The VLDB Journal
https://doi.org/10.1007/s00778-023-00813-0

REGULAR PAPER

A systematic evaluation of machine learning on serverless infrastructure

Jiawei Jiang[1,2] · Shaoduo Gan[4] · Bo Du[3] · Gustavo Alonso[4] · Ana Klimovic[4] · Ankit Singla[5] · Wentao Wu[6] · Sheng Wang[3] · Ce Zhang[4]

Received: 25 May 2022 / Revised: 1 August 2023 / Accepted: 17 August 2023

Abstract

Recently, the serverless paradigm of computing has inspired research on its applicability to data-intensive tasks such as ETL, database query processing, and machine learning (ML) model *training*. Recent efforts have proposed multiple systems for training large-scale ML models in a distributed manner on top of serverless infrastructures (e.g., AWS Lambda). Yet, there is so far no consensus on the design space for such systems when compared with systems built on top of classical "serverful" infrastructures. Indeed, a variety of factors could impact the performance of training ML models in a distributed environment, such as the optimization algorithm used and the synchronization protocol followed by parallel executors, which must be carefully considered when designing serverless ML systems. To clarify contradictory observations from previous work, in this paper we present a systematic comparative study of serverless and serverful systems for distributed ML training. We present a *design space* that covers design choices made by previous systems on aspects such as optimization algorithms and synchronization protocols. We then implement a platform, LambdaML, that enables a *fair* comparison between serverless and serverful systems by navigating the aforementioned design space. We further improve LambdaML toward automatic support by designing a hyper-parameter tuning framework that leverages the ability of serverless infrastructure. We present empirical evaluation results using LambdaML on both single training jobs and multi-tenant workloads. Our results reveal that there is no "one size fits all" serverless solution given the current state of the art—one must choose different designs for different ML workloads. We also develop an *analytic model* based on the empirical observations to capture the cost/performance tradeoffs that one has to consider when deciding between serverless and serverful designs for distributed ML training.

Keywords Serverless computing · Distributed machine learning

Shaoduo Gan and Jiawei Jiang have contributed equally.

✉ Bo Du
dubo@whu.edu.cn

Jiawei Jiang
jiawei.jiang@whu.edu.cn

Shaoduo Gan
sgan@inf.ethz.ch

Gustavo Alonso
alonso@inf.ethz.ch

Ana Klimovic
ana.klimovic@inf.ethz.ch

Ankit Singla
ankit.singla@inf.ethz.ch

Wentao Wu
wentao.wu@microsoft.com

Sheng Wang
swangcs@whu.edu.cn

Ce Zhang
ce.zhang@inf.ethz.ch

1 School of Computer Science, Wuhan University, Wuhan, China

2 OceanBase, Ant Group, Hangzhou, China

3 School of Computer Science, National Engineering Research Center for Multimedia Software, Institute of Artificial Intelligence, Hubei Key Laboratory of Multimedia and Network Communication Engineering, Wuhan University, Wuhan, China

4 Department of Computer Science, ETH Zürich, Zurich, Switzerland

5 Google, Zurich, Switzerland

Published online: 20 September 2023

1 Introduction

Recently, serverless computing has emerged as a new paradigm [13, 58] offered by major cloud platforms, such as AWS Lambda, Azure Functions, and Google Cloud Functions. Serverless computing is favored by many applications (e.g., event processing, API composition, API aggregation, data flow control, etc. [5]) as it lifts the burden of provisioning and managing cloud computation resources (e.g., with auto-scaling) from application developers. Additionally, serverless computing offers a novel "pay by usage" pricing model and can be more cost-effective compared to traditional "serverful" cloud computing that charges users based on the amount of computation resources being reserved. With serverless, the user specifies a *function* that he or she hopes to execute and is only charged for the duration of the function execution. The users can also easily scale up the computation by specifying the number of such functions that are executed concurrently. In this paper, we use the term **FaaS** (function as a service) to denote the serverless infrastructure and use the term **IaaS** (infrastructure as a service) to denote the VM-based infrastructure.

(Data-intensive Workloads) Although FaaS is initially developed for web microservices and IoT applications, there is a trend of exploring FaaS in data-intensive applications, which stimulates intensive interests in the data management community [3, 24, 38, 56, 66]. Previous work has shown that adopting a serverless infrastructure for *certain* types of workloads can significantly lower the cost. Example workloads range from ETL [20] to infrequent analytical queries over cold data [49, 53]. These data management workloads benefit from serverless computing by taking advantage of elasticity, pay per use, lower set-up overhead and faster start-up. Although the full potential and impact of serverless infrastructure on data management systems remain unclear, there are data management workloads for which such an infrastructure *seem* to be a great fit.

(FaaS-based ML Training) Modern data management systems are increasingly tightly integrated with advanced analytics such as data mining and machine learning (ML). Focusing on *performance* and *scalability*, the database community has been one of the driving forces behind recent advancement of distributed machine learning [9, 18, 27, 45, 60, 70].

Inspired by the emerging technological trends in cloud computing and machine learning, in this paper we focus on their intersections by enabling distributed ML *training* on top of serverless computing. While FaaS is a natural choice for ML inference [28], it is unclear whether FaaS can also be beneficial for ML training. Our goal in this work is to *understand the system tradeoff of supporting distributed ML training with serverless infrastructures.* Specifically, we are interested in the following question:

> *When can a serverless infrastructure (FaaS) outperform a VM-based, "serverful" infrastructure (IaaS) for distributed ML training?*

(The Current Landscape) Existing work depicts a complex picture of the relative performance of IaaS and FaaS for data processing (including ML training). On one hand, Hellerstein et al. [23] show 21× to 127× performance gap, with FaaS lagging behind IaaS because of the overhead of data loading and the limited computation power. On the other hand, Fonseca et al. [12] in their CIRRUS system, Gupta et al. [22] in their OVERSKETCHED NEWTON algorithm, and Wang et al. [64] in their SIREN system, depict a more promising picture in which FaaS is 2× to 100× faster than IaaS on a range of workloads. Despite these early explorations, it remains challenging for a practitioner to reach a firm conclusion. In many of these systems, FaaS and IaaS are often not put onto the same ground for comparison. (1) The IaaS implementation could have used a better algorithm (i.e., the collection of training algorithms that IaaS considers should be a superset of those FaaS considers since it is more flexible in communication) and could have used a stronger system implementation (i.e., IaaS should take advantage of the latest advances in distributed learning); (2) The existing FaaS implementations could also be further optimized regarding the communication mechanism (e.g., a better communication pattern and synchronization protocol) and optimization algorithm (the one that achieves better convergence performance). Moreover, similar to the literature comparing IaaS and FaaS for *other* non-ML workloads [49, 53, 55], we also find that, for ML, there is a delicate system tradeoff in which FaaS only outperforms IaaS in specific regimes. A systematic depiction of the tradeoff space in FaaS ML training, with an analytical model, is largely lacking from existing literature of ML training.

Adding to this picture is the potential to explore more thoroughly the design decisions of an FaaS-based training system. For example, in CIRRUS, one design decision that the authors made is to have a parameter server hosted on an `m5.large` virtual machine. This type of hybrid design definitely has its merits, but it raises the question of *how far can a pure FaaS design go?* As we will see, for some workloads (e.g., communication-efficient and bursty) a pure FaaS design can also be competitive due to its flexibility, autoscaling, fast start-up, and pay by usage.

(Overview and Structure of Our Study) We conduct an extensive experimental study inspired by the current landscape of FaaS-based distributed ML training. Specifically, we

[6] Redmond, Microsoft Research, Washington, USA

systematically explore both the algorithm choice and system design for FaaS ML training and depict the tradeoff over a diverse range of models, training workloads, and infrastructure choices.

In addition to the depiction of this empirical tradeoff using today's infrastructure, we further

develop an analytical model that characterizes the tradeoff between FaaS and IaaS-based training, and use it to speculate the performance of potential configurations used by future *systems.*

In our study, we have considered (1) a diverse set of models (both simple and more complex deep models), (2) large-scale datasets (> 30–100 GB), (3) diverse workloads (both single training workload and multi-tenant workloads), (4) hyper-parameter tuning capabilities, and (5) diverse IaaS infrastructures (e.g., different CPU VMs and GPU VMs for deep learning). The goal of this work is not to champion one platform over the other, but to systematically depict the tradeoff to facilitate future research on both. We further organize our study into the following three parts:

Part 1—Optimizing ML Training Over FaaS Infrastructure. To fairly conduct our study, we need to optimize the performances of *both* FaaS and IaaS-based implementations. As IaaS-based training has been studied intensively, we simply take advantage of these advancements and pick a collection of state-of-the-art systems. Here, we focus on optimizing an FaaS-based system. Specifically, we apply *popular*, *existing* techniques in ML systems to the FaaS scenario. Although none of these techniques is new, our work is the first to put them together and analyze their tradeoff *in the context of serverless computing*. As we will see, choosing appropriate techniques often allows us to significantly outperform the design choices in previous work [12, 64], sometimes by orders of magnitude. Although the primary goal of this paper is not to optimize FaaS implementations, such optimizations are *prerequisites* for a fair comparison between FaaS and IaaS ML systems. Consolidating a large design space of existing techniques, we implement LambdaML, an FaaS-based ML training platform (Sect. 4). We study the above *design space* using LambdaML and observe significant improvement by choosing a *different* point in the design space. We further empower LambdaML with a *hyper-parameter tuning* module that can effectively leverage the auto-scaling nature of FaaS.

Part 2—FaaS versus IaaS ML on Single Training Job. Given a well-optimized implementation on top of an FaaS infrastructure, we are able to depict a precise picture of the tradeoff between FaaS and IaaS for ML training via *empirical evaluation* and *analytical modeling*. When there is a single training job, the principle that governs our insights can be summarized by an analytical model developed in Sect. 6.3 for FaaS and IaaS. With all the optimizations we described (Part 1), we observe varying tradeoffs for different models.

Part 3—FaaS versus IaaS ML on Multiple Training Jobs. We further explore the scenarios in which the system needs to serve multiple independent training jobs from different users. We consider different deployment scenarios (e.g., both on-demand and reserved resources), and then study the tradeoff between On-Demand IaaS, Reserved IaaS, and FaaS solutions. Some of these comparisons can be extrapolated from the single job comparison, while others do reveal new insights. The tradeoff is governed by two dimensions of the workload—*the number of concurrent requests* and *the frequency of the incoming requests*.

(Takeaway messages) We observe diverse insights from the empirical results. For single training jobs, FaaS only works for short-running and communication-efficient scenarios; when it comes to communication-intensive algorithms, an FaaS implementation is significantly slower than an IaaS implementation. For multi-tenant workloads, FaaS is superior to IaaS only when the workload is "bursty," short-running, *and* communication-efficient; otherwise, IaaS is better.

2 Related work

2.1 Distributed ML

Distributed machine learning is a category of machine learning tasks that scale the training of ML model to a cluster of machines. The parallel paradigm adopted by distributed ML includes data parallelism [47, 52], model parallelism [2, 29], and hybrid parallelism [32, 51].

Data parallelism is perhaps the most common strategy used by distributed ML systems, which partitions and distributes data evenly across workers. Each worker executes the training algorithm over its local partition and synchronizes with other workers from time to time. A typical implementation of data parallelism is parameter server [16, 25, 29, 42]. Another popular implementation is message passing interface (MPI) [21], e.g., the `AllReduce` MPI primitive leveraged by XGBoost [15], PyTorch [43], etc. [39]. We have also used data parallelism to implement LambdaML. Other research topics in distributed ML include compression [31, 69], decentralization [62, 63], synchronization [14, 26, 65], straggler [57, 61], data partition [1, 35], etc.

2.1.1 Distributed optimization

Different ML models rely on different optimization algorithms. Most of these optimization algorithms are *iterative*. In each iteration, the training procedure would typically scan the training data, compute necessary quantities (e.g., gradi-

ents), and update the model. When a single machine does not have the computation power or storage capacity (e.g., memory) to efficiently run an ML training job, one has to deploy and execute the job across multiple machines. Training ML models in a distributed setting is more complicated, due to the extra complexity of distributed computation as well as coordination of the communication between executors. Lots of distributed optimization algorithms have been proposed. Some of them are straightforward extensions of their single-node counterparts (e.g., k-means), while the others require more sophisticated adaptations dedicated to distributed execution environments (e.g., parallelized SGD [72], distributed ADMM [10]).

(Communication Mechanism) One key differentiator in the design and implementation of distributed optimization algorithms is the communication mechanism employed, including *communication channel*, *communication pattern*, and *synchronization protocol*. For communication channel, one can either rely on pure *message passing* between executors, or use a certain storage medium, such as a disk-based file system [64] or an in-memory key-value store [12, 30], to provide a central access point for the shareable global states. A number of collective communication patterns can be used for data exchange between executors, such as `Gather`, `AllGather`, `AllReduce`, and `ScatterReduce`. The iterative nature of the optimization algorithms introduces synchronizations between executors at certain boundary points. Two common protocols used by existing systems are *bulk synchronous parallel* (BSP) [47, 72] and *asynchronous parallel* (ASP) [57, 71].

2.2 Serverless data processing

Inspired by the increasing popularity of serverless computing, recently there have been quite a few studies devoted to leveraging these serverless platforms for large-scale data processing. For example, Locus [55] explores the trade-off of using fast and slow storage mediums when shuffling data under serverless architectures. Numpywren [59] is a linear algebra library built on top of a serverless architecture to achieve elasticity. Lambada [49] designs an efficient invocation approach for TB-scale data analytics. Starling [53] proposes a query execution engine for data analytics built on cloud function services. Jonas et al. [33] discuss the attractiveness and the limitation of current serverless computing platforms, and forecast the future directions in terms of system and networking. Moneyball [54] studies auto-scaling problem in serverless databases. Polardb [11] proposes a cloud native database for data centers.

(Serverless ML) Due to the elasticity of serverless computing, building ML systems on top of serverless infrastructures has emerged as a new research area. Since ML model inference is a straightforward use case of serverless computing [8, 28, 68], the focus of recent research effort has been on ML model training. For instance, Cirrus [12] is a serverless framework that supports end-to-end training of linear models. It utilizes a VM-based parameter server as storage service. In [19], the authors study training neural networks using AWS Lambda, and study different aggregation structures. SIREN [64] proposes an asynchronous distributed neural network training framework based on AWS Lambda, using S3 to store training data and a shared model. Hellerstein et al. [23] show 21× to 127× performance gap on the task of prediction serving, with FaaS lagging behind IaaS because of the overhead of data loading and the limited computation power. On the other hand, Fonseca et al. [12] in their Cirrus system, Gupta et al. [22] in their OverSketched Newton algorithm, and Wang et al. [64] in their SIREN system, depict a more promising picture in which FaaS is 2× to 100× faster than IaaS on a range of workloads. Despite these early explorations, it remains challenging for a practitioner to reach a firm conclusion about the relative performance of FaaS and IaaS for ML Training.

