Distributed Machine Learning with a Serverless Architecture

Hao Wang[1], Di Niu[2] and Baochun Li[1]
[1]*University of Toronto*, {haowang, bli}@ece.utoronto.ca [2]*University of Alberta*, dniu@ualberta.ca

***Abstract*—The need to scale up machine learning, in the presence of a rapid growth of data both in volume and in variety, has sparked broad interests to develop distributed machine learning systems, typically based on parameter servers. However, since these systems are based on a dedicated cluster of physical or virtual machines, they have posed non-trivial cluster management overhead to machine learning practitioners and data scientists. In addition, there exists an inherent mismatch between the dynamically varying resource demands during a model training job and the inflexible resource provisioning model of current cluster-based systems.**

In this paper, we propose SIREN, an asynchronous distributed machine learning framework based on the emerging serverless architecture, with which stateless functions can be executed in the cloud without the complexity of building and maintaining virtual machine infrastructures. With SIREN, we are able to achieve a higher level of parallelism and elasticity by using a swarm of stateless functions, each working on a different batch of data, while greatly reducing system configuration overhead. Furthermore, we propose a scheduler based on Deep Reinforcement Learning to dynamically control the number and memory size of the stateless functions that should be used in each training epoch. The scheduler learns from the training process itself, in pursuit for the minimum possible training time given a cost. With our real-world prototype implementation on AWS Lambda, extensive experimental results have shown that SIREN can reduce model training time by up to 44%, as compared to traditional machine learning training benchmarks on AWS EC2 at the same cost.

I. INTRODUCTION

It has been widely acknowledged that machine learning (ML) has become fundamentally important in a wide range of research areas, including computer vision, speech recognition, and natural language processing. There has been an imperative need to improve the performance when training machine learning models, especially in the presence of larger volumes of data and increasingly complex models.

To fulfill such a need, a wide range of distributed machine learning frameworks based on the parameter server architecture have been proposed, such as MXNet and TensorFlow. In these frameworks, multiple workers, each operating on a different batch of data, continuously read and update the model parameters using gradient-like algorithms. These parameters are persistently stored and asynchronously updated on a number of parameter servers.

Existing distributed machine learning (ML) frameworks heavily rely on the parallelism achieved by a dedicated cluster of physical servers or virtualized instances. One needs to specify the quantity and types of these servers to be employed for each machine learning training job. Yet, for a general ML practitioner or data scientist, provisioning a physical or virtualized computing platform is a lengthy and error-prone process. For example, it is challenging to choose the optimal configuration from numerous combinations of instance types and settings [1] on AWS EC2, where over 100 instance types are available. In addition, maintaining the launched cluster is costly [2]—users have to pay for every second that the cluster is alive, even if it may largely remain idle during the trial-and-error process of model training.

Following the footsteps of traditional virtual machines and Docker containers, *serverless* (or Function-as-a-Service) architectures, represented by AWS Lambda and Google Cloud Functions, have emerged as a burgeoning computation model to further reduce costs and improve manageability in cloud computing. With serverless architectures, developers do not need to manage dedicated servers or instances for their applications. Instead, they only need to define a set of stateless functions with access to a common data store. Due to its lightweight nature, ease of management, and ability to rapidly scale up, serverless computation has become the trend of building next-generation web services and applications [3].

In this paper, we present the design and implementation of SIREN, a new framework for distributed machine learning based solely on serverless architectures. Our key insights are motivated by the following facts.

First, most current ML training jobs are data parallel, in the sense that a large number of training samples need to be processed by different workers in parallel. Combined with cloud storage, data parallelism aligns with the serverless architecture naturally, in which we can launch a number of stateless functions, each accessing a different batch of data, without managing and maintaining any servers.

Second, the amount of resources invested into a ML training job should be dynamically adjusted as the job progresses, in the best interest of improving the model quality [4]. Not only does the model quality improve at a variable rate during training, but the dependency of model quality improvement on the amount of resources invested is also non-linear and complex. For most models and algorithms, the resource demand decreases as the number of training iterations increases and the target loss stabilizes. Therefore, we need a mechanism to adjust resource provisioning dynamically throughout the model training process, without launching or shutting down server instances (not even Docker containers). With the serverless architecture, we can easily scale resources up or down by controlling the number of functions launched at any point.

Finally, ML training is inevitably a trial-and-error process, which frequently leaves a traditional physical or virtual cluster under-utilized, *e.g.*, a user may need to stop her job, tune hyper-parameters [5] and rerun it. With serverless computation, a user only needs to pay for the execution time of his/her functions, rather than the time that virtual machines remain active.

SIREN is designed to achieve a high level of parallelism and elasticity by using a flexible swarm of serverless functions — called *Lambda functions* in the context of AWS Lambda — in each epoch of model training, where an epoch is defined as one complete presentation of the entire dataset to the training process. We propose a new *hybrid synchronous parallel* (HSP) mode of operation in SIREN: Lambda functions asynchronously push the updated model parameters to the common cloud storage, while a barrier is imposed at the end of each training epoch to allow optimally tuned scheduling decisions.

A highlight of our contribution is a new scheduler based on deep reinforcement learning, designed to dynamically adjust the number of Lambda functions and their memory allocated in each epoch in order to minimize the potential training time under a given cost. With deep reinforcement learning, our scheduler in SIREN learns the best way to trade off between model quality and cost from its previous experience of training similar models, without requiring a large amount of data to kick-start.

We have implemented SIREN based on AWS Lambda and have evaluated SIREN on training various ML models. A comparison with training ML models on AWS EC2 clusters has shown that ML training with SIREN on AWS Lambda has reduced the completion time of a ML training job by up to 44% for the same quality of models at the same cost.

II. BACKGROUND AND MOTIVATION

In this section, we provide a brief background of distributed machine learning systems, and show benefits of serverless (Function-as-a-Service) architectures over parameter server architectures on the Infrastructure-as-a-Service cloud. We then demonstrate convincing evidence that the resource demand in a typical ML training job varies over time dynamically.

