DOLL: Distributed OnLine Learning Using Preemptible Cloud Instances

Harry Jiang
Carnegie Mellon University
Pittsburgh, USA
hhj@alumni.cmu.edu

Xiaoxi Zhang
Sun Yat-sen University
Guangzhou, China
zhangxx89@mail.sysu.edu.cn

Carlee Joe-Wong
Carnegie Mellon University
Pittsburgh, USA
cjoewong@andrew.cmu.edu

1. INTRODUCTION

Most large-scale ML implementations scale to large amounts of data by utilizing multiple servers or virtual machines (VMs) that iteratively compute model updates on local data that are periodically synchronized. Due to the complexity of managing the resulting computing infrastructure, many companies run their ML jobs on external cloud providers' servers. However, cloud resources can be expensive, particularly for large ML jobs with long runtimes.

A particularly popular method to limit the costs of training ML jobs is to utilize *preemptible* cloud instances. These may be interrupted at the cloud provider's discretion, but they are significantly (up to 90%) cheaper than conventional on-demand instances. Most studies of these ML methods, however, assume the availability of large datasets at training time. In practice, training data may arrive at irregular intervals and models may be trained online as new data samples arrive, e.g., when monitoring data from IoT sensors. While some software frameworks like Apache Kafka can feed online data arrivals to ML algorithms, they provide little insight into the resulting costs of ML training. We extend prior work on provisioning preemptible instances to analyze available pools of data in order to run online ML on incoming 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 over preemptible instances.

Research Challenges and Our Contributions: When pools of data are readily available, the bottleneck to distributed ML training often lies in the time required for each VM to compute its model updates. In our scenario, however, *the arrival rate of incoming data may also bottleneck data processing.* An intuitive strategy would then be for each VM to process each data point as it arrives. However, since arrivals at different VMs may not be coordinated, synchronizing the model parameters at each VM between data arrivals may introduce additional delays, while asynchronous SGD methods can lead to slow convergence [1]. DOLL uses a *batching and grouping process* to limit the synchronization delay, which naturally realizes traditional mini-batch SGD so as to provide provable model convergence guarantees.

Handling online data arrivals becomes particularly challenging when we use preemptible instances to compute model updates. Existing methods utilizing preemptible instances for ML jobs largely focus on mitigating training interruptions [2] and their effects on model convergence [3]. When used on datastreams, we face an additional challenge of *interruptions pausing the data arrival process*, which impedes the rate at which we can compute model updates and thus model convergence. Thus, one should ensure that preemptions do not happen "too often," e.g., by computing some updates on on-demand instances. Our work is the first to optimize the number of preemptible VMs used and demonstrate that we can meet ML convergence guarantees.

2. DOLL SYSTEM DESIGN

2.1 System Architecture

System components: We consider a parameter server (PS) architecture [1,3] in which N distributed workers each receive a stream of data (see Figure 1). The goal is to train a model $h(\boldsymbol{z}, \boldsymbol{w})$, with input features $\boldsymbol{z}$ and trainable parameters $\boldsymbol{w}$. Each worker is either an on-demand or preemptible cloud instance of the same type, and receives a stream (series) of data samples independent and identically distributed (i.i.d) across time and data streams. The arrival times of each data sample in the N data streams are modelled as N i.i.d renewal processes, i.e., each interarrival time is i.i.d, both within and between data streams.

Training process: As is typical in ML training, we use a variant of stochastic gradient descent (SGD). The process proceeds iteratively for updates $s = 1, 2, \ldots$: each worker i continually accumulates data samples, sending a *batch indicator* to the PS every time it accumulates b samples since the last indicator. The PS periodically sends a model update trigger to all workers with the current model parameters $\boldsymbol{w}_s$. Each worker with a batch S_k then computes the gradient g_k of the loss function. Workers wait to begin the computation until receiving the update trigger to ensure they are working from the correct global model. Data samples may continue to arrive during the gradient computations. The resulting K gradients are sent to the PS, which aggregates them to perform gradient descent. The next iteration commences once the PS sends another update trigger.

