Practically High Performant Neural Adaptive Video Streaming

SAGAR PATEL, University of California, Irvine, USA
JUNYANG ZHANG, University of California, Irvine, USA
NINA NARODYSTKA, VMware Research by Broadcom, USA
SANGEETHA ABDU JYOTHI, University of California, Irvine, USA and VMware Research by Broadcom, USA

Despite offering early promise, Deep Reinforcement Learning (DRL) suffers from several challenges in adaptive bitrate streaming stemming from the uncertainty and noise in network conditions. However, in this paper, we find that although these challenges complicate the training process, in practice, we can substantially mitigate their effects by addressing a key overlooked factor: the skewed input trace distribution in DRL training datasets.

We introduce a generalized framework, *Plume*, to automatically identify and balance the skew using a three-stage process. First, we identify the critical features that determine the behavior of the traces. Second, we classify the traces into clusters. Finally, we prioritize the salient clusters to improve the *overall* performance of the controller. We implement our ideas with a novel ABR controller, *Gelato*, and evaluate the performance against state-of-the-art controllers in the real world for more than a year, streaming 59 stream-years of television to over 280,000 users on the live streaming platform Puffer. Gelato trained with Plume outperforms all baseline solutions and becomes the first controller on the platform to deliver statistically significant improvements in both video quality and stalling, decreasing stalls by as much as 75%.

CCS Concepts: • **Networks** → *Application layer protocols.*

Additional Key Words and Phrases: Video Streaming, Deep Reinforcement Learning

ACM Reference Format:
Sagar Patel, Junyang Zhang, Nina Narodystka, and Sangeetha Abdu Jyothi. 2024. Practically High Performant Neural Adaptive Video Streaming. *Proc. ACM Netw.* 2, CoNEXT4, Article 30 (December 2024), 23 pages. https://doi.org/10.1145/3696401

1 INTRODUCTION

Video streaming is the prominent Internet application, accounting for over 75% of the entire traffic [14]. Despite this, delivering high quality video over the Internet continues to be challenging, primarily due to the noisy and highly unpredictable network conditions the video is sent over [68, 71]. The primary approach to tackle this is to use Dynamic Adaptive Streaming over HTTP (DASH) [57]. This approach divides the video into small, seconds-long, chunks and pre-encodes them at multiple bitrates. Then, during streaming, an Adaptive Bitrate (ABR) algorithm selects the bitrate of each chunk, adapting to the network conditions and maximizing the quality of experience.

Authors' Contact Information: Sagar Patel, University of California, Irvine, USA, sagar.patel@uci.edu; Junyang Zhang, University of California, Irvine, USA, junyanz9@uci.edu; Nina Narodystka, VMware Research by Broadcom, USA, nina.narodytska@broadcom.com; Sangeetha Abdu Jyothi, University of California, Irvine, USA and VMware Research by Broadcom, USA, sangeetha.aj@uci.edu.

ACM 2834-5509/2024/12-ART30
https://doi.org/10.1145/3696401

Recent work has shown the potential of data-driven machine learning approaches to ABR [9, 58, 65, 70] in surpassing traditional heuristic-based methods [24, 56, 67]. These techniques consist of two main components: a machine learning model that predicts future network conditions (e.g., a transmission time predictor) and a planning algorithm that uses these predictions to select the optimal bitrate (e.g., a dynamic programming algorithm). While these techniques are theoretically optimal with accurate predictors, achieving such performance is difficult due to the requirements of the planning component and the complexities in modeling the Internet (§ 3.1).

Deep Reinforcement Learning (DRL) offers a promising approach to overcoming these limitations by using a fundamentally different mechanism for data-driven decision-making. Instead of modeling the Internet and using predictions to select bitrates, DRL directly learns the bitrate selection strategy by iteratively optimizing a *policy* [59]. This involves evaluating the current policy, performing *actions* to gather *states* and *rewards*, and optimizing it to maximize selecting bitrates that lead to higher rewards. This approach bypasses the need to accurately predict the outcome of sending every bitrate, presenting a natural way out of the current gaps in Internet modeling.

However, despite the promise of the path presented by DRL, obtaining high real-world performance in ABR remains challenging for DRL [36]. This is because in ABR, unlike traditional DRL environments such as gaming or robotics, there exists an unpredictable underlying input process: the wide-area Internet. This process is formally called an "input process" [39]. During training, the inputs are replayed using a dataset of *input traces*, or system logs. Such input-driven DRL environments have several characteristics that make DRL training difficult. First, training in input-driven RL environments is inefficient, requiring a significant number of iterations [37]. Second, the dependence on external inputs such as network conditions introduces high levels of uncertainty and noise [39]. These challenges together make training highly non-trivial, causing several prior work [36, 37, 65] to conclude that addressing them was essential for real-world performance.

Our analysis reveals a key overlooked factor behind these challenges: the skewed distribution of input traces in training datasets. This skew results in limited training on rare or tail-end traces and introduces noise in learning due to updates based on a narrow set of traces. Consequently, the performance on these tail-end traces is often suboptimal, unlike the heavily optimized "common" traces. However, focusing on the performance of the tail-end traces is vital for data-driven controllers in improving the *overall* performance over baselines [27]. Unfortunately, such skew is prevalent in ABR. For example, over an 8-month period on the video streaming platform Puffer [65], low-bandwidth input traces made up less than 20% of the total, with only 4% experiencing any stalls. Therefore, addressing skew is essential for (a) mitigating amplified learning challenges and (b) improving overall controller performance, across both "common" and tail-end input traces.

While techniques for addressing data skew are prevalent in various contexts [16, 18, 31, 34, 44, 51, 69], standard supervised learning solutions such as oversampling or undersampling specific labeled classes do not apply to Reinforcement Learning, where the controller learns using states, actions and rewards (§ 9). The few solutions designed specifically for DRL are inadequate for ABR controllers because they fail to capture the trace-centric nature of the problem (§ 3.3). Thus, to effectively address this skew, we introduce a novel approach targeting the *input traces*.

Input traces, which represent logs of time-dependent complex processes, lack a conventional mechanism to identify and balance the skew with. These traces have no features or labels and do not directly contribute to a loss function. Thus, a mechanism to identify and balance the skew in input-driven environments is needed. To do so, in this work, we introduce a generalizable framework, Plume. Plume employs an automated three-stage process. *Critical Feature Identification:* We automatically determine the critical trace features to identify the traces. *Clustering:* We employ clustering to convert the critical features into salient identifiers. *Prioritization:* In this stage, we prioritize the clusters, such as to expose the controller to traces where it can learn the most (§ 4).

We evaluate our ideas with Gelato, a novel ABR controller. Trained with Plume, Gelato offers SOTA performance on the real-world streaming platform Puffer [65], streaming 59 stream-years of live TV to over 280,000 users [2, 65] in a year. It is the first controller on the platform to show statistically significant improvements in both video quality and stalling. It outperforms previous SOTA data-driven controllers, CausalSim [10] and Fugu [65], reducing stalls by 75% and 78%.

To assess the generalizability of Plume, we evaluate its performance on various network trace distributions and in two other applications, Congestion Control and Load Balancing. To facilitate this evaluation, we introduce TraceBench, a simplified ABR environment with parametrically generated traces to create diverse test trace distributions in a controlled and precise manner. Using it, we demonstrate Plume's dynamic performance across controllers, environments, and trace distributions (§ 7).

In summary, we make the following contributions:

- We systematically study an overlooked aspect of DRL training—skewed datasets—and find that they can have a surprisingly large impact on performance.
- We propose Plume as a generalizable framework for handling skewed datasets and improving the performance of DRL controllers in Video Streaming.
- We introduce Gelato, a new ABR controller. Plume-trained Gelato, deployed on the real-world Puffer platform [65] for more than a year, is the first controller with significant improvements in both video quality and stalling, reducing stalling by 75% over the previous state-of-the-art.
- We demonstrate the generalizability of Plume, across different distributions of network conditions and different networking applications.

This work does not raise any ethical concerns.

2 BACKGROUND

In this section, we give a brief overview of reinforcement learning and adaptive bitrate streaming.

2.1 Reinforcement Learning Preliminiaries

In Deep Reinforcement Learning (DRL), an *agent* interacts with an *environment*, receiving the current system state s_t at each timestep and taking action a_t from policy $\pi(a|s_t)$. The environment *transitions* to state s_{t+1} post action, awarding agent reward r_t [7, 55, 59].

In network environments, non-deterministic network conditions are primary sources of noise and uncertainty. These conditions determine the environment's response to the controller's actions. E.g. in adaptive bitrate streaming, external network conditions dictate whether a stall occurs.

Formally, these conditions are called "inputs", and input-driven environments form an Input-Driven Markov Decision Process [39], defined by $(S, A, Z, P_s, P_z, r, \gamma)$. Here, S is the state set, A the action set, Z the training input traces, P_s and P_z the state and input transition functions, r the reward function, and γ the discount. The state transition function $P_s(s_{t+1}|s_t, a_t, z_{t+1})$ defines the probability distribution of the next state s_{t+1} given the current state s_t, action a_t, and upcoming input z_{t+1}. Meanwhile, the input transition function $P_z(z_{t+1}|z_t)$ defines the probability of the next input value based on current, leading to an effective transition function given by $P_s(s_{t+1}|s_t, a_t, z_{t+1})P_z(z_{t+1}|z_t)$.

