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

Alexander Dietmüller  
ETH Zürich, Switzerland  
adietmuele@ethz.ch  

Romain Jacob  
ETH Zürich, Switzerland  
jacobr@ethz.ch  

Laurent Vanbever  
ETH Zürich, Switzerland  
lvanbever@ethz.ch  

The CRediT statement for this work is available in Appendix A.

ABSTRACT

Machine learning (ML) is a powerful tool to model the complexity of communication networks. As networks evolve, we cannot only train once and deploy. Retraining models, known as continual learning, is necessary. 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, 3.5× the improvement of random sample selection. Memento is model-agnostic and can be applied beyond video streaming.

CCS CONCEPTS

• Computing methodologies → Lifelong machine learning  
• Information systems → Multimedia streaming

KEYWORDS

Video Streaming, Machine Learning, Continual Learning

1 INTRODUCTION

Adaptive Bit Rate (ABR) algorithms aim to avoid video stalls while maximizing the image quality in video streaming. This entails predicting the transfer time of video chunks, a complex task for which researchers are increasingly using machine learning (ML) [43, 71, 73]. Current ML-based ABR algorithms already achieve good average performance; i.e., high image quality while largely avoiding stalls [37, 73]. But rare stalls matter as they are known to impact user experience far more than image quality [17, 35]. 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 [10, 29, 51, 58].

In this paper, we ask if and how we can improve tail performance over time while maintaining the average. For ABR, this translates to reducing stalls while maintaining quality.

Figure 1: On Puffer [1], 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: [1]

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 [2]: slower inference means higher costs in delay, energy, and money.

Instead of using more, we propose to improve performance with a smarter selection of training samples, which we motivate with the following case study.

Case study  The best study of continual learning in networking to date is Puffer [1]. It monitors ABR performance of users streaming TV with randomly assigned algorithms. Its authors proposed their own ML-based ABR and retrained it daily with random samples from the past two weeks (Fig. 1).
We argue that we can do better with a smarter selection strategy that reliably picks up important tail samples. In particular, we show that the only selection strategy that reliably picks up important tail samples is Query-By-Committee (QBC) [60], which 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. Query-By-Committee (QBC) initially helps but degrades over time. Using more models to select training samples (QBC) initially helps but degrades over time. To the author’s surprise [73], 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 (Fig. 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 (Fig. 1, bottom). This illustrates the more training approach falling short. 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 (Fig. 1, bottom)?

We argue retraining did not help because training samples were selected randomly. As most streaming sessions perform similarly, this leads to an imbalanced training set with many similar samples and few tail ones. Doubling the number of training samples in the same setup—more data—does not help either (Fig. 2): we must address this imbalance to improve a model’s tail performance. Query-By-Committee (QBC) 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 that the models fail to identify rare samples reliably and eventually overfit on noise (Fig. 2), i.e., using more models is not enough to improve tail performance.

Problem Fig. 1 and Fig. 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, retraining 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, we must answer the following 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? Given the large resource costs, 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 Fig. 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. Fig. 3 illustrates how Memento integrates with an ML-based video streaming system transmitting videos in chunks. In ML-based ABR, a sample encompasses telemetry data for recent video chunks; samples are collected by the server. 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.

Main contributions

§2 We propose a sample-space-aware continual learning algorithm using density to improve tail performance.

§3 We present Memento (Fig. 3), a prototype implementation that uses sample-space density for sample selection. Memento is publicly available (Appendix B).

§4 We validate the effectiveness of our algorithm by applying Memento on an extensive ABR case study.
- We deployed Memento on Puffer and collected 10 stream-years of real-world data over 9 months.
- Memento stalls 14% less than the static model (3.5× better than daily retraining with random samples).
- Memento achieves this by retraining only 7 times with a marginal degradation on image quality (0.13%).
- Memento has easy-to-tune parameters.

§5 We conduct microbenchmarks using data center workloads to show that our algorithm is not limited to ABR.

Figure 3: Memento maximizes sample space coverage, improving the tail while rationalizing when to retrain.
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 any imbalance 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 Fig. 1, this yields good average performance but is unreliable at the tail. We need to address the imbalance of the training set.

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 [10, 29]. We found that this does not work well on 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 where 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.