3 Preliminaries

(Data and Model) A training dataset D consists of n data examples that are i.i.d. samples generated by the underlying data distribution $\mathcal{D}$. Let $D = \{(\mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathbb{R})\}_{i=1}^N$, where $\mathbf{x}_i$ represents the *feature vector* and y_i represents the *label* of the i^{th} data example. The goal of ML training is to find an ML model $\mathbf{w}$ that minimizes a *loss function* f over the training dataset D: $\arg\min_{\mathbf{w}} \frac{1}{N} \sum_i f(\mathbf{x}_i, y_i, \mathbf{w})$.

(IaaS) In IaaS, users have to build a cluster by renting VMs or reserve a cluster with predetermined configuration parameters. As a result, users pay bills based on the computation resources that have been reserved, regardless of whether these resources are in use or not. Moreover, users have to manage the resources by themselves—there is no elasticity or auto-scaling if the reserved computation resources turn out to be insufficient, even for just a short moment (e.g., during the peak of a periodic or seasonal workload). Therefore, to tolerate such uncertainties, users tend to *overprovisioning* by reserving more computation resources than actually needed.

(FaaS) The move toward FaaS infrastructure lifts the burden of managing computation resources from users. Resource allocation is on-demand in Faas, and users are only charged by their actual resource usages. These features of FaaS are very attractive from the user's point of view. From the system-centric view, FaaS is potentially beneficial for building ML systems—(1) FaaS saves the efforts spent on system management for developers, (2) FaaS-based ML system can be easily integrated with other cloud services (e.g., storage, database, logging), (3) FaaS can increase the system throughput and decrease energy consumption through elas-

tic resource scheduling. However, current offerings by major cloud service providers impose certain limitations and/or constraints that lower some of the values of shifting from IaaS to FaaS. Current FaaS infrastructures only support *stateless* function calls with limited computation resource and duration. For instance, a function call in AWS Lambda has a maximal memory budget and a limited lifetime .[1] Such constraints automatically eliminate some simple yet natural ideas in implementing FaaS-based ML systems. For example, one cannot just wrap the code of SGD in an AWS Lambda function and execute it, which would easily run out of memory or hit the timeout limit on large training data.

4 LambdaML

We now conduct a systematic and in-depth study of the question: *How do we design an ML system using the current FaaS infrastructure?* Our methodology is to first explore the *design space* of building FaaS-based ML systems and then propose solutions to address new challenges arisen in each individual dimension.

4.1 Challenges and solutions

As mentioned in Sect. 3, one in general needs to consider four dimensions when developing distributed ML systems: (1) the *distributed optimization algorithm*, (2) the *communication channel*, (3) the *communication pattern*, and (4) the *synchronization protocol*. These elements remain valid when migrating ML systems from IaaS to FaaS, although new challenges arise. One main challenge is that *current FaaS infrastructures do not allow direct communication between stateless functions*. As a result, one has to use a certain storage channel to allow the functions to read/write intermediate state generated during the iterative training procedure.

Note that, there are other related techniques in distributed ML, e.g., data loading, data parallelism and model parallelism. In this work, since the major bottleneck of FaaS is communication, we focus on the above four dimensions and assume a prevailing setting—the worker loads the training data from a distributed file system and trains a model in a data parallelism manner.

4.1.1 Optimization algorithm

(Distributed SGD) Stochastic gradient descent (SGD) is perhaps the most popular optimization algorithm in today's world, partly attributed to the success of deep neural networks. We consider two variants when implementing SGD in a distributed manner: (1) *gradient averaging* (GA) and (2) *model averaging* (MA). In both implementations, we partition the training data evenly and have one executor be in charge of one partition. Each executor runs mini-batch SGD independently and in parallel, while sharing and updating the global ML model at certain synchronization barriers (e.g., after one or a couple of iterations). The difference lies in the way that the global model gets updated. GA updates the global model in *every* iteration by harvesting and aggregating the (updated) gradients from the executors. In contrast, MA collects and aggregates the local models, instead of the gradients, from the executors and does not force synchronization at the end of each iteration. That is, executors may combine the local model updates accumulated in a number of iterations before synchronizing with others to obtain the latest consistent view of the global model [70].

(Distributed ADMM) Alternating direction method of multipliers (ADMM) is another popular distributed optimization algorithm proposed by Boyd et al. [10]. ADMM breaks a large-scale convex optimization problem into several smaller subproblems, each of which is easier to handle. In a distributed setting, each executor solves one subproblem (i.e., until convergence of the local solution) and then exchanges local model parameters with other executors to obtain the latest view of the global model. While this paradigm has a similar communication pattern as MA, it has been shown that ADMM can have better convergence guarantees [10].

4.1.2 Communication channel

As we mentioned, it is necessary to have a storage component in an FaaS-based ML system to allow stateless functions to read/write intermediate state information generated during the lifecycle of ML training. Often, there are various options for this storage component, with a broad spectrum of cost/performance tradeoffs. For example, in Amazon AWS, one can choose between three alternatives—S3, ElastiCache for Redis, and ElastiCache for Memcached. S3 is a disk-based object storage service, whereas Redis and Memcached are in-memory key-value data stores provided by Amazon ElastiCache. In addition to using external cloud-based storage services, one may also consider building his/her own customized storage layer. For instance, Cirrus [12] implements a parameter server [30] on top of a dedicated virtual machine (VM) on AWS, which serves as the storage access point of the global model shared by the executors (implemented using AWS Lambda functions). This design, however, is not a pure FaaS architecture, as one has to maintain the parameter server by itself. We will refer to it as a *hybrid* design.

Different choices of the communication channel lead to different cost/performance tradeoffs. For example, on AWS, it usually takes some time to start an ElastiCache instance or

[1] https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

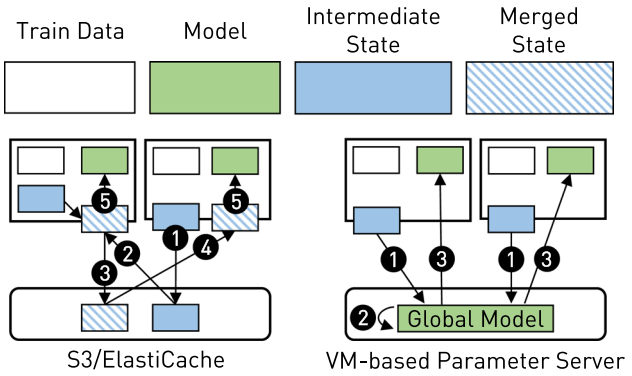


Fig. 1 Design of FaaS-based communication

a VM, whereas S3 does not incur such a startup delay since it is an "always-on" service. On the other hand, accessing files stored in S3 is in general slower but cheaper than accessing data stored in ElastiCache.

(An FaaS-based Scheme) We now design a communication scheme using a storage service, such as S3 or ElastiCache, as the communication channel. As shown in Fig. 1, the entire communication process contains the following steps: (1) Each executor stores its generated intermediate data as a temporary file in S3 or ElastiCache; (2) The first executor (i.e., the *leader*) pulls all temporary files from the storage service and merges them to a single file; (3) The leader writes the merged file back to the storage service; (4) All the other executors (except the leader) read the merged file from the storage service; (5) All executors refresh their (local) model with information read from the merged file.

Figure 1 also presents an alternative implementation using a VM-based parameter server, following the hybrid design exemplified by Cirrus [12]. In this implementation, (1) each executor pushes local updates to the parameter server, with which (2) the parameter server updates the global model. Afterward, (3) each executor pulls the latest model from the parameter server.

4.1.3 Communication pattern

To study the impact of communication patterns, we focus on two popular MPI primitives, `AllReduce` and `ScatterReduce` [70], that have been widely implemented in distributed ML systems. Here we assume using an external storage service such as S3 or ElastiCache.

(AllReduce) With `AllReduce`, all executors first write their local updates to the storage. Then the first executor (i.e., the leader) reduces/aggregates the local updates and writes the aggregated updates back to the storage service. Finally, all the other executors read the aggregated updates back from the storage service.

(ScatterReduce) When there are too many executors or a large amount of local updates to be aggregated, the single leader executor in `AllReduce` may become a performance bottleneck. This is alleviated by using `ScatterReduce`. Here, all executors are involved in the reduce/aggregate phase, each taking care of one partition of the local updates being aggregated. Specifically, assume that we have n executors. Each executor divides its local updates into n partitions, and then writes each partition separately (e.g., as a file) to the storage service. During the reduce/aggregate phase, the executor i ($1 \leq i \leq n$) collects the ith partitions generated by all executors and aggregates them. It then writes the aggregated result back to the storage service. Finally, each executor i pulls aggregated results produced by all other executors to obtain the entire model.

4.1.4 Synchronization protocol

We focus on two synchronization protocols that have been adopted by many existing distributed ML systems.

(Synchronous) We design a two-phase synchronous protocol, which includes a merging and an updating phase. We leverage an external storage service to implement this in FaaS architecture.

- *Merging phase.* All executors first write their local updates to the storage service. The reducer/aggregator (e.g., the leader in `AllReduce` and essentially every executor in `ScatterReduce`) then needs to make sure that it has aggregated local updates from all other executors. Otherwise, it should just wait.
- *Updating phase.* After the aggregator finishes aggregating all data and stores the aggregated information back to the storage service, all executors can read the aggregated information to update their local models and then proceed to the next round of training.

All executors will be synchronized properly using this two-phase framework. Moreover, one can rely on certain atomicity guarantees provided by the storage service to implement these two phases.

(Asynchronous) Following SIREN [64], the implementation of asynchronous communication is simpler. One replica of the trained model is stored on the storage service as a global state. Each executor runs independently—it reads the model from the storage service, updates the model with training data, writes the new model back to the storage service—without caring about the speeds of the other executors.

4.2 Implementation of LambdaML

In the following, we present the additional implementation details of LambdaML.

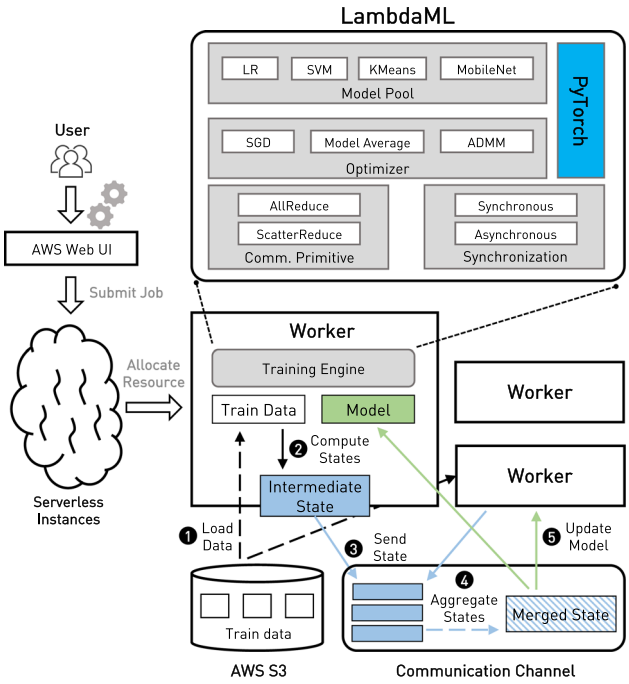


Fig. 2 Framework of LAMBDAML

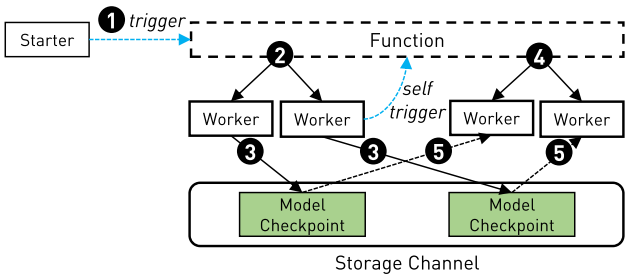


Fig. 3 Invocation structure of Lambda workers

4.2.1 Overview of LAMBDAML

Figure 2 shows the overview framework of LAMBDAML. The users upload their code and settings to Amazon AWS Lambda. The platform then allocates serverless resources and launches workers. We develop a training engine with PyTorch that handles model training and data communication. We assume that the user uploads the input dataset to AWS S3, and the input dataset is partitioned (LambdaML also provides utility tools to automatically partition the dataset). The worker creates the target model and initializes the model parameters. The training process is as follows: (1) the responsible training subset is retrieved from AWS S3, (2) intermediate states (e.g., gradients) are computed using training data and model parameters, (3) intermediate states are sent to communication channel, (4) intermediate states are merged and (5) each worker pulls the merged state and updates local model parameters accordingly. The above process iterates until convergence.

4.2.2 Synchronization implementation

Our proposed synchronous protocol contains a merging phase and an updating phase.

(Merging Phase) We name the files that store local model updates using a scheme that includes all essential information, such as the training epoch, the training iteration, and the partition ID. The reducer/aggregator can then request the list of file names from the storage service (using APIs that are presumed *atomic*[2]), filter out uninteresting ones, and then count the number of files that it has aggregated. Only if the number of aggregated files matches the number of workers should the aggregator proceed. Otherwise, it waits and keeps polling the storage service until the desired number of files is reported.

(Updating Phase) We name the merged file that contains the aggregated model updates in a similar manner, which consists of the training epoch, the training iteration, and the partition ID. For any executor that is pending on the merged file, it can then keep polling the storage service until the merged file shows up.

4.2.3 Handling limited lifetime

One major limitation of Lambda functions is their (short) lifetime, that is, the execution cannot be longer than 15 min. We implement a *hierarchical invocation* mechanism to schedule their executions, as illustrated in Fig. 3. Assume that the training data has been *partitioned* and we have one executor (i.e., a Lambda function) for each partition. We start Lambda executors with the following steps: (1) a *starter* Lambda function is triggered once the training data has been uploaded into the storage service, e.g., S3; (2) the starter triggers *n worker* Lambda functions where *n* is the number of partitions of the training data. Each worker is in charge of its partition, which is associated with metadata such as the path to the partition file and the ID of the partition/worker. Moreover, (3) a worker monitors its execution to watch for the 15-minute timeout. It pauses execution when the timeout is approaching, and saves a checkpoint to the storage service that includes the latest local model parameters. (4) It then resumes execution by triggering its Lambda function with a new worker. (5) The new worker inherits the same worker ID and thus would take care of the same training data partition (using model parameters saved in the checkpoint).

Note that, we establish a starter Lambda to ease the trigger of distributed training tasks. The user only needs to send the

[2] https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#ConsistencyModel.

