, 113.

[102] Abdullah Alomar, Pouya Hamadanian, Arash Nasr-Esfahany, Anish Agarwal, Mohammad Alizadeh, and Devavrat Shah. "{CausalSim}: A Causal Framework for Unbiased {Trace-Driven} Simulation". In: *20th USENIX Symposium on Networked Systems Design and Implemen-*

tation (NSDI 23). https://www.usenix.org/conference/nsdi23/presentation/alomar. 2023, 1115.

[103] Chandan Bothra, Jianfei Gao, Sanjay Rao, and Bruno Ribeiro. "Veritas: Answering Causal Queries from Video Streaming Traces". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. https://dl.acm.org/doi/10.1145/3603269.3604828. New York, NY, USA: Association for Computing Machinery, 2023, 738.

[104] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. "A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. https://dl.acm.org/doi/10.1145/2619239.2626296. New York, NY, USA: Association for Computing Machinery, 2014, 187.

[105] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. "A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. https://doi.org/10.1145/2785956.2787486. New York, NY, USA: Association for Computing Machinery, 2015, 325.

[106] P. C. Sruthi, Sanjay Rao, and Bruno Ribeiro. "Pitfalls of Data-Driven Networking: A Case Study of Latent Causal Confounders in Video Streaming". In: *Proceedings of the Workshop on Network Meets AI & ML*. NetAI '20. https://dl.acm.org/doi/10.1145/3405671.3405815. New York, NY, USA: Association for Computing Machinery, 2020, 42.

[107] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Networks". In: *Communications of the ACM* 63.11 (2020). https://dl.acm.org/doi/10.1145/3422622, 139.

[108] Jeremy Coulson, Henk van Waarde, and Florian Dorfler. "Robust Fundamental Lemma for Data-driven Control". In: ().

[109] Arthur S. Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A. Ferreira, Arpit Gupta, and Lisandro Z. Granville. "AI/ML for Network Security: The Emperor Has No Clothes". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. https://dl.acm.org/doi/10.1145/3548606.3560609. New York, NY, USA: Association for Computing Machinery, 2022, 1537.

[110] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-Independent Packet Processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014). https://doi.org/10.1145/2656877.2656890, 87.

[111] Gilberto Fernandes, Joel J. P. C. Rodrigues, Luiz Fernando Carvalho, Jalal F. Al-Muhtadi, and Mario Lemes Proença. "A Comprehensive Survey on Network Anomaly Detection". In: *Telecommunication Systems* 70.3 (2019). https://doi.org/10.1007/s11235-018-0475-8, 447.

[112] B. Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954. 2004.

[113] P. Phaal, S. Panchen, and N. McKee. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC 3176. 2001.

[114] Paul Tune and Darryl Veitch. "Towards Optimal Sampling for Flow Size Estimation". In: *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*. IMC '08. https://dl.acm.org/doi/10.1145/1452520.1452550. New York, NY, USA: Association for Computing Machinery, 2008, 243.

[115] Nick Duffield, Carsten Lund, and Mikkel Thorup. "Estimating Flow Distributions from Sampled Flow Statistics". In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. https://dl.acm.org/doi/10.1145/863955.863992. New York, NY, USA: Association for Computing Machinery, 2003, 325.

[116] Arpit Gupta, Rob Harrison, Ankita Pawar, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. "Sonata: Query-Driven Network Telemetry". In: *arXiv:1705.01049 [cs]* (2017). http://arxiv.org/abs/1705.01049.

[117] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. "Language-Directed Hardware Design for Network Performance Monitoring". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, 85.

[118] Minlan Yu, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch." In: *NSDI*. Vol. 13. 2013, 29.

[119] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. http://doi.acm.org/10.1145/2934872.2934906. New York, NY, USA: ACM, 2016, 101.

[120] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. "Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22. https://dl.acm.org/doi/10.1145/3544216.3544263. New York, NY, USA: Association for Computing Machinery, 2022, 693.

[121] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. "CONGA: Distributed Congestion-aware Load Balancing for Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. http://doi.acm.org/10.1145/2619239.2626316. New York, NY, USA: ACM, 2014, 503.

[122] M. P. Wand and M. C. Jones. *Kernel Smoothing*. 1st ed. Monographs on Statistics and Applied Probability 60. London ; New York: Chapman & Hall, 1995.

[123] *The CAIDA UCSD Anonymized Internet Traces*. http://www.caida.org/data/passive/passive_dataset.xml.

[124] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. "HPCC: High Precision Congestion Control". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. http://doi.acm.org/10.1145/3341302.3342085. New York, NY, USA: ACM, 2019, 44.

[125] Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene-rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe

Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. *Scikit-Optimize/Scikit-Optimize: V0.5.2*. Zenodo. https://zenodo.org/record/1207017. 2018.

[126] François Chollet et al. "Keras". In: (2015). https://keras.io.

[127] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning*. http://proceedings.mlr.press/v37/ioffe15.html. PMLR, 2015, 448.

[128] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. Ed. by Johannes Fürnkranz and Thorsten Joachims. http://www.icml2010.org/papers/432.pdf. Haifa, Israel: Omnipress, 2010, 807.

[129] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (2017). http://arxiv.org/abs/1412.6980.