Fig. 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 §3). The top row of Fig. 4 shows the mean loss and density per batch. We then use each metric to select 1 M samples and retrain the model. The bottom row compares the effect of each selection 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 does not preserve enough average samples and degrades performance on most other batches.

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 (§4.4 and Fig. 8).

These results also shed some light on why we observed QBC degrade over time (§1, Fig. 2). Overall, QBC has some inertia, as models are gradually replaced (more details in §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. These changes can be broadly categorized into covariate shift [61], i.e., previously unseen traffic patterns emerge or the prevalence of patterns changes and concept drift [68], 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; 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 C.

### 3 COVERAGE MAXIMIZATION

The core of Memento is Algorithm 1: its sample selection to approximately maximize the sample-space coverage.

Memento achieves this by estimating the sample space density from the distances between samples: the higher the density of a point in sample space (i.e., the more samples are close to it) the better the memory covers this part of sample 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 distributions of sample batches, leveraging batching and black-box dimensionality reduction (BBDR) for scalability (§3.2).
2. It estimates input- and output-space density using kernel density estimation (KDE, §3.3).
3. It discards batches probabilistically until fitting the memory constraints. It maps density to discard probability, balancing tail-focus and noise rejection (§3.4).
4. Once new samples are selected, it approximates how much the memory coverage has increased to decide whether retraining might be beneficial (§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 §4.2 and analyze the impact of each parameter on Memento’s performance in §4.4.

#### 3.1 Definitions

**Process** We consider a process $y = f(x)$ that maps inputs $x \in X = \mathbb{R}^n$ to outputs $y \in Y$, where $Y = \mathbb{R}$ 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 training samples $S$, where $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, 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 $|S'| \leq C$.

#### Algorithm 1: Memento Coverage Maximization

**Parameters:** Capacity $C$, threshold $r$, batch size $b$, bandwidth $h$, temperature $T$

**Input:** In-memory $S_{mem}$ and incoming $S_{new}$ samples

**Output:** Selected samples $S'$, decision retrain

begin

1. $S' \leftarrow S_{new} \cup S_{mem}$
2. $B' \leftarrow \{\}$; // or last train batches
3. retrain $\leftarrow$ False

<table>
<thead>
<tr>
<th>// §3.2: Distance measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \leftarrow$ BatchSamples($S'$, $b$)</td>
</tr>
<tr>
<td>$B \leftarrow$ BBDR($B$); // ($x, y$) $\rightarrow$ ($\hat{y}, y$)</td>
</tr>
<tr>
<td>($\hat{D}<em>{\text{pred}}, \hat{D}</em>{\text{out}}$) $\leftarrow$ DistributionDistances($B$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>// §3.3: Density estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\rho}_k \leftarrow$ KDE($D_k$, $h$)</td>
</tr>
<tr>
<td>For $k \in {\text{pred, out}}$</td>
</tr>
<tr>
<td>$\hat{\rho} \leftarrow$ min($\hat{\rho}<em>{\text{pred}}, \hat{\rho}</em>{\text{out}}$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>// §3.4: Sample selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $</td>
</tr>
<tr>
<td>$\hat{\rho}_{\text{discard}} \leftarrow$ soft max($\hat{\rho} / T$)</td>
</tr>
<tr>
<td>$i \leftarrow$ WeightedRandomChoice($\hat{\rho}_{\text{discard}}$)</td>
</tr>
<tr>
<td>$S', B \leftarrow$ DiscardBatch($S'$, $B$, $i$)</td>
</tr>
<tr>
<td>$\hat{\rho}, D \leftarrow$ UpdateDensities($\hat{\rho}, D, i$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>// §3.5: Retraining decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>if RCI($B, B'$) $\geq r$ then</td>
</tr>
<tr>
<td>retrain $\leftarrow$ True</td>
</tr>
<tr>
<td>$B' \leftarrow B$; // remember train batches</td>
</tr>
</tbody>
</table>

