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

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

Submission Type: Research

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 *continually* 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

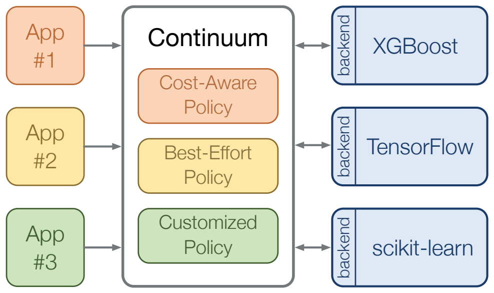


Figure 1: Overview of Continuum.

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-specified *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 $\mathcal{S} = (s_1, s_2, \dots)$ 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 θ, that correctly predicts label y for each instance x. To this end, ML algorithms *iteratively* update model parameter θ 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 $f_b(\cdot)$ over all samples. That is, in each iteration k, it updates the model parameter as

$$\theta_{k+1} \leftarrow f_b(\theta_k, \mathcal{S}).$$

Compared with offline learning, model updating in online learning is much more lightweight as it only incorporates one data sample a time.[1] 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

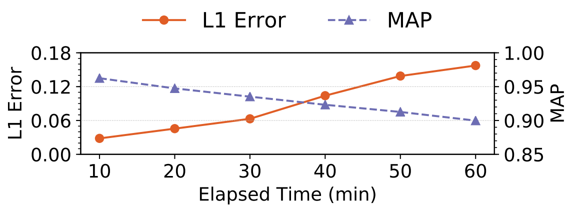
Methodology We study three popular online learning applications based on Personalized PageRank [8], topic modeling [44], and classification prediction [32], respectively. For each application, we replay real-world traces containing the time series of dynamic data generation. We train a *base model* using 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.

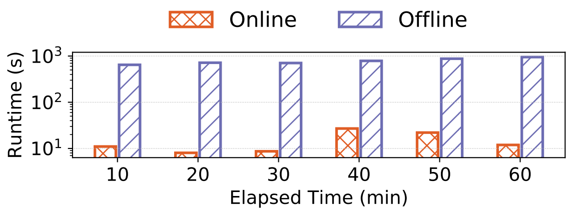
Figure 2a illustrates how the quality of the base model deteriorates over time compared with the updated models. Specifically, we consider two metrics, L1 error [89] and Mean Average Precision (MAP) [7, 20]. The former measures how far the recommendation given by the base model deviates from that given 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

[1]In practice, model updating is usually performed over a small batch of new data, as per-sample updating can be expensive to implement.



(a) L1 error (lower is better) and MAP (higher is better) of the base model quickly deteriorate over time.



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

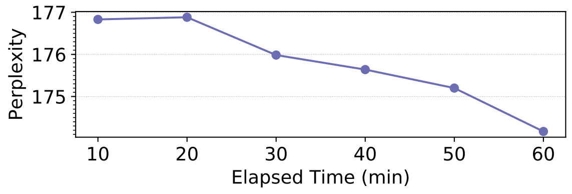
Figure 2: Results of Personalized PageRank algorithm on Twitter dataset.

available data, it is orders of magnitude slower than online learning, making it a poor fit in dynamic environments.

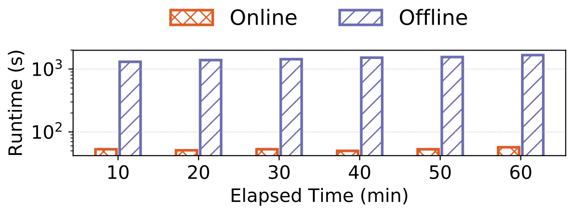
Topic Modeling We next turn to topic modeling [44], a popular data mining tool [16, 77, 82, 87] that automatically detects themes from text corpora. We implemented a topic trend monitor in MALLET [57], a natural language processing toolkit offering an efficient implementation of Latent Dirichlet Allocation (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 feed 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

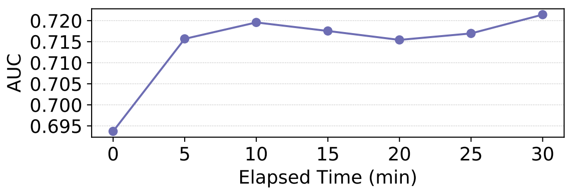


(a) Perplexity of LDA model decreases as more data are incorporated into the model, meaning better performance.

