A Case for Serverless Machine Learning

Joao Carreira
UC Berkeley
joao@berkeley.edu

Pedro Fonseca
Purdue University
pfonseca@purdue.edu

Alexey Tumanov
UC Berkeley
atumanov@berkeley.edu

Andrew Zhang
UC Berkeley
andrewmzhang@berkeley.edu

Randy Katz
UC Berkeley
randykatz@berkeley.edu

Abstract

The scale and complexity of ML workflows makes it hard to provision and manage resources—a burden for ML practitioners that hinders both their productivity and effectiveness. Encouragingly, however, serverless computing has recently emerged as a compelling solution to address the general problem of data center resource management. This work analyzes the resource management problem in the specific context of ML workloads and explores a research direction that leverages serverless infrastructures to automate the management of resources for ML workflows. We make a case for a serverless machine learning framework, specializing both for serverless infrastructures and Machine Learning workflows, and argue that either of those in isolation is insufficient.

1 Introduction

ML users are currently faced with several daunting challenges that significantly hinder their productivity and effectiveness. First, users are often required to manually configure numerous *system-level* parameters, such as number of workers/parameter servers, memory allocation, number of CPUs, physical topology, etc. Second, users need to specify numerous *ML-specific* parameters, such as learning rate, learning algorithms, neural network structure, that interact in non-obvious ways with the system-level parameters. Third, ML workflows are increasingly comprised of multiple stages, including (a) pre-processing, (b) training, and (c) hyperparameter search, each of which has different computational requirements that ML users have to account for. The need to account for the heterogeneous resource requirements of each stage is a burden for ML users that also often leads to resource overprovisioning—current ML frameworks are generally specialized for coarse-grained VM-based clusters that do not have the flexibility required for such workloads.

Serverless benefits for ML. The serverless computing model is emerging as a compelling approach to address the data center resource management problem. Serverless computing relies on stateless *lambda functions* that are submitted by developers and automatically scheduled by the cloud infrastructure. Thus, they obviate the need for developers to explicitly configure, deploy, and manage long-term compute units (e.g., VMs).

Related Work. General serverless development frameworks have been proposed, such as PyWren [9], that are specifically designed to address the resource constraints of serverless building blocks. However, we find that such tools face fundamental challenges when employed out-of-the-box for ML tasks. For instance, PyWren [9] uses external storage in a functional fashion for intermediate computation results, adding significant and unrecoverable overhead to fine-grain compute tasks, which in turn causes prohibitive system inefficiency in this context. Other systems used for ML

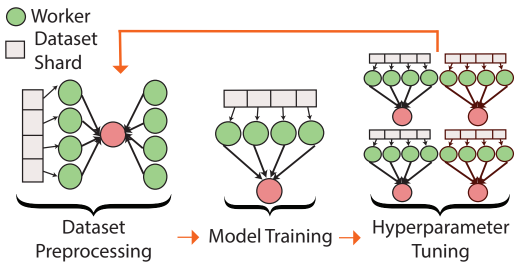
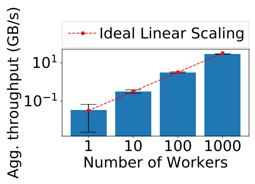
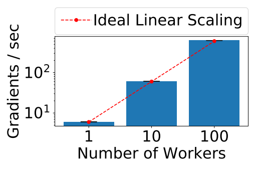


Figure 1: **Typical end-to-end machine learning workflow.** (1) dataset preprocessing typically involves an expensive map/reduce operation on data. It's common to take multiple passes over data, e.g., when normalization is required. (2) model training (parameter server). Workers consume data shards, compute gradients, and synchronize with a parameter server. (3) hyperparameter optimization to tune model and training parameters involves running multiple training instances, each configured with a different set of tuning parameters.

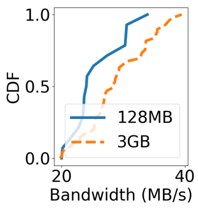


(a) AWS S3

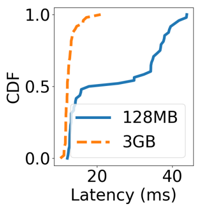


(b) Lambda

Figure 2: This graphs shows the linear scalability achieved by AWS storage (GB/s), and AWS serverless compute (gradients/sec). Each worker consumes 30MB/s of training data.