ACM SIGCOMM Computer Communication Review
3.2 Distance measurement

Batching Before discussing the distance computation, we need to consider scalability. Algorithm 1 has two main bottlenecks: (i) computing pairwise distances ($O(n^2)$); and (ii) updating density estimates after discarding a sample ($O(n)$). Memento [1] 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. Fig. 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 [39] that has been found to outperform alternatives like PCA [55]. 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) [20] which is closely related to the Kullback-Leibler Divergence (KL) [36]. 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 [20]:

\[
JSD(P, Q) = \sqrt{\frac{1}{2} (KL(P, M) + KL(Q, M))}
\]

where

\[
KL(P, M) = \sum_{x \in X} P(x) \log_2 \left( \frac{P(x)}{M(x)} \right)
\]

if $P$ and $Q$ are discrete distributions in the space $X$; 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 $m = \frac{1}{2}(p + q)$.

Inputs and Outputs It is not clear a priori whether distances in the sample input 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 (§3.3).

Probabilistic predictions During batching, Memento leverages probabilistic predictions; e.g., the transition time predictor in Puffer [73] outputs a probability distribution over 21 transition time bins. This probability distribution captures whether predictions are somewhat uncertain—thus indicating a need for training using more such samples—or certain. However, we must handle probabilistic estimates differently than point estimates: (i) When batching samples, Memento groups probabilistic predictions first by the distribution mode, then by their probability; (ii) The batch distribution is computed as a mixture distribution, i.e., the prediction distribution for batch $b_j$ is $P_j(x) = (1/|b_j|) \sum P_j(x)$, where $P_j$ is the probabilistic prediction for sample $j$.

---

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

---

Figure 5: Selecting samples based on the Euclidean distance between batch averages does not improve tail performance. Computing the Jensen-Shannon distance between batch distributions is a better choice. Mean and 90% CI over a two-week sliding window (see §4 for details).
### 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) [52].

**Definition 2 (KDE).** Let \( b \) be a sample batch and \( B \) a set of sample batches, and let \( d^k(b, b') = \text{JS}(P^k, P^{k'}) \) with \( k \in \{\text{pred}, \text{out}\} \) 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 \( \hat{\rho} \) at the location of batch \( b \) is defined as:

\[
\hat{\rho}^k_B(b) = \frac{1}{\sqrt{2\pi} |B|} \sum_{b' \in B} \exp \left( -\frac{d^k(b, b')^2}{2h^2} \right)
\]  

(1)

Intuitively, the kernel density estimate of \( b \) is inversely proportional to the distances to other batches, with diminishing weights for more distant ones. The bandwidth \( h \), the “smoothing factor”, determines how quickly this 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 the prediction or output space:

\[
\hat{\rho}_B(b) = \min \left( \hat{\rho}^{\text{pred}}_B(b), \hat{\rho}^{\text{out}}_B(b) \right)
\]  

(2)

If necessary, this approach can be generalized to fewer or more densities: for unsupervised learning without ground truth, Memento can only consider \( \hat{\rho}^{\text{pred}} \). For models with multiple outputs and thus multiple predictions, we can compute the minimum in Eq. (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 is flexible and can operate with any other provided distance metric.

### 3.4 Sample selection

Memento optimizes the sample-space coverage by iteratively discarding high-density batches. It computes the densities for all samples in \( S' = S_{\text{new}} \cup S_{\text{mem}} \), randomly discards batches with a density-dependent probability (Algorithm 1, Lin. 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 [23]:

\[
p^{\text{discard}} = \text{soft max} \left( \frac{\hat{\rho}_B}{T} \right)
\]  

(3)

where \( \hat{\rho}_B \) is a vector of densities for all batches \( b \in B \), and \( p^{\text{discard}} \) 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 uniformly random.

### 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 3 (Coverage increase).** Let \( B \) be a set of batches with density estimates \( \hat{\rho}_B \). We can approximate the region of the sample space covered by the samples in \( B \):

\[
\text{Coverage}(B) = \sum_{b \in B} \hat{\rho}_B(b)
\]  

(4)

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

\[
CI(B, B') = \min_{b \in B} (\hat{\rho}_B(b) - \hat{\rho}_{B'}(b), 0)
\]  

(5)

The relative coverage increase \( RCI \) from \( B' \) to \( B \) is then

\[
RCI(B, B') = CI(B, B') / \text{Coverage}(B)
\]  

(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 sample 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 \( r \), Memento triggers retraining. This approach presents a rational trade-off: the larger \( r \), the longer we wait for changes to accumulate before retraining. With a small \( r \), 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).
4 EVALUATION: REAL-WORLD BENEFITS

We use Puffer [1] 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 a public dataset with 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 retrained 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. Moreover, 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.