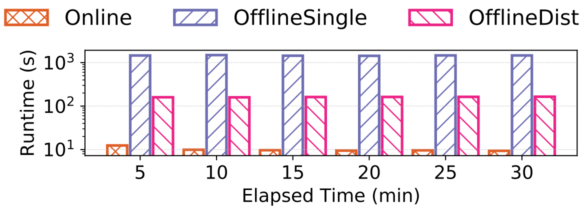


(b) Online and offline training time of LDA models.

Figure 3: Results of LDA algorithm on Twitter dataset.



(a) AUC of GBDT models rises as more data are included, meaning better overall prediction performance.



(b) Online and offline training time of GBDT models. Offline settings include a single machine (OfflineSingle) and a 5-node cluster (OfflineDist).

Figure 4: Results of GBDT algorithm on Criteo dataset.

(GBDT) algorithm [32], along with its scalable implementation, XGBoost [18].

We evaluate the model with *area under an ROC curve* (AUC) [39], a metric that quantifies the overall prediction performance of a classifier (larger is better). Figure 4a depicts the improvement of online learning that updates the model every 5 minutes. We also compare the training time of online learning with offline retraining in Figure 4b. In addition to performing offline retraining on a single machine (*OfflineSingle*), we further scale out the training to five `c5.4xlarge` instances (*OfflineDist*). Despite using more machines, offline retraining remains lagging far behind online learning.

Complexity in Practice Our case studies have highlighted the benefit of maintaining *data recency* [29, 42] through continual learning. However, realizing this benefit in practice can be *complex*. As system support for continual learning remain lacking in many existing ML frameworks, we had to write our own training loop to manually feed the new data to the training instance and periodically trigger model updates there, all in custom scripts. This includes much *tedious, repetitive labor work*. In our case studies, we have repeated the implementation of the common training loop across all the three applications, using Scala in PPR, Java in MALLET [57], and C++ in XGBoost [18]. In fact, this has taken us even more coding efforts than implementing the key logic of online learning. As evidenced in Table 1, compared with the model updating scripts, the training loop takes 5-12x lines of code in our implementation, meaning much effort has been wasted reinventing the wheel.

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

Algorithm	Training Loop	Model Updating
PPR [89]	211	41
LDA [13]	377	56
GBDT [32]	558	44

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.

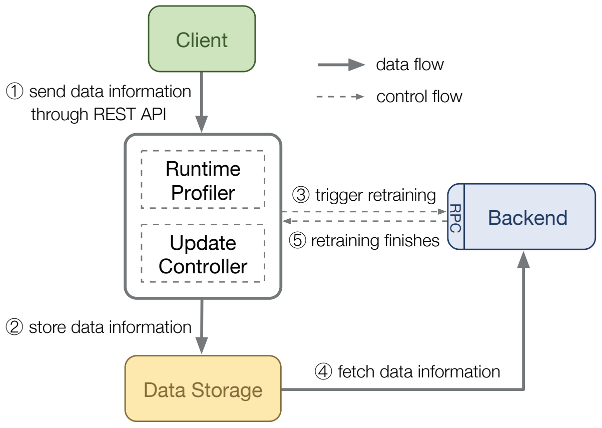


Figure 5: The training loop in Continuum.

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 retrains 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.

Listing 1 Common interface for backends.

```
interface Backend<X, Y> {
    X retrain(X prev_model, List<Y> data)
}
```

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 $a_1, \cdots, a_m$. The system performs n model updates, where the i-th update starts at s_i and takes τ_i time to complete. Let $\mathcal{D}_i$ 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:

$$\mathcal{D}_i = \{ k \mid s_{i-1} \le a_k < s_i \}.$$

Here, we assume $s_0 = 0$. Since all samples in $\mathcal{D}_i$ get incorporated after the i-th update completes, the *cumulative latency*, denoted L_i, is computed as

$$L_i = \sum_{k \in \mathcal{D}_i} s_i + \tau_i - a_k. \tag{1}$$

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

$$L = \sum_i L_i. \tag{2}$$

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 τ_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 τ_i of the i-th update follows a *linear model* $f(\cdot)$ against the data amount $|\mathcal{D}_i|$:

$$\tau_i = f(|\mathcal{D}_i|) = \alpha|\mathcal{D}_i| + \beta, \tag{3}$$

where α and β are *algorithm-specific* parameters.

