Experience Paper: Towards Enhancing Cost Efficiency in Serverless Machine Learning Training

Marc Sánchez-Artigas
marc.sanchez@urv.cat
Universitat Rovira i Virgili
Spain

Pablo Gimeno Sarroca
pablo.gimeno@urv.cat
Universitat Rovira i Virgili
Spain

ABSTRACT

Function-as-a-Service (FaaS) has raised a growing interest in how to "tame" serverless to enable domain-specific use cases such as data-intensive applications and machine learning (ML), to name a few. Recently, several systems have been implemented for training ML models. Certainly, these research articles are significant steps in the correct direction. However, they do not completely answer the nagging question of when serverless ML training can be more cost-effective compared to traditional "serverful" computing. To help in this task, we propose MLLess, a FaaS-based ML training prototype built atop IBM Cloud Functions. To boost cost-efficiency, MLLess implements two key optimizations: a significance filter and a scale-in auto-tuner, and leverages them to specialize model training to the FaaS model. Our results certify that MLLess can be 15X faster than serverful ML systems [24] at a lower cost for ML models (such as sparse logistic regression and matrix factorization) that exhibit fast convergence.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning**.

KEYWORDS

Serverless computing, Machine Learnig

ACM Reference Format:
Marc Sánchez-Artigas and Pablo Gimeno Sarroca. 2021. Experience Paper: Towards Enhancing Cost Efficiency in Serverless Machine Learning Training. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3494884

1 INTRODUCTION

A vivid interest has recently arisen over the issue of serverless computing and its implications for general-purpose computations. Originally geared towards web microservices and IoT applications, recently researchers have investigated its potential in data-intensive applications [2, 9, 12, 19, 35]. Altogether, these works have led to a clear identification of "what" workloads are best suited to serverless computing.

Analogously, a recent trend on building machine learning (ML) on top of Function-as-a-Service (FaaS) platforms has emerged as a new research area [3, 5, 7, 15, 17, 36]. Since ML inference is a trivial use case of serverless computing [3, 15], attention has turned into ML model training, which is more difficult. Despite the above efforts, it still remains uncertain under what conditions ML tranining on top of FaaS may be beneficial. This is not a trivial question, as the evaluation of serverless ML training is not as simple as running VM-based ML systems such as PyTorch or TensorFlow on top of cloud functions. The main reason is that traditional ML systems have not been prepared to deal with the idiosyncrasies of the FaaS model such as the impossibility of function-to-function communication, the limited memory and transient nature of serverless functions [5, 20].

Our aim in this work is to understand the feasibility of supporting distributed ML training over FaaS platforms. Concretely, we are interested in the following question:

> *When can a FaaS platform be more cost-efficient than a VM-based, "serverful" substrate (IaaS) for distributed ML training?*

To help in this endeavor, we introduce MLLess, a prototype FaaS-based ML training system atop IBM Cloud Functions. To pick a point in the design space that is more cost-efficient than the prior serverless ML systems [5, 17, 36], MLLess comes up with two new optimizations tailored to the traits of the FaaS model. Our view is that in the same way that serverful ML training has been specialized for coarse-grained VM-based clusters, a fair comparison between FaaS and IaaS is not possible unless model training is specialized to address the limitations of the FaaS computing model. Our two novel optimizations pursue this noble goal. The first optimization reduces the bandwidth requirements of exchanging model updates between workers using shared external storage, yet assuring convergence. The rationale behind this optimization is the "stateless" essence of FaaS, which does not allow concurrent functions to directly share state. Thus, any model update (e.g., a gradient) must be exchanged through remote storage.

The second is a scale-in auto-tuner to increasingly reduce the number of workers, so as the cost of training, with no side effects on convergence. This method benefits from the "pay-per-usage" cost model of FaaS to save money, instead of the reservation-based model that charges end users for idle VM resources. From an ML perspective, FaaS thus promises more savings, since only the active workers at any given time will be billed, bringing out a superior cost-efficiency than IaaS if the pool of workers is optimally adjusted during model training.

Middleware '21, December 6–10, 2021, Virtual Event, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3494884

Next, we use MLLess to study the cost-efficiency of FaaS for ML training. Since the per-minute cost of executing a cloud function is higher than its resource-equivalent VM instance (see Table 2), we focus on *fast-convergent models* here, which intuitively are the most amenable to serverless computing. Models that take hours to converge are presumably more cost-optimal to be trained on VM instances with today's offerings.

▹ **Experimental insights.** Our study yields two key insights:

(1) *FaaS can be more cost-efficient than "serverful" libraries* such as PyTorch [24] for models that quickly converge. Although indirect communication severely penalizes FaaS-based ML training, MLLess ameliorates its impact with the aid of its two main optimizations, being 15X faster while 6.3X cheaper than PyTorch.

(2) *Specializing distributed ML training to FaaS is crucial to yield a higher cost-efficiency.* This requires dealing with low-level issues such as sparse gradients, filtering out non-significant updates, or dynamically scaling down the pool of workers, i.e., abilities that are not available in VM-based ML systems such as PyTorch.

▹ **Reproducibility and open source artifacts.** MLLess is publicly available at https://github.com/pablogs98/MLLess.

▹ **Roadmap.** The rest of the article is structured as follows: §2 discusses the challenges of FaaS for ML training. §3 presents MLLess's design. §4 details MLLess major optimizations. §5 gives implementations details, and §6 presents experimental results. §7surveys related work, and §8 concludes.

2 IS FAAS APPROPRIATE FOR ML TRAINING?

Although the main innovation of serverless is hiding servers, what makes serverless computing so powerful for training models is:

- A "pay-as-you-go" model that does not charge users for idle resources; and
- Rapid and unlimited scaling up and down of resources to zero if necessary.

By playing out with the two essential qualities to a greater or lesser extent, state-of-the-art FaaS-based ML systems [5, 17, 36] have inadvertently established a rich design space in their attempt to circumvent the stringent limitations of the FaaS model. In our case, we leverage these two properties to our favor to design our scale-in auto-tuner (§4.2). Moreover, we take the other tack and optimize indirect communication, as it is the primary training bottleneck (§4.1). Altogether, these two forces, namely auto-scalability and general performance optimization, have enabled us to show that FaaS can be more cost-efficient than "serverful" computing (IaaS).

Despite the good news, it is important to remind ourselves about the most prominent hurdles to serverless ML training. First, today's FaaS platforms only support stateless function calls with limited resources and duration. For instance, a function call in IBM Cloud Functions can use up to 2GB of RAM and must finish within 10 minutes[1]. Such limits automatically discard some natural practices such as loading all training data into local memory, while inhibiting the use of any ML framework that has not been designed with these constraints in mind [5].

[1]https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits

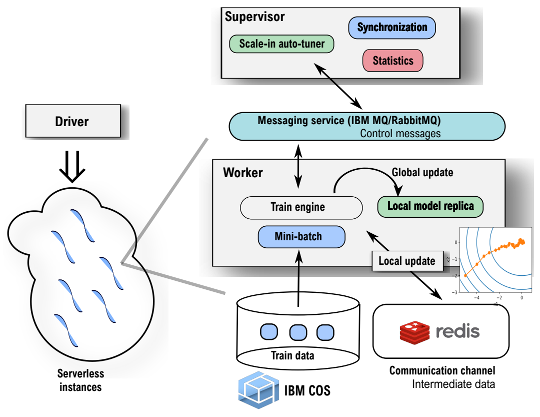


Figure 1: MLLess system architecture.

Nevertheless, the most critical issue is the impossibility of direct communication, which requires a trip through shared external storage to pass state between functions. This not only contributes significant extra latency, often hundreds of milliseconds, but also prevents exploiting HPC communication topologies adopted in ML such as tree-structured and ring-structured `all-reduce` [10].

3 MLLESS

We implement MLLess, a prototype FaaS-based ML training system built on top of IBM Cloud Functions. In this section, we describe its main components and defer the explanation of our two key optimizations to §4.

3.1 System Overview

An architectural overview of MLLess is shown in Fig. 1. MLLess consists of a *driver* that runs on the local machine of the scientist. When the user launches a ML training job, the driver invokes the requested number of serverless *workers*, who execute the job in a data-parallel manner. Each worker maintains a *local replica of the model* and uses the library of MLLess to train it. We have chosen this decentralized design for MLLess as it better abides by to a pure FaaS architecture compared to the *VM-based parameter server* [16] model, e.g., followed by other works such as Cirrus [5].

▹ **Supervisor.** Since the driver is typically far from the data center (e.g., at a university lab), tasks, such as aggregating statistics to find whether the convergence criterion has been reached, can introduce significant delays. To minimize latency, the driver also starts up a serverless function which acts as a supervisor. The role of the supervisor is to collect and aggregate statistics, synchronize worker progress, e.g., in order to bound the divergence between model copies, and terminate the training job when the stopping criteria is fulfilled, among other tasks. However, one of the core attributions of the supervisor is to automatically remove workers when their marginal contribution to convergence is minor, or even negative due to increased communication costs (please, see §4.2 for details). Although unneeded for our experiments, it would not be laborious for the supervisor to pause execution when the 10-minute timeout is close, checkpoint its internal state to storage and re-launch it as a new worker.