4.1 The Puffer project

The Puffer project is an ongoing experiment comparing ABR algorithms for video streaming [1]. 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 [78] and the time spent with stalled video.

Fugu is the ABR algorithm proposed by Puffer’s authors; it features 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 a random assignment of ABR algorithms and collects transmission time and \( p(y) \) the TTP-predicted probability. Score improvement equals TTP loss decrease:\(^2\) Available in deployment and replay.

4.2 Retraining with Memento

We use Memento to retrain Fugu’s TTP: every day, we use Memento to select the training samples and decide whether to retrain. 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 \( T = 0 \), §3.4). We collected data over 292 days (from Oct. 2022 to Jul. 2023), totaling around 10.8 stream-years of video data per variant. This experiment 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 video data each and a total of 90 stream-years of video data. This experiment allows answering Q4 and Q5.

Metrics We access Memento along three dimensions:

- **Prediction quality** is measured with the logarithmic score \( \logscore(y) = \log p(y) \) [22]; \( y \) is the video chunk transmission time and \( p(y) \) the TTP-predicted probability. Score improvement equals TTP loss decrease.\(^2\) Available in deployment and replay.
- **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 utilization** is measured by the number of retraining events in deployment and replay.

Parameters We set Memento’s memory capacity \( C \) to 1 M samples (same as the original Fugu model). Default parameters are a retraining threshold \( \tau \) 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 §4.4.

4.3 Deployment results

In this section, we show that Memento effectively improves the tail prediction quality with little retraining and that this translates into QoE improvements over FuguFeb.

In Appendix C, we provide additional plots with the evolution of QoE and predictions over time, and aggregate plots like those published on the Puffer paper and website [1, 73].\(^3\)

\(^2\)The logarithmic score is a commonly used metric for probabilistic predictions [22] 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.

\(^3\)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
(a) Memento improves the prediction quality at the tail, with a relatively 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 6:** Memento achieves its goal: it improves the tail prediction quality with minimal impact on the average (Fig. 6a). This requires little retraining (Fig. 6b) and translates into notable QoE improvements (Figs. 6c and 6d).

**Prediction quality**  Fig. 6a shows the score difference ECDF between Memento and FuguFeb in median and 1st-percentile scores on the whole deployment (higher is better). We observe that Memento improves tail prediction performance as intended, with a slight—yet expected—degradation on average: as memory is finite and Memento purposely 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**  Fig. 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 retrained 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.

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**  Figs. 6c and 6d show the relative QoE improvements achieved over FuguFeb for the SSIM and the time stalled (higher is better), respectively. First, we observe that Memento only marginally affects the SSIM; the average SSIM is 17.12 and 17.14 for Memento and FuguFeb, 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 FuguFeb (Fig. 6c).

Second, Fig. 6d shows that the improvement in stalls compared to FuguFeb is almost the same for both variants of Memento, even though there are large day-to-day variations: some days, FuguFeb 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 FuguFeb. In relative terms, Memento spent a 14% smaller fraction of stream time stalled than FuguFeb.

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 C). The latest version of Memento made the deterministic variant more robust but we see 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 Fig. 6b.

Finally, we observe that Memento reduces stalls by 3.5 times as much as retraining daily with random samples. In Appendix C, Fig. 14 we show Fig. 6a overlaid with the score improvements of Fugu in the past. We observe that random retraining improved the tail prediction scores significantly less and even worsened them on 20% of days.

---

slow,” while we consider all data points. As the filtering is ABR-independent, it does not impact relative results between ABRs.

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

---

Appendix C, Fig. 14 must be considered with caution, as the underlying data comes from different time periods and may not be comparable.
From predictions to QoE  One may wonder why the average prediction degradation (Fig. 6a) does not seem to strongly impact image quality (Fig. 6c), and, conversely, why significant tail improvements yield only a modest reduction in stalls (Fig. 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.

