Towards Demystifying Serverless Machine Learning Training

Jiawei Jiang*,†, Shaoduo Gan*,†, Yue Liu†, Fanlin Wang†
Gustavo Alonso†, Ana Klimovic†, Ankit Singla‡, Wentao Wu‡, Ce Zhang‡
†Systems Group, ETH Zürich ‡Microsoft Research, Redmond
{jiawei.jiang, sgan, alonso, ana.klimovic, ankit.singla, ce.zhang}@inf.ethz.ch,
{liuyue, fanwang@student.ethz.ch, wentao.wu@microsoft.com

ABSTRACT
The appeal of serverless (FaaS) has triggered a growing interest on how to use it in data-intensive applications such as ETL, query processing, or machine learning (ML). Several systems exist for training large-scale ML models on top of serverless infrastructures (e.g., AWS Lambda) but with inconclusive results in terms of their performance and relative advantage over “serverful” infrastructures (IaaS). In this paper we present a systematic, comparative study of distributed ML training over FaaS and IaaS. We present a design space covering design choices such as optimization algorithms and synchronization protocols, and implement a platform, LambdaML, that enables a fair comparison between FaaS and IaaS. We present experimental results using LambdaML, and further develop an analytic model to capture cost/performance tradeoffs that must be considered when opting for a serverless infrastructure. Our results indicate that ML training pays off in serverless only for models with efficient (i.e., reduced) communication and that quickly converge. In general, FaaS can be much faster but it is never significantly cheaper than IaaS.

CCS CONCEPTS
• Computer systems organization → Cloud computing; • Computing methodologies → Machine learning; Parallel algorithms.

KEYWORDS
Serverless Computing, Machine Learning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD ’21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8345-1/21/06...
https://doi.org/10.1145/3448016.3459240

1 INTRODUCTION
Serverless computing has recently emerged as a new type of computation infrastructure. While initially developed for web microservices and IoT applications, recently researchers have explored the role of serverless computing in data-intensive applications, which stimulates intensive interests in the data management community [5, 44, 58, 82, 95]. Previous work has shown that adopting a serverless infrastructure for certain types of workloads can significantly lower the cost. Example workloads range from ETL [35] to analytical queries over cold data [76, 80]. These data management workloads benefit from serverless computing by taking advantage of the unlimited elasticity, pay per use, and lower start-up and set-up overhead provided by a serverless infrastructure.

Serverless Computing and FaaS. Serverless computing has been offered by major cloud service providers (e.g., AWS Lambda [14], Azure Functions [73], Google Cloud Functions [37]) and is favored by many applications (e.g., event processing, API composition, API aggregation, data flow control, etc. [20]) as it lifts the burden of provisioning and managing cloud computation resources (e.g., with auto-scaling) from application developers. Serverless computing also 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 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.

ML and Data Management. Modern data management systems are increasingly tightly integrated with advanced analytics such as data mining and machine learning (ML). Today, many database systems support a variety of machine learning training and inference tasks [32, 43, 49, 78, 85]. Offering ML functionalities inside a database system reduces data movement across system boundaries and makes it possible for the ML components to take advantage of built-in database mechanisms such as access control and integrity checking. Two important aspects of integrating machine learning into DBMSs are performance and scalability. As a result, the database community has been one of the driving forces behind recent advancement of distributed machine learning [22, 31, 33, 40, 43, 47, 51, 67, 69, 79, 88, 103].

Motivation: FaaS Meets ML Training. Inspired by these two emerging technological trends, in this paper we focus on one of their
intersections by enabling distributed ML training on top of serverless computing. While FaaS is a natural choice for ML inference [48], it is unclear whether FaaS can also be beneficial when it comes to ML training. Indeed, this is a nontrivial problem and there has been active recent research from both the academia and the industry. For example, AWS provides one example of serverless ML training in AWS Lambda using SageMaker and AutoGluon [10]. Such supports are useful when building "training-as-a-service platforms" in which requests of ML model training come in continuously from multiple users or one model is continuously re-trained when new training data arrives, and are also useful when providing users with "code-free" experience of ML training without worrying about managing the underlying infrastructure. Not surprisingly, training ML models using serverless infrastructure has also attracted increasingly intensive attention from the academia [21, 24, 25, 34, 39, 42, 92]. We expect to see even more applications and researches focusing on training ML models using FaaS infrastructures in the near future.

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?

State of the Landscape. Despite of these recent interests, these early explorations depict a "mysterious" picture of the relative performance of IaaS and FaaS for training ML models. Although previous work [25, 39, 42, 92] has illustrated up to two orders of magnitude performance improvements of FaaS over IaaS over a diverse range of workloads, the conclusion remains inconclusive. In many of these early explorations, FaaS and IaaS are often not put onto the same ground for comparison (see Section 6): either the IaaS or FaaS implementations could be further optimized or only micro-benchmarking is conducted. Moreover, similar to other non-ML workloads, we expect a delicate system tradeoff in which FaaS only outperforms IaaS in specific regimes [76, 80, 81]. However, a systematic depiction of this tradeoff space, with an analytical model, is largely lacking for ML training.

Summary of Contributions. In this paper, we conduct an extensive experimental study inspired by the current landscape of FaaS-based distributed ML training. Specifically, we systematically explore both the algorithm choice and system design for both FaaS and IaaS ML training strategies and depict the tradeoff over a diverse range of ML 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 performances of potential configurations used by future systems.