▹ **Communication channels.** Due to the absence of direct communication between the serverless workers or with the supervisor, MLLess makes use of three channels of indirect communication. For exchanging control messages between the workers and the supervisor, it uses a messaging service — built upon RabbitMQ[2], though it could be replaced by the native IBM's MQ messaging service[3] without complications. Second, for sharing intermediate data (e.g., local gradients) generated during model training, MLLess uses Redis[4], a low-latency, in-memory key-value store that supports thousands of requests/s [30]. Finally, it uses IBM COS — a serverless object storage service, to store the dataset mini-batches.

▹ **Synchronization.** The iterative nature of ML algorithms may imply certain dependencies across successive iterations. To keep consistency, synchronizations between workers must happen at certain boundary points. To this aim, MLLess incorporates the Bulk Synchronous Parallel (BSP) model where the workers must wait for each other at the end of every iteration. Although less strict synchronization models such as SSP [13] are easy enough to integrate, we set BSP as the default synchronization model because it simplifies the reasoning about the impact of our optimizations on model convergence. But MLLess also includes a variant of BSP where the workers only send those updates that are significant. This variant reduces communication costs, but allows local model copies to diverge across workers (see §4.1).

3.2 Model Training

MLLess assumes that a training dataset D consists of N i.i.d. data samples drawn by the underlying data distribution $\mathcal{D}$. Let $D = \{(\mathbf{v}_i \in \mathbb{R}^n, l_i \in \mathbb{R})\}_{i=1}^{N}$, where $\mathbf{v}_i$ denotes the feature vector and l_i represents the label of the i^{th} data sample. The objective of training is to find an ML model $\mathbf{x}$ that minimizes a loss function f over the dataset D: $\text{argmin}_{\mathbf{x}} \frac{1}{N} \sum_i f(\mathbf{v}_i, l_i, \mathbf{x})$.

As serverless workers have very limited memory, e.g., IBM Cloud Functions can only access at most 2 GB of local RAM, it is infeasible to replicate all training data into memory. Hence, MLLess assumes that the training dataset is stored in an object store, i.e., IBM COS, and split into mini-batches of size B. To generate the mini-batches in the appropriate format (e.g., feature normalization), MLLess leverages PyWren-IBM [33], a FaaS-based map-reduce framework. For instance, by chaining two map-reduce jobs, it is straightforward to normalize a dataset using min-max scaling, where the first map-reduce job gets the minimum and maximum values of each feature, and the second one does the actual scaling.

▹ **Job execution.** Once up and running, each worker creates a local copy of the model with the aid of the MLLess library, and starts to optimize the loss function f. In each iteration, each worker separately fetches a mini-batch from IBM COS, and then it calculates a local update from its model replica before synchronization takes place. The type of local update depends on the ML algorithm. In the case of the Stochastic Gradient Descent (SGD) [31] algorithm, local gradients are averaged to obtain a global gradient update. Due to the lack of direct communication, each worker independently of the others pulls from external storage — i.e., Redis, all the local updates, and aggregates them to update its local model copy. The availability of a local update is announced to the rest of workers through the messaging service. It is worth to note here that this decentralized design is easy to scale out, since no single component is responsible for merging all the local updates, as it occurs in LambdaML [17].

▹ **Weak scaling.** MLLess parallelism strategy keeps the mini-batch size B the same when the number of worker reduces by the action of our scale-in auto-tuner (§4.2). The reason is to avoid that every change in the number of workers incurs costly data repartitioning transfers to adjust the mini-batch size, since it may lower the net benefit of worker downscaling. Nevertheless, this entails that the global batch size B_g decreases linearly with the number of workers P, i.e., $B_g = PB$, which may affect the convergence speed of the optimizer [27]. To prevent significant deviation, the auto-tuner only removes a worker if the degradation in loss reduction does not exceed a certain threshold (see §4.2 for details).

4 OPTIMIZATIONS

FaaS is typically more expensive in terms of \$ per CPU cycle than "serverful" computing. This means that a priori, a user optimizing for cost would likely prefer IaaS over FaaS. Fortunately, FaaS-based training runtimes still show a large margin of improvement that can lead to more cost-effective training, particularly, for models that converge fast. Here we describe two optimizations to confirm this intuition. In §4.1, we elaborate on an optimization to improve throughput, and discuss the scale-in autotuner details in §4.2.

4.1 Significance Filter

As cloud providers disallow direct communications between functions, fast aggregation of gradients cannot be made with optimal primitives such as ring all-reduce [10], and must be done through external storage. Despite MLLess uses a low-latency key-value store such as Redis for this purpose, the exchange of updates is still a high-cost operation, which can significantly diminish the benefits of parallelism. This is particularly visible for the Bulk Synchronous Parallel (BSP) model of computation, where no worker can proceed to the next step without having all workers finish the current step.

To reduce strain on external storage, MLLess comes along with a variant of the Approximate Synchronous Parallel (ASP) model [14], we name it *'Insignificance-bounded Synchronous Parallel' (ISP)* to distinguish it from the original model. ASP was originally proposed to break the communication bottleneck over WANs in geo-distributed ML systems. The central idea of ASP was to remove insignificant communication across data centers, yet ensuring the correctness of ML algorithms.

Particularly, ISP borrows from ASP the idea of filtering non-significant updates, but applies it to accelerate the broadcast of local gradients between workers *within the same data center*. Therefore, ISP does not need complex synchronization mechanisms between data centers such as the ASP selective barrier and mirror [14], which simplifies its implementation.

[2]https://www.rabbitmq.com
[3]https://www.ibm.com/products/mq
[4]At the time of writing this paper, there is no serverless cache service in IBM Cloud, so users still need to provision cache instances themselves.

In a nutshell, ISP can be viewed as data reduction method, as it decreases the size of the local update after a worker goes through its mini-batch, which improves system throughput, so as job training times. This is because ISP benefits from the robustness of many ML algorithms (e.g., logistic regression, matrix factorization, collapsed Gibbs, etc.), which tolerate a bounded amount of inconsistency. To ensure equal algorithmic progress per-step, ISP enables users to tune the strictness of the significance filter to achieve the sweet spot. More concretely, the strictness is controlled by a threshold v, which is reduced over time. That is, if the original threshold is v, then the threshold value v_t at step t of the ML algorithm is given by $v_t = \frac{v}{\sqrt{t}}$.

It is important to note here that the original definition of the ASP model [14] does not presume a specific significance function, as clearly reflected in its proof of convergence in Theorem 1, which only provides a general analysis of ASP. However, to yield a more robust evidence of ISP validity, we "incarnate" the ISP model with a concrete significance filter, and prove its convergence exclusively for this function.

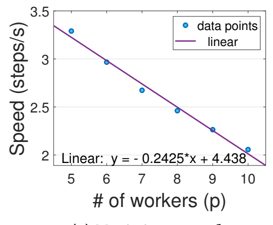
▹ **Significance function.** To trim communication while bounding deviation between any two model replicas, a "clever" compression technique is to have each worker aggregate its local updates while they are non-significant. In this way, if the accumulated update eventually becomes significant, the worker will be able to broadcast the complete history of its non-significant updates encoded as a single update, thereby minimizing both the communication burden and deviation from the "true" mini-batch gradient.

More formally, let $\mathbf{x}_t \in \mathbb{R}^n$ be the parameters of the model at step t, and $\mathbf{u}_t$ be the associated update s.t. $\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{u}_t$. As the update operation is associative and commutative, we simply aggregate the non-significant updates for any model parameter by summing them up. Eventually, the *per-parameter* accumulated update may become significant and be pushed to the rest of workers. Let t_{p_i} be the last propagation time for the i^{th} parameter. Then, we define the per-parameter significance filter as: $\left|\frac{\sum_{t'=t_{p_i}}^{t} u_{i,t'}}{x_{i,t}}\right| > v_t$.
Note that with the above significance filter, the compression factor becomes proportional to the number of accumulated updates, i.e., $m_t := (t - t_{p_i})$. This number can be arbitrarily big, provided that the magnitude of the accumulated update relative to the current model parameter value is less than v_t. For this reason, it is key to show that ISP is able to maintain an approximately-correct copy of the global model in each worker. We formulate this in Theorem 1:

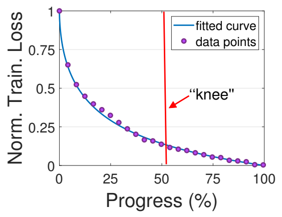
Theorem 1. *Suppose we want to find the minimizer* $\mathbf{x}^*$ *of a convex function* $f(\mathbf{x}) = \sum_{t=1}^{T} f_t(\mathbf{x})$ *(components* f_t *are also convex) via SGD on one component* ∇f_t *at a time. Also, the algorithm is replicated across* P *workers with synchronization at every step* t. *Let* $\mathbf{u}_t := -\eta_t \nabla f_t(\widetilde{\mathbf{x}}_t)$, *where the step size* η_t *decreases as* $\eta_t = \frac{\eta}{\sqrt{t}}$. *As per-parameter significance filter, we use* $\left|\frac{\delta_{i,t}}{\widetilde{x}_{i,t}}\right| > v_t$, *where* $\widetilde{x}_{i,t}$ *is the* i^{th} *parameter of the noisy state* $\widetilde{\mathbf{x}}_t := (\widetilde{x}_{0,t}, \widetilde{x}_{1,t}, \ldots, \widetilde{x}_{n,t})$ *at step* t, $\delta_{i,t} := \sum_{t'=t_{p_i}}^{t} u_{i,t'}$ *denotes the accumulated update for the* i^{th} *parameter since the last propagation time* t_{p_i}, *and* v_t *is the significance threshold that decreases as* $v_t = \frac{v}{\sqrt{t}}$. *Then, under suitable conditions:* f_t *are L-Lipschitz and the distance between any* $\mathbf{x}, \mathbf{x}'$ *in the parameter space* $D(\mathbf{x}, \mathbf{x}') \leq \Delta^2$ *for some constant* Δ:

$$R[X] := \sum_{t=1}^{T} f_t(\widetilde{\mathbf{x}}_t) - f_t(\mathbf{x}^*) = O\left(\sqrt{T}\right),$$

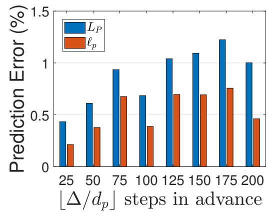
and thus $\lim_{T\to\infty} \frac{R[X]}{T} = 0$.



