DOLL: Distributed OnLine Learning Using Preemptible Cloud Instances

Harry Jiang
Department of Electrical and Computer Engineering
Carnegie Mellon University, USA
Email: hhj@alumni.cmu.edu

Xiaoxi Zhang
School of Computer Science and Engineering
Sun Yat-sen University, China
Email: zhangxx89@mail.sysu.edu.cn

Carlee Joe-Wong
Department of Electrical and Computer Engineering
Carnegie Mellon University, USA
Email: cjoewong@andrew.cmu.edu

***Abstract*— Many companies are increasingly making machine learning integral to their businesses. To defray the resulting compute costs, much work has proposed using preemptible cloud instances, a discount tier of virtual machine rentals that may be interrupted at the cloud provider's discretion, to run machine learning jobs. However, processing datastreams on preemptible instances presents new challenges: processing data as they arrive may engender bottlenecks when scaling the system to handle higher throughput, particularly if the instances are frequently interrupted. Ours is the first work to design, analyze, and optimize a system that uses a set of datastreams to train a machine learning model on preemptible instances. Our system, DOLL, uses queueing and batching to parallelize and scale SGD (stochastic gradient descent)-based optimizers to large numbers of workers and datastreams, as well as heterogeneous data arrival rates across streams. Expected error convergence guarantees for convex and non-convex loss functions are then derived for DOLL's training process. We use this guarantee to optimize the cost of requisitioning preemptible and on-demand instances in the face of a performance target and wall-clock time deadline; this optimization is validated on experiments demonstrating substantial cost savings with little impact on model error, compared to on-demand instances.**

I. Introduction

Machine learning (ML) promises to transform many different industries through its use of data to derive predictions and insights for how systems should be run. By automating tasks like sensor data fusion [1] or anomaly detection [2], learning-based systems can drastically reduce the operational costs of many businesses and users. Such a transformation, however, relies on ML algorithm implementations' ability to process large amounts of data. Most large-scale ML implementations utilize a distributed training architecture, i.e., the (large) dataset is spread across multiple servers or virtual machines (VMs) that iteratively compute local model updates that are periodically synchronized. Due to the complexity of managing the resulting computing infrastructure, many companies run their ML jobs at external cloud providers. However, they then face a new challenge: cloud resources can be expensive for large ML jobs with long runtimes. Such expenses can be significant, particularly for startup companies [3], driving numerous recent efforts to reduce ML costs [4], [5].

Running ML jobs on *preemptible* cloud instances is a popular method to limit their training costs [5]–[7]. Preemptible instances are significantly (sometimes 90%) cheaper than conventional on-demand instances [8], [9], but may be interrupted at the cloud provider's discretion. Most studies of ML on preemptible VMs, however, assume large pools of data are available at training time. In practice, training data may arrive at irregular intervals and models may be trained online as new data arrives, e.g., when monitoring intrusion events in computer networks [2], or data from IoT sensors [1] or healthcare records [10]. If this data is collected frequently or is high-dimensional, e.g., live video analytics [11] or updates to hospitals' patient records, distributing its analysis across servers may alleviate compute bottlenecks compared to centralized learning, and updating models as data arrives will reduce data storage costs. Thus, it is important to understand how to economically scale ML jobs on incoming datastreams.

Some software frameworks like Apache Kafka [12] are designed to feed online data arrivals to ML algorithms [13], [14], but they provide little insight into the costs of running ML jobs. Running ML on preemptible VMs, for example, typically requires checkpointing or migration methods to handle unexpected interruptions [15], [16]. Prior work has shown that careful provisioning of preemptible VMs can reduce ML costs while guaranteeing the model convergence on available pools of data [7]. We extend this idea to ML *on datastreams*, which presents new challenges due to the need to carefully handle data arrivals. We design, analyze, and optimize DOLL, which to the best of our knowledge is the first system that provides provable performance guarantees for Distributed OnLine Learning using preemptible instances.

Research Challenges: An intuitive solution to distributed ML on streaming data would be to have each ML worker immediately process a data point as it arrives and synchronize workers' models when all VMs finish their gradient computations. This design can cause *synchronization delays*, as the worker VMs would need to wait for all VMs to receive and compute updates on data. Asynchronous methods [17] eliminate such delays but can increase the training iterations needed for convergence [18], potentially negating the synchronization delay reduction. Inspired by batching techniques for general datastream processing [14], we thus propose a batching and grouping process that limits synchronization delays, while mimicking traditional mini-batch SGD (stochastic gradient descent), allowing us to derive convergence guarantees. Our

next challenge is then to *ensure that a sufficient number of VMs are provisioned* to process the data: if the data arrives faster than a VM can compute updates on it, the data queue at that VM will back up. We must therefore analyze the stability of our batching, grouping, and model updating process.

Provisioning sufficient VMs should ensure not only queue stability but also model convergence, which is made more difficult when we run some VMs on preemptible instances. Existing system designs for ML jobs on preemptible instances focus on mitigating training interruptions [7], [15]. In our setting, such *interruptions may also pause data arrivals*, which impedes the rate at which we can compute model updates. To quantify this effect, we characterize model convergence as a function of the wall-clock training time, unlike prior works that focus on the number of training iterations [7]. By doing so, we can limit the number of workers that use preemptible instead of on-demand VMs to ensure that preemptions are "rare enough" such that the training finishes by a given deadline. This analysis becomes more complex when *different VMs experience different data arrival rates*, e.g., due to receiving data from different sources, and thus generate model updates at different rates, affecting their ability to run on preemptible VMs. VM provisioning will also affect the incurred cost, introducing a *cost-convergence tradeoff*. Moreover, *data arrival rates may be initially unknown*, requiring us to both estimate and optimize over them when training the model. Our work is the ***first to show that we can meet ML convergence guarantees on preemptible VMs for datastreams***.

We give an overview of related work in Section II before making the following **research contributions**:

DOLL Design *(Section III):* We design DOLL, which runs on multiple VMs in a parameter server architecture to distributedly train an ML model on datastreams. DOLL batches arriving data samples at each VM and triggers model updates once enough batches have accumulated across all VMs, which limits delays due to model synchronization between updates and ensures that no single VM has an excessive backlog of data samples. Once an update is triggered, the VMs compute updates for each accumulated batch; these updates are then sent to the parameter server for aggregation.

ML performance analysis *(Section III-B):* While batching and grouping mechanisms similar to DOLL's are used in industry [13], [19], we are the first to analytically quantify the rate at which they can compute ML updates. Under mild assumptions on the (stochastic) data arrival process at each VM, we derive conditions for the stability of the training process, i.e., ensuring data samples do not accumulate at each VM (Proposition 1). We then find a closed-form expression for the asymptotic number of updates that arrive within a given time interval (Theorem 1) and the resulting ML convergence with respect to wall-clock training time (Theorems 2 and 3).

Cost optimization *(Section IV):* We provision preemptible VMs so as to minimize the monetary cost of the ML training, given constraints on model accuracy achieved before a wall-clock deadline. We find the optimal number of preemptible VMs when all VMs have the same data arrival rates (Theorem 4) and an efficient algorithm to do so with heterogeneous data arrivals (Theorem 5), ensuring the "right" degree of preemption to balance cost and performance. These solutions enable an adaptive provisioning strategy that estimates VMs' data arrival rates and availability as the job runs.

Experimental validation *(Section V):* We implement and evaluate DOLL with large-scale experiments on Amazon EC2 spot prices. By judiciously optimizing VM preemption, DOLL achieves 14% to 50% lower cost than standard on-demand instances. We validate our theoretical cost analysis and show that DOLL's benefits are robust to heterogeneous data arrivals. Our adaptive provisioning mechanism reduces the cost by another 15% compared to static optimization.

We outline future directions and conclude our work in Section VI. All proofs are abbreviated to conserve space.

II. Related Work

Distributed online learning: Many recent works have proposed distributing ML training using a variant of mini-batch SGD [20], in which workers iteratively compute the gradients of a given objective function with respect to model parameters over stochastic samples from each worker. Much prior work [21] analyzes the convergence of training error in SGD, e.g., with respect to the wall-clock training time when workers' gradient computations have stochastic runtimes [18], or when the number of workers changes in different iterations, e.g., when workers run on preemptible instances [7]. In the federated learning context, when worker data is not i.i.d. (independently and identically distributed), algorithmic adjustments ensure that SGD on interruptible devices converges [22], [23]. In contrast, we analyze ML convergence under stochastic, online data arrivals that themselves affect the ML update rate.

