Continuum: A Platform for Cost-Aware, Low-Latency Continual Learning

Huangshi Tian, Minchen Yu, Wei Wang
Hong Kong University of Science and Technology
htiana@cse.ust.hk, myuaj@connect.ust.hk, weiwa@cse.ust.hk

ABSTRACT
Many machine learning applications operate in dynamic environments that change over time, in which models must be continually updated to capture the recent trend in data. However, most of today’s learning frameworks perform training offline, without a system support for continual model updating.

In this paper, we design and implement Continuum, a general-purpose platform that streamlines the implementation and deployment of continual model updating across existing learning frameworks. In pursuit of fast data incorporation, we further propose two update policies, cost-aware and best-effort, that judiciously determine when to perform model updating, with and without account for the training cost (machine-time), respectively. Theoretical analysis shows that cost-aware policy is 2-competitive. We implemented both polices in Continuum, and evaluate their performance through EC2 deployment and trace-driven simulations. The evaluation shows that Continuum results in reduced data incorporation latency, lower training cost, and improved model quality in a number of popular online learning applications that span multiple application domains, programming languages, and frameworks.

KEYWORDS
Continual Learning System, Online Algorithm, Competitive Analysis

1 INTRODUCTION
Machine learning (ML) has played a key role in a myriad of practical applications, such as recommender systems, image recognition, fraud detection, and weather forecasting. Many ML applications operate in dynamic environments, where data patterns change over time, often rapidly and unexpectedly. For instance, user interests frequently shift in social networks [48]; fraud behaviors dynamically adapt against the detection mechanisms [31]; climate condition keeps changing in weather forecasting [62]. In face of such phenomenon—known as concept drift [33, 71, 78, 84]—predictive models trained using static data quickly become obsolete, resulting in a prominent accuracy loss [21, 42].

An effective approach to tame concept drift goes to online learning [14], where model updating happens continuously with the generation of the data. Over the years, many online learning algorithms have been developed, such as Latent Dirichlet Allocation (LDA) [43] for topic models, matrix factorization [25, 55] for recommender systems, and Bayesian inference [15] for stream analytics. These algorithms have found a wide success in production applications. Notably, Microsoft reported its usage of online learning in recommendation [73], contextual decision making [2], and click-through rate prediction [37]. Twitter [50], Google [58], and Facebook [41] use online learning in ad click prediction.

Despite these successes, the system support for online learning remains lagging behind. In the public domain, mainstream ML frameworks, including TensorFlow [1], MLlib [60], XGBoost [18], Scikit-learn [67], and MALLET [57], never explicitly support continual model updating. Instead, users have to compose custom training loops to manually retrain models, which is cumbersome and inefficient (§2.2). Owing to this complexity, models are updated on much slower time scales (say, daily, or in the best case hourly) than the generation of the data, making them a poor fit for dynamic environments. Similar problems have also been found prevalent in the private domain. In Google, many teams resorted to ad-hoc scripts or glue code to continually update models [10].

In light of these problems, our goal is to provide a system solution that abstracts away the complexity associated with continual model updating (continual learning). This can be challenging in both system and algorithm aspects. In the system aspect, as there are diverse ML frameworks each having its own niche market, we require our system to be a general-purpose platform, not a point solution limited to a particular framework. In the algorithm aspect, we should judiciously determine when to perform updating over new data. Common practice such as periodic update fails to adapt to dynamic data generation, and is unable to timely update models in the presence of flash crowd. Moreover, model updating incurs non-trivial training cost, e.g., machine-time. Ideally, we should keep models fresh, at low training cost.

We address these challenges with Continuum, a thin layer that facilitates continual learning across diverse ML
frameworks. Figure 1 gives an overview of Continuum. Continuum exposes a common API for online learning, while encapsulating the framework-dependent implementations in backend containers. This design abstracts away the heterogeneity of ML frameworks and models. Continuum also provides a user-facing interface through which user applications send new data to refine their models. Continuum determines, for each application, when to update the model based on the application-specific update policy. It then notifies the corresponding backend to retrain the model over new data. Upon the completion of updating, the model is shipped as a container to the model serving systems such as Clipper [23], and is ready to serve prediction requests.

While Continuum supports customizable update policies for applications, it implements two general policies for two common scenarios. The first is best-effort, which is tailored for users seeking fast data incorporation without concerning about the retraining cost, e.g., those performing online learning in dedicated machines for free. Under this policy, the model is retrained continuously (one retraining after another) in a speculative manner. In particular, upon the generation of the flash crowd data, the policy speculatively aborts the ongoing update and restarts retraining to avoid delaying the incorporation of the flash crowd data into the model.

The second policy, on the other hand, is cost-aware, in that it strives to keep the model updated at low training cost. It is hence attractive to users performing online learning in shared clusters with stringent resource allocations or in public cloud environments with limited budget. We show through measurement that the training cost is in proportion to the amount of new data over which the model is updated. Based on this observation, we formulate an online optimization problem and design an update policy that is proven to be 2-competitive, meaning, the training cost incurred by the policy is no more than twice of the minimum under the optimal offline policy assuming the full knowledge of future data generation.

We have implemented Continuum in 4,000 lines of C++ and Python. Our implementation is open-sourced [75].

To evaluate Continuum, we have ported a number of popular online learning algorithms to run over it, each added in tens of lines of code. These algorithms span multiple application domains and were written in diverse ML frameworks such as XGBoost [18], MALLET [57], and Velox [24]. We evaluate their performance using real-world traces under the two update policies. Compared with continuous update and periodic update—the two common practices—best-effort policy and cost-aware policy accelerate data incorporation by up to 15.2% and 28%, respectively. Furthermore, cost-aware policy can save up to 32% of the training cost. We also compare Continuum with Velox [24], the only system that supports (offline) model retraining to our knowledge. We have implemented a movie recommendation app in both systems. Evaluation result shows that Continuum improves the average recommendation quality by 6x.