A. Distributed ML Training and Parameter Servers

A ML model (such as deep neural networks, support vector machines, and decision trees) with parameters w maps certain input features to an output, which could be a label in a classification problem or a continuous value in a prediction problem. The output is referred to as the *label* throughout this paper. Suppose we have a set of M training samples, where the ith training sample consists of an input feature vector x_i and its label y_i. The ML training process is to adjust the set of model parameters w to

$$\text{minimize}_w \; F(w) := \sum_{i=1}^{M} \ell(w; x_i, y_i) \tag{1}$$

where the loss function $\ell(w; x_i, y_i)$ represents the gap between the predicted label and true label for the ith training sample. The traditional centralized gradient descent algorithm solves this problem by iteratively performing the update:

$$w_{k+1} := w_k - \eta \sum_{i=1}^{M} \nabla \ell(w_k; x_i, y_i),$$

where the model updates in the kth synchronized iteration only happens after the gradients $\nabla \ell(w_k; x_i, y_i)$ have been computed for all M training samples. The algorithm can also be parallelized by partitioning the entire training dataset among multiple workers, each computing the gradients $\nabla \ell(w_k; x_i, y_i)$ only for its local samples, leading to a bulk synchronous parallel (BSP) implementation [6].

The parameter server architecture [7, 8] is widely adopted to perform distributed ML training. One fundamental idea of current parameter server systems is to replace Bulk Synchronous Parallel (BSP) with *Stale Synchronous Parallel* (SSP) [9] or Bounded Delay Synchronization of MXNet, allowing the progress of straggler workers to be out of sync with other workers to a certain extent.

With a popular stochastic gradient descent (SGD) algorithm [10] using *mini-batches*, each worker in a parameter server cluster asynchronously computes an error gradient based on a mini-batch of local data, and pushes this gradient to the parameter server to update the model. Note that gradients in mini-batches from different workers may arrive at the parameter server in an arbitrary order. From a global perspective, let the set $\Xi_k \subset \{1, \ldots, M\}$ represent the kth mini-batch whose gradient $\nabla \ell(w_{k-\tau_{k,i}}; x_i, y_i)$ reaches the server from *any* worker in the cluster. Then, the SGD algorithm using mini-batches performs the following model update for the kth mini-batch at the parameter server:

$$w_{k+1} := w_k - \eta \sum_{i \in \Xi_k} \nabla \ell(w_{k-\tau_{k,i}}; x_i, y_i),$$

where $\tau_{k,i}$ denotes the *staleness* of the model, $w_{k-\tau_{k,i}}$, used to compute gradient $\nabla \ell(w_{k-\tau_{k,i}}; x_i, y_i)$ for sample i. The *staleness* of the model is caused by the fact that when a worker has pulled the model from the parameter server and is computing its local gradient based on that, the model on the server has already been updated by many other workers. It is theoretically shown that both SGD [9] (with a single sample in each mini-batch) and SGD with mini-batches [10] can converge for a wide variety of convex and non-convex loss functions under certain assumptions, including bounded staleness.

B. Serverless v.s. IaaS Architectures

With a simple example, we now evaluate the cost of running ML training jobs in the serverless (Function-as-a-Service) architecture, as compared to running them in the Infrastructure-as-a-Service (IaaS) architecture. IaaS provides a convenient abstraction of *virtual machines* by multiplexing them over the underlying physical machines. In contrast, the serverless architecture provides the abstraction of *stateless*

	Loss Value	Time (s)	Cost ($)
20 functions	0.009725	237.40	0.019
8-core EC2	0.009779	307.87	0.029
150 functions	0.009699	50.04	0.031
`X functions`	0.009761	202.55	0.019

TABLE I: Loss values, time and cost of different resource configurations, after the ML training job converges.

functions over the programming language runtime. Unlike IaaS, there are only two steps to run an application on the serverless architecture: submit an application to the serverless cloud platform, and then invoke the application as stateless functions with necessary input data. The pricing structure in serverless cloud platforms, *e.g.*, AWS Lambda, is based on the number of functions invoked and the time they have been executing. Existing work [11] has shown that running web applications on a serverless cloud platform is more cost effective than using an IaaS cloud service, such as AWS EC2.

To compare both training times and monetary cost, we trained a logistic regression model using (1) SGD on AWS Lambda; and (2) an AWS EC2 instance. The training dataset is a 100,000×2 matrix stored in AWS S3 storage. With AWS Lambda, functions were configured with 256 MB memory and each function call incurred a cost of `$0.00000417` per second. We invoked 20 concurrent stateless functions, each acting as a worker in the SGD algorithm described previously. Each function (worker) directly reads and updates the model stored in Amazon S3. With EC2, we chose a `c5.2xlarge` instance with 8 CPU cores and 16 GB memory, which costs `$0.34` per hour, equivalent to `$0.0000919` per second. On this instance, we ran MXNet, a typical parameter server framework, which automatically parallelizes the workload across 8 cores with SGD. The model was stored and processed in memory. The training is terminated when the change of loss value is below a certain threshold.

As shown in Table I and Fig. 1, the serverless cloud (with 20 functions) reduced the training time needed to reach the same loss by 22.9%. In the meantime, even though only one instance was used in EC2, the serverless cloud incurred a cost that is 34.5% lower. It is worth noting that in this comparison, we excluded the instance setup time in EC2, while in reality, users do have to pay for this portion of time, including launching the instance and building a software environment on it. This example has demonstrated that a substantial amount of cost savings can be achieved by replacing dedicated IaaS cloud clusters with a serverless architecture.

C. Resource Provisioning Policies

Now that we have provided a convincing argument for using a serverless architecture, it is still a major challenge to find the optimal number of functions, as well as their memory size configurations, to be used for each ML training job. Needless to say, ML training is a complex non-deterministic process in nature, and it is hard to obtain *a priori* knowledge on a job's runtime characteristics.