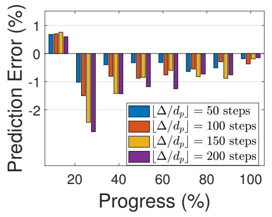
(a) Training speed.



(b) Reference curve fitting: $\theta_0 = 0.05$, $\theta_1 = 1.58$, $\theta_2 = 0.58$, $\theta_3 = 0.49$.



(c) Prediction error in estimation of 50-200 steps in advance.



(d) Prediction error of the model $\ell_p(t)$ **as job progresses.**

Figure 2: Training speed and prediction error of training a matrix factorization (PMF) model [32] on MovieLens-1M data [11].

We provide the details of the proof of Theorem 1 and the notations in Appendix A.

4.2 Scale-in Scheduler

Compared with cluster computing, one major advantage of the FaaS model is that it enables the rapid adjustment of the number of workers over time. For instance, the removal of a worker in the middle of a training job does not leave cluster resources unallocated, or demand a prompt re-allocation of them to other concurrent jobs like in the case of reserved VMs [28, 37]. This ability opens the door to the invention of novel schedulers that, for example, minimize monetary cost by dynamically adjusting the number of workers as the job progresses. This may result in a more cost-effective training, compared to traditional "serverful" cloud computing, which charge customers based on the time that the reserved VMs remain active.

To show that a better cost-efficiency ratio is possible with FaaS computing, MLLess includes a dynamic and fine-grained scheduler designed to remove "unneeded" workers. ML training is typically an iterative process where the level of quality improvement decreases as the number of training steps increases [37]. To wit, SGD reduces loss approximately as a geometric series on convex problems [4]. This implies that, while a higher number of workers is desirable during the first training steps to steeply reduce loss, a large worker pool gives only marginal returns when loss reduction slows down, which ends ups worsening the cost-efficiency ratio. In this sense, the core objective of MLLess's scale-in scheduler is to increasingly cut down the training cost as the job progresses in order to maintain cost-effectiveness.

▹ **Algorithm.** From an initial number of workers P, the scale-in scheduler dynamically reduces the worker pool based on the feedback of the ML algorithm, which includes not only the loss values but also the speed of the training steps. Using the loss information, the scheduler first detects the "knee" in the convergence rate, after which loss reduction slows down significantly, and uses the history of loss values at this time to fit the reference training loss curve $L_P(t)$. This curve will be used by the scheduler to quantify the deviation from the original convergence rate introduced by a future removal of a worker. Further, the scheduler estimates the reference step duration d_P by averaging the duration of all training steps up to this time.

After estimation of these quantities, the scheduler removes the worker with the lowest-quality replica of the model from the pool, and waits for the next scheduling interval. Now let $1 < p \leq P - 1$ denote the current number of workers. Then, the scheduler repeats the following sequence of operations upon each scheduling interval:

(1) *Estimation phase.* It fits a new training loss curve $\ell_p(t)$. But, at this time, it uses only the loss values collected so far since the last worker removal. The key reason is that the removal of a worker may affect convergence due to weak scaling [27], for it is required a new fitting to capture the potential deviation from the reference curve. Also, it estimates the current step duration d_p by the same procedure as above. Computation of this estimate is necessary as $d_p < d_P$. This occurs because the per-step communication overhead is $\widetilde{O}(p)$, where $\widetilde{O}$ hides the dependence on the model size. This is easy to see in Fig. 2a, where a matrix factorization model is trained with a varying number of workers. The figure shows how training speed decreases linearly with the number of workers. As we fix the local mini-batch size to avoid repartitioning data, less workers implies less data to pull from external storage per iteration, so as the communication overhead.

(2) *Decision phase.* In this phase, the scheduler decides to remove a new worker based on the relative error in the projected loss reduction in time horizon Δ:

$$s_\Delta(t) := \left[\frac{L_P\left(t + \left\lfloor \frac{\Delta}{d_P} \right\rfloor\right) - \ell_p\left(t + \left\lfloor \frac{\Delta}{d_p} \right\rfloor\right)}{L_P\left(t + \left\lfloor \frac{\Delta}{d_P} \right\rfloor\right)} \right], \tag{1}$$

where:

$$t := \text{current training step,}$$

$$\left\lfloor \frac{\Delta}{d_p} \right\rfloor := \text{no. of steps to be completed in } \Delta \text{ time units with } p \text{ workers,}$$

$$L_P\left(t + \left\lfloor \frac{\Delta}{d_P} \right\rfloor\right) := \text{expected loss with all } P \text{ workers,}$$

$$\ell_p\left(t + \left\lfloor \frac{\Delta}{d_p} \right\rfloor\right) := \text{expected loss with the } p \text{ workers,}$$

Then, the scaling-down condition is simply whether this term is below a certain threshold: $s_\Delta(t) < S, S \in [0, 1]$. Intuitively, this term tells how much the convergence rate of the ML algorithm may worsen with p workers compared to the original P-worker configuration in the region of slow convergence. Note that the value of $s_\Delta(t)$ can be negative, which means that system throughput is indeed better as a result of removing workers. This can happen if the decrease in the communication cost outweighs the loss of parallelism, for instance.

Finally, we want to signal that although the parameter Δ can take arbitrary values, it has been designed to anticipate the behavior of the system before a new scheduling interval arrives. Presume a fixed scheduling epoch of duration T. As a new scheduling decision can be made after time T, the idea is to set $\Delta \leq T$ to ascertain whether the removal of a worker is beneficial in a short time horizon T. In general terms, the value of Δ will vary depending upon the specific ML job. The reason is that while iterations may last 10-100 ms in some ML jobs, they may take a few seconds to complete in others. Irrespective of the ML algorithm, performing scheduling on short intervals could be disproportionally expensive due to the scheduling overhead, which involves function fitting in our case.

▹ **Loss deviation.** To predict how far a declining worker pool may deviate from the initial convergence rate, as set up in Eq. (1), the scheduler performs online fitting on two types of learning curves: the reference curve, $L_P(t)$, and the family of curves, $\{\ell_p(t)\}_{1<p\leq P-1}$, drawn as the number of workers decreases over time. To improve prediction accuracy, each type of curve has a different shape for the following reason. While $L_P(t)$ is built on the loss values from the region of fast convergence, the curves $\{\ell_p(t)\}_{1<p\leq P-1}$ are much more flat, as they correspond to the region where loss reduction slows down and stabilizes, so assuming an appropriate curve for each region makes prediction more fine-grained.

We observe that most ML jobs use first-order algorithms such as mini-batch SGD [5], which exhibits a convergence rate of $O(1/\sqrt{Bt} + 1/t)$ [23], where B denotes the mini-batch size. Consequently, we use the following model for the reference curve:

$$L_P(t) := \frac{1}{\theta_0 t^{\theta_1} + \theta_2} + \theta_3, \tag{2}$$

where $\theta_0, \theta_1, \theta_2$ and θ_3 are non-negative coefficients. An example of online curve fitting when training a PMF model is depicted in Fig. 2b. For the slow-convergence curves, we set:

$$\ell_p(t) := \frac{1}{\theta_0 t^2 + \theta_1 t + \theta_2} + \theta_3, \tag{3}$$

as in [37], where θ_0, θ_1, θ_2 and θ_3 are also non-negative. We use a non-negative least squares solver [6] to fit the points in all the curves. Before doing curve fitting, the loss values are always passed through an exponentially weighted moving average (EWMA) filter to remove outliers.

Continuing with the MF training example, Fig. 2c reports the error when estimating the loss values for an increasing number of steps ahead from the "knee". Here the prediction error is the difference between the actual and estimated loss values, divided by the actual one. As shown in Fig. 2c, both the reference curve $L_P(t)$ and the slow-convergence model $\ell_p(t)$ achieve a prediction error inferior to 1.5%, even when predicting up to 200 steps in advance.

[5]Assume the loss function f is convex, differentiable, and ∇f is Lipschitz continuous.

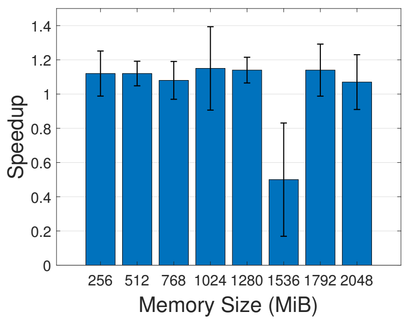


Figure 3: Speedup of two threads relative to single-thread performance within a function as memory size is varied.