In summary, our key contributions are:

- The design and implementation of a general-purpose platform which, for the first time, facilitates continual learning across existing ML frameworks;
- The design and analysis of two novel online algorithms, best-effort and cost-aware, which judiciously determine when to update models in the presence of dynamic data generation, with and without accounting for the training cost, respectively.

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce online learning (§2.1) and motivate the need for continual model updating through case studies (§2.2). We discuss the challenges of having system support for continual learning (§2.3).

2.1 Online Learning

In many machine learning (ML) problems, the input is a training set \( S = (s_1, s_2, \ldots) \) where each sample \( s_i \) consists of a data instance \( x \) (e.g., features) and a target label \( y \). The objective is to find a model, parameterized by \( \theta \), that correctly predicts label \( y \) for each instance \( x \). To this end, ML algorithms iteratively update model parameter \( \theta \) over the training samples, in an online or offline manner, depending on whether the entire training set is readily available.

In online learning [14] (continual learning), the training samples become available in a sequential order, and the model is updated continually with the generation of the samples. Specifically, upon the generation of sample \( s_i \), an online learning algorithm updates the model parameter as

\[
\theta_{i+1} \leftarrow f_o(\theta_i, s_i),
\]

where \( f_o(\cdot) \) is an optimization function chosen by the algorithm. As opposed to online learning, offline learning (batch learning) assumes the entire batch of training set...
and trains the model by iteratively applying an optimization function \( f_k(\cdot) \) over all samples. That is, in each iteration \( k \), it updates the model parameter as
\[
\theta_{k+1} = f_k(\theta_k, S).
\]

Compared with offline learning, model updating in online learning is much more lightweight as it only incorporates one data sample at a time. This allows online learning to efficiently adapt to dynamic environments that evolve over time, in which the model must be frequently updated to capture the changing trend in data. We next demonstrate this benefit through case studies.

2.2 Case Study in Dynamic Environments

**Methodology** We study three popular online learning applications based on Personalized PageRank [8], topic modeling [44], and classification prediction [32], respectively. For each application, we replay real-world traces containing the time series of dynamic data generation. We train a base model using a small portion of the dataset at the beginning of the trace, and periodically update the model through online learning. We evaluate how online updating improves the model quality by incorporating new data. We also compare online learning with offline retraining. All experiments were conducted in Amazon EC2 [6], where we used c5.4xlarge instances (16 vCPUs, 32 GB memory and 5 Gbps link).

**Personalized PageRank** Our first application is based on Personalized PageRank (PPR) [8, 53, 54, 89], a popular ML algorithm that ranks the relevance of nodes in a network from the perspective of a given node. We have implemented a friend recommendation app in Scala using dynamic PPR [89]. We evaluate its performance over a social network dataset [27] crawled from Twitter.

Figure 2a illustrates how the quality of the base model deteriorates over time compared with the updated models. Specifically, we consider two metrics, L1 error [89] and Mean Average Precision (MAP) [7, 20]. The former measures how far the recommendation given by the base model deviates from that given by the updated model (lower is better); the latter measures the quality of recommendation (higher is better). It is clear that without model updating, the error accumulates over time and the quality of recommendation (MAP) given by the base model quickly decays (10% in one hour).

We further compare online learning with offline retraining. In particular, every 10 minutes, we retrain the PPR model in both offline and online manner. Figure 2b compares the training time of the two approaches. As offline learning retrains the model from scratch over all available data, it is orders of magnitude slower than online learning, making it a poor fit in dynamic environments.

**Topic Modeling** We next turn to topic modeling [44], a popular data mining tool [16, 77, 82, 87] that automatically detects themes from text corpora. We implemented a topic trend monitor in MALLET [57], a natural language processing toolkit offering an efficient implementation of Latent Dirichlet Allocation (LDA) [13] for topic modeling. We evaluate its performance over a large dataset of real-world tweets [86] by periodically updating the topic model every 10 minutes.

To illustrate how the incorporation of new data improves the quality of the model, we measure its perplexity [13, 80], an indicator of the generalization performance of topic model (lower is better). Figure 3a depicts the results. As we seed more data into the model, its perplexity decreases, meaning better performance. We next evaluate online learning against offline retraining by measuring their training time. As shown in Figure 3b, it takes orders of magnitude longer time to retrain the model offline using all historical data than updating it online.

**Classification Prediction** Our final study goes to classification prediction [35, 37, 56]. We consider an ad recommendation system in which there are many online ads that might be of interest to users. The system determines which ad should be displayed to which user by predicting the probability of user click. Based on the user’s feedback (click), it improves the prediction. We use real-world traffic logs from Criteo Labs [47] as the data source. We choose as our tool Gradient-Boosted Decision Tree
(a) Perplexity of LDA model decreases as more data are incorporated into the model, meaning better performance.

(b) Online and offline training time of LDA models.

Figure 3: Results of LDA algorithm on Twitter dataset.

Figure 4: Results of GBDT algorithm on Criteo dataset.

Table 1: Lines of code used to implement the training loop and model updating in our implementation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training Loop</th>
<th>Model Updating</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPR [89]</td>
<td>211</td>
<td>41</td>
</tr>
<tr>
<td>LDA [13]</td>
<td>377</td>
<td>56</td>
</tr>
<tr>
<td>GBDT [32]</td>
<td>558</td>
<td>44</td>
</tr>
</tbody>
</table>

The complexity manifested in our first-hand experience has also been found prevalent in industry. Notably, Google has reported in [10] that many of its teams need to continuously update the models they built. Yet, owing to the lack of system support, they turned to custom scripts or glue code as a workaround.

