An update to this article is included at the end

Journal of Parallel and Distributed Computing 183 (2024) 104764

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

MLLess: Achieving cost efficiency in serverless machine learning training

Pablo Gimeno Sarroca, Marc Sánchez-Artigas *

Computer Science and Maths, Universitat Rovira i Virgili, Spain

ARTICLE INFO

Article history:
Received 28 October 2022
Received in revised form 11 July 2023
Accepted 30 August 2023
Available online 9 September 2023

Keywords:
Serverless computing
Function-as-a-Service
Machine learning

ABSTRACT

Function-as-a-Service (FaaS) has raised a growing interest in how to "tame" serverless computing 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 endeavor, we propose MLLess, a FaaS-based ML training prototype built atop IBM Cloud Functions. To boost cost-efficiency, MLLess implements two innovative optimizations tailored to the traits of serverless computing: on one hand, a significance filter, to make indirect communication more effective, and on the other hand, a scale-in auto-tuner, to reduce cost by benefiting from the FaaS sub-second billing model (often per 100 ms). Our results certify that MLLess can be 15X faster than serverful ML systems [27] at a lower cost for sparse ML models that exhibit fast convergence such as sparse logistic regression and matrix factorization. Furthermore, our results show that MLLess can easily scale out to increasingly large fleets of serverless workers.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

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 started to examine its potential in data-intensive applications [2,10,15,22,37–39,42]. Altogether, these works have led to a clear identification of "what" workloads are best suited to serverless computing.

Similarly, a recent trend on building machine learning (ML) on top of Function-as-a-Service (FaaS) platforms has emerged as a new research area [4,6,8,18,21,44]. Since ML inference is a trivial use case of FaaS computing [4,18], attention has turned into ML model training, which is a deal more difficult. Despite all the preceding efforts, it still remains uncertain under what conditions ML training 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 fundamental 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 [6,23].

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 [6,21,44], MLLess comes up with two novel 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

* Corresponding author.
E-mail addresses: pablo.gimeno@urv.cat (P. Gimeno Sarroca), marc.sanchez@urv.cat (M. Sánchez-Artigas).

https://doi.org/10.1016/j.jpdc.2023.104764
0743-7315/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

functions to directly share state. Thus, any model update (e.g., a gradient) must be exchanged through remote storage.

The second specialization is a scale-in auto-tuner to gradually decrease 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 shrunk during model training.

Equipped with this specialized training architecture, we next use MLLess to investigate 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. ML models that take hours to converge are presumably more cost-optimal to be trained on VM instances with today's offerings. We also examine the scalability of MLLess, and compare the effectiveness of our significance filter to loose synchronization models such as the Stale Synchronous Parallel (SSP) [16]. This comparison is very interesting, since while SSP restricts how stale, or "old", a model parameter can be, our significant filter bounds how inaccurate a parameter can be. This should shed light on what type of synchronization strategy is more appropriate for the indirect communication model of serverless computing is of vital importance.

Main insights. Our study yields four key insights:

1. *FaaS can be more cost-efficient than "serverful" libraries* such as PyTorch [27] for models that quickly converge, typically in a few hundreds of seconds with good accuracy. Although indirect communication severely penalizes FaaS-based ML training, MLLess ameliorates its impact with the aid of its two main optimizations, being 15X faster yet 6.3X cheaper than PyTorch running on CPUs. It must be noticed that for dense models, the benefits of MLLess are narrower, which suggests that FaaS-based model training excels in *sparse* models.
2. When running PyTorch on a GPU, the computational efficiency of PyTorch increases compared with FaaS-based training. Nevertheless, the performance/$ (here performance is the inverse of execution time) becomes similar in both cases, which hints that *FaaS can still be attractive for users that wish to optimize on cost.*
3. *Specializing distributed ML training to FaaS is crucial to deliver a higher cost-efficiency.* This requires dealing with low-level issues such as high gradient sparsity, filtering out non-significant updates, or dynamically scaling down the pool of workers, i.e., abilities that are not always available in VM-based ML systems such as PyTorch.
4. *Filtering non-significant updates is better than bounded staleness for FaaS-based model training.* While SSP has proven to be very effective for distributed "serverful" ML training, it renders only a marginal benefit for model training over FaaS. Since the communication of updates is the major bottleneck in FaaS, the flexibility to delay the propagation of an update until it eventually becomes significant is more effective than tolerating some amount of staleness.

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

Roadmap. A preliminary version of this work has appeared at Middleware'21 [40]. The rest of the article is structured as follows: §2 discusses the challenges of FaaS for ML training. §3 presents the MLLess' 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 [6,21,44] 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).

Limitations. 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, which must be downloaded remotely from shared storage in mini-batches, while inhibiting the use of any ML library that has not been designed with these constraints in mind. For instance, the authors of Cirrus [6], a serverless ML system, found impossible to run Tensorflow [1] or Spark [45] on AWS lambdas with such resource-constrained setups.

However, 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* [11]. For this reason, a careful optimization of communication, ranging from serialization of high sparsity models to the development of communication-reduction techniques, such as our significance filter, is crucial.

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 illustrated in Fig. 1. MLLess consists of a *driver* that runs on the local machine of the data scientist. For the implementation of the driver, MLLess extended PyWren [37], a serverless data analytics framework. When the user

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

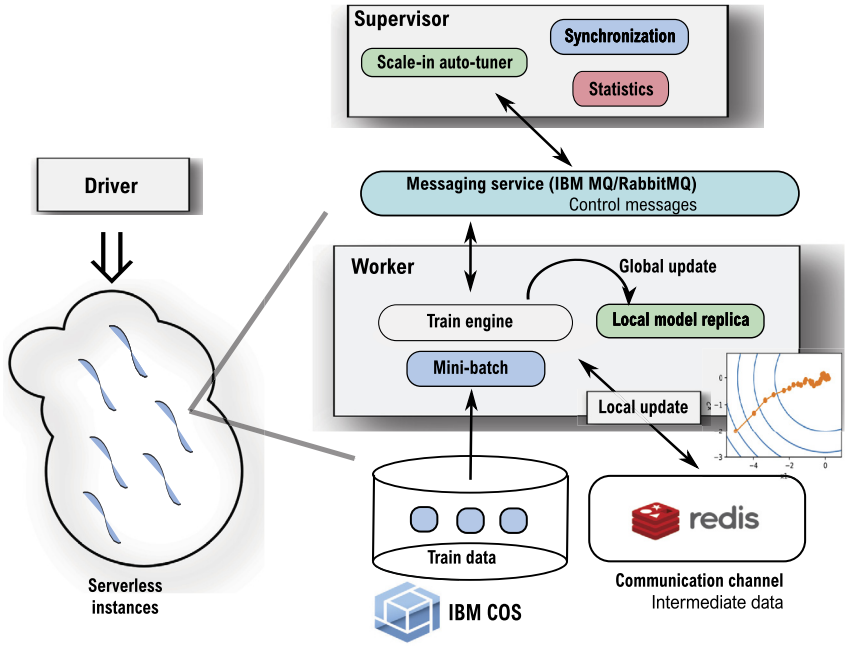


Fig. 1. MLLess system architecture.

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 since it better abides by to a pure FaaS architecture compared to the *VM-based parameter server* [20] model, e.g., followed by other works such as Cirrus [6].

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 criterion is fulfilled, among other tasks. Nevertheless, 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).

Since the supervisor is a serverless function, it is subjected to the time constraints of the underlying FaaS platform, in this case, to a maximum execution time of 600 seconds (IBM Cloud Functions). Although the supervisor never ran out of time in 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.

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 supports different consistency models: the Bulk Synchronous Parallel (BSP) model where the workers must wait for each other at the end of every iteration, and the Stale Synchronous Parallel (SSP) [16], a synchronization model that relaxes consistency by permitting workers to read stale parameter values as long as they are not too "stale". SSP was proposed to overcome the *straggler problem* suffered by BSP, where each iteration proceeds at the pace of the slowest worker. For this reason, SSP defines an explicit "slack" parameter for coordinating progress among the workers. The slack specifies how many iterations out-of-date the local replica of a worker can be, which implicitly dictates how far ahead of the slowest worker any worker is allowed to progress. For instance, with a slack of s, a worker at iteration t is guaranteed to see all updates from iterations 1 to $t - s - 1$, and it may see (not guaranteed) the updates from iterations $t - s$ to $t - 1$. We set BSP as the default synchronization model because it simplifies the reasoning about the impact of our optimizations on model convergence.

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). Compared with SSP, our variant restricts how inaccurate the aggregated update for a model parameter can be, in comparison to its current value, instead of bounding staleness in terms of the iteration count.

Communication channels. Due to the absence of direct communication between the workers or with the supervisor, MLLess establishes two channels of *indirect communication:*

- **Signaling channel.** For exchanging control messages between the workers and the supervisor (e.g., to signal a worker to advance to the next iteration), it leverages a messaging service built on RabbitMQ,[2] though it could be replaced by the native IBM's MQ messaging service[3] without complications.
- **Intermediate state.** For sharing the intermediate state generated during model training (e.g., local gradients), MLLess employs Redis,[4] a low-latency, in-memory key-value store that supports thousands of requests/s [34].

To store the input dataset mini-batches, MLLess uses IBM COS, the serverless object storage service from IBM Cloud. Since it is an "always on" service, it does not incur any startup delay, though

[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.

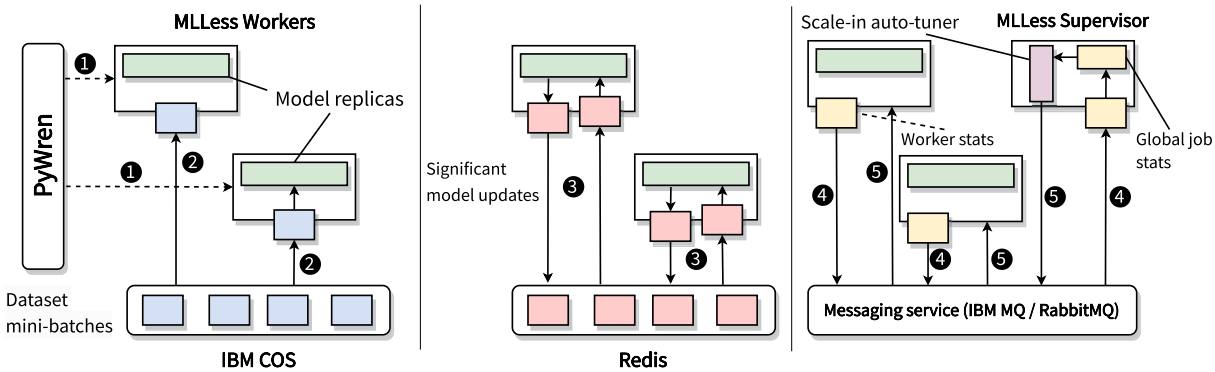


Fig. 2. Typical execution of a training job with MLLess.

it is a little bit slower compared to Redis, but sufficient for our purposes.