The DRL learning process aims to guide the policy π towards higher cumulative reward through a loop involving two steps: a *policy evaluation* step and a *policy improvement* step [23]. In policy evaluation, the agent assesses its policy by gaining experience through *acting* in the environment and using it in *function learning*. It updates its neural network to learn value function $v^\pi(s) = \mathbb{E}_\pi[G|s_0 = s]$, the expected return G from state s, where G is the discounted reward sum $G = \sum_{t=0}^{\infty} \gamma^t r_t$. Next, in policy improvement, the agent alters π to maximize v^π, iteratively learning by estimating and maximizing the value function.

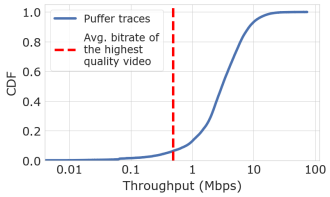


Fig. 1. **Puffer Input Trace distribution**: Distribution of Puffer trace effective throughput from Apr '21 - May '21. Under 6.5% of traces are below the highest quality video's average bitrate. Each stream in Puffer is considered a trace.

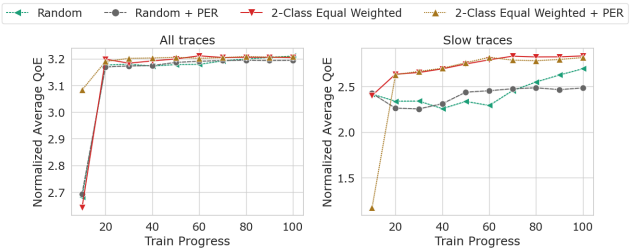


Fig. 2. **Comparing Prioritization Techniques**: Evaluating transition sampling (PER on/off) versus trace sampling (Random vs. 2-Class Equal Weighted) using Ape-X DQN [23]. 2-Class Equal Weighted Trace Selection excels in performance and training efficiency, unlike PER. Error bands represent 95% confidence interval.

On-policy and Off-policy DRL. DRL algorithms are broadly divided into two categories based on their policy evaluation stages. On-policy RL algorithms redo policy evaluation each iteration, using data from the latest policy [59]. These algorithms have found wide use in networking [26, 35, 38]. Off-policy RL algorithms partially use old policy data for better efficiency. They maintain a window of environment transitions, described by the tuple (s_t, a_t, r_t, s_{t+1}), in a FIFO buffer called Experience Replay [42]. Off-policy algorithms are similarly popular in networking, as used by [6, 64].

2.2 Adaptive Bitrate Streaming

In HTTP-based video streaming, the video is divided into chunks and encoded, in advance, at multiple discrete bitrates. During streaming, the most appropriate bitrate is chosen per chunk based on network conditions. The client also has a short buffer that can hold received chunks that have not been viewed yet. The ABR algorithm is responsible for sequentially selecting the video bitrate on a chunk level to maximize the viewer's Quality of Experience (QoE). Typically, the QoE is measured with a numerical function that awards higher quality, and penalizes both quality fluctuations and rebuffering. The quality may be defined by the encoded bitrate or by complex measures such as Structural Similarity Index Measure (SSIM) [61].

3 MOTIVATION

We discuss why Deep Reinforcement Learning (DRL) is used for ABR and the challenge of skewed training datasets. We then overview current techniques and the need for prioritized trace sampling.

3.1 Why use DRL for Adaptive Bitrate Streaming

Several data-driven controllers for Adaptive Bitrate Streaming exist today [9, 58, 65, 70]. These build on classical controllers like MPC [67], with a machine learning predictor replacing heuristics like the harmonic mean. They include two components: the machine learning predictor for future network conditions and the planning algorithm for bitrate selection.

With an accurate predictor, these techniques are theoretically optimal. However, practical performance is limited by difficulties predicting Internet behavior [17, 46]. The planning component needs accurate predictions for all bitrates in every condition, but predictions for rarely chosen bitrates can be inaccurate due to their out-of-distribution nature [10]. Additionally, planning often involves multiple future chunks, compounding prediction errors over time as current predictions are used for future ones (e.g., predicted buffer used as starting point for the next chunk) [25, 32].

Deep Reinforcement Learning (DRL) overcomes these challenges with a fundamentally different data-driven decision-making process. Instead of predicting Internet behavior and then selecting bitrates, DRL learns bitrate selection through a loop between policy evaluation and improvement [59], iteratively optimizing the viewer's experience. A key advantage is that it only considers bitrates similar to those already chosen by the policy [42]. Additionally, it relies on a bootstrap of aggregated performance rather than precisely needing outcomes for future chunks [22, 42]. These advantages make DRL promising for ABR while the research community catches up on Internet simulation.

3.2 Challenges with DRL Training

Having established the promise of DRL for video streaming, we now discuss its challenges.
Challenge 1: Inefficient exploration. In input-driven environments, most of the state-action space shows little reward feedback difference [37]. Standard exploration techniques, selecting random actions with ϵ probability and following greedy actions otherwise, have a low chance of finding a successful policy and require many training iterations. The imbalance in training datasets, especially the under-representation of rare traces, exacerbates this complexity. Such traces are rarely encountered by the controller, limiting its opportunity to discover successful strategies for them. However, performance in these tail-end traces can be crucial for higher overall performance [27].
Challenge 2: Noise and Uncertainty. Network conditions, or inputs, determine the environment's behavior and are the main source of uncertainty. For instance, when an ABR controller chooses a bitrate, it lacks knowledge of the client's link bandwidth. This unobserved factor directly impacts the client's wait time for the chunk. Such variability introduces noise into the learning process, causing identical states to yield widely different outcomes based on network conditions [39]. This noise is amplified when network trace distribution is skewed. In these cases, a single training iteration may not represent the full spectrum of input traces, leading to divergent or noisy updates.
Other Challenges with skew. Skew in the distribution of input traces presents challenges during the function learning phase of DRL training (§ 2.1). Since states are dependent on these input traces, a skewed input distribution leads to a skewed state distribution. This imbalance in the state distribution degrades the neural network performance, making it vulnerable to overfitting [28, 66].

3.3 Towards Prioritizing Trace Sampling

Next, we discuss prior ML techniques for handling skew and show the need for a new approach.
Prioritized Experience Replay (PER). Off-policy DRL algorithms use a buffer to store past state transitions and apply Prioritized Experience Replay (PER) [51] to sample them during function learning. PER employs prioritization, also known as importance sampling, to select state transitions based on their Temporal Difference error, focusing on transitions with higher error to improve the controller's predictions where most needed.

While PER is effective in traditional DRL settings [22, 23], it is limited in input-driven environments. PER addresses state skew in the function learning phase, but input trace skew affects the *acting* phase (§ 2.1). The controller has limited opportunities to act in tail-end traces. Without modifying trace selection during the acting phase, PER cannot increase exploration in tail-end traces or ensure comprehensive evaluation across the entire trace distribution.
Prioritized Trace Sampling. We reexamine the DRL workflow and identify a better location for prioritization. We propose a simple training paradigm in input-driven environments: prioritizing *trace sampling* during the acting step. This achieves high state-action space exploration and representative evaluation on all trace types.

To test our hypothesis, we enable prioritization at two points in the DRL workflow: sampling transitions in the experience buffer at the function learning step (PER enabled vs. disabled) and sampling input traces in the acting step (Random sampling vs. 2-Class Equal Weighted). 2-Class

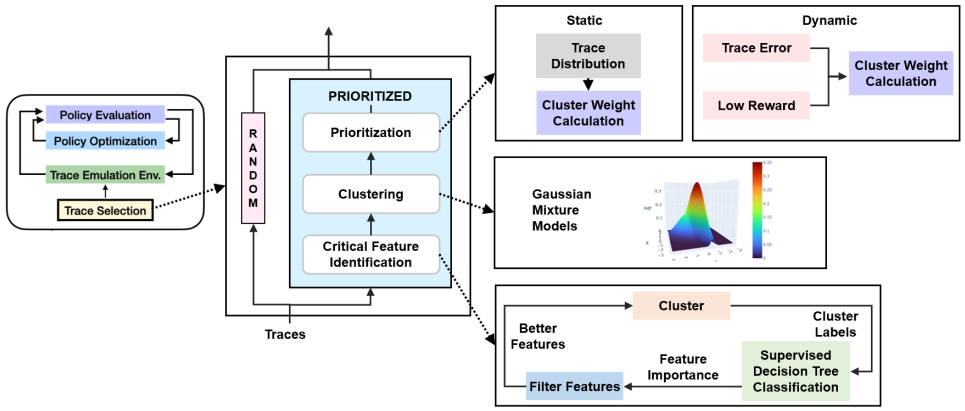


Fig. 3. **Plume System Diagram**: The Plume Workflow involves three key stages: (1) Critical Feature Identification, where we characterize the traces and their skew, (2) Clustering, where we try to simplify the prioritization problem by grouping traces, (3) Prioritization, where we prioritize important trace clusters.