In designing our experimental study, we follow a set of principles for a fair comparison between FaaS and IaaS:

(1) Fair Comparison. To provide an objective benchmark and evaluation of FaaS and IaaS, we stick to the following principled methodology in this empirical study: (1) both FaaS and IaaS implement the same set of algorithms (e.g., SGD and ADMM) to avoid apple-to-orange comparisons such as comparing FaaS running ADMM with IaaS running SGD; and (2) we compare FaaS and IaaS running the most suitable algorithms with the most suitable hyper-parameters such as VM type and number of workers.

(2) 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.

(3) Strong IaaS Competitors and Well-optimized FaaS System. We use state-of-the-art systems as the IaaS solution, which are often much faster than what has been used in previous work showing FaaS is faster than IaaS. We also conduct careful system optimizations and designs for FaaS. The prototype system, LambdaML, can often pick a point in the design space that is faster than what has been chosen by previous FaaS systems.

Summary of Insights. Our study leads to two key insights:

(1) FaaS can be faster than IaaS, but only in a specific regime: when the underlying workload can be made communication efficient, in terms of both convergence and amount of data communicated. On one hand, there exists realistic datasets and models that can take advantage of this benefit; on the other hand, there are also workloads under which FaaS performs significantly worse.

(2) When FaaS is much faster, it is not much cheaper: One insight that holds across all scenarios is that even when FaaS is much faster than IaaS, it usually incurs a comparable cost in dollar. This mirrors the results for other workloads in Lambda [76] and Starling [80], illustrating the impact of FaaS pricing model.

(Paper Organization) We start by an overview of existing distributed ML technologies and FaaS offerings (Section 2). We then turn to an anatomy of the design space of FaaS-based ML systems, following which we implement a Serverless ML platform called LambdaML (Section 3). We present an in-depth evaluation of various design options when implementing LambdaML (Section 4). We further present a systematic study of FaaS-based versus IaaS-based ML systems, both empirically and analytically (Section 5). We summarize related work in Section 6 and conclude in Section 7.

(Reproducibility and Open Source Artifact) LambdaML is publicly available at https://github.com/DS3Lab/LambdaML. All experiments can be reproduced following the instructions at https://github.com/DS3Lab/LambdaML/blob/master/reproducibility.md.

2 PRELIMINARIES

In this section, we present a brief overview of state-of-the-art distributed ML technologies, as well as the current offerings of FaaS (serverless) infrastructures by major cloud service providers.

2.1 Distributed Machine Learning

2.1.1 Data and Model. A training dataset \( D \) consists of \( n \) i.i.d. data examples that are generated by the underlying data distribution \( D \). Let \( D = \{ (x_i, y_i) \in \mathbb{R}^n \times \mathbb{R} \}_{i=1}^{N} \), where \( x_i \) 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 \( w \) that minimizes a loss function \( f \) over the training dataset \( D \): \( \min_w \frac{1}{D} \sum_i f(x_i, y_i, w) \).

2.1.2 Optimization Algorithm. 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., gradients), and update the model. This iterative procedure terminates/converges when there are no more updates to the model. During the training procedure, each pass over the entire data is called an epoch. For instance, mini-batch stochastic gradient descent (SGD) processes one batch of data during each iteration, and thus one epoch involves multiple iterations; on the other hand, k-means processes all data, and thus one epoch, in each iteration.

(Distributed Optimization) When a single machine does not have the computation power or storage capacity (e.g., memory) to host an ML training job, one has to deploy and execute the job across multiple machines. Training ML models in a distributed setting is more complex, 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 [106], distributed ADMM [23]).

2.1.3 Communication Mechanism. One key differentiator in the design and implementation of distributed ML systems is the communication mechanism employed. In the following, we present a brief summary of communication mechanisms leveraged by existing systems, with respect to a simple taxonomy regarding communication channel, communication pattern, and synchronization protocol.

(Communication Channel) The efficiency of data transmission relies on the underlying communication channel. While one can rely on pure message passing between executors, this shared-nothing mechanism may be inefficient in many circumstances. For example, when running SGD in parallel, each executor may have to broadcast its local versions of global states (e.g., gradients, model parameters) to every other executor whenever a synchronization point is reached. As an alternative, one can use certain storage medium, such as a disk-based file system or an in-memory key-value store, to provide a central access point for these shareable global states.

(Communication Pattern) A number of collective communication primitives can be used for data exchange between executors [70], such as Gather, AllReduce, and ScatterReduce.

(Synchronization Protocol) The iterative nature of the optimization algorithms may imply certain dependencies across successive iterations, which force synchronizations between executors at certain boundary points [94]. A synchronization protocol has to be specified regarding when such synchronizations are necessary. Two common protocols used by existing systems are bulk synchronous parallel (BSP) and asynchronous parallel (ASP). BSP is preferred if one requires certain convergence or reproducibility guarantees, where no work can proceed to the next iteration without having all workers finish the current iteration. In contrast, ASP does not enforce such synchronization barriers, but could potentially hurt the empirical convergence rate in some applications.

2.2 FaaS vs. IaaS for ML

Most of the aforementioned distributed ML technologies have only been applied in IaaS environments on cloud, where users have to build a cluster by renting VMs or reserve a cluster with predetermined configuration parameters (e.g., Azure HDInsight [74]). 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 overprovision by reserving more computation resources than actually needed.