Finally, Fig. 2d shows how estimation improves as more and more data points are collected for fitting the curve $\ell_p(t)$, irrespective of how many steps are predicted in advance.

▹ **Automatic "knee" detection.** To favor convergence, the scale-in scheduler never eliminates a worker before passing the "knee". The reason is to maximize the time that the ML algorithm stays within the region of fast convergence, only scaling down the number of workers once the learning curve starts to flatten out. There are several methods out there to automatically identify "knee" points from discrete data (e.g., [8] and Kneedle [34]), which can be plugged into MLLess without further adaptations. For all ML jobs considered in this work, though, a simple threshold-based heuristic on the first derivative of the learning curve, i.e., the slope of the tangent line, worked well in all cases.

▹ **Eviction policy.** By default, the scheduler eliminates the worker with the lowest-quality replica of the model from the pool. If the significance threshold $v_t > 0$, the leaving worker p stores its local replica of the model $\widetilde{\mathbf{x}}_{t,p}$ to external storage before terminating itself. Subsequently, each active worker $p' \neq p$ downloads $\widetilde{\mathbf{x}}_{t,p}$ from external storage and averages it with its local model, i.e., $\widetilde{\mathbf{x}}_{t,p'} = \frac{1}{2}\left(\widetilde{\mathbf{x}}_{t,p} + \widetilde{\mathbf{x}}_{t,p'}\right)$, to reintegrate the non-significant updates from the leaving worker into its local model. For $v_t = 0$, we note that the ISP model reduces to the BSP model (see Appendix A), and thus, this additional one-shot synchronization is unneeded.

5 IMPLEMENTATION

We implement MLLess by extending PyWren-IBM [33] — a Python-based serverless data analytics framework. Although PyWren-IBM allows users to execute user-defined functions (UDFs) as serverless workers, it is painful slow for ML training [5]. So, to make MLLess competitive with the "serverful" ML libraries, we reimplemented part of PyWren-IBM's runtime, the models and optimizers (SGD, SGD with momentum, ADAM, etc.), including sparse data structures, in Cython[6], using C-style static type declarations that allow compilation. ML frameworks such as PyTorch rely heavily on C++ and math libraries such as Intel MKL[7] to speed up computations on CPU. Thus, a pure Python implementation for MLLess would have been degraded system throughput to a large extent.

[6]http://cython.org/
[7]https://www.nsc.liu.se/software/math-libraries/

A final important observation to make is the lack of thread-level parallelism of IBM Cloud Functions. For the maximum memory allocation of 2GB, we can get the equivalent of one vCPU. This implies that we cannot exploit data parallelism within a worker as ML systems such as PyTorch do — e.g., through OpenMP. To corroborate this, we ran a small micro-benchmark. Concretely, a probabilistic matrix Factorization (PMF) [32] model was trained running SGD on either one or two threads. We measured the per-step running time of the computations inside the workers and computed the speedup of the two threads relative to single-threaded performance. The results are plotted in Fig. 3. As can be seen in the figure, PyTorch is able to extract some parallelism within a worker, but it is clearly not enough to exploit data parallelism. For workers with 1536 MiB of memory, we even found that the performance with 2 threads was worse than single-threaded performance due to a misallocation of resources.

6 EVALUATION

In this section, we compare MLLess' performance and cost-efficiency against PyTorch [24], a specialized ML library, and a non-specialized, serverless data-analytics system.

6.1 Methodology

▹ **Competing systems.** Concretely, we compare MLLess with the following implementations:

- **Distributed PyTorch [24] on CPUs**. Due to the lack of hardware accelerators such as GPUs and TPUs [21] in IBM Cloud Functions, we run PyTorch v1.8.1 with Intel MKL enabled in a cluster of VM servers utilizing all the available cores. We use the all-reduce operator of Gloo [1]—rule of thumb for CPU training—, a MPI-like library for cross-machine communication. Mini-batches are downloaded from IBM COS.
- **PyWren-IBM [33]**. We use PyWren-IBM as non-specialized serverless ML library and optimized to run on IBM Cloud Functions. Since it is a MapReduce framework, we leverage the `map` phase to process mini-batches in parallel and `reduce` tasks to aggregate the local updates. All communication is done through IBM COS, including the sharing of updates, to keep its pure serverless, general-purpose architecture.

▹ **Datasets.** We utilize three datasets in our evaluation. First, we use the **Criteo** display ads dataset [22], which contains 47M samples and has 11GB of size in total. Each sample has 13 numerical and 26 categorical features. Before training, we normalized the dataset and hashed the categorical features to a sparse vector of size 10^5 — i.e., "hashing trick". Also, we use the **MovieLens-10M** and **MovieLens-20M** datasets [11]. The former consists of 10M movie reviews from $N_u = 10,681$ users on $N_m = 71.567$ movies. The latter bears 20M reviews from $N_u = 27,278$ users on $N_m = 138,493$ movies. Notice that all the datasets are (highly)-sparse to verify MLLess support for sparse data.

▹ **ML models.** As shown in Table 1, we train different models on different datasets, i.e., Criteo for logistic regression (**LR**), and MovieLens-10M/20M for probabilistic matrix factorization (**PMF**) [32]. For **PMF**, we factorize the partially filled matrix of review ratings **R**

Table 1: ML models, datasets, and experimental settings. B **means mini-batch size, and** r **means targeted rank of PMF.**

Model	Dataset	Optimizer	# Workers	Setting
LR	Criteo	Adam	12, 24	$B = 6,250$
PMF	ML-10M	SGD + Nesterov momentum	12, 24	$B = 6,250$, $r = 20$
PMF	ML-20M	SGD + Nesterov momentum	12, 24	$B = 12K$, $r = 20$

Table 2: Pricing from IBM Cloud (`us-east`, April. 2021).

Instance type	Description	Price
C1.4x4 (4vCPUs, 4GB RAM)	MLLess messaging service	0.15 \$/hour
M1.2x16 (2vCPUs, 16GB RAM)	Redis	0.17 \$/hour
Functions (1vCPU, 2GB RAM)	MLLess worker	$3.4\text{x}10^{-5}$ \$/s (0.122 \$/hour)
B1.4x8 (4vCPUs, 8GB RAM)	PyTorch worker	0.2 \$/hour

of size $N_u \times N_m$ into two latent matrices: $\mathbf{U}_{N_u \times r}$ and $\mathbf{M}_{N_m \times r}$, such that $\mathbf{R} \approx \mathbf{UM}$.

▹ **Setup.** The VM instances used for the experiments are deployed on the IBM Cloud. When running MLLess, we use 2 VM instances: a C1.4x4 instance (4vCPUs, 4GB of RAM) to host the messaging service, and a single M1.2x16 instance (2vCPUs, 16GB of RAM) to deploy Redis, in addition to the chosen number of FaaS workers. To use as many workers for PyTorch as MLLess, the PyTorch cluster will consist of 3 or 6 B1.4x8 instances (4vCPUs, 8GB of RAM). All instances have a 1Gbps NIC. As MLLess workers, we use the largest-sized functions of 2GB of memory. All VMs and MLLess workers are deployed on the same region (`us-east`).

▹ **Cost computation.** Although IBM Cloud charges hourly per VM type, we are conservative and assume that VM cost is measured as $\$/s$. This clearly favors PyTorch, as it equates the reservation-based model of VM instances with the "pay-per-usage" model of serverless computing while the price of functions per time unit is proportionally much higher than VM instances (see Table 2). In practice, PyTorch would cost more. We do not include the cost of IBM COS as it equivalent in all systems. MLLess cost comprises the individual cost of all components —i.e., workers + the two VM instances.

▹ **Sanity check.** Before doing any experiment, we first realized a sanity check to make sure that all the models were identical in all systems. To this end, we fixed a random seed, and trained all models in each system using a single worker. We then verified that the convergence rate at each step was exactly the same in all systems. This guarantees no technical advantage of one system over the other due to subtle model artifacts such as ℓ_1- and ℓ_2-regularization, etc.

6.2 Experimental Results

▹ **Significance Filter.** We first assess the effectiveness of ISP to improve system throughput as the significance threshold v increases, i.e., it becomes more strict. As a metric, we make use of the *execution time until algorithm convergence.* For **LR**, we fix a Binary Cross Entropy (BCE) loss threshold of 0.58, and stop training when the threshold is reached. For **PMF**, we set a Root Mean Squared Error (RMSE) loss threshold of 0.82. The key point to reckon here is that by *decreasing the execution time, ISP cuts the cost forthwith*, because of the "pay-as-you-go" model of cloud functions. As a baseline, we use the BSP synchronization model.

The results are illustrated in Fig. 4. When training PMF on both MovieLens datasets, ISP is able to improve system throughput significantly with no side effects on convergence. For ML-20M, speedup reaches 3X. This result indicates that with effective optimizations in communication, which could be done automatically by the cloud provider (e.g., via canary inputs), FaaS-based ML training can be importantly enhanced despite restricted networking. As expected, the benefits of ISP on LR are small. This occurs since the size of updates is small due to the high number of zeroed features, which acts as an intrinsic filter in communication.