(a) Bandwidth



(b) Latency

Figure 3: S3 Bandwidth and Latency for the smallest (128MB) and largest (3GB) lambda sizes. Network bandwidth in lambdas is scarce – limited at ≈40MB/s.

workloads, such as Vowpal Wabbit [5] and Multiverso [3], do not tolerate individual worker failure, which is a common scenario in serverless environments, because they rely on MPI [8].

2 Modern ML Workflow Challenges

2.1 End-to-end Machine Learning Workflow

Machine learning users routinely execute a number of different tasks during the process of training models. For instance, a common workflow consists of dataset preprocessing, followed by model training and finally by hyperparameter tuning (Figure 1). These training workflows have been traditionally deployed using systems designed for clusters of virtual machines (VMs) [14, 4, 7, 5, 13]. However, such traditional designs create two important challenges for users.

Explicit resource management. The established approach of *exposing low-level VM resources*, such as storage and CPUs, puts a significant burden on ML developers who are faced with the challenge of provisioning, configuring, and managing these resources for each of their ML workloads.

Over-provisioning. The heterogeneity of the different ML workflow tasks leads to a significant resource imbalance during the execution of a training workflow. The rigidity of VM-based clusters and the ML frameworks specialized for these environments causes developers to *over-provision* resources for peak consumption, which leads to significant waste of datacenter resources [12]. This problem is exacerbated by the fact that, in practice, developers go back and forth between different stages of the workflow to experiment with different parameters.

2.2 Serverless Computing

Serverless computing is a promising approach to address the resource provisioning challenges that ML users face. In particular, it simplifies deployment, given its intuitive interface, and provides mechanisms for avoiding over-provisioning, given its fine-grain serverless functions that can run with as few as 128MB of memory (fine spatial granularity) and time out within a few minutes (fine temporal granularity). Importantly, its ability to efficiently scale, regarding both computation and storage, to hundreds of concurrent workers (Figure 2) also contributes to make it an attractive environment for distributed ML workloads. However, despite the advantages of serverless computing in this context, serverless design principles are at odds with a number of design principles of *existing* ML frameworks. We identify some limitations of existing serverless frameworks and the impact they have for machine learning systems in this section.

Small local memory and storage. Lambda functions, by design have very limited amount of memory and local disk. For instance, AWS lambdas can only access at most 3GB of local RAM and 512MB of local disk. For instance, we have not been able to run Tensorflow [4] or Spark [14] on AWS lambdas nor, in fact, on VMs with such resource-constrained configurations.

Low Network bandwidth and no P2P communication. Lambda functions have little available bandwidth when compared with a regular VM (Figure 3). We find that the largest AWS Lambda can only sustain 40MB/s of bandwidth, in contrast with $>$ 1GB/s in the case of medium-sized VMs. Similarly, common communication topologies used for datacenter ML, such as tree-structured or ring-structured AllReduce [11], are automatically impossible to implement efficiently in such environments!

Short-lived and unpredictable launch times. Lambda functions are short-lived and their launch times are highly variable – AWS lambdas can take up to 2 minutes to start. This means that during training, lambdas start at unpredictable times and can finish in the middle of training. This requires ML runtimes for lambdas to tolerate the frequent departure and arrival of workers.

Lack of Fast Shared Storage. Because lambda functions cannot connect between themselves, shared storage needs to be used. To achieve good performance, this shared storage needs to be low-latency, high-throughput and optimized for the type of communications in ML workloads. However, as of today there is no fast serverless storage for the cloud that provides all these properties.

3 Leveraging Serverless Environments

The constraints imposed by serverless infrastructures have implications on the design of serverless machine learning frameworks for end-to-end workflows. In this section we discuss the design goals and one potential architecture for such systems.