2.3 Challenges

Given the complexity associated with dynamic model updating, there is a pressing need to have a system solution to streamline this process. However, this can be challenging in two aspects, system and algorithm.

System Challenge There are a wide spectrum of ML frameworks, each having its own niche market in specific domains. For instance, Tensorflow [1] is highly optimized for deep neural network and dataflow processing; XGBoost [18] is tailored for decision tree algorithms; MALLET [57] offers optimized implementations for statistical learning and topic modeling; MLlib [60] stands...
out in fast data analytics at scale. As there is no single dominant player in the market, we expect a general-purpose platform that enables model updating across diverse frameworks. This requires the platform to provide an abstraction layer that abstracts away the heterogeneity of existing ML frameworks—including programming languages, APIs, deployment modes (e.g., standalone and distributed), and model updating algorithms. Such an abstraction should be a thin layer, imposing the minimum performance overhead to the underlying frameworks.

Algorithm Challenge In theory, to attain the optimal data recency, a model should be updated as soon as a new data sample becomes available (§2.1). In practice, however, model updating takes time (§2.2) and may not complete on arrival of new data. An update policy is therefore needed to determine when to perform model updating, and the goal is to minimize the latency of data incorporation. Note that the decisions of model updating must be made online, without prior knowledge of future data generation, which cannot be accurately predicted. This problem becomes even more complex when the training cost (e.g., instance-hour spent on model updating in public clouds) is concerned by users. Intuitively, frequent updating improves data recency, at the expense of increased training cost. Ideally, we should reduce the latency of data incorporation, at low training cost.

3 CONTINUUM

In this section, we provide our answer to the aforementioned system challenge. We present Continuum, a thin layer atop existing ML frameworks that streamlines continual model updating. We have implemented Continuum in 4,000 lines of C++ and Python, and released the code as open-source [75]. Continuum achieves three desirable properties:

- **General purpose**: Continuum exposes a common interface that abstracts away the heterogeneity of ML frameworks and learning algorithms (§3.3). It also supports pluggable update policies that users can customize to their needs. (§3.4).
- **Low overhead**: Continuum employs a single-thread architecture in each instance (§3.2), allowing it to efficiently orchestrate model updating across multiple applications, without synchronization overhead.
- **Scalability**: Continuum’s modular design enables it to easily scale out to multiple machines (§3.2).

3.1 Architecture Overview

Continuum manages multiple online learning applications (Figure 1). Figure 5 gives an overview of the training loop for a single application. Before we explain how it works, we briefly introduce each component in the loop.

Figure 5: The training loop in Continuum.

An application interacts with Continuum through a client. The application is also associated with a backend container, which encapsulates the framework-dependent implementation of the training logic. The backend performs model training and communicates the results with the Continuum kernel through RPC.

The Continuum kernel mainly consists of two modules: runtime profiler and update controller. The runtime profiler logs and profiles the training time for each application. Specifically, it uses linear regression (§4.1) to predict the training time based on the amount of the data to be incorporated into the model. The prediction result is then used as an estimation of the training cost, with which the update controller decides, for each application, when to perform updating based on the application-specified policy.

Putting it altogether, the training loop in Continuum goes as follows (Figure 5). ① Upon the generation of new data, the application calls the client REST API to send the data information, which contains the raw data, or—if the data is too large—its storage locations (§6). ② Continuum maintains the received data information in an external storage (e.g., Redis [69]), and then decides whether to trigger model retraining. ③ Once the decision is made, Continuum notifies the corresponding backend to perform retraining. ④ The backend fetches the data information from the storage and retraines the model with the new data. ⑤ Upon finishing retraining, the backend notifies Continuum to update the training time profile for that application. The updated model can then be shipped to model serving system (e.g., Clipper [23]) for inference (§6).

3.2 Single-Thread Architecture

When Continuum is used to manage multiple applications, runtime profiler and update controller need to maintain application-specific information, e.g., the amount of new data to be incorporated, last model update time, etc.
A possible choice to organize that information is multi-thread architecture, i.e., allocating one thread for each application and managing information using that thread dedicatedly.

However, we found that multi-threading suffers from context-switch overhead. Instead, we adopt a single-thread approach that maintains information of all applications in the main thread and let other components access them asynchronously. Our design shares similarity with the delegation approach in accessing shared memory [70].

Scalability is attainable even if we seemingly place all data together. We minimize the coupling between applications by separating system-wide data to the database. When in need of scaling, we launch multiple instances of Continuum and let each manage information for a set of applications. Our evaluation (§5.4) has demonstrated the linear scalability of Continuum.

3.3 Backend Abstraction

As a general-purpose system, Continuum abstracts away the heterogeneity of ML frameworks and learning algorithms with a common abstraction and a lightweight RPC layer. Listing 1 presents the interface we require our user to implement. It abstracts over the common update process and imposes nearly no restriction to the application. Besides, we equip our system with an RPC layer that decouples the backends, further relaxing the restriction on them. To demonstrate the generality Continuum and its ease of use, we have implemented several online learning applications in our evaluation (§5.1). While they span multiple programming languages and frameworks, each application was implemented using only tens of lines of code.

3.4 Customizable Policy

Users are able to implement customized update policies with the interfaces we provide. On one hand, if users want to access internal information (e.g., data amount, estimated training time) and decide by their own, we design an abstract policy class that users could extend with their own decision logic. On the other hand, if users have external information source, we expose REST APIs to let them manually trigger retraining. For instance, users may have some systems monitoring model quality or cluster status. When model starts to underperform or the cluster gets idle, they could call the API to update the model. Besides those customized situations, Continuum implements two general policies for two common scenarios, which is the main theme of the next section.