Equal Weighted is a simple input trace prioritization scheme that divides traces into two classes, those with mean throughput higher/lower than 0.98 Mbps (Figure 1), and samples both classes equally. We evaluate each technique's impact on a DQN variation of Gelato controller for ABR, trained using the Ape-X DQN algorithm [23] (training settings detailed in § 5 and § 6.2)

In Figure 2, we observe that the simple 2-Class Equal Weighted gives the highest performance and training stability. By prioritizing tail-end slow throughput traces, we achieve high performance in both all and slow network traces without compromises. Enabling PER does not improve performance, even though the replay buffer can store 2M transitions (over 5000 traces). PER's performance falls short of the naive trace prioritization scheme. This highlights that trace distribution skew cannot be overcome at the function learning step.

4 DESIGN

Toward improving the performance of DRL training by balancing skew, we put forward the idea that trace selection is the aptest location for prioritization.

To balance the skew during trace selection, we take advantage of a key observation: input traces inherently correspond to users or workloads, with groups of them sharing similar characteristics. To ensure a balanced representation of the underlying users, the dataset must contain a roughly uniform number of input traces across them. We define input traces to have a set of user attributes $\Phi = [\phi_1, \phi_2, ...]$ given by the function $\Phi = X(trace)$, where X depends on the domain. These features identify similarities between user traces, and play a key role in balancing the skew.

Plume is a systematic framework to automatically balance this skew in input traces. Plume allows the agent to have balanced exploration and stable learning updates. Figure 3 gives an overview of the Plume workflow. Plume is implemented in the Trace Selection module which is responsible for supplying traces to the simulation environment. This module sits outside of the DRL training loop and is queried by the environment to get traces to replay. Plume has three key stages: critical feature identification, clustering, and prioritization.

In the critical feature identification stage (§ 4.1), Plume identifies the attributes of the input traces. In the clustering stage (§ 4.2), it simplifies the prioritization problem by clustering the attributes. Finally, in the Prioritization (§ 4.3) stage, Plume prioritizes the traces to balance input traces using one of two techniques: static or dynamic.

4.1 Critical Feature Identification

Input traces, which are time-dependent series of values that define complex external conditions, can be incredibly difficult to characterize and prioritize directly. Hence, the first step towards automated prioritization of traces is identifying the attributes Φ using critical feature identification.

To extract all features of the time series trace data, we rely on the popular feature extraction tool for the time series data, tsfresh [13]. We extract a large set of features $[\phi_1, \phi_2, ...\phi_n]$ broadly applicable to all input-driven DRL environments, such as truncated mean, ratio of values beyond a certain standard deviation, mean absolute change, or autocorrelation. For the full list of these features, see Appendix A. However, because this large set of features may not be relevant to every application, we introduce an automated three-step process to narrow down to the critical ones, inspired by recursive feature elimination in supervised learning [4].

First, we start with the large feature set and apply clustering to create a small number of clusters. This is denoted by $c = C([\phi_1, \phi_2, ...\phi_n])$, where c is the cluster labels, and C is the clustering function.

Second, we obtain the features most relevant in producing this mapping. To do so, we use the cluster labels c and train decision trees based on the features $[\phi_1, \phi_2, ...\phi_n]$. With this training, we can compute the information gain $IG(c, \phi_i) = H(c) - H(c|\phi_i)$ for each feature ϕ_i. Here, H is the Shannon entropy of the cluster labels, which is a measure of the average level of "uncertainty".

Third, we eliminate features with the lowest IG values. We continue this cycle of clustering, classification, and feature elimination until we are left with only the features that have high information gain. As we eliminate less useful features, we increase the number of clusters to ensure that the final feature set is sufficiently expressive. We note that this process, while empirically performant, is imperfect and can be improved with expert knowledge of the features.

Note that the clustering at this stage is solely for feature selection and has no impact on the clustering phase (§ 4.2).

4.2 Clustering

The second stage involves clustering traces using the critical features identified in the previous stage. In this stage, we attempt to reduce the complexity of balancing the skew by obtaining their salient clusters.

To detect skew in the dataset, we can look at the attributes Φ of input traces. However, balancing the skew based solely on these attributes proves to be a complex task. This difficulty arises because the attributes, represented as $[\phi_1, \phi_2, \ldots, \phi_n]$, are continuous random variables that may not be independent. In other words, modifying the skew of one attribute could negatively affect the skew of another. To address this, we cluster the traces to obtain a single distribution to balance. We use a clustering algorithm C to obtain the labels c so that the mapping again becomes $c = C(\Phi)$. By doing so, we create a ranking function that allows us to instead prioritize a categorical distribution of input traces, where the cluster labels act as the categories. We represent this distribution as y, where y_i is a category, or salient trace cluster within it.

To cluster the traces, we employ Gaussian Mixture Models (GMM) with Kmeans++ [47]. Gaussian Mixture Models use a generalized Expectation Maximization algorithm [1] and can effectively deal with the large variations found in input data. Note here GMMs must also balance the number of clusters with the variation to ensure that prioritization does not collapse the distribution. We balance this by conducting a search for the number using Silhouette scores [5]. This entire clustering process can add a one-time overhead in the order of minutes.

We visualize the clustering of Puffer traces automatically produced by Plume in Fig. 4 where we plot the clusters across two identified critical features. It produces minimal clusters while separating salient characteristics such as mean and variation in throughput. Note that the ratio of throughput

beyond 2.5 std is a measure of variation that calculates the proportion of the trace that lies beyond 2.5× standard deviation of the mean within that trace. score [5].

4.3 Prioritization

With critical feature identification and clustering stages complete, we have a categorical distribution of input traces y that we can balance by prioritization.

So far, we have discussed balancing the distribution y. While this can be done in a number of ways, to ensure that the balancing leads to meaningful performance improvements, we introduce a target function to balance the distribution around: "reward-to-go". Reward-to-go represents the additional rewards that a controller can still achieve. This can be formally defined by Equation 1:

$$\Delta G_{y_i} = \mathbb{E}_{y_i}[G^{\pi^*} - G^{\pi^\theta}] \tag{1}$$

In this equation, y_i is a category (§ 4.2) in the input trace distribution y, $G = \sum_{t=0}^{\infty} \gamma^t r_t$ is the discounted return of the trace as described in Section 2.1, G^{π^*} is the return under the optimal policy π^*, and G^{π^θ} is the return under the current policy. We aim to balance the input trace distribution based on how suboptimally the current policy performs, ensuring a uniform gap across all traces. In other words, we seek to ensure that target function $\Delta G_{y_i} = \Delta G_{y_j}$ for all categories y_i and y_j. However, calculating reward-to-go is often not possible in real-world situations because it depends on variables such as state features, and can require solving an NP-hard problem [40]. In this work, we introduce two strategies to approximate this prioritization: Static and Dynamic.

Static Prioritization. In this approach, we tackle skew by statically balancing the distribution of input traces. Specifically, we adjust the sampling weights to be the inverse of the distribution y, as expressed in Equation 2:

$$W_{y_i} = \frac{1}{f(y_i)} \tag{2}$$

Here, W_{y_i} signifies the prioritization weight for category y_i, and $f(y_i)$ is the original probability density function for the categorical distribution y. When we sample according to these prioritization weights, we modify the effective probability density function, which now becomes

$$f'(y_i) = \frac{W_{y_i} f(y_i)}{\sum_{y_k \in y} W_{y_k} f(y_k)}. \tag{3}$$

While there exists no analytical way to compute ΔG_{y_i}, in some cases, we can show that static prioritization effectively balances the skew. First, consider that under random trace sampling, the imbalance can be arbitrarily large:

Proposition 4.1. *Let L be a constant and y be a categorical distribution of input traces. Suppose*

$$\frac{\Delta G_{y_i}}{\Delta G_{y_j}} \approx \frac{f(y_j)}{f(y_i)},$$

then there exists a distribution of traces y such that

$$\frac{\Delta G_{y_i}}{\Delta G_{y_j}} \geq L.$$

Proof. Consider a distribution with two categories where

$$f(y_1) = \frac{1}{1+L} \quad \text{and} \quad f(y_2) = 1 - f(y_1) = \frac{L}{L+1}.$$

From the above, it follows that

$$\frac{\Delta G_{y_1}}{\Delta G_{y_2}} \approx L.$$

□

However, using static prioritization, this imbalance no longer exists:

Proposition 4.2. *Let y' denote the re-weighted categorical distribution of input traces. Suppose*

$$\frac{\Delta G'_{y_i}}{\Delta G'_{y_j}} \approx \frac{f'(y_j)}{f'(y_i)},$$

then

$$\Delta G'_{y_i} \approx \Delta G'_{y_j}.$$

Proof. From the given condition, we have

$$\frac{\Delta G'_{y_i}}{\Delta G'_{y_j}} \approx \frac{f'(y_j)}{f'(y_i)} = \frac{W_{y_j} f(y_j)}{W_{y_i} f(y_i)} \cdot \frac{\sum_{y_k \in y} W_{y_k} f(y_k)}{\sum_{y_k \in y} W_{y_k} f(y_k)} = 1.$$

□

Given these propositions, it is evident that under static prioritization, irrespective of the initial input trace distribution, the relative reward-to-go ratio $\frac{\Delta G' y_i}{\Delta G' y_j}$ is close to one, while this ratio can be large under random trace sampling. For both propositions, an underlying assumption is that the ratio $\frac{\Delta G_{y_i}}{\Delta G_{y_j}}$ is approximately equal to the inverse of the ratio of probability densities for the relevant categories. This assumption rests on the observation that the mean loss reduction used in optimizing the controller is expected to accrue larger error on the space less represented in its samples. Consequently, the reward-to-go gap decreases with increasing sampling probability.