Looking closer, 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 Figs. 6c and 6d). Further investigations of the interplay between prediction quality and application performance would be interesting but are beyond the scope of this work. To facilitate further research, we publish all our retrained models including their training sample selection alongside the Puffer QoE data.

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 C.

Replicability All plots in Fig. 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 Fig. 7a shows Memento’s prediction performance over the batch size; larger sizes improve scalability but make sample selection more coarse-grained, which should hurt performance (§3.2). We observe a slight tail performance drop for larger batch sizes with 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 For each batch, Memento estimates how close nearby batches are; the kernel bandwidth $h$ determines what “nearby” means (§3.3). As the computed distances $JS(D(P, Q)) \leq |0, 1|$, bandwidths $> 1$ over-smooth (all batches are always “nearby”), and bandwidths $< 1$ under-smooth (no other batches are ever “nearby”). Both cases nullify the idea of estimating density, effectively making the sample selection random. Fig. 7b confirms this intuition: at the extremes, Memento performs like a random selection. We obtain the best tail improvement with a bandwidth around $1 \times 10^{-1}$.

Temperature Fig. 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 (§3.4). As expected, a 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 $1 \times 10^{-2}$.

Figure 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.
Figure 8: Loss-based selection is worse and less robust.

Alternative selection metrics  Fig. 8 shows the performance of using loss as an alternative selection metric: we still use temperature-based probabilistic selection but prefer to discard samples with a low loss rather than high density. At best, it gives tail improvements about half as Memento, but 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, the model performance decreases drastically, which mirrors our observations in §2.

We evaluate additional metrics in Appendix C: 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 Fig. 7c). We also show detailed results for Euclidean distances (§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 Fig. 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 attempting to remove the random fluctuations. We leave this challenge for future work.

Figure 9: Memento complements training and prediction improvements such as JTT and Matchmaker.

**Memento + Matchmaker + JTT**  MatchMaker [42] improves predictions by using an ensemble of models (default 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 Fig. 9).

JTT [40] improves performance by training twice: after the first training, misclassified samples are upsampled in the second and final training. Fig. 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 [73] 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 (Fig. 9).

Figure 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 [60] (Fig. 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 and compare an ensemble of Memento-trained models with an ensemble of QBC-trained models.
Fig. 10 shows that the tail improvements of QBC initially exceed Memento, and are comparable to Memento with JTT (Fig. 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 (§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.

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 [57] 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 (§5.2);
(2) picks up new patterns quickly (§5.3);
(3) is applicable to classification and regression (§5.4).

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, [10]). LARS uses several improvements to 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 §4) for two reasons: (i) We aim to show the limitations of different sample selection strategies, which is easier to do using small memories; (ii) LARS scales poorly, making comparing performance on larger memory sizes impractical—we optimized the original LARS implementation to scale to 20 k samples. Our optimization is available in our artifacts.

Workloads The simulation setup (Appendix D, Fig. 15) consists of two nodes and applications sending messages whose sizes follow three empirical traffic distributions from the Homa project (Fig. 16, [46]): Facebook web server (W1), DC-TCP (W2), and Facebook Hadoop (W3). W1 and W3 are similar size-wise, 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 combine in various iterations in the following experiments.

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 D 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.

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

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 ≤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 (Fig. 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.
Table 1: Memento can handle various traffic patterns and prediction types, outperforming other approaches.

(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.

5.3 Classification: Incremental learning

Next, we show that Memento quickly picks up new patterns and avoids catastrophic forgetting in an "Incremental Learning" setting, which is known for its challenging nature [21].

Setup We use the same model and setup as §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 (Fig. 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 D, Table 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 to retain samples from the new label.

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 (§5.2 and §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 under the current traffic conditions.

Results Memento generally shows the lowest 99th percentile prediction error (Fig. 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.

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. Combining both would likely be beneficial.

Finally, density computations limit Memento’s scalability. Batching helps (Fig. 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 [16]: 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 [44]. 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 [10]. Prior-based methods aim to prevent catastrophic forgetting by protecting model parameters from later updates [34, 51]. 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 [10, 29]. Hybrid methods combine both, e.g., training with a replay memory and a loss term penalizing performance degradation on old samples [58].