4 WHEN TO PERFORM UPDATING

In this section, we study when to update models for improving data recency. We define two metrics that respectively measure the latency of data incorporation and the training cost (§4.1). We show through empirical study that the training cost follows a linear model of the data size. We then present two online update policies, best-effort (§4.2) and cost-aware (§4.3), for fast data incorporation, with and without accounting for the training cost, respectively.

4.1 Model and Objective

Data Incorporation Latency Given an update policy, to quantify how fast the data can be incorporated into the model, we measure, for each sample, the latency between its arrival and the moment that sample gets incorporated.

Specifically, let \( m \) data samples arrive in sequence at time \( a_1, \ldots, a_m \). The system performs \( n \) model updates, where the \( i \)-th update starts at \( s_i \) and takes \( \tau_i \) time to complete. Let \( D_i \) be the set of data samples incorporated by the \( i \)-th update, i.e., those arrived after the \((i-1)\)-th update starts and before the \(i\)-th:

\[
D_i = \{ k \mid s_{i-1} \leq a_k < s_i \}.
\]

Here, we assume \( s_0 = 0 \). Since all samples in \( D_i \) get incorporated after the \(i\)-th update completes, the cumulative latency, denoted \( L_i \), is computed as

\[
L_i = \sum_{k \in D_i} s_i + \tau_i - a_k.
\]

(1)

Summing up \( L_i \) over all \( n \) updates, we obtain the data incorporation latency, i.e.,

\[
L = \sum_i L_i.
\]

(2)

Training Cost Updating models over new data incurs non-trivial training cost, which is directly measured by the machine-time in public, pay-as-you-go cloud or shared, in-house clusters. Without loss of generality, we assume in the following discussions that the training is performed in a single machine. In this case, the training cost of the \(i\)-th update is equivalent to the training time \( \tau_i \). The overall training cost is \( \sum_i \tau_i \).

To characterize the training cost, we resort to empirical studies and find that the cost is in proportion to the amount of data over which the training is performed. More precisely, the training cost \( \tau_i \) of the \(i\)-th update follows a linear model \( f(\cdot) \) against the data amount \( |D_i| \):

\[
\tau_i = f(|D_i|) = \alpha|D_i| + \beta,
\]

(3)

where \( \alpha \) and \( \beta \) are algorithm-specific parameters.
4.2 Best-Effort Policy

We start with a simple scenario where users seek fast data incorporation without concerning about the training cost. This typically applies to users who perform model updating using some dedicated machines, where the training cost becomes less of a concern.

A straightforward approach to reduce data incorporation latency is to continuously retrain the model, where the \((i+1)\)-th update follows immediately after the \(i\)-th, i.e., \(s_{i+1} = s_i + \tau_i\) for all \(i\). However, this simple approach falls short in the presence of the flash crowd data, e.g., a surging number of tweets followed by some breaking news. In case that the flash crowds arrive right after an update starts, they have to wait until the next update to get incorporated, which may take a long time. In fact, in our evaluation, we observed 34% longer latency using continuous update than using the optimal policy (§5.2).

To address this problem, we propose a simple policy that reacts to the surge of data by speculatively aborting the ongoing update and restarting the training to timely incorporate the flash crowd data. To determine when to abort and restart, we speculate, upon the arrival of new data, if doing so would result in reduced data incorporation latency.

Specifically, suppose that the \(i\)-th update is currently ongoing, in which \(D\) data samples are being incorporated. Let \(\delta\) be the time elapsed since the start of the \(i\)-th update. Assume \(B\) data samples arrive during this period. We could either (i) defer the incorporation of the \(B\) samples to the \((i+1)\)-th update, or (ii) abort the current update and retrain the model with \(D+B\) samples. Let \(L_{\text{defer}}\) and \(L_{\text{abort}}\) be the data incorporation latency of the two choices, respectively. We speculate \(L_{\text{defer}}\) and \(L_{\text{abort}}\) and restart the training if \(L_{\text{abort}} \leq L_{\text{defer}}\).

More precisely, let \(f(\cdot)\) be the linear model used to predict the training time. We have

\[
L_{\text{defer}} = Df(D) + B(f(D) + f(B) - \delta),
\]

where the first term accounts for the latency of the \(D\) samples covered by the \(i\)-th update, and second term accounts for the latency of the \(B\) samples covered by the \((i+1)\)-th.

On the other hand, if we abort the ongoing update and restart the training with \(D+B\) samples, we have

\[
L_{\text{abort}} = D(f(D+B) + \delta) + Bf(D+B),
\]

where the first term is the latency of the \(D\) samples and the second term the latency of the \(B\) samples. Plugging Equation 3 into the equations above, we shall abort and restart \((L_{\text{abort}} \leq L_{\text{defer}})\) if the following inequality holds:

\[
B\beta \geq D\alpha + (D+B)\delta.
\]

4.3 Cost-Aware Policy

For cost-sensitive users who cannot afford unlimited retraining, we propose a cost-aware policy for fast data incorporation at low training cost. We model the cost sensitivity of a user with a “knob” parameter \(w\), meaning, for every unit of training cost it spends, it expects the data incorporation latency to be reduced by \(w\). In this model, latency \(L_i\) and cost \(\tau_i\) are “exchangeable” and are hence unified as one objective, which we call latency-cost sum, i.e.,

\[
\min_{\{s_i\}} \sum_i L_i + w\tau_i.
\]

Our goal is to make the optimal update decisions \(\{s_i\}\) online, without assuming future data arrival.

Online Algorithm At each time instant \(t\), we need to determine if an update should be performed. A key concern in this regard is how many new data samples can be incorporated. Figure 7 gives a pictorial illustration of data generation over time. Assume that \(D\) samples have become available since the last training and will be incorporated if update is performed at time \(t\). The colored area (yellow in the left) illustrates the cumulative latency (Equation 1) of these \(D\) samples.