Dynamic Prioritization. In dynamic prioritization, we compute an approximation of reward-to-go that adapts to the training process. Reward-to-go of a category can vary as the training progresses, and hence, the extent of prioritization needed for a category can differ across training.

$$\begin{aligned}
\Delta G_{y_i} &= \mathbb{E}_{y_i}[G^{\pi^*} - G^{\pi^\theta}] \\
\Delta G_{y_i} &\approx \mathbb{E}_{y_i}[\hat{G}(\Phi) - G^{\pi^\theta}] && \text{Approximate policy} \\
\Delta G_{y_i} &\approx \mathbb{E}_{y_i}[\hat{G}(\Phi) - G^{\pi^\theta}] - \mathbb{E}_{y_i}[G^{\pi^\theta}] && \text{Compensate bias}
\end{aligned} \tag{4}$$

As the optimal return cannot be calculated, we replace the return G^{π^*} with the learned expected return $\hat{G}(\Phi)$. $\hat{G}$ is a function approximator trained alongside controller training and exploration to map the trace attributes Φ to the observed return obtained based on a rolling set. The difference from this learned estimate serves as a measure for improvement yet to be achieved by the controller. However, because this estimate is based on the return samples seen so far, this approximation can be pessimistic and require an explicit optimism compensation. To address this concern, we introduce the second term, $-\mathbb{E}_{y_i}[G^{\pi^\theta}]$, which gives priority to traces that have low returns.

The dynamic weights are proportional to the normalized sum of components of ΔG_{y_i} (Eq. 4). Note that the prioritization is outside the DRL algorithm's training loop in Trace Selection (Fig. 3).

5 GELATO

We introduce Gelato, a novel ABR controller architecture. Unlike simpler DRL environments, ABR benefits from this new architecture, enhancing training efficiency and performance. As shown in Section 6, combining Gelato with the Plume framework yields a controller that can outperform all existing ABR controllers in real-world and simulated settings. Refer to Figure 5 for an overview.

Rewards. We optimize for SSIM, using reward coefficients from Fugu [65] (+SSIM, −stalls, −ΔSSIM). We utilize video chunk sizes and SSIM values from Puffer's public logs. Rewards are normalized

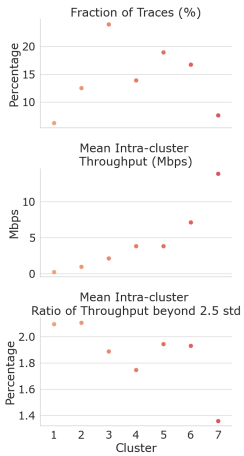


Fig. 4. **Clustering Visualization**: We show the clustering produced by Plume.

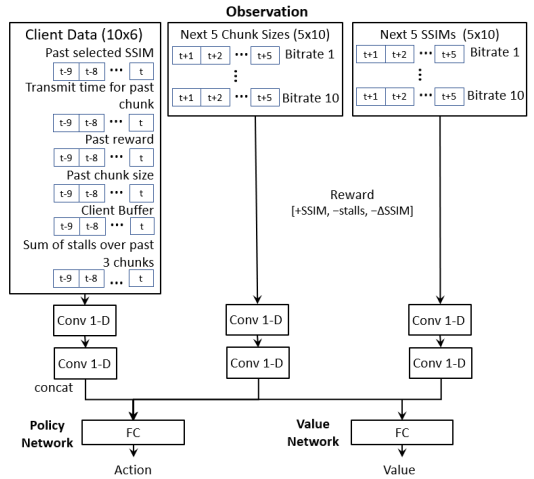


Fig. 5. **Architecture of Gelato**: Gelato takes as input complex features of the video stream.

using $r : sign(r)(\sqrt{|r|+1}-1)+\epsilon r$ and clipping, with $\epsilon = 10^{-2}$. This normalization, empirically effective for varied, large-scale rewards [48], mitigates the impact of extreme reward values.

Features. Gelato employs comprehensive application-level features, including client buffer and past rewards history, and a longer history of stalls over 30 chunks, aggregated every 3 chunks. This approach enhances the controller's adaptability to network conditions. Unlike Fugu, Gelato omits low-level TCP statistics, yet similarly utilizes transmit time and the sizes and SSIMs of upcoming chunks, available due to chunks being pre-encoded in ABR.

Neural Architecture. Gelato's neural network is optimized for efficiency, featuring an extra convolutional layer to downsample inputs, thus reducing FC layer input size. This deeper network enables advanced feature extraction while cutting trainable parameters and Mult-Add operations by 76% and 68% respectively, compared to Pensieve [38].

For Gelato's off-policy DQN variant (contrasted with PER in Figure 2), we employ the same architecture, substituting the policy and value networks with a single dueling Q-network [62]. Details are in Appx. B.

6 EXPERIMENTS

In this section, we present the findings of testing the impact of Plume across multiple agent architectures, and across simulation and real-world trials.

6.1 Implementation

We now turn to detail our implementation of all the experiments performed in this paper. We implement Plume as a Python library compatible with all major DRL frameworks.

Training environments and algorithms. We implement the ABR environment by extending the Park Project [37] and interfacing with Puffer traces [65]. We use the OpenAI Gym [12] interface and the RL libraries Stable-Baselines 3 [49] and RLlib [33].

Plume[1]. We implement Plume completely outside of the DRL workflow in the Trace Selection Module. To implement the critical feature identification stage, we use tsfresh [13] for its feature-extraction tools and Scikit-Learn [47] for its decision tree and clustering implementation. To

[1]https://github.com/sagar-pa/plume

implement the clustering stage, we use Scikit-Learn for its Gaussian Mixture Model and Silhouette scoring implementation. To implement the prioritization stage, we employ Numpy [20] and PyTorch [45]. For more details, see Appx. F.

6.2 Settings

In this section, we present the settings used in our experiments. We present our results as averages over 4 instances (4 controllers trained using the same scheme with different initial random seeds). This is consistent with the standard reporting practice in the RL community [23, 29, 41]. For testing on the Puffer platform, we select the best seed for benchmarking. For details, see Appx. B and D.

Simulation. For ABR, we use the Puffer platform network traces from April 2021 - May 2021. We enforce a trace length requirement of $3 - 17$ stream-minutes to reduce I/O overhead and prevent long traces from dominating training, randomly splitting long traces. This results in more than $75,000$ traces, of which we randomly select about $55,000$, representing over 4.25 stream-years, for our analysis. We use $40,000$ for training and about $15,000$ for testing. We evaluate every controller using the same train and test set.

Puffer Platform. We test Gelato with both random sampling and Plume on the live streaming platform Puffer from 01 Oct 2022 - 01 Oct 2023. The Puffer platform streams live TV channels such as ABC, NBC or CBS over the wide-area Internet to more than $280,000$ users [2, 65]. Over this time, we analyzed the ABR algorithms streamed over 58.9 stream-years of video. We report the performance as SSIM vs. stall ratio, following the convention used by the Puffer platform [65].

We compare Gelato-Random and Gelato-Plume-Static with the performance of the Buffer-based controller BBA [24], the classical planning controller MPC [67], Puffer optimized versions of the BOLA, v1 and v2 [3, 56], the in-situ continuous training controller Fugu's February version, Fugu-Feb [65], and CausalSim [10], a version of Bola tuned by trace-driven causal simulation.

6.3 Results

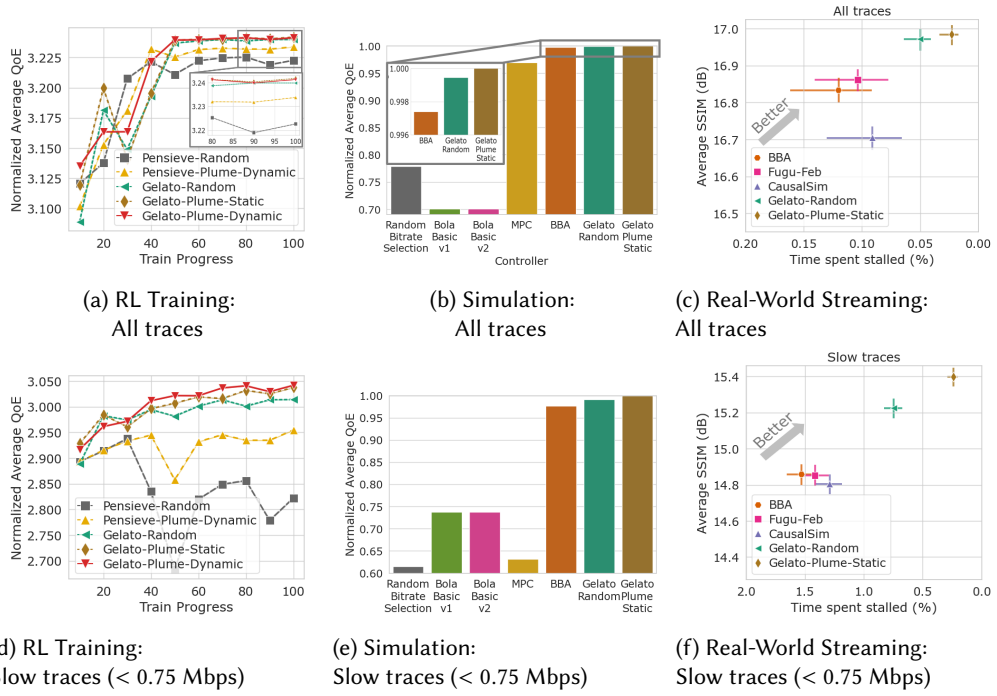
In this section, we present the results of our experiments in simulated and real-world ABR.