A framework for serverless machine learning needs to meet three critical goals. First, its API needs to support a wide range of ML tasks: dataset preprocessing, training, and hyperparameter optimization. In order to ease the transition from existing ML systems, such API should be developed in a high-level language such as Python. Second, to provide storage for intermediate data and message passing between stateless workers, it needs to provide a low-latency scalable datastore with a rich interface. Third, to efficiently run on resource-constrained lambdas, its worker runtime needs to be light-weight and high-performance.

Next we describe what we believe are the necessary building blocks of a framework that together can successfully meet these goals.

Python frontend. Provides an API for all stages of the ML workflow that is practical and easy to use by the broader ML community. First, the API is totally contained within a Python package which allows ML developers to transition easily. Second, the API provides a high-level interface that abstracts the underlying system-level resources. For instance, developers can run experiments with thousands of concurrent workers without having to provision any of those resources. Last, the Python package provides a user interface through which developers can visualize the progress of their work.

```
s3_input = "criteo_dataset"
s3_output = "criteo_norm"

load_libsvm(
        path,
        s3_input)

normalize(
        s3_input,
        s3_output,
        MIN_MAX_SCALING)
```

(a) Pre-process

```
params = {
 'n_workers': 5, 'n_ps': 1
,
 'worker_size': 1024,
 'dataset': s3_output,
 'epsilon': 0.0001,
 'timeout': 20 * 60,
 'model_size': 2**19,
}

lr_task =
  LogisticRegression(params)
result = lr_task.run()
```

(b) Train

```
# learning rates
lrates = np.arange(0.1, 10, 0.1)
minibatch_size = [100, 1000]

gs = GridSearch(
 task=LRegression,
 param_base=basic_params,
 hyper_vars=["learning_rate",
             "minibach_size"],
 hyper_params=[lrates,
               minibatch_sizes])
results = gs.run()
```

(c) Tune

Figure 4: **API example.** An API for serverless ML should support different phases of ML development workflow: (a) preprocessing, (b) training, and (c) hyperparameter tuning.

Stateful client-side backend The Python frontend provides an interface to the client backend. This backend is responsible for managing transient compute resources and scheduling tasks. Within this backend, different submodules encode the logic for each particular stage of the ML workflow (e.g., preprocessing). These submodules launch the workers on lambdas, keep track of the progress of computations and return the results to the Python frontend once computations complete. The client backend makes use of an internal low-level scheduler that encapsulates all the logic related to launching, killing and regenerating tasks that run on serverless lambdas. This scheduler also keeps track of the state of all the tasks.

Worker runtime Provides a light-weight runtime that encapsulates all the functions that are shared between the different computations the system supports. This simplifies the development of new algorithms. The worker runtime provides two interfaces. First, it provides a smart iterator for training datasets stored in S3. This iterator prefetches and buffers minibatches in the lambda's local memory in parallel with the worker's computations to mitigate the high-latency ($>$10ms) of accessing S3. Second, it provides an API for the distributed data store.

Distributed data store Provides a shared store with a rich interface for intermediate data and communication between workers. This interface has two types of APIs: (1) key-value store for general message passing, intermediate data storage, and data reduction, and (2) parameter server interface. To achieve the required low-latency this datastore is deployed on cloud VMs. To efficiently utilize the scarce network resources, the datastore interface should implement optimizations such as: (1) data compression, (2) sparse datastructures, and (3) asynchronous communication.

4 Usage Model Requirements

To achieve the goal of simplifying the execution of machine learning workflows, an ideal system should provide a simple, but general enough, API. This API needs to let users perform a wide-range of ML tasks, such as: (1) dataset loading, with support for commonly used data formats, (2) dataset preprocessing, (3) model training, and (4) hyperparameter tuning at scale, from within a single, integrated framework.

We demonstrate the capabilities of the API we envision for this system with a toy example – the example in Figure 4 consists of developing an efficient model for the prediction of the probability of a user clicking an ad for a dataset of display ads – based on the Criteo Kaggle competition [1].

The first step in the workflow is to load the dataset and upload it to S3. For instance, the user can call the *load_libsvm* method to load a dataset stored in the LIBSVM [6] format. Behind the scenes parses the data, automatically creates partitions for it and then uploads it to S3.