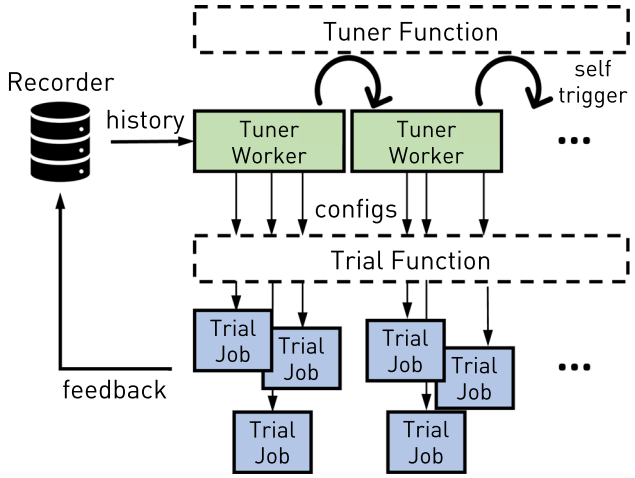


Fig. 4 Hyper-parameter tuning framework

code and settings to Lambda, and the rest is automatically managed by the starter function.

4.3 Hyper-parameter tuning

In the training process of an ML model, an important, yet expensive, step is tuning the related hyper-parameters, e.g., learning rate, batch size, regularization term, and number of workers. Since one major merit of serverless computing is ease-of-use, it is of great value to provide hyperparameter tuning functionality in LambdaML so that ordinary users do not need to have the tuning knowledge nor spend time on it. To this end, we implement a module for hyperparameter tuning inside LambdaML. However, the goal of this work is by no means to provide a complete solution for automatic machine learning in the serverless infrastructure. We believe, with our framework, other automatic machine learning techniques can be integrated in the future.

Regarding prior works, Hyperopt [7], Hyperband [41], and BOHB [17] are AutoML libraries; HyperSched [44] and RubberBand [48] study resource allocation and job scheduling in IaaS-based hyperparameter tuning. In contrast, our goal is to provide AutoML capabilities as a service in a pure FaaS environment that has autoscaling and pay-by-use advantages.

4.3.1 Framework of tuning module

Figure 4 shows the framework of hyper-parameter tuning. There are three components—*tuner function*, *trial function*, and *recorder*.

(Workload estimator) We have implemented an estimator that measures the space cost according to the dataset and model. Specifically, the estimated space cost is the sum of the dataset size, the model size, and all the intermediate data (e.g., forward activation and backward gradient). Conceptually, storing only input data, model, and the maximum size of live variables would be sufficient with techniques such as CPU offload and checkpoint. Our current version is designed for simplicity and independence of the underlying execution engine. Based on this estimator, we choose appropriate parallelism for each job.

(Tuner function) A tuner worker is launched by the tuner function to supervise the hyper-parameter tuning process. It periodically obtains history of hyper-parameters that have been evaluated, with which new hyper-parameters are generated. The tuner worker sends requests to the trial function to trigger training jobs with new configurations. Since the maximal concurrency of trials is restricted, we infer the number of currently running trials according to the number of trials ever submitted and the number of records in the recorder. If the maximal concurrency is met, the tuner function waits until there is a free quota. Besides, to resolve the limited lifetime of AWS Lambda (15 min), we leverage our proposed hierarchical invocation mechanism to overcome this limitation. Specifically, when the execution time of the tuner worker is approaching 15 min, the tuner worker triggers the tuner function, after which the new tuner worker starts running. We call this mechanism *self-trigger*.

(Trial function) Each training job is called a *trial* in this tuning framework. Once receiving the related hyper-parameters from the tuner worker, a trial job is launched and executed accordingly. Each trial job will independently launch its workers and then conduct training. When the trial job finishes, the leader worker will send the feedback to the recorder, including the used hyper-parameters and output metrics (e.g., model accuracy and runtime).

(Recorder) All the configurations and feedbacks are stored in a recorder. With the tuning history maintained, the tuner can perform better suggestions of new trials. In our implementation of the recorder, we use DynamoDB as the underlying storage. We store one record for each trial that contains the values of hyper-parameters, the execution time, and the quality of the output model.

4.3.2 Tuning approaches

During the tuning process, a critical technique is to generate new trials. Since each specific hyper-parameter has its range and possible values, we support different kinds of hyper-parameters and tuning strategies.

(Hyper-parameter Space) Since different hyper-parameters can have diverse data types and distributions, we provide various spaces for hyper-parameters. `ContHyper` represents a floating-point numerical hyper-parameter that is continuous within a range. `DiscHyper` represents a numerical hyper-parameter that contains discrete values (floating-point or integer) within a range. `CateHyper` represents a categor-

ical hyper-parameter that contains several mutually exclusive choices (e.g., the choice of optimizer).

(Tuning Strategy) We provide four strategies — `Grid Search`, `Random Search`, `Bayesian Optimization`, and `Bandit Algorithm`. These methods select hyper-parameter values in different manners. Grid search generates a set of trials by enumerating every possible combination of hyper-parameter values. Random search randomly chooses each hyper-parameter from possible values. Bayesian optimization is an effective method for optimizing blackbox, expensive, and potentially noisy functions that do not have any gradient information. We choose an open-source library, RoBO [36], in this work. Bandit algorithm formulates hyper-parameter optimization as a pure-exploration nonstochastic infinite-armed bandit problem, where a predefined resource like iterations or samples is allocated to randomly sampled configurations. We choose Hyperband algorithm that extends the SuccessiveHalving algorithm [41].

5 Evaluation of LambdaML

We evaluate LambdaML by comparing the design options covered in Sect. 4.1. We report evaluation results with respect to each dimension of the design space.

5.1 Experiment settings

(Datasets) Table 1 presents the datasets used in our evaluation. We choose these datasets due to their diverse dataset type, dataset size and feature dimensionality [4, 40]. In this section, we focus on the first three datasets to understand the system behavior, and leave the larger dataset (YFCC100M) to the next section when we conduct the end-to-end study. Higgs is a dataset for binary classification, produced by using Monte Carlo simulations. RCV1 is a two-class classification corpus of manually categorized newswire stories made available by Reuters [40]. Cifar10 is an image dataset that consists of 60 thousand 32×32 images categorized by 10 classes, with 6 thousand images per class. We resize the Cifar10 dataset to 224×224 pixel images so that they can be fed into the evaluated neural network models.[3]

(ML Models) We use the following ML models in our evaluation. Logistic Regression (**LR**) and Support Vector Machine (**SVM**) are linear models for classification that are trained by mini-batch SGD or ADMM.[4] MobileNet (**MN**) is a neural network model that uses depth-wise separable convolutions to build lightweight deep neural networks. The size of each input image is $224 \times 224 \times 3$, and the size of model parameters is 12MB. ResNet50 (**RN**) is a neural network model that was the first to introduce identity mapping and shortcut connection. The size of each input image is $224 \times 224 \times 3$, and the size of model parameters is 89MB. KMeans (**KM**) is a clustering model for unsupervised problems, trained by using *expectation maximization* (EM).

Table 1 Datasets used in this work

Dataset	Size	# Ins	# Feat	# Class
Cifar10	220 MB	60 K	1 K	10
RCV1	1.2 GB	697 K	47 K	2
Higgs	8 GB	11 M	28	2
YFCC100M	110 GB	100 M	4 K	2

(Protocols) We choose t2.medium instance for IaaS and LambdaML-3 G for FaaS since they have the similar memory size and vCPU cores. We randomly shuffle and split the data into a training set (90%) and a validation set (10%). The batch size for training **LR** over **Higgs** is 100K, whereas it is 128 for training **MN** and 32 for training **RN** over **Cifar10**. We tune the learning rate for each ML model in the range from 0.001 to 1.

5.2 Optimization algorithms

> *Carefully choosing the right algorithm is important in optimizing FaaS-based system, and the widely adopted SGD algorithm is not "one-size-fits-all."*

We implemented GA-SGD (i.e., SGD with gradient averaging), MA-SGD (i.e., SGD with model averaging), and ADMM, using ElastiCache for Memcached as the external storage service. Figure 5 presents the results for various data and ML models we tested.

(LR and SVM) When training **LR** on **Higgs** using 300 workers, GA-SGD is the slowest because transmitting gradients every batch can lead to high communication cost. ADMM converges the fastest, followed by MA-SGD. Compared with GA-SGD, MA-SGD reduces the communication frequency from every batch to every epoch, which can be further reduced by ADMM. Moreover, MA-SGD and ADMM can converge with fewer communication steps in spite of reduced communication frequency. We observe similar results when training **SVM** on **Higgs**: ADMM converges faster than GA-SGD and MA-SGD.

(MN) We have different observations when turning to training neural network models. Figure 5c presents the results of training **MN** on **Cifar10**. First, we note that ADMM is mostly used for optimizing convex objective functions and

[3] Although we did a lot of efforts to run larger image dataset (e.g., ImageNet), the performance is extremely slow, since Lambda does not provide GPUs and limits the maximal memory.

[4] We do not include regression models such as linear regression, but we believe the trade-off space would be the same, since the model complexity is similar.

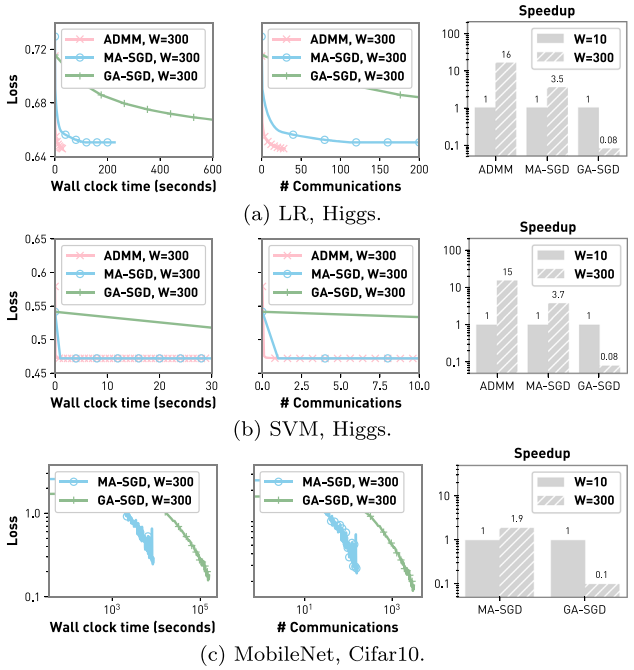


Fig. 5 Comparison of optimization algorithms. *W* denotes the number of workers

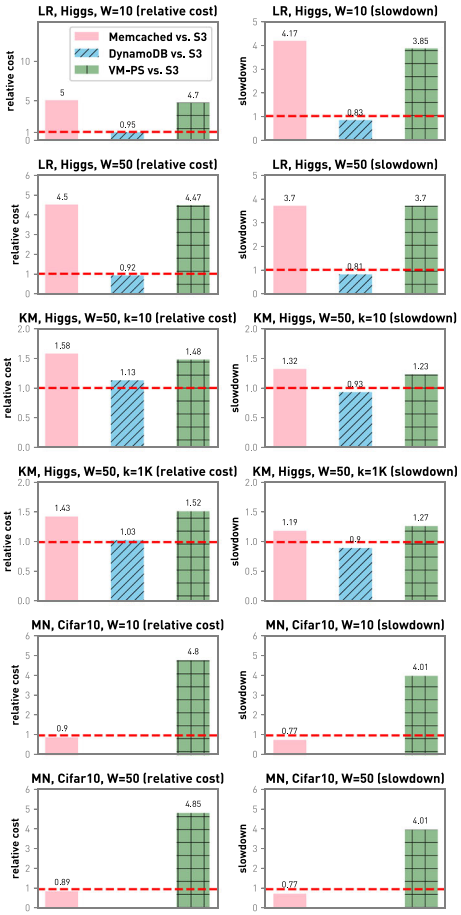


Fig. 6 Comparison of S3, Memcached, DynamoDB, and VM-based parameter server. We present the *slowdown* and *relative costs* of using different mediums w.r.t. using S3. A relative cost larger than 1 means S3 is cheaper, whereas a slowdown larger than 1 means S3 is faster

therefore is not suitable for training neural network models. Comparing GA-SGD and MA-SGD, we observe that the convergence of MA-SGD is unstable. On the other hand, GA-SGD can converge steadily and achieve a lower loss.

5.3 Communication channels

For many workloads, a pure FaaS architecture can be competitive to the hybrid design given the right choice of the algorithm. A dedicated PS can definitely help in principle, but its potential is currently bounded by the communication between FaaS and IaaS.

We next evaluate the impact of communication channels. **LR** is optimized by ADMM, **MN** is optimized by GA-SGD, and **KM** is optimized by EM. Figure 6 compares disk-based S3 with other memory-based mediums. A memory-based medium is *slower* than S3 if its slowdown is greater than 1; on the other hand, it is less economical than S3 if its *relative cost* is larger than 1.

(Pure FaaS Solutions) We compare S3 with Memcached, Redis, and DynamoDB.

– *Memcached versus S3.* Memcached introduces a lower latency than S3, therefore one round of communication using Memcached is significantly faster than using S3. Furthermore, Memcached has a well-designed multi-threading architecture. As a result, its communication is faster than S3 over a large cluster with up to 50 workers, showing 7× and 7.7× improvements when training **LR** and **KM**. Nonetheless, the overall execution time of Memcached is actually longer than S3, because it takes more than two minutes to start Memcached, whereas starting S3 is instant. Since **LR** and **KM** converge quickly on **Higgs**, the startup cost overshadows the savings on communication cost by using Memcached. When training **MN** on **Cifar10**, using Memcached becomes faster, since it takes much longer for **MN** to converge. Meanwhile, although the pricing of ElastiCache is higher than S3 (1000× higher per GB), the total cost of using Memcached is actually lower than using S3 as it saves the computation.
– *Redis versus Memcached.* According to our benchmark, Redis is similar to Memcached when training small ML models. However, when an ML model is large or is trained on a big cluster, Redis is inferior to Memcached since Redis lacks a similar high-performance multi-threading mechanism that underlies Memcached. Since Redis is

worse than Memcached inside ElastiCache, we do not present its results in Fig. 6.

– *DynamoDB versus S3.* Compared to S3, DynamoDB reduces the communication time by roughly 20% when training LR on Higgs, though it remains significantly slower than IaaS if the startup time is not considered. Nevertheless, DynamoDB only allows messages smaller than 400KB, making it infeasible for many median models or large models (e.g., **MN** on Cifar10).