In Fig. 6, we present our results evaluating Plume. We present our observations below.

Plume outperforms random trace sampling in both simulation and real-world testing. In Figures 6a and 6d, we analyze the performance of Plume across training progress. We observe that Plume converges to a higher normalized QoE (defined as the reward in § 5), in both all traces and slow traces. We additionally see that Pensieve-Plume-Dynamic significantly improves upon Pensieve-Random, but that the improvement is not enough to match the performance of Gelato. In Fig. 6b and Fig. 6e, we benchmark the trained RL controllers with classical controllers. Due to ABR's tail-end nature, we also add random bitrate selection as a baseline for visualization. We find Plume to outperform random trace sampling and the classical controllers BBA, MPC, and Bola. We note that while the numerical differences may appear small due to the inherent scale of the metrics, they exceed the 95% confidence interval bands, and translate to large real-world differences as we will see.

Plume-Static closely tracks Plume-Dynamic. In Figs. 6a, 6d, we observe that Plume-Static, which employed a simpler prioritization strategy, closely tracks the performance of Plume-Dynamic. This is likely due to the fact that in these scenarios, the impact of shifting reward-to-go values or difficult input traces is minimal. However, as we will see later in Sec. 7, when the training distribution is anomalous or is significantly different from the testing distribution, Plume-Dynamic can prove effective over Plume-Static.

Gelato outperforms state-of-the-art controllers in the real world streaming live television over a 1-year period. To further understand the benefit of Plume, we run Gelato with Plume-Static and random sampling on the real-world live-streaming Puffer platform [65]. We opted for Gelato



(a) RL Training: All traces

(b) Simulation: All traces

(c) Real-World Streaming: All traces

(d) RL Training: Slow traces (< 0.75 Mbps)

(e) Simulation: Slow traces (< 0.75 Mbps)

(f) Real-World Streaming: Slow traces (< 0.75 Mbps)

Fig. 6. **Plume performance over Simulation and Real-World Streaming**: Plume surpasses random sampling in both controlled simulation-based experiments and in real-world settings. The simulation and training plots measure the QoE of the client, defined as reward in Sec. 5. Real-world Streaming plots are based on data from Puffer streams (Oct '22-Oct '23), aggregating over 58.9 stream-years. Data is re-plotted from its site [2] to combine different experiment periods. Error bars and bands show 95% confidence intervals. Plot axes vary due to differing objective scales.

combined with Plume-Static for this evaluation given its analogous performance to Plume-Dynamic in ABR, but with a simpler design. Additionally, we included Gelato with random sampling as a baseline for comparative analysis. In Figures 6c and 6f, we see that Gelato-Plume-Static outperforms the current state-of-the-art controllers Fugu-Feb and CausalSim, alongside the heuristic-based BBA in both SSIM and stalling. Although prior work [10, 65] reported statistically significant stalling improvements on Puffer, Gelato distinguishes itself by becoming the first ABR controller to achieve statistically significant improvements in both quality and stall reduction. This is particularly noteworthy as Gelato does not depend on low-level TCP metrics like Fugu or intricate simulation techniques that CausalSim uses.

Over this 1 year period, the algorithms streamed over 58.9 stream-years of videos to over 280,000 viewers across the Internet [2, 65]. Over this duration, Gelato-Plume-Static achieves 75%, 78% and 81% stall reduction compared to CausalSim, Fugu and BBA respectively (Fig. 6c). Gelato-Plume-Static additionally achieves SSIM improvements of 0.28, 0.12 and 0.15 dB over CausalSim, Fugu and BBA respectively. This quality improvement over BBA is more than 5× that of Fugu, which only managed a 0.03 dB improvement over BBA. CausalSim did not provide an SSIM improvement over BBA over this period. Gelato-Plume-Static has an average SSIM variation of 0.77 dB, compared to

0.67, 0.53 and 0.78 dB of CausalSim, Fugu and BBA respectively. Moreover, we find that Gelato-Random is a strong baseline, achieving 0.27 dB SSIM improvement and 45% stall reduction over CausalSim.

7 GENERALIZATION

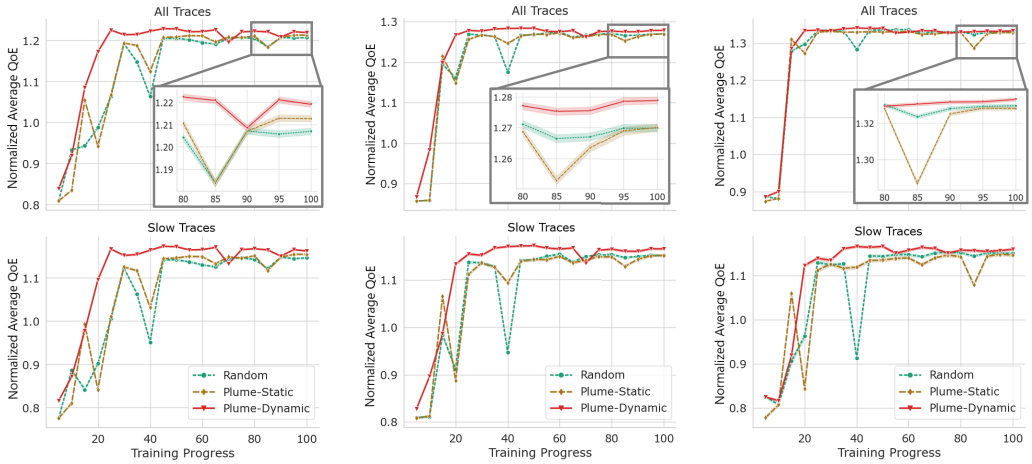
Having established the performance of Plume on real-world controllers and experiments in Section 6, in this section, we thoroughly microbenchmark Plume to study its generalizability across the distribution of traces used in ABR, as well as across other networking applications.

7.1 Settings

7.1.1 Generalization across input trace distributions: TraceBench. To assess Plume's generalizability across various input trace distributions and not just the one given by the users of Puffer, we introduce a controlled ABR environment, TraceBench. TraceBench implements two principal modifications to the standard ABR setting: simplifying quality-of-experience measurement to quality and stalling, and parameterizing traces by real-world trace attributes: mean and variance of network throughput. These changes enable comprehensive controller evaluation under diverse network conditions. Although a simplification, TraceBench closely approximates a broad spectrum of realistic scenarios. Note that parameterized trace generation, integral to TraceBench for creating varied scenarios, is not a component of the prioritization strategies themselves. For TraceBench traces, we focus on two mean throughput levels, slow and fast, and two throughput variance levels, high and low. We create three dataset sets with different trace proportions: Majority Fast, Balanced, and Majority Slow. Example trace visualizations are in Fig. 12 (Appx. C).

7.1.2 Generalization across networking applications: congestion control and load balancing.
Congestion Control. Congestion Control (CC) algorithms are responsible for determining the most suitable transmission rate for data transfer over a shared network. Based on network signals such as round-trip time between the sender and receiver and the loss rate of packets, a CC algorithm estimates sending rate that maximizes throughput and minimizes loss and delay. We evaluate Plume in CC by extending the code of Aurora [26]. Here, each trace is represented by 4 key simulation parameters: throughput, latency, maximum queue size, and loss. For training, we sample throughput from range $[100, 500]$ packets per second, latency from $[50, 300]$ milliseconds, max queue size from $[2, 50]$ packets, and loss rate from $[0, 2]$ percent. For testing, we broaden the ranges and sample throughput from $[50, 1000]$, latency from $[25, 500]$, max queue size from $[2, 75]$, and loss from $[0, 3]$. We sample latency uniformly evenly in the range, while sampling the rest evenly on a geometric progression. We note that we do this sampling only once and fix it for both training and testing.
Load Balancing. A Load Balancing (LB) algorithm in a distributed cluster decides which server to serve a new job at, such as to minimize the job's total processing time. When a job arrives, the LB algorithm does not know how busy each server is or how long each server will take to process the job. To make a good decision, it uses data such as the time between job arrivals, the duration of past jobs, and the number of jobs already waiting at each server. To evaluate, we use the Park Project [37]'s implementation. Each trace represents a time series indicating the size of arriving jobs over time. Following standard parameters, the inter-arrival times are sampled from the exponential distribution $exp(\lambda = 55)$, and the job sizes from the pareto distribution $pareto(x_m = 1.5, \alpha = 100)$. We limit the trace length to 650 to ensure that the variance of returns G is finite. As in congestion control, we perform this sampling once and fix it for both training and testing.