In case that this latency turns large, we may wish we could have performed another update earlier, say, at time
We search all possible \( t' \) shows the latency of these samples. The latency could have been reduced should another update perform earlier at \( t' < t \) (shown as the green area in the right).

\( t' < t \). Figure 7 (right) illustrates how this imaginary, early update could have reduced the latency (green area). We search all possible \( t' \) and compute the maximum latency reduction due to an imaginary update, which we call the regret. Our algorithm keeps track of the regret at every time instant \( t \), and triggers updating once the regret exceeds \( w \beta \) (Algorithm 1).

We explain the intuition of this threshold-based update policy as follows. While having an imaginary update earlier could have reduced the latency, the price paid is an increased training cost by \( \beta \). Referring back to Figure 7, as the training cost follows a linear model of data size (Equation 3), the cost incurred in the left and right figures are \( aD + \beta \) and \( aD + 2\beta \), respectively. To justify this additional cost \( \beta \), the latency reduction due to the imaginary update (i.e., regret) must be greater than \( w \beta \) (see Equation 7). Therefore, whenever the regret exceeds \( w \beta \), we should have performed an update earlier. Realizing this mistake, we now trigger an update as a late compensation. We note that similar algorithmic behaviors of late compensation have also been studied in the online algorithm literature [5, 46].

**2-competitiveness** To study the performance of our algorithm, we follow competitive analysis [5], which compares an online algorithm with its optimal offline counterpart and use the ratio between their results as a performance indicator. We first present its definition, then our analysis of the algorithm.

### Algorithm 1 Cost-Aware Policy

- \( a[1, \cdots, n] \): arrival time of untrained data
- \( a, \beta \): parameters in runtime model (Equation 3)
- \( w \): weight in minimization objective (Equation 7)

1. **function** \( \text{CalcLat}(i, j) \) \( \triangleright \) calculate latency
2. \( \tau \leftarrow \alpha(j-i+1) + \beta \) \( \triangleright \) estimate runtime
3. \( e \leftarrow \tau + a[j] \) \( \triangleright \) end time of training
4. **return** \( \sum_{k=i}^{j} e - a[k] \)
5. **function** \( \text{DecideUpdate} \)
6. \( l \leftarrow \text{CalcLat}(1, n) \)
7. **for all** \( k \in \{2, \cdots, n-1\} \) **do** \( \triangleright \) iterate over possible \( t' \)
8. \( l' \leftarrow \text{CalcLat}(1, k) + \text{CalcLat}(k+1, n) \)
9. **if** \( l - l' > w \beta \) **then** \( \triangleright \) estimate regret
10. **return** true
11. **return** false

Given an input instance \( I \), let \( \text{ALG}(I) \) denote the cost incurred by a deterministic online algorithm \( \text{ALG} \), and \( \text{OPT}(I) \) that given by the optimal offline algorithm.

**Definition 4.1.** We say algorithm \( \text{ALG} \) is \( k \)-competitive if there exists a constant \( a \) such that, for any input instance \( I \), the following inequality holds:

\[
\text{ALG}(I) \leq k \cdot \text{OPT}(I) + a.
\]

The \( k \)-competitiveness roughly guarantees that the algorithm \( \text{ALG} \) performs no worse than the optimum by \( k \) times.

Then let \( \text{ALG} \) denote cost-aware policy (Algorithm 1), and \( \text{OPT} \) the optimal offline policy. For the ease of proof, we present a lemma that relates the algorithm result to the optimal one.

**Lemma 4.2.** Between any two consecutive updating performed by \( \text{ALG} \), \( \text{OPT} \) will perform one updating.

**Proof.** Let \( t_1 \) and \( t_2 \) denote the times of two consecutive updating. Suppose \( \text{OPT} \) performs no updating between them. We can add one updating, which will increase training cost by \( w \beta \) and reduce the latency by at least \( w \beta \). The guarantee of latency reduction is because, otherwise, \( \text{ALG} \) will wait longer than \( t_2 \). After the addition of an extra updating, the result will perform no worse than the optimal one. Hence there exists an optimal sequence of updating which includes one updating between \( t_1 \) and \( t_2 \). \( \square \)
Then we present the main theorem regarding the competitive ratio of our algorithm.

**Theorem 4.3.** $ALG$ is 2-competitive.

**Proof.** Lemma 4.2 guarantees that for each updating performed by $ALG$, there is a corresponding one performed by $OPT$, so they will have same training time. For any input instance $I$, we decompose $ALG(I)$ into two parts, the latency-cost sum of $OPT$ and the difference of latency between $ALG$ and $OPT$.

$$ALG(I) = OPT(I) + \text{latency}(ALG\backslash OPT)$$

For convenience, we further decompose $OPT(I)$ into two parts, $time(OPT)$ and $latency(OPT)$

$$ALG(I) = time(OPT) + \text{latency}(OPT) + \text{latency}(ALG\backslash OPT)$$

Our algorithm operates on the basis that, for each updating in $OPT$, it can save the latency by at most $w\beta$. Otherwise, it will perform updating earlier to reduce the possible saving. Therefore, $\text{latency}(ALG\backslash OPT)$ will never exceed $time(OPT)$ because $w\beta$ constitutes only a fraction in each training time. Substituting the relation into the inequality, we have

$$ALG(I) = time(OPT) + \text{latency}(OPT) + \text{latency}(ALG\backslash OPT)$$

$$\leq 2 \cdot time(OPT) + \text{latency}(OPT)$$

$$\leq 2 \cdot OPT(I),$$

which proves the 2-competitiveness. $\square$

**Remarks** If we substitute Equation 3 and 1 into the objective Equation 7, after expansion and factoring, the objective becomes

$$\min_{\{s_i\}} \text{const.} + \sum_{k \in \{1, 2, \ldots\}}\left(\frac{w\beta}{\text{cost}} + \frac{a|D_k|^2}{\text{squared term}} + \sum_{j \in D_k} (s_k - d_j), \frac{\text{latency}}{\text{latency}}\right),$$

which can be viewed as an extension to the classic TCP Acknowledgment Problem [46]. The $\text{cost}$ corresponds to the acknowledgment cost and $\text{latency}$ accords with packet latency. Our objective differs due to the $\text{squared term}$. If we set $a$ to 0, then the problem degenerates to its canonical form. The $\text{squared term}$ could bear an interpretation that, for each batch of packets, we need to count total latency as well as pay extra cost proportional to its squared size. For problems with a similar setting, our algorithm can directly apply to them.