(Hybrid Solutions) CIRRUS [12] presents a hybrid design—having a dedicated VM to serve as parameter server and all FaaS workers communicate with this centralized PS. This design definitely has its merit in principle—giving the PS the ability of doing data aggregation can potentially save 2× communication compared with an FaaS communication channel via S3/Memcached. However, we find that this hybrid design has several limitations, which limit the regime under which it outperforms a pure FaaS solution. Note that some of these limitations are artifacts of the existing FaaS platform, not fundamental to the hybrid strategy.

When training **LR** and **KM**, VM-based PS performs similarly as using Memcached or Redis, which are slower than S3 considering the start-up time. In this case, a pure FaaS solution is competitive even without the dedicated VM. This is as expected—when the model size is small and the runtime is relatively short, communication is not a significant bottleneck.

When the model is larger and the workload is more communication-intensive (**MN**), we would expect that the hybrid design performs significantly better. However, this is not the case *under the current infrastructure*. To confirm our claim, we use two RPC frameworks (Thrift and gRPC), vary CPUs in Lambda (by varying memory size from 1 to 3 GB), use different EC2 types, and evaluate the communication between Lambda and EC2 by transferring an array of 75 MB. We find several constraints of communication between Lambda and VM-based parameter server: (1) The communication speed from the PS is much slower than Lambda-to-EC2 bandwidth (up to 70 MBps reported by [37, 66]) and EC2-to-EC2 bandwidth (e.g., 10 Gbps for c5.4xlarge). In contrast, the hybrid solution takes at least 1.85 s to transfer 75 MB. (2) Increasing the number of vCPUs can decrease the communication time by accelerating data serialization and deserialization. But the serialization performance is eventually bounded by the limited CPU resource of Lambda (proportional to the memory size). (3) Model update on the parameter server is costly when the workload scales to a large cluster due to frequent locking operation of parameters. As a result, HybridPS is currently bounded not only by the network bandwidth, but also serialization/deserialization and model update. *However, if this problem is fixed, we would expect that a hybrid design might be a good fit for deep learning. We will explore this in Sect.* 6.4.1.

Table 2 Comparison of communication patterns

Model & Dataset	Model size	AllReduce	ScatterReduce
LR, Higgs, $W = 50$	224 B	9.2 s	9.8 s
MN, Cifar10, $W = 10$	12 MB	3.3 s	3.1 s
RN, Cifar10, $W = 10$	89 MB	17.3 s	8.5 s

W denotes the number of workers

5.4 Communication patterns

Communication using ScatterReduce is faster than AllReduce across different ML models.

We use another model, called ResNet50 (**RN**), in this study to introduce a larger model than **MN**. We train **LR** on **Higgs**, and **MN** and **RN** on **Cifar10**, using S3 as the storage service. Table 2 presents the time spent on communication by `AllReduce` and `ScatterReduce`. We observe that using `ScatterReduce` is slightly slower when training **LR**. Here, communication is not a bottleneck and `ScatterReduce` incurs extra overhead due to data partitioning. On the other hand, the communication costs of `AllReduce` and `ScatterReduce` are roughly the same when training **MN**. `AllReduce` is 2× slower than `ScatterReduce` when training **RN**, as communication becomes heavy and the aggregator in `AllReduce` becomes a bottleneck.

5.5 Synchronization protocols

Synchronous communication (BSP) is generally a better choice than asynchronous communication (ASP) given its stable convergence.

Finally, we study the impact of the two synchronization protocols: Synchronous and Asynchronous. Note that the asynchronous protocol here is different from ASP in traditional distributed learning. In traditional distributed learning, ASP is implemented in the parameter server architecture, where there is an in-memory model replica that can be requested and updated by workers [16, 25, 30]. However, this ASP routine is challenging, if not infeasible, in FaaS infrastructure. We thus follow SIREN [64] to store a global model on S3 and let every FaaS instance rewrite it. In this implementation, if the workers read the same model and compute gradients, only the last model rewriting is effective.

We use GA-SGD to train **LR** on **Higgs**, **LR** on **RCV1**, and **MN** on **Cifar10**, with Asynchronous or Synchronous enabled for the executors. As suggested by previous work [25], we use a learning rate decaying with rate $1/\sqrt{T}$ for

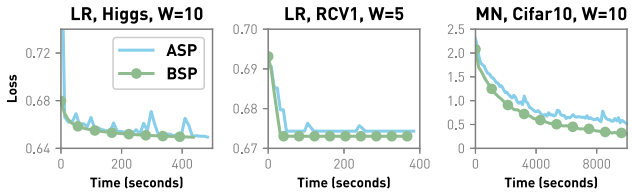


Fig. 7 Comparison of synchronization protocols

ASP where T denotes the number of iterations. Figure 7 presents the results. We observe that Synchronous converges steadily, whereas Asynchronous suffers from unstable convergence, although Asynchronous runs faster per iteration. The convergence problem of Asynchronous is caused by the inconsistency between local model parameters. Those faster executors may read stale model parameters from the stragglers. Consequently, the benefit of system efficiency brought by Asynchronous is offset.

6 FaaS versus IaaS for single training workload

We now turn to the second theme of this paper, where we compare FaaS and IaaS for *single model training*.

6.1 Empirical study

(Principles) We follow a set of principles:

1. **Best versus Best.** When comparing FaaS with IaaS, we compare the best configuration of an FaaS implementation with the best configuration of an IaaS implementation. This includes the algorithm that each implementation uses, the hyper-parameters, the VM types, the number of workers and VMs, etc.
2. **Strong IaaS Competitors.** When comparing with IaaS, we should compare with state-of-the-art machine learning systems.
3. **End-to-end Benchmark.** We focus on the end-to-end training performance—the wall-clock time (or cost in dollar) that each system needs to converge to the *same* loss.

6.1.1 Experimental settings

(Competing Systems) We compare our serverless ML system LambdaML with the following systems:

– *Distributed PyTorch.* We partition the training data and run PyTorch across multiple machines. We use all CPU cores on each machine, if possible. To manage a PyTorch cluster, we use StarCluster,[5] a toolkit for EC2 clusters. We use the `AllReduce` operator of PyTorch for cross-machine communication, and we implement both mini-batch SGD and ADMM for training linear models.
– *Distributed PyTorch on GPUs.* For deep learning models, we also consider GPU instances. The other settings are the same as above.
– *Angel.* Angel is an open-source ML system based on parameter servers [32]. Angel works on top of the Hadoop ecosystem (e.g., HDFS, Yarn, etc.) and we use Angel 2.4.0 in our evaluation.
– *HybridPS.* Following the hybrid architecture proposed by Cirrus [12], we implement a parameter server on a VM using gRPC, a cross-language RPC framework. Lambda instances use a gRPC client to pull and push data to the parameter server. We also implement the same SGD framework as in Cirrus.

Table 3 Experimental settings

Model, dataset	Setting	Threshold (loss, accuracy)
LR/SVM, Higgs	$W = 10$, $B = 10\,\text{K}$	0.66/0.48, 62%/62%
KM, Higgs	$W = 10$, $k = 10$	0.15
LR/SVM, RCV1	$W = 5$, $B = 2\,\text{K}$	0.68/0.05, 95%/92%
KM, RCV1	$W = 50$, $k = 3$	0.01
LR/SVM, YFCC	$W = 100$, $B = 800$	2/0.9, 95%/90%
KM, YFCC	$W = 100$, $k = 10$	50
MN, Cifar10	$W = 10$, $B = 128$	0.2, 90%
RN, Cifar10	$W = 10$, $B = 128$	0.4, 93%

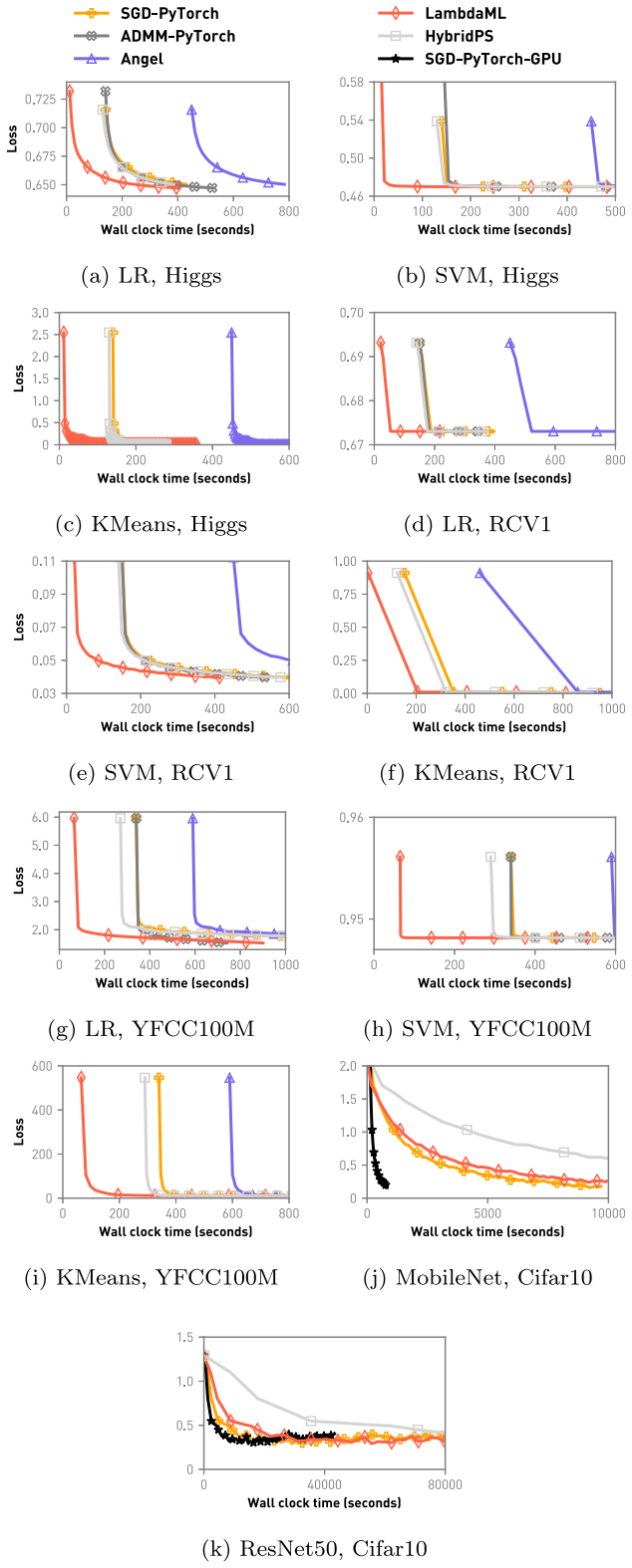
W denotes the number of workers, B the batch size, and k the number of clusters

(Datasets) In addition to **Higgs**, **RCV1** and **Cifar10**, Table 1 presents one more dataset **YFCC100M**,[6] which consists of approximately 99.2 million photos and 0.8 million videos. In YFCC100M, each image has several label tags and a feature vector of 4096 dimensions. We randomly sample 4 million data points, and convert this subset into a binary classification dataset by treating the "animal" tag as the positive label and the other tags as negative labels. After this conversion, there are about 300K (out of 4 M) positive data examples.

(ML Models) As shown in Table 3, we evaluate different ML models, including **LR**, **SVM**, KMeans (**KM**), MobileNet (**MN**), and ResNet50 (**RN**). As the readers might suspect, some previous works (e.g., Hogwild! [57]) have reported that a single machine can train SVM over RCV1 within ten seconds. Indeed, training a linear model on RCV1 is not costly since RCV1 is relatively small. Therefore, we treat linear

[5] http://star.mit.edu/cluster/

[6] http://projects.dfki.uni-kl.de/yfcc100m/



(a) LR, Higgs (b) SVM, Higgs

(c) KMeans, Higgs (d) LR, RCV1

(e) SVM, RCV1 (f) KMeans, RCV1

(g) LR, YFCC100M (h) SVM, YFCC100M

(i) KMeans, YFCC100M (j) MobileNet, Cifar10

(k) ResNet50, Cifar10

Fig. 8 End-to-end FaaS versus IaaS comparison

models on RCV1 as toy scenarios, and the other training workloads as realistic scenarios.

(Hardware) We tune the optimal EC2 instance from the t2 and c5 family. To run PyTorch on GPUs, we tune the optimal GPU instances from the g3 family. We use one c5.4xlarge instance as the parameter server in the hybrid architecture. For FaaS, we use 3GB memory for all tasks.

(Protocols) We choose the optimal learning rate between 0.001 and 1. We vary the number of workers from 1 to 150. Before running the competing systems, we partition the training data on S3. We trigger Lambda functions after the data is uploaded and Memcached is launched (if required). We use one cache.t3.small Memcached node. Each ADMM round scans the training data ten times. We stop training when the loss or accuracy is below a threshold, as summarized in Table 3.

("COST" Validation Check) Before we report end-to-end experimental results, we first report a sanity check as in COST [46] to make sure all scaled-up solutions outperform a single-machine solution. Taking Higgs, RCV1, and Cifar10 as examples, we store the datasets in a single machine and use a single EC2 instance to train the model and compare the performance with FaaS/IaaS.

For the Higgs dataset, using a single t2 instance (PyTorch) would converge in 960 s for LR trained by ADMM, while our FaaS (LambdaML) and IaaS (distributed PyTorch) solutions, using 10 workers, converge in 107 and 98 s. Similarly, on Higgs and SVM/KMeans, FaaS and IaaS achieve 9.4/6.2 and 9.9/7.2 speedups using 10 workers. We also choose a more powerful c4.xlarge instance, which is equipped with more CPUs than FaaS. The single-threading convergence time of LR is 719 s, and the convergence time using all four threads is 247 s. This verifies the benefit of FaaS and distributed IaaS on the evaluated workloads, compared with a single thread or multiple threads in a dedicated machine.

On RCV1 and LR, using a t2 instance converges in 380 s, including the startup time and data loading time; and using a c4.xlarge instance needs 352 s (one thread) and 186 s (four threads). FaaS (LambdaML) and IaaS (distributed PyTorch) solutions, using 10 workers, converge in 46 and 174 s. [7]

On Cifar10 and MobileNet, FaaS and IaaS achieve 4.8 and 6.7 speedups. We run distributed serverless jobs since: (1) the resource of serverless platform is limited, therefore using a single instance is infeasible for relatively large datasets or incurs too long runtime (e.g., deep learning models); (2) the maximal lifetime is limited, so that using a single instance may easily encounter timeout or multiple self-invocations;

[7] Note that, Hogwild! [57] uses a single machine to train RCV1 within 9.5 s (without startup and data loading time). The model training time of LambdaML is about 27 s. Hogwild! uses a lock-free asynchronous strategy and prefers sparse datasets. Although the training algorithm is different from our setting, we believe it is important to report these numbers as a reference.