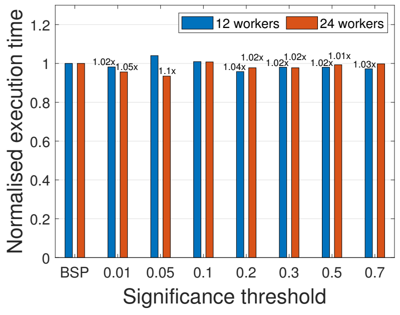
▹ **Scale-in auto-tuner.** We assess in isolation the effect of scaling down dynamically the amount of workers. To draw an unbiased picture of its performance, it is insufficient to only look at the cost profile. A bad adjustment policy could trade off convergence speed for cost, e.g., by aggressively evicting workers from the pool. Ideally, both metrics should dwindle in parallel. To capture the effect of the auto-tuner in a single metric, we use Perf/\$ defined as: Perf/\$:= $\frac{1}{\text{Exec.time(s)}} \times \frac{1}{\text{Price(\$)}}$, so that any improvement in latency, cost, or both, caused by the auto-tuner is reflected in this composite metric. We also use raw execution time as a secondary metric, to detach the \$-cost normalization effect. For Perf/\$, higher is better. As before, we run all the ML algorithms until convergence, defined as a threshold on the observed loss. Concrete values for thresholds are given in the caption of Fig 5 itself. For the scale-in auto-tuner, we set the scheduling interval to 20s and fix the parameter Δ at a half of the scheduling epoch, that is, $\Delta = 10$ sec.

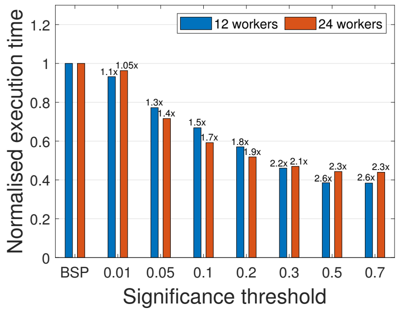
Results are illustrated in Fig 5. For LR, the results of the auto-tuner are excellent. The auto-tuner improves the Perf/\$ between 1.4X-1.5X, while reducing the running time slightly by up to 10%. For PMF, the results are also good. For all settings, the auto-tuner improves the Perf/\$. For the ML-20M dataset, it even leads to 1.6X gain since it also delivers a significant improvement in speed. The small degradation of around 7.1% in execution time for the ML-10M dataset is due to an aggressive purge of the workers too much early by the auto-tuner, which can be solved by adjusting the "knee" finder (see §4.2). Interestingly, the fact that the running time grows with more workers for the LR use case is attributable to a loss of 'statistical efficiency' [25], rather than to a deficit of scalability of a MLLess component (e.g., Redis). To verify this claim, we repeat the same experiment, but now adjusting the mini-batch size as we vary the number of workers, such that the global batch remains the same at all times. We got comparable results, as shown in Table 3.

Table 3: Execution time of LR, Criteo (BCE=0.58) as global batch remains constant. B **refers to mini-batch size.**

# Workers	12 ($B = 6,250$)	24 ($B = 3,125$)	48 ($B = 1,562$)
Execution time (s)	437.1	395.3	426.3

As a main insight, we see that for users who must curtail costs, a competent exploitation of the FaaS "pay-as-you-go" model as ours can be of great help to manage their budgets.


(a) LR, Criteo.


(b) PMF, ML-10M.

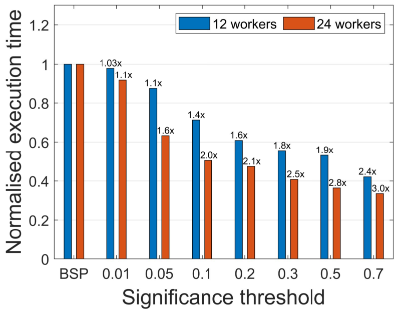
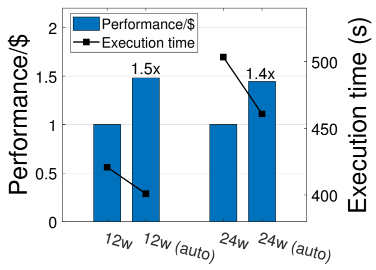

(c) PMF, ML-20M.

Figure 4: Normalized execution time until convergence as the significance threshold v increases.


(a) LR, Criteo (BCE=0.58)

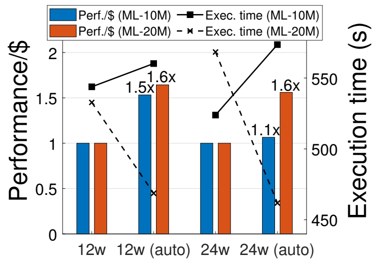

(b) PMF, ML-10M (RMSE=0.72), ML-20M (RMSE=0.77).

Figure 5: Effect of the scale-in auto-tuner. Two metrics are used: Perf/\$ (bars; left axis); execution time (lines; right axis).

▹ **Performance comparison.** To assess the benefits of a specialized system for serverless ML training, we compare MLLess against PyTorch [24] and PyWren-IBM [33]. We use PyTorch as a representative of an IaaS-based ML library. We adopt PyWren-IBM to verify that a vanilla, non-specialized design of MLLess would have been dramatically inefficient.

For this experiment, we execute three variants of MLLess. The baseline version using the BSP synchronization model, and labeled '**MLLess**' in the figures. A second variant with ISP replacing BSP, termed '**MLLess + ISP**', and a third one, with both optimizations all at once, labeled '**MLLess + All**'. For ISP, we set the significance threshold $v = 0.7$. For the auto-tuner, we set the scheduling epoch to 20s with $\Delta = 10$s. For all the systems, we only report the results for $P = 24$ workers. The trends were similar for 12 workers.

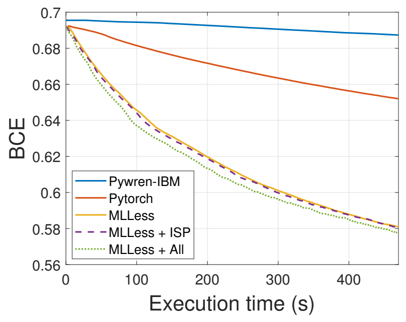
The results are shown in Fig. 6. The first observation to be made is that PyWren-IBM is very inefficient in all jobs. This is mostly due to two facts. The first is that local updates are communicated across workers through slow storage only, i.e., IBM COS. The second is the non-specialization of PyWren-IBM for iterative ML training.

The second observation to be made is that MLLess is able to converge significantly faster than PyTorch. To give a sense of the performance gap, let us focus on the **PMF+ML-10M** application. To achieve a loss value of 0.9, MLLess needs 23 seconds while PyTorch gets to this loss only after 90 seconds. This gap increases over time and to converge to a "prudent" RMSE loss of 0.738, PyTorch spends 2,029 seconds. MLLess, though, reaches this loss value after 140 seconds. This yields a speedup of 14.49X.

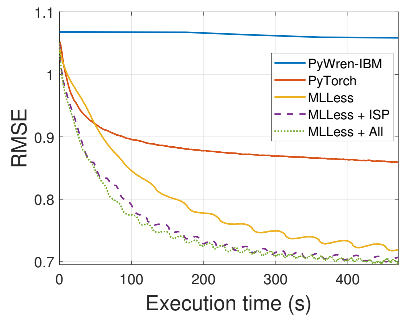
For the **PMF+ML-20M** job, we get similar results. To reach a loss of 0.821, PyTorch spends 1,800 seconds. MLLess gets to this loss within 115 seconds, 15.65X faster than PyTorch. Via thorough analysis, we found that PyTorch's speed is affected by the high sparsity of the datasets as it occurs to TensorFlow [18]. Unlike PyTorch, MLLess employs Cython to directly operate on sparse data and sparse gradients, and hence, save significant time on serializing and deserializing data. In this way, MLLess leads to faster convergence. Either way, the gap between plain MLLess and the optimizations is significant for PMF, which demonstrates that an optimized treatment of sparsity by its own cannot realize such savings. To wit, plain MLLess spends 334 seconds to reduce RMSE to 0.821, 3X slower than with all the optimizations present.

As a final observation, it is worth to note that the auto-tuner does not slow down convergence in any job as shown by the '**MLLess + All**' curves. On the contrary, it helps to improve convergence speed in addition to reduce cost. Also, the use of ISP consistency for large models such as ML-20M has been vital to ensure fast convergence for the few initial seconds.

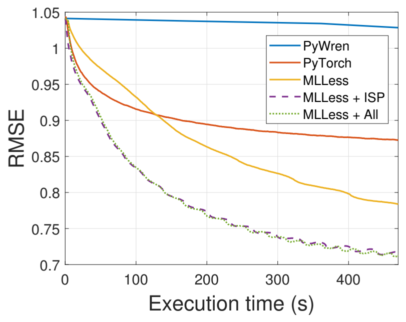
▹ **Cost comparison.** Although researchers are more averse in the cost dimension than performance analysis, reduced cost is of vital importance to the value proposition of any cloud vendor. Following the same path traced above, here we compare MLLess against PyTorch and PyWren-IBM in terms of cost. We extract the cost of each system from the executions in the prior evaluation to ease cross comparison. As a headline observation, MLLess is cheaper than



(a) LR, Criteo.