Learning on online data arrivals may require training time-varying models [24], or stochastic approximation techniques to obtain convergence bounds [25], [26]. Many existing works, however, use SGD-based update methods on streaming data to achieve faster convergence, e.g., SAG [27], SDCA [28], SVRG [29], SAGA [30] and recent variants [31]. Indeed, SGD itself remains a popular tool for datastreams, e.g., in health analytics [10], network intrusion detection [2], and live video analytics [11]. We thus frame our convergence guarantees assuming SGD-based ML updates for streaming data. Frameworks to run these data stream ML algorithms have also received attention recently, with resource management and resilience to infrastructure disruptions primary research challenges [14], [32]. However, these frameworks do not explicitly optimize for ML convergence or preemptible instances.

ML on cloud instances: Infrastructure support for large-scale ML has attracted much recent attention [33], with some frameworks focusing on online datastreams [12]–[14], [32]. Cloud operators can optimize the configuration of servers provisioned for distributed ML jobs so as to minimize operational costs [5], [34] and migrate datastream analysis across different servers to load-balance dynamic workloads as data arrives [35]. ML users can design systems for utilizing "transient" (i.e., preemptible) resources to run distributed data

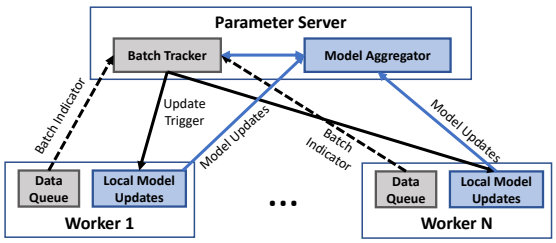


Fig. 1: Workers and parameter server interactions in DOLL. Each worker may run on a preemptible or on-demand instance.

analytics [5], [6], [15], [36], [37], but unlike our work they do not consider stochastic datastream arrivals. Prior work includes using transient instances for parameter servers [38], adjusting the learning rate to compensate for interruptions [16], or utilizing asynchronous SGD [17]. While [7] optimizes preemption probabilities in distributed ML, these are often not controllable [39]. DOLL instead optimizes the provisioning of preemptible instances for ML on incoming datastreams.

III. DOLL System Design

We present DOLL's architecture (Section III-A) and our data batching and grouping mechanisms (Section III-B).

A. System Architecture

System components: We consider a parameter server architecture [7], [18] in which N_{tot} distributed workers or VMs each receive a stream of data. The overall goal is to train a model $h(\boldsymbol{z}, \boldsymbol{w})$, where $\boldsymbol{w}$ are the parameters to be trained, and $\boldsymbol{z}$ represents the model's input features. Each worker n may run on either an on-demand or preemptible cloud instance, and receives samples from a datastream $D_n = \{\boldsymbol{x}_{n,i} : i \in \{1,2,\ldots\}\} : n \in \{1,\ldots,N_{tot}\}$, consisting of a series of data samples $\boldsymbol{x}_{i,n} = (\boldsymbol{z}_{n,i}, y_{n,i}) \overset{i.i.d}{\sim} \mathcal{X}$ which are independent and identically distributed (i.i.d) across time and across every worker's datastream D_n. For example, social networking sites, equipment or network monitoring systems [2], or hospitals in a health system might analyze data inputs across multiple monitoring points or locations that have statistically similar data. We use $\boldsymbol{z}_{n,i}$ to denote the model input features and $y_{n,i}$ to denote the corresponding labels. The arrival events $\tau_{n,i} : i \in \{1,2,\ldots\}$ of successive $\boldsymbol{x}_{n,i}$ in datastream i over time are modelled as a renewal process, i.e., each interarrival time $A_{n,i}$ is i.i.d, both within and between datastreams:

$$A_{n,i} \equiv \tau_{n,i} - \tau_{n,i-1} \overset{i.i.d}{\sim} \mathcal{A}, \forall n \in \{1,\ldots,N_{tot}\}, i \in \{1,2,\ldots\}$$
$$\tau_{n,0} = 0$$

For example, if data is routed to workers from a central collection point, each worker will likely experience i.i.d. interarrivals. In the next section, we relax the assumption that the $A_{n,i}$ are i.i.d. across workers n, e.g., if samples are directly sent to workers from observation points with different sample generation frequencies. As shown in Figure 1, each worker maintains a queue of arrived data samples and a module for computing updates to the model parameters $\boldsymbol{w}$.

Each worker communicates with a parameter server (PS) that stores a global model $h(\boldsymbol{z}, \boldsymbol{w})$ to be trained, where $\boldsymbol{w}$ are the parameters to be determined through the training process. We assume that the parameter server is always available to coordinate the workers, e.g., if it runs on an on-demand instance. Note that the parameter server may be physically implemented with multiple physical servers that coordinate with each other, e.g., an all-reduce architecture; this does not impact our subsequent analysis [38]. As shown in Figure 1, the PS both monitors arrivals at workers (explained below) and aggregates model updates received from workers.

Training process: The goal of training the ML model is to minimize the expected loss, or risk, of the model h, i.e., to solve $\min_{\boldsymbol{w}} \mathbb{E}_{\boldsymbol{x}\sim\mathcal{X}} L(h(\boldsymbol{z}, \boldsymbol{w}), \boldsymbol{y})$, where $L(h, y)$ denotes the loss of the model prediction h compared to the ground truth y. In practice, however, $\mathcal{X}$ is unknown. Thus, as is typical in ML and streaming analytics applications [5], [10], we train the model with a variant of SGD on available data samples. We break up the training into running "producer" and "consumer" processes at each worker and the PS, as described below.

The process proceeds iteratively for updates $J = 1, 2, \ldots$: each worker i continually accumulates data samples, sending a *batch indicator* $S_{B;j,n}$ to the PS when it accumulates b samples since the last indicator. The batching process exists as the "producer" process running on a given worker (see Algorithm 1). Receipt of the batching indicators then induces a "producer" process at the PS (see Algorithm 3). The PS periodically sends a model update trigger to all workers with the current model parameters $\boldsymbol{w}_J$. The resulting gradient computation process at each worker constitutes its "consumer" process (see Algorithm 2). Workers begin computing gradients when they receive the update trigger, to ensure they are working from the correct global model. Let $\{B_k,\ k = 1, 2, \ldots, K\}$ denote the batches accumulated across all workers since update $J-1$. Each worker with a batch B_k then computes the gradient $g_{J,k} = \frac{\partial}{\partial \boldsymbol{w}_{J-1}} \left(\sum_{\boldsymbol{x}_{n,i} \in B_k} L(h(\boldsymbol{z}_{n,i}, \boldsymbol{w}_{J-1}), \boldsymbol{y}_{n,i}) \right)$ of the loss function. Data samples may continue to arrive during the gradient computations; if they arrive quickly relative to the computation time, a new training round may be triggered shortly after the previous round completes. The resulting gradients from each batch are sent to the PS, which aggregates them to perform gradient descent: $\boldsymbol{w}_J \leftarrow \boldsymbol{w}_{J-1} - \eta \sum_k g_{J,k}$, where η is a pre-determined step size (see Algorithm 4). This aggregation constitutes the PS "consumer" process. The next iteration $J+1$ commences once the PS sends another update trigger. We let C denote the (stochastic) gradient computation and upload time at each worker and assume C is i.i.d. across workers, e.g., as for workers of the same cloud instance type.

Throughout the paper, we use $\left(\mu_Y, \lambda_Y = \frac{1}{\mu_Y}, \sigma_Y^2\right)$ to respectively denote the mean, reciprocal mean, and variance of a random variable Y.

Handling preemptions: The cloud provider may preempt any worker at any time. Workers do not receive data samples or compute model updates while preempted but retain any samples received before the preemption begins, via standard

Algorithm 1: Producer process on a given worker n.

Data: stream $D_n = \{\boldsymbol{x}_{i,n} \overset{i.i.d}{\sim} \mathcal{X} : i \in \{1, 2, \ldots\}\}$
Parameter: batch size b

$i \leftarrow 0$ //data point count
$j \leftarrow 0$ //local batch count; J is global count
while $G(\boldsymbol{w}) - G^* > \epsilon$ **do** //dependent on PS model
  $B_j \leftarrow \{\}$
  for $a \leftarrow 0$ **to** b **do** //batch gathering
    wait for arrival of data point $\boldsymbol{x}_{i,n}$ from D_n
    $B_j \leftarrow B_j || \boldsymbol{x}_{i,n}$ //add data point to batch
    $i \leftarrow i + 1$
  send batch indicator $S_{B;j,n}$ to PS
  store B_j in memory for consumer
  $j \leftarrow j + 1$

Algorithm 2: Consumer process on a given worker n.