We use $\left(\mu_X, \lambda_X = \frac{1}{\mu_X}, \sigma_X^2\right)$ to denote the mean, reciprocal mean, and variance of a random variable X.

Handling preemptions: Workers may be preempted by the cloud provider at any time. While preempted, they cannot receive data samples or compute model updates; however, any samples received before the preemption begins are retained, through standard fault tolerance mechanisms [4]. If a preemption occurs between the time of sending a batch

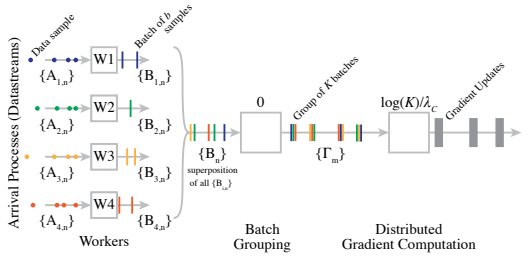

Figure 1: Data progresses through arrivals ($\{A_{i,n}\}$), batching ($\{B_{i,n}\}$), grouping ($\{\Gamma_m\}$), and gradient computations in DOLL. Data does not leave the workers; only trigger signals, local gradients, and global model weights are communicated. The numbers above the grouping and gradient computation boxes are the mean holding time for each queue.

indicator and the PS issuing an update trigger, then that worker does not return an update during this iteration.

2.2 Solution Design

We next analyze the stability of the DOLL architecture, which we show in Section 2.3 will allow us to optimize its cost. Figure 1 shows that the update process (denoted as $\{U_m\}$) is the departure process resulting from a composition of the data arrivals ($\{A_n\}$), batching of the data ($\{B_m\}$), grouping of the batches ($\{\Gamma_m\}$), and gradient computations.

We will assume that data sample arrival times at each worker are i.i.d. processes of rate λ_A (our technical report [5] considers heterogeneous, time-varying rates). For simplicity, we initially assume that workers are not preempted.

Batching: Each model update is computed on a *batch* of b data points, as in mini-batch SGD. The departure rate of batches is then λ_A/b, and $\{B_n\}$ is a superposition of the $\{B_{i,n}\}$ batching processes at the output of each worker i (Figure 1). The input process to Γ is then $\{B_n\}$. Since the superposition of N identical renewal processes tends towards a Poisson distribution as $N \to \infty$, the superposition of $\{B_{i,n}\}$, $\{B_n\}$, will be approximated by a Poisson process [6]. In practice, there are usually enough workers ($N = 64$ in Section 3) for this approximation to hold [7]. The Poisson approximation of $\{B_n\}$ will have rate $\lambda_B \equiv N\lambda_A/b$.

Grouping: We prevent backups of blocking actions by placing mini-batches of data arrivals in a queue. The PS initiates a model update once K mini-batches have arrived across all workers; such a queue will not affect system stability [5]. With $B_n \sim Pois(N\lambda_A/b)$, it follows that $\Gamma_m \sim Erlang\,(K, b\mu_A/N)$. Process $\{\Gamma_m\}$ is then stable if $\frac{b\mu_A}{N} > \mu_C$, where μ_C is the mean gradient computation time by a worker. Its departure process then has the same mean interarrival time μ_U as that of $\{\Gamma_m\}$, μ_Γ.

Model Updates: We model the event of the PS model being updated as the departure process from the grouping Γ, denoted as U. The effective update rate, mean inter-update time μ_U, and inter-update time variance σ_U^2 are then:

$$\lambda_U = \frac{N\lambda_A}{Kb},\ \mu_U = \frac{Kb\mu_A}{N},\ \sigma_U^2 \leq \frac{Kb^2\mu_A^2}{N^2} + K^2\sigma_C^2 \qquad (1)$$

for a given variance of computation time σ_C^2 at an individual worker. 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. Following the Central Limit Theorem, or large wall-clock time t, the number of updates after elapsed time t, J_t, converges in distribution to a normal distribution, or $J_t \to \mathcal{N}\left(t\lambda_U, t\sigma_J^2\right)$, where $\sigma_J^2 \equiv \sigma_U^2/\mu_U^3$. Through the properties of the normal distribution, we can also state that

$$\mathbb{E}[J_t] = t\lambda_U, \quad \mathbb{E}[runtime(J)] = J/\lambda_U. \qquad (2)$$