Once the data is loaded into S3 it can be immediately preprocessed. The system should provide a number of common preprocessing methods used by developers. For instance, the user can normalize the dataset by calling the *normalize* function with the path of the dataset in S3. Once the data is loaded, the user can train different models and see how they perform by looking at the test loss

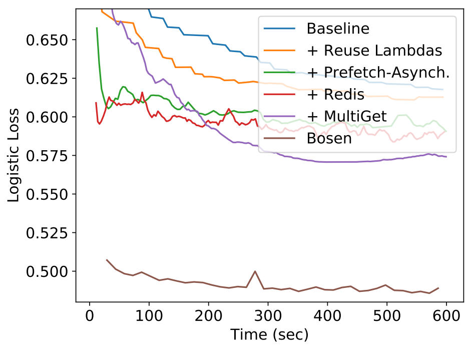


Figure 5: Performance of PyWren and Bosen on the Criteo Kaggle Logistic Regression workload. The PyWren baseline is incrementally improved by reusing lambdas, adding prefetching, switching to asynchronous computation, replacing S3 with a higher performance Redis storage backend, and supporting getting multiple keys on a single RPC.

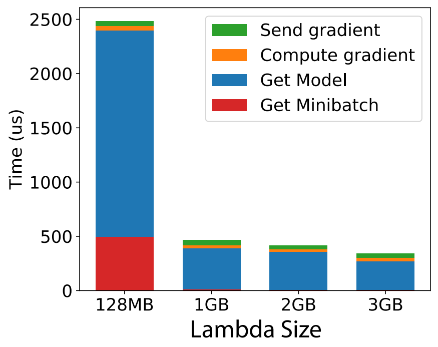


Figure 6: Time per SGD iteration of our preliminary prototype. We breakdown this time by the four main steps: (1) send gradient to data store, (2) compute gradient, (3) get model from datastore and (4) get minibatch from S3. Prefetching of minibatches completely masks the S3 delay when getting minibatches for larger lambdas.

measured by the system. Once the user achieves a reasonable loss with a particular model, they can fine tune it through hyperparameter search.

We envision such a system to allow a number of interactions by the user during the execution of each stage. For instance, when a hyperparameter search task is running, the user should have the ability to monitor the test loss of each individual experiment. For experiments that are not performing well (e.g., test loss is diverging) the user should have the ability to terminate them. This feature can be implemented with a user interface in an interactive environment such as a Jupyter notebook.

5 Experimental evaluation

In this section we provide insights into the performance of PyWren, a system that specializes along a single dimension (serverless architectures). We show that specialization for both serverless and machine learning is required to achieve good performance.

In addition, we implement and evaluate a prototype of a serverless machine learning framework that follows the design goals we outlined in Section §3. We show that mechanisms that prefetch from the remote store and communicate with the datastore, when efficiently implemented for these architectures, are sufficient to match the performance of VM-based ML frameworks.

5.1 The need for a Serverless ML framework

To evaluate the need for a system specialized both for ML and serverless, we compare the performance of two systems, each specialized along a different dimension: PyWren [9] and Bosen [13]. PyWren is a map-reduce framework specialized for serverless architectures. PyWren provides *map* and *reduce* primitives that scale to thousands of workers. Bosen is a distributed parameter-framework specialized for VM-based ML algorithms.

To perform this evaluation we implemented an asynchronous SGD training algorithm for Logistic Regression on PyWren. On top of our PyWren baseline implementation we also implemented a set of optimizations. We use Bosen's out-of-the-box implementation of Logistic Regression with the best performing learning rate. We used the Criteo dataset of display ads [1] from the Criteo Kaggle competition for this benchmark. We ran PyWren on 10 of the largest AWS lambdas (3GB of RAM) and we ran Bosen on 8 cores within a single VM (m5.2xlarge[1]).

[1]AWS m5.large instances have 2 CPUs, 8GB of RAM, 10Gbps networks.