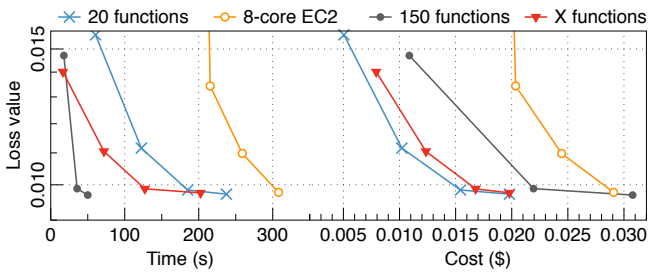


Fig. 1: The time and cost of training a logistic regression model on AWS Lambda and EC2. Each point indicates a training epoch.

Furthermore, it is generally believed that the resources required by a ML training job change across iterations [4], *e.g.*, a smaller amount of resources is needed in later phases as the model quality tends to stabilize. Yet, state-of-the-art frameworks only use a fixed amount of resources (*e.g.*, a fixed number of machines or cores) throughout a training job, primarily because they rely on a dedicated cluster of instances that have already been launched. With the elasticity offered by the serverless architecture, our search space for optimal resource provisioning also extends to dynamic policies.

To illustrate how the training time and cost vary with different amounts of resources provisioned, we further trained the same logistic regression model on AWS Lambda with two additional configurations: (1) using 150 functions throughout the job, and (2) `X functions`, using 120 functions only in the first epoch, 20 functions in intermediate epochs, and 10 functions in the last epoch, where an *epoch* is defined as one complete presentation of the training dataset to the SGD algorithm.

From Table I and Fig. 1, we can see that using 150 functions is faster to converge, yet costs much more. As Fig. 1 suggests, the dynamic resource provisioning scheme, `X functions`, not only strikes a reasonable balance between training time and cost, but also effectively reduces the job completion time, as compared to the static scheme of using 20 functions, at the same cost of `$0.019`.

However, it is a significant challenge to determine the best function configuration policy in the serverless architecture, especially when both static and dynamic policies are possible. A highlight of this work is a new scheduler based on deep reinforcement learning (DRL), with the objective of determining the optimal number of functions and their memory configurations that should be employed in each epoch of the job, based only on online observations of the ongoing job's runtime environment.

III. SIREN: ARCHITECTURAL OVERVIEW

We show an architectural overview of SIREN in Fig. 2. SIREN consists of a local client that makes resource scheduling decisions using a deep reinforcement learning (DRL) agent, and a serverless cloud platform (such as Amazon Lambda) that launches stateless functions for the ML training job based on these scheduling decisions.

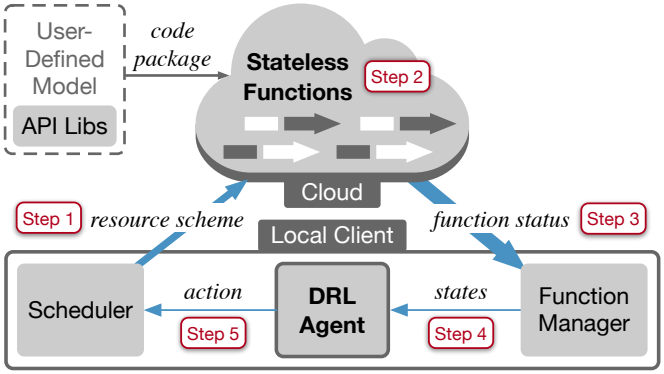


Fig. 2: The system architecture and workflow of SIREN.

First, a code package must be deployed to the serverless cloud platform, containing the user-defined ML model and the libraries that it depends upon. Then, a swarm of stateless functions are launched according to the initial resource scheme (*i.e.*, the number of functions and their memory sizes) to perform SGD-based training for the first epoch. At the end of the first epoch, function status and statistics about the job will be collected and fed as states into the DRL agent at the local client, which will take actions and make resource scheduling decisions for the next epoch. SIREN adaptively adjusts resource scheduling decisions as epochs progress in a training job: different numbers of functions with different memory configurations may be launched in different epochs.

A. SGD on Stateless Functions

SIREN adopts an SGD algorithm using mini-batches and runs over a number of Lambda functions, each of which acts like a *worker* in a traditional parameter server architecture. A major difference between SIREN and parameter server architectures is that no parameter server exists to handle model parameter updates in SIREN. Instead, both the data and model are stored in a common data store (*e.g.*, Amazon S3), accessible by all the functions. Each function reads the current model from the common storage, calculates a gradient based on a mini-batch of training data, and then directly updates the model in the common storage with the newly computed gradient. Thus, the entire architecture is *serverless*.

In SIREN, we propose a *hybrid synchronous parallel* (HSP) computing mode, as shown in Fig. 3. Within each epoch, all the functions can update the model asynchronously (or more precisely, in a SSP fashion [9]), while there is a synchronization barrier imposed at the end of each epoch to allow intelligent resource scheduling for the next epoch.

With HSP, in epoch t, the kth mini-batch $\Xi_{t,k}$ (sorted by the times that their gradients are computed) will lead to the following model update:

$$w_{t,k+1} := w_{t,k} - \eta \sum_{i \in \Xi_{t,k}} \nabla \ell(w_{t,k-\tau_{t,k,i}}; x_i, y_i), \qquad (2)$$

which is executed by any function that becomes available. In addition, $w_{t,0}$ is equal to the model w at the end of epoch $t-1$. HSP is efficient in a serverless architecture

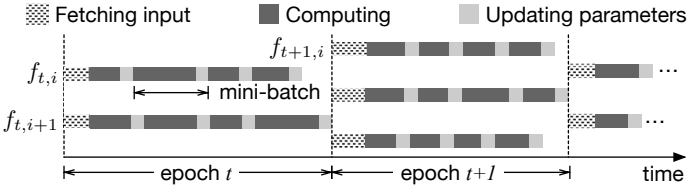


Fig. 3: The hybrid synchronous parallel (HSP) processing on the serverless cloud.

because the launched functions are homogeneous, leading to a low synchronization overhead in each epoch. Invoking and terminating functions are also lightweight with a serverless cloud platform.