The move towards FaaS infrastructures lifts the burden of managing computation resources from users. Resource allocation in FaaS is on-demand and auto-scaled, and users are only charged by their actual resource usages. The downside is that FaaS currently does not support customized scaling and scheduling strategies. Although the merits of FaaS are very attractive, current offerings by major cloud service providers (e.g., AWS Lambda [14], Azure Functions [73], Google Cloud Functions [37]) impose certain limitations and/or constraints that shed some of the values by shifting from IaaS to FaaS infrastructures. Current FaaS infrastructures only support stateless function calls with limited computation resource and duration. For instance, a function call in AWS Lambda can use up to 3GB of memory and must finish within 15 minutes [15]. Such constraints automatically eliminate some simple yet natural ideas on 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. Indeed, state-of-the-art FaaS systems raise lots of new challenges for designing ML systems and leads to a rich design space, as we shall cover in the next section.

3 LAMBDA ML

We implement LambdaML, a prototype FaaS-based ML system built on top of Amazon Lambda, and study the trade-offs in training ML models over serverless infrastructure.

3.1 System Overview

(Challenges) As mentioned in Section 2, we need 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 infrastructures, though 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 certain storage channel to allow the functions to read/write intermediate state information generated during the iterative training procedure. Figure 1 highlights this key difference between IaaS and FaaS designs of ML training systems.

(Framework of LambdaML) Figure 2 shows the framework of LambdaML. When one user specifies the training configurations in AWS web UI (e.g., data location, resources, optimization algorithm,
and hyperparameters), AWS submits job to the serverless infrastructure that allocates resources (i.e., serverless instances) according to the user request. Each running instance is a worker in LambdaML. The training data is partitioned and stored in S3, a distributed storage service in AWS. Each worker maintains a local model copy and uses the library of LambdaML to train a machine learning model.

**Job Execution** A training job in LambdaML has the steps below:

1. **Load data**. Each worker loads the corresponding partition of training data from S3.
2. **Compute statistics**. Each worker creates the ML model with PyTorch and computes statistics for aggregation using the training data and the local model parameters. Different optimization algorithms may choose different statistics for aggregation, e.g., gradient in distributed SGD and local model in distributed model averaging (see Section 3.2.1 for more details).
3. **Send statistics**. In a distributed setting, the statistics are sent to a communication channel (see Section 3.2.2).
4. **Aggregate statistics**. The statistics from all the workers, which are considered as intermediate states, are aggregated using a certain pattern, generating a global state of the statistics (see Section 3.2.3).
5. **Update model**. Each worker gets the merged state of the statistics, with which the local model is updated. For an iterative optimization algorithm, if one worker is allowed to proceed according to a synchronization protocol (see Section 3.2.4), it goes back to step (2) and runs the next iteration.

**Components of LambdaML Library** As shown in Figure 2, the computation library of LambdaML is developed on top of PyTorch, which provides a wide range of ML models and autograd functionalities. To run distributed optimization algorithms, the communication library of LambdaML relies on some communication channel to aggregate local statistics via certain communication pattern and governs the iterative process using a synchronization protocol.

3.2 Implementation of LambdaML

In this section we elaborate the aforementioned four major aspects in the implementation of LambdaML—distributed optimization algorithm, communication channel, communication pattern, and synchronization protocol. Each aspect contains a rich design space which should be studied carefully.

3.2.1 Distributed Optimization Algorithm. In our study, we focus on the following distributed optimization algorithms.

**(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 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 (updated) local models, instead of the gradients, from the executors and do 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. We refer readers to [103] for a more detailed comparison between GA and MA.

**(Distributed ADMM)** Alternating direction method of multipliers (ADMM) is another popular distributed optimization algorithm proposed by Boyd et al. [23]. ADMM breaks a large-scale convex optimization problem into several smaller subproblems, each of which is easier to handle. In the distributed implementation of LambdaML, each executor solves one subproblem (i.e., until convergence of the local solution) and then exchanges local models with other executors to obtain the latest view of the global model. While this paradigm has a similar pattern as model averaging, it has been shown that ADMM can have better convergence guarantees [23].

3.2.2 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. With this storage component, we are able to aggregate data between running instances in the implementation of distributed optimization algorithms. 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 four alternatives—S3, ElastiCache for Redis, ElastiCache for Memcached, and DynamoDB. S3 is a disk-based object storage service [18], whereas Redis and Memcached are in-memory key-value data stores provided by Amazon.
When there are too many executors or a large amount of local updates to be aggregated, the single leader executor by himself. We will refer to it as a hybrid design.

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

(A) FaaS-baased Data Aggregation We now design a communication scheme for data aggregation using a storage service, such as S3 or ElasticCache, as the communication channel. As shown in Figure 3, the entire communication process contains the following steps: 1) Each executor stores its generated intermediate data as a temporary file in S3 or ElasticCache; 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 3 also presents an alternative implementation using a VM-based parameter server as in the hybrid design exemplified by Cirrus [25]. In this implementation, 1) each executor pushes local updates to the parameter server, with which 2) the parameter server further updates the global model. Afterwards, 3) each executor pulls the latest model from the parameter server.