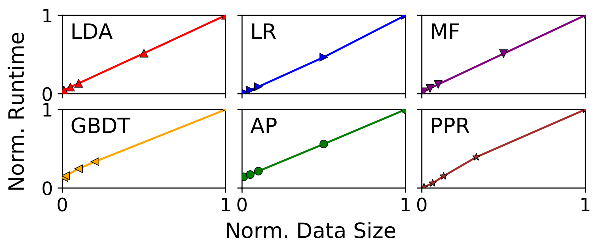


Figure 6: The relationship between runtime and data size (both normalized to $[0, 1]$) is approximately linear for various algorithms and datasets.

We attribute the linear relationship to the fact that most of existing online learning algorithms have *no nested loops* but simply scan through the data samples once or multiple times. To support this claim, we have implemented a number of online learning algorithms spanning multiple ML frameworks, programming languages, and application domains (Table 2). For each algorithm, we measure the training time against varying data sizes and depict the results in Figure 6. Regressions based on *linear least squares* show that the *correlation coefficient* r falls within 0.995 ± 0.005 across all algorithms—an indicator of strong linearity.

Objective Given the two metrics above, our goal is to determine when to perform updating (i.e., $s_1, s_2, \ldots$) to minimize data incorporation latency L at low training cost $\sum_i \tau_i$.

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 δ 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_{defer} and L_{abort} be the data incorporation latency of the two choices, respectively. We speculate L_{defer} and L_{abort} and restart the training if $L_{\text{abort}} \le L_{\text{defer}}$.

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

$$L_{\text{defer}} = Df(D) + B(f(D) + f(B) - \delta), \quad (4)$$

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

$$L_{\text{abort}} = D(f(D + B) + \delta) + Bf(D + B), \quad (5)$$

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 ($L_{\text{abort}} \le L_{\text{defer}}$) if the following inequality holds:

$$B\beta \ge D\alpha B + (D + B)\delta. \quad (6)$$

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 τ_i are "exchangeable" and are hence unified as one objective, which we call *latency-cost sum*, i.e.,

$$\text{minimize}_{\{s_i\}} \quad \sum_i L_i + w\tau_i. \quad (7)$$

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

Table 2: Summary of machine learning algorithms and datasets used in experiments.

Algorithm	Abbreviation	Framework	Programming Language	Dataset
Latent Dirichlet Allocation [65]	LDA	MALLET [57]	Java	Twitter [86]
adPredictor [37]	AP	TensorFlow [1]	C++, Python	Criteo [47]
Matrix Factorization [73]	MF	Infer.NET [61]	C#	MovieLens [40]
Gradient-Boosted Decision Tree [32]	GBDT	XGBoost [18]	C++	Criteo
Personalized PageRank [89]	PPR	/	Scala	Higgs [27]
Logistic Regression	LR	Scikit-learn [67]	Python	Criteo