Memento builds on previous rehearsal-based approaches, incorporating ideas such as coverage maximization [13]. 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. [5] also suggest using the “right samples” to retrain AutoML systems [25]: 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 over the Internet; 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 [27] or changes in empirical loss [62], plus a time-based window (or multiple parallel windows) [16]. When a change is detected, these algorithms advance the time 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 C, Fig. 12) and not all patterns get outdated at once.

Matchmaker [42] 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 [39]) 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 (§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), which is popular approach to ML-based ABR [43, 72]. In fact, RL commonly uses a replay memory [56, 65], and it has been shown that it can greatly improve RL performance [4, 58].

**Transformers and other large models** In recent years, large models with billions of parameters have become popular in natural language processing [14, 50, 63] and computer vision [24, 32], usually based on Transformer architectures [66]; there are also first trials in networking [15, 38, 45].

We have evaluated Memento primarily on small models to investigate to impact of a smarter sample selection instead of using more resources such as more complex models and cannot claim with confidence that we would see comparable improvements for them. However, there is mounting evidence that even large transformer models suffer from data bias [8, 41, 77], 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 [47], and Memento may benefit from computing sample distances in this 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 [53]. 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 [70]. One example is Just Train Twice (JTT) [40], 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 §4.4 that JTT’s upsampling is more effective when Memento has identified tail samples first. 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 [10].

**Generalization** Memento could be applied to other ML-based networking applications, including congestion control [3, 30, 49, 69], traffic optimization [11], routing [64], flow size prediction [18, 54], MAC protocol optimization [31, 75], traffic classification [9, 67], network simulation [76], or DDoS detection [67]. Networking has proven to be a challenging...
environment for ML, and many proposed systems have only delivered modest or inconsistent improvements in real networks [6, 7, 73, 74]. In response, research has focused on providing better model architectures [3, 30, 73] and training algorithms [72], model ensembles for predictions and active learning [25, 42], real-world evaluation platforms [73, 74], uncertainty estimation [59] and model verification [19].

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 §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. We look forward to future research investigating, e.g., how to design ML models to best leverage coverage maximization.

**Ethical issues** This work does not raise any ethical issues.

**REFERENCES**

[12] François Chollet et al. keras. 2015. URL: https://keras.io.


[57] Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, John Schulman, Daniel Selsam,parentId=1609 /1109/1145/3626111.3628194.


[61] Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, John Schulman, Daniel Selsam, parentId=1609 /1109/1145/3626111.3628194.


A CREDIT STATEMENT

This section lists the author’s contributions to this work. The contributions are described using CRedit, the Contributor Roles Taxonomy, an ANSI/NISO standard.

All authors agree with this declaration of contributions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexander Dietmüller</td>
<td>Lead</td>
</tr>
<tr>
<td>Conceptualization</td>
<td>Lead</td>
</tr>
<tr>
<td>Data curation</td>
<td>Lead</td>
</tr>
<tr>
<td>Formal analysis</td>
<td>Lead</td>
</tr>
<tr>
<td>Investigation</td>
<td>Lead</td>
</tr>
<tr>
<td>Methodology</td>
<td>Lead</td>
</tr>
<tr>
<td>Project administration</td>
<td>Lead</td>
</tr>
<tr>
<td>Resources</td>
<td>Lead</td>
</tr>
<tr>
<td>Software</td>
<td>Lead</td>
</tr>
<tr>
<td>Validation</td>
<td>Lead</td>
</tr>
<tr>
<td>Visualization</td>
<td>Lead</td>
</tr>
<tr>
<td>Writing – original draft</td>
<td>Equal</td>
</tr>
<tr>
<td>Writing – review &amp; editing</td>
<td>Equal</td>
</tr>
<tr>
<td>Romain Jacob</td>
<td></td>
</tr>
<tr>
<td>Conceptualization</td>
<td>Supporting</td>
</tr>
<tr>
<td>Formal analysis</td>
<td>Supporting</td>
</tr>
<tr>
<td>Investigation</td>
<td>Supporting</td>
</tr>
<tr>
<td>Methodology</td>
<td>Supporting</td>
</tr>
<tr>
<td>Supervision</td>
<td>Equal</td>
</tr>
<tr>
<td>Visualization</td>
<td>Supporting</td>
</tr>
<tr>
<td>Writing – original draft</td>
<td>Equal</td>
</tr>
<tr>
<td>Writing – review &amp; editing</td>
<td>Equal</td>
</tr>
<tr>
<td>Laurent Vanbever</td>
<td></td>
</tr>
<tr>
<td>Conceptualization</td>
<td>Supporting</td>
</tr>
<tr>
<td>Funding acquisition</td>
<td>Lead</td>
</tr>
<tr>
<td>Methodology</td>
<td>Supporting</td>
</tr>
<tr>
<td>Supervision</td>
<td>Equal</td>
</tr>
<tr>
<td>Writing – original draft</td>
<td>Supporting</td>
</tr>
<tr>
<td>Writing – review &amp; editing</td>
<td>Supporting</td>
</tr>
</tbody>
</table>