3.2.3 Communication Pattern. To study the impact of communication patterns, we focus on two MPI primitives, AllReduce and ScatterReduce, that have been widely implemented in distributed ML systems [103]. Figure 4 presents the high-level designs of AllReduce and ScatterReduce in an FaaS environment with an external storage service such as S3 or ElasticCache.

(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 ≤ i ≤ n) collects the i^{th} 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.

Synchronization Protocol. We focus on two synchronization protocols that have been adopted by many existing distributed ML systems. One can simply implement these protocols on top of serverful architectures by using standard primitives of message passing interface (MPI), such as MPI_Barrier. Implementations on top of FaaS architectures, however, are not trivial, given that stateless functions cannot directly communicate with each other.

(Synchronous) We design a two-phase synchronous protocol, which includes a merging and an updating phase. We illustrate this in FaaS architecture that leverages an external storage service:

- **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 with next round of training. All executors are synchronized using this two-phase framework. Moreover, one can rely on certain atomicity guarantees provided by the storage service to implement these two phases. Here we present the implementation of our proposed synchronous protocol.

- **Implementation of the 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), filter out uninteresting ones, and then count the number of files that it has aggregated. When the number of aggregated files equals the number of workers, the aggregator can proceed. Otherwise, it should wait and keep polling the storage service until the desired number of files is reported.

- **Implementation of the 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 an executor that is pending on the merged file, it can then keep polling the storage service until the name of the merged file shows up.
(Asynchronous) Following the approach of SIREN [92], 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.

3.3 Other Implementation Details
This section provides the additional implementation details of LambdaML that are relevant for understanding its performance.

3.3.1 Handling Limited Lifetime. One major limitation of Lambda functions is their (short) lifetime, that is, the execution time cannot be longer than 15 minutes. We implement a hierarchical invocation mechanism to schedule their executions, as illustrated in Figure 5. 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, 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. It then resumes execution by triggering its Lambda function with a new worker. 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).

(Limitation) Under the current design, this mechanism cannot support the scenario in which a single iteration takes longer than 15 minutes. We have not observed such a workload in our evaluation, and it would require a very large model and batch size for a single iteration to exceed 15 minutes. Especially given the memory limitation (3GB) of FaaS, we do not expect this to happen in most realistic scenarios. A potential workaround to accommodate this limitation is to use a smaller batch size so that one iteration takes less than 15 minutes. A more fundamental solution might be to explore model-parallelism [61, 77] in the context of FaaS, which is an interesting direction for future research.

4 EVALUATION OF LAMBDA ML
We evaluate LambdaML with the goal of comparing the various design options that have been covered in Section 3. We start by laying out the experiment settings and then report evaluation results with respect to each dimension of the design space.

4.1 Experiment Settings
(Datasets) Figure 6a presents the datasets used in our evaluation. In this section, we focus on smaller datasets to understand the system behavior and leave the larger datasets (YFCC100M and Criteo) to the next section when we conduct the end-to-end study. We focus on Higgs, RCV1 and Cifar10. Higgs is a dataset for binary classification, produced by using Monte Carlo simulations. Higgs contains 11 million instances, and each instance has 28 features.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th># Ins</th>
<th># Feat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cifar10</td>
<td>220 MB</td>
<td>60 M</td>
<td>1 K</td>
</tr>
<tr>
<td>RCV1</td>
<td>1.2 GB</td>
<td>697 K</td>
<td>47 K</td>
</tr>
<tr>
<td>YFCC100M</td>
<td>110 GB</td>
<td>100 M</td>
<td>4 K</td>
</tr>
<tr>
<td>Criteo</td>
<td>30 GB</td>
<td>52 M</td>
<td>1 M</td>
</tr>
</tbody>
</table>

(a) Micro benchmark. (b) End-to-end benchmark.

RCV1 is a two-class classification corpus of manually categorized newswire stories made available by Reuters [62]. The feature of each training instance is a 47236-dimensional sparse vector, in which every value is a normalized TF-IDF value. Cifar10 is an image dataset that consists of 60 thousand 32×32 images categorized by 10 classes, with 6 thousand images per class.

(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. The number of the model parameters is equal to that of input features. 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 × 224 × 3, and the size of model parameters is 12MB. ResNet50 (RN) is a famous neural network model that was the first to introduce identity mapping and shortcut connection. KMeans (KM) is a clustering model for unsupervised problems, trained by expectation maximization (EM).

(Protocols) We randomly shuffle and split the data into a training set (with 90% of the data) and a validation set (with 10% of the data). We report the number for Higgs with batch size 100K, while setting it as 10K or 1M will not change the insights and conclusions; whereas it is 128 for MN and 32 for RN over Cifar10 according to the maximal memory constraint (3GB) of Lambda. We tune the optimal learning rate for each ML model in the range from 0.001 to 1. We set a threshold for the observed loss, and we stop training when the threshold is reached. The threshold is 0.68 for LR on Higgs, 0.2 for MN on Cifar10, and 0.1 for KM on Higgs.

(Metrics) We decouple the system performance into statistical efficiency and system efficiency. We measure statistical efficiency by the loss observed over the validation set. Meanwhile, we measure system efficiency by the execution time of each iteration or epoch.

4.2 Distributed Optimization Algorithms
Carefully choosing the right algorithm goes a long way in optimizing FaaS-based system, and the widely adopted SGD algorithm is not necessarily “one-size-fits-all.”