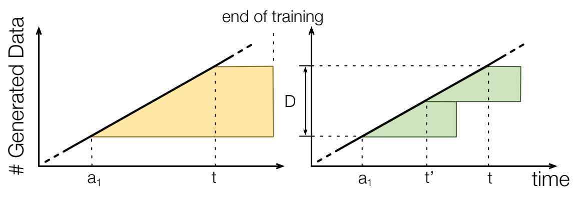


Figure 7: A pictorial illustration of dynamic data generation. Assume that D samples could be incorporated by an update at time t. The yellow area (left) shows the latency of these samples. The latency could have been reduced should another update perform earlier at $t' < t$ (shown as the green area 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 β. 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 $\alpha D + \beta$ and $\alpha D + 2\beta$, respectively. To justify this additional cost β, 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
    – $\alpha, \beta$: parameters in runtime model (Equation 3)
    – $w$: weight in minimization objective (Equation 7)
 1: function CalcLat($i, j$)                                  ▷ calculate latency
 2:     $\tau \leftarrow \alpha(j - i + 1) + \beta$                          ▷ estimate runtime
 3:     $e \leftarrow \tau + a[j]$                                  ▷ end time of training
 4:     return $\sum_{k=i}^{j} e - a[k]$
 5: function DecideUpdate
 6:     $l \leftarrow$ CalcLat($1, n$)
 7:     for all $k \in \{2, \cdots, n-1\}$ do      ▷ iterate over possible $t'$
 8:         $l' \leftarrow$ CalcLat($1, k$) + CalcLat($k+1, n$)
 9:         if $l - l' > w\beta$ then                          ▷ estimate regret
10:             return true
11:     return false
```

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

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

$$ALG(I) \leq k \cdot OPT(I) + a.$$

The k-competitiveness roughly guarantees that the algorithm ALG performs no worse than the optimum by k times.

Then let ALG denote cost-aware policy (Algorithm 1), and 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 ALG, OPT will perform one updating.*

Proof. Let t_1 and t_2 denote the times of two consecutive updating. Suppose 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, 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. □

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*.

$$ALG(I) = OPT(I) + latency(ALG \backslash OPT)$$

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

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

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, $latency(ALG \backslash OPT)$ will never exceeds $time(OPT)$ because $w\beta$ constitutes only a fraction in each training time. Substituting the relation into the inequality, we have

$$\begin{aligned} ALG(I) &= time(OPT) + latency(OPT) + latency(ALG \backslash OPT) \\ &\leq 2 \cdot time(OPT) + latency(OPT) \\ &\leq 2 \cdot OPT(I), \end{aligned}$$

which proves the 2-competitiveness. □

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

$$\min_{\{s_i\}} \quad \text{const.} + \sum_{k \in \{1,2,\cdots\}} \Big(\underbrace{w\beta}_{cost} + \underbrace{\alpha |D_k|^2}_{squared\ term} + \underbrace{\sum_{j \in D_k} (s_k - d_j)}_{latency} \Big),$$

which can be viewed as an extension to the classic *TCP Acknowledgment Problem* [46]. The *cost* corresponds to the acknowledgment cost and *latency* accords with packet latency. Our objective differs due to the *squared term*. If we set α to 0, then the problem degenerates to its canonical form. The *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].

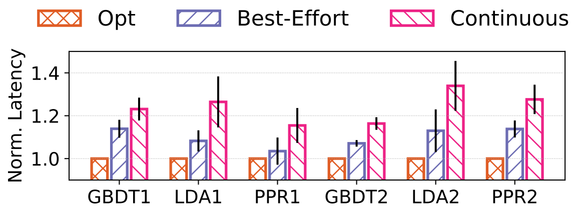


Figure 8: [simulation] Latencies of optimal offline policy, continuous update and best-effort policy. We normalize the latency for each workload against its optimal one.

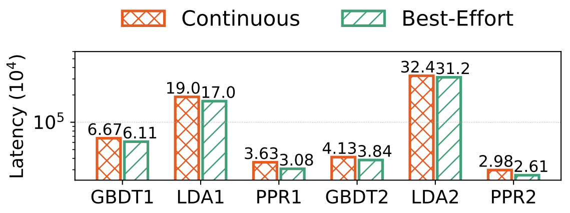


Figure 9: [deployment] Latencies of continuous update and best-effort policy.

Figure 8 compares three algorithms in simulation. Due to the super-linear complexity of *Opt*, we randomly extract several 100-sample data traces from the datasets and use them as input. We observe that the latency incurred by *Continuous* is on average 23% (up to 34% in LDA2) longer than that of *Opt*, a result of delayed incorporation of the flash crowd data. *BestEffort* effectively addresses this problem and comes closer to *Opt*, resulting in 11% shorter latency than *Continuous* on average.

We further evaluate *Continuous* and *BestEffort* in real deployment, and depict the results in Figure 9. Compared with *Continuous*, *BestEffort* reduces latency by 9.5% on average (up to 15.2% in PPR1).