B ARTEFACTS AND REPLICATION.

All code to reproduce the results in this paper is publicly available and provided as an archive in the supplemental material of this paper for review. The archive contains a README with specific instructions on how to run the code. Below, we give an overview of reproducing the Puffer and synthetic results, and how to use Memento in other projects.

Puffer All Puffer-based experiments are based on the data published by the Puffer project [1]. To replicate our results, you need to first download the Puffer data with the commands provided in the README file. It is possible to download only a subset of the published data to save both space and computation time. In particular, we provide a small demonstration script that uses the same code as our main experiments but uses a week of data.

Once data is downloaded, you can run two kinds of commands, both of which are described in the README file: analysis and replay. The analysis aggregates the downloaded data to create the figures shown in §4.3, comparing the performance of Memento to Fugu. The replay retrains the TTP model with data selected by Memento to create the figures shown in §4.4, evaluating different parameter choices with different periods of data to ensure that results are replicable.

Synthetic All synthetic experiments are based on data generated by an ns-3 network simulation. We provide the data used in the paper as well as the simulation scripts to (re-)generate it. The README file describes the commands to replay the simulation data for retraining with Memento to create the figures shown in §5.

Memento We provide the implementation of Memento as a Python package so it can be easily included in other projects. Our implementation offers a clean interface for instantiating a memory with a given capacity, which exposes methods to insert and get samples. We provide implementations for the distance metrics used in this paper, as well as a generic interface to implement new metrics. Finally, we also implement alternative sample selection strategies with the same interface, such as random selection, loss-based selection, or LARS (see §5), as well as an interface to combine multiple memories.

C PUFFER: SUPPLEMENTAL RESULTS

Recurring patterns Fig. 12 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. 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 (see e.g., the step around week 7), then fade away.

Aggregate performance Fig. 17 shows the aggregated SSIM and percent of stream-time spent stalled on a 2D grid. Specifically, we show two sets of ABRs. First Memento (default and $T = 0$) and $\text{Fugu}_{\text{Feb}}$, 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 FuguFeb, aggregated from 2020 until Fugu was discontinued.

Timeseries (real-world) Fig. 18 shows QoE results per day over time for Memento with default parameters, Memento \((T = 0)\) and for FuguFeb since September 19th. Fig. 19 shows the same results, but the mean and bootstrapped 90\% CI over a two-week sliding window. Fig. 20 shows the prediction improvements over FuguFeb for Memento with default parameters and with \(T = 0\).

Past degradation of Memento \((T = 0)\) over time In a previous deployment, we observed Memento \((T = 0)\) to degrade over time, as shown in Fig. 13. 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 again beginning to see the signs of noise accumulation, in particular stronger coverage increase and frequent retraining, and are monitoring the current Memento \((T = 0)\) deployment closely.

Prediction score improvements compared to the past Fig. 14 shows Fig. 6a overlayed with the prediction score improvements of Fugu compared to FuguFeb 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.

Alternative selection metrics Fig. 21 shows further alternative sample selection metrics in addition to loss (§4.4).

Fig. 21a 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.

Fig. 21b 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. Going by label counts alone removes too many of these samples and the model forgets common patterns, similar to the loss metric.
### 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 [26]. We have implemented both models using Keras [12]. For all layers, we use batch normalization [28] and ReLU activation [48]. We train both networks with the Adam optimizer [33] using the default decay parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$. We train for up 200 epochs with early stopping.