3.2. Model training

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

In today's systems, one typical optimizer is Stochastic Gradient Descent (SGD) [35], an iterative training algorithm that adjusts $\mathbf{x}$ based on a few samples at a time:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f_{\mathcal{B}_t}(\mathbf{x}_t),$$

where η_t is the learning rate at step t of the algorithm, $\mathcal{B}_t$ is a mini-batch of B training samples, and $\nabla f_{\mathcal{B}_t}(\mathbf{x}_t)$ is the gradient of the loss function, averaged over the batch samples: $\nabla f_{\mathcal{B}_t}(\mathbf{x}_t) = \frac{1}{B} \sum_{(\mathbf{v},l)\in\mathcal{B}_t} \nabla f(\mathbf{v}, l, \mathbf{x}_t)$. MLLess supports different optimizers (see Table 1 for further details).

Since 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 partitioned into mini-batches of size B. To generate the mini-batches in the appropriate format (e.g., feature normalization), MLLess leverages PyWren-IBM [37], 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. In Fig. 2, we portray a typical execution of a training job with MLLess. It involves the following five steps: ❶ The driver launches the initial number of workers. 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 targeted loss function f; ❷ In every iteration, each worker separately fetches a data mini-batch from IBM COS, and then it calculates a local update from its model replica before synchronization takes place. The specific type of local update depends on the underlying ML algorithm. In the case of the Stochastic Gradient Descent (SGD) [35] algorithm, local gradients are averaged to produce a global gradient update; ❸ Due to the lack of direct communication, each worker independently of the others pulls all the local updates from external storage (a Redis server), and aggregates them to update its local model copy. The availability of a local update is announced to the rest of workers through the signaling channel. To reduce communication costs, the workers can share only *significant updates*, i.e., updates whose magnitude is sufficiently high to change the associated parameters. Other synchronization protocols such as BSP or SSP can be used in lieu of the significance filter; ❹ After synchronization, each worker transmits its local training statistics to the supervisor. The supervisor aggregates them to monitor the progress of the training job, and to terminate it when the global stopping criterion is fulfilled, among other responsibilities; ❺ Based upon these statistics, the scale-in scheduler embedded in the supervisor can finally decide to remove a worker to save monetary costs. The removal of a worker is notified through the signaling channel.

We also note that the removal of a worker can be materialized at any time with no side effects on synchronization. Upon receiving the notification from the supervisor that a given worker is about to abandon the training of the model, the only thing that each active worker will need to do is to apply the unseen updates to its local replica of the model as usual. From that point onwards, no active worker will fetch any more updates from the leaving one, thereby gracefully handling its departure without a complex global coordination.

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 [21]. Indeed, to scale to large input data sizes or number of workers, it suffices to add more Redis instances and shard the local, ephemeral updates from the workers across them, so that the load is evenly distributed among all Redis shards. Further, the separation of control and data flows makes it easy to support different consistency models, from strict models such as BSP to relaxed ones such as SSP, with little or no changes in the iterative optimizers.

Weak (down)scaling. MLLess follows a weak scaling approach. Concretely, MLLess' parallelism strategy maintains the mini-batch size B fixed per iteration to keep the per-worker workload constant. The reason for this is to avoid that every change in the number of workers incurs costly data re-partitioning transfers to adjust the mini-batch size after a worker downscaling (see §4.2). The downside of this approach is that the global batch size B_g will decrease linearly with the number of workers P as they are removed by the scale-in auto-tuner (see §4.2 for details), which could affect the convergence speed of the optimizer [31]. Simply put, let $P' << P$, then the new global batch size $B'_g = P' \cdot B << P \cdot B = B_g$. For a small number of workers, this approach could lead to a very small effective global batch size B'_g, thus hurting convergence. To prevent significant deviation, our 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` [11], 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, as expected, 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 [17], we name it *'Insignificance-bounded Synchronous Parallel' (ISP)* to distinguish it from the original consistency model. In short, 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. In this sense, 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, or cluster*. Since ISP operates at the cluster level, its implementation is much simpler than ASP. It does not need complex synchronization mechanisms between data centers such as the ASP selective barrier and mirror [17], which facilitates its adoption in current serverless architectures. Further, ASP was originally implemented using the parameter server model [20], and not for fully decentralized training systems such as MLLess.

It must be noted that ISP degenerates to BSP when the significance threshold is set to 0, that is, when no local updates are filtered out. For this reason, one can safely claim that ISP is a variant of BSP. This is formally proven in Corollary 1 in Appendix A. To sum up, ISP is a variant of ASP tailored to the serverless model. And in the particular case that the significance threshold is set to zero, it reduces to the classical BSP model.

Overview. In a nutshell, ISP can be viewed as a technique to reduce the per-step communication complexity while preserving the convergence rate, which results in an improvement in system throughput, so as job training times. More concretely, its goal is to reduce the size of the local update to be transferred to the rest of workers, after the local worker goes through its mini-batch. This is possible 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. Typically, the strictness is controlled by a threshold v, which is reduced over time. That is, if the initial threshold is v, then the threshold value v_t at step t of the ML algorithm is given by $v_t = \frac{v}{\sqrt{t}}$.

Very importantly, ISP is synchronous in nature. That is, all workers must finish the current step before proceeding to the next iteration. Therefore, ISP differs from bounded asynchronous consistency models such as SSP [16], which sets a fixed upper-bound on the iteration gap between the fastest worker and the slowest one. The focus of ISP is thus on reducing communication requirements rather than alleviating system heterogeneity as SSP realizes. Observe that a smaller communication complexity also means a lower computation complexity, since less model parameters must be updated per iteration.

It is worth to note here that the original definition of the ASP model [17] 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 ith 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., $m_t := \left(t - t_{p_i}\right)$. 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 ith 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 ith 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 $\lim_{T\to\infty} \frac{R[X]}{T} = 0$.

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

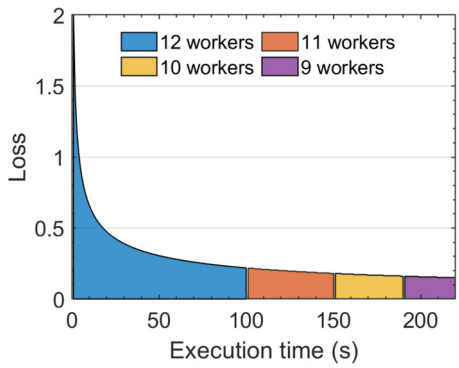


Fig. 3. Sample execution of the scale-in scheduler. In the low convergence zone, workers are progressively removed from the pool.

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 such as in the case of reserved VMs [32,46]. 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 [46]. For instance, SGD reduces loss approximately as a geometric series on convex problems [5]. This implies that, while a higher number of workers is desirable during the first training steps to steeply diminish loss, a large worker pool gives only marginal returns when loss reduction slows down, which ends up worsening the cost-efficiency ratio. In this sense, the primary 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. Fig. 3 illustrates an example of how workers are gradually removed from the system. When loss reduction begins to stagnate after around 90 seconds, the scale-in scheduler starts to progressively withdraw workers from the model training process, which saves cost.

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 [31], 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. 4a, 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:

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

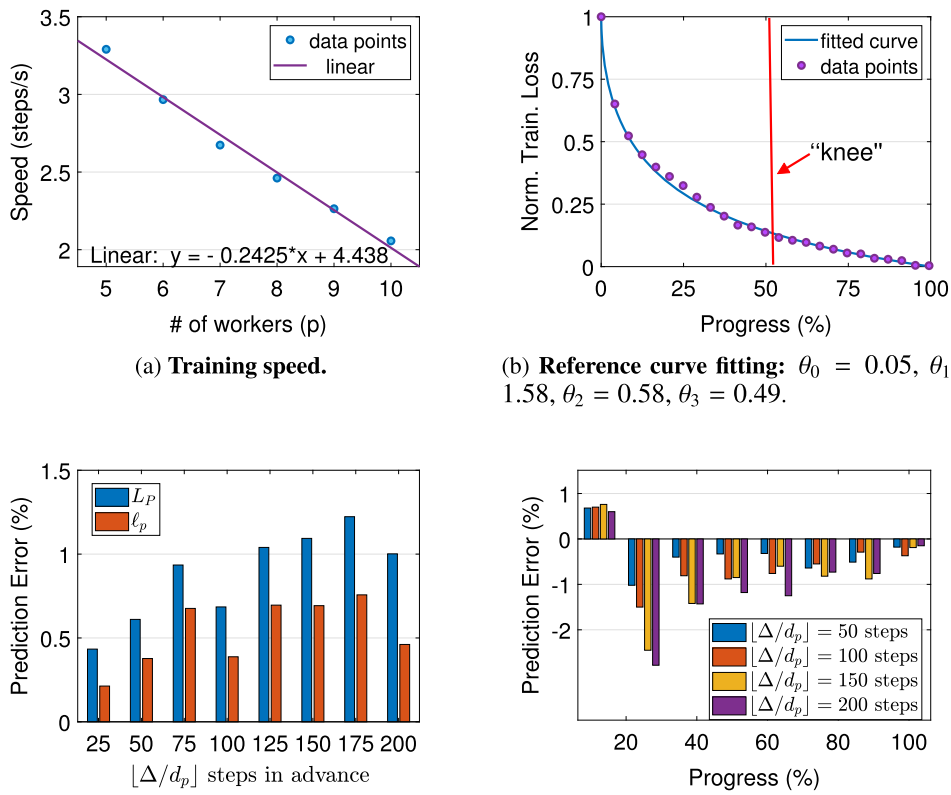
where:

$$\begin{aligned} t &:= \text{current training step,} \\ \left\lfloor \frac{\Delta}{d_p} \right\rfloor &:= \text{no. of steps to be completed in } \Delta \text{ time units with } p \text{ workers,} \\ L_P\left(t + \left\lfloor \frac{\Delta}{d_P} \right\rfloor\right) &:= \text{expected loss with all } P \text{ workers,} \\ \ell_p\left(t + \left\lfloor \frac{\Delta}{d_p} \right\rfloor\right) &:= \text{expected loss with the } p \text{ workers,} \end{aligned}$$

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. Since a new scheduling decision can be made after time T, the idea is to choose $\Delta \leq T$ 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 defined in Eq. (1), the scheduler performs online fitting on two types of learning curves, namely, 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



(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.