Data: stored batches $\{B_j : j \in \{1, 2, \ldots\}\}$
Parameter: loss function $G\left(h\left(\boldsymbol{z}, \boldsymbol{w}\right), \boldsymbol{y}\right)$

$\boldsymbol{w}_n$ //local model weights
while $G(\boldsymbol{w}) - G^* > \epsilon$ **do** //dependent on PS model
  wait for update trigger $S_{C;j',n}$ from PS
  $\boldsymbol{w}_n \leftarrow \boldsymbol{w}$ //copy PS model to worker
  retrieve $B_{j'}$ from memory
  $g_{j',n} \leftarrow \frac{\partial}{\partial \boldsymbol{w}_n}\left(\sum_{\boldsymbol{x}_{i'} \in B_{j'}} G\left(h\left(\boldsymbol{z}_{i'}, \boldsymbol{w}_n\right), \boldsymbol{y}_{i'}\right)\right)$
  send $g_{j',n}$ back to PS

fault tolerance mechanisms [6]. Workers do not return updates if they are preempted after sending a batch indicator and before the PS issues an update trigger. We will incorporate the effect of preemptions in our analysis in the next section.

B. Solution Design

We next analyze the stability of the DOLL architecture, which we show in Section IV will allow us to optimize its cost. Figure 2 shows that the update process ($\{U_j\}$) is the departure process resulting from a composition of the data arrivals at workers ($\{A_{n,i}\}$), batching of these arrivals ($\{B_{n,i}\}$), grouping of the batches ($\{\Gamma_j\}$), and gradient computations.

We will assume that data sample arrival times at each worker are i.i.d. processes of rate λ_A (we later relax this analysis to heterogeneous, time-varying rates). This can for example

TABLE I: Key notation. Figure 2 illustrates further notation for the data progression through batching and grouping.

Symbol	Definition
J	SGD iteration number
$\boldsymbol{w}_J$	ML model parameters at iteration J
$G(\boldsymbol{w})$	Loss function
N, N_{tot}	Number of available workers and total number of workers
K, b	Group and batch size, respectively
$\lambda_A \equiv 1/\mu_A$	Data arrival rate at a given worker
π_s, π_o	Hourly price of a spot instance and on-demand instance
ϵ	Target error of training job
θ	Wall-clock training time

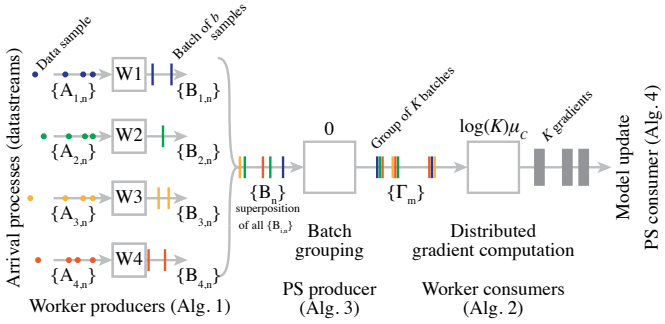


Fig. 2: Data progression diagram of DOLL. Note that data does not physically leave the workers. Data progresses through arrivals ($\{A_{n,i}\}$), batching ($\{B_{n,i}\}$), grouping ($\{\Gamma_j\}$), and gradient computations. The numbers above the batch grouping and gradient computation boxes indicate the mean holding time for each queue. Note that the queue for batching is only a *virtual* queue used to express the bulk processing behaviour of grouping K batches for each update. The computed gradients leave the worker for the PS to aggregate.

model streams of taxi passenger data [40], sensor networks [1], or social networks [41]. We next analyze how the batching and grouping of data arrivals affect DOLL's model updates. For simplicity, we initially assume that workers are not preempted.

1) Batching: Each model update is computed on a *batch* of b data points, as in mini-batch SGD. This batching adds another intermediate process $\{B_{n,i}\}$ at the output of each worker, with interarrival times $B_{n,i} = \sum_{i'=b(i-1)+1}^{bi} A_{n,i'}$. The departure rate of batches is then λ_A/b, and $\{B_n\}$ is a superposition of the individual processes $\{B_{n,i}\}$ (Figure 2).

The input process to Γ is then the departure process $\{B_n\}$. Since the superposition of N identical renewal processes tends towards a Poisson distribution as $N \to \infty$, we use a Poisson process to approximate $\{B_i\}$, the superposition of $\{B_{n,i}\}$ [42]. In practice, there are usually enough workers ($N = 64$ in Section V's experiments) for this approximation to hold [1]. This approximation will have rate $\lambda_B \equiv N\lambda_A/b$.

2) Grouping: A natural sequence of processing data points would be for the PS to send an update trigger after any worker accumulates a batch. However, new batches will continue to arrive while a worker is computing gradients. Thus, this naïve strategy may induce significant backups of batches, which cannot be analyzed until the worker finishes computing the gradient. Moreover, enforcing such synchrony defeats the point of having a distributed system, since batches at other workers could not be processed either until the next model update.

We prevent these backups, and preserve the benefits of a distributed system, by placing data arrivals in a queue at each worker. The PS initiates a model update once K mini-batches have arrived across all workers. This type of queue is called a bulk processing queue [43], such as in roller coaster queues, where visitors enter the queue individually (Algorithm 3), but are processed in groups (Algorithm 4). We can quantify the resulting update arrival rate by viewing the bulk processing

queue as two queueing systems in series (see Figure 2); preceding the main queue is a virtual grouping queue that receives the batches and outputs a group process $\{\Gamma_j\}$ with interarrivals that are the sum of K batch interarrival times: $\Gamma_j = \sum_{i=K(j-1)+1}^{Kj} B_i$.

Since the virtual grouping queue has zero holding time, it will not affect system stability. Letting μ_C denote the mean gradient computation time, the main computation queue has a mean holding time (i.e. iteration computation time) less than $K\mu_C$ (if all K batches in the group are from the same worker) and greater than e.g. $\log(K)\mu_C$ assuming exponential gradient computation time (if all K batches are from different workers). A priority queue can also be used (i.e., each batch is associated with a priority that decreases with the number of batches from the same worker already in the queue) to significantly reduce the probability of multiple arrivals from the same worker appearing in the same group for $K \leq N$. We can then derive stability conditions on the computation process by analyzing the batch interarrival times $\{B_i\}$.

With the Poisson approximation of $\{B_i\}$, the distribution of Γ_j will be $\Gamma_j \sim Erlang\left(K, \frac{b\mu_A}{N}\right)$. We can now find conditions for the stability of the overall process:

Proposition 1 (Update process stability)**.** *For the process $\{\Gamma_j\}$ to be stable, it is sufficient to have $\frac{b\mu_A}{N} > \mu_C$. Moreover, if $\{\Gamma_j\}$ is stable, then its departure process has the same mean interarrival time μ_U as that of $\{\Gamma_j\}$, μ_Γ.*

Proof. Follows directly from batch and computation rates. □

Algorithm 3: Parameter Server producer process.

```
Q ← Queue() //initialize batch queue
while G(w) − G* > ϵ do
    wait for any batch indicator S_{B;j,n} from workers
    Q.enqueue(j, n) //store worker, batch indices
```

Algorithm 4: Parameter Server consumer process.

Parameter: group size K, step size η, target error threshold ϵ

```
Initialize model weights w
while G(w) − G* > ϵ do          //arbitrary end condition in practice
    for γ ← 0 to K do           //group gathering
        j, n ← Q.dequeue() //retrieve batch data
        broadcast update trigger S_{C;j,n} to workers
    broadcast PS model weights w to workers
    g ← 0
    for γ ← 0 to K do
        wait for any gradient g_{j,n} from workers
        g ← g + g_{j,n} //add gradient
    w ← w − ηg
```

3) Model Updates: The event of the PS model being updated, which we denote as U, can be modeled as the departure process from the grouping Γ (Algorithm 4). Proposition 1 allows us to characterize the effective arrival rate, mean inter-update time μ_U and inter-update time variance σ_U^2:

$$\lambda_U \equiv \frac{1}{\mu_U} = \frac{N\lambda_A}{Kb},\ \sigma_U^2 \leq \frac{Kb^2}{N^2\lambda_A^2} + K^2\sigma_C^2 \tag{1}$$

for a given variance of computation time σ_C^2 at an individual worker. The variance of inter-update time is expressed by its upper bound with the worst-scenario in mind, that is, the case of every group consisting of batches from the same worker. Also, as shown above, the limiting factor in update rate is λ_A. We can now find the number of updates within a given wall-clock time interval t; we generally care about the state of the ML model at the end of training, so t will be a large number:

Theorem 1 (Updates vs. wall-clock time)**.** *For large wall-clock time t, the number of updates after elapsed time t, J_t, converges in distribution to a normal distribution.*

$$J_t \to \mathcal{N}\left(t\lambda_U, t\sigma_U^2\lambda_U^3\right) \tag{2}$$

Proof. Follows from the Central Limit Theorem. □

Corollary 1 (Update rate without preemptions)**.** *The relationship between expected number of iterations in a time period t and the expected runtime to achieve J iterations, for sufficiently large J and t, are*

$$\mathbb{E}[J_t] = t\lambda_U, \quad \mathbb{E}[runtime(J)] = \frac{J}{\lambda_U}. \tag{3}$$

Proof. Follows from properties of J_t in Theorem 1. □

4) Preemption: With preemptible instances, the number of workers online at any given moment may not equal N. However, in practice, worker preemptions tend to occur on a slower timescale compared to data arrivals [9]. Thus, we follow prior work in assuming that preemptions do not occur during a model update [7]; equivalently, we may assume that preemptions that occur during updates discard the update altogether and that preemption overhead (shutdown and booting) is negligible on the time scale of the training task. Let $N(t) \in \{0, 1, \ldots, N_{tot}\}$ be the number of active workers at time t; to account for variable workers, N would have to be replaced with $N(t)$ in finding the arrival and departure rates of the update queue as in (1), as well as the number of updates J_t that occur prior to time t:

$$\begin{aligned} \mu_U(t) &= \frac{Kb\mu_A}{N(t)},\ \lambda_U(t) = \frac{N(t)\lambda_A}{Kb} \\ \mathbb{E}[J_t] &= \int_0^t \lambda_U(\tau)d\tau = \frac{\lambda_A}{Kb}\int_0^t N(\tau)d\tau \end{aligned} \tag{4}$$

No updates occur while $N(t) = 0$, and $\lambda_U(t) = 0$ in such situations. For a time-varying rate $\lambda_A(t)$, e.g., if the sampling rate varies by time of day [40], we similarly express the expected number of updates as $\mathbb{E}[J_t] = \frac{1}{Kbt}\int_0^t \lambda_A(\tau)d\tau \int_0^t N(\tau)d\tau$.

We suppose that preemptions are initiated by the cloud provider in a fashion opaque to users, as is the case for most such offerings [8], [44]. Amazon Web Service (AWS)'s spot instances are a possible exception: workers can bid for spot instances, which are preempted when the (dynamic) spot price falls below their bids. However, in practice spot prices change very slowly over time [9]; for example, they varied by at most \$0.02 from November 2021 to January 2022 for `c5.xlarge` instances in AWS's US East region. Most preemptions occur due to Amazon's exogenous actions and not user bids. As such, we can model the state of a single preemptible instance as a continuous-time Markov chain (CTMC) with two states, OFF and ON, which has the stationary distribution $(1-\alpha, \alpha)$, where $\alpha \in [0,1]$ is the fraction of time the preemptible instance is available. With N_{tot} preemptible instances, we define the CTMC $\boldsymbol{N}$ with states $\{0, 1, \ldots, N_{tot}\}$, corresponding to the number of active preemptible instances at a given moment. Its stationary distribution can be shown to be binomial $B(N_{tot}, \alpha)$. With this model in mind, $N(t)$ becomes a path through $\boldsymbol{N}$. We will use $\mathbb{E}[N] = \alpha N_{tot}$ to represent the expectation value of the mean of $N(t)$ across the space of possible paths through $\boldsymbol{N}$.Then for a large enough time period, we can modify Corollary 1 to find the time required for J model updates to be made:

Proposition 2 (Runtime)**.** *The limit of expected runtime for J model updates, Θ_J, as J increases is:*

$$\lim_{J\to\infty} \frac{\mathbb{E}[\Theta_J]}{J} = \frac{Kb}{\lambda_A \mathbb{E}[N]} \tag{5}$$

Proof. As time t progresses, the distribution of time spent in each state of $\boldsymbol{N}$ approaches the stationary distribution of the system. Therefore, the mean value of a path $N(t)$ approaches $\mathbb{E}[N]$ and the integral of $N(t)$ can be expressed with the following limit: $\lim_{t\to\infty} \frac{1}{t}\int_0^t N(\tau)d\tau = \mathbb{E}[N]$. Inserting the above into (4), for high runtimes and any path $N(t)$, we have

$$\lim_{t\to\infty} \frac{1}{t}\mathbb{E}[J_t] = \frac{\lambda_A}{Kb}\int_0^t N(\tau)d\tau = \frac{\lambda_A}{Kb}\mathbb{E}[N]$$

If we fix J to a large value and instead replace t with a random variable representing the necessary runtime θ_J, the analogous relation will also hold true: $\lim_{J\to\infty} \frac{\lambda_A}{JKb}\mathbb{E}[N]\mathbb{E}_{N(t)}[\theta_J] = 1$, where the expectation of θ_J is over the space of paths $N(t)$ through $\boldsymbol{N}$. Rearranging the above yields (5). □

Given Proposition 2, we will define an effective mean runtime given J updates, $\bar{\Theta}_J$ as

$$\bar{\Theta}_J \equiv \frac{J}{\lambda_U} \equiv \frac{JKb}{\lambda_A \mathbb{E}[N]} \tag{6}$$

IV. Performance Analysis

The goal of DOLL's design is to minimize the cost of running ML jobs on preemptible instances, subject to convergence constraints: while we would like the model training to cost as little as possible, the final trained model must be accurate enough, and delivered promptly enough, to be useful. We can formalize this goal in an optimization framework as:

$$\begin{aligned} \min \quad & \mathbb{E}[Cost] \quad \text{s.t.} \quad \mathbb{E}[G(\boldsymbol{w}_{J_\theta})] - G^* \leq \epsilon \\ \text{where} \quad & \theta = \text{wall-clock time deadline}, \epsilon = \text{target error} \end{aligned} \tag{7}$$

Here, G is the loss function associated with the machine learning task. Thus, in this section we use Section III-B's analysis to solve this optimization problem. We first analyze the model convergence with respect to wall-clock time (Section IV-A), allowing us to derive expressions for our constraint in (7), and then solve the resulting optimization problem in Section IV-B.

A. Model Convergence

The bounds determined in this subsection are novel and serve to unify the convex and non-convex cases with DOLL.
Convex loss: We define $G(\boldsymbol{w})$ as the expected loss over $\mathcal{X}$:

$$G(\boldsymbol{w}) = \int_{\mathcal{X}} L(h(\boldsymbol{z};\boldsymbol{w}), y(\boldsymbol{z}))dp(\boldsymbol{z}), \tag{8}$$

where $h(\boldsymbol{z};\boldsymbol{w})$ is the hypothesis function with input features $\boldsymbol{z}$ and parameters $\boldsymbol{w}$, and $L(h,y)$ is the divergence function selected for the machine learning task. We upper bound the expected model error assuming J model updates over t time:

Theorem 2 (Convex convergence)**.** *For an L-Lipschitz smooth and c-strongly convex machine learning loss function G, the expected error converges geometrically towards an error floor*

$$\mathbb{E}[G(\boldsymbol{w}_{J+1}) - G^*] \leq \int_0^\infty \phi(j) f_{J_t}(j)dj \quad \leq \beta e^{-\underline{\rho} t} + \gamma \tag{9}$$

for a given $\beta > 0$, $\gamma \geq 0$, and $0 < \underline{\rho} < 1$. f_{J_t} is the pdf (probability distribution function) of J_t in Theorem 1.

Proof. The model updates in a synchronous manner, as dictated by the main queue which processes groups of batches. As such, the convergence of the expected loss in relation to the number of model updates would be the same as a K-synchronous SGD [18] implementation. With modifications made to account for groups of batches being processed, we can upper-bound $\mathbb{E}[G(\boldsymbol{w}_{J+1})] - G(\boldsymbol{w}^*)$ as

$$\leq (1-\eta c)^J \left(\mathbb{E}[G(\boldsymbol{w}_0)] - G(\boldsymbol{w}^*) - \frac{\eta LB}{2Kbc}\right) + \frac{\eta LB}{2Kbc} \tag{10}$$

We assume that gradients are upper-bounded in magnitude such that $\|\bar{g}_j(\boldsymbol{w}_j)\|^2 \leq B$. For simplicity, the following are defined: $\rho \equiv 1-\eta c$, $\beta \equiv \mathbb{E}[G(\boldsymbol{w}_0)] - G(\boldsymbol{w}^*) - \frac{\eta LB}{2Kbc}$, and $\gamma \equiv \frac{\eta LB}{2Kbc}$, allowing us to express Equation 10 as $\phi(j) \equiv \beta\rho^j + \gamma = \beta e^{log(\rho)j} + \gamma$, where $\phi(j)$ is equivalent to the right hand side of Equation 10. We then bound the loss expectation in time, where $G^* \equiv G(\boldsymbol{w}^*)$ is the minimum loss:

$$\bar{G}(t) \equiv \mathbb{E}_{N(t)}[G(\boldsymbol{w}_{J+1})] = \sum_{j=0}^{\infty} \mathbb{E}[G(\boldsymbol{w}_j)] p_{J_t}(j)$$

$$\bar{G}(t) - G^* = \sum_{j=0}^{\infty} \left(\mathbb{E}[G(\boldsymbol{w}_j)] - G^*\right) p_{J_t}(j)$$

For large t, we substitute the summation with an integral and the pdf of J_t with that of the asymptotic normal distribution,

then take the inner product of the convergence bound $\phi(j)$ and the asymptotic pdf of the number of updates $f_{J_t}(j)$:

$$\bar{G}(t) - G^* \leq \int_0^\infty \phi(j) f_{J_t}(j) dj, \tag{11}$$

which is upper bounded by $\beta e^{-\underline{\rho} t} + \gamma$ to give the result. □

We can incorporate the impact of preemption on (9) by using Proposition 2 to replace J_t with $\mathbb{E}[J_t]$.