We measured the performance of both systems by recording the test loss over time (Figure 5). For PyWren, we report this value after implementing each optimization. We implemented the following optimizations cumulatively: (1) reusing lambdas across iterations, (2) minibatch prefetching with asynchronous SGD, (3) using a low-latency store (Redis) instead of S3, and (4) using sparse data transfers with multi-get operations.

We observe that these optimizations significantly improve the final test loss achieved by PyWren after 600 seconds (from 0.61 to 0.57). Despite this improvement, PyWren is still significantly slower than Bosen. Further performance analysis shows that PyWren suffers from several overheads, such as serializing/deserializing data, and using remote stores with interfaces that are a poor fit for ML workloads (e.g., Redis or S3). This result suggests the need for carefully considering the performance requirements for ML workloads early on in the design of serverless compute frameworks.

5.2 Towards a Serverless ML Framework

We also built a prototype of the system we outlined in §3. This prototype includes: (1) a high-performance data store with parameter-server interface, (2) ring-buffer prefetching of minibatch data, (3) logistic regression SGD training algorithm.

To understand the benefits of this design, we evaluated it on the same Logistic Regression workload. We measured the average per-worker SGD iteration time (see Figure 6). This time is an indication of the performance of the workers; lower iteration times mean more frequent model updates and faster convergence. We also break down this time into the four main steps of our SGD algorithm: (1) getting the latest model from the data store, (2) getting a minibatch from remote storage (e.g., S3), (3) computing the SGD gradient, and (4) sending the gradient to the datastore.

We find that our prototype, despite the overheads inherent to serverless computing, achieves low time per iteration ($\approx 500\mu$s) – on par with a system like Bosen. This performance stems from two mechanisms: (1) efficient prefetching and buffering of remote minibatches, and (2) communicating with the datastore whenever possible. First, the minibatch prefetching mechanism is effective in masking the time it takes to get a minibatch from S3 by doing it in parallel with the computations. In fact, for medium/large-sized lambdas it takes a negligible amount of time to start computing on a new minibatch, since most of the times this data is cached in memory before the worker needs it. This happens even though getting a single minibatch from S3 takes $\approx$ 10ms. Second, we see that communication with the datastore is efficient (e.g., time to send gradient is negligible). This positive results stems from carefully communicating asynchronously with the datastore.

6 Discussion and Conclusion

We have proposed a general architecture for serverless ML frameworks that overcomes some of the most immediate challenges that serverless architectures impose on ML frameworks. However, a few missing pieces are still required to support a broader set of ML computation patterns and workloads.

No support for GPU workloads Existing serverless architectures do not support GPUs. This prevents a quickly growing class of ML workloads (e.g., deep learning training) to reap the benefits of serverless computing. Support for these workloads can be challenging due to the need to multiplex GPUs at finer time granularities, and the need to provide faster network performance to enable efficient sharing of large parameters. Broadening the users of serverless to these workloads will require rethinking the software stacks that manage these increasingly faster resources such as networks, GPUs and hardware accelerators.

No fast shared storage service The lack of a fast shared storage means every different serverless framework needs to develop its own fast storage system – which is costly – or use existing systems such as Crail [2] or Pocket [10] that don't easily support all the use cases of end-to-end ML workflows. For instance, we have identified the need for a serverless, elastic datastore with a rich interface to support very different use cases such as: data reduce, message passing and parameter sharing. These use cases stretch the design of existing fast storage systems and will push us to go beyond existing fast storage systems such as Redis. It is unclear how the design of such storage systems will look like in the future. Nonetheless, there is no doubt that a fast serverless storage is critical to a vast number of serverless applications, even beyond the domain of machine learning. For this reason, we expect cloud providers to offer such a service in the future.

References

[1] Display Advertising Challenge . `https://www.kaggle.com/c/criteo-display-ad-challenge`.

[2] Apache Crail. `https://crail.apache.org/`.

[3] Multiverso. `https://github.com/Microsoft/Multiverso`.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning.

[5] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[6] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[8] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[9] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017.

[10] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, 2018. USENIX Association.

[11] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[12] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[13] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394. ACM, 2015.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.