Fig. 4. Training speed and prediction error of training a matrix factorization (PMF) model [36] on MovieLens-1M data [14].

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 $O(1/\sqrt{Bt}+1/t)$ [26], 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 θ_0, θ_1, θ_2 and θ_3 are non-negative coefficients. An example of online curve fitting when training a PMF model is depicted in Fig. 4b. 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 [46], where θ_0, θ_1, θ_2 and θ_3 are also non-negative. We utilize a non-negative least squares solver [7] 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.

Retaking the MF training example, Fig. 4c gives 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. 4c, 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. Finally, Fig. 4d 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., [9] and Kneedle [41]), 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 model replica from the pool. If the significance filter is enabled, i.e., $v_t > 0$, 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 [37] — 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 [6]. So, to make MLLess competitive with the "serverful" ML libraries, we

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

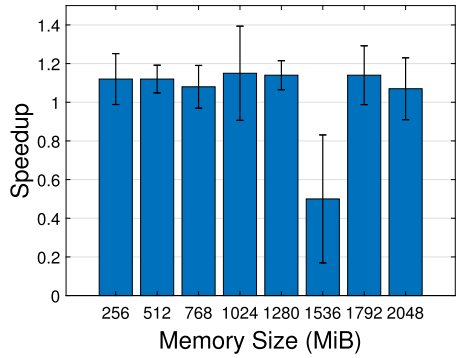


Fig. 5. Speedup of two threads relative to single-thread performance within a function as memory size is varied.

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 degraded system throughput to a large extent.

Intra-worker parallelism. 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) [36] 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. 5. 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 perform a series of experiments to answer the following main questions:

- *What is the individual contribution of each optimization to cost-efficiency?* For we perform a number of micro-benchmarks.
- *Is it possible to achieve better cost-efficiency with an optimized FaaS platform than a VM-based, i.e., "serverful" substrate (IaaS) for distributed ML training?* For we run several ML training jobs of different flavors, including both dense and sparse ML models. We use PyTorch [27], a specialized "serverful" ML library, but also a non-specialized, serverless data-analytics system, to determine what happens when FaaS is not specialized to model training.
- *Is the ISP consistency model much more effective than other bounded staleness models such as SSP?* The goal is to infer what type of synchronization strategy is more appropriate for the indirect communication model of FaaS cloud platforms.

To conclude, we also evaluate the scalability of MLLess on the exchange of intermediate training state, which is the main system bottleneck due to the impossibility of function-to-function communication. Note that the scalability of object storage for the storage of training datasets has already been assessed in other works [6]. Consistent with these works, we have observed no bottleneck for the download of mini-batches from IBM COS.

6.1. Methodology

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

- **Distributed PyTorch [27] on CPUs**. Due to the lack of hardware accelerators such as GPUs and TPUs [24] 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 [12]—rule of thumb for CPU training—, a MPI-like library for cross-machine communication. Mini-batches are downloaded from IBM COS.
- **PyWren-IBM [37]**. We leverage PyWren-IBM as non-specialized serverless ML representative. PyWren-IBM has been 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 performed through IBM COS, including the sharing of updates, to keep its pure serverless, general-purpose architecture.
- **PyTorch [27] on GPU**. Although functions do not support hardware acceleration, we find interesting to compare MLLess against Pytorch v1.8.1 running in the cheapest virtual machine equipped with a single GPU card. The GPU card is an NVIDIA P100 with a double-precision and single-precision performance of 4.7 teraFLOPS and 9.3 teraFLOPS, respectively. This is significantly higher than the 67.57 GFLOPs for double-precision and the 115.72 GFLOPs for single-precision of an IBM Cloud Function,[8] which gives PyTorch a great advantage over MLLess. The specs of the VM are given in Table 2.

Datasets. We utilize three datasets in our evaluation. First, we use the **Criteo** display ads dataset [25], which contains 47M samples and has 11GB of size in total. Each sample consists of 13 numerical and 26 categorical features. Before training, we normalize the dataset. In particular, we manipulate this dataset in two forms. On one hand, we only use the 13 numerical features to produce a *dense* dataset. On the other hand, we hash all the categorical dimensions to a sparse vector of size 10^5 ("hashing trick"), along with the 13 numerical features, to produce a *sparse* dataset. In this way, we can evaluate the impact of sparsity on the cost-efficiency of FaaS over IaaS as another evaluation dimension.

Also, we use the **MovieLens-**10**M** and **MovieLens-**20**M** datasets [14]. The former consists of 10M movie reviews from $N_u = 10,681$ users on $N_m = 71,567$ movies. The latter bears around 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**) [36]. Concretely, for **PMF**, we factorize the partially filled matrix of review ratings $\mathbf{R}$ 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 $\mathbf{R} \approx \mathbf{UM}$.

Loss functions. For training the **LR** models, we will leverage the **Binary Cross Entropy (BCE)** as the loss function, which is the most

[6] http://cython.org/.
[7] https://www.nsc.liu.se/software/math-libraries/.
[8] We ran the same benchmark as NVIDIA to compute the GFlops: the GEneral Matrix Multiply (GEMM) within the Basic Linear Algebra Subroutine (BLAS) library.

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

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

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)	Four PyTorch workers	0.2 $/hour
GPU ACL1.8x60 (8vCPUs, 60GB RAM, 1 x Nvidia P100 GPU)	PyTorch on GPU	1.948 $/hour

common loss function used for binary classification problems. Likewise, we will make use of the **Root Mean Squared Error (RMSE)** as the loss function for training the **PMF** models.

Setup. The VM instances used for the experiments are deployed on the IBM Cloud. Unless otherwise noted, when running MLLess, we use two 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. The way to account for cost is vital to measure cost-efficiency, so we included all costs incurred by MLLess. That is, MLLess's cost comprised the individual cost of each component, namely the serverless workers plus the two VM instances: one to host the signaling service (C1.4x4 instance), and the other to exchange the intermediate training state (M1.2x16). 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, whereas the price of functions per time unit is proportionally much higher than VM instances. In practice, PyTorch would cost more. To verify this, Table 2 reports the exact pricing of each component. As shown in this table, a serverless worker has the same amount of provisioned resources as a PyTorch worker: 1vCPU, 2GB RAM. The only difference is that while serverless workers are provisioned individually, Pytorch workers are provisioned in groups of four due to their deployment on VM instances. Consequently, a PyTorch worker costs $\frac{0.2\$/\text{hour}}{4} = 0.05\$/\text{hour}$, which is more than two times cheaper than a serverless worker: 0.122 $/hour.

We further observe that the use of VMs confers some extra advantage to PyTorch, as the exchange of intermediate training state across all processes (`AllReduce`) can leverage the fact that some PyTorch workers are physically located in the same machine.

We finally note that we do not include the storage costs incurred by IBM COS in the cost computation. The reason is that all systems use COS to store the mini-batches of the different datasets, and thereby have the exact same storage costs. We also recall that there is no charge for data retrieval within the same region, so mini-batch fetches are free of charge in all competing systems.

Table 3
COST sanity check. Available instance types with a 2:1 memory-core ratio in IBM Cloud. All instances have 1 Gbps NIC. (`us-east`, June 2023).

Instance type	vCPUs	RAM (GiB)	Price
B1.4x8	4	8	0.178 $/hour
B1.8x16	8	16	0.348 $/hour
B1.32x64	32	64	1.379 $/hour

Table 4
Execution time, cost and Perf/$ of PMF on ML-10M using 24 workers (RMSE=0.86).

Configuration	Exec. time (s)	Cost ($)	Perf/$ (Eq. (4))
PyTorch (6 VMs of type B1.4x8)	579.1	0.171	0.010
PyTorch (3 VMs of type B1.8x16)	285.4	0.091	0.034
PyTorch (1 VM of type B1.32x64)	320.5	0.110	0.032
MLLess	130.8	0.117	0.065

Model sanity check. Before conducting any experiment, we first realized a sanity check to ensure 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.

COST sanity check. To make sure that the choice of the VM instances for PyTorch still does not benefit MLLess as of June 2023, we performed an additional sanity check. Throughout our evaluation, one of our major premises was to keep a 2GB memory-per-core ratio in PyTorch to provision equivalent resources as those available in the IBM Cloud Functions,[9] and thus allow a fair comparison. As reported in Table 3, there exist only two possibilities to preserve this ratio in the IBM Cloud, namely, instance types B1.4x8 and B1.8x16. In particular, we chose the former instance type for our experiments, because it is the one that enables cluster setups that most closely resemble the fully distributed training model behind MLLess, yet allowing Pytorch to benefit from intra-worker communication.

Nevertheless, one can certainly argue that with larger VM instances, the improvements achieved by MLLess in the following sections may not hold true. To disprove this intuition, we verified as in COST [29] the behavior of PyTorch when running in a single machine. For this, we needed to use an overprovisioned VM instance of type B1.32x64 (32 vCPUs, 64GB RAM). Taking MovieLens-10M as a representative example, we ran a PMF training job on 24 workers with a target RMSE threshold of 0.86, using all the VM configurations listed in Table 3.

The results are shown in Table 4. As per the results in this table, we can confirm that MLLess is always faster and more cost-efficient than IaaS Pytorch, even in the single-machine configuration. To wit, MLLess is up to 2.19X faster than Pytorch in this configuration, while providing a 2.03X higher performance per dollar (Perf/$). We observe that the Perf/$ metric is very useful to

[9] For the maximum memory allocation of 2GB, the IBM Cloud Functions service delivers the equivalent of one vCPU.