Non-convex loss: We extend our convergence analysis to non-convex loss functions, omitting details for brevity. [45] From the point of view of the PS, the model is updated in the same manner as conventional SGD, so we may alternatively use known bounds:

Proposition 3 (SGD non-convex convergence)**.** *For a smooth loss function G, its expected gradient converges such that*

$$\min_{j=1:J} \mathbb{E}\left[\|\nabla G(\boldsymbol{w}_j)\|^2\right] \leq O\left(1\Big/\sqrt{J}\right) \tag{12}$$

given that computed gradients are unbiased estimators $\mathbb{E}[\bar{g}_j(\boldsymbol{w}_j)] = \nabla G(\boldsymbol{w}_j)$, the magnitudes of gradients are bounded by B, and an $\eta \equiv c/\sqrt{J}$ is selected for a $c > 0$.

Proof. Convergence of SGD is proven in [46]. □

Theorem 3 (Non-convex convergence)**.** *For a smooth loss function G, its expected gradient converges in time such that*

$$\min_{j=1:J_t} \mathbb{E}\left[\|\nabla G(\boldsymbol{w}_j)\|^2\right] \leq O\left(1\Big/\sqrt{J_t}\right) = O\left(1\Big/\sqrt{t}\right) \tag{13}$$

given the assumptions stated in Proposition 3.

Proof. By the elementary renewal theorem, $\lim_{t\to\infty} \mathbb{E}[J_t]/t = \lambda_J$, so we extend Proposition 3 to convergence in time. [47, Chap. 5.6.2] □

For simplicity of expression, in the next section we will measure the loss function error, as opposed to gradient magnitude, in optimizing the cost of ML training.

B. Cost Optimization

Having established Theorems 2 and 3's convergence guarantees, we now turn our attention to solving our cost optimization problem in (7). We separate the single constraint into two constraints, that each deal solely with error and runtime:

$$\begin{aligned} \min_{N_s, J} \quad & \mathbb{E}[Cost(N_s)] \\ \text{s.t.} \quad & \mathbb{E}[G(\boldsymbol{w}_J)] - G^* \leq \epsilon,\ \mathbb{E}[runtime(J)] \leq \theta \\ \text{where} \quad & \theta = \text{wall-clock time deadline}, \epsilon = \text{target error} \end{aligned} \tag{14}$$

For the non-convex case, we would apply the error condition ϵ to the magnitude of the gradient such that $\|\nabla G(\boldsymbol{w}_j)\|^2 \leq \epsilon$. We use Theorems 2 and 3 to verify the feasibility of meeting both constraints in this optimization problem. Here we have two decision variables: N_s is the number of preemptible instances to request, with the remaining $N_o \equiv N_{tot} - N_s$ instances being on-demand. The number of iterations J should be chosen large enough to satisfy the error constraint, but not so large so as to violate the runtime constraint. We first consider homogeneous and then heterogeneous arrival rates.

We model the cost of each preemptible instance to be a fixed unit cost per time, as offered by Google [8]. AWS spot instances [9], a popular type of preemptible instances, do allow users to bid on spot instances, potentially resulting in variable prices. As discussed in Section III-B4, however, spot prices in practice change extremely slowly. Thus, it is reasonable to bid the current spot price, as recommended by AWS [9], and we may model this price as a constant. Note that we use real spot price traces in Section V's experiments. Microsoft Azure offers variable price discounts on preemptible VMs [44], but the user has no control or visibility into the causes of price variability or preemptions, so we can simply use their expected price value and assume they are independent of preemptions.

1) Homogeneous arrival rates: We denote the preemptible instance posted price as π_s, offered at a lower rate than the on-demand unit price π_o for the same VM type. Instances are preempted arbitrarily if on-demand usage surpasses leftover VM supply, and new preemptible instances are only available when demand is below leftover VM supply. Since exact supply and demand are opaque to customers, availability of preemptible instances can be abstracted into a value $\alpha \in [0, 1]$ (that is, an instance is available for computation according to a Bernoulli distribution with parameter α (see Section III-B4).

As in Section III-B4, the number of available preemptible VMs follows a binomial distribution $N_a \sim B(N_s, \alpha)$. From (4), the effective model update rate λ_U can be computed using the expected value of $N(t)$: $\lambda_U = \frac{(N_o + \mathbb{E}[N_a])\lambda_A}{Kb} = \frac{(N_o + \alpha N_s)\lambda_A}{Kb}$ By (6), the effective runtime is then:

$$\mathbb{E}[runtime(J(\epsilon))] = \bar{\Theta}_{J(\epsilon)} = \frac{J(\epsilon) Kb}{\lambda_A (N_o + \mathbb{E}[N_a])} \tag{15}$$

$J(\epsilon)$ in this setting refers to the number of model updates required to achieve a given error threshold ϵ. By multiplying $\bar{\Theta}_{J(\epsilon)}$ with the effective total price per unit time $(N_{tot} - N_s)\pi_o + \alpha N_s \pi_s$, we closely approximate the expected cost:

$$\mathbb{E}[Cost(N_s)] = \frac{J(\epsilon) Kb}{\lambda_A} \frac{N_{tot}\pi_o + (\alpha\pi_s - \pi_o)N_s}{N_{tot} + (\alpha - 1)N_s} \tag{16}$$

Lemma 1 (Cost monotonicity)**.** *The expected cost function* (16) *is monotonously decreasing in N_s.*

Proof. We first relax N_s to be any real number in $[0, N_{tot}]$. The derivative of $\mathbb{E}[Cost(N_s)]$ in N_s is then as follows:

$$\frac{d\mathbb{E}[Cost(N_s)]}{dN_s} = \frac{J(\epsilon) Kb}{\lambda_A} \frac{\alpha N_{tot}(\pi_s - \pi_o)}{(N_{tot} + (\alpha - 1)N_s)^2} < 0. \tag{17}$$

As such, the expected cost monotonously decreases in N_s. □

Theorem 4 (Optimal preemptible instances)**.** *The number of preemptible instances N_s solving* (14) *by optimizing the formulation of expected cost in* (16) *is*

$$N_s^* = \min\left\{ \left\lfloor \frac{N_{tot} - \frac{J(\epsilon) Kb}{\lambda_A \theta}}{1 - \alpha} \right\rfloor, N_{tot} \right\} \tag{18}$$

where $J(\epsilon)$ is the expected number of updates J for which the error constraint is tight.

Proof. Lemma 1 shows that either the runtime constraint or $N_s \leq N_{tot}$ must be active. Knowing runtime, the constraint can be expressed as $\mathbb{E}[runtime(J(\epsilon))] = \frac{J(\epsilon)Kb}{\lambda_A(N_o+\mathbb{E}[N_a])} \leq \theta$. Rearranging and seting N_s to an integer yields the result. □

We can use Theorem 4 to estimate the cost savings of using the optimal number of preemptible instances:

Corollary 2 (Cost savings guarantee with preemptible instances)**.** *Letting $\theta_0 \equiv \frac{J(\epsilon)Kb}{N_{tot}\lambda_A}$ denote the expected runtime without interruptions, DOLL's expected cost satisfies*

$$\frac{\mathbb{E}[Cost(N_s^*)]}{\mathbb{E}[Cost(0)]} \geq \max\left\{\frac{\theta}{\theta_0} + \frac{(\alpha\pi_s - \pi_o)\left(\frac{\theta}{\theta_0} - 1\right)}{\pi_o(1-\alpha)}, \frac{\pi_s}{\pi_o}\right\} \tag{19}$$

Thus, if α is large (high availability), we only use preemptible instances and the cost savings is the ratio of spot to on-demand prices. Otherwise, it is a more complex function of α.