While these improvements seem relatively insignificant in numbers, we still consider them practically valuable in that shorter data latency means that users could enjoy improved model earlier (§2.2). Given that machine learning has been widely deployed in many billion-dollar applications (e.g., massive online advertising), it is broadly accepted that even a marginal quality improvement can turn in significant revenue [52, 81].

Cost-Aware Policy We next evaluate cost-aware policy (*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 **??**), it comes much closer to *Opt* in practice (≤ 1.26x on average).

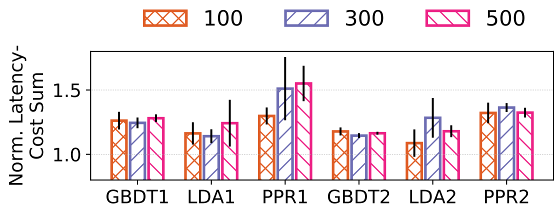
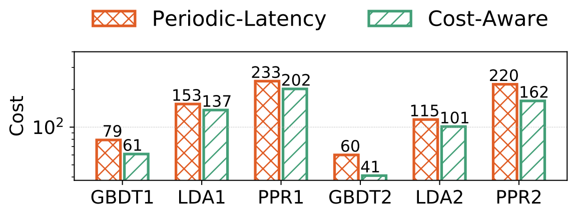
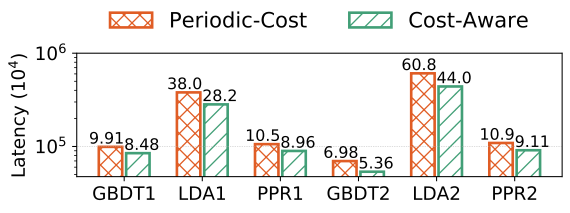


Figure 10: [simulation] Latency-cost sum of cost-aware policy, normalized by the minimum given by the optimal offline policy.



(a) The cost incurred by cost-aware policy and periodic update under the condition of same latency.



(b) The latency incurred by cost-aware policy and periodic update under the condition of same cost.

Figure 11: [deployment] Cost and latency of periodic update and cost-aware policy.

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

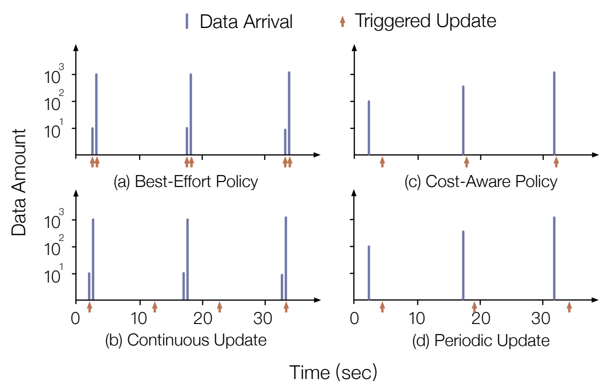
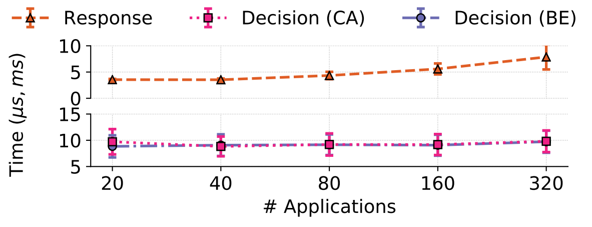
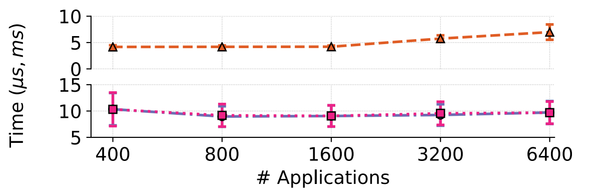


Figure 12: Micro-benchmark of algorithm behaviors. We compare (a) best-effort with (b) continuous update, and (c) cost-aware with (d) periodic update.



(a) Single-instance deployment.



(b) Cluster deployment on a 20-node cluster.

Figure 13: System performance measured by: (1) response time of data sending; (2) decision time using best-effort (BE) policy and cost-aware (CA) policy respectively.

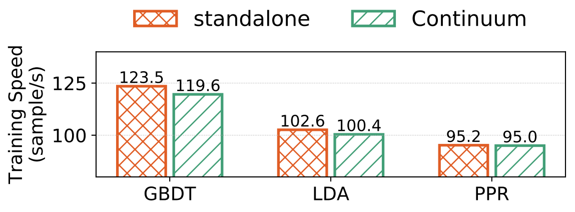


Figure 14: A standalone backend and Continuum yield approximately the same training speeds, meaning little overhead on the backend.

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 the measured training speed. Across all three workloads, Continuum results in only 2% slowdown on average.

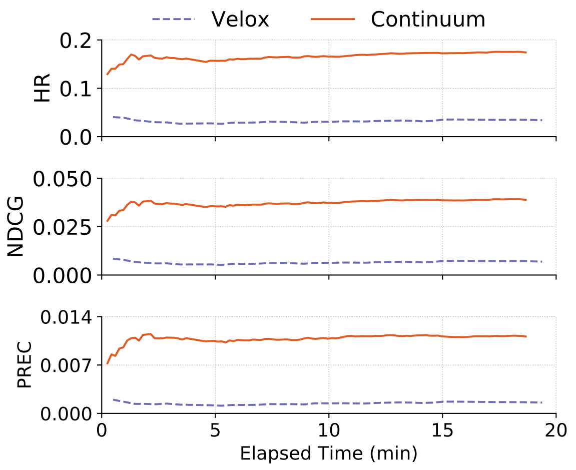


Figure 15: Results of comparison with Velox in movie recommendation.

5.6 Comparison with the Status Quo

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 improves faster in Continuum than in Velox (e.g., linear regression of HR curves yields slopes of 1.2×10^{-3} in Continuum, as compared with 4.8×10^{-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 perioidc 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 Bosën [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 leaning 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

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*.

[2] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, et al. 2016. Making Contextual Decisions with Low Technical Debt. *arXiv preprint arXiv:1606.03966* (2016).

[3] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. 2003. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 81–92.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[5] Susanne Albers. 2003. Online algorithms: a survey. *Mathematical Programming* 97, 1 (01 Jul 2003), 3–26. https://doi.org/10.1007/s10107-003-0436-0

[6] Amazon AWS. 2018. Amazon EC2. (2018). https://aws.amazon.com/ec2/

[7] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. 1999. *Modern information retrieval*. Vol. 463. ACM press New York.

[8] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast Incremental and Personalized PageRank. *Proc. VLDB Endow.* 4, 3 (Dec. 2010), 173–184. https://doi.org/10.14778/1929861.1929864

[9] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2005. Fault-tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1066157.1066160

[10] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, USA, 1387–1395. https://doi.org/10.1145/3097983.3098021

[11] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. 2010. MOA: Massive Online Analysis. *Journal of Machine Learning Research* 11 (2010), 1601–1604.