Preemption: With preemptible instances, the number of workers online at any given moment may not equal N. We follow prior work in assuming that preemptions do not occur during a model update [3, 8], which is justifiable if preemptions that occur during updates discard the update altogether. 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 (1) and (2):

$$\mu_U(t) = \frac{Kb\mu_A}{N(t)},\ \lambda_U(t) = \frac{N(t)\lambda_A}{Kb},\ \mathbb{E}[J_t] = \frac{\lambda_A}{Kb}\int_0^t N(\tau)d\tau \qquad (3)$$

No updates occur while $N(t) = 0$, for which $\lambda_U(t) = 0$.

We suppose that preemptions are initiated by the cloud provider in a fashion opaque to users, as is the case in practice [9]. Assuming independent preemption probabilities, we can show that the number of active workers N at any given moment follows the binomial distribution $B(N_{tot}, \alpha)$, so that $\mathbb{E}[N] = \alpha N_{tot}$. Then for a large enough time period, we can find the time required for J model updates with the Law of Large Numbers [5]: $\lim_{J\to\infty} \frac{\mathbb{E}[\Theta_J]}{J} = \frac{Kb}{\lambda_A \mathbb{E}[N]}$.

2.3 Performance Analysis

We next use our DOLL design 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:

$$\begin{aligned} \min \quad & \mathbb{E}[Cost] \\ \text{s.t.} \quad & \mathbb{E}[G(\boldsymbol{w}_{J_\theta})] - G^* \leq \epsilon \\ \text{where} \quad & \theta = \text{wall-clock time deadline}, \epsilon = \text{target error} \end{aligned} \qquad (4)$$

where G is the loss function associated with the machine learning task, and G^* its minimum.

To analyze the convergence of the model, we assume that the loss function $G(\boldsymbol{w})$ is L-Lipschitz smooth and c-strongly convex. We can upper bound the expected model error with J model updates over t time:

$$\mathbb{E}[G(\boldsymbol{w}_{J+1}) - G^*] \leq \int_0^\infty \phi(j) f_{J_t}(j) dj \quad \leq \beta e^{-\underline{\rho} t} + \gamma \qquad (5)$$

The bound is provable through the properties of a known distribution of updates in time and the properties of SGD convergence [5]. We can incorporate the impact of preemption on (5) by replacing J_t with $\mathbb{E}[J_t]$.

Since the optimization problem (4) treats cost and error in expectation, we separate our constraint into two constraints, that each deal solely with error and runtime: $\mathbb{E}[G(\boldsymbol{w}_J)] - G^* \leq \epsilon$, $\mathbb{E}[runtime(J)] \leq \theta$. We have two decision variables: N_s is the number of preemptible instances to request (out of a total of N VMs), 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.

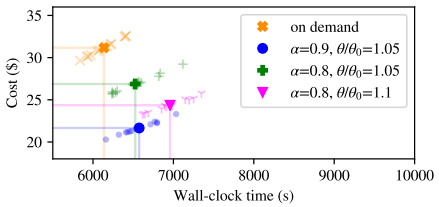

Figure 2: Using Theorem 1 for requisitioning preemptible instances yields target performances (solid dots) at a lower cost than the on-demand-only setting, costing on average 69.50% ($\alpha = 0.9$ and $\theta/\theta_0 = 1.05$), 86.22% ($\alpha = 0.8$ and $\theta/\theta_0 = 1.05$), and 78.05% ($\alpha = 0.8$ and $\theta/\theta_0 = 1.10$) of the on-demand cost. Costs diminish with higher α and θ/θ_0.