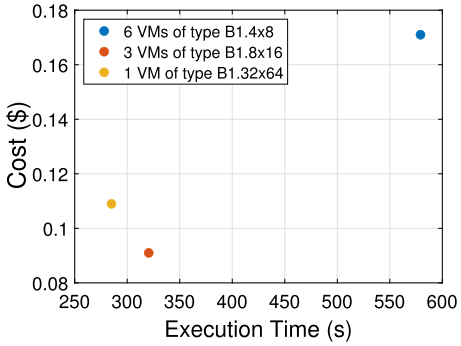


Fig. 6. Cost-performance comparison of PyTorch using different IaaS setups for PMF on MovieLens-10M dataset.

capture the cost-efficiency in a single metric (see Eq. (4) for further details).

Additionally, Fig. 6 illustrates that choosing the fastest VM configuration is not always desirable when the cost is part of the equation. For this specific job, this figure outlines that PyTorch is slightly slower in three instances than in the single-machine setup. However, this figure also reveals that the single-machine setup yields a lower Perf/$. This attests that choosing the IaaS configuration that optimizes the cost-performance trade-off is far from trivial. In general, multiple IaaS-based configurations are available. And selecting the best will depend on the model to be trained, the dataset size and the number of workers. This choice requires an individualized study for every training job, with the associated cost of redeploying the cluster and rerunning the training job with PyTorch.

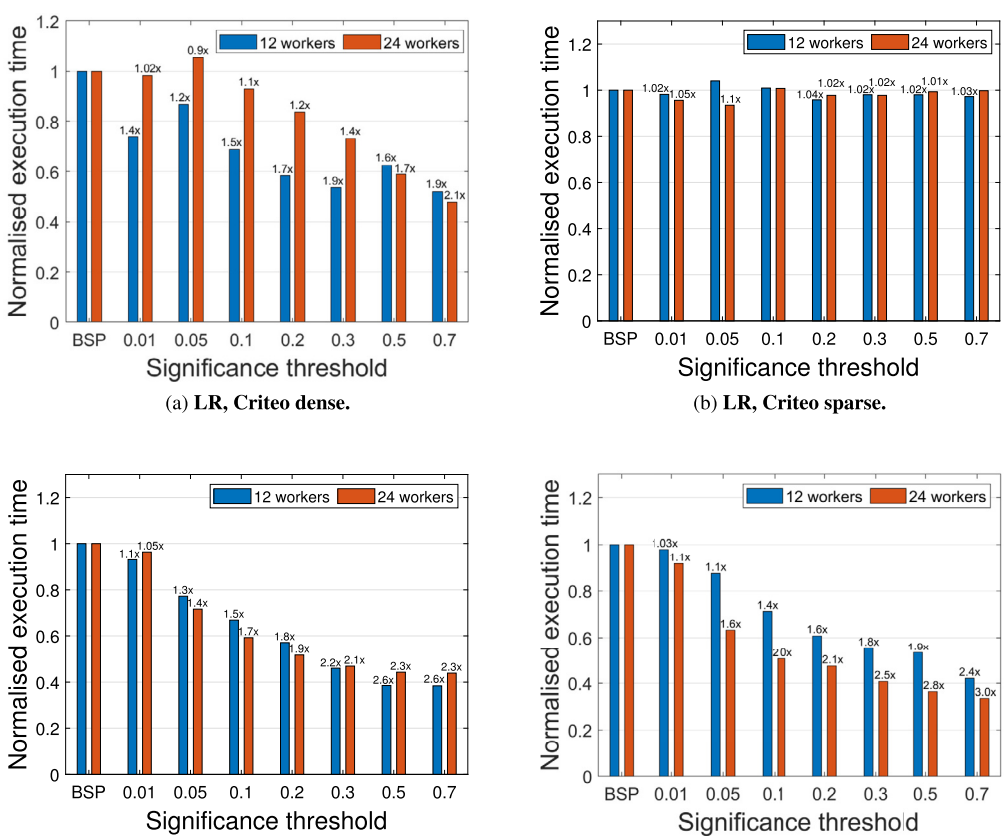
6.2. Micro-benchmarks

To better understand the individual contribution of each optimization to cost-efficiency, we run a number of micro-benchmarks.

6.2.1. Significance filter

We first evaluate the effectiveness of ISP to improve system throughput as the significance threshold v increases, i.e., it becomes more strict, thereby filtering out more aggressively those updates than produce small changes to the model. 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. Because of the "pay-as-you-go" model of cloud functions, the key point to note here is that by *decreasing the execution time, ISP cuts the cost forthwith*. We use the BSP synchronization model.

The results are plotted in Fig. 7. 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, FaaS-based ML training can be importantly enhanced despite the impossibility of function-to-function communication. The results for LR reinforce this idea and give further sense of the potential improvements brought by ISP. Non-surprisingly, ISP has a stronger effect on communication for Criteo dense compared to sparse logistic regression. Actually, the difference in execution time of about 2X between sparse and dense LR mostly lies in model sparsity. More precisely, sparse LR produces highly sparse gradients per se due to the "hashing trick". On one hand, MLLESS filters zeroed features, which acts as an intrinsic filter in communication and reduces the size of updates. On the



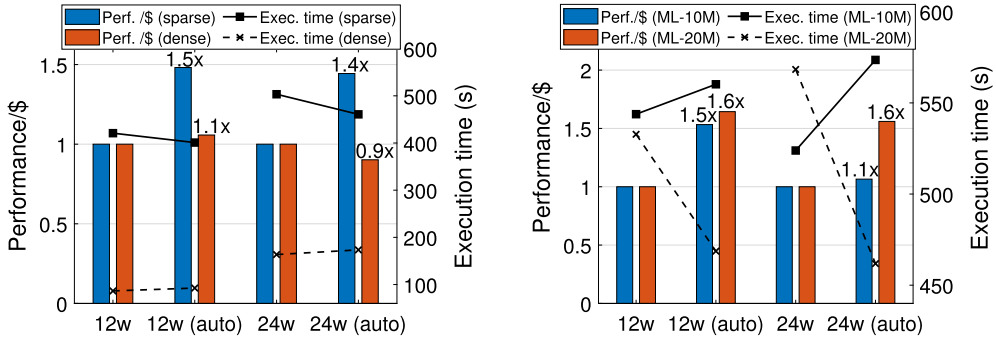
(a) **LR, Criteo dense.**

(b) **LR, Criteo sparse.**

(c) **PMF, ML-10M.**

(d) **PMF, ML-20M.**

Fig. 7. Normalized execution time until convergence as the significance threshold v increases.



(a) **LR**, **Criteo** (BCE=0.58)

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

Fig. 8. Effect of the scale-in auto-tuner. Two metrics are used: Perf/$ (bars; left axis); execution time (lines; right axis).

other hand, the "hashing trick"' results in dissimilar gradient updates, which lend themselves to little compression. Consequently, the small gains in communication end up being more significant in dense LR, despite the smaller model size of 13 numerical features.

6.2.2. 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:

$$\text{Perf}/\$:= \frac{1}{\text{Exec.time(s)}} \times \frac{1}{\text{Price}(\$)}, \tag{4}$$

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 execute 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. 8 itself. For the scale-in auto-tuner, we set the scheduling interval to 20 s and fix the parameter Δ at a half of the scheduling epoch, that is, $\Delta = 10$ sec.

Results are illustrated in Fig. 8. For sparse 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 dense LR, the auto-tuner in isolation is only capable of slightly increasing the Perf/$ for 12 workers, leading to an improvement of 1.1X over the baseline. For $P = 24$ workers, the Perf/$ worsened a little bit due to a ∼9% underestimation of the original convergence rate caused by an imprecise fitting of the reference curve $L_P(t)$. We leave for future work the development of a more precise estimation method for the reference curve to prevent any degradation of Perf/$.

Interestingly, the fact that the execution time increases with more workers for the LR use case is attributable to a loss of *statistical efficiency* [28] due to weak scaling, rather than to a poor scalability of MLLess. To corroborate this claim, we repeated the same experiment, but now adjusting the mini-batch size B as we varied the number of workers to keep the global batch size B_g the same at all times. We got comparable results, as listed in Table 5, which shows that the converge rate was equivalent in all worker configurations for a constant B_g. By adapting the mini-batch size B, model replicas synchronized more frequently as the number of workers grew, thus preserving statistical efficiency.

For PMF, the results were also nice. For all settings, the auto-tuner improved the Perf/$. For the ML-20M dataset, it even led to 1.6X gain since it also delivered a significant improvement in speed. The small degradation of around 7.1% in execution time for the ML-10M dataset was 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).

Table 5
Execution time of **LR, Criteo sparse** (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.*

6.3. Cost-efficiency

In this section, we explore the cost-efficiency of MLLess, while seeking to answer the nagging question of whether FaaS can outperform a VM-based, IaaS infrastructure for distributed ML learning.