B. Implementation on AWS Lambda

We have implemented SIREN with 4K+ lines of Python code, supporting ML model training on top of AWS Lambda, and with full support of MXNet APIs. Machine learning practitioners can run their legacy MXNet projects on SIREN without refactoring their existing code. Our real-world implementation consists of three parts in Fig. 2: (1) the code package encapsulating MXNet machine learning libraries; (2) the local client built with AWS SDK `boto3`, which invokes and manages stateless functions in AWS Lambda; and (3) a DRL agent implemented with TensorFlow to make dynamic resource configuration decisions, which we shall describe in Sec. IV in detail. A series of constraints are enforced by AWS Lambda to keep stateless functions lightweight and portable. Our implementation of SIREN is designed to overcome these constraints.

Since the programming runtime of AWS Lambda does not support ML training algorithms natively, we have introduced a part of MXNet ML libraries into the code package. On AWS Lambda, the maximum size of the code package is limited to 250 MB, which makes it infeasible to directly load any off-the-shelf ML libraries (*e.g.*, MXNet, TensorFlow) onto AWS Lambda. To shrink the size of the MXNet code package, we recompiled the MXNet source code with different combinations of compilation options, and excluded the unnecessary compilation options for the serverless cloud. For example, we disabled the options `USE_CUDA`, `USE_CUDNN` and `USE_OPENMP`, since here we focus on achieving parallelism through a large number of Lambda functions instead of through multiple cores.

The computing capacity of a single function is also restricted on AWS Lambda: each Lambda function is allowed a maximum of 300 seconds[1] to finish execution, and the maximum memory size is 3 GB. However, since AWS Lambda supports as many as 3,000 functions executing concurrently in each AWS account, SIREN uses a high level of parallelism by parallelizing the ML training workload using a large number of Lambda functions.

[1]AWS Lambda now has extended the maximum function timeout to 900 seconds.

IV. Deep Reinforcement Learning for Dynamic Resource Provisioning

In this section, we present our deep reinforcement learning (DRL) technique for dynamic resource provisioning in SIREN. Reinforcement learning (RL) [12] is an experience-driven approach where an agent learns how to behave in a dynamic environment by interacting with it and receiving rewards for performing actions. Deep reinforcement learning further leverages deep neural networks to solve reinforcement learning problems. The agent observes various noisy signals from a dynamic environment, which are called *states*, and feeds these states to a deep neural network (DNN), which generates an action. The agent will take the action in the environment and receive a reward, which is in turn used to update the parameters in the DNN to make better decisions. DRL works in a closed loop to improve decision making with the goal of maximizing a total reward, which we will describe in detail.

The key difference between RL and supervised learning is that in RL, the current action of an agent will affect the state of the environment it will see next, and thus affecting the decision on the next action it should take. In contrast, in supervised learning, the current decision, either in a batch setting or in an online setting, does not affect what the agent sees in the future. Due to this unique characteristic, DRL has recently achieved enormous success in many areas that require sequential decision making and interaction with the environment, *e.g.*, in games such as AlphaZero, and in the placement of computational workloads in TensorFlow on a mixture of heterogeneous devices [13]. DRL naturally fits the resource allocation problem in SIREN— the resource allocated in one epoch will fundamentally change the job progress, cost incurred and many other factors to be described. These factors will in turn affect the resource provisioning decisions for future epochs.

A. Problem Formulation

We consider a ML training workload on a dataset of M samples, with a monetary budget B. The training is terminated if a certain loss value $\mathcal{L}$ is reached or the budget is used. At any epoch t, the scheduler will make decisions about the number of functions, denote by n_t, to be invoked in parallel as well as the memory size m_t of each function.

Let $f_{t,i}$, $i = 1, \ldots, n_t$, represent the ith (alive) function launched for epoch t, as shown in Fig. 3. Note that if function i has reached its lifetime, *e.g.*, in AWS Lambda, a new function will be invoked to replace it and is still denoted by $f_{t,i}$ so that there will always be n_t functions concurrently being executed in epoch t. Each function $f_{t,i}$ repeatedly computes an aggregated gradient for a new mini-batch of data and updates the model parameters according to the SGD in the HSP mode (Eqn. (2)).

In epoch t, suppose the function $f_{t,i}$ spends a total time period of $P^F_{t,i}$ in fetching mini-batches of data, $P^C_{t,i}$ in computing gradients and $P^U_{t,i}$ in updating model parameters. The total execution time of function i in epoch t is $P_{t,i} = P^F_{t,i} + P^C_{t,i} + P^U_{t,i}$. Thus, the total duration of epoch t is given by $P_t = \max_i P_{t,i}$ in HSP. At the end of epoch t, the loss value of the ML job is updated to ℓ_t.

Symbol	Meaning
c	the price of function execution for every GB-second
M	the input data size of the training workload
B	The monetary budget for the training workload
$\mathcal{L}$	The convergence threshold of of the training loss
t	the epoch index of the training workload
T	the index of the final training epoch, $t \in [1, T]$
n_t	the number of concurrent functions in epoch t
m_t	the memory size of each function in epoch t
$f_{t,i}$	the function i in the t-th epoch, $i \in [1, n_t]$
$P^F_{t,i}$	the time period for function $f_{t,i}$ fetching input data
$P^C_{t,i}$	the time period for function $f_{t,i}$ computing gradients
$P^U_{t,i}$	the time period for function $f_{t,i}$ updating parameters
P_t	the whole time period of the epoch t
ℓ_t	the loss value achieved at the end of the epoch t
b_t	the remaining budget at epoch t
r_t	the rewards obtained at the end of the epoch t
u_t	the average memory utilization observed in epoch t
w_t	the average CPU utilization observed in epoch t

TABLE II: Notations used in the paper.