2) Heterogeneous Data Arrival Rates: In some distributed applications, workers do not all receive data at an even rate; this may be due to physical constraints, such as geography, or localized communication bottlenecks. We next generalize Theorem 4's results to heterogeneous arrival rates. We can follow a similar approach to that for homogeneous data arrivals in order to find the optimal number of preemptible instances. Through the Palm-Khintchine theorem, we approximate the superposition of a large number N of renewal processes (with rates $\lambda_1, \ldots, \lambda_N$) with a single Poisson process with rate $\sum_{n=1}^{N} \lambda_n$ [48]. Intuitively, ordering arrival rates by magnitude and assigning preemptible instances to workers with the lowest arrival rates minimizes the impact of preemption. For ordered arrival rates $\lambda_{A,1} \leq \lambda_{A,2} \leq \cdots \leq \lambda_{A,N_{tot}}$, the effective update rate is the sum of the effective arrival rates of each worker:

$$\lambda_U = \frac{\sum_{n=1}^{N_{tot}} \lambda_{A,n} - (1-\alpha)\sum_{n=1}^{N_s} \lambda_{A,n}}{Kb} \tag{20}$$

and the effective mean runtime given $J(\epsilon)$ updates is:

$$\bar{\Theta}_{J(\epsilon)}) = \frac{J(\epsilon)Kb}{\sum_{n=1}^{N_{tot}} \lambda_{A,n} - (1-\alpha)\sum_{n=1}^{N_s} \lambda_{A,n}} \tag{21}$$

We can derive an expression for the expected cost in a similar fashion to (16) by multipling the expected runtime with the effective total price per unit time $(N_{tot} - N_s)\pi_o + \alpha N_s \pi_s$:

$$\mathbb{E}[Cost(N_s)] = J(\epsilon)Kb\frac{N_{tot}\pi_o + (\alpha\pi_s - \pi_o)N_s}{\sum_{n=1}^{N_{tot}} \lambda_{A,n} - (1-\alpha)\sum_{n=1}^{N_s} \lambda_{A,n}} \tag{22}$$

Lemma 2 (Necessary conditions on N_s)**.** *Given ordered arrival rates $\{\lambda_{A,n}\}$, the cost-optimal $N_s \leq N_{tot}$ must satisfy:*

$$(1-\alpha)\sum_{n=1}^{N_s} \lambda_{A,n} \leq \sum_{n=1}^{N_{tot}} \lambda_{A,n} - \frac{J(\epsilon)Kb}{\theta} \tag{23}$$

Proof. Follows by rearranging the runtime constraint. □

Theorem 5 (Optimal preemptible instances for heterogeneous arrivals)**.** *The expected cost function* (22) *contains at most one local minimum in the range $N_s \in [0, N_{tot}]$.*

Proof. This can be shown by relaxing $\{\lambda_{A,n}\}$ into a continuous monotonically increasing function $\lambda(n)$. The expected cost function can thus be expressed as:

$$\mathbb{E}[Cost(N_s)] = J(\epsilon)Kb\frac{N_{tot}\pi_o + (\alpha\pi_s - \pi_o)N_s}{\int_0^{N_{tot}} \lambda(n)dn - (1-\alpha)\int_0^{N_s} \lambda(n)dn}$$

As $\pi_s < \pi_o$ and $\alpha \leq 1$, the numerator of the expected cost function decreases linearly. As $\lambda(n)$ monotonically increases on $N_s \in [0, N_{tot}]$, the denominator monotonically decreases on $N_s \in [0, N_{tot}]$. Moreover, the derivative of the denominator monotonically decreases on $N_s \in [0, N_{tot}]$. Because the derivatives of the numerator and denominator intersect at most once within $N_s \in [0, N_{tot}]$, the expected cost function contains at most one turning point in $N_s \in [0, N_{tot}]$. As the second derivative of the denominator is negative on $N_s \in [0, N_{tot}]$, any turning point must be a local minimum. □

This result allows a simple algorithm for determining the optimal N_s: we simply evaluate the expected cost at each N_s starting at $N_s = 0$, increasing N_s until either the constraint in Lemma 2 is broken or the expected cost begins increasing.

We finally note that Theorems 4 and 5 also allow us to determine the preemptible instance region in which to run our ML job, based on their prevailing prices [9]. Since we can easily solve for the optimal number of preemptible VMs, Theorems 4 and 5 also suggest that it is feasible to design an adaptive algorithm that re-optimizes our provisioning as the job runs, e.g., in case we do not know VM availability *a priori*, which we experimentally validate in Section V-C.

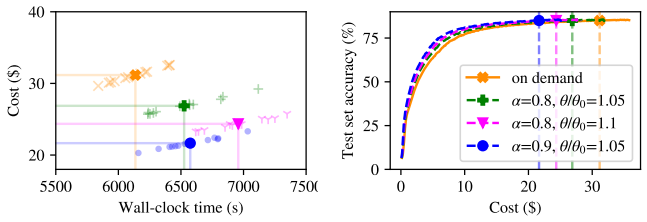
V. Experiments and Results

We finally evaluate our system design and optimization from Sections III and IV through experiments on AWS (Amazon Web Services). We first describe our experimental environment (Section V-A) and then show that DOLL can substantially reduce ML costs compared to on-demand instances (Section V-B). We analyze the impact of different deadline and price parameters on the cost savings, and finally introduce an adaptive version of our preemptible instance optimization (Section V-C) that adjusts the provisioning over time.

A. Experimental Environment

We implement DOLL with a parameter server on an AWS EC2 `m4` VM and $N_{tot} = 64$ workers on `r5` VMs. A separate `g4dn` VM hosts a test dataset, allowing us to monitor test accuracy without interfering with the training.

Unless otherwise stated, the learning task used is the handwriting classification problem using the Extended MNIST (EMNIST) dataset, composed of 814225 images [49]. (We additionally report results for the Infinite MNIST [50], which can infinitely generate virtual handwriting examples and is used as a benchmark dataset in online ML experiments [51] [52], and CIFAR-10 [53] image classification datasets under



(a) Cost vs. wall-clock time (b) Accuracy vs. Cost (averaged)

Fig. 3: Using Theorem 4 for requisitioning preemptible instances yields target performances (solid dots in the figures) at on average 69.50% ($\alpha = 0.9$, $\theta/\theta_0 = 1.05$), 86.22% ($\alpha = 0.8$, $\theta/\theta_0 = 1.05$), and 78.05% ($\alpha = 0.8$, $\theta/\theta_0 = 1.10$) of the on-demand cost. Costs diminish with higher α (preemptible instance variability) and θ/θ_0 (longer deadlines), consistent with Corollary 2. Figure 3a displays cost and wall-clock time for 12 experiments and their mean values per setting. Figure 3b shows that preemptions do not affect the final test accuracy.

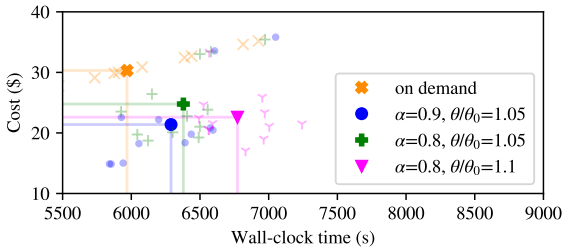


Fig. 4: For heterogeneous arrival rates, DOLL incurs costs on average 70.54% ($\alpha = 0.9$ and $\theta/\theta_0 = 1.05$), 81.71% ($\alpha = 0.8$ and $\theta/\theta_0 = 1.05$), and 74.54% ($\alpha = 0.8$ and $\theta/\theta_0 = 1.10$) below the on-demand training cost. The figure displays cost and wall-clock time for 12 experiments per setting, as well as their means. Randomness in the arrival rates affects both cost and wall-clock time; as such, savings may vary significantly.

homogeneous and heterogeneous arrival rates in Table II.) The dataset is evenly divided into N_{tot} shards and assigned to a worker; data samples are acquired by iterating through shards in a random order. Datastreams are simulated by augmenting the dataset; the uniqueness of training inputs is achieved by filtering each sample through a random transform layer (random perspective plus random affine transformations) before gradient computations. The classification objective uses the ByMerge split, which contains 47 unbalanced classes corresponding to alphanumeric characters [49], and cross-entropy loss. For the EMNIST and I-MNIST tasks, we train a convolutional neural network (CNN) with 2 convolutional and 2 dense linear layers; for CIFAR-10, we train a CNN with 2 convolutional and 3 linear layers. We use PyTorch and Ray for the learning algorithm and node communication, respectively.