Table 4 End-to-end convergence time (time to reach loss threshold)

Model, dataset	PyTorch	Angel	LambdaML	HybridPS
LR, Higgs	260	588	107	233
KM, Higgs	149	468	33	139
SVM, RCV1	226	610	87	213
LR, YFCC100M	358	711	103	389
MN, Cifar10	725	–	11397	33759
RN, Cifar10	5965	–	22217	88625

The numbers are in seconds. We report the best of PyTorch among SGD, ADMM, and GPU

(3) one main goal of this paper is to study the scalability of serverless ML.

6.1.2 Experimental results

End-to-end Comparison. We first run all competing systems with the same number of workers. We illustrate the end-to-end convergence w.r.t wall-clock time in Fig. 8, and report the convergence time of six representative tasks to reach the loss threshold in Table 4.

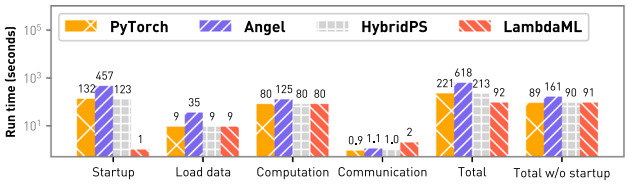
1. **LR, SVM, KM.** We train **LR**, **SVM**, and **KM** over YFCC100M. Angel is the slowest as a result of slow start-up and computation. Running ADMM on PyTorch is slightly faster than SGD, verifying ADMM saves considerable communication while assuring convergence meanwhile. HybridPS outperforms PyTorch as it only needs to launch one VM and it is efficient in communication when the model is relatively small. LambdaML achieves the fastest speed due to a swift start-up and the adoption of ADMM.
2. **MN.** As was analyzed above, data communication between Lambda and VM is bounded by the serialization overhead, and therefore the hybrid approach is slower than a pure FaaS approach with a large model. PyTorch is faster than LambdaML because communication between VMs is faster than using ElastiCache in Lambda. PyTorch-GPU is the fastest as GPU can accelerate the training of deep learning models.

Runtime Breakdown.

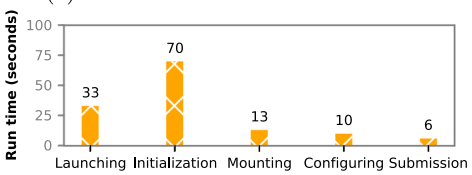
The fundamental tradeoff between start-up time and communication overhead makes FaaS-based implementation significantly faster on some datasets.

Figure 9a presents a breakdown for runtime, taking **LR** on **Higgs** (10 epochs) as an example.

1. **Start-up.** It takes more than 2 min to start a 10-node EC2 cluster, including the time spent on launching VMs, instance initialization, mounting shared volumes, configuring secure communication channels, and submitting the training job. We further conduct a breakdown for the start-up time of a 10-node PyTorch EC2 cluster (t2.xlarge instance) in Fig. 9b. It takes even more time to start Angel, as it needs to first start dependent libraries such as HDFS and Yarn. The hybrid solution also needs to start and configure VMs, but it avoids the time spent on submitting job due to quick startup of FaaS functions. In contrast, LambdaML took 1.3 s to start. The number is cold start-up time and does not include data loading. Since the platform does not reserve resources, we cannot recover previous transactions. The FaaS platform manages a pool of instances and allocates them to new jobs, so that the start-up is fast.
2. **Data Loading and Computation.** In terms of data loading and computation, PyTorch, HybridPS, and LambdaML spend similar amount of time because they all read datasets from S3 and use the same training engine. Angel spends more time on data loading since it loads data from HDFS. Its computation is also slower due to inefficient matrix calculation library.
3. **Communications.** Communication in LambdaML is slower than in other baselines since LambdaML uses S3 as the medium.
4. **Total Run Time.** In terms of the total run time, LambdaML is the fastest when including the startup time.



(a) End-to-end Runtime Breakdown



(b) EC2 Start-up Time Breakdown

Fig. 9 Time breakdown (LR, Higgs, $W = 10$, 10 epochs)

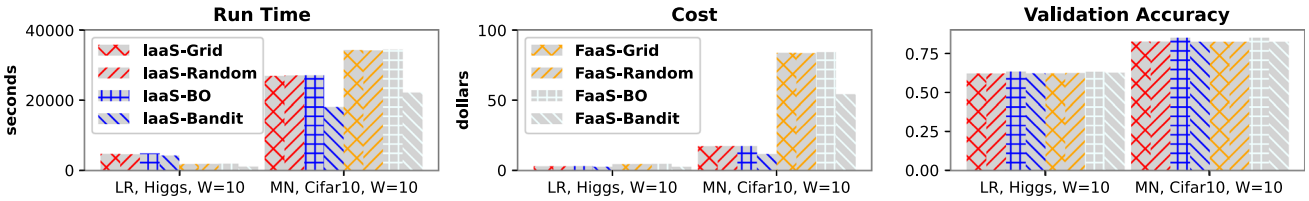


Fig. 10 Tuning learning rate and regularization term (time is in seconds and cost is in $)

However, if the startup time is excluded, PyTorch outperforms LAMBDAML. Especially, PyTorch does not incur start-up cost with reserved IaaS resources. One related issue is how to serve workloads during off-peak periods, as we will discuss for multi-tenant workloads.

6.2 Evaluation of hyper-parameter tuning

To evaluate our proposed framework for tuning hyper-parameters, we conduct experiments on Higgs and Cifar10 datasets, by tuning the learning rate and the number of workers.

(Tune Learning Rate and Regularization) For FaaS, we use our proposed hyper-parameter tuning framework, as elaborated in Sect. 4.3. Each trial job launches multiple workers (the number of workers is calculated by the workload estimator) and runs for 10 epochs using S3 as the communication medium, and the maximal number of concurrent trials is 5. We compare grid search, random search, Bayesian optimization (RoBO), and bandit algorithm (Hyperband). We tune learning rate within $[0.01, 1]$ and regularization term within $[0.001, 0.1]$ The total number of trials is 100, and we set $R = 10$ and $\eta = 3$ for Hyperband. The other settings are the same as Sect. 6.1.1. For IaaS, we run the tuner worker in a t2.medium instance and run each trial job for 10 epochs using ten t2.medium instances.

The results of hyper-parameter tuning are shown in Fig. 10, including end-to-end runtime, cost, and validation accuracy. On LR, FaaS is faster than IaaS, however, is not cheaper. Our proposed FaaS-based tuning framework benefits from the fast start-up and auto-scaling of serverless infrastructure, and hence achieves faster execution. We find that *GridSearch* and *RandomSearch* strategies achieve comparable validation accuracy, which resonates with prior works [6]. *BO* achieves slightly higher accuracies, at the expense of extra computation. The *bandit* algorithm achieves the fastest performance since the Hyperband approach we choose introduces early stopping. On MobileNet, however, IaaS is both significantly faster and cheaper than FaaS since each trial job is long-running and communication-intensive.

(Tune Number of Workers) Comparing FaaS and IaaS by forcing them to use the same number of workers is not necessarily a fair comparison—an end-to-end comparison should also tune the optimal number of workers to use for each case. We expect this could result in different trade-offs regarding runtime and cost.

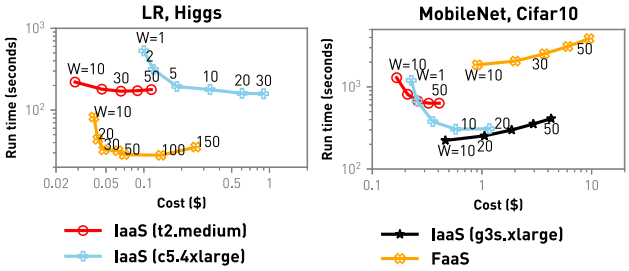


Fig. 11 Tuning number of workers

To study the impact of workers, we use grid search to tune the number of workers for both FaaS and IaaS. Figure 11 illustrates two representative runtime versus cost profiles. Here we choose two types of CPU instances for **LR** and one GPU instance for **MN**. Note that, we do not report the accuracy numbers since they are almost the same. On LR and Higgs, adding workers initially makes both FaaS and IaaS systems faster since this task can take advantage of communication-efficient algorithms. Then, the curve becomes flattened (e.g., FaaS at 100 workers) since the acceleration of computation is compensated by the increase of communication. Different systems plateaued at different runtime levels, illustrating the difference in its start-up time and communication cost. On the other hand, the more workers we add, the more costly the execution is. On MobileNet that cannot take advantage of communication-efficient algorithms, the FaaS system flattens earlier, illustrating the difficulty of scale-up.

6.3 Analytical model

Based on the empirical observations, we now develop an analytical model that captures the cost/performance tradeoff between different configuration points in the design space.

Given an ML task, for which the dataset size is s MB and the model size is m MB, let the start-up time of w FaaS (resp. IaaS) workers be $t^F(w)$ (resp. $t^I(w)$); the *bandwidth* of S3, EBS (Elastic Block Store), network, and ElastiCache be B_{S3}, B_{EBS}, B_n, B_{EC}; the *latency* of S3, EBS, network, and ElastiCache be L_{S3}, L_{EBS}, L_n, L_{EC}. N_{cp} denotes the number of checkpointing in FaaS. Assuming that the algorithm used

by FaaS (resp. IaaS) requires R^F (resp. R^I) epochs to converge with one single worker, we use $f^F(w)$ (resp. $f^I(w)$) to denote the "scaling factor" of convergence which means that using w workers will lead to $f^F(w)$ times more epochs. Let C^F (resp. C^I) be the time that a single worker needs for computation of a single epoch. Table 5 lists symbols in the cost model. With w workers, the execution time of FaaS and IaaS can be modeled as follows (to model the cost in dollar, we can simply multiply the unit cost per second):

$$FaaS(w) := \underbrace{t^F(w) + \frac{s}{B_{S3}}}_{\text{start up \& loading}} + \underbrace{N_{cp}\frac{m}{B_{S3}}}_{\text{checkpoint}} + \overbrace{R^F f^F(w)}^{\text{convergence}} \times \underbrace{\left((3w-2)\left(\frac{m/w}{B_{S3/EC}} + L_{S3/EC}\right) + \overbrace{\frac{C^F}{w}}^{\text{computation}}\right)}_{\text{communication}}$$

$$IaaS(w) := \underbrace{t^I(w) + \frac{s}{B_{S3}}}_{\text{start up \& loading}} + \overbrace{R^I f^I(w)}^{\text{convergence}} \times \underbrace{\left((2w-2)\left(\frac{m/w}{B_n} + L_n\right) + \overbrace{\frac{C^I}{w}}^{\text{computation}}\right)}_{\text{communication}}$$

where the color-coded terms represent the "built-in" advantages of FaaS/IaaS (green means holding advantage)—FaaS incurs smaller start-up overhead, while IaaS incurs smaller communication overhead because of its flexible mechanism and higher bandwidth. Assuming FaaS adopts ScatterReduce communication primitive, each worker writes local model partitions (size of $(w-1)m/w$) to storage, reads responsible partitions (size of $(w-1)m/w$), writes merged partition (size of m/w), and reads other partitions (size of $(w-1)m/w$). Therefore, the term of FaaS's communication cost is $(3w-2)$. The difference in the constant, i.e., $(3w-2)$ and $(2w-2)$, is caused by the fact that FaaS can only communicate via external storage services that do not have a computation capacity. The latency terms $L_{S3/EC}$ and L_n could dominate for smaller messages.

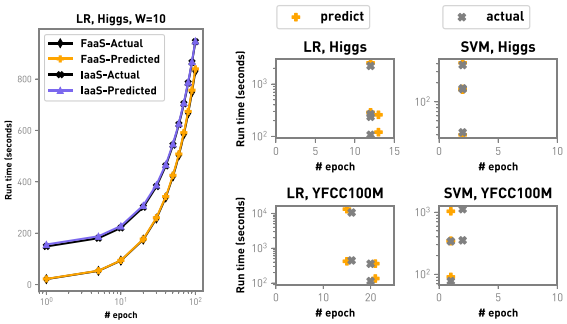
Recall our experimental results, FaaS performs better for ML workloads that are communication efficient and converge quickly. If we look at the analytical model, that means the convergence term $R^F f^F(w)$ is small—the model can converge within a few epochs and communication rounds.

6.3.1 Validation of analytical model

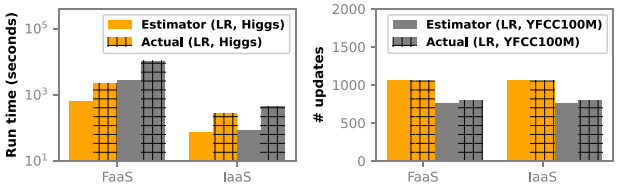
We provide an empirical validation of this analytical model. First, we show that *given the right constant, this model correctly reflects the runtime performance of FaaS-based and IaaS-based systems.* We train a logistic regression model on Higgs with ten workers (the results on other models and datasets are similar) and show the analytical model versus the actual runtime in Fig. 12a. Across a range of fixed number of epochs (from 1 to 100), and using the representative

Table 5 Symbols in the analytical model

Symbol	Definition
s	Size of dataset
m	Size of ML model
b	Batch size
w	Number of workers
$t^F(w)/t^I(w)$	Startup time of w FaaS/IaaS workers
R^F/R^I	Epochs to converge for FaaS/IaaS
$f^F(w)/f^I(w)$	Scaling factor of convergence for FaaS/IaaS
C_F/C_I	Epoch computation time for FaaS/IaaS
B_{S3}	Bandwidth of S3
B_{EBS}	Bandwidth of EBS
B_n	Bandwidth of network
B_{EC}	Bandwidth of ElastiCache
L_{S3}	Latency of S3
L_{EBS}	Latency of EBS
L_n	Latency of network
L_{EC}	Latency of ElastiCache



(a) Analytical Model and Actual Runtime. (b) Predicted Runtime (Sampling-based Estimator plus. Analytical Model) vs. Actual Runtime.



(c) Cost of Estimator. We compare the runtime of the estimator and the actual training, and then compare the number of model updates until convergence.

Fig. 12 Evaluation of analytical model

values for the symbols in Table 5, we see that the analytical model approximates the actual runtime reasonably well.