## 5 EVALUATION

We evaluate our implementation of Continuum and the two proposed algorithms through trace-driven simulations and EC2 deployment. The highlights are summarized as follows:

- Best-effort policy can achieve up to 15.2% reduction of latency compared with continuous update;
- Cost-aware policy saves hardware cost by up to 32% or reduces latency by up to 28% compared with periodic update;
- Continuum achieves 6x better model quality than the state-of-the-art system, Velox [24], in a representative online learning application.

### 5.1 Settings

**Testbed** We conduct all experiments in Amazon EC2 [6] with c5.4xlarge instance, each equipped with 16 vCPUs (Intel Xeon Platinum processors, 3.0 GHz), 32GB memory and 5 Gbps Ethernet links.

**Workloads** We choose three representative applications in Table 2: LDA, PPR and GBDT. They cover different learning types (supervised and unsupervised), multiple data formats (unstructured text, sparse graph, dense features, etc.) and ML frameworks. For each application, we randomly extract two 15-minute-long data traces from the datasets, and respectively use the two traces as input, e.g., LDA1 (PPR2) refers to LDA (PPR) against trace-1 (trace-2).

**Methodology** Our experiments are based on the scenario where data is continuously generated and fed to the system for model updating. To emulate real data generation, we send data according to their timestamps in the traces. Upon the receipt of the data, the system determines whether to trigger retraining based on the specified policy.

We also conduct simulation studies, in which the optimal offline algorithms are used as a baseline. These algorithms assume future data arrival and cannot be implemented in practice. The simulation follows a similar setting as the deployment except that, instead of performing real training, we simulate the training time with the linear model (Equation 3) using parameters profiled from the experiments.

### 5.2 Effectiveness of Algorithms

**Metrics** Throughout the evaluation, we use three metrics: (i) data incorporation latency defined in Equation 1, (ii) training cost, and (iii) latency-cost sum defined in Equation 7.

**Best-Effort Policy** We evaluate best-effort policy (Best-Effort) against continuous update (Continuous) and offline optimal policy (Opt). Continuous performs updates continuously, one after another. Opt is a dynamic programming algorithm. The detail is deferred to our technical report [76].
We evaluate cost-aware policies in two settings, real deployment and simulation. We define a cost-aware policy as one that minimizes training cost while achieving a certain level of latency. The optimal offline policy is the one that minimizes training cost and latency simultaneously. We compare cost-aware policies with a best-effort policy and a continuous update policy.

**Cost-Aware Policy**

We next evaluate cost-aware policies (CostAware) against two baselines, Opt and periodic update (Periodic). Periodic triggers retraining at intervals of a predefined period. We carefully choose two periods, one (PeriLat) resulting in the same latency as CostAware, and the other (PeriCost) leading to an equal training cost.

We start with simulations, in which we evaluate CostAware against Opt. Owing to the super-linear complexity of Opt [76], we ran two algorithms on 100-, 300- and 500-sample data traces randomly extracted from the datasets. Figure 10 depicts the latency-cost sum of CostAware, normalized by the minimum given by Opt. While in theory, CostAware can be 2x worse than the optimum (Theorem 7), it comes much closer to Opt in practice ($\leq 1.26x$ on average).

We next compare Periodic and CostAware in EC2 deployment. Figure 11a plots the training cost of CostAware and PeriLat. CostAware incurs 19% less training cost and the maximum saving reaches 32% (GBDT2). We attribute such cost efficiency to our algorithm adapting to dynamic data generation: few updates were performed in face of infrequent data arrival. Figure 11b compares the latencies incurred by CostAware and PeriCost. CostAware speeds up data incorporation by 20% on average (up to 28% for LDA2)—a result of its quick response to the flash crowd data.

**5.3 Micro-benchmark**

We evaluate our algorithms in a more controllable manner, where we feed synthetic data traces to Continuum...
and examine the behaviors of BestEffort (CostAware) against Continuous (Periodic). Figure 12 depicts the data arrival (blue lines) over time and the triggering of model updates by different algorithms (orange arrows).

We refer to Figures 12a and 12b to illustrate how BestEffort reacts to the flash crowds as opposed to Continuous. While both algorithms trigger update upon the arrival of the first data batch, BestEffort restarts the training to quickly incorporate the flash crowds coming hot on heels. In contrast, Continuous defers the incorporation of that data to the next update, resulting in longer latency.

We next compare CostAware with Periodic in Figures 12c and 12d, respectively, where we send data in three batches of increasing size. We observe increasingly faster incorporation of three data batches under CostAware (Figures 12c), suggesting that the regret reaches the threshold more quickly as more data arrives. On the contrary, Periodic triggers update at fixed intervals, regardless of data size or the training cost.

5.4 System Efficiency and Scalability

Methodology We stress-test Continuum in both single-instance and cluster environments with a large number of pseudo applications that emulate the training time based on the linear model (Equation 3 and Figure 6). We send data batches to each application every second, with batch size randomly sampled between 1 and 30.