Serverless cloud charges its users based on the function execution time and function memory sizes. Let c represent the unit price of executing a function with 1 GB of memory for one second. Thus, the cost incurred in epoch t is $\sum_{i=1}^{n_t} cm_t P_{t,i}$. And the total monetary cost of the ML job is given by $\sum_{t=1}^{T}\sum_{i=1}^{n_t} cm_t P_{t,i}$, where T is the total number of epochs it takes for the loss value of the ML job to reach a threshold $\mathcal{L}$.

The objective is to minimize the job completion time $\sum_t^T P_t$, subject to a certain budget B on the monetary cost, which is to solve the following optimization problem:

$$\begin{aligned} \min_{\{n_t\},\{m_t\}} \quad & \sum_{t=1}^{T} P_t \\ & T \geqslant 1,\ \ell_T \leqslant \mathcal{L} \\ & \sum_{t=1}^{T}\sum_{i=1}^{n_t} cm_t P_{t,i} \leqslant B \end{aligned} \tag{3}$$

This problem is a challenging sequential decision problem, which can hardly be solved by existing dynamic programming algorithms, due to the absence of a deterministic mapping from the current status and actions, *e.g.*, ℓ_t, n_t, and m_t, to the completion time P_t of epoch t. We propose to solve Eqn. (3) using deep reinforcement learning, where a DRL agent can learn to apply optimized strategies under different states of the ML job from its experience of executing similar jobs.

B. The DRL Agent

At the beginning of each epoch t, the DRL agent in SIREN decides the resource provisioning scheme (n_t, m_t), which are called *actions* for epoch t, based on the current *states* of

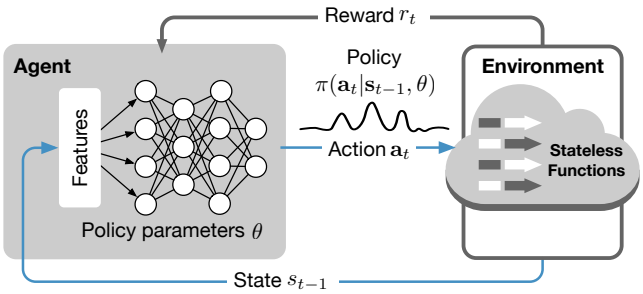


Fig. 4: DRL with policy represented by a DNN.

the environment, as shown in Fig. 4. The effectiveness of the chosen action (n_t, m_t) is then quantified by a numerical *reward* observed by the end of epoch t, depending on the duration P_t of this epoch as well as whether the budget is overrun or the job is completed. We define the states, actions, and the reward in our DRL agent in detail in the following.

State: Let the state of epoch t be represented by a vector $\mathbf{s}_t = (t, \ell_t, P_t, P_t^F, P_t^C, P_t^U, u_t, w_t, b_t)$, in which ℓ_t is the achieved loss value at the end of epoch t. Here t and ℓ_t help the DRL agent learn how the loss values of the ML job evolve. P_t^F, P_t^C, P_t^U are the *average* data fetching time, average computing time and average model parameter updating time, respectively, while P_t is the epoch completion time. These time measurements reflect the training speed of SIREN under the previous resource provisioning decision. Furthermore, u_t and w_t are the average memory utilization and average CPU utilization in the serverless cloud in epoch t, while b_t is the remaining budget at the end of epoch t. These utilization metrics help to indicate whether the resource provisioned in epoch t was excessive or insufficient.

Action: An action $\mathbf{a}_t = (n_t, m_t)$ will be chosen at the beginning of epoch t, where $n_t, m_t \in \mathbb{Z}^+$. Recall that n_t is the number of functions to be invoked and m_t is the memory size of each function. The DRL agent chooses actions based on a *policy*, which is by definition a probability distribution $\pi(\mathbf{a}|\mathbf{s})$ over the entire action space given the current state. We will apply the policy gradient methods to approximate the policy $\pi(\mathbf{a}|\mathbf{s})$ by a function with parameters θ. Thus, the policy π can be written as $\pi(\mathbf{a}|\mathbf{s}, \theta)$, where θ are the parameters to be learned.

Note that our action space is a discrete space including as many as $n_t \times m_t$ possible choices. For example, AWS Lambda supports up to 3,000 concurrent functions and 46 levels of memory sizes ranging from 128 MB to 3,008 MB[2]. Therefore, there are 138,000 possible actions in AWS Lambda. Training a DRL agent with such a large action space would be costly. To make the reinforcement learning efficient, we define the policy π as a Gaussian probability density over a real-valued space, *i.e.*,

$$\pi(\mathbf{a}|\mathbf{s}, \theta) = \frac{1}{\sigma(\mathbf{s}, \theta)\sqrt{2\pi}} \exp\Big(- \frac{(\mathbf{a} - \mu(\mathbf{s}, \theta)^2}{2\sigma(\mathbf{s}, \theta)^2}\Big), \quad (4)$$

[2]https://docs.aws.amazon.com/lambda/latest/dg/limits.html

and choose an action $\mathbf{a}_t$ based on the conditional probability $\pi(\mathbf{a}_t|\mathbf{s}_{t-1}, \theta)$. Then, learning the probability mass function over a large discrete action space is converted to finding the parameters $(\mu(\mathbf{s}, \theta), \sigma(\mathbf{s}, \theta))$ in a 2-D continuous space.