We implemented GA-SGD (i.e., SGD with gradient averaging), MA-SGD (i.e., SGD with model averaging), and ADMM on top of LambdaML, using ElastiCache for Memcached as the external storage service. Figure 7 presents the results for various data and ML models we tested. We measure the convergence rate in terms of both the wall clock time and the number of rounds for communication.

(Results for 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.

**Results for MN** We have different observations when turning to training neural network models. Figure 7c presents the results of training MN on Cifar10. First, we note that ADMM can only be used for optimizing convex objective functions and 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, though it can reduce the communication cost. On the other hand, GA-SGD can converge steadily and achieve a lower loss. As a result, in this case, we have no choice but to use GA-SGD.

### 4.3 Communication Channels

For many workloads, a pure FaaS architecture can be competitive to the hybrid design with a dedicated VM as parameter server, 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 evaluate the impact of communication channels. We train LR, MN, and KM using LambdaML. LR is optimized by ADMM, MN is optimized by GA-SGD, and KM is optimized by EM. Table 1 presents the settings and compares using disk-based S3 with other memory-based mediums.

**Pure FaaS Solutions** We compare design choices including Memcached, S3, Redis, and DynamoDB.

- **Memcached vs. 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 [17]. 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 (as it is an “always on” service). When we turn to training MN on Cifar10, using Memcached becomes faster than using S3, since it takes much longer for MN to converge.

- **Redis vs. 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 with a big cluster, Redis is inferior to Memcached since Redis lacks a similar high-performance multi-threading mechanism that underlies Memcached.

- **DynamoDB vs. 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 [12], making it infeasible for many median models or large models (e.g., RCV1 and Cifar10).

<table>
<thead>
<tr>
<th>Workload</th>
<th>Memcached vs. S3</th>
<th>DynamoDB vs. S3</th>
<th>VM-P5 vs. S3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cost</td>
<td>slowdown</td>
<td>cost</td>
</tr>
<tr>
<td>LR, Higgs W=10</td>
<td>2.3s / 32s</td>
<td>2.3s / 0.4s</td>
<td>4.8 / 4.01</td>
</tr>
<tr>
<td>LR, Higgs W=50</td>
<td>2.3s / 32s</td>
<td>2.3s / 0.4s</td>
<td>4.8 / 4.01</td>
</tr>
<tr>
<td>KM(n:W=10,k=10)</td>
<td>2.3s / 32s</td>
<td>2.3s / 0.4s</td>
<td>4.8 / 4.01</td>
</tr>
<tr>
<td>KM(n:W=10,k=10)</td>
<td>2.3s / 32s</td>
<td>2.3s / 0.4s</td>
<td>4.8 / 4.01</td>
</tr>
</tbody>
</table>

Table 1: 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. DynamoDB cannot handle a large model such as MobileNet.

<table>
<thead>
<tr>
<th>Lambda Type</th>
<th>EC2 Type</th>
<th>Data Transmission</th>
<th>Model Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda-1GB (1.8vCPU) t2.xlarge</td>
<td>2.6s / 21s</td>
<td>2.9s / 0.5s</td>
<td></td>
</tr>
<tr>
<td>Lambda-1GB (0.6vCPU) t2.xlarge</td>
<td>3.02s / 34s</td>
<td>2.9s / 0.5s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lambda Type</th>
<th>EC2 Type</th>
<th>Data Transmission</th>
<th>Model Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda-3GB (1.8vCPU) t2.xlarge</td>
<td>1.85s / 19s</td>
<td>2.3s / 0.4s</td>
<td></td>
</tr>
<tr>
<td>Lambda-3GB (0.6vCPU) t2.xlarge</td>
<td>2.36s / 32s</td>
<td>2.3s / 0.4s</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Hybrid solution: Communication between Lambda and VM-based parameter server. Transferred data size is 75MB. The time is averaged over ten trials. Transfer time includes time spent on serialization/deserialization. In each pair, the left is result of gRPC and the right is result of Thrift.
When model is larger and workload is more communication-intensive, 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), use different EC2 types, and evaluate the communication between Lambda and EC2. The results in Table 2 reveal 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 70MBps reported by [57, 95]) and EC2-to-EC2 bandwidth (e.g., 10Gbps for c5.4xlarge). Hybrid solution takes at least 1.85 seconds to transfer 75MB. (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 limited CPU resource of Lambda (up to 1.8 vCPU). (3) Model update on 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 maximal 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 perfect fit for FaaS-based deep learning. We will explore this in Section 5.3.1.

We also study the impact of the number of parameter servers. Intuitively, adding parameter servers can increase the bandwidth for model aggregation. However, when we increase the number of parameter servers from 1 to 5 for the hybrid solution, we do not observe significant performance change. As we explained above, the hybrid architecture is not bounded by the bandwidth; instead, the bottleneck is the serialization/deserialization operation in Lambda. Therefore, adding parameter servers cannot solve this problem.

### 4.4 Communication Patterns

We use another model, called ResNet50 (RN), in this study to introduce a larger model than MN. We train LR on Higgs, and train MN and RN on Cifar10, using S3 as the external storage service. Table 3 shows the time spent on communication by AllReduce and ScatterReduce. We observe that using ScatterReduce is slightly slower than AllReduce 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 2X slower than ScatterReduce when training RN, as communication becomes heavy and the single reducer (i.e., aggregator) in AllReduce becomes a bottleneck.