Metrics We consider two metrics, (i) the response time, which measures how long it takes to process a request of data sending, and (ii) the decision time, which measures the time spent in making an update decision.

Figure 13a shows the performance measured in single-instance deployment. The response time grows linearly with the number of applications. The decision time proves the efficiency of our implementation, as well as the lock-free and asynchronous architecture. Figure 13b illustrates the performance measured in a 20-node cluster. We observe similar response time and decision time to that of a single instance, an indicator of linear scalability in cluster environments.

5.5 System Overhead

We now evaluate the overhead Continuum imposes on the backend, which encapsulates the user’s implementation of the learning algorithm in existing ML frameworks. We compare the training speed (samples per second) of a backend running in a standalone mode and in Continuum. We configured Continuum to use best-effort policy and continuously fed data to the standalone backend. Figure 14 shows the measured training speed. Across all three workloads, Continuum results in only 2% slowdown on average.
Finally, we compare Continuum with Velox [24], a model serving system closely integrated with Spark MLlib [60]. To our best knowledge, Velox is the only learning system that explicitly supports (offline) model updating.

**Methodology** We consider the scenario of movie recommendation with MovieLens dataset [40]. In both Velox and Continuum, we train the base model using 95% of data and dynamically feed the remaining 5% to the two systems in a 20-minute window. We deploy both systems on 5-node clusters. In Velox, we use ALS algorithm shipped with Spark MLlib. We set its algorithm parameters according to its recommendations. We keep it retraining to simulate the best-effort policy. In Continuum, we have implemented element-wise ALS algorithm (an online version of ALS) proposed in [42], set parameters as mentioned therein, and configured the best-effort policy.

**Metrics** Following [42], we measure the recommendation quality using three metrics, hit ratio (HR), Normalized Discounted Cumulative Gain (NDCG) and precision (PREC). The first checks whether the user’s interested movie appears in the recommendation list, and the last two quantify the relevance of the recommendation.

Figure 15 shows the comparison results, where Continuum outperforms Velox in all metrics. Such a performance gap is mainly attributed to Velox’s Spark lock-in, which restricts its choice to the ML algorithms shipped with Spark. As opposed to Velox, Continuum comes as a general-purpose system without such a limitation, allowing it to take a full advantage of an improved algorithm. Furthermore, we observe that all three metrics improve faster in Continuum than in Velox (e.g., linear regression of HR curves yields slopes of \(1.2 \times 10^{-3}\) in Continuum, as compared with \(4.8 \times 10^{-4}\) in Velox). This validates the benefit of online learning, which incorporates new data more quickly with improved data recency.

### 6 DISCUSSION

**Auto Scaling** Though in §5.4 we have shown the scalability of Continuum, the scaling process could further be automated. Due to the loosely-coupled architecture, it only requires users to replicate the system on multiple instances to achieve scalability. Therefore, they could directly leverage existing auto scaling solutions, be it cloud-based (e.g., AWS Auto Scaling, GCP Autoscaling Group) or in-house (e.g., Openstack Heat), together with any load balancing service (e.g., AWS Elastic Load Balancing, Google Cloud Load Balancing, Openstack Load Balancer).

**Fault Tolerance** There are two types of failures, system failures that involve the Continuum itself, and backend failures. To handle system failure and achieve high availability, we could deploy Continuum in cluster mode. We further utilize process manager (e.g., supervisord, systemd) to enable auto restart after failure, then the load balancing service will gloss over the failure from users’ perception. The backend failure will be detected and handled by the system as we maintain a periodic heartbeat with backends through RPC. Once a failed backend gets detected, we start a new one to replace it.

**Workflow Integration** Continuum facilitates the process of model retraining, but only after deploying can those updated models exert their influence. We take Clipper [23] as an example to illustrate how to integrate Continuum with model serving systems. After retraining the model, we export the updated model with the template provided by Clipper and deploy it as a serving container. Clipper will then detect it and route further prediction requests to the new model, thus completing the workflow. Furthermore, other deployment procedures can also be included. For instance, they could validate the model quality before deploying (as described in [10]) to ensure the prediction quality.

**Distributed Training** Continuum also supports backends that conduct distributed training which typically involves a master node and several slave nodes. Users could wrap the master in the backend abstraction, then let it programmtically spawn slaves to perform training. The master will be responsible for communicating with Continuum and managing the data.

**Data Management** Continuum accepts multiple types of data information when users need to inform Continuum of recent data arrival. They can either send raw data to the system, or merely the information about data. For large-sized data (e.g., image, audio), users could store the
raw data in some dedicated and optimized storage system, only sending the data location and size to Continuum. For small-sized data (e.g., user click), they could skip the intermediate storage and directly send raw data for better efficiency.

7 RELATED WORK

Stream Processing Systems Stream processing has been widely studied, both in single-machine [9, 17] and distributed settings [4, 68]. Several systems [64, 79, 88] and computation models [30, 59] have been proposed. Continuum can be broadly viewed as a stream processing system in a sense that it process continuously arriving data. Similar to [26] that studies the throughput and latency in stream processing, our algorithms can help streaming systems strike a balance between processing cost and latency.

Machine Learning Systems Machine learning systems have been a hot topic in both academia and industry. For traditional machine learning, Vowpal Wabbit [49] and Scikit-learn [67] are representative systems of the kind. In the era of deep learning, people have developed frameworks with special support for neural network operators, e.g., Caffe [45], Theano [74], Torch [22], TensorFlow [1], etc. An emerging trend is to distribute learning with the Parameter Server [51] architecture. The poster-child systems include MXNet [19] and Bosen [83]. However, none of previous systems have provided explicit support for online learning. Therefore Continuum is orthogonal to them and complement them.