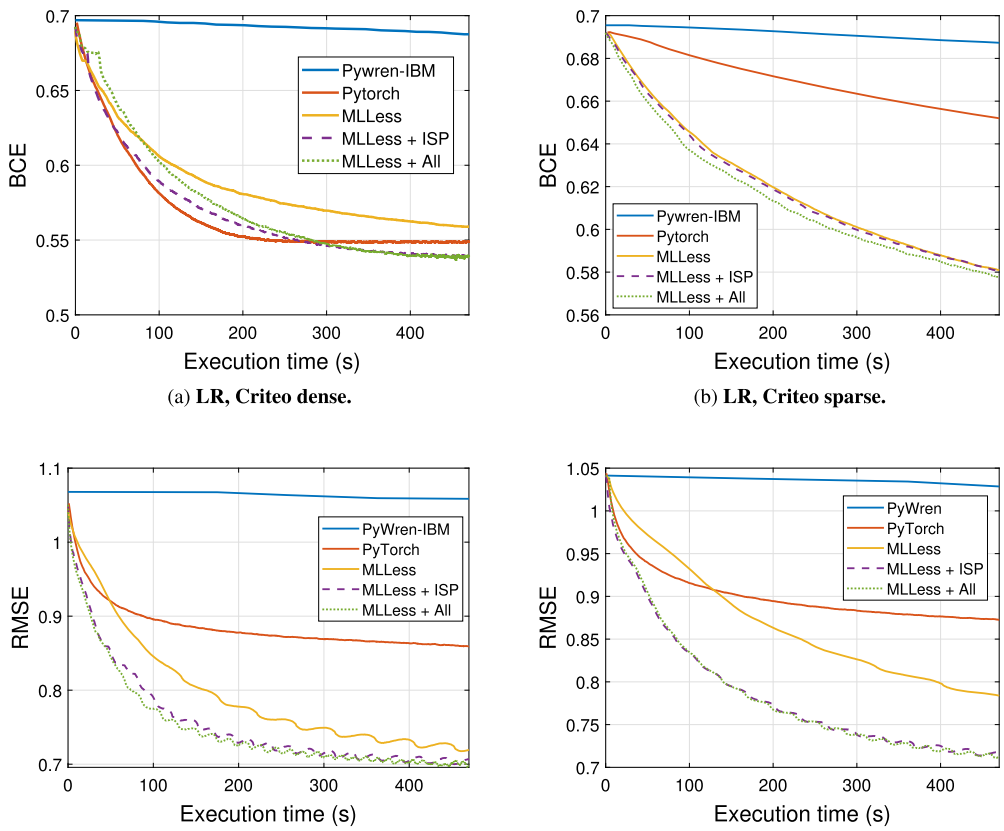
6.3.1. Performance comparison on CPUs only

To assess the benefits of a specialized system for serverless ML training, we compare MLLess against distributed PyTorch [27] running on CPUs and PyWren-IBM [37]. 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 20 s with $\Delta = 10$ s. For all the systems, we only report the results for $P = 24$ workers. The trends were similar for 12 workers.

Results. The results are shown in Fig. 9. 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-**10**M** 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



(a) **LR, Criteo dense.** (b) **LR, Criteo sparse.** (c) **PMF, ML-10M.** (d) **PMF, ML-20M.**

Fig. 9. 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.

and to converge to a "prudent" RMSE loss of 0.738, PyTorch spends 2,029 seconds. MLLess, however, reaches this loss value after 140 seconds. This yields a speedup of 14.49X.

For the **PMF+ML-20M** application, we get similar results. To converge to a loss of 0.821, PyTorch spends 1,800 seconds. MLLess achieves 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 [19]. 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.

The **LR+Criteo dense** job produced another interesting result, which further buttresses the idea that optimizations tailored to the FaaS environment are crucial to be competitive against IaaS-based ML training. Unlike in all the other experiments, Pytorch is able to outperform plain MLLess in this case. However, when the MLLess optimizations are enabled, MLLess overtakes Pytorch in the middle of the execution and is able to converge to a lower BCE level.

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 decrease 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.

Main insight. As a key conclusion, we find that *FaaS can be more performant than IaaS* under the same conditions (i.e., number of workers and memory per worker, running on CPUs only) for, at least, fast-convergence models if ML training is specialized to FaaS architectures.

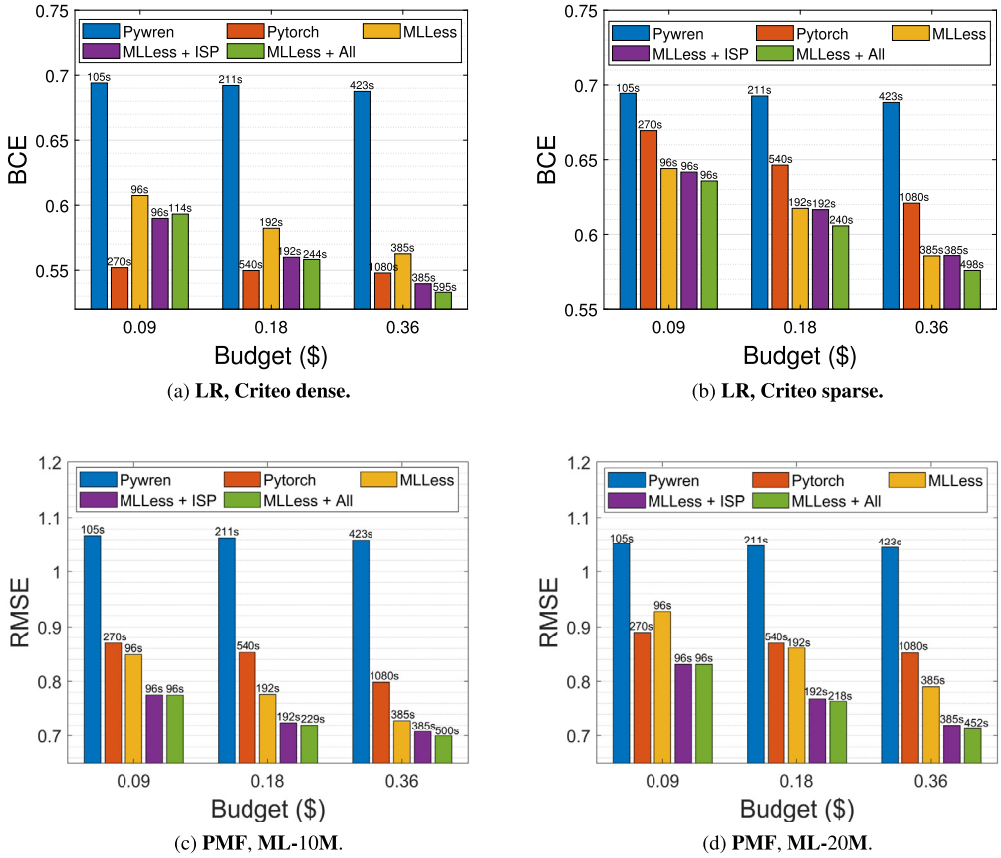
6.3.2. *Cost comparison of on CPUs only*

As shown above in §6.3.1, FaaS can outperform IaaS-based training. However, the price/time unit of a serverless worker is typically higher than IaaS-based worker. As given in Table 2, a PyTorch CPU-only worker costs $\frac{0.2\$/\text{hour}}{4} = 0.05\$/\text{hour}$, which is more than 2X cheaper than a serverless worker: 0.122 \$/hour. Therefore, a better cost-efficiency for FaaS-based ML training is a priori more difficult, but plausible, mostly because of the possibility to dynamically adjust the number of workers, among other abilities.

Following the same path traced above, here we compare MLLess against distributed PyTorch running on CPUs only, 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.

General results. As a headline observation, MLLess is cheaper than 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 leads to a 6.32X savings on cost. Likewise, 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.

Fixed-budget cost results. While MLLess helps to save 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. Therefore, it is interesting to examine what would be the performance of MLLess for a fixed budget. To answer this question,



(a) **LR, Criteo dense.** (b) **LR, Criteo sparse.** (c) **PMF, ML-10M.** (d) **PMF, ML-20M.**

Fig. 10. 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.

Fig. 10 illustrates 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 execution time affordable by MLLess. Per contra, MLLess is significantly more efficient per time unit and better adjusts to the cost plan. We note that 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.

The minimal exception to the above rule of thumb is for **LR+Criteo dense** (Fig. 10a). For this task, Pytorch is able to deliver the best performance for the 9¢- and 18¢-budgets, mostly because of its optimal, ring-based `All-reduce` primitive for dense data. Fortunately, by leveraging the combined effect of the scale-in auto-tuner and ISP, MLLess manages to incrementally improve the cost-efficiency ratio, achieving a lower BCE value for 36¢.

Main insight. As a preeminent takeaway, *FaaS-based model training can be more cost-efficient than IaaS CPU-based model training* for fast-convergent models if training is specialized to the traits of the serverless environment: indirect communication and fine-grained "pay-as-you-go" billing.

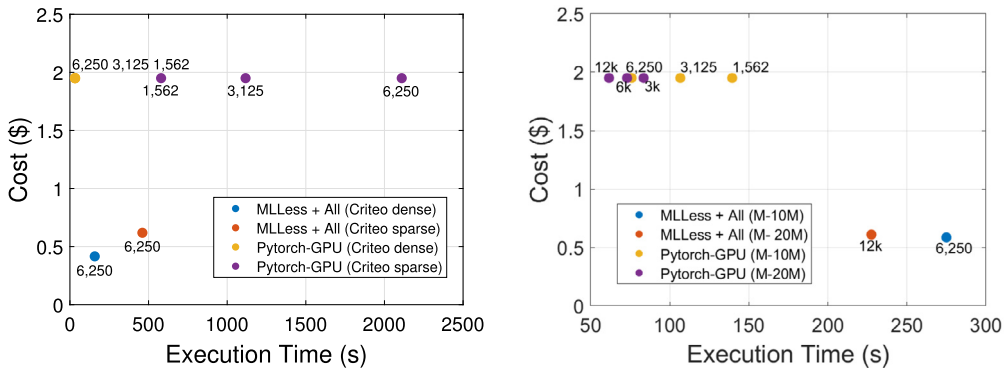
6.3.3. *Cost-efficiency comparison of* MLLess *against PyTorch on GPU*

As demonstrated in the preceding sections, FaaS-based model training can be more cost-efficient than IaaS CPU-only model training. Since GPUs are excellent at handling ML computations, we considered relevant to compare FaaS-based model training against GPU-based training. As discussed in §6.1, we chose the cheapest instance type equipped with a single GPU card for this experiment. Despite this, we made sure that the compute power of the GPU card was higher than the aggregate compute power of our setup of 24 serverless workers (see Table 6 for further details). Not only the PyTorch-GPU setup was more powerful, but it faced no penalties related to distributed training (e.g., due to gradient communication and synchronization). Needless to say, this gave PyTorch-CPU a greater advantage in terms of execution time. To compute the cost, we used hour prices as the billing granularity for two reasons. First, because billing is more exact. And second, because GPU instances are only billed per hour in IBM Cloud. In this experiment, we trained the same models as above, using the same convergence thresholds. We tried different mini-batch sizes for PyTorch-GPU to provide a better understanding of its performance.

Table 6
Compute power in teraFLOPS.