### 4.5 Synchronization Protocols

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 [30, 45, 51]. However, this ASP routine is challenging, if not infeasible, in FaaS infrastructure. We thus follow SIREN [92] to store a global model on S3 and let every FaaS instance rewrite it. This makes the impact of Asynchronous on convergence in our scenario more significant than that of ASP in distributed learning. 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 [104], we use a learning rate decaying with rate $\frac{1}{\sqrt{T}}$ for S-ASP (our Asynchronous implementation) where $T$ denotes the number of epochs. Figure 8 presents the results. We observe that Asynchronous 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. If stragglers exist, those faster executors may read stale model parameters from the stragglers. Consequently, the benefit of system efficiency brought by Asynchronous is offset by its inferior statistical efficiency.

### 5 FaaS VS. IaaS FOR ML TRAINING

We now compare FaaS and IaaS for ML training using LambdaML. Here we focus on the case of training a single model. In this scenario, a user submits a training job over a dataset stored on S3; the system then (1) starts the (FaaS or IaaS) infrastructure and (2) runs the job until it reaches the target loss.

#### 5.1 Experiment Settings

Our experimental protocols are as follows:

- **Competing Systems** We compare LambdaML, a pure FaaS-based implementation, with the following systems:
  - **Distributed PyTorch.** We partition training data and run PyTorch 1.0 in parallel across multiple machines. We use all available CPU cores on each machine, if possible. To manage a PyTorch cluster, we use StarCluster [75], a managing toolkit for EC2 clusters. We use the AllReduce operator of Gloo, a collective communication library, 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 [51]. Angel works on top of the Hadoop ecosystem (e.g., HDFS, Yarn, etc.) and we use Angel 2.4.0 in our evaluation. We chose Angel because it reports state-of-the-art performance on workloads similar to our evaluations.
  - **HybridPS.** Following the hybrid architecture proposed by Cirrus [25], 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.

- **Datasets** In addition to Higgs, RCV1 and Cifar10, Figure 6b presents two more datasets that are used for the current set of performance evaluations, YFCC100M and Criteo. YFCC100M

<table>
<thead>
<tr>
<th>Model &amp; Dataset</th>
<th>Model Size</th>
<th>AllReduce</th>
<th>ScatterReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR,Higgs,W=10</td>
<td>224B</td>
<td>9.2s</td>
<td>9.8s</td>
</tr>
<tr>
<td>LR,Cifar10,W=10</td>
<td>9MB</td>
<td>17.1s</td>
<td>8.5s</td>
</tr>
</tbody>
</table>

Table 3: Impact of different communication patterns.
We illustrate the convergence w.r.t wall-clock time in Figure 9. We first present two micro-benchmarks using the same number of 
workers for FaaS and IaaS, and then discuss end-to-end results.

### (ML Models)
As shown in Table 4, we evaluate different ML models on different datasets, including LR, SVM, KM and MN. We also consider a more complex deep learning model ResNet50. ResNet50 (RN) is a famous neural network model that was the first to introduce identity mapping and shortcut connection.

### (EC2 Instances)
We tune the optimal EC2 instance from the t2 (EC2 Instances) family and the c5 family [16]. To run PyTorch on GPUs, we tune the optimal GPU instances from the g3 family. We use one c5.4xlarge EC2 instance as the parameter server in the hybrid architecture.

### (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 data is uploaded and Memcached is launched (if required). We use one cache.15.small Memcached node whose pricing is $0.034 per hour. Each ADMM round scans the training data ten times. We stop training when the loss is below a threshold, as summarized in Table 4.

#### “COST” Sanity Check
Before we report end-to-end experimental results, we first report a sanity check as in COST [71] to make sure all scaled-up solutions outperform a single-machine solution. Taking Higgs 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 12 instance (PyTorch) would converge in 960 seconds for LR trained by ADMM; while our FaaS (LambdaML) and IaaS (distributed PyTorch) solutions, using 10 workers, converge in 107 and 98 seconds. Similarly, on SVM/KMeans, FaaS and IaaS achieve 9.4/6.2 and 9.7/7.2 speedups using 10 workers; on Cifar10 and MobileNet, FaaS and IaaS achieve 4.8 and 6.7 speedups.

### 5.2 Experimental Results

We first present two micro-benchmarks using the same number of workers for FaaS and IaaS, and then discuss end-to-end results.

#### Algorithm Selection
We first analyze the best algorithm to use for each workload. We first run all competing systems with the minimum number of workers that can hold the dataset in memory. We illustrate the convergence w.r.t wall-clock time in Figure 9.

(1) We first train LR, SVM, and KM over Higgs, RCV1, and 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. To assess the performance of the baselines over high-dimensional features, we train models using the Criteo dataset. LambdaML is still the fastest for LR (and the results on other models are similar) while the speed gap is smaller. This is unsurprising due to the high dimensionality of Criteo.

(2) For MN and RN, as 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. Distributed PyTorch is faster than LambdaML because communication between VMs is faster than using ElasticCache in Lambda. PyTorch-GPU is the fastest as GPU can accelerate the training of deep learning models. The improvement of PyTorch-GPU on ResNet50 is smaller than on MobileNet because ResNet50 brings a heavier communication cost. For RN, we
also increase the batch size to 64, which incurs a larger memory footprint during the back-propagation training. FaaS encounters an out-of-memory error due to hitting the memory limit of 3GB, while PyTorch can finish and achieve similar performance as using a batch size of 32. This demonstrates the limitation of the current FaaS infrastructure when training large models.