[12] Albert Bifet, Silviu Maniu, Jianfeng Qian, Guangjian Tian, Cheng He, and Wei Fan. 2015. StreamDM: Advanced Data Mining in Spark Streaming. *2015 IEEE International Conference on Data Mining Workshop (ICDMW)* (2015), 1608–1611.

[13] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[14] Léon Bottou and Yann L Cun. 2004. Large scale online learning. In *Advances in neural information processing systems*. 217–224.

[15] Tamara Broderick, Nicholas Boyd, Andre Wibisono, Ashia C. Wilson, and Michael I. Jordan. 2013. Streaming Variational Bayes. In *NIPS*.

[16] Michael Cafarella, Edward Chang, Andrew Fikes, Alon Halevy, Wilson Hsieh, Alberto Lerner, Jayant Madhavan, and S Muthukrishnan. 2008. Data management projects at Google. *ACM SIGMOD Record* 37, 1 (2008), 34–38.

[17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 668–668. https://doi.org/10.1145/872757.872857

[18] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[20] Wei Chen, Tie yan Liu, Yanyan Lan, Zhi ming Ma, and Hang Li. 2009. Ranking Measures and Loss Functions in Learning to Rank. In *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta (Eds.). Curran Associates, Inc., 315–323. http://papers.nips.cc/paper/3708-ranking-measures-and-loss-functions-in-learning-to-rank.pdf

[21] Wei Chu, Martin Zinkevich, Lihong Li, Achint Thomas, and Belle Tseng. 2011. Unbiased online active learning in data streams. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 195–203.

[22] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

[23] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *NSDI*. 613–627.

[24] Joseph E. Gonzalez Haoyuan Li Zhao Zhang Michael J. Franklin Ali Ghodsi Michael I. Jordan Daniel Crankshaw, Peter Bailis. 2015. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*.

[25] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. 2007. Google News Personalization: Scalable Online Collaborative Filtering. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. ACM, New York, NY, USA, 271–280. https://doi.org/10.1145/1242572.1242610

[26] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2670979.2670995

[27] Manlio De Domenico, Antonio Lima, Paul Mougel, and Mirco Musolesi. 2013. The anatomy of a scientific rumor. *Scientific reports* 3 (2013), 2980.

[28] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 71–80.

[29] Anlei Dong, Ruiqiang Zhang, Pranam Kolari, Jing Bai, Fernando Diaz, Yi Chang, Zhaohui Zheng, and Hongyuan Zha. 2010. Time is of the Essence: Improving Recency Ranking Using Twitter Data. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 331–340. https://doi.org/10.1145/1772690.1772725

[30] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning Fast Iterative Data Flows. *Proc. VLDB Endow.* 5, 11 (July 2012), 1268–1279. https://doi.org/10.14778/2350229.2350245

[31] Tom Fawcett and Foster Provost. 1997. Adaptive fraud detection. *Data mining and knowledge discovery* 1, 3 (1997), 291–316.

[32] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[33] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* 46, 4, Article 44 (March 2014), 37 pages. https://doi.org/10.1145/2523813

[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.

[35] Todd R Golub, Donna K Slonim, Pablo Tamayo, Christine Huard, Michelle Gaasenbeek, Jill P Mesirov, Hilary Coller, Mignon L Loh, James R Downing, Mark A Caligiuri, et al. 1999. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *science* 286, 5439 (1999), 531–537.

[36] Heitor Murilo Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. 2017. Adaptive random forests for evolving data stream classification. *Machine Learning* 106 (2017), 1469–1495.

[37] Thore Graepel, Joaquin Q Candela, Thomas Borchert, and Ralf Herbrich. 2010. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 13–20.

[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.

[39] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* 143, 1 (1982), 29–36.

[40] F Maxwell Harper and Joseph A Konstan. 2016. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4 (2016), 19.

[41] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. ACM, 1–9.

[42] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. 2016. Fast Matrix Factorization for Online Recommendation with Implicit Feedback. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '16)*. ACM, New York, NY, USA, 549–558. https://doi.org/10.1145/2911451.2911489