Preemption is simulated through a two-state (preempted/running) Markov chain sampled every 2 seconds, with specified availability value α for a expected on-off cycle length of 1000 seconds. To simulate job cost under AWS spot instance prices, we use the historical price trace of `c5.xlarge` instances in the Canada (Central) region from January to March 2021. We use the spot pricing for nodes in the `ca-central-1b` availability zone running the SUSE operating system, due to this trace's high cost (mean value of $0.158/hour) relative to the on-demand price ($0.286/h). Thus, we *conservatively* estimate DOLL's cost savings. The trace is traversed at 2000× speed; this and the short expected on-off cycle allow us to emulate the system behavior for longer training applications (e.g., wall-clock training time on the order of days and weeks). To account for this, we train relatively simple models that have significantly shorter (a few hours) runtime compared to enterprise applications. To show applicability to all SGD-based optimizers, we use both conventional SGD and the Adam optimizer (see Table II).

We set $J = 10000$, at which the CNN model achieves our target ∼85% test classification accuracy. Homogeneous arrival rates are set to $\lambda_A = 125$ with Poisson arrivals; heterogeneous Poisson arrival rates are sampled from the uniform distribution $\lambda_A \sim \mathcal{U}(1, 250)$ for each worker in each experiment. To maintain queue stability, we take group and batch sizes of $K = 20$, $b = 256$ (Proposition 1). Experiment deadlines are expressed relative to $\theta_0 = 6400$ seconds, the training time to 85% test accuracy for 64 on-demand instances.

B. Cost Optimization

To show cost savings in a variety of settings, we select three pairs of availability α and deadlines θ: $(\alpha, \theta/\theta_0) = (0.9, 1.05), (0.8, 1.05), (0.8, 1.1)$. We select α based on AWS posted frequencies of interruption, which are below 20% for most instances [39]. Training with each setting, plus on-demand-only, is repeated 12 times. We only measure worker costs, which dominate parameter and test server costs. These deadlines are relatively tight: while preemptible instances are assumed unavailable 10% to 20% of the time, we allow only 5% to 10% longer runtimes than the on-demand case.

As shown in Figure 3, cost savings of up to 30.50% can be enjoyed for relatively lax deadlines ($\theta = 1.1 \times \theta_0$, or 10% longer runtime than without preemptible instances) and high availability ($\alpha = 0.9$, or 90% preemptible instance availability) when using Theorem 4 to requisition preemptible instances. For $\alpha = 0.8$ and $\theta = 1.05 \times \theta_0$, savings are lower (13.78%). Note that these are *conservative* savings estimates due to using relatively high spot prices: on other traces, we obtain twice as much savings (Figure 5). Figure 5 shows that the empirical savings surpass Corollary 2's expected bounds, likely because Theorem 2's convergence upper bound is conservative: in practice, we need fewer model updates to achieve the desired accuracy. The variation of cost savings across different $(\alpha, \theta/\theta_0)$ settings further indicates the need to optimize the number of preemptible VMs provisioned. For example, when $\alpha = 0.8$, choosing even slightly fewer preemptible VMs (as recommended for the tighter deadline $\theta/\theta_0 = 1.05$ instead of $\theta/\theta_0 = 1.1$) changes the cost by $> 10\%$.

We next show that DOLL can also reduce costs when workers have **heterogeneous arrival rates**, e.g., due to each

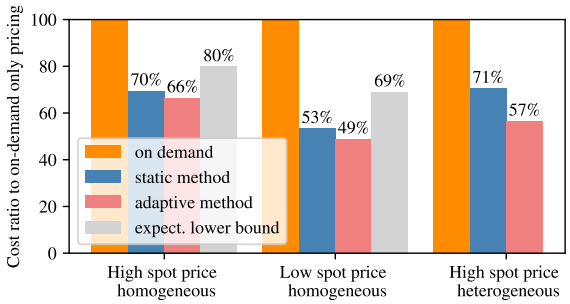


Fig. 5: Cost incurred as a % of on-demand-only cost for $\alpha = 0.9$ and $\theta/\theta_0 = 1.05$. Both the *a priori* and adaptive optimization methods achieve more savings than guaranteed by Corollary 2. "High spot price" refers to `c5.xlarge` instances running SUSE in the `ca-central-1b` availability zone, with an on-demand price of $0.286/h and a mean spot price of $0.158/h. Running another Linux OS at an on-demand price of $0.186/h and mean spot price of $0.058/h ("Low spot price"), shows Corollary 2's viability in different price settings.

TABLE II: DOLL's cost as a percentage of on-demand-only cost for $(\alpha, \theta/\theta_0) = (0.9, 1.05)$ on three datasets, with both Adam and SGD optimizers. The adaptive method nearly halves the cost relative to the on-demand-only setting and consistently achieves lower cost than the static method, whose cost is generally lower than that of Corollary 2's bound.

Dataset	Opt.	Arrival rate λ_A	K	Static method cost	Adap. method cost	Expect. lower bound
EMNIST	SGD	125	20	69.5%	66.5%	79.8%
EMNIST	SGD	$\mathcal{U}(1, 250)$	20	70.5%	56.5%	–
EMNIST	Adam	125	20	78.2%	54.6%	80.1%
I-MNIST	SGD	125	10	75.8%	54.6%	80.1%
I-MNIST	SGD	$\mathcal{U}(1, 250)$	10	77.3%	52.9%	–
CIFAR10	SGD	125	4	82.6%	57.2%	79.8%
CIFAR10	SGD	$\mathcal{U}(1, 250)$	4	64.7%	53.7%	–

worker receiving data from different geographical regions. In each training run, $\lambda_{A,n}$ for each worker is sampled anew from $\mathcal{U}(1, 250)$. Due to the wide distribution of possible arrival rates, not all runs display a cost savings relative to the on-demand-only setting (see outliers in Figure 4). However, on average, using preemptible instances will yield 18.71% to 29.46% savings relative to only using on-demand instances.

Comparing the variability of different runs in Figure 4, whose runs include randomness in the individual workers' data arrival rates and the arrival and preemption processes, with that of Figure 3a, which only includes randomness in arrivals and preemptions, shows that cost and runtime variability is significantly affected by the statistics of the individual arrival rates. Enterprises may thus wish to encourage homogeneity in arrival rates across workers by centrally routing incoming datastreams to workers. Conversely, for the same global arrival rate $\sum_{n=1}^{N_{tot}} \lambda_{A,n}$, having more heterogeneity across arrival rates can allow for more preemptible instances to be requisitioned, increasing savings (Figure 4 has lower costs than Figure 3a). The DOLL architecture supports both options.

C. Adaptive Cost Optimization

Since requisitioning preemptible instances *a priori* for heterogeneous arrival rates may cause worse performance, particularly if they are unknown, we devise an adaptive optimization method. Assuming that the time to switch a worker between an on-demand and preemptible instance is negligible at the timescale of the training task, whenever the spot prices change we use Theorems 4 and 5 to compute N_s using running tallies of $\lambda_{A,n}$ and α, then adjust instance types as needed. The adaptive method initializes with $N_s = N_{tot}$. Since we set our spot pricing bids to be the exact spot price, a spot price change may preempt every preemptible instance, allowing us to switch workers between on-demand and preemptible instances.

Experiments are run with $\alpha = 0.9, \theta = 1.05 \times \theta_0$. On average, the cost incurred using the adaptive method amounts to only 66.50% of that incurred when only using on-demand instances (Figure 5), representing a savings of 4.32% compared to the cost of using static *a priori* N_s in the homogeneous setting. Table II shows even greater savings for the adaptive method on the Infinite MNIST and CIFAR-10 datasets, for both Adam and SGD optimizers in the homogeneous case. For heterogeneous arrivals, Table II and Figure 5 show that adaptive optimization is even more beneficial due to greater randomness in the arrivals. The average adaptive cost was 56.54% of the on-demand cost in Figure 5, with 19.85% additional savings compared to *a priori* computation of N_s.

VI. Conclusion and Future Work

We propose a novel system design, DOLL, which allows the training of ML models on a distributed set of datastreams in a scalable manner. Our key theoretical contribution is the design of batching and grouping to handle large inflows of data while maintaining both data processing queue stability and a convergence guarantee. We validate with benchmark tests DOLL's ability to handle large amounts of high-throughput datastreams, and our experimental results clearly show that with our proposed requisition strategy, DOLL can achieve desired performance at a lower cost while meeting training deadlines and can adapt to unknown environment parameters.