The goal of our analytical model is to understand the fundamental tradeoff governing the runtime performance, instead of serving as a predictive model. To use it as a predictive model, one has to estimate the number of epochs that each algorithm needs. This problem has been the focus of many previous works (e.g., [34]) and is thus orthogonal to this paper. Nevertheless, we implement the sampling-based estimator in [34], use 10% of training data and 3 epochs to estimate the number of epochs needed for convergence. Then, the estimated epoch number and unit runtime are used to predict the end-to-end runtime. We choose four workloads (LR/SVM & Higgs/YFCC100M) and train them with two optimization algorithms (SGD and ADMM) in both FaaS (LambdaML) and IaaS (distributed PyTorch). Figure 12b shows that this simple estimator can estimate the number of epochs well for both SGD and ADMM, and the analytical model can also estimate the runtime accurately (see the "predict" datapoints). It is interesting future work to develop a full-fledged predictive model for FaaS and IaaS by combining the insights obtained from this paper and [34]. To evaluate the efficiency of the estimator, we also report the runtime of the estimator, as well as the number of model updates on sampled and full data until convergence. We choose two workloads (LR/Higgs and LR/YFCC100M) as examples, and show the results of FaaS and IaaS in Fig. 12c. The estimator takes less time (about 24% and 21%) than running on the full data, since this speculation-based estimator samples a subset and does not need to train the model to convergence. We then use the estimator to predict the number of model updates until convergence, and we observe that the numbers are quite close to the actual execution.

6.4 Insights and case studies

(Summary of Insights) This model provides us a framework to reason about performance. However, to depict the full picture of the tradeoff space we need to substantiate all constants with well-optimized system designs and algorithm choices. With all optimizations we described (Part 1), we have the following insights.

Insight 1. Comparable Algorithms. For many models, the optimal choices of algorithm are the same for FaaS and IaaS. For example, when training GLMs and KMeans, both FaaS and IaaS implementations benefit from communication-efficient algorithms such as ADMM for GLMs. Another example is deep learning—both FaaS and IaaS need to use communication-intensive algorithms (i.e., GA-SGD) because of its convergence behavior. For such cases, we have $R^F f^F(w) = R^I f^I(w)$ in the analytical model, and therefore, the tradeoff is dominated by the relative total cost of communication with respect to the start-up cost.

1. *Case 1.1 Communication-Efficient Algorithms.* For models such as GLMs and KMeans, we see scenarios under which an FaaS implementation can outperform an IaaS implementation. This claim holds for both the wall-clock runtime and the cost in dollar.
2. *Case 1.2 Communication-Intensive Algorithms.* For models such as deep neural networks, we see scenarios under which an FaaS implementation is significantly slower than an IaaS implementation. This holds for both the wall-clock runtime and the cost in dollar.

Insight 2. Incomparable Algorithms. In principle, there are cases where the optimal choices of algorithm for FaaS and IaaS can be different. However, given our current choices of models and algorithms, we did not observe such cases in our experimentation.

(Limitation of Our Empirical Study) Despite the efforts in this empirical study, there inevitably exist limitations considering the diversity of ML workloads. For example, there are other related ML techniques (e.g., data loading and model parallelism) that are not covered in this work; the comparison of cost is not fully scientific, since the pricing is determined by the platform's commercial strategy; we have not chosen all the IaaS instances, and considering other types might change the FaaS/IaaS trade-offs; the experiments that benchmark ML pipelines might favor FaaS because the fine-grained elasticity is advantageous for fluctuations in pipeline workload. In addition, this empirical study does not choose an extremely high-dimensional dataset (e.g., kdd2012) and provide sparse operations, since FaaS only provides limited memory resources and cannot efficiently handle such dataset.

6.4.1 Case studies

We can further use this analytical model to explore alternative configurations that may be leveraged by potential future infrastructures and understand how they would impact the relative performance of FaaS and IaaS. We provide two example case studies in the following.

Q1: What if Lambda-to-VM communication becomes faster (and support GPUs)? As we previously analyzed, the performance of HybridPS is bounded by the communication speed between FaaS and IaaS. How would accelerating the FaaS-IaaS communication change our tradeoff? This is possible in the future by having higher FaaS-to-IaaS bandwidth, faster RPC frameworks, or more CPU resources in FaaS. To simulate this, we assume that bandwidth between FaaS and IaaS can be fully utilized and change the bandwidth to 10GBps in our analytical model. As shown in Fig. 13, the performance of HybridPS could be significantly improved. When training **LR** over **YFCC100M**, HybridPS-10GBps is worse than FaaS since FaaS saves the start-up time of one VM and uses ADMM instead of SGD. When training **MN**

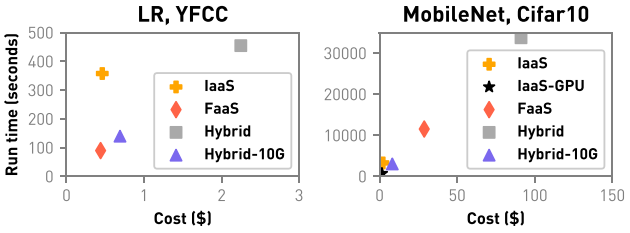


Fig. 13 Simulation: Faster FaaS-IaaS communication

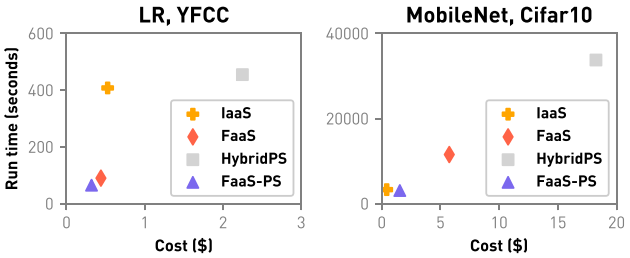


Fig. 14 Simulation: an always-on PS service

over **Cifar10**, HybridPS-10GBps would be about 10% faster than IaaS; however, it is still slower than IaaS-GPU.

If future FaaS further supports GPUs and offers similar pricing compared with comparable IaaS infrastructure—$0.75/hour for g3s.xlarge, HybridPS-10GBps would be 18% cheaper than IaaS. This would make FaaS a promising platform for training deep neural networks; otherwise, under the current pricing model, IaaS is still more cost-efficient even compared with HybridPS-10GBps.

Q2: What if there is an always-on dedicated PS service with smaller start-up time? Current FaaS solutions leverage S3 or ElastiCache to store model parameters. But these two services are not like PS-style, where a centrally stored model replica can be accessed and updated. Therefore, more data transmissions are required during model synchronization. If there is an always-on PS service which is also fast in start-up, training ML model over FaaS can be further accelerated. Figure 14 shows the simulated performance of FaaS with an always-on PS service (FaaS-PS) via assuming the same communication speed as IaaS. FaaS-PS is 1.38× faster than FaaS over YFCC100M and 3.74× faster over Cifar10. This improvement is mainly brought by communication saving during model synchronization.

7 FaaS versus IaaS for multi-tenant workloads

In this section, we consider a more realistic setting other than training a single model and focus on the scenario in which multiple training jobs are executed during a period of time. In reality, one often needs to execute multiple ML training jobs, perhaps on a cloud-based ML service that can be either FaaS or IaaS. It is therefore worthy to understand the performance/cost tradeoffs between FaaS-based and IaaS-based ML services in this broader context. Specifically, we study the following two *multi-tenant* workloads.

Table 6 Models used in multi-tenant workloads

Model	Dataset	# Workers
LR/SVM	Higgs	$W = 10$–50
KM	Higgs	$W = 10$-50
LR/SVM	RCV1	$W = 5$–20
KM	RCV1	$W = 50$–200
LR/SVM	YFCC100M	$W = 100$–500
KM	YFCC100M	$W = 100$–500
MN	Cifar10	$W = 10$–50

- **Training as a Service.** Many machine learning platforms on the cloud, such as Microsoft Azure, Google AI Platform and Alibaba MaxCompute, provide training services to users. One user uploads a dataset to the platform, and the platform returns the trained model to the user.
- **Incremental Training.** For a range of applications, it is infeasible to train all the historical data periodically since these applications generate data continuously. Alternatively, incremental training is adopted to read new data for one loop and update the model accordingly. For example, many e-commerce companies train models incrementally to recommend items to users. The recommendation engine receives new data if one user performs actions such as clicking and purchasing. With newly generated data, the recommendation model is accordingly updated.

7.1 Protocol

(Baselines) We use the same baselines described in Sect. 6.1.1. To simplify the presented results, we choose the best result from all the PyTorch-based IaaS baselines, denoted by PyTorch[8].

(Datasets and ML Models) Table 6 presents the (combination of) datasets and ML models used in our multi-tenant evaluation. The number of workers was tuned to obtain the best performance. The other settings are the same as those presented in Table 3.

[8] Distributed PyTorch with ADMM achieves the best results when training **LR** and **SVM**, distributed PyTorch achieves the best results when training **KM**, and distributed PyTorch with SGD achieves the best results when training **MN**.

(Deployment of IaaS Baselines) While we can simply create one instant FaaS job for each incoming request, we need to consider two deployment scenarios for IaaS-based systems:

- *On-demand IaaS.* Once receiving a training job, we launch the requested number of VMs, run the job, and shutdown the VMs upon job completion.
- *Reserved IaaS.* We reserve the maximum number of VMs requested (by any job) and use them to run all training jobs.

(Principles) We follow the same principle of *Best versus best* in Sect. 6. For each workload, we have two ways to pick the best system configurations:

- *Min-Cost*, in which we pick the configuration that minimizes the cost in dollar.
- *Min-Time*, in which we pick the configuration that minimizes the runtime.

(Experimental Design) Combing deployment scenarios and principles, we obtain four experimental designs—*"On-demand" plus "Min-cost"*, *"On-demand" plus "Min-time"*, *"Reserved" plus "Min-cost"*, and *"Reserved" plus "Min-time"*.

(Related Variables) In the experiments of reserved IaaS, there are two related variables:

- *ReqRate* refers to the *average* throughput of workload, i.e., the average number of training jobs within a unit time (e.g., per hour).
- *MaxConc* refers to the *peak* of the workload, i.e., the maximum number of training jobs that are concurrently executing.

To generate a workload using a certain combination of ReqRate and MaxConc, we assume a period T having $T \times ReqRate$ jobs in total and a bursting peak of $MaxConc$ jobs.

7.2 Training as a service

In the multi-tenant workload of training as a service, the platform receives requests from users and trains ML models until convergence.

7.2.1 "On-demand" plus "min-cost"

For minimizing cost, FaaS-based frameworks are better for ML tasks that are short-running and have minimal communication. For long-running or communication-heavy tasks, IaaS is better.

For minimizing overall cost, Fig. 15 presents the results using on-demand IaaS. We observe that LambdaML is either the most or the second most economical when training linear or clustering models, although the hourly pricing of Lambda functions is higher than comparable VM instances. LambdaML can be more economical as its execution is much faster than the baselines. Interestingly, the hybrid approach is the most expensive one among the participants. This is not surprising, though, as the Lambda functions have to wait for the startup of the EC2 VM (as the parameter server), which significantly increases the cost. On the other hand, when training deep learning models such as **MN**, LambdaML becomes much more expensive (e.g., 5× more than PyTorch), due to both heavier communication and higher pricing.

7.2.2 "On-demand" plus "min-time"

For the goal of minimizing time, the results are similar as minimizing cost. Besides, FaaS-based system achieves better scalability.

As shown in Fig. 16, LambdaML is the fastest when training all the models except **MN**, with a speedup of up to 11×. An interesting observation we have is that we can increase the number of workers for LambdaML if an ML model only needs a few number of communications to converge. However, this is not true for the IaaS-based baselines—it might not be worthwhile to rent more VMs due to the increased startup time of the VMs. When training **MN** on **Cifar10**, however, distributed PyTorch outperforms LambdaML, due to the significant increase in communication cost that slows down FaaS functions. **MN** is optimized by GA-SGD rather than MA-SGD or ADMM, which is communication-intensive. Moreover, compared to linear or clustering models, **MN** requires solving a nonconvex optimization problem and therefore needs more epochs and longer time before convergence, alleviating the start-up saving of FaaS infrastructure.

7.2.3 "Reserved" plus "min-cost"

The relative cost of reserved IaaS and FaaS is affected by the peak and the average rate of jobs. IaaS-based system is cheaper with low degree of peak (maximal concurrency) or high average rate, and is more expensive in the opposite cases.

(MaxConc and ReqRate) By keeping the reserved IaaS instances (VMs) running, one can avoid the problem of slow startup of VMs. However, it raises a new problem regarding the number of VMs to be reserved. One needs to reserve as many VMs as required by the *peak* of the workload (i.e., the *maximal concurrency* of training jobs that are concurrently executing). In our evaluation, we assume that this peak num-

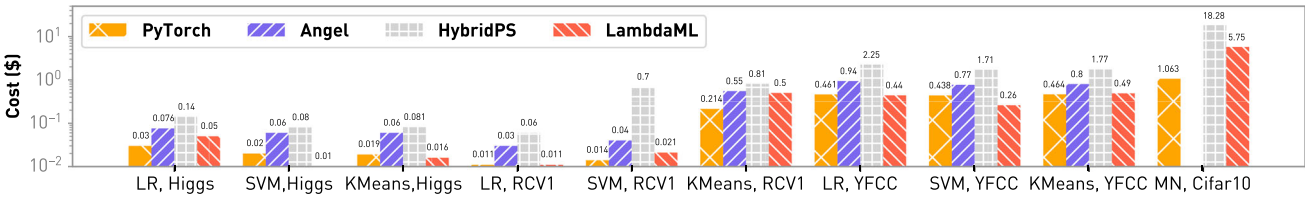


Fig. 15 Training as a service ("on-demand" plus "min-cost"). y-axis is the cost in US dollars

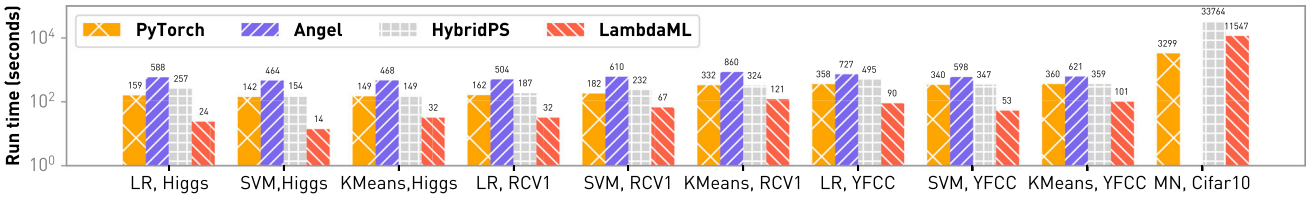


Fig. 16 Training as a service ("on-demand" plus "min-time"). y-axis is the execution time in seconds

ber of VMs is known, although in practice one may have to make an estimation that often results in *overprovisioning*. For example, assume that a training job requires 10 workers, each of which needs 1 vCPU and 3GB memory. We then need in total 50 EC2 VMs, if there are at most 10 concurrent jobs and each EC2 VM is equipped with 2 vCPUs and 8GB. This overprovisioning, however, might make resource in idle not within the peak period. On the other hand, the cost of FaaS functions is sensitive to the *average request rate* of ML training jobs instead of the maximum degree of concurrency, due to its "on demand" and "pay by usage" nature.