Stream Mining Systems Besides machine learning, data streams also attract attention from data mining community. A bunch of data mining algorithms have been adapted to data streams, e.g., decision trees [28, 36], pattern discovery [34, 72], clustering [3, 38]. Various systems have been built to carry out stream mining, e.g., MOA [11], SAMOA [63] and StreamDM [12]. However, they also focus on implementing algorithms, neglecting the updating and deployment of the models. Therefore, our system is applicable to them by extending their functionality.

Model Serving Systems As the final step of machine learning workflow, model serving has not received adequate attention until recently. Researchers have been built various systems to unify and optimize the process of model serving, e.g., Velox [24], Clipper [23], TensorFlow Serving [66] and SERF [85]. Continuum can be integrated with those systems to continuously improve the model quality by incorporating fresh data.

8 CONCLUSION

In this paper, we have presented Continuum, the first general-purpose platform for continual learning, which automates the updating process and judiciously decides when to retrain with different policies. To enable fast data incorporation, we have proposed best-effort policy that achieves low latency by speculatively restarting update in face of flash crowd. For users concerning about training cost, we have further designed cost-aware policy, an online algorithm jointly minimizing latency-cost sum with proven 2-competitiveness. Evaluations show that Continuum outperforms existing systems with significantly improved model quality.

REFERENCES


[34] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S Yu. 2003. Mining frequent patterns in data streams at multiple time granularities. Next generation data mining 212 (2003), 191–212.


[38] Michael Hahsler and Matthew Bolaños. 2016. Clustering Data Streams Based on Shared Density between Micro-Clusters. IEEE Transactions on Knowledge and Data Engineering 28 (2016), 1449–1461.


A OPTIMAL OFFLINE POLICIES

A.1 Best-Effort Policy

We first formulate the offline version of the problem. Following the notation in §4.1, we suppose \( n \) data samples have arrived at \( a_1, \ldots, a_n \) respectively, and our goal is to find optimal starting times \( s_i \)'s of updating so that the total data incorporation latency \( L \) is minimized.

Since our simulation only concerns the minimal latency instead of actual updating times, we consider a variant problem: for all possible combinations of starting times, find the minimal data incorporation latency over all data samples. We solve it with the technique of dynamic programming. Define \( M_{i,j} \) to be the minimal latency over data samples arriving after \( a_i \), under the constraint that first updating happens no earlier than \( a_j \). It corresponds to the situation where not until the \( j \)-th data sample arrives has the previous training finished. Then the problem is equivalent to finding \( M_{n,1} \). Our observation is that, when \( j < k \),

\[
M_{i,j} = \min_{1 \leq k \leq n} \{ \text{Lat}(i,k) + M_{k,j'} \}, \tag{8}
\]

where \( \text{Lat}(i,k) \) is the latency incurred by the \( i \)-th to \( k \)-th data samples if we train them in one batch, and \( j' \) is the index of next data arrival that immediately follows the end of the training. The equation enumerates all possible partitioning among data samples and each partition corresponds to one training.

We present Algorithm 2 which implements the idea behind Equation 8. For the case where the training ends after all data arrival, we set \( j' \) to \( n + 1 \) and suppose all remaining data is trained in one batch.

A.2 Cost-Aware Policy

For cost-aware policy, its optimal offline counterpart closely resembles that of best-effort policy. We also leverage dynamic programming to find the minimal latency-cost sum for all possible combinations of start times. We present Algorithm 3, which is similar to Algorithm 2 except that the calculation of data incorporation latency is replaced with that of latency-cost sum.
Algorithm 2 Optimal Best-Effort Policy

- \( a[1 \cdots n] \): arrival time of untrained data
- \( \alpha, \beta \): parameters in runtime model (Equation 3)

1: function \( \text{CalcLat}(i, k) \)
2: \( \tau \leftarrow a(k-i+1) + \beta \)
3: \( e \leftarrow a[k] + \tau \)
4: return \( \sum_{m=i}^{k} e - a[m] \)

5: function \( \text{FindNext}(i, k) \)
6: \( \tau \leftarrow a(k-i+1) + \beta \)
7: \( e \leftarrow a[k] + \tau \)
8: for all \( m \in \{k+1, \cdots, n\} \) do
9: if \( a[m] \geq e \) then
10: return \( m \)
11: return \( n+1 \) \( \triangleright \) training ends after all data arrival

12: function \( \text{MinLat}(i, j) \) \( \triangleright \) compute \( M_{i,j} \)
13: if \( j \geq n \) then
14: return \( \text{CalcLat}(i, n) \)
15: res \( \leftarrow \infty \)
16: for all \( k \in \{j, \cdots, n\} \) do
17: \( j' \leftarrow \text{FindNext}(i, k) \)
18: \( \text{res} \leftarrow \min(\text{res}, \text{CalcLat}(i, k) + \text{MinLat}(k, j')) \)
19: return \( \text{res} \)

Algorithm 3 Optimal Cost-Aware Policy

- \( a[1 \cdots n] \): arrival time of untrained data
- \( \alpha, \beta \): parameters in runtime model (Equation 3)
- \( w \): weight in minimization objective (Equation 7)

1: function \( \text{CalcSum}(i, k) \)
2: \( \tau \leftarrow a(k-i+1) + \beta \)
3: \( e \leftarrow a[k] + \tau \)
4: return \( we + \sum_{m=i}^{k} e - a[m] \)

5: function \( \text{MinSum}(i, j) \)
6: if \( j \geq n \) then
7: return \( \text{CalcSum}(i, n) \)
8: res \( \leftarrow \infty \)
9: for all \( k \in \{j, \cdots, n\} \) do
10: \( j' \leftarrow \text{FindNext}(i, k) \)
11: \( \text{res} \leftarrow \min(\text{res}, \text{CalcSum}(i, k) + \text{MinSum}(k, j')) \)
12: return \( \text{res} \)