System	Double-precision	Single precision
NVIDIA P100 GPU	4.7 teraFlops	9.3 teraFlops
MLLess (24 workers)	1.6 teraFlops	2.7 teraFlops

Results. The results are depicted in Fig. 11. The main observation to be made is that if the user optimizes for execution time, IaaS-based training with GPUs is faster in general. However, FaaS-based training is cheaper in all cases. This is easy to visualize in the PMF application. While PyTorch-GPU is 3.6X faster than MLLess in both MovieLens datasets, FaaS-based training becomes between 3.2X to 3.3X cheaper than PyTorch-GPU. Interestingly, MLLess is faster than



(a) **LR, Criteo.** (b) **PMF, MovieLens.**

Fig. 11. Cost-performance comparison between PyTorch running on a GPU (**PyTorch-GPU**) and MLLess with all optimizations (**MLLess+ All**).

Table 7
Perf/$.

Job	Mini-batch size (B)	Perf/$ (10^{-3})	
		PyTorch-GPU	MLLess
LR, Criteo dense	6,250	16.11	15.22
	3,125	15.36	
	1,562	16.04	
LR, Criteo sparse	6,250	0.24	3.51
	3,125	0.46	
	1,562	0.89	
PMF, ML-10M	6,250	6.77	6.22
	3,125	4.81	
	1,562	3.68	
PMF, ML-200M	12,000	8.34	7.22
	6,000	7.04	
	3,000	6.15	

PyTorch-GPU for **LR+Criteo sparse** due to the large sparsity of the tensors. Concretely, MLLess is 4.5X faster than PyTorch-GPU in this training job.

To compare both systems using a single metric, we provide the Perf/$ as defined in Eq. (4) for each configuration in Table 7. Remember that the higher the Perf/$, the better the cost-efficiency of the system. In other words, this metric enables to capture into a single value whether the rate at which the execution time decreases is higher than the rate at which the cost grows. As shown in the table, the Perf/$ for PyTorch-GPU is slightly better than MLLess (less than 15% in the worst case), with the exception of LR on the Criteo sparse dataset, where MLLess achieves 3.9X higher Perf/$ than PyTorch-GPU.

Main insight. As a main takeaway, *FaaS-based model training can be much cheaper than IaaS GPU-based model training* for fast-convergent workloads. Non-surprisingly, *GPU-based training is faster.* When jointly taking cost and execution time, both FaaS-based and IaaS GPU-based training exhibit a similar Perf/$, making FaaS still appealing for those users who wish to optimize on cost.

6.4. SSP vs. ISP for FaaS-based ML training

Due to the need of indirect communication in FaaS-based ML training, another interesting question is to ascertain whether ISP is better suited for serverless model training than other popular yet loose consistency models such as SSP [16]. To this goal, we integrated SSP into MLLess, and compared it with ISP, as well as with the baseline BSP-based version. To carry out this comparison, we experimented with PMF on the **ML-20M** dataset for an increasing number of workers P. To not compromise statistical efficiency due to weak scaling (see §6.2.2 and Table 5 for further details), we

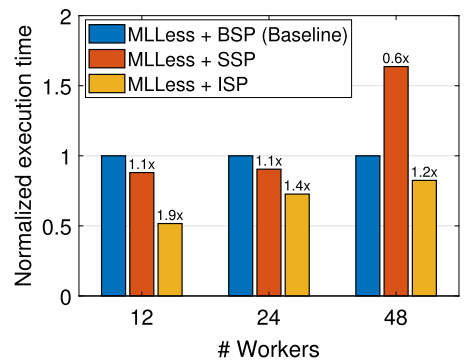
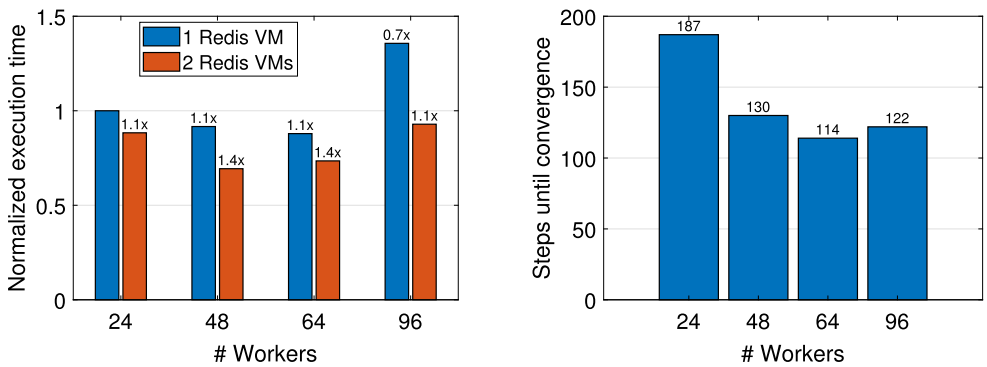


Fig. 12. Comparison of **SSP** against **ISP** synchronization model for serverless ML training.

fixed the global batch size B_g and adjusted the mini-batch size B accordingly. Concretely, we set $B = 12$ K for 12 workers, $B = 6$ K for 24 workers and $B = 3$ K for 48 workers. In this way, we made sure that the effect of stale updates came out neatly for each synchronization model. For SSP, we set a slack of $s = 3$ iterations.

Results. The results are depicted in Fig. 12. As expected, SSP shows a better average speedup of 1.1X over the default BSP implementation for 12 and 24 workers. For 48 workers, SSP, however, performs worse than the synchronous BSP model due to the lack of intra-function parallelism. More technically, SSP is agnostic to the computation capacity of workers, but merely ensures that the number of iterations between the fastest and the slowest workers does not exceed the *staleness bound* s. In a distributed setting such as that of MLLess, where each worker is responsible to aggregate the updates from the rest, i.e., there are no global parameters, the lack of intra-function parallelism means there is no way for the slowest workers to hide the latency of downloading and applying the missing updates. A solution to this problem would be to use a serverless backend such as Crucial [2] to perform the storage-side aggregation of gradients.

The more relevant finding of this experiment is, however, that ISP outperforms SSP in all cases, yielding a speedup of 1.9X and 1.4X for 12 and 24 workers, respectively. The reason why ISP is way better than SSP is that ISP permits any worker to delay the synchronization of a parameter *indefinitely* as long as its aggregated update is non-significant. Under SSP, however, update synchronization is delayed up to most s iterations for the fastest workers, but sooner or later, all the committed updates from the workers are added to the model replicas, thus not reducing the communication overhead at all. Put another way, the loose synchronization property of SSP is not enough to outweigh the reduction in network traffic achieved by ISP, being the latter more



(a) Effect of Redis instances.

(b) Statistical efficiency vs. # of workers.

Fig. 13. Scalability of MLLess on the **ML-20M** dataset.

effective to yield faster convergence for FaaS-based ML training due to the need of indirect communication.

Main insight. For FaaS-based model training, where the exchange of intermediate updates is the main limiting factor, *parameter staleness is not of practical utility unless it saves network traffic.*

6.5. Scalability

Finally, scalability is a critical property of any ML training system, and MLLess is not the exception. Cloud object storage as a means to hold mini-batches has proven to provide good scalability [6], and we empirically found that the signaling channel was able to support thousands of messages per second, enough to scale to hundreds of concurrent workers. Certainly, among all the MLLess components, we observed that the indirect communication channel built to exchange updates is the one subjected to a major strain. Fortunately, this channel can be easily "scaled out" by adding more Redis instances and "sharding" intermediate updates over the pool of servers using the worker IDs. To verify this claim, we ran multiple training jobs with the **ML-20M** dataset for an increasing number of workers. For each worker size, we trained the model with 1 and 2 Redis instances, stopping at a RMSE value of 0.77 in all settings. To preserve statistical efficiency, i.e., a similar convergence progress per second, we adjusted the batch size as in the prior test.

Results. The scalability results are illustrated in Fig. 13a. For ease of comparison, we normalized the execution times, choosing the configuration of 24 workers with 1 Redis server as the baseline. Non-surprisingly, doubling the number of Redis instances has almost no effect for a small number of workers. Nonetheless, its effect becomes more apparent as the number of workers increases and a single server cannot keep up with the high rate of updates. With two Redis servers, MLLess is able to deliver a speedup of 1.4X for 64 workers, despite a super-linear increase in the relation of the number of workers to the Redis servers with respect to the baseline setup, i.e., $\frac{64\ \#\text{workers}}{2\ \#\text{Redis servers}} / \frac{24\ \#\text{workers}}{1\ \#\text{Redis servers}} = 1.33$. Likewise for 96 workers, the execution time is a 10% better with 2 Redis servers, despite featuring 4X more number of workers than the baseline setup. This confirms that the addition of more servers enables MLLess to scale to a larger number of workers.

It is worth to mention here that as in IaaS-based ML training, the scaling of the training process in FaaS platforms is equally challenging. Simply put, users expect the training time to go down with the number of workers. However, even if the sharing of updates is not a bottleneck with the addition of more Redis instances, the relation between the mini-batch size and the number of workers may hinder *statistical efficiency*. This is reflected in Fig. 13b, where the number of training steps until reaching the threshold is shown. We use the number of training steps, instead of time, as a metric, because the total number of steps to convergence is independent of the size of the Redis cluster. It is a quality measure that depends on the global batch size and the number of workers, which captures very well how frequently the workers synchronize in relation to the processed training data.

As seen in this figure, the number of iterations to convergence decreases until 96 workers, the point beyond which adding more workers starts hurting convergence. At this point, adding more Redis servers can help reduce I/O time, so as the training time. But eventually, the increase in the number of iterations caused by the usage of more workers will limit scalability. For this reason, users try to compensate this reduction in statistical efficiency by either increasing the learning rate [13], or adjusting the batch size adaptively [43].

Main insight. We conclude that MLLess is scalable, and that can be easily "scaled out" through in-memory storage sharding. As in traditional VM-based systems, the ultimate scaling of the training process depends on the mini-batch size, the synchronization model, etc., irrespective of whether storage sharding can eliminate the bottlenecks.

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. For example, PyWren [22], IBM-PyWren [37] and Lithops [38,39] are map-reduce frameworks running over FaaS executors that take advantage of object storage to store intermediate data. Lithops [38,39] is multi-cloud and also implements the native `multiprocessing` module available in Python to enable the transparent execution of multiprocessing applications over FaaS platforms. Further, `gg` [10] is a library that uses AWS Lambda for CPU-bound intensive jobs such as video-encoding. Numpywren [42] is an elastic linear algebra library on top of a pure serverless architecture. Starling [33] 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.