(Fix ReqRate) Figure 17 presents the results when we fix the average request rate (ReqRate) and increase the degree of maximal concurrency (MaxConc). [9] We observe that the cost of LambdaML remains the same regardless of the concurrency level. The hybrid approach is more expensive than LambdaML since it needs to keep a powerful VM running as the parameter server. IaaS-based systems, PyTorch and Angel, need to reserve more EC2 instances for the peak concurrency level, leading to linearly increasing costs.

(Fix MaxConc) Figure 18 further presents the results in the other direction, i.e., when we fix the maximum degree of concurrency (MaxConc) as 5 or 10 and increase the average request rate (ReqRate). Now the costs of IaaS-based systems remain the same, while the cost of FaaS-based system increases linearly.

7.2.4 "Reserved" plus "min-time"

When the goal is minimizing time, FaaS is faster than reserved IaaS over short-running linear or clustering jobs, while slower over long-running nonconvex jobs.

Figure 19 presents the results when we switch the optimization goal from minimizing the total cost to minimizing the total execution time. Note that we use a reserved Memcached for LambdaML to maximize the training speed. We observe that LambdaML is faster than PyTorch and Angel in most of the cases. Note that, although we do not need to start VMs for PyTorch and Angel, it still takes time to submit training jobs. The hybrid approach is also slower than LambdaML when training linear models, because it uses GA-SGD, whereas LambdaML uses more efficient ADMM. On the other hand, LambdaML is slower than PyTorch when training **MN** on **Cifar10**, since our implementation using Memcached requires more communication time.

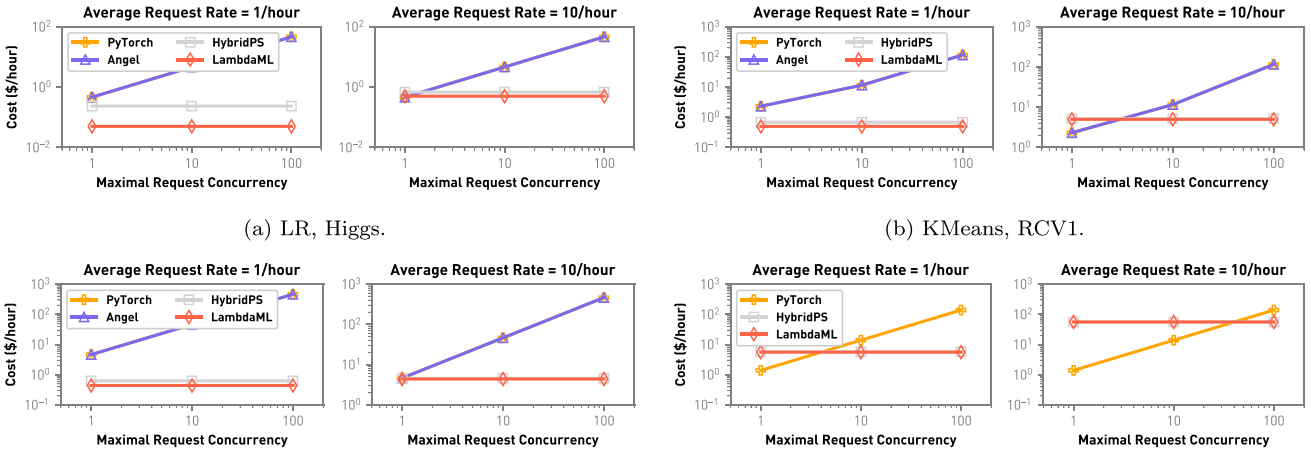
7.3 Incremental training

We study another multi-tenant service, i.e., incremental training, in which an ML system stores an ML model, receives new data from users, and updates the model incrementally. The key difference to the first workload (training as a service) is that incremental training only needs to scan new incoming data for one pass.

7.3.1 "On-demand" plus "min-cost"

Figure 20 illustrates the results of incremental training using on-demand IaaS that aims for minimizing cost. LambdaML spends the least money on all models except KMeans/RCV1 and MobileNet/Cifar10, and the cost saving can be at most 91% over linear models. When training KMeans on RCV1, LambdaML runs 1.63× faster than PyTorch, but the cost of LambdaML is higher because the pricing of LambdaML is 3.8× higher than EC2. LambdaML costs 2.6× more on MobileNet owing to the same reason. This resonates with our

[9] Due to space limitations, we show results of four representative tasks. The observed patterns are the same on the other workloads.



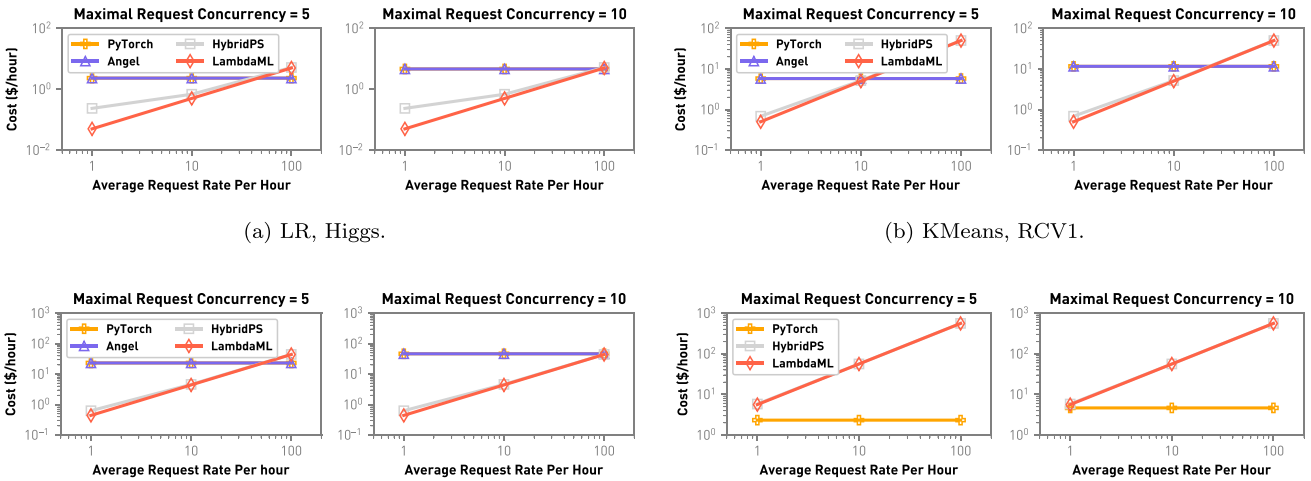
(a) LR, Higgs.

(b) KMeans, RCV1.

(c) LR, YFCC100M.

(d) MobileNet, Cifar10.

Fig. 17 Training as a service ("reserved" plus "min-cost"). We fix ReqRate and increase MaxConc



(a) LR, Higgs.

(b) KMeans, RCV1.

(c) LR, YFCC100M.

(d) MobileNet, Cifar10.

Fig. 18 Training as a service ("reserved" plus "min-cost"). We fix MaxConc and increase ReqRate

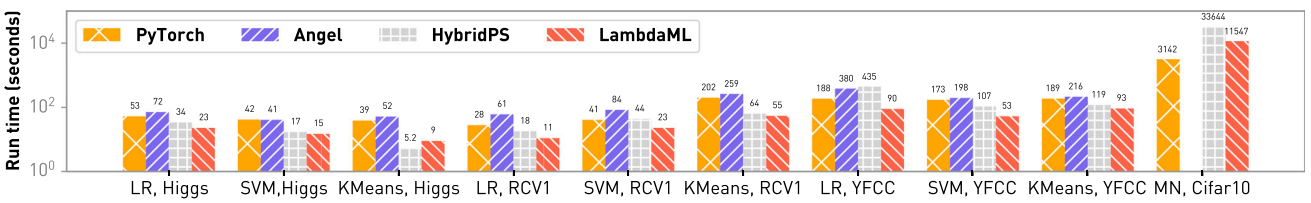


Fig. 19 Training as a service ("reserved" plus "min-time"). y-axis is the execution time in seconds

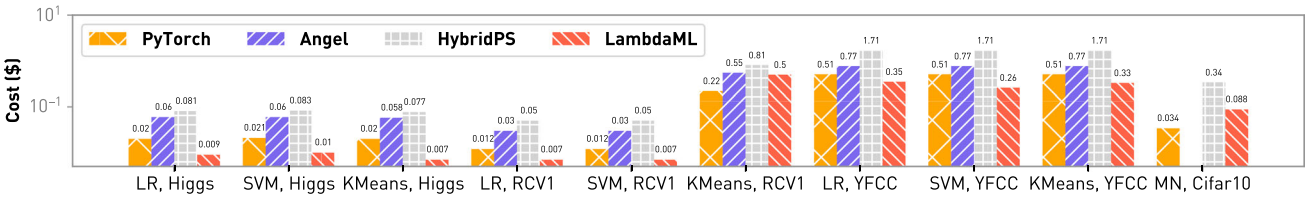


Fig. 20 Incremental training ("on-demand" plus "min-cost"). y-axis is the cost in US dollars

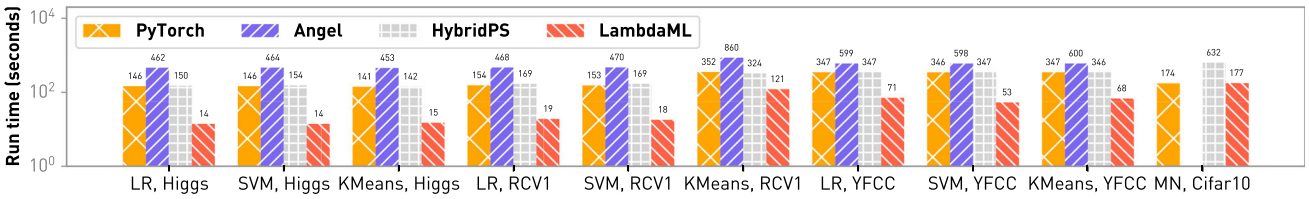


Fig. 21 Incremental training ("on-demand" plus "min-time"). y-axis is the execution time in seconds

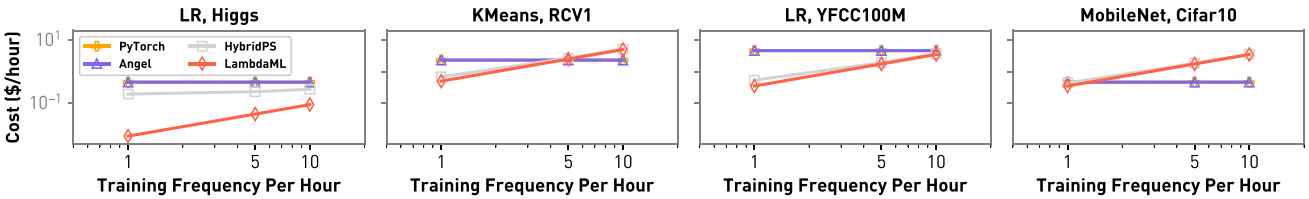


Fig. 22 Incremental training ("reserved" plus "min-cost"). y-axis is the cost in US dollars

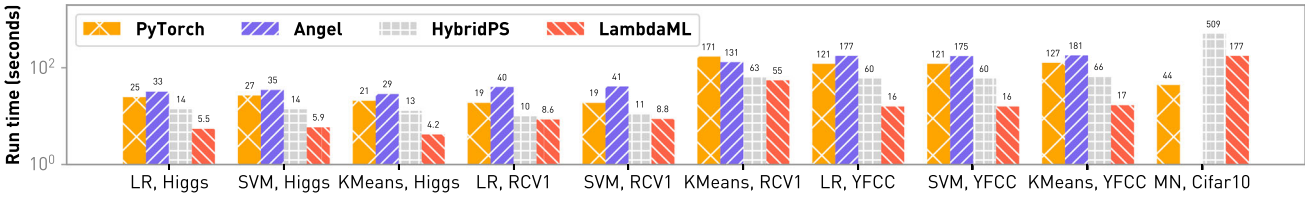


Fig. 23 Incremental training ("reserved" plus "min-time"). y-axis is the execution time in seconds

previous claim, that is, the FaaS-based solution does not fit communication-intensive and long-running algorithms.

7.3.2 "On-demand" plus "min-time"

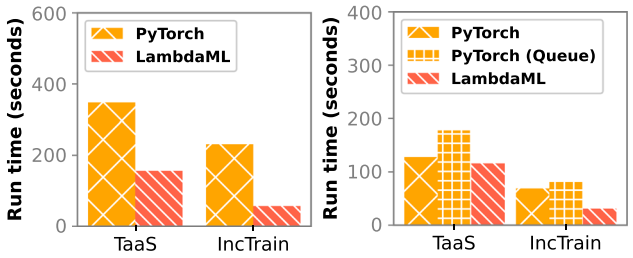
If one user requires minimizing the execution time using on-demand IaaS, LAMBDAML is able to beat all the baselines on all the models, as shown in Fig. 21. Since incremental training scans the input dataset for only one loop, LAMBDAML does not suffer from the communication dilemma of FaaS infrastructure, and the improvement brought by the fast startup of FaaS becomes important. Compared with training as a service, FaaS reveals more benefits for incremental training because the communication cost is much less.

7.3.3 "Reserved" plus 'min-cost"

In this study, we choose reserved IaaS for incremental training and require the minimal cost. The cost of LAMBDAML is affected by the training frequency since FaaS platform charges for every triggering. The results on four representative models are illustrated in Fig. 22. When the hourly training frequency increases from 1 to 10, the cost of each IaaS-based system does not change because their cost is only affected by the number of reserved VMs. The cost of LAMBDAML, however, increases linearly. FaaS-based system is cost-efficient (up to 51× cheaper) over linear models when the training is not frequent. Nevertheless, on communication-intensive or long-running models (MobileNet/Cifar10, KMeans/RCV1), FaaS system is more costly than IaaS system under a high training frequency.

7.3.4 "Reserved" plus "min-time"

Figure 23 shows the results using reserved IaaS under the requirement of minimizing the execution time. Although IaaS-based solutions save the startup time of VMs, they still take some time to submit and initialize the job. Therefore, LAMBDAML is faster than IaaS baselines over linear models and clustering models. Besides, LAMBDAML achieves better scalability than other systems. For example, LAMBDAML achieves the best performance when using 500 workers for LR on YFCC. However, the performance of PyTorch deteriorates when using 500 workers as a result of increased time spent on submitting the tasks. Similar to earlier results, LAMBDAML is slower than PyTorch over MobileNet which is communication intensive. HybridPS utilizes gradient averaging SGD as the optimization method over linear models, therefore it is slower than LAMBDAML which uses ADMM. Overall, LAMBDAML is at most 11× faster than the baselines.