Further details on these settings are in Appendices C, D and E.



(a) Scenario 1: Training on Majority Fast, Testing on Majority Slow dataset.

(b) Scenario 2: Training and Testing on the Balanced dataset.

(c) Scenario 3: Training on Majority Slow, Testing on Majority Fast dataset.

Fig. 7. **Benchmarking Plume across Trace Distributions**: We benchmark prioritization techniques across different training and testing trace distributions. Plume-Dynamic provides generalizable performance improvement, beating the others in scenarios (1), (2) and (3). 95% confidence interval shown as error bands.

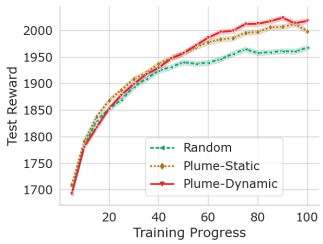


Fig. 8. Performance of Plume in congestion control. 95% confidence interval shown as error bands.

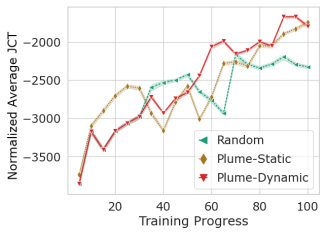


Fig. 9. Performance of Plume in Load Balancing. 95% confidence interval shown as error bands.

7.2 Results

Our experiments investigate two important questions. First, we investigate how the versions of Plume, Plume-Static and Plume-Dynamic generalize generalize to other network distributions in ABR, which can be possible in real-world settings. Second, we evaluate how Plume generalizes to other networking applications, congestion control and load balancing.

In Figure 7, we analyze the performance of Plume across various trace distributions. Particularly,

- Scenario 1: The training distribution is similar to the real world but the testing is adversarially different, i.e., we train on the Majority Fast but test on the Majority Slow dataset.
- Scenario 2: Both training and testing have a balanced set of traces, i.e., we train and test on the Balanced dataset.
- Scenario 3: The training distribution largely consists of the tail end of the testing distribution, i.e., we train on the Majority Slow but test on the Majority Fast dataset.

Plume outperforms random sampling regardless of trace distribution. As we observe in Figures 7a and 7b for the QoE for scenarios (1) and (2), Plume-Dynamic provides a significant

performance improvement over random sampling. Moreover, even in Figure 7c for scenario (3), where we may least expect prioritization to help, Plume-Dynamic is still better than random sampling. We additionally observe that Plume-Static, which performs well in scenario (1), falls behind Plume-Dynamic in scenarios (2) and (3) where the training input trace distributions are either anomalous or are dramatically different from the testing distribution. To better understand how PTS so effectively generalizes across all of these trace distributions, we visualize the selection weight of different traces during training in Fig. 10 in Appx. A.3.

Controllers trained with Plume are robust to trace distribution shifts. In the second row of plots in Figures 7a, 7b and 7c, we visualize the slow trace performance of different prioritization schemes. We observe that random trace sampling's performance in slow traces is largely dependent on its training dataset. If the training dataset had few slow traces, as in scenario (1), the performance is significantly worse than it is in scenario (3), where it had many. However, Plume-Dynamic's performance is robust to the training trace distribution: the controllers all converge to a similar QoE in all three scenarios. In the ever-changing landscape of users, devices, and infrastructure inherent to the network domain, this added robustness can be particularly important to reduce the need for retraining and ultimately the compute and energy requirements of the entire system.

Plume's performance gains are robust across networking applications. In Fig. 8, we visualize the performance of Random trace sampling, Plume-Static and Plume-Dynamic across training in congestion control. Similar to the results for Plume in ABR (§ 6), we observe that Plume converges to a higher performance, with Plume-Static closely tracking Plume-Dynamic. In Fig. 9, we observe a similar pattern in load balancing, with Plume also converging to a lower average job completion time (JCT) than standard random trace sampling.

Below, we summarize the findings of our experiments with ABR, CC and LB presented in Section 6, and the analysis of our extensive Plume benchmarking presented in this section.

- Plume is a generalized solution for DRL training in adaptive bitrate streaming that automatically balances the trace distribution, and offers significant improvement in performance over random sampling in simulation and in real-world testing, over both on-policy and off-policy algorithms.
- Plume's prioritization strategies work across trace distributions and networking applications, providing controllers with greater performance and robustness in all.
- Gelato trained with Plume offers the best performance when compared to prior ABR controllers on the real-world Puffer platform. It achieves 75% and 78% reduction in stalls over CausalSim [10] and Fugu [65] respectively. It also achieves a statistically significant SSIM improvement of 0.28 dB over CausalSim and 0.12 dB over Fugu.

8 DISCUSSION AND LIMITATIONS

We envision Plume to open a new avenue of research for DRL training. Rather than evolve into another hyperparameter, the problem of trace sampling lends itself to principled analysis, and a generalized and broadly applicable solution. However, our work still leaves a gap for future work.

The need for systematic study of input-driven DRL training. Our analysis of Plume highlights the significant impact of skew and the benefits derived from addressing it. This finding provides a strong motivation to explore other overlooked factors that may also influence input-driven DRL training. While the broader ML community has conducted in-depth studies on training parameters [11], DRL environments [15], and evaluation metrics [8], there is a lack of such research in the networking domain. Engaging in systematic studies in this front could enable the research community to better understand the potential of existing solutions and pave way for an empirical assessment of the real challenges faced by optimized input-driven DRL solutions.

Future direction for Plume. In addition to networking environments, Plume can also be beneficial in other trace-driven DRL settings such as drone control, autonomous driving, etc. Plume, as we

presented it, cannot be used directly in such environments with more complex input processes. However, extensions to Plume as presented in this paper may be an interesting future direction.

Sim2Real Gap. Plume changes *which* traces get sampled and not *how* they are simulated. Plume does not address the problem related to the gap between the simulation environment and the real-world setting (Sim2Real Gap). Thus, Plume cannot handle the scenario where training traces are incomplete or have experienced data shift from the runtime environment. Training and runtime solutions that bring simulation closer to reality can be combined with Plume.

Large-Scale Training. The benefits of higher state-action exploration and feature learning offered by Plume may diminish with a very deep neural network over a large number of training steps and parallel environments. Our experimental evidence suggests that Plume is highly relevant for practical DRL environments and training settings. However, we cannot ascertain the effectiveness of Plume at the scale of state-of-the-art Go agents [52], which requires training capabilities only available to large companies.

9 RELATED WORK

Prioritization in Supervised learning. Class imbalance is frequently a challenge in supervised data-driven networking problems, where samples of some classes of network conditions or scenarios occur rarely [16, 31, 34, 69]. A popular technique to address this problem is to oversample or undersample certain classes to ensure that the model does not drown out the error in the minority classes [30]. Such techniques cannot be used in reinforcement learning, where the learning happens using states, actions and rewards rather than a fixed dataset with labels.

Prioritization in DRL. While we present the first systematic methodology of prioritization of *input traces* in DRL, prioritization/importance sampling has been applied at other points in the DRL workflow. PER [51] is used to prioritize transitions in the replay buffer in actor-critic algorithms [60], in the multi-agent setting [18], and in text-based DRL environments [44] to improve sample efficiency. Horgan et.al [23] used PER in conjunction with distributed *acting* to improve feature learning. Schulman et.al [53, 54] employed importance sampling to reduce variance of on-policy training. However, as shown in our experiment (§ 3.3), these solutions do not address the skew in input-driven environments.

DRL for Networking and Systems applications. Following the promise of DRL, a number of prior works have worked to improve its performance in networking, improving sim2real gap and efficiency. Gilad et.al. [19] employed RL to find additional training traces to help the DRL agent generalize to unseen network conditions. Building on this idea, Xia et al. [63] introduced a systematic Curriculum Learning based approach for the same goal. It introduced the metric *Gap-to-baseline* for environment configurations and systematically generated the additional environment configurations needed for greater generalizaiblity. Both of these techniques addressing sim2real gap areff tangential to Plume and can be used alongside it. Mao et.al. [39] introduced the algorithm-side optimization of using input-dependent baselines to reduce the variance of on-policy algorithms at the policy optimization step. Doubly Robust estimation [27] helps in estimating performance variations during input-driven evaluation but does not address the skew in learning. These solutions, addressing various other challenges in DRL, serve as crucial motivation to address the skew underpinning DRL in adaptive bitrate streaming and can be combined with Plume.

10 CONCLUSION

Practical adoption of DRL-based ABR controllers is limited because the research community does not fully know how to produce high-performance controllers. We uncover that skew in the input datasets of DRL controllers plays a significant role in performance, and put forward Plume, a systematic, generalizable, and high-performant methodology for addressing that skew.