Another important work is Crucial [2,3]. Crucial is a framework for building stateful FaaS-based multi-threaded applications, and as such, it includes fine-grained synchronization primitives such as semaphores and barriers. As part of its evaluation, Crucial was compared to Spark using two classical ML algorithms: K-means and logistic regression, showing an on-par performance with Spark. Although Crucial is not cost-efficient per se, we believe that it would be a good option to implement a parameter server-like interface for server ML training.

Serverless ML. A number of works have been devoted to leveraging FaaS platforms for building ML systems. Since ML model inference is a representative use case of serverless computing [4,18], recent research efforts have been directed towards model training [6,21,44]. All these works use AWS Lambda, which confers them some advantage over MLLess, and make direct comparison problematic. First, AWS Lambda enables multi-threaded parallelism [6,30], while IBM Cloud Functions are limited to 1vCPU at most. Also, AWS Lambda workers can access 10GB of local RAM, which allows them to hold larger data partitions and mini-batches compared with MLLess that is restricted to 2GB of memory. Despite this, MLLess outweighs these limitations and manages to deliver speedups superior to 15X and with 6.3X lower cost than PyTorch, and very importantly, *excluding the start-up time,* which is longer in PyTorch (e.g., a cluster of 6 VMs takes > 1 min. to boot up).

To put in a nutshell, Cirrus [6] is a serverless ML system that implements a parameter server (PS) [20] 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 [23], Cirrus is 3X-5X faster than VMs, but up to 7X more costly. Compared to MLLess, Cirrus is thus not cost-efficient, mostly because it does not exploit well the definitory properties of the FaaS model such as "pay-per-usage", which allows to save money through the fine-grained, dynamic allocation of serverless workers.

Siren [44] 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.

Finally, [21] proposes LambdaML, a FaaS-based training system to determine the cases where FaaS holds a sway over IaaS. Interestingly, our results mirrors their observation that FaaS is more cost-efficient for models that quickly converge. Unlike LambdaML, however, we reach the same conclusion without penalizing PyTorch with the start-up time. In any public cloud such as IBM Cloud, it is well-known that starting a VM instance takes much more time than launching an equivalent number of serverless functions. As reported by the own authors of LambdaML, while it took them to spend more than 2 minutes to start a 10-server EC2 cluster, unleashing the equivalent computing power in terms of serverless functions just took 1.3 seconds, i.e., 60X less time. Compared with them, we have shown in this paper that MLLess outperforms PyTorch, yet being cheaper, without needing to include the start-up time to artificially decrease the cost-efficiency of Pytorch running on a cluster of VMs. That is, to avoid any bias in the results, we only considered the job execution time, which provides more generalizable results. In particular, if the start-up time were to be excluded, LambdaML would become 670 slower than PyTorch, while MLLess would continue to be more cost-efficient. In this sense, we believe that MLLess opens the door to the adoption of serverless ML training as a truly cost-efficient option in the cloud.

8. Conclusion and future work

We have examined the question of whether serverless ML training can be more cost-effective than traditional IaaS-based 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 demonstrate that MLLess is more cost-efficient than serverful ML libraries at a lower cost for ML models with fast convergence when training on CPUs. We also validate the scalability of MLLess, and the benefits of loose synchronization models that allow to smoothly trade off communication bandwidth and convergence time.

The cost-effectiveness of serverless ML training suggests a variety of potential future work. An interesting avenue of research would be to investigate the advantage of supporting ML-specific logic on the server side through serverless data stores (e.g., Crucial [2,3]). Another topic would be to adapt MLLess to Federated Learning (FL) environments. A commonplace practice in FL is to select a random subset of the available clients in each training step, which results in many clients staying idle for a long time. By extending MLLess to run the clients as functions on edge devices only when needed, it would be possible to improve cost-efficiency. A final research topic would be to examine the potential effects of lossy gradient compression techniques such as gradient quantization on serverless ML training.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This research work has been partly supported by the EU through the projects Extract (No. 101092749), NearData (No. 101092644) and CloudSkin (No. 101092646), and the Spanish Government under grant agreement PID2019-106774RB-C22. Marc Sánchez-Artigas is a Serra Húnter Fellow. Pablo Gimeno is a Martí Franquès Research Grant Fellow - Banco Santander Edition.

Appendix A. Convergence analysis

In this section, we show that the SGD algorithm for convex objectives is ensured 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. Nevertheless, 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:

$$\widetilde{\mathbf{x}}_{p,c} := \mathbf{x}_0 + \sum_{c'=0}^{c} \mathbf{u}_{p,c} + \sum_{p' \neq p} \sum_{i \in \mathcal{S}_{p',c}} \mathbf{u}_i, \tag{5}$$

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. (5), the difference between the "true" view $\mathbf{x}_t$ and the noisy view $\widetilde{\mathbf{x}}_t$ becomes:

$$\widetilde{\mathbf{x}}_t - \mathbf{x}_t = \widetilde{\mathbf{x}}_{p,c} - \mathbf{x}_t = \widetilde{\mathbf{x}}_{t \bmod P, \lfloor \frac{t}{P} \rfloor} - \mathbf{x}_t = -\sum_{p' \neq p} \sum_{i \in \mathcal{I}_{p',c}} \mathbf{u}_i \tag{6}$$

Equipped with Eq. (6), we are now ready to start the proof of Theorem 1.

Similarly to [16], the Insignificance-bounded Synchronous Parallel (ISP) generalizes the BSP model:

Corollary 1. *For zero significance threshold $v = 0$, ISP reduces to BSP.*

Proof. Observe that $v = 0$ 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 ith 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 ith 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 $\lim_{T \to \infty} \frac{R[X]}{T} = 0$.

Proof. We follow the proof of [16]. 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:

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

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 $R[X]$:

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

We now upper-bound each of the terms:

$$\begin{aligned} \sum_{t=1}^{T} \frac{1}{2}\eta_t \|\widetilde{\mathbf{g}}_t\|^2 &\leq \sum_{t=1}^{T} \frac{1}{2}\eta_t L^2 \qquad (L\text{—Lipschitz assumption}) \\ &= \frac{1}{2}\eta L^2 \sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq \eta L^2 \sqrt{T}, \quad \left(\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{T} \right) \end{aligned} \tag{7}$$

and

$$\begin{aligned} &\sum_{t=1}^{T} \frac{D(\mathbf{x}^*, \mathbf{x}_t) - D(\mathbf{x}^*, \mathbf{x}_{t+1})}{\eta_t} = \\ &\quad \frac{D(\mathbf{x}^*, \mathbf{x}_1)}{\eta_1} - \frac{D(\mathbf{x}^*, \mathbf{x}_{T+1})}{\eta_T} + \sum_{t=2}^{T} \left[D(\mathbf{x}^*, \mathbf{x}_t) \left(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}} \right) \right] \\ &\quad \leq \frac{\Delta^2}{\eta} - 0 + \frac{\Delta^2}{\eta} \sum_{t=2}^{T} \left[\sqrt{t} - \sqrt{t-1} \right] \text{(Bounded diameter)} \\ &\quad = \frac{\Delta^2}{\eta} + \frac{\Delta^2}{\eta} \left[\sqrt{T} - 1 \right] = \frac{\Delta^2}{\eta} \sqrt{T}. \end{aligned} \tag{8}$$

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 $\frac{\|\mathbf{x}\|}{\|\mathbf{y}\|} \leq \|\mathbf{x} \oslash \mathbf{y}\|$, 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,

$$\begin{aligned}
&\sum_{t=1}^{T} \sum_{p' \neq p} \left[- \sum_{i \in \mathcal{I}_{p',c}} \eta_i \langle \widetilde{\mathbf{g}}_i, \widetilde{\mathbf{g}}_t \rangle \right] \\
&\quad \leq \sum_{t=1}^{T} (P-1) \left\langle \mathbf{s}_{p',c}, \widetilde{\mathbf{g}}_t \right\rangle \leq (P-1) \sum_{t=1}^{T} \left| \left\langle \mathbf{s}_{p',c}, \widetilde{\mathbf{g}}_t \right\rangle \right| \\
&\quad \leq (P-1) \sum_{t=1}^{T} \|\mathbf{s}_{p',c}\| \|\widetilde{\mathbf{g}}_t\| \quad \text{(Cauchy–Schwarz inequality)} \\
&\quad \leq (P-1) L \sum_{t=1}^{T} \frac{|\mathbf{s}_{p',c}|}{|\widetilde{\mathbf{x}}_t|} |\widetilde{\mathbf{x}}_t| \qquad (L\text{−Lipschitz assumption}) \\
&\quad \leq (P-1) L \sum_{t=1}^{T} \left| \mathbf{s}_{p',c} \oslash \widetilde{\mathbf{x}}_t \right| |\widetilde{\mathbf{x}}_t| \quad \text{(Lemma 2)} \\
&\quad \leq (P-1) Ln \sum_{t=1}^{T} v_t |\widetilde{\mathbf{x}}_t| \qquad \text{(Non-significance of updates)} \\
&\quad \leq (P-1) Ln \sum_{t=1}^{T} v_t \sqrt{n} \|\widetilde{\mathbf{x}}_t\|
\end{aligned} \tag{9}$$

$$\begin{aligned}
&\leq (P-1) L \left(n\sqrt{n}\right) \sum_{t=1}^{T} v_t \sqrt{2}\Delta \qquad \text{(Bounded diameter)} \\
&\leq \sqrt{2}\Delta (P-1) L \left(n\sqrt{n}\right) \sum_{t=1}^{T} \frac{v}{\sqrt{t}} \\
&\leq 2\sqrt{2}\Delta (P-1) L \left(n\sqrt{n}\right) v\sqrt{T}. \quad \left(\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{T} \right)
\end{aligned} \tag{10}$$

Hence,

$$\begin{aligned}
R[X] &\leq \eta L^2 \sqrt{T} + \frac{\Delta^2}{\eta} \sqrt{T} \\
&\quad + 2\sqrt{2}\Delta (P-1) L \left(n\sqrt{n}\right) v\sqrt{T} = O\left(\sqrt{T}\right),
\end{aligned} \tag{11}$$

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

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, Tensorflow: a system for large-scale machine learning, in: 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16), 2016, pp. 265–283.