(a) Poisson Arrival Process. (b) Workload Scheduling

Fig. 24 More workload scenarios. TaaS represents training as a service, and IncTrain represents incremental training

7.4 Summary

After evaluating LAMBDAML over multi-tenant workloads, we can draw the following conclusions:

> *For multi-tenant workloads, the performances of training systems are highly affected by the concurrency and frequency of requests. Due to quick startup and strong scalability, FaaS can be more promising than IaaS when the workload is bursty, short-running, and communication-efficient.*

7.5 More workload scenarios

In this section, we evaluate more flexible workload scenarios, instead of fixed workloads in the above study.

(Poisson Arrival Process) Following a previous ML workload trace [50], we assume the job arrival pattern is a Poisson process with $\lambda = 2$. The job type is randomly sampled from Table 6, with the sample rate inversely proportional to the average completion time. The maximal request concurrency and request rate are set to 10 and 10 per hour. Under the setting of "On-demand" plus "Min-time," we compare LambdaML with PyTorch and report the average job runtime in Fig. 24a. We observe that LambdaML outperforms PyTorch on two workloads, verifying that it can adapt to various job distributions.

(Workload Scheduling) For reserved IaaS, it is also possible to preserve fewer resources and queue the incoming jobs. We conduct an experiment that allocates 50% resources for the peak period and uses an FIFO queue to schedule the jobs. We also choose the Poisson arrival process to generate jobs. As shown in Fig. 24b, introducing the job scheduler inevitably increases the job completion time, but can save significant costs meanwhile.

8 Conclusion

We conducted a systematic study regarding the tradeoff between FaaS-based and IaaS-based systems for training ML models. We started by an anatomy of the design space that covers the optimization algorithm, the communication channel, the communication pattern, and the synchronization protocol, which had not yet been explored by previous work. We then implemented LAMBDAML, a prototype system of FaaS-based training on Amazon Lambda, following which we systematically depicted the tradeoff space and identified cases where FaaS holds an edge. Our results indicate that ML training pays off in serverless infrastructures only for models with efficient (i.e., reduced) communication and that quickly converge, or multi-tenant scenarios that are "bursty."

(Future Work) Since serverless computing is a busy area, we will keep studying new techniques proposed by the community. For example, there exists some early work that tries to add networking in serverless infrastructure [67]; we will explore how it would impact the trade-offs in our future work if the serverless platforms include such features.

Acknowledgements This work was sponsored by National Key R&D Program of China (No. 2022ZD0116315), Key R&D Program of Hubei Province (No. 2023BAB077), the Fundamental Research Funds for the Central Universities (No. 2042023kf0219). This work was supported by Ant Group. We gratefully acknowledge the help from Quanqing Xu (xuquanqing.xqq@oceanbase.com) and Chuanhui Yang (rizhao.ych@oceanbase.com) from OceanBase, AntGroup.

References

1. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: SIGMOD, pp. 967–980 (2008)
2. Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: a system for large-scale machine learning. In: OSDI, pp. 265–283 (2016)
3. Akkus, I.E., Chen, R., Rimac, I., et al.: Sand: towards high-performance serverless computing. In: USENIX ATC, pp. 923–935 (2018)
4. Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nat. Commun. **5**(1), 1–9 (2014)
5. Baldini, I., Castro, P., Chang, K., et al.: Serverless computing: current trends and open problems. In: Research Advances in Cloud Computing, pp. 1–20 (2017)
6. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. JMLR **13**(2), 281–305 (2012)
7. Bergstra, J., Yamins, D., Cox, D.D., et al.: Hyperopt: a python library for optimizing the hyperparameters of machine learning algorithms. In: SciPy, vol. 13, p. 20 (2013)
8. Bhattacharjee, A., Barve, Y., Khare, S., Bao, S., Gokhale, A., Damiano, T.: Stratum: a serverless framework for the lifecycle management of machine learning-based data analytics tasks. In: OpML, pp. 59–61 (2019)
9. Boehm, M., Tatikonda, S., Reinwald, B., et al.: Hybrid parallelization strategies for large-scale machine learning in systemML. VLDB **7**(7), 553–564 (2014)
10. Boyd, S., Parikh, N., Chu, E., et al.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Found. Trends Mach. Learn. **3**(1), 1–122 (2011)
11. Cao, W., Zhang, Y., Yang, X., Li, F., Wang, S., Hu, Q., Cheng, X., Chen, Z., Liu, Z., Fang, J., et al.: PolarDB serverless: a cloud

native database for disaggregated data centers. In: SIGMOD, pp. 2477–2489 (2021)
12. Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., Katz, R.: Cirrus: a serverless framework for end-to-end ML workflows. In: SoCC, pp. 13–24 (2019)
13. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Commun. ACM **62**(12), 44–54 (2019)
14. Chaturapruek, S., Duchi, J.C., Ré, C.: Asynchronous stochastic convex optimization: the noise is in the noise and SGD don't care. In: NeurIPS, pp. 1531–1539 (2015)
15. Chen, T., Guestrin, C.: Xgboost: a scalable tree boosting system. In: SIGKDD, pp. 785–794 (2016)
16. Dean, J., Corrado, G., Monga, R., et al.: Large scale distributed deep networks. In: NeurIPS, pp. 1223–1231 (2012)
17. Falkner, S., Klein, A., Hutter, F.: BOHB: robust and efficient hyperparameter optimization at scale. In: ICML, pp. 1437–1446 (2018)
18. Fard, A., Le, A., Larionov, G., Dhillon, W., Bear, C.: Vertica-ML: distributed machine learning in vertica database. In: SIGMOD, pp. 755–768 (2020)
19. Feng, L., Kudva, P., Da Silva, D., Hu, J.: Exploring serverless computing for neural network training. In: CLOUD, pp. 334–341 (2018)
20. Fingler, H., Akshintala, A., Rossbach, C.J.: USETL: unikernels for serverless extract transform and load why should you settle for less? In: APSys, pp. 23–30 (2019)
21. Gropp, W., Gropp, W.D., Lusk, E., Lusk, A.D.F.E.E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, vol. 1 (1999)
22. Gupta, V., Kadhe, S., Courtade, T., Mahoney, M.W., Ramchandran, K.: Oversketched Newton: fast convex optimization for serverless systems. arXiv:1903.08857 (2019)
23. Hellerstein, J.M., Faleiro, J.M., Gonzalez, J., et al.: Serverless computing: one step forward, two steps back. In: CIDR (2019)
24. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: HotCloud (2016)
25. Ho, Q., Cipar, J., Cui, H., et al.: More effective distributed ml via a stale synchronous parallel parameter server. In: NeurIPS, pp. 1223–1231 (2013)
26. Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G.R., Gibbons, P.B., Mutlu, O.: Gaia: geo-distributed machine learning approaching LAN speeds. In: NSDI, pp. 629–647 (2017)
27. Huang, Y., Jin, T., Wu, Y., et al.: FlexPS: flexible parallelism control in parameter server architecture. VLDB **11**(5), 566–579 (2018)
28. Ishakian, V., Muthusamy, V., Slominski, A.: Serving deep learning models in a serverless platform. In: IC2E, pp. 257–262 (2018)
29. Jiang, J., Cui, B., Zhang, C., Fu, F.: DimBoost: boosting gradient boosting decision tree to higher dimensions. In: SIGMOD, pp. 1363–1376 (2018)
30. Jiang, J., Cui, B., Zhang, C., Yu, L.: Heterogeneity-aware distributed parameter servers. In: SIGMOD, pp. 463–478 (2017)
31. Jiang, J., Fu, F., Yang, T., Cui, B.: SketchML: accelerating distributed machine learning with data sketches. In: SIGMOD, pp. 1269–1284 (2018)
32. Jiang, J., Yu, L., Jiang, J., Liu, Y., Cui, B.: Angel: a new large-scale machine learning system. Natl. Sci. Rev. **5**(2), 216–236 (2018)
33. Jonas, E., Schleier-Smith, J., Sreekanti, V., et al.: Cloud programming simplified: a Berkeley view on serverless computing. arXiv:1902.03383 (2019)
34. Kaoudi, Z., Quiané-Ruiz, J.A., Thirumuruganathan, S., Chawla, S., Agrawal, D.: A cost-based optimizer for gradient descent optimization. In: SIGMOD, pp. 977–992 (2017)
35. Kara, K., Eguro, K., Zhang, C., Alonso, G.: ColumnML: column-store machine learning with on-the-fly data transformation. VLDB **12**(4), 348–361 (2018)
36. Klein, A., Falkner, S., Mansur, N., Hutter, F.: RoBO: a flexible and robust Bayesian optimization framework in Python. In: NIPS 2017 Bayesian Optimization Workshop, pp. 4–9 (2017)
37. Klimovic, A., Wang, Y., Kozyrakis, C., Stuedi, P., Pfefferle, J., Trivedi, A.: Understanding ephemeral storage for serverless analytics. In: USENIX ATC, pp. 789–794 (2018)
38. Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., Kozyrakis, C.: Pocket: elastic ephemeral storage for serverless analytics. In: OSDI, pp. 427–444 (2018)
39. Kraska, T., Talwalkar, A., Duchi, J.C., Griffith, R., Franklin, M.J., Jordan, M.I.: MLbase: a distributed machine-learning system. In: CIDR, vol. 1, pp. 2–1 (2013)
40. Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: RCV1: a new benchmark collection for text categorization research. JMLR **5**(4), 361–397 (2004)
41. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: a novel bandit-based approach to hyperparameter optimization. JMLR **18**(1), 6765–6816 (2017)
42. Li, M., Andersen, D.G., Smola, A.J., Yu, K.: Communication efficient distributed machine learning with the parameter server. In: NeurIPS, pp. 19–27 (2014)
43. Li, S., Zhao, Y., Varma, R., et al.: PyTorch distributed: experiences on accelerating data parallel training. VLDB **13**(12), 3005–3018 (2020)
44. Liaw, R., Bhardwaj, R., Dunlap, L., Zou, Y., Gonzalez, J.E., Stoica, I., Tumanov, A.: HyperSched: dynamic resource reallocation for model development on a deadline. In: SoCC, pp. 61–73 (2019)
45. Liu, J., Zhang, C.: Distributed learning systems with first-order methods. Found. Trends Databases **9**, 1–100 (2020)
46. McSherry, F., Isard, M., Murray, D.G.: Scalability! but at what cost? In: HotOS (2015)
47. Meng, X., Bradley, J., Yavuz, B., et al.: MLlib: machine learning in apache spark. JMLR **17**(1), 1235–1241 (2016)
48. Misra, U., Liaw, R., Dunlap, L., Bhardwaj, R., Kandasamy, K., Gonzalez, J.E., Stoica, I., Tumanov, A.: RubberBand: cloud-based hyperparameter tuning. In: EuroSys, pp. 327–342 (2021)
49. Müller, I., Marroquín, R., Alonso, G.: Lambada: interactive data analytics on cold data using serverless cloud infrastructure. In: SIGMOD, pp. 115–130 (2020)
50. Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., Zaharia, M.: Heterogeneity-aware cluster scheduling policies for deep learning workloads. In: OSDI, pp. 481–498 (2020)
51. Ooi, B.C., Tan, K.L., Wang, S., et al.: SINGA: a distributed deep learning platform. In: MM, pp. 685–688 (2015)
52. Paszke, A., Gross, S., Massa, F., et al.: PyTorch: an imperative style, high-performance deep learning library. NeurIPS **32**, 8026–8037 (2019)
53. Perron, M., Castro Fernandez, R., DeWitt, D., Madden, S.: Starling: A scalable query engine on cloud functions. In: SIGMOD, pp. 131–141 (2020)
54. Poppe, O., Guo, Q., Lang, W., Arora, P., Oslake, M., Xu, S., Kalhan, A.: Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless. In: VLDB (2022)
55. Pu, Q., Venkataraman, S., Stoica, I.: Shuffling, fast and slow: scalable analytics on serverless infrastructure. In: NSDI, pp. 193–206 (2019)
56. Rausch, T., Hummer, W., Muthusamy, V., Rashed, A., Dustdar, S.: Towards a serverless platform for edge AI. In: HotEdge (2019)
57. Recht, B., Re, C., Wright, S., Niu, F.: Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In: NeurIPS, pp. 693–701 (2011)
58. Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N.J., Popa, R.A., Gonzalez, J.E., Stoica, I., Patterson, D.A.: What serverless computing is and should become: the next phase of cloud computing. Commun. ACM **64**(5), 76–84 (2021)

59. Shankar, V., Krauth, K., Pu, Q., et al.: Numpywren: serverless linear algebra. arXiv:1810.09679 (2018)
60. Sparks, E.R., Venkataraman, S., Kaftan, T., Franklin, M.J., Recht, B.: KeystoneML: optimizing pipelines for large-scale advanced analytics. In: ICDE, pp. 535–546 (2017)
61. Tandon, R., Lei, Q., Dimakis, A.G., Karampatziakis, N.: Gradient coding: avoiding stragglers in distributed learning. In: ICML, pp. 3368–3376 (2017)
62. Tang, H., Gan, S., Zhang, C., Zhang, T., Liu, J.: Communication compression for decentralized training. In: NeurIPS, pp. 7663–7673 (2018)
63. Tang, H., Lian, X., Yan, M., Zhang, C., Liu, J.: D^2: decentralized training over decentralized data. In: ICML, pp. 4848–4856 (2018)
64. Wang, H., Niu, D., Li, B.: Distributed machine learning with a serverless architecture. In: INFOCOM, pp. 1288–1296 (2019)
65. Wang, J., Joshi, G.: Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. arXiv:1810.08313 (2018)
66. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: USENIX ATC, pp. 133–146 (2018)
67. Wawrzoniak, M., Müller, I., Fraga Barcelos Paulus Bruno, R., Alonso, G.: Boxer: data analytics on network-enabled serverless platforms. In: CIDR (2021)
68. Wu, Y., Dinh, T.T.A., Hu, G., Zhang, M., Chee, Y.M., Ooi, B.C.: Serverless data science-are we there yet? A case study of model serving (2022)
69. Zhang, H., Li, J., Kara, K., Alistarh, D., Liu, J., Zhang, C.: ZipML: training linear models with end-to-end low precision, and a little bit of deep learning. In: ICML, pp. 4035–4043 (2017)
70. Zhang, Z., Jiang, J., Wu, W., Zhang, C., Yu, L., Cui, B.: MLlib*: fast training of GLMs using spark MLlib. In: ICDE, pp. 1778–1789 (2019)
71. Zheng, S., Meng, Q., Wang, T., et al.: Asynchronous stochastic gradient descent with delay compensation. In: ICML, pp. 4120–4129 (2017)
72. Zinkevich, M., Weimer, M., Smola, A.J., Li, L.: Parallelized stochastic gradient descent. In: NeurIPS, pp. 2595–2603 (2010)