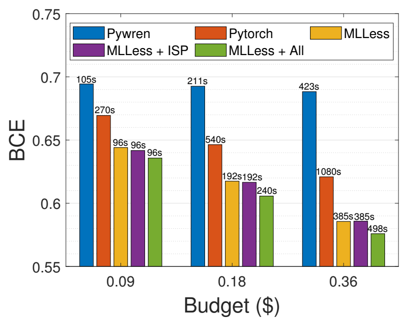


(b) PMF, ML-10M.

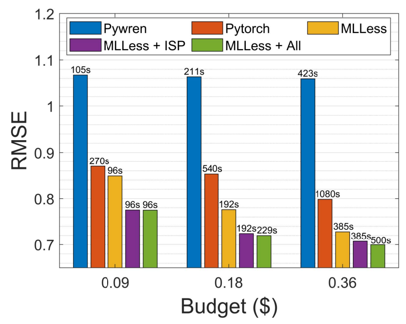


(c) PMF, ML-20M.

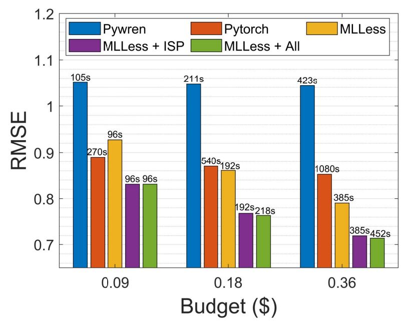
Figure 6: Loss vs. time comparison between PyTorch, PyWren-IBM and MLLess with different variants: BSP synchronization (MLLess), ISP synchronization (MLLess + ISP) and ISP synchronization + auto-tuner (MLLess + All), for 24 workers.



(a) LR, Criteo.



(b) PMF, ML-10M.



(c) PMF, ML-20M.

Figure 7: Cost vs. loss comparison between PyTorch, PyWren-IBM and MLLess with different variants: BSP synchronization (MLLess), ISP synchronization (MLLess + ISP) and ISP synchronization + auto-tuner (MLLess + All), for 24 workers. The numbers above the bars report the maximum execution time affordable with each possible budget.

PyTorch in all applications, but the improvement gap is not as big as in the performance dimension. For example, when training **PMF** on **ML-20M**, MLLess spends \$0.0948 to reach a loss of 0.82, compared to the \$0.6 invested by PyTorch. This gives a 6.32X savings on cost. Analogously, PyTorch spends \$0.667 to achieve to a loss value of 0.738 for the **PMF+ML-10M** job, while MLLess cuts this cost to \$0.1348, 4.94X cheaper than PyTorch.

While MLLess saves money, for some users the "pay-as-you-go" model is in conflict with the way they manage their budgets. For instance, these may be fixed in advance. Thus, it is interesting to examine what would be the performance of MLLess for a fixed budget. To answer this question, Fig. 7 shows to what extent each system is able to converge under a fixed budget in dollars. The numbers above the bars report the maximum execution time affordable with each possible budget. As can be seen in the figure, **MLLess + All** provides the best cost-performance trade-off in all applications, even for the tiny budget of 9 cents. Non-surprisingly, PyTorch is able to run longer than the rest of systems due to the lower pricing of the rented VM instances. For the largest budget, it even doubles the maximum running time affordable by the FaaS-based ML systems. Per contra, MLLess is significantly more efficient per unit of time and better adjusts to the cost plan. Interestingly, the auto-tuner helps to gain some extra seconds, up to 115 seconds, as shown by the **MLLess + All**-labeled bars. This is another experimental evidence of the economic utility of our scale-in auto-tuner.

▹ **Summary of results.** Our results confirm our hypothesis that:

- *FaaS can be more cost-efficient than IaaS* for ML models that exhibit rapid convergence such as sparse logistic regression and probabilistic matrix factorization;
- But very importantly, this is only achievable if *ML training is specialized to FaaS.*

This is clear for probabilistic matrix factorization. As illustrated in Fig. 6c, while the use of sparse representations for gradients and models leads to a certain advantage over PyTorch after 125 seconds, only when communication efficiency is increased (**MLLess + ISP**), convergence is highly accelerated. Also, our experiments show that the pool of FaaS workers can be gently shrunk without impairing ML model convergence, which further raises cost-efficiency. This contrasts with standard IaaS VM-based rental, which incurs fixed costs during model training.

7 RELATED WORK

▹ **Serverless Data Processing.** A large bulk of previous works have proposed high-level frameworks for running large-scale analytics on serverless functions. To wit, PyWren [19, 33] is a map-reduce framework running over FaaS executors that takes advantage of object storage to store input/output data. gg [9] is a library that uses AWS Lambda workers for CPU-bound intensive jobs (e.g., video-encoding). Numpywren [35] is an elastic linear algebra library on top of a pure serverless architecture. Crucial [2] is a framework for building stateful FaaS-based distributed applications. Starling [29] proposes a serverless query execution engine. Serverless ML systems, including MLLess, build upon the lessons learned from these works to increase their performance and cost-efficiency.

▹**Serverless ML.** A few number of works have been devoted to leveraging FaaS platforms for building ML systems. Since ML model inference is a unequivocal use case of serverless computing [3, 15], recent research efforts have been directed towards ML model training [5, 17, 36]. All these works make use of AWS Lambda, which gives them some advantage over MLLess, and make direct comparison problematic. First off, AWS Lambda allows to exploit multi-threaded parallelism [5, 26], while IBM Cloud Functions do not. Furthermore, AWS Lambda workers can access 8GB of extra local RAM — i.e., IBM Cloud Functions are restricted to 2GB of memory, which allows them to hold larger data partitions and mini-batches compared with MLLess. Despite this, MLLess optimizations outweigh these limitations and manage to deliver speedups superior to 15X and with 6.3X lower cost than PyTorch, and very importantly, excluding the startup time, which is longer in PyTorch (e.g., a cluster of 6 VMs takes > 1 min. to boot up).

Shortly, Cirrus [5] is a serverless ML system that implements a parameter server (PS) [16] on top of VMs, where all FaaS workers communicate with this centralized PS layer. Such a hybrid design has its merit, mainly because the ability of PS servers of doing computation delivers 200% communication savings compared with indirect communication via external storage. According to [20], Cirrus is 3X-5X faster than VMs, but up to 7X more costly, that is, less cost-effective than MLLess.

Siren [36] presents an asynchronous ML framework, where each worker runs independently, i.e., it reads a (stale) model from remote storage (e.g., AWS S3), updates it with a mini-batch of local data, writing the new model back to storage. Its major strength is withal its scheduler built upon reinforcement learning (RL) that adjusts the number of workers dynamically, subject to a certain budget. Compared to MLLess, its scheduler is more coarse-grained as it adjusts the number of workers once per epoch, and achieves a lower cost-efficient ratio. Concretely, Siren reduces job execution time by up to 44.3% at the same cost than EC2 clusters.

Almost in parallel with this article, [17] presents LambdaML, a FaaS-based training system to determine the cases where FaaS holds a sway over IaaS. Very interestingly, our results mirrors its conclusion that FaaS is more cost-efficient for models that quickly converge. Unlike LambdaML, though, we reach this conclusion by getting out of the equation start-up times. That is, if the startup time is excluded, LamdaML is slower than PyTorch, while MLLess is equally faster than PyTorch, yet relatively cheaper. In this sense, we believe that MLLess represents a step forward in serverless ML training.

8 CONCLUSION

In this work, we have examined the question of whether serverless ML training can be more cost-effective compared with traditional cluster computing. To answer this question, we have developed MLLess, a prototype system of FaaS-based ML model training built on top of IBM Cloud Functions, and empowered it with two new optimizations: one aimed to reduce communication bandwidth, the other intended to exploit the essential qualities of the FaaS model to jointly decrease cost and execution time. Our results show that MLLess is more cost-efficient than serverful ML libraries at a lower cost for ML models with fast convergence. This result is promising, because if cost-effectiveness for ML can be added up to the features of the serverless computing model, infrequent ML workloads will be handled more effectively than persistent VMs. Areas for future work include the extension of MLLess to deep learning and bursty ML workloads.

ACKNOWLEDGMENTS

We appreciate all anonymous reviewers at Middleware'21, who provide insightful feedback that makes this paper much stronger. This work has been partially supported by EU (No. 825184) and Spanish Government (No. PID2019-106774RB-C22). Marc Sánchez-Artigas is a Serra Húnter Fellow.

REFERENCES

[1] 2021. Gloo: a collective communications library. https://github.com/facebookincubator/gloo. [Online; accessed 22-February-2021].
[2] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *20th International Middleware Conference (Middleware '19)*. 41–54.
[3] Anirban Bhattacharjee, Yogesh Barve, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. 2019. Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks. In *2019 USENIX Conference on Operational Machine Learning (OpML'19)*. 59–61.
[4] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press, USA.
[5] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *ACM Symposium on Cloud Computing (SoCC '19)*. 13–24.
[6] The SciPy community. 2021. Scipy.optimize.curve_fit. SciPy V1.6.3 Reference Guide. [Online; accessed 28-April-2021].
[7] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. 334–341.
[8] Paula Fermín-Cueto, Euan McTurk, Michael Allerhand, Encarni Medina-Lopez, Miguel F. Anjos, Joel Sylvester, and Gonçalo dos Reis. 2020. Identification and machine learning prediction of knee-point and knee-onset in capacity degradation curves of lithium-ion cells. *Energy and AI* 1 (2020), 100006.
[9] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 475–488.
[10] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. http://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/. [Online; accessed 20-February-2021].
[11] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. 5, 4, Article 19 (Dec. 2015), 19 pages.
[12] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*.
[13] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More

Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *26th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'13)*. 1223–1231.