DOLL's contributions can form part of a set of strategies to manage time and cost in enterprise-scale applications. Cross-silo federated learning, for example, introduces data heterogeneity challenges [22], while other methods can track model evolution over time. We can foresee DOLL's use on heterogeneous, federated data sources by extending the availability model for preemptible VMs to more general cases.

References

[1] M. López-Benítez, C. Majumdar, and S. N. Merchant, "Aggregated traffic models for real-world data in the internet of things," *IEEE Wireless Communications Letters*, vol. 9, no. 7, pp. 1046–1050, 2020.

[2] P. Mishra, V. Varadharajan, U. Tupakula, and E. S. Pilli, "A detailed investigation and analysis of using machine learning techniques for intrusion detection," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 686–728, 2019.

[3] K. Quach, "Vcs warn: Pumping millions into an ai startup? you mean, pumping millions into azure, aws or google cloud..." The Register, 2020, https://www.theregister.com/2020/02/20/starting_an_ai_biz/.

[4] T. Taylor, "Want to cut your cloud costs? these startups have the tools to help," TechGenix, 2021, https://techgenix.com/cut-cloud-computing-costs/.

[5] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu, "Bamboo: Making preemptible instances resilient for affordable training of large dnns," *arXiv preprint arXiv:2204.12013*, 2022.

[6] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets," in *Proceedings of ACM EuroSys*, 2017.

[7] X. Zhang, J. Wang, G. Joshi, and C. Joe-Wong, "Machine learning on volatile instances," in *Proc. of INFOCOM*, 2020.

[8] Google Cloud Platform, "Preemptible virtual machines," https://cloud.google.com/preemptible-vms/, 2021.

[9] Amazon EC2, "Amazon ec2 spot instances," https://aws.amazon.com/ec2/spot/, 2021.

[10] G. Manogaran and D. Lopez, "Health data analytics using scalable logistic regression with stochastic gradient descent," *International Journal of Advanced Intelligence Paradigms*, vol. 10, no. 1-2, pp. 118–132, 2018.

[11] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *Proceedings of USENIX NSDI*, 2022, pp. 119–135.

[12] Apache, 2021, https://kafka.apache.org/.

[13] M. Topolnik and V. Schreiner, "Tech talk: Machine learning at scale using distributed stream processing," Hazelcast Webinar, 2021, https://hazelcast.com/resources/tech-talk-machine-learning-at-scale/.

[14] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[15] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "Tr-spark: Transient computing for big data analytics," in *Proceedings of ACM SoCC*. ACM, 2016, pp. 484–496.

[16] H. Lin, H. Zhang, Y. Ma, T. He, Z. Zhang, S. Zha, and M. Li, "Dynamic mini-batch sgd for elastic distributed training: Learning in the limbo of resources," *arXiv preprint arXiv:1904.12043*, 2019.

[17] M. Atre, B. Jha, and A. Rao, "Distributed deep learning using volunteer computing-like paradigm," *arXiv preprint arXiv:2103.08894*, 2021.

[18] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd," in *Proceedings of AISTATS*, 2018.

[19] A. Saucedo, "Real time machine learning with python," EuroPython, 2020, https://ep2020.europython.eu/media/conference/slides/Ccb6D5Z-real-time-stream-processing-with-python-at-massive-scale.pdf.

[20] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches." *Journal of Machine Learning Research*, vol. 13, no. 1, pp. 165–202, 2012.

[21] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.

[22] Y. Ruan, X. Zhang, S.-C. Liang, and C. Joe-Wong, "Towards flexible device participation in federated learning," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2021, pp. 3403–3411.

[23] M. Ryabinin, E. Gorbunov, V. Plokhotnyuk, and G. Pekhimenko, "Moshpit sgd: Communication-efficient decentralized training on heterogeneous unreliable devices," *arXiv preprint arXiv:2103.03239*, 2021.

[24] E. Dall'Anese, A. Simonetto, S. Becker, and L. Madden, "Optimization and learning with information streams: Time-varying algorithms and applications," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 71–83, 2020.

[25] M. Nokleby, H. Raja, and W. U. Bajwa, "Scaling-up distributed processing of data streams for machine learning," *Proceedings of the IEEE*, vol. 108, no. 11, pp. 1984–2012, 2020.

[26] T.-H. Chang, M. Hong, H.-T. Wai, X. Zhang, and S. Lu, "Distributed learning in the nonconvex world: From batch data to streaming and beyond," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 26–38, 2020.

[27] N. Le Roux, M. Schmidt, and F. Bach, "A stochastic gradient method with an exponential convergence rate for finite training sets," *Pereira et al*, 2013.

[28] S. Shalev-Shwartz and T. Zhang, "Stochastic dual coordinate ascent methods for regularized loss minimization," *arXiv preprint arXiv:1209.1873*, 2012.

[29] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," *NeurIPS*, vol. 26, 2013.

[30] A. Defazio, F. Bach, and S. Lacoste-Julien, "Saga: A fast incremental gradient method with support for non-strongly convex composite objectives," *NeurIPS*, vol. 27, 2014.

[31] E. Jothimurugesan, A. Tahmasbi, P. Gibbons, and S. Tirthapura, "Variance-reduced stochastic gradient descent on streaming data," *NeurIPS*, vol. 31, 2018.

[32] E. Mehmood and T. Anees, "Challenges and solutions for processing real-time big data stream: A systematic literature review," *IEEE Access*, vol. 8, pp. 119 123–119 143, 2020.

[33] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–37, 2020.

[34] F. Xu, H. Zheng, H. Jiang, W. Shao, H. Liu, and Z. Zhou, "Cost-effective cloud server provisioning for predictable performance of big data analytics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 5, pp. 1036–1051, 2018.

[35] E. Volnes, T. Plagemann, and V. Goebel, "To migrate or not to migrate: An analysis of operator migration in distributed stream processing," *arXiv preprint arXiv:2203.03501*, 2022.

[36] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, "Flint: Batch-interactive data-intensive processing on transient servers," in *Proceedings of ACM EuroSys*. ACM, 2016, p. 6.

[37] D. Buniatyan, "Hyper: Distributed cloud processing for large-scale deep learning tasks," in *2019 Computer Science and Information Technologies (CSIT)*. IEEE, 2019, pp. 27–32.

[38] M. Wagenländer, L. Mai, G. Li, and P. Pietzuch, "Spotnik: Designing distributed machine learning for transient cloud resources," in *USENIX HotCloud*, 2020.

[39] Amazon EC2, "Spot instance advisor," https://aws.amazon.com/ec2/spot/instance-advisor/, 2021.

[40] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 3, pp. 1393–1402, 2013.

[41] L. Wang and Y. Chi, "Stochastic approximation and memory-limited subspace tracking for poisson streaming data," *IEEE Transactions on Signal Processing*, vol. 66, no. 4, pp. 1051–1064, 2017.

[42] D. R. Cox and W. L. Smith, "On the superposition of renewal processes," *Biometrika*, vol. 41, no. 1/2, pp. 91–99, 1954. [Online]. Available: http://www.jstor.org/stable/2333008

[43] G. Grimmett, *Probability and random processes*, 3rd ed., ser. Texts from Oxford University Press. Oxford ;: Oxford University Press, 2001.

[44] Microsoft Azure, "Use spot VMs with batch," https://docs.microsoft.com/en-us/azure/batch/batch-spot-vms, 2021.

[45] H. Jiang, X. Zhang, and C. Joe-Wong, "Doll: Distributed online learning using preemptible cloud instances - technical report," https://research.ece.cmu.edu/lions/Papers/DOLL.pdf, Carnegie Mellon University, Tech. Rep., 2021.

[46] S. Ghadimi and G. Lan. [Online]. Available: https://arxiv.org/abs/1309.5549

[47] R. G. Gallager, *Stochastic Processes: Theory for Applications*. Cambridge University Press, 2013.

[48] D. Heyman and M. Sobel, *Stochastic Models in Operations Research: Stochastic Processes and Operating Characteristics*, ser. Dover Books on Computer Science Series. Dover Publications, 2004. [Online]. Available: https://books.google.ca/books?id=IcVlwPS0qCwC

[49] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.

[50] G. Loosli, S. Canu, and L. Bottou, "Training invariant support vector machines using selective sampling," *Large scale kernel machines*, vol. 2, 2007.

[51] D. Sahoo, Q. Pham, J. Lu, and S. C. H. Hoi, "Online deep learning: Learning deep neural networks on the fly," 2017. [Online]. Available: https://arxiv.org/abs/1711.03705

[52] L. Saul, "An online passive-aggressive algorithm for difference-of-squares classification," in *NeurIPS*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 21 426–21 439. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/b2ea5e977c5fc1ccfa74171a9723dd61-Paper.pdf

[53] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html.