[2] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, P. García-López, On the faas track: building stateful distributed applications with serverless architectures, in: 20th International Middleware Conference (Middleware '19), 2019, pp. 41–54.

[3] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, P. García-López, Stateful serverless computing with crucial, ACM Trans. Softw. Eng. Methodol. 31 (3) (2022).

[4] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, A. Gokhale, T. Damiano, 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), 2019, pp. 59–61.

[5] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, USA, 2004.

[6] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, R. Katz, Cirrus: a serverless framework for end-to-end ml workflows, in: ACM Symposium on Cloud Computing (SoCC '19), 2019, pp. 13–24.

[7] T. S. community, Scipy.optimize.curve_fit, SciPy V1.6.3 Reference Guide [Online], apr 2021. (Accessed 28 April 2021).

[8] L. Feng, P. Kudva, D. Da Silva, J. Hu, Exploring serverless computing for neural network training, in: IEEE 11th International Conference on Cloud Computing (CLOUD'18), 2018, pp. 334–341.

[9] P. Fermín-Cueto, E. McTurk, M. Allerhand, E. Medina-Lopez, M.F. Anjos, J. Sylvester, G. dos Reis, Identification and machine learning prediction of knee-point and knee-onset in capacity degradation curves of lithium-ion cells, Energy AI 1 (2020) 100006.

[10] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, K. Winstein, From laptop to lambda: outsourcing everyday jobs to thousands of transient functional containers, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 475–488.

[11] A. Gibiansky, Bringing hpc techniques to deep learning [Online], http://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/, feb 2017. (Accessed 20 February 2021).

[12] Gloo: a collective communications library [Online], https://github.com/facebookincubator/gloo, feb 2021. (Accessed 22 February 2021).

[13] P. Goyal, P. Dollár, R.B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch SGD: training imagenet in 1 hour, CoRR, arXiv:1706.02677.

[14] F.M. Harper, J.A. Konstan, The movielens datasets: history and context, ACM Trans. Interact. Intell. Syst. 5 (4) (2015) 19.

[15] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Serverless computation with openlambda, in: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16), 2016.

[16] Q. Ho, J. Cipar, H. Cui, J.K. Kim, S. Lee, P.B. Gibbons, G.A. Gibson, G.R. Ganger, E.P. Xing, More effective distributed ml via a stale synchronous parallel parameter server, in: 26th International Conference on Neural Information Processing Systems - Volume 1, NIPS'13, 2013, pp. 1223–1231.

[17] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G.R. Ganger, P.B. Gibbons, O. Mutlu, Gaia: geo-distributed machine learning approaching LAN speeds, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, 2017, pp. 629–647.

[18] V. Ishakian, V. Muthusamy, A. Slominski, Serving deep learning models in a serverless platform, in: IEEE International Conference on Cloud Engineering (IC2E'18), 2018, pp. 257–262.

[19] B. Jiang, et al., 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), 2019.

[20] J. Jiang, B. Cui, C. Zhang, L. Yu, Heterogeneity-aware distributed parameter servers, in: 2017 ACM International Conference on Management of Data (SIGMOD '17), 2017, pp. 463–478.

[21] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, C. Zhang, Towards demystifying serverless machine learning training, in: ACM SIGMOD International Conference on Management of Data (SIGMOD'21), 2021.

[22] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: distributed computing for the 99%, in: 2017 Symposium on Cloud Computing (SoCC'17), 2017, pp. 445–451.

[23] E. Jonas, et al., Cloud programming simplified: a Berkeley view on serverless computing, CoRR, arXiv:1902.03383.

[24] N.P. Jouppi, et al., In-datacenter performance analysis of a tensor processing unit, Comput. Archit. News 45 (2) (2017) 1–12.

[25] C. Labs, Kaggle display advertising challenge dataset [Online], http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/, feb 2014. (Accessed 15 May 2021).

[26] M. Li, T. Zhang, Y. Chen, A.J. Smola, Efficient mini-batch training for stochastic optimization, in: 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14), 2014, pp. 661–670.

[27] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, S. Chintala, Pytorch distributed: experiences on accelerating data parallel training, Proc. VLDB Endow. 13 (12) (2020) 3005–3018.

[28] D. Masters, C. Luschi, Revisiting small batch training for deep neural networks, CoRR, arXiv:1804.07612.

[29] F. McSherry, M. Isard, D.G. Murray, Scalability! But at what cost?, in: 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15), 2015, p. 14.

[30] I. Müller, R. Marroquín, G. Alonso, Lambada: interactive data analytics on cold data using serverless cloud infrastructure, in: 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20), 2020, pp. 115–130.

[31] A. Or, H. Zhang, M. Freedman, Resource elasticity in distributed deep learning, in: I. Dhillon, D. Papailiopoulos, V. Sze (Eds.), Machine Learning and Systems, vol. 2, 2020, pp. 400–411, https://proceedings.mlsys.org/paper/2020/file/006f52e9102a8d3be2fe5614f42ba989-Paper.pdf.
[32] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Guo, Optimus: an efficient dynamic resource scheduler for deep learning clusters, in: Thirteenth EuroSys Conference (EuroSys '18), 2018.
[33] M. Perron, R. Castro Fernandez, D. DeWitt, S. Madden, Starling: a scalable query engine on cloud functions, in: 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20), 2020, pp. 131–141.
[34] Q. Pu, S. Venkataraman, I. Stoica, Shuffling, fast and slow: scalable analytics on serverless infrastructure, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 2019, pp. 193–206.
[35] H. Robbins, S. Monro, A stochastic approximation method, Ann. Math. Stat. 22 (3) (1951) 400–407.
[36] R. Salakhutdinov, A. Mnih, Probabilistic matrix factorization, in: 20th International Conference on Neural Information Processing Systems (NIPS'07), 2007, pp. 1257–1264.
[37] J. Sampé, G. Vernik, M. Sánchez-Artigas, P. García-López, Serverless data analytics in the ibm cloud, in: 19th International Middleware Conference Industry (Middleware'18), 2018, pp. 1–8.
[38] J. Sampé, P. García-López, M. Sánchez-Artigas, G. Vernik, P. Roca-Llaberia, A. Arjona, Toward multicloud access transparency in serverless computing, IEEE Softw. 38 (1) (2021) 68–74.
[39] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, P. García-López, Outsourcing data processing jobs with lithops, IEEE Trans. Cloud Comput. 11 (1) (2023) 1026–1037, https://doi.org/10.1109/TCC.2021.3129000.
[40] M. Sánchez-Artigas, P.G. Sarroca, Experience paper: towards enhancing cost efficiency in serverless machine learning training, in: 22nd International Middleware Conference (Middleware'21), 2021, pp. 210–222.
[41] V. Satopaa, J. Albrecht, D. Irwin, B. Raghavan, Finding a "kneedle" in a haystack: detecting knee points in system behavior, in: 31st International Conference on Distributed Computing Systems Workshops (ICDCSW '11), 2011, pp. 166–171.
[42] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, S. Venkataraman, Serverless linear algebra, in: 11th ACM Symposium on Cloud Computing (SoCC '20), 2020, pp. 281–295.
[43] S.L. Smith, P.-J. Kindermans, Q.V. Le, Don't decay the learning rate, increase the batch size, in: International Conference on Learning Representations (ICLR'18), 2018.
[44] H. Wang, D. Niu, B. Li, Distributed machine learning with a serverless architecture, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1288–1296.
[45] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10), 2010, p. 10.
[46] H. Zhang, L. Stafman, A. Or, M.J. Freedman, Slaq: quality-driven scheduling for distributed machine learning, in: 2017 Symposium on Cloud Computing (SoCC'17), 2017, pp. 390–404.

Pablo Gimeno Sarroca is a second-year PhD student at Universitat Rovira i Virgili, Spain. He received a bachelor's degree in Computer Engineering in 2020 from Universitat Rovira i Virgili, as well as a joint master's degree in Computational Engineering and Mathematics in 2021 from Universitat Rovira i Virgili and Universitat Oberta de Catalunya. He has recently published in ACM/IFIP Middleware'21 and IEEE INFOCOM'23. His main research interests are cloud computing, specifically serverless computing, in addition to machine learning and stream data processing.

Marc Sánchez-Artigas received his Ph.D. degree in Computer Science in 2009 from the Universitat Pompeu Fabra, Spain. During his Ph.D. studies, he worked at Ecole Polytechnique Fédérale de Lausanne (EPFL). In the same year, he joined the Universitat Rovira i Virgili, where he currently works as an Associate Professor. He received the Best Paper Award from IEEE LCN'07 and the Best Dataset Award from ACM IMC'15. He has published 80+ articles in important venues such as IEEE P2P, ACM/IFIP Middleware, IEEE ICDCS, IEEE INFOCOM and USENIX FAST, among others. Now, he is coordinating the Horizon Europe project CloudSkin (grant no. 101092646), as well as actively being involved in the coordination of several Spanish and European projects in Cloud and Edge Computing.

Update

Journal of Parallel and Distributed Computing
Volume 189, Issue , July 2024, Page

DOI: https://doi.org/10.1016/j.jpdc.2024.104871

Journal of Parallel and Distributed Computing 189 (2024) 104871

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Corrigendum

Corrigendum to "MLLess: Achieving Cost Efficiency in Serverless Machine Learning Training" [Journal of Parallel and Distributed Computing 183 (2024) 104764]

Pablo Gimeno Sarroca, Marc Sánchez-Artigas *

Computer Science and Maths, Universitat Rovira i Virgili, Spain

The authors regret that in the "Acknowledgements" section of the article, the grant agreement id for the Horizon Europe project "Extract" should be 101093110 instead of 101092749.

The authors would like to apologise for any inconvenience caused.

DOI of original article: https://doi.org/10.1016/j.jpdc.2023.104764.

* Corresponding author.
E-mail address: marc.sanchez@urv.cat (M. Sánchez-Artigas).

https://doi.org/10.1016/j.jpdc.2024.104871

Available online 13 March 2024
0743-7315/© 2023 The Author(s). Published by Elsevier Inc. All rights reserved.