[14] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 629–647.

[15] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. In *IEEE International Conference on Cloud Engineering (IC2E'18)*. 257–262.

[16] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-Aware Distributed Parameter Servers. In *2017 ACM International Conference on Management of Data (SIGMOD '17)*. 463–478.

[17] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'21)*.

[18] Biye Jiang et al. 2019. XDL: An Industrial Deep Learning Framework for High-Dimensional Sparse Data. In *1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data (DLP-KDD '19)*. Article 6, 9 pages.

[19] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *2017 Symposium on Cloud Computing (SoCC'17)*. 445–451.

[20] Eric Jonas et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[21] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12.

[22] Criteo Labs. 2014. Kaggle Display Advertising Challenge Dataset. http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/. [Online; accessed 15-May-2021].

[23] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. 2014. Efficient Mini-Batch Training for Stochastic Optimization. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. 661–670.

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3005–3018.

[25] Dominic Masters and Carlo Luschi. 2018. Revisiting Small Batch Training for Deep Neural Networks. *CoRR* abs/1804.07612 (2018). http://arxiv.org/abs/1804.07612

[26] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 115–130.

[27] Andrew Or, Haoyu Zhang, and Michael Freedman. 2020. Resource Elasticity in Distributed Deep Learning. In *Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 400–411. https://proceedings.mlsys.org/paper/2020/file/006f52e9102a8d3be2fe5614f42ba989-Paper.pdf

[28] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Thirteenth EuroSys Conference (EuroSys '18)*. Article 3, 14 pages.

[29] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 131–141.

[30] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 193–206.

[31] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *The Annals of Mathematical Statistics* 22, 3 (1951), 400–407.

[32] Ruslan Salakhutdinov and Andriy Mnih. 2007. Probabilistic Matrix Factorization. In *20th International Conference on Neural Information Processing Systems (NIPS'07)*. 1257–1264.

[33] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *19th International Middleware Conference Industry (Middleware'18)*. 1–8.

[34] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *31st International Conference on Distributed Computing Systems Workshops (ICDCSW '11)*. 166–171.

[35] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *11th ACM Symposium on Cloud Computing (SoCC '20)*. 281–295.

[36] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 1288–1296.

[37] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. 2017. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *2017 Symposium on Cloud Computing (SoCC'17)*. 390–404.

A CONVERGENCE ANALYSIS

In this section, we show that the SGD algorithm for convex objectives is guaranteed to converge under our consistency model. Recall that a step t of the SGD algorithm is defined as:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f_t(\mathbf{x}_t) = \mathbf{x}_{t-1} - \eta_t \mathbf{g}_t = \mathbf{x}_{t-1} + \mathbf{u}_t,$$

where n_t is the step size and $\mathbf{u}_t := -\eta_t \mathbf{g}_t$ is the update of step t.

As described in Section 4.1, the accumulated updates are broadcast to the rest of workers only in the case that they are significant. This means that significant updates are always seen by all workers. However, insignificant updates remain local to the workers, so different workers will "see" different, noisy versions of the true state $\mathbf{x}_t$.

To formally capture the difference between the "true" state $\mathbf{x}_t$ and the noisy views, let us define an order of the updates up to step t. Suppose that the algorithm is distributed across P workers, and the logical clocks that mark progress start at 0. Then,

$$\mathbf{u}_t := \mathbf{u}_{p,c} := \mathbf{u}_{t \bmod P, \lfloor \frac{t}{P} \rfloor},$$

defines a mapping between step t and $[0, P-1] \times \mathbb{N}_0$, which loops through clocks ($c = \lfloor \frac{t}{P} \rfloor$), and for each clock c loops through workers ($p = t \bmod P$).

We now define a reference sequence of states that a single-worker serial execution would follow if the updates were to be seen under the above ordering: $\mathbf{x}_t = \mathbf{x}_0 + \sum_{t=0}^{t'} \mathbf{u}_t$. Let $\mathcal{S}_{p,c}$ denote the set of significant updates propagated by worker p up through clock c. Similarly, let $\mathcal{I}_{p,c}$ denote the set of the insignificant updates up through clock c not broadcast from worker p. Clearly, $\mathcal{S}_{p,c}$ and $\mathcal{I}_{p,c}$ are disjoint, and their union includes all the updates accumulated by p until exactly clock c.

Using the above notation, we define the noisy view $\widetilde{\mathbf{x}}_t$ as:

$$\widetilde{\mathbf{x}}_{p,c} := \mathbf{x}_0 + \sum_{c'=0}^{c} \mathbf{u}_{p,c} + \sum_{p' \neq p} \sum_{i \in \mathcal{S}_{p',c}} \mathbf{u}_i, \quad (4)$$

where $\mathbf{x}_0$ are the initial parameters, the second term refers to the local updates applied by worker p, and the last term aggregates all the significant updates shared by the rest of workers other than p.

Finally, by using Eq. (4), the difference between the "true" view $\mathbf{x}_t$ and the noisy view $\widetilde{\mathbf{x}}_t$ becomes:

$$\widetilde{\mathbf{x}}_t - \mathbf{x}_t = \widetilde{\mathbf{x}}_{p,c} - \mathbf{x}_t = \widetilde{\mathbf{x}}_{t \bmod P, \lfloor \frac{t}{P} \rfloor} - \mathbf{x}_t = -\sum_{p' \neq p} \sum_{i \in \mathcal{I}_{p',c}} \mathbf{u}_i \quad (5)$$

Equipped with Eq. (5), we are now ready to start the proof of Theorem 1.

Similarly to [13], the Insignificance-bounded Synchronous Parallel (ISP) generalizes the BSP model:

Corollary. *For zero significance threshold $v = 0$, ISP reduces to BSP.*

Proof. $v = 0$ implies that the set $\mathcal{I}_{p,c} = \emptyset$ at all clocks, so that $\sum_{p' \neq p} \sum_{i \in \mathcal{S}_{p',c}} \mathbf{u}_i = \sum_{c'=0}^{c} \sum_{p' < p} \mathbf{u}_{p',c'}$. Therefore, $\widetilde{\mathbf{x}}_{p,c}$ exactly consists of all updates until the current clock. □

Theorem 1. *Suppose we want to find the minimizer $\mathbf{x}^*$ of a convex function $f(\mathbf{x}) = \sum_{t=1}^{T} f_t(\mathbf{x})$ (components f_t are also convex) via SGD on one component ∇f_t at a time. Also, the algorithm is replicated* *across P workers with synchronization at every step t. Let $\mathbf{u}_t := -\eta_t \nabla f_t(\widetilde{\mathbf{x}}_t)$, where the step size η_t decreases as $\eta_t = \frac{\eta}{\sqrt{t}}$. As per-parameter significance filter, we use $\left|\frac{\delta_{i,t}}{\widetilde{x}_{i,t}}\right| > v_t$, where $\widetilde{x}_{i,t}$ is the i^{th} parameter of the noisy state $\widetilde{\mathbf{x}}_t := (\widetilde{x}_{0,t}, \widetilde{x}_{1,t}, \ldots, \widetilde{x}_{n,t})$ at step t, $\delta_{i,t} := \sum_{t'=t_{p_i}}^{t} u_{i,t'}$ denotes the accumulated update for the i^{th} parameter since the last propagation time t_{p_i}, and v_t is the significance threshold that decreases as $v_t = \frac{v}{\sqrt{t}}$. Then, under suitable conditions: f_t are L-Lipschitz and the distance between any $\mathbf{x}, \mathbf{x}'$ in the parameter space $D(\mathbf{x}, \mathbf{x}') \le \Delta^2$ for some constant Δ:*

$$R[X] := \sum_{t=1}^{T} f_t(\widetilde{\mathbf{x}}_t) - f_t(\mathbf{x}^*) = O\left(\sqrt{T}\right),$$

and thus $\lim_{T\to\infty} \frac{R[X]}{T} = 0$.

Proof. We follow the proof of [13]. Define $D(\mathbf{x}, \mathbf{x}') := \frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|^2$, where $\|\cdot\|$ is the ℓ_2-norm. Because f_t are convex, we have:

$$R[X] := \sum_{t=1}^{T} f_t(\widetilde{\mathbf{x}}_t) - f_t(\mathbf{x}^*) \le \sum_{t=1}^{T} \left\langle \widetilde{\mathbf{g}}_t, \widetilde{\mathbf{x}}_t - \mathbf{x}^* \right\rangle.$$

The high level idea is to show that $R[X] = O\left(\sqrt{T}\right)$, which means $\mathbb{E}_t\left[f_t(\widetilde{\mathbf{x}}_t) - f_t(\mathbf{x}^*)\right] \to 0$, thus convergence. First, we shall something about the term $\langle \widetilde{\mathbf{g}}_t, \widetilde{\mathbf{x}}_t - \mathbf{x}^* \rangle$.