REFERENCES

[1] [n. d.]. Expectation–maximization algorithm - Wikipedia. https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm. (Accessed on 01/16/2023).

[2] [n. d.]. Puffer. https://puffer.stanford.edu/results/. (Accessed on 04/20/2022).

[3] [n. d.]. Puffer. https://puffer.stanford.edu/bola/. (Accessed on 06/09/2024).

[4] [n. d.]. Scikit-Learn Recursive Feature Elimiation. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html#sklearn.feature_selection.RFE. (Accessed on 01/15/2023).

[5] [n. d.]. Silhouette (clustering) - Wikipedia. https://en.wikipedia.org/wiki/Silhouette_(clustering). (Accessed on 01/16/2023).

[6] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. 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*. 632–647.

[7] Joshua Achiam. 2018. Spinning Up in Deep Reinforcement Learning. (2018).

[8] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. 2021. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems* 34 (2021), 29304–29320.

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

[10] Abdullah Alomar, Pouya Hamadanian, Arash Nasr-Esfahany, Anish Agarwal, Mohammad Alizadeh, and Devavrat Shah. 2023. {CausalSim}: A Causal Framework for Unbiased {Trace-Driven} Simulation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1115–1147.

[11] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. 2020. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[13] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W Kempa-Liehr. 2018. Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package). *Neurocomputing* 307 (2018), 72–77.

[14] V Cisco. 2018. Cisco visual networking index: Forecast and trends, 2017–2022. *White paper* 1, 1 (2018).

[15] Kaleigh Clary, Emma Tosch, John Foley, and David Jensen. 2019. Let's Play Again: Variability of Deep Reinforcement Learning Agents in Atari Environments. *arXiv preprint arXiv:1904.06312* (2019).

[16] Shi Dong. 2021. Multi class SVM algorithm with active learning for network traffic classification. *Expert Systems with Applications* 176 (2021), 114885.

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

[18] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. 2017. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*. PMLR, 1146–1155.

[19] Tomer Gilad, Nathan H Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. 2019. Robustifying network protocols with adversarial examples. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 85–92.

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[21] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[22] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

[23] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. 2018. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933* (2018).

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

[25] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. 2019. When to trust your model: Model-based policy optimization. *Advances in neural information processing systems* 32 (2019).

[26] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on internet congestion control. In *International conference on machine learning*. PMLR, 3050–3059.

[27] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. 2017. Unleashing the potential of data-driven networking. In *International Conference on Communication Systems and Networks*. Springer, 110–126.

[28] Justin M Johnson and Taghi M Khoshgoftaar. 2020. The effects of data sampling with deep learning and highly imbalanced big data. *Information Systems Frontiers* 22, 5 (2020), 1113–1131.

[29] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. 2018. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*.

[30] Harsurinder Kaur, Husanbir Singh Pannu, and Avleen Kaur Malhi. 2019. A systematic review on imbalanced data challenges in machine learning: Applications and solutions. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–36.

[31] Joffrey L Leevy, Taghi M Khoshgoftaar, and Jared M Peterson. 2021. Mitigating class imbalance for iot network intrusion detection: a survey. In *2021 IEEE Seventh International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 143–148.

[32] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643* (2020).

[33] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[34] Xiaoyu Liang and Taieb Znati. 2019. An empirical study of intelligent approaches to DDoS detection in large scale networks. In *2019 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 821–827.

[35] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 50–56.

[36] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. 2020. Real-world video adaptation with reinforcement learning. *arXiv preprint arXiv:2008.12858* (2020).

[37] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. 2019. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems* 32 (2019).

[38] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 197–210.

[39] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. 2018. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264* (2018).

[40] Melika Meskovic, Mladen Kos, and Amir Meskovic. 2015. Optimal chunk scheduling algorithm based on taboo search for adaptive live video streaming in CDN-P2P. In *2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 205–209.

[41] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[43] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[44] Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. 2015. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941* (2015).

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

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

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[48] Tobias Pohlen, Bilal Piot, Todd Hester, Mohammad Gheshlaghi Azar, Dan Horgan, David Budden, Gabriel Barth-Maron, Hado Van Hasselt, John Quan, Mel Večerík, et al. 2018. Observe and look further: Achieving consistent performance on Atari. *arXiv preprint arXiv:1805.11593* (2018).

[49] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* 22, 268 (2021), 1–8. http://jmlr.org/papers/v22/20-1364.html

[50] Antonin Raffin, Jens Kober, and Freek Stulp. 2022. Smooth exploration for robotic reinforcement learning. In *Conference on Robot Learning*. PMLR, 1634–1644.

[51] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[52] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

[53] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International conference on machine learning*. PMLR, 1889–1897.

[54] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[55] David Silver. 2015. Lectures on Reinforcement Learning. URL: https://www.davidsilver.uk/teaching/.

[56] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. 2020. BOLA: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1698–1711.

[57] Thomas Stockhammer. 2011. Dynamic adaptive streaming over HTTP– standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*. 133–144.

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

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

[60] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. 2016. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224* (2016).

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

[62] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1995–2003.

[63] Zhengxu Xia, Yajie Zhou, Francis Y Yan, and Junchen Jiang. 2022. Genet: automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 397–413.

[64] Zhiying Xu, Francis Y Yan, Rachee Singh, Justin T Chiu, Alexander M Rush, and Minlan Yu. 2023. Teal: Learning-Accelerated Optimization of WAN Traffic Engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 378–393.

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

[66] Han-Jia Ye, Hong-You Chen, De-Chuan Zhan, and Wei-Lun Chao. 2020. Identifying and compensating for feature deviation in imbalanced deep learning. *arXiv preprint arXiv:2001.01385* (2020).

[67] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. 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*. 325–338.

[68] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. 2015. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 509–522.

[69] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. 2021. MimicNet: fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 287–304.

[70] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. 2021. {SENSEI}: Aligning video streaming quality with dynamic user sensitivity. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 303–320.

[71] Xuan Kelvin Zou, Jeffrey Erman, Vijay Gopalakrishnan, Emir Halepovic, Rittwik Jana, Xin Jin, Jennifer Rexford, and Rakesh K Sinha. 2015. Can accurate predictions improve video streaming in cellular networks?. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. 57–62.

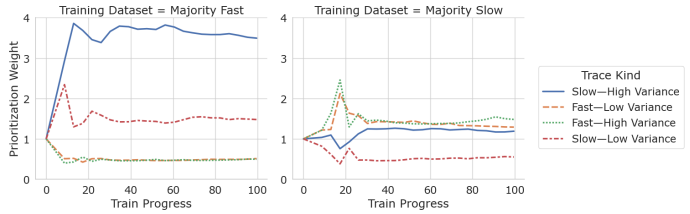


Fig. 10. **Visualization of the prioritization found by Plume-Dynamic in various datasets**: The relative change in sampling weight for each kind of traces over the training. Selecting all kinds of traces at weight 1 is equivalent to random sampling.

A PLUME DETAILS

In this section, we provide details, visualizations, and analysis of the Plume and its three stages.

A.1 Critical Feature Identification

We recall that in the Critical Feature Identification stage, Plume identifies traces by first extracting a wide range of features and then filtering them to find the critical features.

A wide range of features is extracted for each trace in the dataset of traces. Then, this set of features goes through our automated filtering process. During this process, about 40% of the features are eliminated. We start with 16 features, of which 7 describe the central tendency and 9 describe the spread. The features of central tendency include Mean, Quantiles of the 2.5^{th}, 5^{th}, and 95^{th}, Truncated mean of 5^{th}, 12.5^{th}, and $25th$ quantiles, and the Spectral Centroid of the Absolute Fourier Transform Spectrum. The 9 features of the spread are the Ratio of values beyond 1× and 2.5× standard deviation, Coefficient of Variation, Central approximation of Second Derivative, Mean Absolute Change truncated beyond the $[5^{th}, 95^{th}]$ and $[1.25^{th}, 98.75^{th}]$ quantiles, and Autocorrelation with lag of 3, 5, and 8.

A.2 Clustering

We recall that in the Clustering stage of Plume, we group similar traces together to attempt to reduce the complexity of the prioritization problem from a trace-level to a cluster.

We do this by automatically finding both the clustering and the optimal number of features through a search procedure. In ABR, we search for the number of clusters in the range $[6, 15]$, $[3, 7]$ in TraceBench, $[4, 9]$ in CC, and in the range $[3, 8]$ in LB.

A.3 Prioritization

We recall that in the Prioritization stage of Plume-Dynamic, we observe the controller's training and dynamically prioritize clusters to focus on those with the most to learn from.

Plume-Dynamic effectively adapts to all training trace distribution. To better understand how Plume-Dynamic so effectively generalizes across all of these trace distributions, we visualize the sampling weight of different traces during training in Figure 10. We observe that while training on the Majority Fast dataset, it undersamples the Fast traces and oversamples the Slow ones. In the Majority Slow dataset, it undersamples the Slow–Low Variance traces while oversampling the Fast and Slow–High Variance ones. This highlights the power of Plume-Dynamic's automated prioritization: It adapts itself to the distribution in each dataset and allows the controller to focus on clusters with the most to learn from.