Reward: We set the reward observed at the end of each epoch t as $r_t = -\beta P_t$, $t = 1, \ldots, T-1$, where β is a coefficient that regularizes the reward. The longer the epoch t takes, the less reward the agent will receive. Recall that T is the epoch after which the ML job stops. In other words, at epoch T, either we have $\ell_T \leqslant \mathcal{L}$ or the budget B is used up ($b_T \leqslant 0$). The reward in the final epoch T is defined as

$$r_T = \begin{cases} -\beta P_T + C & \text{if } \ell_T \leqslant \mathcal{L} \text{ and } b_T \geqslant 0, \\ -\beta P_T - C & \text{otherwise.} \end{cases}$$

In other words, if the job stops with success, *i.e.*, the convergence threshold $\mathcal{L}$ is met without overrunning the budget B, a positive C will be awarded to the agent. Otherwise, if the job fails, *i.e.*, it has not converged before using up the budget, a negative C will be added to the reward.

Finally, in DRL, the agent should learn to maximize the expectation of the cumulative discounted reward, defined as $\sum_{t=1}^{T} \gamma^t r_t$ [12], where $\gamma \in (0, 1]$ is a factor discounting future rewards. During DRL training, this objective will guide the agent to find good approximate solutions to (3).

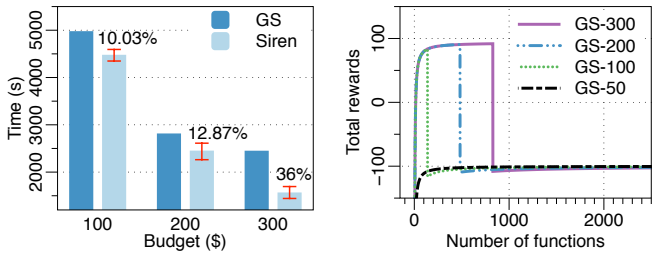
C. Training the DRL Agent

We briefly describe the techniques we use to train the DRL agent. We refer readers to [12] for a detailed survey and rigorous derivations for DRL training algorithms. The DRL training process is based on the iterative interaction between the agent and the environment, as shown in Fig. 4. The goal of DRL training is to maximize the *expected cumulative discounted reward*.

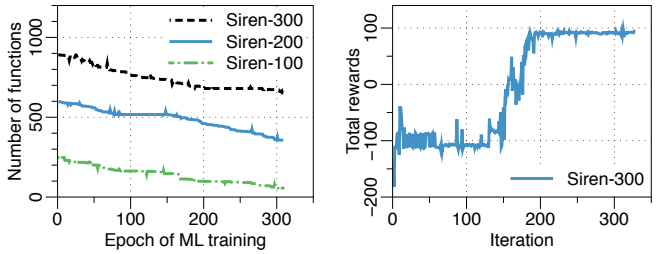
There are two phases to train a DRL agent: forward propagation and backward propagation. In the forward propagation, the agent observes some state $\mathbf{s}_t$ from the environment at epoch t. A policy DNN takes the state $\mathbf{s}_{t-1}$ as the input and outputs an action $\mathbf{a}_t$ for epoch t. After the action $\mathbf{a}_t$ is taken in the environment, a reward r_t is generated and observed, and the interaction moves to the next epoch $t+1$. The backward propagation phase starts after the forward propagation is processed for all T epochs, *i.e.*, when the ML training job has stopped. This phase focuses on maximizing the expected cumulative discounted reward through a gradient descent algorithm applied to the policy parameters θ. The gradient of this objective with respect to θ is given by

$$\nabla_\theta \mathbb{E}_\pi \left[\sum_{t=1}^{T} \gamma^t r_t \right] = \mathbb{E}_\pi [\nabla_\theta \ln \pi(\mathbf{a}|\mathbf{s}, \theta) q_\pi(\mathbf{s}, \mathbf{a})], \quad (5)$$

where $q_\pi(\mathbf{s}, \mathbf{a})$ is the expected cumulative discounted reward from choosing action $\mathbf{a}$ in state $\mathbf{s}$ and subsequently following policy π [12]. Then the policy parameters θ can be updated using gradient descent with backward propagation.



(a) Comparison on ML training time. (b) Rewards by the grid search (GS).



(c) Resource schemes by SIREN. (d) The learning curve of SIREN.

Fig. 5: A comparison between SIREN and the grid search for the best number of functions.

V. EVALUATION

We conduct extensive performance evaluation of SIREN through both simulation and real experiments done on the cloud platforms including AWS Lambda and EC2. We show that in our experiments, SIREN deployed on AWS Lambda can reduce job completion time by up to 44.3% as compared to running it in a parameter server architecture (MXNet) on EC2 clusters, at the same cost for the same quality of models. Before presenting the evaluation results, we briefly describe our methodology and settings as follows.

Simulation: We simulated a serverless cloud environment running mini-batched SGD algorithms controlled by the DRL agent. We implemented the simulation environment with the OpenAI Gym[3], which is an open source interface for evaluating reinforcement learning algorithms. The purpose of simulation is to verify the advantage by using SIREN for scheduling as compared to the optimal (static) policy found through a thorough grid search.

Testbed: We compare the completion time and cost of training ML jobs with SIREN on AWS Lambda and with MXNet on EC2 clusters. We select three types of EC2 instances to build the testbed clusters: `m4.large` (2 vCPU, 8GB memory), `m4.xlarge` (4 vCPU, 16GB memory) and `m4.2xlarge` (8 vCPU, 32GB memory), which charges `$0.1`, `$0.2` and `$0.4` per hour, respectively.

Workload: Our experimental evaluation is done for several popular machine learning models, including: 1) LeNet [14] on the MNIST dataset[4]; 2) sentiment-analysis of a movie review dataset[5] via convolutional neural network (CNN); and 3) linear classification using sparse matrix multiplication on the Avazu click-through prediction dataset[6].

[3]https://gym.openai.com

[4]http://yann.lecun.com/exdb/mnist/

[5]https://www.cs.cornell.edu/people/pabo/movie-review-data/

	Function #	Cost ($)	Time (s)
Grid Search	828	299.89	2452.3
SIREN	652 – 892	299.92	1569.5
Grid Search	482	199.67	2816.9
SIREN	355 – 597	199.73	2454.4
Grid Search	138	99.99	4979.7
SIREN	56 – 258	99.82	4480.4
Grid Search	3000	47.76	Fail
SIREN	1293 – 2995	49.82	Fail

TABLE III: A comparison between SIREN and the grid search for the best number of functions under different budgets.