Runtime Breakdown. To help understand the difference between FaaS and IaaS, Figure 10 presents a breakdown of time spent on executing 10 epochs, taking LR on Higgs as an example.

1. Start-up. It takes more than 2 minutes to start a 10-node EC2 cluster, including the time spent on starting the VMs and starting the training job. The start-up of VMs also contains mounting shared volumes and configuring secure communication channels between VMs. The launching of a job also requires a master node dispensing scripts to all workers, meaning that it costs more time for a larger cluster. 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 seconds to start.

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 underlying training engine. Angel spends more time on data loading since it needs to load 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. However, if the startup time is excluded, PyTorch outperforms LambdaML since LambdaML spends more time on communication. Especially, PyTorch does not incur start-up cost with reserved IaaS resources.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Run time</th>
<th>Test accuracy</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>FaaS (LR,Higgs,W=10)</td>
<td>233</td>
<td>62.1%</td>
<td>0.31</td>
</tr>
<tr>
<td>IaaS (LR,Higgs,W=10)</td>
<td>1712</td>
<td>80.45%</td>
<td>3.87</td>
</tr>
<tr>
<td>FaaS (MobileNet,Cifar10,W=10)</td>
<td>1350</td>
<td>80.52%</td>
<td>1.74</td>
</tr>
</tbody>
</table>

Table 5: ML Pipeline (time is in seconds and cost is in $). End-to-end Comparison. 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.

For some communication-efficient workloads, FaaS-based implementation can be significantly faster, but not significantly cheaper in dollar; on other workloads, FaaS-based implementation can be significantly slower and more expensive.

Figure 11 illustrates two representative runtime vs. cost profiles. For models that can take advantage of communication-efficient algorithms (e.g., LR, SVM, Kmeans), adding workers initially makes both FaaS and IaaS systems faster, and then flattened (e.g., FaaS at 100 workers). Different systems plateaued at different runtime levels, illustrating the difference in their start-up time and communication cost. On the other hand, the more workers we add, the more costly the execution is. For models that cannot take advantage of communication-efficient algorithms (e.g., MN, RN), the FaaS system flattened earlier, illustrating the hardness of scale-up.

In Figure 12, we plot for different configurations such as learning rate and instance type (GPU instance for MN). In addition to G3 GPU instances (NVIDIA M60), we also consider a G4 GPU instance (NVIDIA T4) for MN. The red points refer to IaaS systems, and the orange points refer to FaaS systems. Note that there are more red points than orange points because we need to tune different instance types for IaaS. For LR and SVM, there is an FaaS configuration that is faster than all IaaS configurations in terms of runtime; however, they are not significantly cheaper, mirroring the result similar to Lambda [76] and Starling [80]. For KMeans, a user minimizing for cost would use IaaS while FaaS implementations are significantly faster if the user optimizes for runtime. For MN, the opposite is true — there exists an IaaS configuration that outperforms all FaaS configurations in both runtime and cost — using T4 GPUs is 8x faster and 9.5x cheaper than the best FaaS execution (15% faster and 30% cheaper than M60).

Pipeline Comparison. A real-world training workload of an ML model is typically a pipeline, consisting of data preprocessing, model training, and hyperparameter tuning. To assess the performance of IaaS- and FaaS-based systems on ML pipelines, we construct a pipeline using Higgs and Cifar10: (1) normalize original features to [-1, 1], and (2) grid-search the learning rate in the range [0.01, 0.1] with an increment of 0.01. We perform preprocessing using a single job with 10 workers, and parallelize hyperparameter tuning using multiple jobs (each job with 10 workers and 10 epochs). For IaaS, we use ten t2.medium instances as workers. For FaaS, we (1) use a serverless job to perform the preprocessing and store the transformed dataset to S3, and then (2) trigger one serverless job for each hyperparameter, using S3 as the communication medium. The other settings are the same as in Section 4.1. As Table 5 shows, we observe similar results as in the end-to-end comparison. FaaS is faster than IaaS for LR, however, is not cheaper. For MobileNet, IaaS is significantly faster and cheaper.
5.3 Analytical Model

Based on the empirical observations, we now develop an analytical model that captures the cost/performance tradeoff between different configurations in the design space covered in Section 3.

Given an ML task, for which the dataset size is \( w \) 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, network, and ElastiCache be \( b_{BS}, b_{EC} \), the latency of S3, EBS, network, and ElastiCache be \( L_{BS}, L_{EC} \), and latency of the communication \( L_n \). 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. 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):

\[
FaaS(w) = t_f(w) + \frac{s}{b_{BS}} + R^f f^f(w) \times \left( (3w-2) \frac{m}{b_{BS}} + L_{BS/EC} \right) + \frac{C^F}{w},
\]

\[
IaaS(w) = t_i(w) + \frac{s}{b_{BS}} + R^i f^i(w) \times \left( (2w-2) \frac{m}{b_{BS}} + L_n \right) + \frac{C^I}{w},
\]

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. The difference in the constant, i.e., \((3w-2)\) and \((2w-2)\), is caused by the fact that FaaS can only communicate via storage services that do not have a computation capacity (Section 3.2.2). The latency term \( L_{BS/EC} \) and \( L_n \) could dominate for smaller messages.