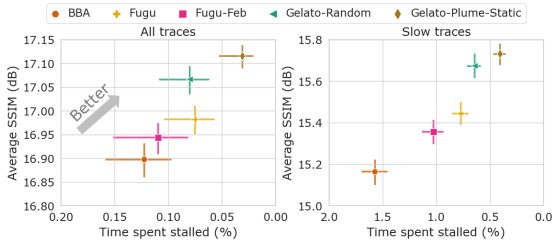


Fig. 11. Performance Plots from the Puffer Platform [2], presenting results from 07 Mar' 2022—05 Oct' 2022. The results visualize 25.5 steam-years of data. Similarly to our main results, we see that Gelato-Plume-Static (maguro) outperforms all other state-of-the-art ABR controllers in both video quality and stalling and that Gelato-Random (unagi) improves overall video quality while achieving similar stalling performance.

B ADAPTIVE BITRATE STREAMING DETAILS

In ABR, we introduce the novel controller architecture Gelato.

Gelato's neural architecture uses frame-stacking with 10 past values for the client data, and 5 future values of chunk sizes and SSIMs at every encoded bitrate. The client data is passed through a 1D convolution with a kernel size of 3 and 64 filters, followed by another 1D convolution of the same kernel size and filters. The chunk sizes and SSIMs are each passed through their own 1D convolution with a kernel size of 5 and 32 filters, each followed by another 1D convolution with the same kernel size and number of filters. The second layer of convolutions reduces the size of the resulting output by a factor proportional to the size of the kernel. The resulting features are concatenated and passed through a policy and a value network each made up of a single hidden layer of 256 neurons. Note that the value network is not used outside of training. An inference on Gelato's neural network takes less than 0.35 ms on average on a core of our $x86-64$ CPU server in Python—a minimal per-chunk overhead for Puffer's 2.002 second chunk duration. To train Gelato, we use the A2C algorithm [41] using a standard reward normalization strategy [48] and the training parameters: learning rate of 0.001, 64 parallel envs., $4e8$ training steps, t_{max} of 15, GAE N-step return of 15, γ of 0.95, 0..9 value function coefficient, Entropy of [5.75, .0025] annealed over $2e8$ steps, and Max Gradient Norm of 0.4.

The off-policy DQN variant of Gelato uses the same architecture, swapping the final policy and value networks for a single dueling Q-network made up of a single hidden layer of 256 neurons. We additionally use a standard reward normalization function [48] to normalize the rewards. To train this variant of Gelato, we use the Ape-X DQN algorithm [23] using the training parameters: 64 actors, $1e9$ training steps, learning rate of $7.5e-6$, replay batch size of 128, 0.95γ, replay buffer size of $2M$, N-step return of 7 and value clipping between $[-32, 32]$.

We use the Puffer Platform to gather traces for our simulation environment. The traces are system logs of the video streams—time series that include (i) the chunk sizes and SSIMs at all bitrates, (ii) the bitrate chosen by the ABR algorithm, and (iii) the time taken to transmit that chunk. We calculate the effective throughput over time and use it alongside the chunk sizes and SSIMs for simulation.

We train Pensieve [38] using its original architecture. However, because the original implementation could only work with the traces provided by the authors, to adapt Pensieve to new traces, we use the same training environment and DRL parameters as Gelato.

In presenting the results for Gelato in the real world, we re-plot the data found on the Puffer Platform [65] in Figure 6 in Section 6. In our analysis, we present the data from dates 01 Oct' 2022 through 01 Oct 2023. However, because the platform was experiencing issues and benchmarking

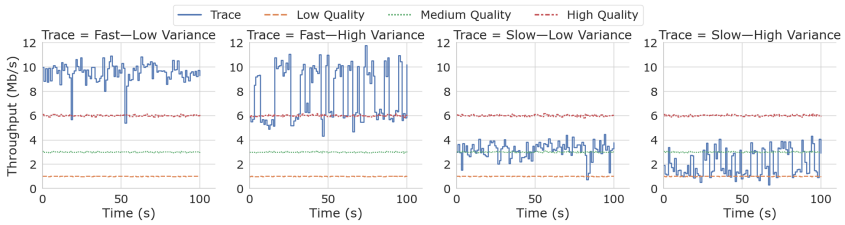


Fig. 12. **Visualization of Traces generated in TraceBench**: A Throughput vs Time plot of example traces used in TraceBench. The broad coverage of the mean and variance of the throughput requires the agent to learn to adapt to each kind of trace differently.

other ABR controllers, this data is split across multiple plots. To aggregate the data together, we first download the pre-processed public data available from the Puffer Website [2]. Second, we follow the same technique used by the platform and employ a sampling-based approach to estimate the mean and 95% confidence interval of quality, quality change, and stalling for each ABR algorithm. We ignore all the days when the platform was under maintenance (such as 16 January 2023) and days when the platform produced faulty data due to a known bug (such as 21 January 2023).

For completeness, we present the older results from the Puffer Platform in Figure 11 benchmarking the original version of the Fugu controller, which was taken off the platform on 06 October 2022. In this plot, we analyze 25.5 stream-years of data, collected from 07 March 2022 through 05 October 2022. We observe that Gelato-Plume-Static still outperforms the state-of-the-art ABR algorithms in both quality and stalling. This result highlights how Plume can successfully train robust and high-performant controllers in simulation, even outperforming in-situ trained controllers updated daily.

C TRACEBENCH DETAILS

In designing TraceBench, our objective is to create an environment to thoroughly evaluate and validate different prioritization techniques.

We build our environment on top of the standard ABR implementation found in the Park Project [37]. We allow the client to have a maximum buffer of 15 seconds. We consider traces with a maximum length of 100 seconds, with chunks of 1 second. The chunk sizes are generated by sampling a Gaussian distribution around the bitrates [1.0, 3.0, 6.0] megabytes per second.

When generating the traces, we consider two levels of throughput, fast and slow, and two levels of variance, high-variance and low-variance. When generating a trace, we use a 2-state Markov model switching between high and low throughput with different switching probabilities for each kind of trace. In Figure 12, we present a throughput vs. time visualization of each of the four different kinds of traces.

When training the controllers in TraceBench, we use the state-of-the-art feed-forward DQN algorithm Ape-X Dqn [23]. We use framestacking of history length 10. We additionally use a standard reward normalization function [48] to normalize the rewards. We use a simple fully connected architecture with 2 layers of 256 units. We additionally use the dueling and double DQN architecture with a hidden fully connected layer of 256 units. We use the training parameters: 4 actors, $4e6$ training steps, 32 replay batch size, $.975\gamma$, 250000 replay buffer size, N-step return of 7, ϵ annealing over $7e5$ steps and value clipping between $[-32, 32]$.

D CONGESTION CONTROL DETAILS

In CC, we train and evaluate Aurora [26] with different prioritization techniques. We use frames-tacking with a history length of 25. We use a 2-layer fully connected neural architecture with 64 units for both the policy and value function. We additionally use State-Dependent noise for exploration [50] and reward scaling. We use the algorithm A2C [41] with training parameters: learning rate of .000125, 16 parallel envs., $5e6$ training steps, t_{max} of 15, GAE N-step return of 15 steps, $.975\gamma$, value coefficient of 0.05, entropy of $[.1, .005]$ annealed over interval $2.5e6$ steps and max gradient norm of 0.25.

E LOAD BALANCING DETAILS

In LB, we evaluate different prioritization techniques using standard parameters. We use a 2-layer fully connected neural architecture with [256, 128] units and GeLU activation [21] for both the policy and value function. We additionally use reward scaling, and the algorithm PPO [54] with training parameters: learning rate of $2e-4$, 16 parallel envs., $5e6$ training steps, batch size of 256, GAE λ of .975, no advantage normalization, 30 epochs per update, $1e-4$ value function coefficient, entropy of $[.1, 1e-6]$ annealed over $5e6$ steps, clip range of 0.1 and max gradient norm of 0.2.

F IMPLEMENTATION DETAILS

A straightforward implementation of Plume can directly interfere with the various distributed training paradigms used in many DRL algorithms [23, 41]. To this degree, we implement our prioritization strategy using the distributed shared object-store paradigm in Ray [43]. This allows us to share the sampling weights across RL processes without interfering with any DRL workflows.

With our implementation, the overhead for Plume is minimal. The Critical Feature Identification and clustering stages are completed once before training, with runtimes in the order of minutes. In Plume-Dynamic, we train a neural network to map the attributes Φ of an input trace to the return G^{π^θ} in that trace parallel to the training. We maintain a short bounded history of the trace-return pairs for each category and use this history to compute the two components of our prioritization function. To compute the first term in our approximation, we take the ground-truth samples of trace feature-return pairs, measure the mean absolute error of the neural network for these samples, and average them across each category. To calculate the compensation term, we take the negative of the mean return found in each category. We do this prioritization process continuously, adjusting the weights to the controller's current needs. This dynamic prioritization calculation adds a computational overhead on the order of milliseconds per iteration. This added prioritization computation is handled in parallel to the DRL training and does not slow it down.

Received June 2024; revised September 2024; accepted October 2024