Performance metrics: We measure the training completion time and calculate the total cost when the training job completes (converges). It should be noted that on EC2, it takes around 10 minutes to prepare all bare Linux instances to use MXNet for ML jobs, although we automate the deployment of EC2 clusters with `ansible`. However, we will exclude such instance preparation time for EC2 in our time and cost evaluation, though users still have to pay for it in reality. We also ignore the monthly AWS Lambda free quotas.

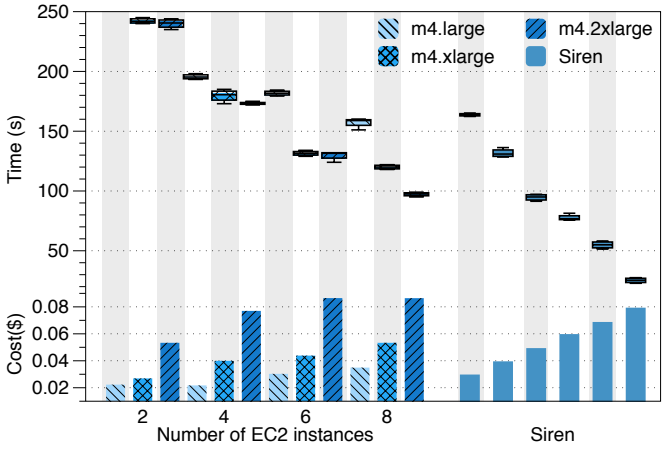
A. Simulation

We simulate a serverless cloud running a mini-batched SGD algorithm with SIREN. For each training epoch, we input the number of functions to the simulated serverless cloud, and in return we receive a reward as well as the states, including loss value ℓ_t and time metrics P_t, P_t^F, P_t^C and P_t^U. We obtain ℓ_t and P_t^U by simulating stateless functions with Python threads on a local server. Each thread calculates a gradient and updates the model in a local `redis` key-value database as Eqn. 2. P_t^F and P_t^U are simulated by adding a time delay to the I/O operations of threads as the network transmission time.

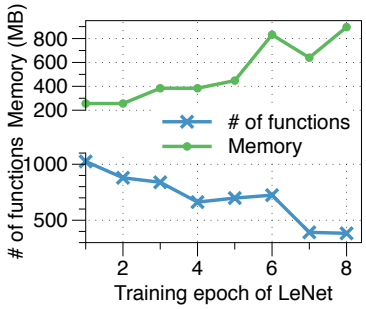
Upon the simulated serverless cloud, we compare SIREN with a baseline that performs a grid search over all possible numbers of functions. It searches for a particular number of functions that minimizes the training time. The baseline is cheating since it is infeasible to perform such an exhaustive search in reality.

Fig. 5(a) compares the training time achieved by the grid search and SIREN. Compare to the grid search, SIREN reduces the training time by at most 36%, given a budget of `$300`. The grid search enumerates the total rewards by different numbers of functions under different budgets as in Fig. 5(b). For example, under a budget of `$300`, training with 828 functions takes the least time: 2,452.3 seconds (Table III). SIREN dynamically launches 652 to 892 functions and finishes the training within 1,569.5 seconds. Both grid search and SIREN fail to finish the training with a budget of `$50`. As there is no way to finish the training job with `$50`, SIREN

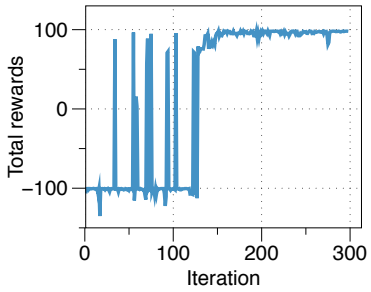
[6]https://www.kaggle.com/c/avazu-ctr-prediction/data



(a) Completion time and cost of EC2 and SIREN under different settings.



(b) Resource schemes by SIREN.



(c) The learning curve of SIREN.

Fig. 6: Training LeNet on the MNIST dataset by SIREN and by MXNet on EC2.

aggressively launches many more functions than the functions launched when given a budget of $200 and $300.

SIREN dynamically adjusts the number of functions based on its experience. Fig. 5(c) presents the number of functions assigned to each epoch. In the first several epochs, SIREN launches a large number of functions to rapidly reduce the loss value; in the later epochs, the agent decreases the number of functions to save money. The DRL agent of SIREN is trained online by iteratively interacting with the simulated serverless cloud. The learning curve in Fig. 5(d) shows that the agent learns to maximize the total reward by exploring different numbers of functions. The training of the agent is completed after around 200 iterations.

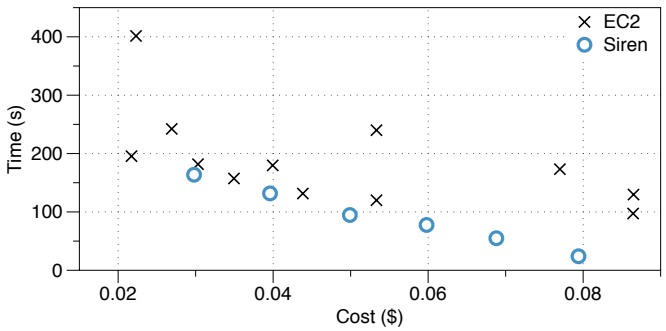


Fig. 7: Training completion time and cost of EC2 clusters and SIREN.

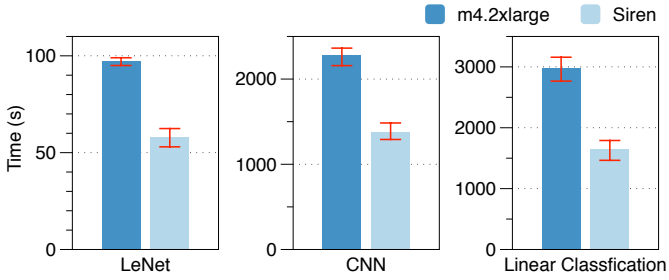


Fig. 8: A comparison between SIREN and EC2 under the same cost budget for different models.