When will FaaS outperform IaaS? From the above analytical model it is clear to see the regime under which FaaS can hold an edge. This regime is defined by two properties: (1) scalability—the FaaS algorithm needs to scale well, i.e., a small \( f^f(w) \) (thus a small \( f^i(w) \) for large \( w \)), such that the overall cost is not dominated by computation; and (2) communication efficiency—the FaaS algorithm needs to converge with a small number of rounds, i.e., a small \( R^f \).

(Validation) 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 vs. the actual runtime in Figure 13(a). Across a range of fixed number of epochs (from 1 to 100), and using the constant in Table 6, 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., [54]) and is thus orthogonal to this paper. Nevertheless, we implement the sampling-based estimator in [54] and use 10% of training data to estimate the number of epochs needed. Then, the estimated epoch numbers and unit runtime, together with the constant in Table 6, 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 13(b) 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. 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 [54].

5.3.1 Case Studies. We can further use this analytical model to explore alternative configurations in future infrastructures.

**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 bandwidth between FaaS and IaaS...
IaaS can be fully utilized and change the bandwidth to 10GBps in our analytical model. As shown in Figure 14, 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 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 the data is hot? Throughout this work, we assumed that both FaaS-based and IaaS-based implementations read data from remote disk such as S3. What if the data is “hot” and has already been stored in the VM? To understand this, suppose that the YFCC100M dataset is stored in a powerful VM (m5a.12xlarge), IaaS-based system (t2.medium), FaaS-based system, and HybridPS all read the dataset from the VM. As shown in Figure 15, IaaS significantly outperforms FaaS and HybridPS, due to the slow communication when FaaS reads hot data. This is consistent with observations by Hellerstein et al. [42, 76, 80] for ML training, and resonates with previous work on non-ML workloads [76, 80].

Q3: What about multi-tenancy? When the system is serving multiple users/jobs, FaaS-based solution can potentially provide a more flexible provisioning strategy: start an FaaS run for each job on-demand. Given the short start-up time of FaaS-based solution, this might provide a benefit over both reserved VM solutions and on-demand VM solutions, especially for “peaky” workloads. This observation has been reported for other non-ML workloads [76] and we expect something similar for ML training. Given the space limitations, we leave this aspect to future work.

6 RELATED WORK

(Distributed ML) Data parallelism is a 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 [2, 29, 30, 45, 50, 63, 84]. Another popular implementation is message passing interface (MPI) [38], e.g., the AllReduce MPI primitive leveraged by MLlib [72], XGBoost [27], PyTorch [64], etc. [60]. We have also used data parallelism to implement LambdaML. Other research topics in distributed ML include compression [6, 7, 52, 53, 93, 96, 97, 101], decentralization [28, 41, 59, 65, 90, 91, 100], synchronization [4, 19, 26, 46, 66, 68, 87, 94, 102], straggler [8, 56, 83, 89, 98, 105], data partition [1, 3, 36, 55, 77], etc.

(Serverless Data Processing) Cloud service providers have introduced their serverless platforms, such as AWS Lambda [14], Google Could Functions [37], and Azure Functions [73]. Quite a few studies have been devoted to leveraging these serverless platforms for large-scale data processing. For example, Locus [81] explores the trade-off of using fast and slow storage mediums when shuffling data under serverless architectures. Numpywren [86] is an elastic linear algebra library on top of a serverless architecture. Lambda [76] designs an efficient invocation approach for TB-scale data analytics. Starling [80] proposes a serverless query execution engine.

(Serverless ML) 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 [21, 48], the focus of recent research effort has been on ML model training. For instance, Cirrus [25] is a serverless framework that supports end-to-end ML workflows. In [34], the authors studied training neural networks using AWS Lambda. SIREN [92] proposes an asynchronous distributed ML framework based on AWS Lambda. Hellerstein et al. [42] show 21× to 127× performance gap, with FaaS lagging behind IaaS because of the overhead of data loading and the limited computation power. 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. On the other hand, Fonseca et al. [25] in their Cirrus system, Gupta et al. [39] in their OVERSKETCHED NEWTON algorithm, and Wang et al. [92] in their SIREN system, depict a more promising picture in which FaaS is 2× to 100× faster than IaaS on a range of workloads. The goal of this work is to provide a systematic, empirical study.

7 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 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. In general, FaaS can be much faster but it is never significantly cheaper than IaaS.

ACKNOWLEDGMENT

We are appreciative to all anonymous reviewers at VLDB’21 and SIGMOD’21, who provide insightful feedback that makes this paper much stronger. We also appreciate the help from our anonymous shepherd and area chair at SIGMOD’21 to their shepherding.

CZ and the D3slab gratefully acknowledge the support from the Swiss National Science Foundation (Project Number 200021 184682), Innosuisse/SNF BRIDGE Discovery (Project Number 40B2 187132), European Union Horizon 2020 Research and Innovation Programme (DAPHINE, 957407), Botnar Research Centre for Child Health, Swiss Data Science Center, Alibaba, Cisco, eBay, Google, Google Focused Research Awards, Oracle Labs, Swisscom, Zurich Insurance, Chinese Scholarship Council, and the Department of Computer Science at ETH Zurich.