[43] Matthew Hoffman, Francis R Bach, and David M Blei. 2010. Online learning for latent dirichlet allocation. In *advances in neural information processing systems*. 856–864.

[44] Thomas Hofmann. 1999. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 50–57.

[45] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[46] Anna R. Karlin, Claire Kenyon, and Dana Randall. 2001. Dynamic TCP Acknowledgement and Other Stories About e/(e-1). In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing (STOC '01)*. ACM, New York, NY, USA, 502–509. https://doi.org/10.1145/380752.380845

[47] Criteo Labs. 2018. Search Conversion Log Dataset. (2018). http://research.criteo.com/criteo-sponsored-search-conversion-log-dataset/

[48] Wai Lam and Javed Mostafa. 2001. Modeling user interest shift using a bayesian approach. *JASIST* 52 (2001), 416–429.

[49] John Langford, Lihong Li, and Alex Strehl. 2007. Vowpal wabbit online learning project. (2007).

[50] Cheng Li, Yue Lu, Qiaozhu Mei, Dong Wang, and Sandeep Pandey. 2015. Click-through prediction for advertising in twitter timeline. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1959–1968.

[51] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 1. 3.

[52] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, Feng Sun, and Hucheng Zhou. 2017. Model Ensemble for Click Prediction in Bing Search Ads. ACM. https://www.microsoft.com/en-us/research/publication/model-ensemble-click-prediction-bing-search-ads/

[53] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. 2016. Personalized PageRank Estimation and Search: A Bidirectional Approach. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining (WSDM '16)*. ACM, New York, NY, USA, 163–172. https://doi.org/10.1145/2835776.2835823

[54] Peter A. Lofgren, Siddhartha Banerjee, Ashish Goel, and C. Seshadhri. 2014. FAST-PPR: Scaling Personalized Pagerank Estimation for Large Graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 1436–1445. https://doi.org/10.1145/2623330.2623745

[55] Julien Mairal, Francis R. Bach, Jean Ponce, and Guillermo Sapiro. 2010. Online Learning for Matrix Factorization and Sparse Coding. *Journal of Machine Learning Research* 11 (2010), 19–60.

[56] Christopher D Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press.

[57] Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit. (2002).

[58] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. 2013. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1222–1230.

[59] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow. In *Proceedings of CIDR 2013*. https://www.microsoft.com/en-us/research/publication/differential-dataflow/

[60] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Bosagh Zadeh, Matei Zaharia, and Ameet S. Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17 (2016), 34:1–34:7.

[61] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. Infer.NET 2.7. (2018). http://research.microsoft.com/infernet

[62] C Monteiro, R Bessa, V Miranda, A Botterud, J Wang, G Conzelmann, et al. 2009. *Wind power forecasting: state-of-the-art 2009.* Technical Report. Argonne National Laboratory (ANL).

[63] Gianmarco De Francisci Morales and Albert Bifet. 2015. SAMOA: scalable advanced massive online analysis. *Journal of Machine Learning Research* 16, 1 (2015), 149–153.

[64] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[65] David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. 2009. Distributed Algorithms for Topic Models. *J. Mach. Learn. Res.* 10 (Dec. 2009), 1801–1828. http://dl.acm.org/citation.cfm?id=1577069.1755845

[66] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[67] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jacob VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[68] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2465351.2465353

[69] RedisLab. 2018. Redis. https://redis.io. (2018).

[70] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 342–358. https://doi.org/10.1145/3132747.3132771

[71] Jeffrey C. Schlimmer and Richard Granger. 1986. Beyond Incremental Processing: Tracking Concept Drift. In *AAAI*.