B. Testbed and Experimental Results

It is infeasible to find the optimal cluster to compare with SIREN, due to the complexity of EC2 configurations. To broaden the coverage of potential optimal cluster settings, we setup twelve EC2 clusters of the three types of EC2 instances with four scales: 2, 4, 6 and 8 instances. For example, the `m4.large`×2 cluster is a cluster of two `m4.2xlarge` instances.

Fig. 6(a) plots the completion time and the corresponding cost of training LeNet with the twelve EC2 clusters and with SIREN. Fig. 6(a) reveals that the expense on EC2 is non-linear to the training completion time due to the heterogeneity of EC2 clusters. For example, both the `m4.xlarge`×6 cluster and the `m4.2xlarge`×6 cluster finish the training almost at the same time, but the latter incurred a cost twice as high as the former. In contrast, SIREN achieves shorter completion times with more investment. As in Fig.7, SIREN is faster than EC2 clusters at the same cost. Compared to the EC2 clusters, SIREN reduces training times by 16.2%, 26.7% and 28% at a cost of around $0.03, $0.04 and $0.05.

Fig. 6(b) presents that SIREN dynamically adjusts functions and their memory for each training epoch. When the number of functions decreases, each function receives larger training data partitions and needs larger memory to process the data partitions. The DRL agent in SIREN is trained by online interactions with AWS Lambda. The learning curve in Fig. 6(c) shows that the training of the DRL agent is completed after around 150 iterations.

We further train LeNet, the CNN model and the linear classification model on a cluster of eight `m4.2xlarge` instances,

and we collect the cost. We then train the same models with SIREN at the same cost of the `m4.2xlarge`×8 cluster. Fig. 8 shows that SIREN reduces the training time by 40%, 39.4% and 44.3% with these models respectively, compared to the EC2 cluster at the same cost.

VI. Related Work

To train ML models with massive datasets, a few ML training frameworks have been developed, such as TensorFlow and MXNet. These frameworks are mostly developed based on the architectures of parameter servers [15] or the Message Passing Interface (MPI), which heavily rely upon a dedicated cluster of physical servers or virtualized instances. However, it is non-trivial to choose the optimal configuration for a cluster to run distributed ML training jobs. Recently, a number of studies attempted to explore an optimal cluster configuration for distributed computing workloads (*e.g.*, [1]).

Meanwhile, serverless computing has emerged as a new cloud computing paradigm, where various applications have been built, including parallel data processing [2] and low-latency video processing [16]. [11] showed that running applications on the serverless cloud is more cost efficient than monolithic services such as AWS EC2. PyWren [2] performed ML training jobs as MapReduce tasks on the serverless cloud, where training is performed synchronously and developers have to refactor their existing models with PyWren's MapReduce-like APIs.

In contrast, as a new asynchronous distributed machine learning framework based on the serverless cloud, SIREN applies reinforcement learning techniques to achieve a balanced tradeoff between model quality and cost in order to minimize possible training times. With full support of MXNet APIs, SIREN is compatible with existing MXNet models.

SLAQ [4] is a quality-driven scheduler designed for multiple ML training jobs. It aims at maximizing the quality of models by adaptively allocating resources to different ML training jobs. In contrast, with SIREN, we focused on speeding up a single ML training job by making sequential resources provisioning decisions, and designed an experience-driven DRL scheduler that learns the best way to provision resources with deep reinforcement learning techniques. DRL has been used for a wide variety of learning tasks ranging from robotics to game playing and device placement [13]. With SIREN, we designed a DRL agent that schedules the number of functions and their memory for ML training jobs on the serverless cloud.

VII. Concluding Remarks

In this paper, we presented our design and implementation of SIREN, an asynchronous distributed machine learning framework based on the emerging serverless architecture. We argue that it is time-consuming and error-prone for ML practitioners and data scientists to maintain a physical or virtualized computing platform. SIREN eliminates the complexity of building and managing virtual machine infrastructures. We designed a DRL-based scheduler that learns the best way to achieve a balanced tradeoff between model quality and cost. We also proposed a new HSP computing mode that is efficient in serverless architectures. We have implemented a prototype of SIREN based on AWS Lambda and evaluated it with a variety of ML models. An extensive comparison with ML training jobs on AWS EC2 clusters has shown that ML training with SIREN on AWS Lambda has reduced the job completion time by up to 44.3% for the same quality of models at the same cost.

VIII. Acknowledgments

This work is supported by a research contract with Huawei Corp. and an NSERC Collaborative Research and Development (CRD) grant.

References

[1] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics." in *Proc. the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proc. 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[3] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," *Elastic*, vol. 60, p. 80, 2016.

[4] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-Driven Scheduling for Distributed Machine Learning," in *Proc. the 2017 Symposium on Cloud Computing (SoCC)*. ACM, 2017.

[5] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[6] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel Coordinate Descent for L1-Regularized Loss Minimization," in *Proc. Int'l Conference on Machine Learning (ICML)*, 2011.

[7] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[8] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[9] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via A Stale Synchronous Parallel Parameter Server," in *Advances in neural information processing systems (NIPS)*, 2013.

[10] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization," in *Advances in Neural Information Processing Systems*, 2015, pp. 2737–2745.

[11] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano *et al.*, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *Proc. the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[13] A. Mirhoseini, H. Pham, Q. Le, M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean, "Device Placement Optimization with Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2017.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proc. the IEEE*, 1998.

[15] A. Smola and S. Narayanamurthy, "An Architecture for Parallel Topic Models," *Proc. the VLDB Endowment*, 2010.

[16] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *Proc. the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.