We model the cost of each preemptible instance to be a fixed unit cost per time since spot prices in practice change extremely slowly.We denote the preemptible instance posted price as $\pi_s < \pi_o$, the on-demand unit price π_o for the same VM type. Availability of preemptible instances can be modeled as a Bernoulli random variable with parameter $\alpha \in [0, 1]$

Assuming independent preemption probabilities across VMs, the number of preemptible instances available N_a for computation follows a binomial distribution $N_a \sim B(N_s, \alpha)$. From (3), the effective model update rate λ_U can be found using the expectation of $N(t)$: $\lambda_U = \frac{(N_o + \mathbb{E}[N_a])\lambda_A}{Kb} = \frac{(N_o + \alpha N_s)\lambda_A}{Kb}$

We can then achieve a close approximation of the expected cost (see the technical report [5] for details):

$$\mathbb{E}[Cost(N_s)] = \frac{J(\epsilon)Kb}{\lambda_A} \frac{N_{tot}\pi_o + (\alpha\pi_s - \pi_o)N_s}{N_{tot} + (\alpha - 1)N_s} \tag{6}$$

THEOREM 1 (OPTIMAL PREEMPTIBLE INSTANCES). *The number of preemptible instances N_s solving (4) by optimizing the formulation of expected cost in (6) is*

$$N_s^* = \min\left\{ \left\lfloor \frac{N_{tot} - \frac{J(\epsilon)Kb}{\lambda_A \theta}}{1 - \alpha} \right\rfloor, N_{tot} \right\} \tag{7}$$

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

3. EXPERIMENTS AND RESULTS

We implement DOLL on clusters with a parameter server and $N_{tot} = 64$ workers on AWS EC2 virtual machines. We train a convolutional neural network (CNN) to solve the handwriting classification problem using the Extended MNIST dataset evenly divided into N_{tot} shards and assigned to a worker. Preemption is simulated through a two-state Markov chain with an expected on-off cycle length of 1000 seconds. To simulate job cost under AWS spot instance prices, we use historical prices of `c5.xlarge` instances in the Canada (Central) region from January to March 2021.

We select $J = 10000$, at which our model achieves around a target accuracy of 85%. We take $\lambda_A = 125$ with Poisson arrivals; to maintain queue stability, we use group and batch sizes of $K = 20$, $b = 256$. Achieving target accuracy using 64 on-demand instances would take on average $\theta_0 = 6400$s.

Results. To show cost savings in a variety of settings, we select three pairs of availability α and deadlines θ: $(\alpha, \theta/\theta_0) = (0.9, 1.05), (0.8, 1.05), (0.8, 1.1)$. These values of α are consistent with AWS interruption frequencies, which are generally below 20% [10]. We repeat each experiment 12 times. Measured costs only include those of the workers, which dominate parameter and test server costs.

As shown in Figure 2, cost savings of up to 30.50% can be enjoyed for relatively lax deadlines ($\theta = 1.1 \times \theta_0$) and high availability ($\alpha = 0.9$) when using Theorem 1 to requisition preemptible instances. The variation of cost savings across different $(\alpha, \theta/\theta_0)$ settings further indicates the value of optimizing the number of preemptible instances provisioned. For example, when $\alpha = 0.8$, choosing even slightly fewer preemptible instances (by enforcing a tighter deadline) can yield $> 10\%$ difference in cost.

4. CONCLUSION AND FUTURE WORK

The contributions in this work can form part of a set of strategies used to manage time and cost in enterprise-scale applications. We can look towards applications in cross-silo settings with the additional challenge of heterogeneous communication, computing capacity, and data across workers, as considered in Federated Learning applications. We can foresee a case where DOLL is applied to heterogeneous and federated data sources, with the availability model for preemptible VMs extended to the general availability of client devices and cost generalized to metrics such as power usage.

5. ACKNOWLEDGEMENTS

This work was supported by grants NSF CNS-1751075 and NSFC 62102460.

6. REFERENCES

[1] 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.

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

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

[4] 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 *Proc. of EuroSys*, 2017.

[5] 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.

[6] D. R. Cox and W. L. Smith, "On the superposition of renewal processes," *Biometrika*, vol. 41, no. 1/2, pp. 91–99, 1954.

[7] 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.

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

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

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