[72] Andreia Silva and Cláudia Antunes. 2015. Multi-relational pattern mining over data streams. *Data Mining and Knowledge Discovery* 29, 6 (2015), 1783–1814.

[73] David H Stern, Ralf Herbrich, and Thore Graepel. 2009. Matchbox: large scale online bayesian recommendations. In *Proceedings of the 18th international conference on World wide web*. ACM, 111–120.

[74] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[75] Huangshi Tian, Minchen Yu, and Wei Wang. 2018. Continuum. (2018). https://github.com/All-less/continuum

[76] Huangshi Tian, Minchen Yu, and Wei Wang. 2018. Continuum: A Platform for Cost-Aware, Low-Latency Continual Learning. (May 2018). https://www.cse.ust.hk/~weiwa/papers/continuum-socc18.pdf

[77] Ivan Titov and Ryan McDonald. 2008. Modeling online reviews with multi-grain topic models. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 111–120.

[78] Alexey Tsymbal. 2004. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin* 106, 2 (2004).

[79] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3132747.3132750

[80] Hanna M Wallach, Iain Murray, Ruslan Salakhutdinov, and David Mimno. 2009. Evaluation methods for topic models. In *Proceedings of the 26th annual international conference on machine learning*. ACM, 1105–1112.

[81] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. *CoRR* abs/1708.05123 (2017). arXiv:1708.05123 http://arxiv.org/abs/1708.05123

[82] Yi Wang, Xuemin Zhao, Zhenlong Sun, Hao Yan, Lifeng Wang, Zhihui Jin, Liubin Wang, Yang Gao, Ching Law, and Jia Zeng. 2015. Peacock: Learning long-tail topic features for industrial applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 6, 4 (2015), 47.

[83] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 381–394.

[84] Gerhard Widmer and Miroslav Kubat. 1996. Learning in the presence of concept drift and hidden contexts. *Machine Learning* 23 (1996), 69–101.

[85] Feng Yan, Yuxiong He, Olatunji Ruwase, and Evgenia Smirni. 2016. SERF: Efficient Scheduling for Fast Deep Neural Network Serving via Judicious Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 26, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014939

[86] Jaewon Yang and Jure Leskovec. 2011. Patterns of temporal variation in online media. In *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 177–186.

[87] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. 2015. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1351–1361.

[88] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*.

[89] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate Personalized PageRank on Dynamic Graphs. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 1315–1324. https://doi.org/10.1145/2939672.2939804

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 $a_1, \cdots, a_n$ 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 $j < k$,

$$M_{i,j} = \min_{j \le k \le n} \{Lat(i,k) + M_{k,j'}\}, \tag{8}$$

where $Lat(i,k)$ is the latency incurred by the i- 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 CalcLat($i, k$)
2:     $\tau \leftarrow \alpha(k - i + 1) + \beta$
3:     $e \leftarrow a[k] + \tau$
4:     return $\sum_{m=i}^{k} e - a[m]$
5: function FindNext($i, k$)
6:     $\tau \leftarrow \alpha(k - i + 1) + \beta$
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 $n + 1$        ▹ training ends after all data arrival
12: function MinLat($i, j$)                    ▹ compute $M_{i,j}$
13:    if $j \geq n$ then                ▹ train all data in one batch
14:        return CalcLat($i, n$)
15:    $res \leftarrow \infty$
16:    for all $k \in \{j, \cdots, n\}$ do
17:        $j' \leftarrow$ FindNext($i, k$)
18:        $res \leftarrow \min\{res,$ CalcLat($i, k$) + MinLat($k, j'$)$\}$
19:    return $res$
```

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 CalcSum($i, k$)
2:     $\tau \leftarrow \alpha(k - i + 1) + \beta$
3:     $e \leftarrow a[k] + \tau$
4:     return $w\tau + \sum_{m=i}^{k} e - a[m]$
5: function MinSum($i, j$)
6:     if $j \geq n$ then
7:         return CalcSum($i, n$)
8:     $res \leftarrow \infty$
9:     for all $k \in \{j, \cdots, n\}$ do
10:        $j' \leftarrow$ FindNext($i, k$)
11:        $res \leftarrow \min\{res,$ CalcSum($i, k$) + MinSum($k, j'$)$\}$
12:    return $res$
```