Lemma 1. *If $X = \mathbb{R}^n$, then for all $t > 0$:*

$$\left\langle \widetilde{\mathbf{x}}_t - \mathbf{x}^*, \widetilde{\mathbf{g}}_t \right\rangle = \frac{1}{2}\eta_t \|\widetilde{\mathbf{g}}_t\|^2 + \frac{D(\mathbf{x}^*, \mathbf{x}_t) - D(\mathbf{x}^*, \mathbf{x}_{t+1})}{\eta_t} + \sum_{p' \neq p}\left[- \sum_{i \in \mathcal{I}_{p',c}} \eta_i \left\langle \widetilde{\mathbf{g}}_i, \widetilde{\mathbf{g}}_t \right\rangle \right].$$

Proof.

$$\begin{aligned}
D(\mathbf{x}^*, \mathbf{x}_{t+1}) - D(\mathbf{x}^*, \mathbf{x}_t) &= \frac{1}{2}\|\mathbf{x}^* - \mathbf{x}_t + \mathbf{x}_t - \mathbf{x}_{t+1}\| - \frac{1}{2}\|\mathbf{x}^* - \mathbf{x}_t\| \\
&= \frac{1}{2}\|\mathbf{x}^* - \mathbf{x}_t + \eta_t \widetilde{\mathbf{g}}_t\| - \frac{1}{2}\|\mathbf{x}^* - \mathbf{x}_t\| \\
&= \frac{1}{2}\eta_t^2 \|\widetilde{\mathbf{g}}_t\|^2 - \eta_t \left\langle \mathbf{x}_t - \mathbf{x}^*, \widetilde{\mathbf{g}}_t \right\rangle \\
&= \frac{1}{2}\eta_t^2 \|\widetilde{\mathbf{g}}_t\|^2 - \eta_t \left\langle \mathbf{x}_t - \widetilde{\mathbf{x}}_t, \widetilde{\mathbf{g}}_t \right\rangle - \eta_t \left\langle \widetilde{\mathbf{x}}_t - \mathbf{x}^*, \widetilde{\mathbf{g}}_t \right\rangle.
\end{aligned}$$

By expanding the second term:

$$\left\langle \mathbf{x}_t - \widetilde{\mathbf{x}}_t, \widetilde{\mathbf{g}}_t \right\rangle = \left\langle \left[\sum_{p' \neq p} \sum_{i \in \mathcal{I}_{p',c}} \eta_i \widetilde{\mathbf{g}}_i \right], \widetilde{\mathbf{g}}_t \right\rangle = \sum_{p' \neq p} \sum_{i \in \mathcal{I}_{p',c}} \eta_i \left\langle \widetilde{\mathbf{g}}_i, \widetilde{\mathbf{g}}_t \right\rangle,$$

and moving $\langle \widetilde{\mathbf{x}}_t - \mathbf{x}^*, \widetilde{\mathbf{g}}_t \rangle$ to the left, we prove the lemma. □

Returning to the proof of the theorem, we use Lemma 1 to expand the regret $R[X]$:

$$R[X] \le \sum_{t=1}^{T} \left\langle \widetilde{\mathbf{g}}_t, \widetilde{\mathbf{x}}_t - \mathbf{x}^* \right\rangle = \sum_{t=1}^{T} \left(\frac{1}{2}\eta_t \|\widetilde{\mathbf{g}}_t\|^2 + \frac{D(\mathbf{x}^*, \mathbf{x}_t) - D(\mathbf{x}^*, \mathbf{x}_{t+1})}{\eta_t} + \sum_{p' \neq p}\left[- \sum_{i \in \mathcal{I}_{p',c}} \eta_i \left\langle \widetilde{\mathbf{g}}_i, \widetilde{\mathbf{g}}_t \right\rangle \right] \right).$$

We now upper-bound each of the terms:

$$\begin{aligned}
\sum_{t=1}^{T} \frac{1}{2}\eta_t \|\widetilde{\mathbf{g}}_t\|^2 &\le \sum_{t=1}^{T} \frac{1}{2}\eta_t L^2 \qquad (L\text{–Lipschitz assumption}) \\
&= \frac{1}{2}\eta L^2 \sum_{t=1}^{T} \frac{1}{\sqrt{t}} \le \eta L^2 \sqrt{T}, \quad \left(\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \le 2\sqrt{T} \right)
\end{aligned} \tag{6}$$

and

$$\begin{aligned}
&\sum_{t=1}^{T} \frac{D(\mathbf{x}^*, \mathbf{x}_t) - D(\mathbf{x}^*, \mathbf{x}_{t+1})}{\eta_t} = \\
&\frac{D(\mathbf{x}^*, \mathbf{x}_1)}{\eta_1} - \frac{D(\mathbf{x}^*, \mathbf{x}_{T+1})}{\eta_T} + \sum_{t=2}^{T} \left[D(\mathbf{x}^*, \mathbf{x}_t) \left(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}} \right) \right] \\
&\le \frac{\Delta^2}{\eta} - 0 + \frac{\Delta^2}{\eta} \sum_{t=2}^{T} \left[\sqrt{t} - \sqrt{t-1} \right] \text{ (Bounded diameter)} \\
&= \frac{\Delta^2}{\eta} + \frac{\Delta^2}{\eta}\left[\sqrt{T} - 1\right] = \frac{\Delta^2}{\eta}\sqrt{T}.
\end{aligned} \tag{7}$$

For the last term, we shall use the following lemma:

Lemma 2. *Let V a normed vector space with norm $\|\cdot\|$. Let $\mathbf{y}$ be a vector in V having nonzero entries. For all $\mathbf{x} \in V$, we have that $\frac{\|\mathbf{x}\|}{\|\mathbf{y}\|} \le \|\mathbf{x} \oslash \mathbf{y}\|$, where $\oslash$ denotes Hadamard division.*

Proof. By contradiction. That is, suppose that $\frac{\|\mathbf{x}\|}{\|\mathbf{y}\|} > \|\mathbf{x} \oslash \mathbf{y}\|$. Then, $\|\mathbf{x}\| > \left\|\mathbf{x} \oslash \frac{\mathbf{y}}{\|\mathbf{y}\|}\right\|$, which is a contradiction since $\frac{\mathbf{y}}{\|\mathbf{y}\|}$ is a vector of unit norm. □

We are now in position to upper bound the last term. For simplicity, let $\mathbf{s}_{p',c} := -\sum_{i \in \mathcal{I}_{p',c}} \eta_i \widetilde{\mathbf{g}}_i$ and $|\cdot|$ denote ℓ_1-norm or *taxicab* norm Then,

$$\begin{aligned}
&\sum_{t=1}^{T} \sum_{p' \neq p} \left[- \sum_{i \in \mathcal{I}_{p',c}} \eta_i \left\langle \widetilde{\mathbf{g}}_i, \widetilde{\mathbf{g}}_t \right\rangle \right] \\
&\le \sum_{t=1}^{T} (P-1) \left\langle \mathbf{s}_{p',c}, \widetilde{\mathbf{g}}_t \right\rangle \le (P-1) \sum_{t=1}^{T} \left| \left\langle \mathbf{s}_{p',c}, \widetilde{\mathbf{g}}_t \right\rangle \right| \\
&\le (P-1) \sum_{t=1}^{T} \|\mathbf{s}_{p',c}\| \|\widetilde{\mathbf{g}}_t\| \quad \text{(Cauchy–Schwarz inequality)} \\
&\le (P-1)\, L \sum_{t=1}^{T} \frac{|\mathbf{s}_{p',c}|}{|\widetilde{\mathbf{x}}_t|} |\widetilde{\mathbf{x}}_t| \qquad (L\text{–Lipschitz assumption}) \\
&\le (P-1)\, L \sum_{t=1}^{T} \left| \mathbf{s}_{p',c} \oslash \widetilde{\mathbf{x}}_t \right| |\widetilde{\mathbf{x}}_t| \quad \text{(Lemma 2)} \\
&\le (P-1)\, Ln \sum_{t=1}^{T} v_t |\widetilde{\mathbf{x}}_t| \qquad \text{(Non-significance of updates)} \\
&\le (P-1)\, Ln \sum_{t=1}^{T} v_t \sqrt{n} \|\widetilde{\mathbf{x}}_t\|
\end{aligned} \tag{8}$$

$$
\begin{aligned}
&\leq (P-1)\, L \left(n\sqrt{n}\right) \sum_{t=1}^{T} v_t \sqrt{2}\Delta \qquad \text{(Bounded diameter)} \\
&\leq \sqrt{2}\Delta\, (P-1)\, L \left(n\sqrt{n}\right) \sum_{t=1}^{T} \frac{v}{\sqrt{t}} \\
&\leq 2\sqrt{2}\Delta\, (P-1)\, L \left(n\sqrt{n}\right) v\sqrt{T}. \quad \left(\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{T}\right) \qquad (9)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
R[X] &\leq \eta L^2 \sqrt{T} + \frac{\Delta^2}{\eta}\sqrt{T} \\
&\quad + 2\sqrt{2}\Delta\, (P-1)\, L \left(n\sqrt{n}\right) v\sqrt{T} = O\left(\sqrt{T}\right), \qquad (10)
\end{aligned}
$$

and thus, $\lim_{T\to\infty} \frac{R[X]}{T} = 0$, which concludes the proof. □