### Workload distributions

Fig. 16 shows the message size distribution for the workloads we use, published by the HOMA project [46]: Facebook web server (W1), DC-TCP (W2), and Facebook Hadoop (W3). Messages are generated with Poisson-distributed inter-arrival times.

### Samples in memory

Table 2 shows the number of samples in memory after the last iteration of §5.2 and §5.3. Memento retains all workloads in memory, whereas other approaches either forget existing or fail to pick up new samples.

### D SIMULATION: SUPPLEMENTAL RESULTS

#### Simulation setup

Fig. 15 illustrates the setup we used for the evaluation of Memento in simulation.

![Simulation setup](image)

**Figure 15: 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 [26]. We have implemented both models using Keras [12]. For all layers, we use batch normalization [28] and ReLU activation [48]. We train both networks with the Adam optimizer [33] using the default decay parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$. We train for up 200 epochs with early stopping.

#### Workload distributions

Fig. 16 shows the message size distribution for the workloads we use, published by the HOMA project [46]: Facebook web server (W1), DC-TCP (W2), and Facebook Hadoop (W3). Messages are generated with Poisson-distributed inter-arrival times.

#### Samples in memory

Table 2 shows the number of samples in memory after the last iteration of §5.2 and §5.3. Memento retains all workloads in memory, whereas other approaches either forget existing or fail to pick up new samples.

### Appendix D

#### Table 1: Model parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Classification</th>
<th>Regression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden Layers</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Hidden Units</td>
<td>512</td>
<td>362</td>
</tr>
<tr>
<td>Learning Batchsize</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>$5.91 \times 10^{-5}$</td>
<td>0.382</td>
</tr>
</tbody>
</table>

**Table 1: Model parameters.**

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

<table>
<thead>
<tr>
<th></th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rare Patterns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td>5696</td>
<td>9053</td>
<td>5219</td>
</tr>
<tr>
<td>Random</td>
<td>252</td>
<td>19634</td>
<td>114</td>
</tr>
<tr>
<td>LARS</td>
<td>3159</td>
<td>14231</td>
<td>2610</td>
</tr>
<tr>
<td>FIFO</td>
<td>0</td>
<td>20000</td>
<td>0</td>
</tr>
<tr>
<td>Incremental Learning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td>5376</td>
<td>5888</td>
<td>8704</td>
</tr>
<tr>
<td>Random</td>
<td>1266</td>
<td>17836</td>
<td>898</td>
</tr>
<tr>
<td>LARS</td>
<td>9540</td>
<td>9557</td>
<td>903</td>
</tr>
<tr>
<td>FIFO</td>
<td>0</td>
<td>0</td>
<td>20000</td>
</tr>
</tbody>
</table>

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

Finally, Fig. 21c 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.

#### Increased memory capacity

Fig. 22 shows the results for Memento compared to a random memory with the same capacity (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)

Fig. 23 shows the evolution of each benchmark experiment over the whole 6-month duration. Fig. 24 and Fig. 25 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.
Figure 17: 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 18: Evolution of QoE metrics over time. Absolute Values for each ABR.

Figure 19: Evolution of QoE metrics over time. Absolute Values for each ABR. Mean and bootstrapped 90% confidence interval over two-week sliding windows.
Figure 20: Logscore improvements compared to Fugu\textsubscript{Feb}.

(a) Confidence

(b) Label counts

(c) Stalled sessions

Figure 21: Additional alternative selection metrics.

Figure 22: Logscore improvements compared to Fugu, concretely the median and tail (1st percentile) improvements for each day. We compute the mean over all three 6-month replays, and plot again the mean of this value and bootstrapped 90\% confidence intervals over two-week windows.
Figure 23: Logscore improvements compared to Fugu (see Fig. 22) for all selection metric experiments.
Figure 24: Logscore improvements compared to Fugu (see Fig. 22) for all selection metric experiments.
(a) JTT with random sample selection.

(b) JTT with Memento's sample selection.

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

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

Figure 25: Logscore improvements compared to Fugu (see Fig. 22) for all combination experiments.